Numerically Robust Implementations of Fast Recursive Least Squares Adaptive Filters using Interval Arithmetic

Christopher Peter Callender, B.Sc. AMIEE.

A thesis submitted for the degree of Doctor of Philosophy to the Faculty of Science at the University of Edinburgh.

1991

University of Edinburgh

Abstract of Thesis

Name of Candidate Christopher Peter Callender
Address [redacted]
Degree Ph.D. *Date* March 25, 1991
Title of Thesis Numerically Robust Implementations of Fast Recursive Least Squares Adaptive Filters using Interval Arithmetic
Number of words in the main text of Thesis Approximately 28,000

Algorithms have been developed which perform least squares adaptive filtering with great computational efficiency. Unfortunately, the fast recursive least squares (RLS) algorithms all exhibit numerical instability due to finite precision computational errors, resulting in their failure to produce a useful solution after a short number of iterations.

In this thesis, a new solution to this instability problem is considered, making use of interval arithmetic. By modifying the algorithm so that upper and lower bounds are placed on all quantities calculated, it is possible to obtain a measure of confidence in the solution calculated by a fast RLS algorithm and if it is subject to a high degree of inaccuracy due to finite precision computational errors, then the algorithm may be rescued, using a reinitialisation procedure.

Simulation results show that the stabilised algorithms offer an accuracy of solution comparable with the standard recursive least squares algorithm. Both floating and fixed point implementations of the interval arithmetic method are simulated and long-term stability is demonstrated in both cases.

A hardware verification of the simulation results is also performed, using a digital signal processor(DSP). The results from this indicate that the stabilised fast RLS algorithms are suitable for a number of applications requiring high speed, real time adaptive filtering.

A design study for a very large scale integration (VLSI) technology coprocessor, which provides hardware support for interval multiplication, is also considered. This device would enable the hardware realisation of a fast RLS algorithm to operate at far greater speed than that obtained by performing interval multiplication using a DSP.

Finally, the results presented in this thesis are summarised and the achievements and limitations of the work are identified. Areas for further research are suggested.

Acknowledgements

There are many people who deserve thanks for the assistance and support which have made the work of this thesis possible. I would like to thank all of the members of the signal processing group, both past and present, for their helpful discussions and comments on my work. They have helped to clarify many of my ideas.

Special thanks must go to Professor Colin Cowan. In his role as my supervisor, he has contributed greatly to the project and his guidance has been very much appreciated. Similarly, I am most grateful to Dr. Bernie Mulgrew for initially being my second supervisor and for taking over when Professor Cowan left the department. Thanks must also go to Professor Peter Grant for his encouragement and advice during my time in the group.

Finally, I would like to make it clear how much I appreciate the support of my parents during the time spent on this work.

Contents

Abbreviations

ADC	Analogue to Digital Converter
ASAP	As Soon As Possible
DAC	Digital to Analogue Converter
dB	deciBel
DSP	Digital Signal Processor
EOC	End Of Conversion
EVR	Eigenvalue Ratio
FAEST	Fast *A Posteriori* Error Sequential Technique
FIR	Finite Impulse Response
FK	Fast Kalman
FTF	Fast Transversal Filters
HF	High Frequency
IIR	Infinite Impulse Response
ISI	Intersymbol Interference
LS	Least Squares
LMS	Least Mean Squares
MLSE	Maximum Likelihood Sequence Estimator
RLS	Recursive Least Squares
SNR	Signal to Noise Ratio
SOC	Start Of Conversion
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration
VSE	VHDL Support Environment

Principal Symbols

$\underline{a}(k)$	Forward prediction coefficients for fast RLS algorithms
$\underline{b}(k)$	Backward prediction coefficients for fast RLS algorithms
$\underline{c}(k)$	Kalman gain vector
$\underline{c}'(k)$	Extended ($N+1$ th order) Kalman gain vector
$\underline{\tilde{c}}(k)$	Alternative Kalman gain vector
$\underline{\tilde{c}}'(k)$	Alternative extended ($N+1$ th order) Kalman gain vector
$d(k)$	k th adaptive filter desired response input
$e^b(k)$	Backward a priori prediction error
$e^f(k)$	Forward a priori prediction error
$e(k)$	A priori filter error
$h_j(k)$	j-th coefficient of FIR filter
$\underline{H}(k)$	Vector $[h_0(k)h_1(k)\cdots h_{N-1}(k)]^T$
$J_0(k)$	Unwindowed least squares cost function
$J_1(k)$	Exponentially windowed least squares cost function
$J_2(k)$	Least squares cost function with initial condition
k	Time index for sampled data signals
N	Length of adaptive filter
$\mathbf{r}_{xx}(k)$	Autocorrelation matrix for least squares algorithms
$\underline{r}_{dx}(k)$	Crosscorrelation vector for least squares algorithms
$x(k)$	k th input to adaptive filter
$\underline{X}(k)$	Vector $[x(k)x(k-1)\cdots x(k-N+1)]^T$
$\underline{X}'(k)$	Vector $[x(k)x(k-1)\cdots x(k-N)]^T$
$\alpha^b(k)$	Minimum value of backwards prediction least squares cost function
$\alpha^f(k)$	Minimum value of forwards prediction least squares cost function
$\epsilon^b(k)$	Backward a posteriori prediction error
$\epsilon^f(k)$	Forward a posteriori prediction error
$\epsilon(k)$	A posteriori filter error
λ	Exponential data window parameter
ρ	Maximum tolerable width of interval filter coefficients
μ	Weighting for soft constrained reinitialisation

Mathematical Notation

$\underline{A}$	Vector A
$\underline{A}_D$	Vector A of dimension (order) D
$\mathbf{A}$	Matrix A
$\|\|\underline{A}\|\|$	Euclidian norm of $\underline{A}$
$\underline{A}^T$	Transpose of $\underline{A}$
$\mathbf{A}^T$	Transpose of $\mathbf{A}$
$\mathbf{A}^{-1}$	Matrix inverse of $\mathbf{A}$
$[a^l, a^u]$	Interval number which contains all real numbers between a^l and a^u
$\frac{\partial f(x)}{\partial x}$	Partial derivative of $f(x)$ with respect to x
$\nabla J(x, y, z \ldots)$	Gradient vector of function J $\left(= \left[\frac{\partial J}{\partial x} \frac{\partial J}{\partial y} \frac{\partial J}{\partial z} \cdots \right]^T \right)$

"The subject we have just treated might give rise to several elegant analytical investigations upon which, however, we will not dwell, that we may be too diverted from our object. For the same reason, we must reserve for another occasion the explanation of the devices by which the numerical calculations may be rendered more expeditious"

- Karl Friedrich Gauss on least squares estimation techniques in *Theoria Motus Corporum Coelestium* †, *1809*

---

† From English translation - Gauss, K.F. "Theory of Motion of Heavenly Bodies", Dover, New York, 1963.

1 Introduction

1.1. Adaptive Filters - Structures and Applications

Fast, recursive least squares algorithms[1-3] have been developed for performing transversal least squares adaptive filtering in a highly computationally efficient manner. Unfortunately, all of these algorithms are numerically unstable, due to the way that finite precision errors are propagated. The important contribution of this thesis is to present a new stabilisation procedure which uses interval arithmetic to perform an error analysis for the algorithm whilst it is operating. The significance of this is that it enables a guaranteed limit to be placed on the magnitude of numerical errors, preventing instability and divergence. A hardware demonstration of an adaptive filter using the new methods has been developed, showing that interval arithmetic may be used in a practical application of adaptive filtering.

An adaptive filter[4-7] is a programmable filter, which automatically attempts to adjust its variable parameters so as to optimise its performance in some way. Figure 1.1 shows the general configuration of an adaptive system. There are two important elements to the system. The filter structure modifies the input signal in some way defined by its parameters and generates an output signal. The adaptive algorithm is responsible for monitoring the performance of this filter structure and adjusting its parameters, so as to maximise system performance.

Interestingly, a great number of adaptive systems occur in nature and in living things. One example of a biological adaptive system is the iris of the eye[8], which may be thought of as a filter which controls the amount of light which enters the eye. The filter has one programmable parameter - the radius of the iris. An adaptive algorithm in the brain monitors the brightness of the images which it receives (which is a measure of the performance of the iris filter). If the brightness does not meet some desired target, then the radius of the iris is adjusted, so as to improve its performance. In so doing, the eye is capable of good image detection over a much wider range of light levels than would be possible with a fixed iris radius.

This example illustrates one of the key advantages of adaptive filters over their fixed filter counterparts. A fixed filter can only give optimum performance in a limited number of situations, whereas the adaptive filter, with its ability to self-adjust, offers potentially better performance in a wide range of different circumstances. In some applications, the optimum filter may not be known *a priori*, as the conditions which affect the input signals may not be known exactly. Moreover, in many applications, the optimum solution varies with time, perhaps due to environmental factors and so a fixed filter cannot be applied. An adaptive filter, however, has the ability to *track* the changing optimum solution. In a large number of cases, the self-adjusting adaptive filter, therefore, has the potential for improved performance, as compared with a fixed filter.

A number of different structures and algorithms for adaptive systems have been proposed. The discussion in this thesis will be restricted to digital filters. These may be subdivided into linear and non-linear structures. Linear digital filters may be further subdivided into finite and infinite impulse response structures. For the finite impulse response filter the transfer function is realised by zeros only, as all of the poles of the filter are located at the origin. In the case of the infinite impulse response filter, however, both poles and zeros are used to realise the transfer function. One example of a finite impulse response filter is the linear transversal

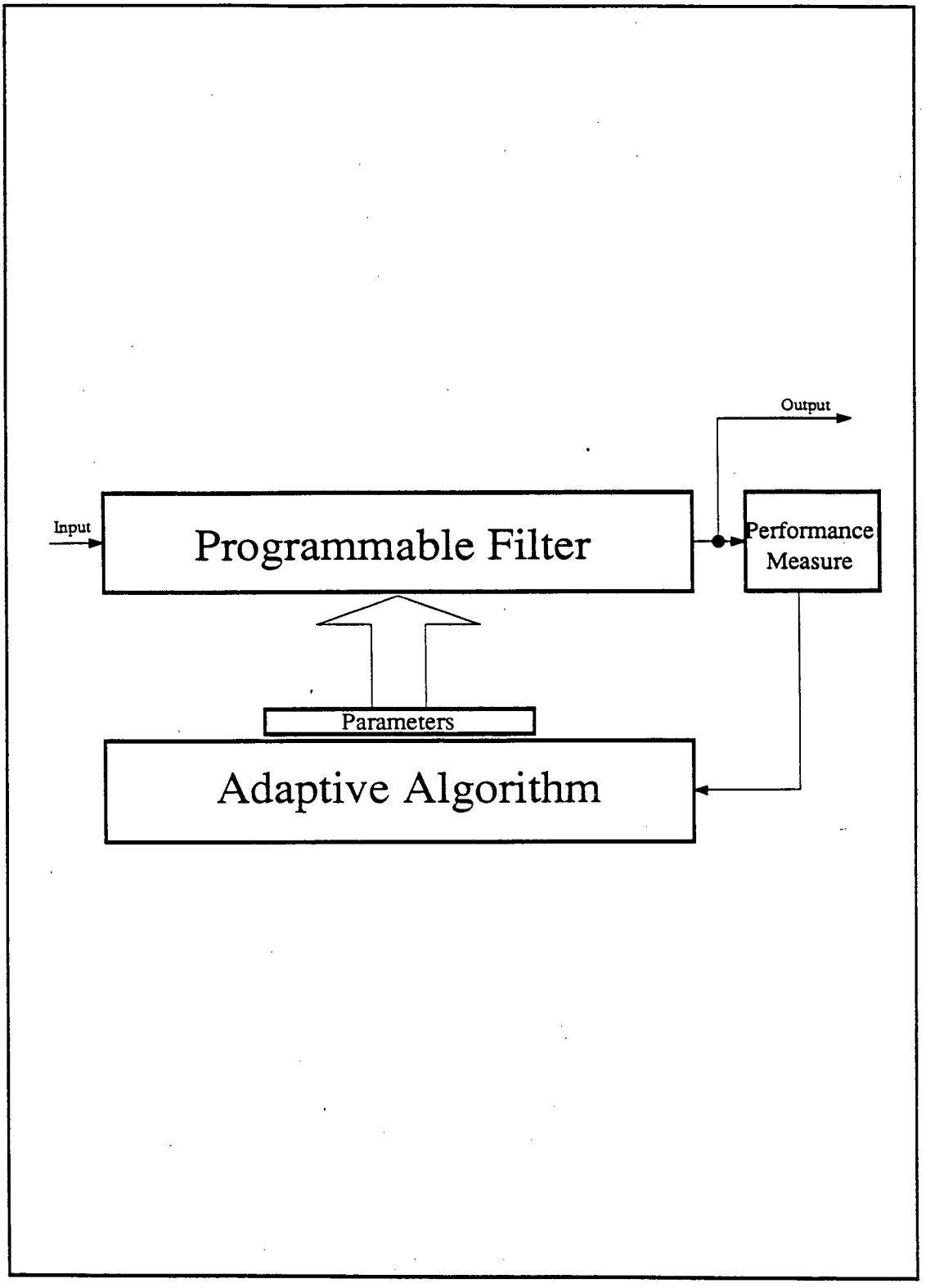


Figure 1.1 An adaptive filter. The device consists of two key parts : a programmable filter which is controlled by a number of parameters, and an adaptive algorithm which attempts to adjust these parameters so as to obtain optimum system performance.

filter,[9-13] shown in Figure 1.2a. It is also possible to generate lattice filters[14-16] which have a finite impulse response, such as the structure in Figure 1.2b. An example of a system which has a transfer function realised with both poles and zeros is the direct form infinite impulse response (IIR)[4, 6, 17-21] filter shown in Figure 1.3a. The difficulties associated with developing adaptive techniques for the IIR filter are considerable, because the filter is not unconditionally stable, as it has both poles and zeros in its transfer function. The danger is that the adaptive algorithm will choose a set of coefficients which place poles outside the unit circle in the z-plane and so provoke an unstable response. The filter error surface is also non-quadratic, which makes the task of developing an adaptive algorithm considerably more difficult.

Various non-linear digital filter structures have also been suggested for adaptive filtering applications including a range of artificial neural networks[22-27], which model the filter on a simplified brain-like structure. An example of a neural network is shown in Figure 1.3b. Whilst adaptive neural networks are currently an area of very active research, the theoretical aspects of non-linear structures are not nearly as well understood as the linear structures. The work of this thesis is, therefore, concerned with the linear transversal filter structure and the emphasis is on developing highly efficient algorithms for this well understood and often used structure.

1.2. Families of Adaptive Algorithms

A large number of algorithms for adaptive filters has been proposed. Indeed, adaptive filtering is an example of an optimisation problem and optimisation techniques form an important part of mathematics[28-31]. The additional constraint in adaptive filtering is that many of the applications require this optimisation to be performed in real time and so the complexity of the computations required must be kept to a minimum.

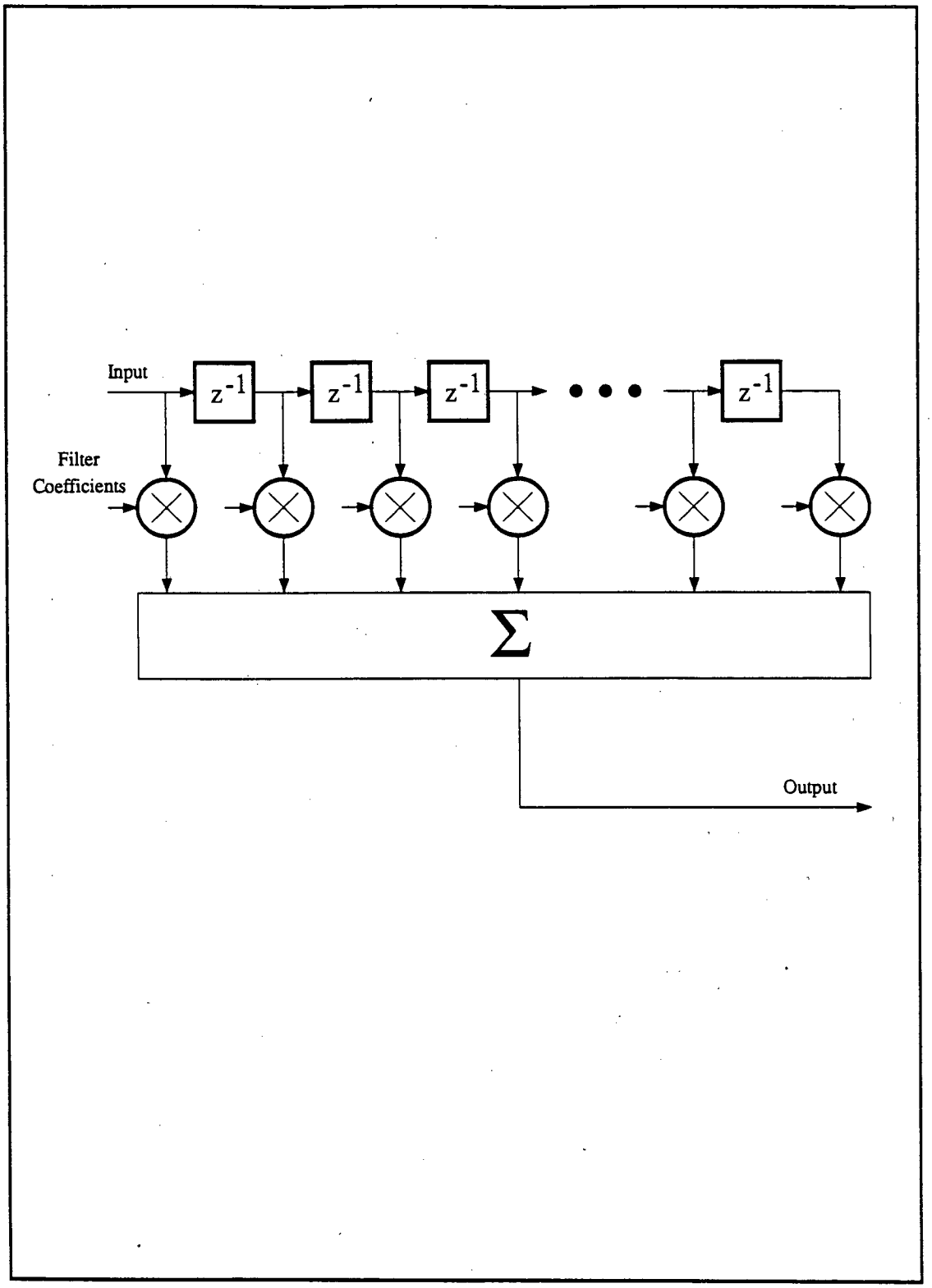


Figure 1.2a Structure of a linear transversal finite impulse response (FIR) filter.

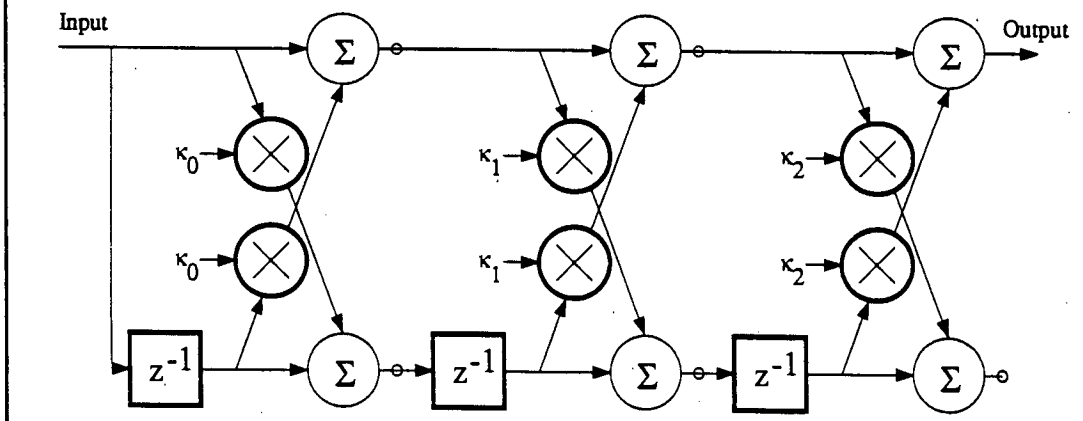


Figure 1.2b Structure of an all zero lattice filter.

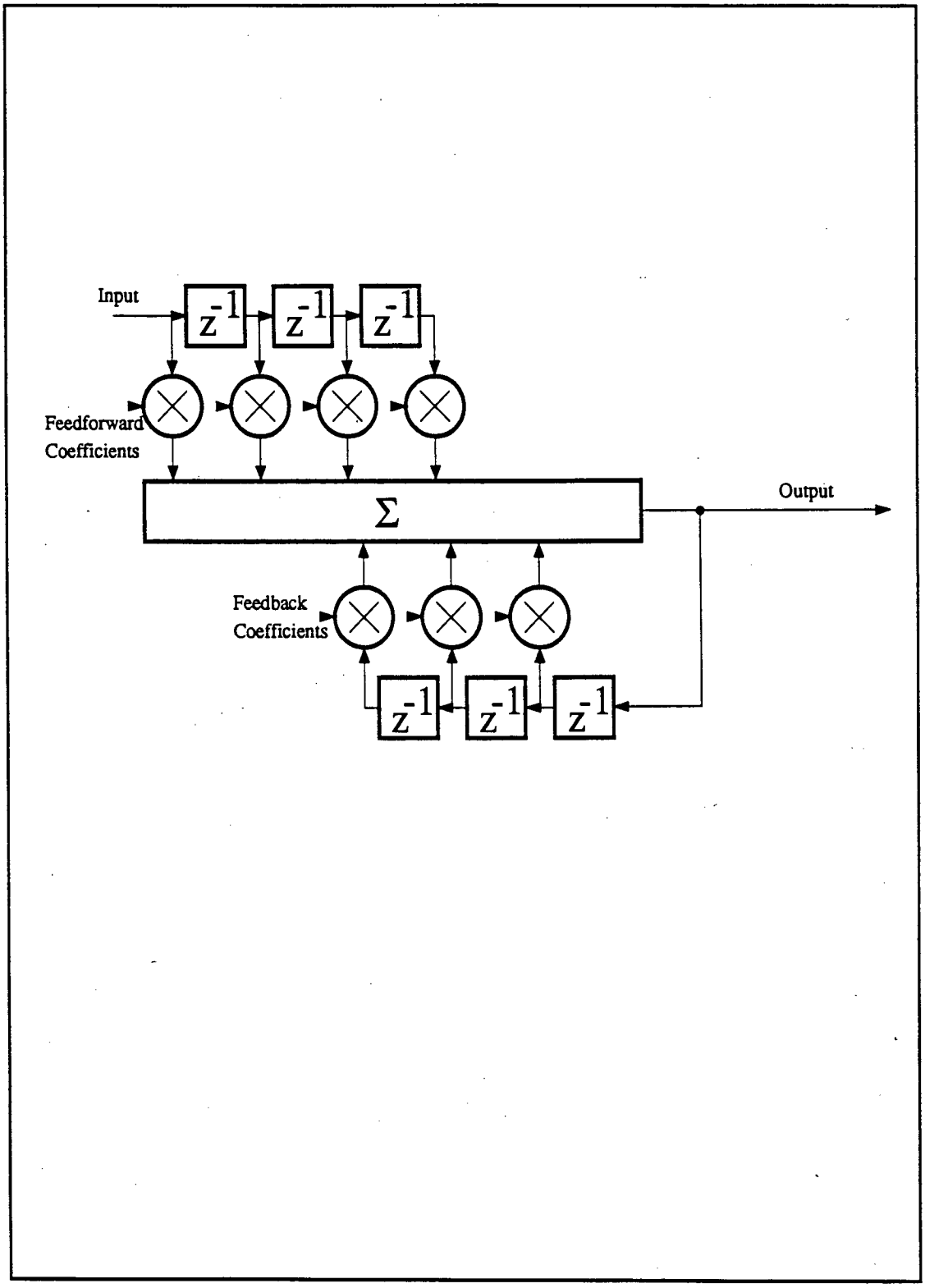
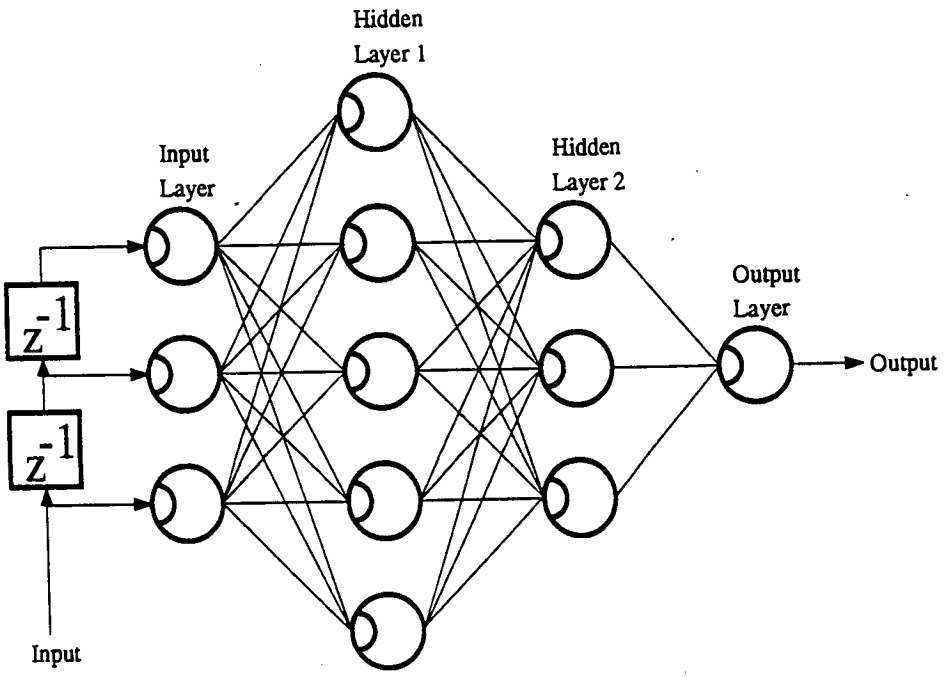
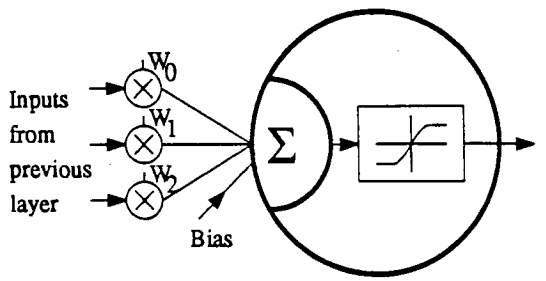


Figure 1.3a Structure of a recursive infinite impulse response (IIR) digital filter.



Conection of a number of processing elements
to form a neural network



A single neural processing element

Figure 1.3b Structure of the multi-layer perceptron which is one class of neural network.

Figure 1.4 shows some of the families of adaptive algorithms which have been suggested. Conceptually, one of the simplest techniques is the random search technique.[32] A random perturbation is made to the parameters of the programmable filter and the output is examined to see if this alteration improves the filter performance. If the performance is not improved, the perturbation is discarded and a new perturbation is tried. Random search techniques are interesting, as they have much in common with the mechanism of evolution suggested by the Darwinian theory of natural selection[33, 34], which may also be regarded as an example of an optimisation procedure, in which the performance measure being maximised is the probability of survival of life. It must be noted, however, that within the context of adaptive filtering, random search techniques are very slow to converge to a solution which is close to the optimum value and are therefore, of little practical value. This is due to their reliance on random perturbations to the filter parameters. There is a fairly low probability that any particular perturbation will change the filter parameters in the direction of their optimum values.

Before proceeding to discuss other adaptive algorithms, it is necessary to discuss the performance measure which is often used in adaptive filtering. It is normal to assume that a desired response signal is available and that the target of the adaptation algorithm is to minimise in some way the filter error, which is the difference between the filter output and the desired response input. The introduction of a desired response or reference signal does not seriously limit the usefulness of the adaptive filter and many important applications in which a desired response signal may readily be made available to the adaptive filter are presented later in this chapter. It is helpful when considering adaptive algorithms to imagine the error surface which is generated by measuring the mean value of the square of filter error as the filter coefficients are varied. Figure 1.5 shows a typical error surface for a transversal filter with two coefficients denoted by h_0 and h_1. In general, for the

linear transversal structure, the surface will be quadratic, with a single global minimum. The goal of an adaptation algorithm is to set the filter coefficients so as to obtain an operating point at this minimum, where the filter gives optimum performance.

One method by which this may be achieved is the stochastic gradient technique, which has resulted in algorithms such as the least mean squares (LMS) algorithm[4-6, 35, 36]. These algorithms operate by estimating the gradient of the error surface at the current operating point and then moving the coefficients in the direction of steepest descent of the error surface. By performing this operation repeatedly, the algorithm seeks out the minimum of the error surface in a number of steps. If the error surface changes in shape and the position of the minimum moves, as would be the case in a non-stationary environment, then the adaptive algorithm can track the optimum solution. In the case of the LMS algorithm, a noisy estimate of the gradient of the error surface is made from a single sample of the input data vector and error signal. This is used to update the filter coefficients and it can be shown that this procedure is guaranteed to converge close to the optimum solution, provided that certain restrictions are placed on the step-size[37].

Least squares algorithms[6, 16, 38] rely on a somewhat different technique. Instead of attempting to minimise the mean square value of the filter error, these algorithms minimise a cost function, such that the goal is to minimise the total sum of all the filter errors squared from when the algorithm was started to the current time. The important difference is that this involves the minimisation of a completely deterministic expression, rather than the statistical quantity of the stochastic gradient methods. This minimisation may be performed in principle by differentiation. As the filter is linear and a squared error cost function is used, this differentiation yields a set of linear simultaneous equations for the filter parameters. Least squares algorithms for signal processing concentrate on numerically efficient ways of solving this set of equations. The conventional recursive least squares (RLS) algo-

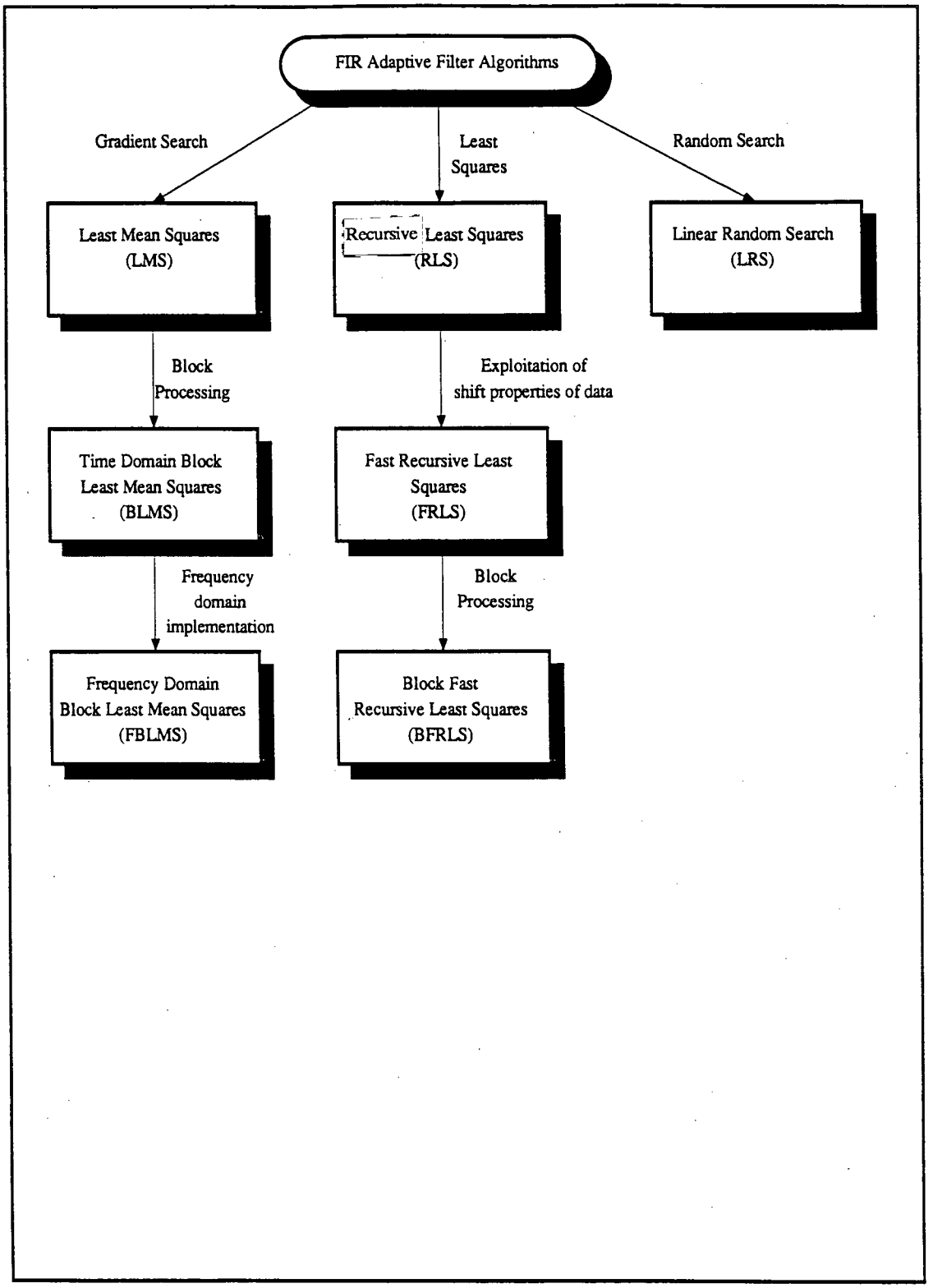


Figure 1.4 The main families of algorithms for performing adaptive filtering

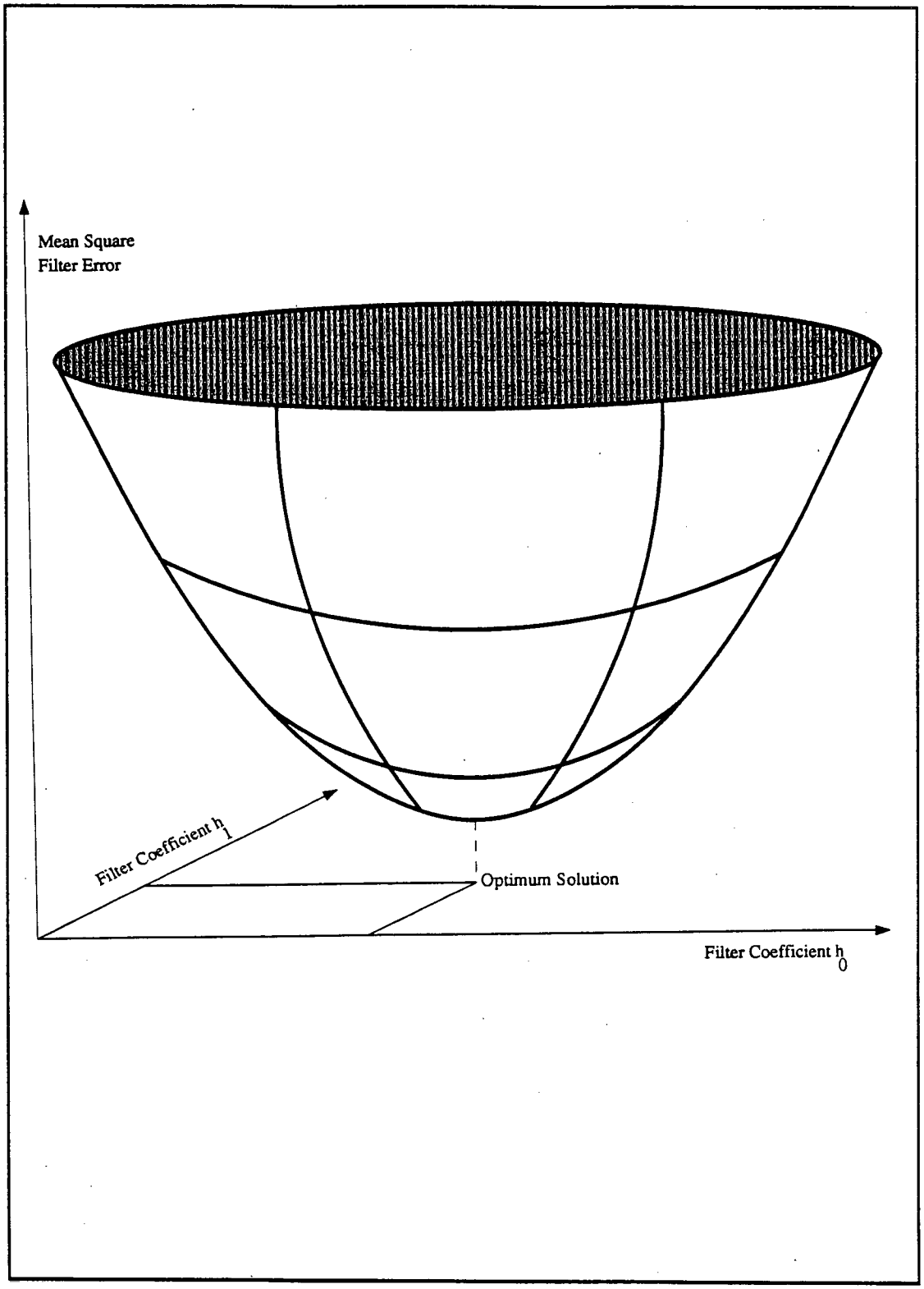


Figure 1.5 The quadratic mean square error surface of a linear transversal filter

rithm[5, 6, 16, 39] uses information about the previous solution to the system of equations, so as to reduce the computation in finding a new solution to the equations when a new squared error is added to the cost function. Various fast recursive least squares algorithms[1-3] have been developed which, in addition, exploit the shifting properties of the input data vector to provide a further saving in computational complexity.

Least squares techniques and stochastic techniques have a number of differences in the way that they perform[40]. In general, the time taken for a stochastic gradient algorithm to converge close to the optimum solution is much longer than for a least squares algorithm, due to the reliance of the stochastic algorithm on the statistics of the input data sequence and the need for an averaging process to occur with the 'noisy' gradient estimate. However, the computational complexity of these algorithms is very low and they are, therefore, suitable for high speed real time applications, where the speed of convergence is not critical. Least squares techniques have a much higher computational complexity, but their principal advantage over stochastic methods is their much more rapid initial convergence, which is independent of the statistics of the input signal. They have a higher computational complexity than the stochastic gradient methods, but in the case of the fast RLS algorithms, this complexity is of a comparable order of magnitude to the LMS algorithm.

Also of importance in considering the performance of an adaptive system is its ability to track the optimum solution in applications where the optimum solution varies with time. The comparison of the tracking performances of the two classes of algorithms is an area of current research[41-43]. Results show that the more rapid initial convergence of the least squares techniques does not necessarily imply better tracking performance in a non-stationary environment and that gradient techniques may offer comparable or even better performance.

Nevertheless, the rapid, data independent convergence of the least squares methods

makes them attractive for many applications. For example, in data communications, a known training sequence has to be transmitted until the algorithm has converged, reducing the throughput of useful data transmitted. The more rapidly the adaptation algorithm converges, the more useful data can be transmitted in a given time. The fast RLS algorithms are particularly attractive in this respect, as they are also suited to higher data rates than the conventional RLS algorithm due to their lower computational complexity. However, it is well known that these algorithms suffer from severe numerical instability[44]. Small numerical errors at each iteration of the algorithm accumulate, until they eventually cause divergence of the algorithm, resulting in a completely incorrect solution to the optimisation problem. The work in this thesis is concerned with finding solutions to the divergence phenomenon and making these potentially very efficient algorithms sufficiently robust to be of practical value.

1.3. Applications of Adaptive Filters

The versatility of a self-adjusting filter structure is such that the number of applications for adaptive techniques is very great. Adaptive filtering has found application in areas such as digital communications[45-49], telecommunications[50], noise cancellation[35, 51], speech coding[52, 53] and control systems[54-58]. Much of the work in this thesis will concentrate on the digital communications application, as this is probably the most widespread of all of the applications mentioned, but the new techniques developed could be applied to other adaptive filtering applications.

Four of the main configurations in which adaptive filters are often used are shown in Figures 1.6a-d. Each of these configurations will now be considered.

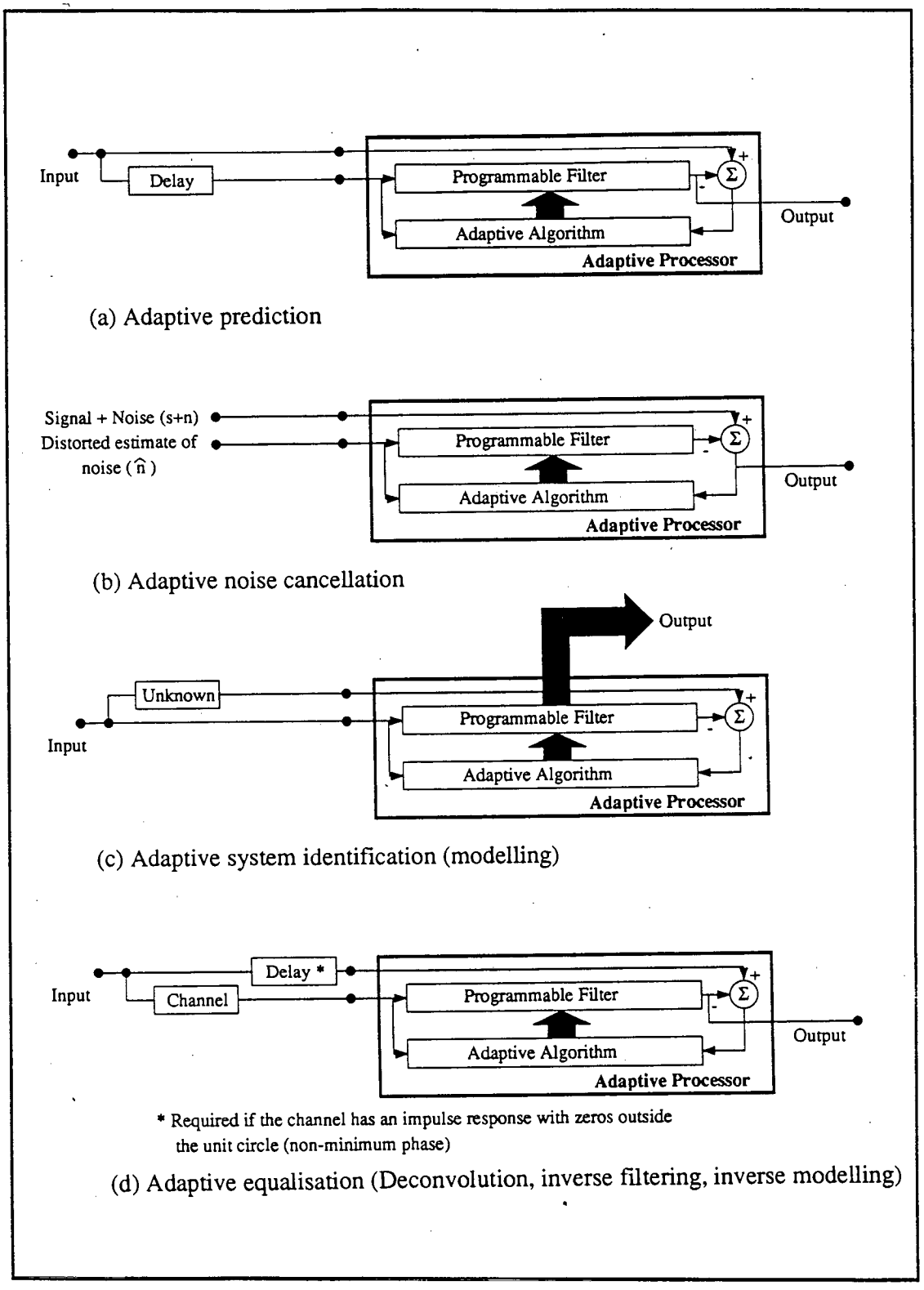


Figure 1.6 Important configurations for adaptive filtering

1.3.1. Prediction

Figure 1.6a shows an adaptive system configured to perform prediction of a signal, based upon its previous values. The signal is fed through a delay stage into the adaptive filter input and it is also input directly to the desired response input. The goal of the adaptive processor is, as always, to minimise the filter error signal. To do this, the filter output must approximate the current value of the signal. The only samples which are available to the filter, however, due to the delay stage, are previous values of the signal. The current value of the signal has not yet appeared as an input. It must, therefore, predict the current value based upon previous values of the signal.

Applications of adaptive predictors include cancellation of periodic interference from a non-periodic signal[51]. This is possible, since the predictor can predict only the periodic component of the signal, the non-periodic component usually being unpredictable. Another application is the efficient encoding of speech signals[52, 53], which are highly predictable over short time intervals.

1.3.2. Noise Cancellation

In Figure 1.6b, an adaptive processing system is configured to cancel interference. A signal, s, has been corrupted by some additive noise, n, to give a signal $s+n$. A correlated, but distorted, estimate of this noise, $\hat{n}$ is also available. Obviously, if this estimate was not distorted, it could simply be subtracted from the corrupted signal $s+n$, so as to recover the signal s. In this case, however, the noisy signal $s+n$ is fed into the desired response input of the adaptive processor and the estimate of the noise, $\hat{n}$ is fed into the filter input. To minimise the filter error in this

configuration corresponds to filtering the estimate of the noise, $\hat{n}$, so as to make it as close as possible to the actual noise, n. This filter output is then subtracted from $s+n$, in order to form a signal which closely resembles s.

Applications are widespread and include cancelling mains hum interference from medical signals[35,60], cancelling donor-heart interference when examining electrocardiograms during heart transplant operations [35], and cancelling additive noise from speech signals [35,61].

1.3.3. System Identification

This configuration is shown in Figure 1.6c. The aim is to find a system with transfer function, $\hat{H}(z)$ which closely approximates to the transfer function, $H(z)$, of the unknown system. A signal, s is fed into the adaptive processor and also into the unknown system. The output which the unknown system gives in response to this input is the desired response of the adaptive system and so it is fed into the desired response input. Therefore, the adaptive system learns to respond like the unknown system and when it has done this, parameters may be extracted from it, which also pertain to the unknown system. The output of the unknown system may be corrupted by a small amount of 'plant' noise, so that it cannot be identified exactly.

One important application of the adaptive system identifier is in digital communications. A maximum likelihood sequence estimator (MLSE)[62] may be used to give very good performance when attempting to recover a sequence at the receiver which has been corrupted by intersymbol interference. The maximum likelihood sequence estimator requires an estimate of the current impulse response of the transmission channel, however, so as to calculate which is the most probable transmitted sequence. Adaptive system identification provides a method for finding the impulse response of the channel for the sequence estimator.

Another important application in which this configuration is used is adaptive echo cancellation[50,59] for telecommunication.

1.3.4. Inverse Modelling

In the configuration of Figure 1.6d, a signal has been distorted by an unknown system, such as a communications channel, a transducer or some other system. The adaptive processor attempts to remove this distortion by performing inverse filtering on the output from the unknown system. This application is similar to system identification, except that the unknown system is in the filter input path, rather than in the desired response input path, so that the algorithm converges to find the inverse to the unknown system.

Applications of adaptive processors being used in this configuration include channel equalisation for digital radio communications[45, 46, 63, 64] allowing faster data rates with an acceptably low probability of error.

In this application, the desired response signal is generated locally at the receiver initially by using a known training sequence. After convergence of the adaptive algorithm, it is possible to switch to decision directed mode in which the desired response signal is generated by a threshold device, which makes a decision upon the output from the equaliser, allowing the filter error to be calculated and adaptive updating of the equaliser to take place. In practice, a delay may have to be introduced into the desired response path as shown in Figure 1.6d, so as to ensure that the channel and delay combination is minimum phase and suitable for equalisation by a linear structure.

1.4. Organisation of Thesis

As was previously mentioned, the primary aim of the work in this thesis is to study ways in which the highly computationally efficient fast RLS adaptive algorithms may be applied to practical applications, without the numerical instability problem

making itself apparent. The goal of this research is to find a method by which the algorithms may be stabilised and then to demonstrate that this stabilisation procedure results in algorithms which are of practical value in a number of applications.

Chapter 2 will begin by presenting much of the background to this work. The concept of least squares estimation as applied to the linear transversal filter will be developed and a number of algorithms which solve the least squares estimation problem will be derived. The first algorithm to be presented will be the conventional recursive least squares (RLS) algorithm, which has a computational complexity proportional to the square of the filter length and is therefore, too numerically intensive for many applications. The chapter will then proceed to discuss the fast RLS algorithms. A derivation of the fast Kalman algorithm, historically the first of the fast RLS algorithms to be discovered, will be given and then, two other algorithms for fast RLS transversal filtering will be examined. The reasons for the instability problems of the fast RLS algorithms will be looked at in some detail and various solutions, which have already been proposed to solve these problems, will be discussed. The benefits and limitations of the existing stabilisation schemes will be considered.

The theoretical aspects of a new solution to the numerical divergence problems are introduced in chapter 3. A scheme of arithmetic known as interval arithmetic is used. Effectively, this enables an error analysis to be performed in parallel with the computations of the algorithm, taking into account the effects of finite precision numerical errors. If the analysis indicates that the results calculated by the algorithm are being adversely affected by numerical errors, then the algorithm is rescued using a 'soft-constraint' rescue procedure. A number of new design parameters are introduced into the new interval arithmetic fast RLS algorithms and chapter 3 is concluded by some results relating to the correct choice of these parameters.

Chapter 4 gives simulation results relating to the new interval arithmetic algorithms.

The central aim of these simulations is to explore many different configurations and possibilities. To this end, simulations are performed using both floating and fixed point arithmetic, direct and inverse system modelling is performed and the simulations are applied in both the stationary and non-stationary scenario.

Having demonstrated successfully the performance of the interval fast RLS algorithms in simulations, chapter 5 considers a hardware implementation of the new algorithms. A digital signal processor is used and the operation of the 16 bit fixed point interval arithmetic fast RLS algorithm is demonstrated in real time as an equaliser.

Chapter 6 contains a design and feasibility study for a very large scale integration (VLSI) technology coprocessor, which would enable interval arithmetic algorithms to work at greater speed on a digital signal processor. The coprocessor design was developed using an advanced software package, which can convert from a high level behavioural description of the algorithm to a low level structural description of the gates and components required to implement it.

Finally, chapter 7 forms the conclusions to this work. Both the successes and the limitations of the new interval methods are discussed and areas for further research are identified.

2 Least Squares Algorithms for Adaptive Filtering

2.1. Introduction

A least squares adaptive algorithm[6, 16, 38, 65] is one in which some cost function involving total squared error is minimised by appropriate choice of the parameters of a filter. The filter structure which will be focussed upon in this chapter will be the linear transversal filter[9-13], although least squares algorithms for lattice filters[14-16] will be mentioned.

The principle advantage of a least squares algorithm over the popular stochastic gradient methods[4-6, 35, 36] for adaptive filtering is the greatly improved initial convergence[40, 59]. For the stochastic gradient methods, the initial convergence time is strongly dependent upon the statistical properties of the input signal[37, 66] and in the case of an ill-conditioned input, these algorithms will be slow to converge. Least squares algorithms, however, have convergence properties which are independent of the data statistics[67-69] and these algorithms will converge close to the optimum solution within $2N$ iterations where N denotes the order or length of the adaptive filter.

One problem with the application of least squares techniques to high speed real time systems is the relatively high computational complexity of the algorithms. The conventional recursive least squares[5, 6, 16, 39] (RLS) algorithm has a computational

complexity which is proportional to the square of the filter length. This inhibits its application to systems which require a high filter order, N, as the computational burden becomes unacceptably large. One such application is that of adaptive echo cancellation, where filter lengths of ≥ 1000 taps are typically required. To implement such a filter using the RLS algorithm would need several million additions and multiplications per iteration and such an implementation would clearly not be feasible.

The high complexity of the RLS algorithm may be reduced by exploiting the shifting properties of the input sequence with time. This has resulted in several fast RLS algorithms such as the fast Kalman (FK) algorithm[1, 70-72], the fast *a posteriori* error sequential technique (FAEST)[2, 73-76] and the fast transversal filters (FTF) algorithm[3, 77, 78], all of which are characterised by a computational complexity which is directly proportional to the filter length, N.

Unfortunately all of the highly efficient fast RLS algorithms suffer from severe numerical instability[44] when implemented using either fixed or floating point digital arithmetic[79-81]. They are highly sensitive to small numerical errors at each iteration and will often diverge suddenly from the correct least squares solution. It is the solution to this problem which is the basis for the work in the remainder of this thesis.

2.2. The Least Squares Problem for Linear Transversal Adaptive Filtering

The linear transversal filter operates by convolving a filter input sequence, $x(k)$ with a set of filter coefficients $h_i(k)$, to produce an output $y(k)$, given by :-

$$y(k)=\underline{H}^T(k)\underline{X}(k) \qquad [2.1]$$

where

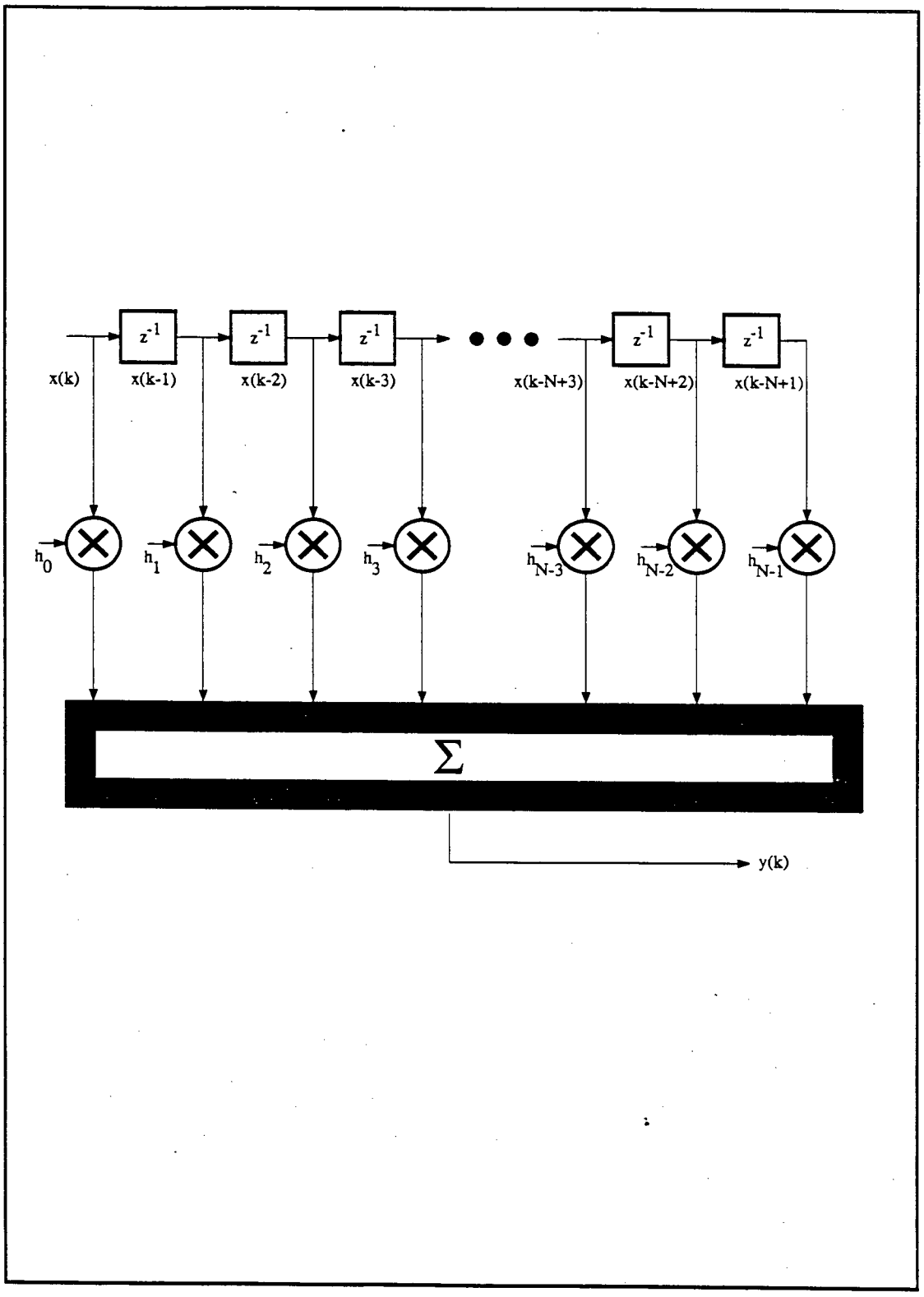


Figure 2.1 The linear transversal finite impulse response (FIR) filter

$$\underline{X}(k)=\begin{bmatrix} x(k) \\ \cdot \\ \cdot \\ x(k-N+1) \end{bmatrix}$$

and

$$\underline{H}(k)=\begin{bmatrix} h_0(k) \\ \cdot \\ \cdot \\ h_{N-1}(k) \end{bmatrix}$$

N is the length or order of the filter and the structure is shown in Figure 2.1.

In adaptive filtering, a desired response sequence $d(k)$ is introduced and the objective of the adaptive filtering algorithm is to find the set of coefficients, $\underline{H}(k)$, which produce an output $y(k)$ which is as close as possible to the desired response, $d(k)$.

We therefore define the error at time k by

$$\tilde{e}(k)=d(k)-y(k) \qquad [2.2]$$

In least squares filtering, the algorithm finds the coefficients $\underline{H}(k)$ which minimise a cost function $J_0(k)$, which is of the form

$$J_0(k)=\sum_{i=0}^{k}\tilde{e}^2(i) \qquad [2.3]$$

As as first stage to obtaining the solution to this minimisation problem, the partial derivatives of $J_0(k)$ with respect to each of the filter coefficients h_0, h_1, h_j ... h_{N-1} are evaluated

$$\frac{\partial J_0(k)}{\partial h_j}=\frac{\partial\left\{\sum_{i=0}^{k}\tilde{e}^2(i)\right\}}{\partial h_j}$$

$$=\sum_{i=0}^{k}\frac{\partial\tilde{e}^2(i)}{\partial h_j}$$

$$=\sum_{i=0}^{k}2\tilde{e}(i)\frac{\partial\tilde{e}(i)}{\partial h_j}$$

$$=\sum_{i=0}^{k}2\tilde{e}(i)\frac{\partial}{\partial h_j}\left\{d(i)-\underline{H}^T(k)\underline{X}(i)\right\}$$

$$= \sum_{i=0}^{k} 2\tilde{e}(i) \frac{\partial}{\partial h_j} \left\{ d(i) - \sum_{j=0}^{N-1} h_j(k) x(i-j) \right\}$$

$$= \sum_{i=0}^{k} -2\tilde{e}(i) x(i-j) \qquad [2.4]$$

It is now possible to evaluate the gradient vector, $\underline{\nabla} J_0(k)$ which will enable the optimum filter coefficients to be found.

$$\underline{\nabla} J_0(k) = \begin{bmatrix} \partial J_0(k)/\partial h_0 \\ \partial J_0(k)/\partial h_1 \\ \vdots \\ \partial J_0(k)/\partial h_{N-1} \end{bmatrix}$$

$$= -2 \sum_{i=0}^{k} \begin{bmatrix} x(i)\tilde{e}(i) \\ x(i-1)\tilde{e}(i) \\ \vdots \\ x(i-N)\tilde{e}(i) \end{bmatrix}$$

$$= -2 \sum_{i=0}^{k} \left\{ \begin{bmatrix} x(i) \\ x(i-1) \\ \vdots \\ x(i-N+1) \end{bmatrix} \tilde{e}(i) \right\}$$

$$= -2 \sum_{i=0}^{k} \left\{ \underline{X}(i) \tilde{e}(i) \right\}$$

$$= -2 \sum_{i=0}^{k} \left\{ \underline{X}(i) \left[d(i) - \underline{X}^T(i) \underline{H}(k) \right] \right\}$$

$$= -2 \sum_{i=0}^{k} \left\{ \underline{X}(i) d(i) - \underline{X}(i) \underline{X}^T(i) \underline{H}(k) \right\}$$

$$= -2 \sum_{i=0}^{k} \left\{ \underline{X}(i) d(i) \right\} + 2 \sum_{i=0}^{k} \left\{ \underline{X}(i) \underline{X}^T(i) \right\} \underline{H}(k) \qquad [2.5]$$

It is convenient to introduce the matrix

$$\mathbf{r}_{xx}(k)=\sum_{i=0}^{k}\underline{X}(i)\underline{X}^{T}(i) \qquad [2.6]$$

and the vector

$$\underline{r}_{dx}(k)=\sum_{i=0}^{k}d(i)\underline{X}(i) \qquad [2.7]$$

This enables [2.5] to be written as

$$\nabla J_0(k)=-2\underline{r}_{dx}(k)+2\mathbf{r}_{xx}(k)\underline{H}(k) \qquad [2.8]$$

and setting $\nabla J_0(k)=\underline{0}$ to obtain the optimum solution $\underline{H}^{opt}(k)$ gives

$$\mathbf{r}_{xx}(k)\underline{H}^{opt}(k)=\underline{r}_{dx}(k) \qquad [2.9]$$

and therefore

$$\underline{H}^{opt}(k)=\mathbf{r}_{xx}^{-1}(k)\underline{r}_{dx}(k) \qquad [2.10]$$

provided that $\mathbf{r}_{xx}(k)$ is non-singular.

In principle, this result could be used to implement an adaptive algorithm, as it enables the optimum coefficients to be calculated from the filter and desired response inputs. It should be noted, however, that equation [2.10] requires a matrix inversion to be performed on the $N\times N$ matrix $\mathbf{r}_{xx}(k)$. If this inversion is to be performed by a conventional matrix inversion method such as the Gauss - Jordan technique[82], then the number of operations per iteration of the algorithm will be of order N^3. This is likely to yield an unacceptable computational burden if N is even moderately large.

If certain assumptions are made, then the matrix $\mathbf{r}_{xx}(k)$ will become Toeplitz in structure and the Levinson - Durbin algorithm[83, 84] may be used to find the solution from equation [2.9]. To obtain this structure, both the pre-windowed assumption

$$x(i)=0,\quad i<0 \qquad [2.11]$$

and the post-windowed assumption

$$x(i)=0,\quad i>k-N+1 \qquad [2.12]$$

must be invoked. When the pre-windowed and post-windowed assumptions are used together in this way, this is known as the autocorrelation form. When no assump-

tions are made about values of the data, $x(i)$ outside the range $0 \leq i \leq k-N+1$, then this is known as the covariance form. If assumptions about the data windowing cannot be made, then the Levinson - Durbin algorithm cannot be used. Other efficient solutions to the problem have therefore been developed.

2.3. The Conventional Recursive Least Squares Algorithm

In developing this algorithm, the aim is to update the value of the matrix $\mathbf{r}_{xx}^{-1}(k-1)$ which is assumed to be available, so as to obtain $\mathbf{r}_{xx}^{-1}(k)$. In so doing, the need to perform matrix inversion at every iteration of the algorithm is eliminated and the computational complexity is reduced.

What is required is to update the values of $\underline{H}(k-1)$ and $\mathbf{r}_{xx}^{-1}(k-1)$ so as to include the new data which becomes available at time k. This is done by writing

$$\mathbf{r}_{xx}(k) = \mathbf{r}_{xx}(k-1) + \underline{X}(k)\underline{X}^T(k) \qquad [2.13]$$

and

$$\underline{r}_{dx}(k) = \underline{r}_{dx}(k-1) + d(k)\underline{X}(k) \qquad [2.14]$$

Substituting for $\underline{r}_{dx}(k)$ in [2.14] using [2.9] gives

$$\mathbf{r}_{xx}(k)\underline{H}(k) = \mathbf{r}_{xx}(k-1)\underline{H}(k-1) + d(k)\underline{X}(k) \qquad [2.15]$$

It is then possible to use [2.13] to substitute for $\mathbf{r}_{xx}(k-1)$, yielding

$$\mathbf{r}_{xx}(k)\underline{H}(k) = \left[\mathbf{r}_{xx}(k) - \underline{X}(k)\underline{X}^T(k)\right]\underline{H}(k-1) + d(k)\underline{X}(k) \qquad [2.16]$$

If we define

$$\underline{c}(k) = \mathbf{r}_{xx}^{-1}(k)\underline{X}(k) \qquad [2.17]$$

and the *a priori* filter error by

$$e(k) = d(k) - \underline{H}^T(k-1)\underline{X}(k) \qquad [2.18]$$

then [2.16] may be rearranged as follows

$$\begin{aligned}\underline{H}(k) &= \underline{H}(k-1) - \mathbf{r}_{xx}^{-1}(k)\underline{X}(k)\underline{X}^T(k)\underline{H}(k-1) + \mathbf{r}_{xx}^{-1}(k)d(k)\underline{X}(k) \\ &= \underline{H}(k-1) - \underline{c}(k)\underline{X}^T(k)\underline{H}(k-1) + \underline{c}(k)d(k) \\ &= \underline{H}(k-1) + \underline{c}(k)e(k)\end{aligned} \qquad [2.19]$$

To obtain the recursive update for $\mathbf{r}_{xx}^{-1}(k)$, it is necessary to make use of the Sherman Morrison matrix inversion lemma[85, 86]. For all **A**,**B**,**C** and **D** of compatible dimensions,

$$\left[\mathbf{A}+\mathbf{BCD}\right]^{-1}=\mathbf{A}^{-1}-\mathbf{A}^{-1}\mathbf{B}\left(\mathbf{C}+\mathbf{DA}^{-1}\mathbf{B}\right)^{-1}\mathbf{DA}^{-1} \qquad [2.20]$$

We note that

$$\mathbf{r}_{xx}^{-1}(k)=\left[\mathbf{r}_{xx}(k-1)+\underline{X}(k)\underline{X}^{T}(k)\right]^{-1} \qquad [2.21]$$

so using identity [2.20] with $\mathbf{A}=\mathbf{r}_{xx}^{-1}(k-1)$, $B=\underline{X}(k)$, $C=1$ and $D=\underline{X}^{T}(k)$, yields the update

$$\mathbf{r}_{xx}^{-1}(k)=\mathbf{r}_{xx}^{-1}(k-1)-\mathbf{r}_{xx}^{-1}(k-1)\underline{X}(k)\left(1+\underline{X}^{T}(k)\mathbf{r}_{xx}^{-1}(k-1)\underline{X}(k)\right)^{-1} \qquad [2.22]$$
$$\bullet\underline{X}^{T}(k)\mathbf{r}_{xx}^{-1}(k-1)$$

This result may now be used to substitute for $\mathbf{r}_{xx}^{-1}(k)$ in [2.17] to give

$$\begin{aligned}
c(k)&=\mathbf{r}_{xx}^{-1}(k)\underline{X}(k)\\
&=\mathbf{r}_{xx}^{-1}(k-1)-\mathbf{r}_{xx}^{-1}(k-1)\underline{X}(k)\left(1+\underline{X}^{T}(k)\mathbf{r}_{xx}^{-1}(k-1)\underline{X}(k)\right)^{-1}\\
&\qquad\bullet\underline{X}^{T}(k)\mathbf{r}_{xx}^{-1}(k-1)\underline{X}(k)\\
&=\mathbf{r}_{xx}^{-1}(k-1)\underline{X}(k)\\
&\quad\bullet\left\{1-\left(1+\underline{X}^{T}(k)\mathbf{r}_{xx}^{-1}(k-1)\underline{X}(k)\right)^{-1}\underline{X}^{T}(k)\mathbf{r}_{xx}^{-1}(k-1)\underline{X}(k)\right\}\\
&=\mathbf{r}_{xx}^{-1}(k-1)\underline{X}(k)\left\{1+\underline{X}^{T}(k)\mathbf{r}_{xx}^{-1}(k-1)\underline{X}(k)\right\}^{-1}
\end{aligned} \qquad [2.23]$$

This completes the derivation of the RLS algorithm, which consists of equations [2.18], [2.22], [2.23] and [2.19]. It is normal to take initial values as $\mathbf{r}_{xx}^{-1}(0)=\sigma\mathbf{I}$ and $\underline{H}(0)=\underline{0}$, where σ is a small positive number and **I** is the identity matrix.

2.4. Data Windows

The cost function $J_0(k)$ defined in [2.3] is inappropriate for use in a time variant environment, where the optimum solution $\underline{H}^{opt}(k)$ varies with time. As all errors are penalised equally, any algorithm which minimises $J_0(k)$ will have a growing

memory and, therefore, cannot track the time-varying solution as required.

To overcome this problem, it is common to introduce a 'forgetting factor', λ which is used to window the terms in the cost function exponentially, so as to give greater importance to more recent error terms in the sum of squared error cost function. The cost function is modified to become

$$J_1(k)=\sum_{i=0}^{k}\lambda^{k-i}e^2(i) \qquad [2.24]$$

where λ is slightly less than 1.

If this cost function is minimised with respect to $\underline{H}(k)$ by differentiation, then a solution of the same form as [2.10] is obtained, provided that the definition of $\mathbf{r}_{xx}$ is modified to be :-

$$\mathbf{r}_{xx}(k)=\sum_{i=0}^{k}\lambda^{k-i}\underline{X}(i)\underline{X}^T(i) \qquad [2.25]$$

and $\underline{r}_{dx}$ is defined as :-

$$\underline{r}_{dx}(k)=\sum_{i=0}^{k}\lambda^{k-i}d(i)\underline{X}(i) \qquad [2.26]$$

It is then possible to proceed in the same way as in section 2.3 to derive the exponentially windowed RLS adaptive algorithm. This algorithm is summarised in Table 2.1

A number of other windowing functions[77] have also been proposed, including the sliding rectangular window. Using this windowing method, errors occurring more than a certain time before the current sample are ignored completely. This may give some improvement in highly non-stationary operation, but the resulting algorithms are generally more computationally complicated than that which would be obtained using an exponential window.

- Initialisation

$\underline{H}(0)=\underline{0}$, $\mathbf{r}_{xx}^{-1}(0)=\frac{1}{\sigma}\mathbf{I}$, $\underline{X}(0)=\underline{0}$ where σ is a small positive number.

- At time k, do

$$e(k)=d(k)-\underline{X}^T(k)\underline{H}(k-1)$$

$$\mathbf{r}_{xx}^{-1}(k)=\frac{\mathbf{r}_{xx}^{-1}(k-1)-\mathbf{r}_{xx}^{-1}(k-1)\underline{X}(k)\left[\lambda+\underline{X}^T(k)\mathbf{r}_{xx}^{-1}(k-1)\underline{X}(k)\right]^{-1}\underline{X}^T(k)\mathbf{r}_{xx}^{-1}(k-1)}{\lambda}$$

$$\underline{c}(k)=\mathbf{r}_{xx}^{-1}(k-1)\underline{X}(k)\left[\lambda+\underline{X}^T(k)\mathbf{r}_{xx}^{-1}(k-1)\underline{X}(k)\right]^{-1}$$

$$\underline{H}(k)=\underline{H}(k-1)+\underline{c}(k)e(k)$$

Table 2.1: Conventional RLS algorithm with exponential windowing

2.5. Computational Complexity

One of the major limitations in the application of the RLS algorithm of Table 2.1 is its computational complexity. Making use of the symmetry of the matrix $\mathbf{r}_{xx}^{-1}(k)$, it is possible to implement the algorithm with $2.5N^2+4.5N$ additions and multiplications per iteration. As the complexity of the algorithm is dependent upon the square of the filter order, it will become unacceptably large for use with long adaptive filters.

It is the high computational complexity which has motivated the development of the fast RLS algorithms, which provide a means of calculating the same least squares solution as the conventional algorithm, but with a computational complexity which is directly proportional to the filter length. This saving in computation is obtained by exploiting the shifting properties with time of the data vector $\underline{X}(k)$, which results in the matrix $\mathbf{r}_{xx}(k)$ having a near to Toeplitz structure.

2.6. The Fast Kalman Algorithm

The fast Kalman algorithm was presented by Ljung, Morf and Falconer in 1978[1]. The derivation begins by developing some special cases of the least squares problem of section 2.2 for forward and backward prediction. An Nth order linear forward predictor may be defined by

$$\hat{x}^f(k) = \underline{a}^T(k)\underline{X}(k-1) \qquad [2.27]$$

where $\underline{a}^T(k) = [a_0 \ \cdots \ a_{N-1}]$ is a set of forward prediction coefficients.

That is to say that an estimate of the current value, $x(k)$ is to be made using a linear combination of N previous observations of a signal. Similarly, backwards prediction may be defined by

$$\hat{x}^b(k-N) = \underline{b}^T(k)\underline{X}(k) \qquad [2.28]$$

where $\underline{b}^T(k) = [b_0 \ \cdots \ b_{N-1}]$ is a set of backwards predictor coefficients.

The optimum predictor coefficients $\underline{a}(k)$ and $\underline{b}(k)$ may be chosen by least squares methods. For forwards prediction, the appropriate sum of squared errors cost function is

$$J^f(k) = \sum_{i=0}^{k} \left[x(i) - \underline{a}^T(k)\underline{X}(i-1)\right]^2 \qquad [2.29]$$

and for the backwards coefficients

$$J^b(k) = \sum_{i=0}^{k} \left[x(i-N) - \underline{b}^T(k)\underline{X}(i)\right]^2 \qquad [2.30]$$

These correspond to two special cases of the least squares problem which has already been solved in section 2.2. The forward prediction case corresponds to a desired response of $x(k)$ given an input vector $\underline{X}(k-1)$ for which the solution is

$$\underline{a}(k) = \mathbf{r}_{xx}^{-1}(k-1)\underline{r}_x^f(k)$$

where

$$\underline{r}_x^f = \sum_{i=0}^{k} x(i)\underline{X}(i-1) \qquad [2.31]$$

Similarly, backwards prediction corresponds to a least squares adaptive filtering with a desired response of $x(k-N)$, using the input data vector $\underline{X}(k)$, for which the solution may be written as

$$\underline{b}(k)=\mathbf{r}_{xx}^{-1}(k)\underline{r}_x^b(k)$$

where

$$\underline{r}_x^b=\sum_{i=0}^{k}x(i-N)\underline{X}(i) \qquad [2.32]$$

It is possible to use the recursive methods of section 2.3 to update the predictors $\underline{a}(k)$ and $\underline{b}(k)$. The following recursions are obtained

$$e^f(k)=x(k)-\underline{a}^T(k-1)\underline{X}(k-1) \qquad [2.33]$$

$$\underline{a}(k)=\underline{a}(k-1)+\underline{c}(k-1)e^f(k) \qquad [2.34]$$

for forwards prediction and

$$e^b(k)=x(k-N)-b^T(k-1)\underline{X}(k) \qquad [2.35]$$

$$\underline{b}(k)=\underline{b}(k-1)+\underline{c}(k)e^f(k) \qquad [2.36]$$

$e^f(k)$ and $e^b(k)$ are known as the *a priori* forward and backward prediction errors respectively.

Note that the gain vector, $\underline{c}(k)$, used to update $\underline{a}(k)$ in [2.34] and $\underline{b}(k)$ in [2.36] is the same gain vector as that used in the recursion for $\underline{H}(k)$ in [2.19]. That is to say that $\underline{c}(k)$ is

$$\underline{c}(k)=\mathbf{r}_{xx}^{-1}(k)\underline{X}(k) \qquad [2.37]$$

for all the problems considered.

The values of the cost functions $J^f(k)$ and $J^b(k)$ may be evaluated at their minima to give

$$\begin{aligned}\alpha^f(k)&=\min(J^f(k))\\&=\sum_{i=0}^{k}x^2(i)-\underline{a}^T(k)\underline{r}_x^f(k)\\&=r_0^f(k)-\underline{a}^T(k)\underline{r}_x^f(k)\end{aligned} \qquad [2.38]$$

where $r_0^f(k)=\sum_{i=0}^{k}x^2(i)$

and

$$\begin{aligned}\alpha^b(k) &= min\,(J^b(k)) \\ &= \sum_{i=0}^{k} x^2(i-N) - \underline{b}^T(k)\underline{r}_x^b(k) \\ &= r_0^b(k) - \underline{b}^T(k)\underline{r}_x^b(k)\end{aligned} \qquad [2.39]$$

where $r_0^b(k) = \sum_{i=0}^{k} x^2(i-N)$

This completes the preliminary results relating to forwards and backwards prediction. The method used to exploit these results in the main RLS algorithm is to consider a system in which the order has been increased from N to $N+1$. All quantities relating to this increased order system will be denoted by a $'$ symbol to discriminate them from their N th order counterparts.

We define the $N+1$th order data vector by

$$\underline{X}'(k) = \begin{bmatrix} x(k) \\ \vdots \\ x(k-N) \end{bmatrix} \qquad [2.40]$$

and we immediately note that the $N+1$th order system data vector can be related to the N th order data vector by

$$\underline{X}'(k) = \begin{bmatrix} x(k) \\ \hline \underline{X}(k-1) \end{bmatrix} \text{ and } \underline{X}'(k) = \begin{bmatrix} \underline{X}(k) \\ \hline x(k-n) \end{bmatrix} \qquad [2.41]$$

It is these relationships which enable the use of definitions from the forward and backward predictors developed earlier.

An equivalent to the $\mathbf{r}_{xx}$ matrix for the $N+1$th order system may be defined by

$$\mathbf{r}_{xx}'(k) = \sum_{i=0}^{k} \underline{X}'(i)\underline{X}'^T(i) \qquad [2.42]$$

and we may use the relationships of [2.41] to relate this to the N th order system by

$$\begin{aligned}\mathbf{r}_{xx}'(k) &= \sum_{i=0}^{k} \begin{bmatrix} x(i) \\ \hline \underline{X}(i-1) \end{bmatrix} \begin{bmatrix} x(i) \mid \underline{X}^T(i-1) \end{bmatrix} \\ &= \begin{bmatrix} \sum_{i=0}^{k} x(i)x(i) & \mid & \sum_{i=0}^{k} \underline{X}^T(i-1)x(i) \\ \hline \sum_{i=0}^{k} \underline{X}(i-1)x(i) & \mid & \mathbf{r}_{xx}(\mathbf{k-1}) \end{bmatrix}\end{aligned} \qquad [2.43]$$

and using the definitions from the work on forward predictors, this is

$$\mathbf{r}_{xx}'(k)=\left[\begin{array}{c|c} r_0^f(k) & \underline{r}_x^{f\,T}(k) \\ \hline \underline{r}_x^f(k) & \mathbf{r}_{xx}(k-1) \end{array}\right] \qquad [2.44]$$

In exactly the same way, using the second part of [2.41] and the backward predictor definitions,

$$\mathbf{r}_{xx}'(k)=\left[\begin{array}{c|c} \mathbf{r}_{xx}(\mathbf{k}) & \underline{r}_x^b(k) \\ \hline \underline{r}_x^{b\,T}(k) & r_0^b(k) \end{array}\right] \qquad [2.45]$$

The matrix $\mathbf{r}_{xx}'(k)$ may now be inverted, using the Sherman Morrison matrix identity[85-87] and the inversion rule for partitioned matrixes[88, 89], giving

$$\mathbf{r}_{xx}'(k)=\left[\begin{array}{c|c} \frac{1}{\alpha^f(k)} & \frac{-\underline{a}^T(k)}{\alpha^f(k)} \\ \hline \frac{-\underline{a}(k)}{\alpha^f(k)} & \mathbf{r}_{xx}^{-1}(k-1)+\frac{\underline{a}(k)\underline{a}^T(k)}{\alpha^f(k)} \end{array}\right]$$

$$=\left[\begin{array}{c|c} \mathbf{r}_{xx}^{-1}(k)+\frac{\underline{b}(k)\underline{b}^T(k)}{\alpha^b(k)} & \frac{-\underline{b}(k)}{\alpha^b(k)} \\ \hline \frac{-\underline{b}^T(k)}{\alpha^b(k)} & \frac{1}{\alpha^b(k)} \end{array}\right] \qquad [2.46]$$

Having derived expressions for the increased order matrix $\mathbf{r}_{xx}(\mathbf{k})$, we may now calculate the increased order gain vector $\underline{c}'(k)$ defined by

$$\mathbf{r}'_{xx}(k)\underline{c}'(k)=\underline{X}'(k) \qquad [2.47]$$

Using the forward form of [2.41] and [2.46], along with definition [2.47], the following result is obtained

$$\underline{c}'(k)=\begin{bmatrix}0\\ \underline{c}(k-1)\end{bmatrix}+\frac{\epsilon^f(k)}{\alpha^f(k)}\begin{bmatrix}1\\ -\underline{a}(k)\end{bmatrix} \qquad [2.48]$$

where $\epsilon^f(k)=x(k)-\underline{a}^T(k)\underline{X}(k-1)$. $\epsilon^f(k)$ is known as the *a posteriori* forward prediction error.

Similarly, using the backward form of [2.41] and [2.46] along with definition [2.47],

$$\underline{c}'(k)=\begin{bmatrix}\underline{c}(k)\\0\end{bmatrix}+\frac{\epsilon^b(k)}{\alpha^b(k)}\begin{bmatrix}-\underline{b}(k)\\1\end{bmatrix} \qquad [2.49]$$

where $\epsilon^b(k)=x(k-N)-\underline{b}^T(k)\underline{X}(k)$ is the *a posteriori* backwards prediction error.

Next, the extended gain vector, $\underline{c}'(k)$ is considered to be partitioned as

$$\underline{c}'(k)=\begin{bmatrix}\underline{d}(k)\\---\\\delta(k)\end{bmatrix} \qquad [2.50]$$

It is clear from [2.49] that

$$\delta(k)=\frac{\epsilon^b(k)}{\alpha^b(k)} \qquad [2.51]$$

and

$$\underline{d}(k)=\underline{c}(k)-\delta(k)\underline{b}(k) \qquad [2.52]$$

Substituting from [2.36] into [2.52],

$$\begin{aligned}\underline{d}(k)&=\underline{c}(k)-\delta(k)\left[\underline{b}(k-1)+\underline{c}(k)e^b(k)\right]\\&=\underline{c}(k)\left[1-\delta(k)e^b(k)\right]-\delta(k)\underline{b}(k-1)\end{aligned} \qquad [2.53]$$

This readily yields the important fast update for the gain vector, given by

$$\underline{c}(k)=\frac{\underline{d}(k)+\delta(k)\underline{b}(k-1)}{\left[1-\delta(k)e^b(k)\right]} \qquad [2.54]$$

To summarise, the calculation of the new gain vector is as follows.

(1) The extended gain vector, $\underline{c}'(k)$ may be computed from the previous N th order gain vector $\underline{c}(k-1)$ using [2.48].

(2) The values of $\underline{d}(k)$ and $\delta(k)$ may be extracted from $\underline{c}'(k)$ by partitioning as in [2.50].

(3) Using [2.54], a fast update of the gain vector may now be performed

All that remains to complete the algorithm is to derive a recursion for $\alpha^f(k)$, which is required to perform step (1) above. From definition [2.31], it is clear that $\underline{r}_x^f(k)$ can be generated recursively, using

$$\underline{r}_x^f(k)=\underline{r}_x^f(k-1)+\underline{X}(k-1)x(k) \qquad [2.55]$$

By definition,

$$\begin{aligned}\alpha^f(k)&=r_0^f(k)-\underline{a}^T(k)\underline{r}_x^f(k)\\&=r_0^f(k-1)+x^2(k)-\left[\underline{a}^T(k-1)+e^f(k)\underline{c}^T(k-1)\right]\underline{r}_x^f(k)\end{aligned}$$

Using the relationship $\underline{r}_x^f(k)=\underline{r}_x^f(k-1)+\underline{X}(k-1)x(k)$, this may be rewritten

$$\begin{aligned}\alpha^f(k)=r_0^f(k-1)+x^2(k)-\underline{a}(k-1)&\left[\underline{r}_x^f(k-1)+\underline{X}(k-1)x(k)\right]-e^f(k)\\&\bullet\underline{c}^T(k-1)\underline{r}_x^f(k)\end{aligned}$$

Using $\underline{r}_x^f(k)=\mathbf{r}_{xx}(k-1)\underline{a}(k)$ and $\alpha^f(k-1)=r_0^f(k-1)-\underline{a}^T(k-1)\underline{r}_x^f(k-1)$

$$\begin{aligned}\alpha^f(k)&=\alpha^f(k-1)+x^2(k)-\underline{a}^T(k-1)\underline{X}(k-1)x(k)-e^f(k)\underline{c}^T(k-1)\\&\quad\bullet\mathbf{r}_{xx}(k-1)\underline{a}(k)\\&=\alpha^f(k-1)+x^2(k)-\underline{a}^T(k-1)\underline{X}(k-1)x(k)-e^f(k)\underline{X}^T(k-1)\underline{a}(k)\\&=\alpha^f(k-1)+x^2(k)-\underline{a}^T(k-1)\underline{X}(k-1)x(k)\\&\quad-\left[x(k)-\underline{a}^T(k-1)\underline{X}(k-1)\right]\underline{X}^T(k-1)\underline{a}(k)\\&=\alpha^f(k-1)+x^2(k)-\underline{a}^T(k-1)\underline{X}(k-1)x(k)-x(k)\underline{X}^T(k-1)\underline{a}(k)\\&\quad+\underline{a}^T(k-1)\underline{X}(k-1)\underline{X}^T(k-1)\underline{a}(k)\\&=\alpha^f(k-1)+\left\{X(k)-\underline{a}^T(k-1)\underline{X}(k-1)\right\}\left\{X(k)-\underline{X}^T(k-1)\underline{a}(k)\right\}\\&=\alpha^f(k-1)+e^f(k)\epsilon^f(k)\end{aligned} \qquad [2.56]$$

This completes the derivation of the fast Kalman algorithm. The complete algorithm is listed in Table 2.2.

- Initialisation

$\underline{H}(0)=\underline{0}$, $\underline{X}(0)=\underline{0}$

$\underline{a}(0)=\underline{0}$, $\underline{b}(0)=\underline{0}$

$\alpha^f(0)=\sigma$, a small positive number

- At time k, do

$$e^f(k)=x(k)-\underline{a}^T(k-1)\underline{X}(k-1)$$

$$\underline{a}(k)=\underline{a}(k-1)+\underline{c}(k-1)e^f(k)$$

$$\epsilon^f(k)=x(k)-\underline{a}^T(k)\underline{X}(k-1)$$

$$\alpha^f(k)=\lambda\alpha^f(k-1)+\epsilon^f(k)e^f(k)$$

$$\underline{c}'(k)=\begin{bmatrix} \dfrac{\epsilon^f(k)}{\alpha^f(k)} \\ \underline{c}(k-1)-\underline{a}(k)\dfrac{\epsilon^f(k)}{\alpha^f(k)} \end{bmatrix}$$

Partition $\underline{c}'(k)$ as $\left[\begin{array}{c} \underline{d}(k) \\ \hline \delta(k) \end{array}\right]$

$$e^b(k)=x(k-N)-\underline{b}^T(k-1)\underline{X}(k)$$

$$\underline{b}(k)=\frac{\underline{b}(k-1)+\underline{d}(k)e^b(k)}{1-\delta(k)e^b(k)}$$

$$\underline{c}(k)=\underline{d}(k)+\delta(k)\underline{b}(k)$$

This completes the fast update of the gain vector. $\underline{H}(k)$ is updated in the same way as the conventional least squares algorithm.

$$e(k)=d(k)-\underline{H}^T(k-1)\underline{X}(k)$$

$$\underline{H}(k)=\underline{H}(k-1)+e(k)\underline{c}(k)$$

Table 2.2 : The fast Kalman algorithm

2.7. The Fast A Posteriori Error Sequential Technique

The fast a posteriori error sequential technique (FAEST) is derived in a similar way to the fast Kalman algorithm presented above. It was proposed by Carayannis, Manolakis and Kalouptsidis in 1983[2] and is computationally more efficient than the fast Kalman algorithm.

Inspection of Table 2.2 reveals that the fast Kalman algorithm is more dependent upon the *a priori* error formulation than the *a posteriori* error formulation, requiring the calculation of both forward and backward *a priori* errors, but only using the forward *a posteriori* error, the *a priori* forward prediction error only being required to update the predictor coefficients $\underline{a}(k)$ to enable the *a posteriori* error to be calculated. The FAEST algorithm, however, is mainly *a posteriori* error based and it also manages to exploit the relationships which exist between *a priori* errors and *a posteriori* errors. An alternative gain vector defined by

$$\tilde{\underline{c}}(k)=\frac{\underline{c}(k)}{\alpha^f(k)} \qquad [2.57]$$

is used. A recursive scheme for updating $\tilde{\underline{c}}(k)$ can be developed using an extended gain vector $\tilde{\underline{c}}'(k)$ in exactly the same way as was done for the fast Kalman algorithm in section 2.6.

The algorithm is presented in Table 2.3.

- Initialisation

$\underline{H}(0)=\underline{0}$, $\underline{X}(0)=\underline{0}$

$\underline{a}(0)=\underline{0}$, $\underline{b}(0)=\underline{0}$

$\alpha^b(0)=\sigma$, a small positive number

$\alpha^f(0)=\sigma\lambda^N$

- At time k, do

$e^f(k)=x(k)-\underline{a}^T(k-1)\underline{X}(k-1)$

$\epsilon^f(k)=\dfrac{e^f(k)}{\alpha(k-1)}$

$\underline{a}(k)=\underline{a}(k-1)+\underline{c}(k-1)\epsilon^f(k)$

$\alpha^f(k)=\lambda\alpha^f(k-1)+\epsilon^f(k)e^f(k)$

$\bar{\underline{c}}'(k)=\begin{bmatrix}0\\ \bar{\underline{c}}(k-1)\end{bmatrix}-\dfrac{e^f(k)}{\lambda\alpha^f(k)}\begin{bmatrix}1\\ \underline{a}(k)\end{bmatrix}$

Partition $\bar{\underline{c}}'(k)$ as $\left[\begin{array}{c}\underline{d}(k)\\ \hline \delta(k)\end{array}\right]$

$e^b(k)=-\delta(k)\alpha^b(k-1)$

$\underline{c}(k)=\underline{d}(k)-\delta(k)\underline{b}(k-1)$

$\alpha'(k)=\alpha(k-1)+\dfrac{e^f(k)}{\lambda\alpha^f(k-1)}e^f(k)$

$\alpha(k)=\alpha'(k)+\delta(k)e^b(k)$

$\epsilon^b(k)=\dfrac{e^b(k)}{\alpha(k)}$

$\alpha^b(k)=\alpha^b(k-1)+\epsilon^b(k)e^b(k)$

$\underline{b}(k)=\underline{b}(k-1)+\bar{\underline{c}}(k)\epsilon^b(k)$

This completes the fast update of the gain vector. $\underline{H}(k)$ is updated, using the alternative gain vector, $\bar{\underline{c}}(k)$ as follows

$e(k)=d(k)-\underline{H}^T(k-1)\underline{X}(k)$

$\epsilon(k)=\dfrac{e(k)}{\alpha(k)}$

$\underline{H}(k)=\underline{H}(k-1)+\epsilon(k)\bar{\underline{c}}(k)$

Table 2.3 : The FAEST Algorithm

2.8. The Fast Transversal Filters Algorithm

The fast transversal filters algorithm was first presented by Cioffi and Kailath in 1984[3]. The most significant feature of this algorithm as compared with the FAEST algorithm is the availability of a fast exact initialisation procedure.

Solving the least squares problem corresponds to solving the set of N linear simultaneous equations described by relationship [2.9]. Unfortunately at time $k<N$, the solution to [2.9] is underdetermined, as there are N equations to be solved, but less than N data points available. This situation corresponds to the matrix $\mathbf{r}_{xx}(k)$ being singular. It is for this reason that the conventional least squares algorithm has an initialisation procedure which involves setting $\mathbf{r}_{xx}^{-1}(0)=\sigma\mathbf{I}$ and the fast Kalman and FAEST algorithms set $\alpha^f(0)=\sigma$, where σ is a small positive number. These initialisations ensure that the matrix $\mathbf{r}_{xx}(k)$ has an inverse for $k<N$, but they also result in a small transient in the solution produced by the algorithm just after it is started.

The FTF algorithm overcomes this as it is simultaneously time and order recursive for time $k<N$. This means that at time $k=1$, a first order filter is generated and this is updated at time $k=2$ to produce a second order filter and so on until the full Nth order filter is determined. In this way, the number of simultaneous equations which are being solved by the algorithm never exceeds the number of data points available to it and the solution is always uniquely determined, avoiding the need for inexact initialisation.

The exact initialisation procedure for the FTF algorithm is listed in Table 2.4 and the steady state algorithm is listed in Table 2.5. A rescue procedure for restarting the FTF algorithm is given in Table 2.6

k=0: $\underline{a}_1(0)=\underline{b}_1(0)=1$, $\tilde{\underline{c}}_0(0)=0$(zero dimension)
where the subscript associated with a vector indicates its dimensionality
$\underline{H}_1(0)=\frac{-d(0)}{x(0)}$, $\gamma(0)=1$, $\alpha^f(0)=x(0)^2$

A simultaneous time and order recursive process is now started

$1\leq k\leq N$:

$e^f(k)=\underline{a}_k(k-1)\left[x(k),\ldots,x(1)\right]$

$\underline{a}_{k+1}(k)=\left[\underline{a}_k(k-1),\frac{-e^f(k)}{x(0)}\right]^T$

$\epsilon(k)=e^f(k)\gamma(k-1)$

$\alpha^f(k)=\lambda\alpha^f(k-1)$

$\alpha^f_{k-1}(k)=\alpha^f(k)+e^f(k)\epsilon^f(k)$

$\gamma(k)=\gamma(k-1)\frac{\alpha^f(k)}{\alpha^f_{k-1}(k)}$

$\tilde{\underline{c}}_k(k)=\left[0\ \tilde{\underline{c}}_{k-1}(k-1)\right]^T-\frac{\epsilon^f(k)}{\alpha^f(k)}\underline{a}_k(k-1)$

$\underline{b}_{k+1}(k)=\left[\left(x(0)\gamma(k)\tilde{\underline{c}}_k(k)\right)^T\ 1\right]^T$ (Only when $k=N$)

$\alpha^b(k)=x(0)^2\gamma(k)$ (Only when $k=N$)

$e(k)=d(k)-\underline{H}_k(k-1)\underline{X}_k^T(k)$

$\epsilon(k)=e(k)\gamma_k(k)$

if $k<N$, $\underline{H}_{k+1}(k)=\left[\underline{H}_k(k-1)\ \ \frac{-e(k)}{x(0)}\right]^T$

if $k=N$, $\underline{H}_{k+1}(k)=\underline{H}_k(k-1)+\epsilon(k)\tilde{\underline{c}}(k)$

Table 2.4:The fast exact initialisation procedure for the FTF algorithm

During the time and order recursive initialisation procedure, subscripts are used to indicate the order of each of the vectors $\underline{a}(k)$, $\underline{b}(k)$, $\tilde{\underline{c}}(k)$ and $\underline{H}(k)$. After initialisation, $\underline{a}(k)$ and $\underline{b}(k)$ will be of dimension $N+1$. $\tilde{\underline{c}}(k)$ and $\underline{H}(k)$ will be of order N. After initialisation, the algorithm is time recursive only and the order subscripts notation will be dropped to simplify the algorithm and facilitate comparision with the other fast RLS algorithms.

$$e^f(k) = \underline{a}(k-1)\underline{X}'^T(k)$$

$$\epsilon^f(k) = e^f(k)\gamma(k-1)$$

$$\alpha^f(k) = \lambda\alpha^f(k-1) + e^f(k)\epsilon^f(k)$$

$$\gamma'(k) = \frac{\lambda\alpha^f(k-1)}{\alpha^f(k)}\gamma(k-1)$$

$$\tilde{c}'(k) = \begin{bmatrix} 0 & \tilde{c}(k-1) \end{bmatrix}^T - \frac{1}{\lambda}e^f(k)\alpha^{f-1}(k-1)\underline{a}(k-1)$$

Partition $\tilde{c}'(k)$ as $\left[\begin{array}{c} \underline{d}(k) \\ \hline \delta(k) \end{array}\right]$

$$\underline{a}(k) = \underline{a}(k-1) + \epsilon^f(k)\begin{bmatrix} 0 & \tilde{c}(k-1) \end{bmatrix}^T$$

$$e^b(k) = -\lambda\alpha^b(k-1)\delta(k)$$

$$\gamma(k) = \left[1 + e^b(k)\gamma'(k)\delta(k)\right]^{-1}\gamma'(k)$$

rescue variable † $= \left[1 + e^b(k)\gamma'(k)\delta(k)\right]$

$$\epsilon^b(k) = e^b(k)\gamma(k)$$

$$\alpha^b(k) = \lambda\alpha^b(k-1) + e^b(k)\epsilon^b(k)$$

$$\begin{bmatrix} \tilde{c}(k) & 0 \end{bmatrix}^T = \tilde{c}'(k) - \delta(k)\underline{b}(k-1)$$

$$\underline{b}(k) = \underline{b}(k-1) + \epsilon^b(k)\begin{bmatrix} \tilde{c}(k) & 0 \end{bmatrix}^T$$

$$e(k) = d(k) - \underline{H}(k-1)\underline{X}(k)$$

$$\epsilon(k) = e(k)\gamma(k)$$

$$\underline{H}(k) = \underline{H}(k-1) + \epsilon(k)\tilde{c}(k)$$

Table 2.5:The steady state FTF algorithm.

† Rescue using reinitialisation procedure of Table 2.6 if rescue variable is negative

$$\underline{a}(-1) = \left[1,0,\ldots,0\right]^T$$

$$\underline{b}(-1) = \left[0,\ldots,0,1\right]^T$$

$$\underline{c}(-1) = \left[0,\ldots,0\right]^T$$

$$\alpha^f(-1) = \lambda^N \mu$$

$$\alpha^b(-1) = \mu$$

$$\gamma(-1) = 1$$

$$\underline{H}(-1) = \underline{H}_{init}$$

Table 2.6:The reinitialisation procedure for the FTF algorithm. μ is a soft constraint which determines the influence of the initial solution, $\underline{H}_{init}$, on future solutions.

2.9. Comparison of the Least Squares Algorithms

It is of interest to compare the resource requirements of the various algorithms that have been presented so far. The algorithms will be compared by considering the number of additions/subtractions, the number of multiplications and the number of divisions required per iteration. Often, only the number of multiplications per iteration is considered, making the assumption that in an implementation multiplication is considerably more complicated to perform than addition or subtraction. On most digital signal processors (DSPs)[90-96], however, multiplication can be performed in a single instruction cycle and is not therefore any more time consuming to perform than addition or subtraction. Division, on the other hand often has to be implemented using the binary equivalent of a long division process and can therefore contribute heavily to the computational load of an algorithm. In deriving Table 2.7, it has been assumed that changing the sign of a number involves a subtraction

operation.

Also of importance in determining the resources required to implement any algorithm on a digital processor are the number of storage cells used by it. A storage cell is the amount of memory required to store a single quantity used by the algorithm. In the case of a fixed point implementation, it is likely to be a single word of memory, but in the case of a floating point implementation, a number of words of memory are likely to be used to store each variable. It should be noted that in the calculation of the number of storage cells required, it has been assumed that variables which are no longer needed by the algorithm may be overwritten, so that the same memory location may store several different intermediate results during the updating of the algorithm

Algorithm	*	/	+,-	Storage Cells
RLS	$2.5N^2+4.5N$	1	$1.5N^2+2.5N$	$\frac{1}{2}N^2+3N$
Fast Kalman	$10N+3$	2	$9N+6$	$5N+5$
FAEST	$7N+10$	4	$7N+8$	$5N+5$
FTF(steady state)	$7N+14$	3	$7N+7$	$6N+11$

Table 2.7 : A comparision of recursive least squares algorithms

2.10. Numerical Instability

Unfortunately, all of the fast RLS algorithms are numerically unstable[44, 97] when implemented using either a fixed or floating point[79-81] digital processor. This means that small numerical errors which occur due to the finite precision of the arithmetic at each iteration of the algorithm accumulate until the algorithm diverges and produces a solution which is completely invalid in a least squares sense. It is for this reason that few practical adaptive filtering systems have made use of the fast

algorithms.

The cause of the problem may be illustrated as follows[44]. Essentially, all of the fast RLS algorithms have a core involving the following recursions:

$$\begin{bmatrix} \underline{a}(k) \\ \tilde{\underline{c}}'(k) \end{bmatrix} = \Theta(k) \begin{bmatrix} \underline{a}(k-1) \\ [0 \; \tilde{\underline{c}}(k-1)] \end{bmatrix} \qquad [2.58]$$

$$\begin{bmatrix} \underline{b}(k) \\ [\tilde{\underline{c}}(k) \; 0] \end{bmatrix} = \Phi(k) \begin{bmatrix} \underline{b}(k-1) \\ \tilde{\underline{c}}'(k) \end{bmatrix} \qquad [2.59]$$

The various fast algorithms are associated with slightly different 2×2 transformation matrices $\Theta(k)$ and $\Phi(k)$, which apply different time varying scalings to the filters $\underline{a}(k)$, $\underline{b}(k)$ and $\tilde{\underline{c}}(k)$. The properties of the transformation and hence the numerical properties of the algorithm may be determined by an eigenvalue and eigenvector analysis of the matrices. In particular, eigenvalues with a magnitude of greater than unity indicate numerical instability, as they indicate that small errors are magnified by the transformation.

Considering the FTF algorithm, the matrix $\Theta(k)$ is given by

$$\Theta(k) = \begin{bmatrix} 1 & \epsilon^f(k) \\ -\dfrac{e^f(k)}{\lambda \alpha^f(k-1)} & 1 \end{bmatrix} \qquad [2.60]$$

which has the eigenvalues

$$q_\Theta(k) = 1 \pm j \sqrt{\frac{e^f(k)\epsilon^f(k)}{\lambda \alpha^f(k-1)}} \qquad [2.61]$$

and $\Phi(k)$ is given by

$$\Phi(k) = \begin{bmatrix} 1 & \epsilon^b(k) \\ -\delta(k) & 1 \end{bmatrix} \qquad [2.62]$$

yielding the eigenvalues

$$q_\Phi(k) = 1 \pm j \sqrt{\epsilon^b(k)\delta(k)} \qquad [2.63]$$

It should be noted from [2.61] and [2.63] that both transformations always have eigenvalues which are greater in magnitude than unity. Performing an infinite

sequence of these transformations is therefore an unstable process.

2.10.1. Normalised Algorithms

Normalised versions of the FTF algorithm have been developed[77, 78] which have improved numerical precision in finite precision implementations. By transforming the variables in the algorithm, it is possible to reduce the dynamic range of the quantities which are to be stored and so they may be represented more accurately.

The disadvantage of the normalised algorithms is their increased computational requirements. The normalised form requires $O(11N)$ multiplications per iteration, as compared with $O(7N)$ for the unnormalised form. Furthermore, normalisation requires a number of square root operations to be performed at each iteration. The practical difficulties in implementing a fast square root operation may make the use of the normalised versions impractical.

It should also be noted that the normalised algorithms still have numerical instability problems, although they will take a larger number of iterations to diverge than the unnormalised forms. Following the eigenvalue analysis of section 2.10, it can be shown that the $\Theta(k)$ matrix defined in equation [2.58] has eigenvalues of $\pm$ 1 and so propagates numerical errors in a stable manner. The matrix $\Phi(k)$, defined in [2.5], however, has eigenvalues greater in magnitude than unity and so the associated 2×2 transformation causes numerical errors to be magnified, leading to eventual instability and divergence.

2.10.2. Lattice Algorithms

As well as the fast RLS algorithms for transversal filtering which have been discussed in this chapter, there are also a number of fast least squares algorithms[98-101] using the lattice filter structure[14]. The main difference between the lattice algorithms and the transversal algorithms is that whilst the transversal algorithms are time recursive, the lattice algorithms are time and order recursive. At each time iteration, k, a recursive process is started, which calculates an $m+1$th order least squares solution from the current m-th order solution, until the desired Nth order solution is obtained.

It can be shown that for these algorithms, the transformation required to perform time updating is an orthogonal rotation, which is well known to be numerically stable. The order update transformation is a hyperbolic rotation, which is numerically unstable, having at least one eigenvalue which is greater than unity. Fortunately, this unstable transformation is only performed in a finite sequence, until the Nth order solution is obtained. For this reason the lattice forms of the fast RLS algorithms can be made to be numerically stable, unless the filter order N is very large.

Unfortunately, the computational complexity of the lattice algorithms is at least double that of their transversal filter counterparts. Moreover, in certain applications such as adaptive channel identification, it is the transversal filter coefficients which are of interest. Methods do exist to convert lattice coefficients to yield an equivalent transversal filter[5], but the conversion requires $\frac{1}{2}\left[(N-1)(N-2)\right]$ multiplications and subtractions to convert an Nth order filter. This complexity is dependent upon the square of the filter length and so the main advantage of using a fast algorithm is lost. For these reasons, there are several applications in which the use of a fast RLS transversal filter algorithm would be highly desirable.

Fast RLS algorithms can also be implemented using QR decomposition techniques[102-104]. These methods use a transformation known as the Givens rotation[105], which has good error propagation properties. These implementations of the fast RLS algorithms are of interest, as the structure which is obtained is a systolic array, which is suitable for implementation using a parallel processing system, or a dedicated VLSI architecture.

2.10.3. Stabilisation by Regular Reinitialisation

One way of using the fast RLS transversal algorithms is to reinitialise them before divergence occurs. The reinitialisation may be performed either periodically in time[97], or when the internal variables of the algorithm suggest that divergence is beginning to occur[3, 106, 107].

In either case, the prewindowed assumption that all data is zero before the algorithm starts will clearly not be valid immediately after reinitialisation and hence the post-windowed or covariance forms of the algorithms must be used.

It would be undesirable for the algorithm to have to reconverge after reinitialisation. Fortunately, it is possible to circumvent this by means of a 'soft-constrained' initial solution. This corresponds to modifying the algorithm to minimise the modified least squares cost function

$$J_3(k)=\sum_{i=0}^{k}\lambda^{k-i}e^2(i)+\mu\lambda^k\,\|\underline{H}(k)-\underline{H}_{init}\|^2 \qquad [2.64]$$

The first term of this cost function is the usual sum of errors squared term. The second term limits the difference between the current solution $\underline{H}(k)$ and some initial solution denoted by $\underline{H}_{init}$. The factor μ controls the balance between the two terms and determines how strongly the initial solution will influence the minimisation. As $k\rightarrow\infty$, the first term will dominate the cost function and so the effects of the

initial solution die out as k becomes large.

The principle advantage in using reinitialisation methods to stabilise fast RLS algorithms is that they can be implemented with little or no additional computational burden, as compared with the unstable forms of the algorithm.

There are two disadvantages in using the reinitialisation methods. Firstly, the tracking performance of the algorithm in a non-stationary environment could be significantly impaired if μ in [2.64] is chosen to be large, due to the constraining effect of the initial solution $\underline{H}_{init}$. Secondly, there is a difficulty in determining how frequently the algorithm must be reinitialised so as to guarantee that divergence will never occur, or alternatively to provide a sufficient method of monitoring the internal variables which will always indicate the imminent divergence of the algorithm.

2.10.4. Error Feedback

One promising development in improving the stability of the FTF algorithm has been the use of error feedback techniques.[108-110]. Whilst the absolute stability of these techniques is still not guaranteed, a very worthwhile improvement in stability is obtained, as compared with the unstabilised algorithm. The penalty is that the computational complexity of the algorithm is somewhat increased by the improvements - for the algorithm of[110] it is increased from $7N$ multiplications per iteration for the unstable unnormalised algorithm to $10N$ for its stabilised counterpart and from $10N$ for the unstable normalised algorithm to $11N$ for its stabilised counterpart.

These techniques rely on the ability to compute certain variables in the algorithm in two different ways. The difference between the two variables should be representative of the amount of numerical error which has accumulated. By modifying the least squares cost function to have a joint objective of minimising both the filter

error and also the numerical error, a numerical error feedback path is introduced. The effect of this should be that the fast least squares algorithm seeks not only to solve the adaptive filtering problem, but that it also attempts to cancel the effects of its own finite precision errors.

The stabilised algorithms will produce a solution which is slightly sub-optimal in a least squares sense, due to the combined cost function which involves not only filter error but also numerical error. Moreover the proof of absolute stability for these techniques is almost impossible. These methods have been shown in simulation[110], however, to give good performance with a solution which does not differ significantly from that obtained using the conventional RLS algorithm over of the order of ½ million iterations.

2.11. Conclusions

This chapter has introduced least squares adaptive filtering. Various algorithms for performing the least squares updating of the filter coefficients have been presented, such as the conventional recursive least squares (RLS) algorithm and the highly computationally efficient fast algorithms - the fast Kalman algorithm, the FAEST algorithm and the FTF algorithm.

The major problem associated with the transversal forms of the more efficient algorithms is their numerical instability problems. Small truncation errors which occur at each iteration of the algorithm due to the finite precision of the arithmetic used to implement it accumulate, until eventually the algorithm must diverge.

Various ways of improving the stability of the algorithm have been considered. Normalisation reduces the dynamic range of the quantities which have to be stored, improving somewhat the numerical properties of the algorithm. Rescue procedures are available which may be used either periodically in time, or when some

divergence detector indicates that the algorithm has accumulated too much numerical error. Error feedback has also been considered as a means of stabilising the FTF algorithm.

In the next chapter, a new method of stabilising the fast RLS algorithms will be considered. It will use a rescue procedure, reinitialising the algorithm before divergence occurs. To detect that divergence is about to occur, a scheme of arithmetic known as interval arithmetic is used. The algorithm is modified, so that it not only calculates the least squares solution to an adaptive filtering problem, but it also calculates upper and lower bounds to that solution, taking into account the numerical errors which may have occurred. If the difference between the upper and lower bounds is excessive, the reinitialisation procedure is performed, preventing divergence.

3 Interval Arithmetic

3.1. Introduction

In this chapter, a new solution to the stability problems associated with the fast RLS algorithms will be introduced. By using a scheme of arithmetic known as interval arithmetic[111], an error analysis is effectively performed in parallel with the computations of the algorithm[112-114]. The result of this error analysis is a measure of the confidence which may be placed upon the performance of the algorithm. If the error analysis indicates that divergence is about to occur, then the algorithm may be rescued, using the reinitialisation procedures discussed in the previous chapter.

This chapter will begin by defining the interval number system and will then discuss how interval numbers may be combined to yield results which are also interval numbers. The arithmetic operations { $+$, $-$, $\times$ and $\div$ } will be defined for interval numbers and then, the more complicated operations such as the scalar product of two vectors of interval numbers will be discussed.

Having defined the various operations which are required to perform one iteration of a fast RLS algorithm, the exact way in which interval arithmetic is applied to these adaptive algorithms will then be described. A number of design parameters are introduced into the rescue procedure by the use of interval arithmetic and the chapter will conclude with a discussion on how these parameters may best be chosen.

3.2. Interval Numbers

An interval number is a range of real numbers. The range is bounded by a lower endpoint and an upper endpoint. The interval number is the set of all real numbers which lie between the lower and upper bounds.

The notation used to represent an interval is defined as

$$[a^l, a^u] = \left\{ x : a^l \le x \le a^u, x \in R \right\} \qquad [3.1]$$

Hence the interval number $[a^l, a^u]$ consists of the set of all real numbers which lie between lower bound a^l and upper bound a^u.

Two further definitions will be useful in the application of interval numbers to the fast RLS adaptive algorithms. The width of an interval number is defined by

$$width\left(\left[a^l, a^u \right] \right) = a^u - a^l \qquad [3.2]$$

and the centre of the interval $[a^l, a^u]$ as:

$$centre\left(\left[a^l, a^u \right] \right) = \frac{a^l + a^u}{2} \qquad [3.3]$$

A single real value can be represented by a *degenerate* interval. Therefore, the single real number ψ is represented by the interval number $[\psi, \psi]$.

3.3. Scalar Interval Arithmetic

The operation $\bullet$ where $\bullet$ is one of { +, −, × or ÷ } is defined for interval numbers by

$$[a, b] \bullet [c, d] = \left\{ x \bullet y : a \le x \le b, c \le y \le d, x \in R, y \in R \right\} \qquad [3.4]$$

except if $\bullet$ is the division operation and $c \le 0$ and $d \ge 0$, which is undefined.

Hence the result of the operation $\bullet$ on two interval numbers is the range which is obtained when both the intervals being combined take their entire range of values.

An equivalent set of definitions is

$$[a^l,a^u]+[b^l,b^u]=[a^l+b^l,a^u+b^u] \quad [3.5]$$

$$[a^l,a^u]-[b^l,b^u]=[a^l-b^u,a^u-b^l] \quad [3.6]$$

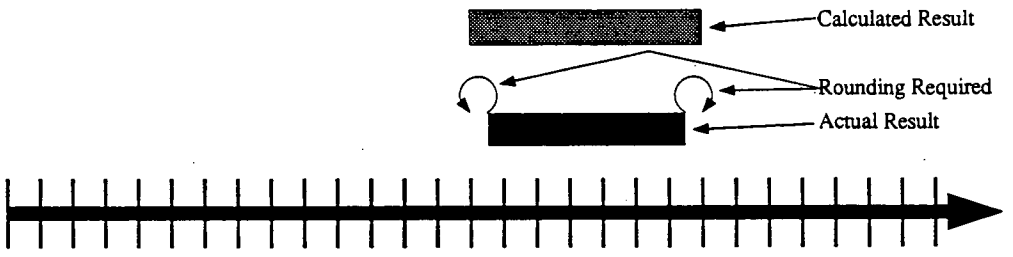
$$[a^l,a^u]\times[b^l,b^u]=[\min(a^lb^l,a^lb^u,a^ub^l,a^ub^u), \quad [3.7]$$
$$\max(a^lb^l,a^lb^u,a^ub^l,a^ub^u)]$$

$$[a^l,a^u]\div[b^l,b^u]=[\min(a^l\div b^l,a^u\div b^l,a^l\div b^u,a^u\div b^u),$$
$$\max(a^l\div b^l,a^u\div b^l,a^l\div b^u,a^u\div b^u)] \quad [3.8]$$

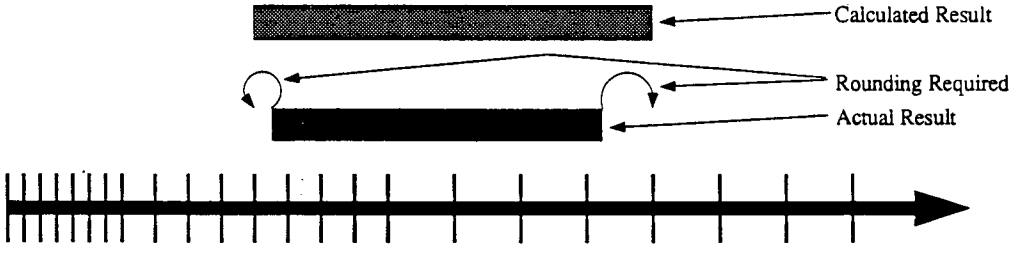
3.4. Scalar Interval Arithmetic with a Finite Precision Processor

There are several practical considerations when implementing functions to perform the operations defined in equations [3.5] - [3.8] on a finite precision processor[115-117]. The motivation for using interval arithmetic rather than single valued real number arithmetic with the fast RLS algorithms is so that the effects of finite precision numerical errors may be considered. Equations [3.5] - [3.8] assume that there are operators { $+$, $-$, $\times$ and $\div$ } which produce an exact result. However, so that the interval arithmetic includes the effects of numerical error, it will be assumed that there are operators { $+\downarrow$, $-\downarrow$, $\times\downarrow$, $\div\downarrow$, $+\uparrow$, $-\uparrow$, $\times\uparrow$ and $\div\uparrow$ } where the symbol $\bullet\downarrow$ is taken to mean the next machine representable number below the infinite precision result of the operation $\bullet$ and the symbol $\bullet\uparrow$ is taken to mean the next machine representable number above the infinite precision result of the operation $\bullet$. This is illustrated on the number lines in Figure 3.1 for fixed and floating point arithmetic.

Using these symbols, the finite precision implementation of the scalar interval arithmetic operations may be defined by



3.1a Fixed Point Rounding



3.1b Floating Point Rounding

Figure 3.1 Fixed and floating point arithmetic number lines showing the effects of the rounding operations ↑ and ↓ required for interval arithmetic.

$$[a^l,a^u]+[b^l,b^u]=[a^l +\downarrow b^l, a^u +\uparrow b^u] \quad [3.9]$$

$$[a^l,a^u]-[b^l,b^u]=[a^l -\downarrow b^u, a^u -\uparrow b^l] \quad [3.10]$$

$$[a^l,a^u]\times[b^l,b^u]=[\min(a^l \times\downarrow b^l, a^l \times\downarrow b^u, a^u \times\downarrow b^l, a^u \times\downarrow b^u), \quad [3.11]$$
$$\max(a^l \times\uparrow b^l, a^l \times\uparrow b^u, a^u \times\uparrow b^l, a^u \times\uparrow b^u)]$$

$$[a^l,a^u] \div[b^l,b^u]=[\min(a^l \div\downarrow b^l, a^u \div\downarrow b^l, a^l \div\downarrow b^u, a^u \div\downarrow b^u),$$
$$\max(a^l \div\uparrow b^l, a^u \div\uparrow b^l, a^l \div\uparrow b^u, a^u \div\uparrow b^u)] \quad [3.12]$$

The reason for modifying the definitions of the interval operations is to ensure that the range calculated using finite precision arithmetic covers all of the infinite precision range. The endpoints are slightly wider apart for the finite precision range than for the infinite precision range. This represents the additional uncertainty in the result produced by the finite precision effects of that calculation.

It should be noted that for fixed point addition and subtraction, the result of combining two machine representable numbers is, in general, another machine representable number, assuming that overflow does not occur and so the operations $+\uparrow$ and $+\downarrow$ are both equivalent to the operation $+$ and similarly $-\uparrow$ and $-\downarrow$ are equivalent to $-$.

The definitions of equations [3.9] - [3.12] could be used to implement a set of functions to perform interval arithmetic on a finite precision processor, but there are more efficient ways of performing multiplication and division than that suggested by [3.11] and [3.12]. By examining the signs of the endpoints a^l, a^u, b^l and b^u, it is usually possible to predict which of the four products $a^l \times b^l$, $a^u \times b^l$, $a^l \times b^u$ or $a^u \times b^u$ will be the greatest and which will be the smallest. In the case of division, it is always possible to predict which of the four results $a^l \div b^l$, $a^u \div b^l$, $a^l \div b^u$ or $a^u \div b^u$ will be the largest and which will be the smallest from a knowledge of the signs of the endpoints. This means that usually only two real multiplications are required to be performed to implement interval multiplication and that interval division may be implemented with two real division operations. The functions required

to perform finite precision scalar interval arithmetic are summarised in Table 3.2.

```
RANGE_ADD(a,b,c,d)

/*A procedure to calculate the result [e,f]=[a,b] + [c,d] */

e=a+c;
f=b+d;

End of procedure.
```

```
RANGE_SUBTRACT(a,b,c,d)

/*A procedure to calculate the result [e,f]=[a,b] - [c,d] */

e=a-d;
f=b-c;

End of procedure.
```

```
RANGE_DIVIDE(a,b,c,d)

/* A procedure to calculate the result [e,f]=[a,b] / [c,d] */

if (c≤0 and d≥0) {
        print "Division by zero error"
        exit
        }
if (c<0) {
   if (b>0) e=b/d; else e=b/c;
   if (a>=0) f=a/c; else f=a/d;
   }
else {
   if (a<0) e=a/c; else e=a/d;
   if (b>0) f=b/c; else f=b/c;
   }

End of procedure.
```

```
RANGE_MULTIPLY(a,b,c,d)

/* Procedure to calculate the result [e,f]=[a,b] * [c,d] */
if (a<0 && c>=0) {
   temp=a; a=c; c=temp; temp=b; b=d; d=temp;
   }
if (a>=0) {
   if (c>=0) {
      e=a*c;
      f=b*d;
      }
   else {
   e=b*c;
   if (d>=0) f=b*d; else f=a*d;
   }
}
else {
   if (b>0) {
      if (d>0) {
         e=min(a*d,b*c);
         f=max(a*c,b*d);
         }
      else   {
         e=b*c;
         f=a*c;
      }
   }

else {
      f=a*c;
      if (d<=0) e=b*d; else e=a*d;
   }
}

End of procedure.
```

Table 3.2 Procedures for performing scalar interval arithmetic.

3.5. Vector Interval Arithmetic

As all of the fast RLS algorithms require vector operations, it is necessary to extend the definitions for interval arithmetic to include vectors. An N dimensional interval vector is written in the form

$$\underline{A}_N = \begin{bmatrix} [a_1^l, a_1^u] \\ [a_2^l, a_2^u] \\ \vdots \\ [a_N^l, a_N^u] \end{bmatrix} \qquad [3.13]$$

Vector addition and subtraction may be easily defined using the existing definitions for scalar interval addition and subtraction. For addition,

$$\underline{A}_N + \underline{B}_N = \begin{bmatrix} [a_1^l, a_1^u] \\ [a_2^l, a_2^u] \\ \vdots \\ [a_N^l, a_N^u] \end{bmatrix} + \begin{bmatrix} [b_1^l, b_1^u] \\ [b_2^l, b_2^u] \\ \vdots \\ [b_N^l, b_N^u] \end{bmatrix} \qquad [3.14]$$

$$= \begin{bmatrix} [a_1^l + b_1^l, a_1^u + b_1^u] \\ [a_2^l + b_2^l, a_2^u + b_2^u] \\ \vdots \\ [a_N^l + b_N^l, a_N^u + b_N^u] \end{bmatrix}$$

and similarly for subtraction which is defined by

$$\underline{A}_N - \underline{B}_N = \begin{bmatrix} [a_1^l, a_1^u] \\ [a_2^l, a_2^u] \\ \vdots \\ [a_N^l, a_N^u] \end{bmatrix} - \begin{bmatrix} [b_1^l, b_1^u] \\ [b_2^l, b_2^u] \\ \vdots \\ [b_N^l, b_N^u] \end{bmatrix} \qquad [3.15]$$

$$= \begin{bmatrix} [a_1^l - b_1^u, a_1^u - b_1^l] \\ [a_2^l - b_2^u, a_2^u - b_2^l] \\ \vdots \\ [a_N^l - b_N^u, a_N^u - b_N^l] \end{bmatrix}$$

It is also necessary to define the scalar product of two interval vectors. This is defined by

$$\underline{A}_N \cdot \underline{B}_N = \begin{bmatrix} [a_1^l, a_1^u] \\ [a_2^l, a_2^u] \\ \vdots \\ [a_N^l, a_N^u] \end{bmatrix} \cdot \begin{bmatrix} [b_1^l, b_1^u] \\ [b_2^l, b_2^u] \\ \vdots \\ [b_N^l, b_N^u] \end{bmatrix} \qquad [3.16]$$

$$= \sum_{i=1}^{N} \left[\min(a_i^l b_i^l, a_i^u b_i^l, a_i^l b_i^u, a_i^u b_i^u), \max(a_i^l b_i^l, a_i^u b_i^l, a_i^l b_i^u, a_i^u b_i^u) \right]$$

3.6. Application of Interval Arithmetic to the Fast RLS Algorithms

Having defined all of the operations required to perform an iteration of the algorithm using interval arithmetic, it is possible to replace all of the single real valued variables in the algorithm with interval numbers.

If this is done, then the solution which is calculated by the algorithm will also become an interval. The difference between the endpoints of this interval, or width as defined in equation [3.2], represents the extent to which the solution has been corrupted by numerical errors. If this difference exceeds some preset limit, then the algorithm must be reinitialised using a rescue procedure such as the one in Table 2.6, so as to prevent divergence.

It is also necessary to reinitialise the algorithm if a division is attempted of the form $[a^l, a^u] \div [b^l, b^u]$ where $b^l \leq 0$ and $b^u \geq 0$, as division by an interval of this form cannot be defined, since zero is a member of the range by which division is being attempted.

Real valued inputs to the adaptive filter are represented by degenerate intervals of the form $[\psi, \psi]$, which is equivalent to the single real value, ψ. Real valued outputs may be obtained by using the centre of the interval output, as defined in equation

[3.3]. Alternatively, either the upper or lower endpoint may be used, assuming that the difference between them is small.

To summarise, the non-interval version of a fast RLS algorithm is converted to its interval counterpart as follows

- All scalar quantities in the algorithm are converted to interval scalar numbers, as described in section 3.2

- All scalar operations are performed using the interval operations in Table 3.2, noting the rounding directions for the upper and lower endpoints discussed in section 3.4

- All vector quantities are similarly replaced by interval vectors and all vector operations by their interval counterparts, as described in section 3.5

- The solution calculated by the algorithm now becomes an interval, with the difference between the upper and lower endpoints representing the extent to which the solution has been corrupted by finite precision errors. Specifically, if the width of any of the filter coefficients exceeds some predefined limit which will be denoted by ρ, then the algorithm should be rescued. To do this, a reinitialisation is performed using the techniques described in section 2.10.3. The initial solution, $\underline{H}_{init}$ is obtained by taking

$$\underline{H}_{init} = \begin{bmatrix} centre\,[h_0^l, h_0^u] \\ centre\,[h_1^l, h_1^u] \\ \vdots \\ centre\,[h_{N-1}^l, h_{N-1}^u] \end{bmatrix} \qquad [3.17]$$

and weighting the initial solution with a soft-constraint factor μ. The choice of ρ and μ is discussed in section 3.7

- Real valued inputs to the filter and desired response inputs are represented by degenerate intervals.

- Real valued outputs are either obtained by taking the centre of the interval output, or the upper or lower endpoint of the interval output.

3.7. Choice of Design Parameters for the Interval Fast RLS Algorithm

There are three design parameters associated with the interval versions of the fast RLS algorithm. The first is the forgetting factor, denoted by λ, which is introduced by the exponential weighting of the input data. This parameter is common to all exponentially windowed least squares adaptive filtering algorithms and it is well known that the choice of λ controls the effective length of the data window. The time constant is approximately $\frac{1}{1-\lambda}$ where λ is just less than 1. This factor controls the tracking performance of the algorithm in non-stationary environments and it will not be discussed further.

The second parameter, ρ is the threshold for reinitialising the algorithm. If the difference between the upper and lower endpoints of any of the filter coefficients exceeds ρ, then the algorithm will be reinitialised. A suitable value for ρ may be chosen if the performance level for the adaptive filter is known. ρ is chosen such that the noise introduced onto the output by arithmetic errors is insignificant compared with the noise from other sources. Assuming a uniform distribution, the noise introduced to the coefficients by arithmetic errors will have a variance $\frac{N\rho^2}{12}$ where N is the length of the adaptive filter. Hence if $\frac{N\rho^2}{12}$ is chosen to be equal to the mean square value of the filter error after convergence, which will be represented by ζ^2, then acceptable performance is usually obtained.

The third design parameter, μ, is introduced by the rescue procedure. It controls the influence of the initial solution $\underline{H}_{init}$ after the algorithm is reinitialised. If it is chosen to be too small, the algorithm will have to reconverge after reinitialisation, resulting in poor performance immediately after each rescue. Too large a value of μ will result in the solution $\underline{H}_{init}$ being weighted with too much importance. As $\underline{H}_{init}$ may be incorrect due to numerical errors, this is undesirable. Moreover, too large a value of μ will impair the tracking performance of the algorithm in a non-stationary environment. A relationship between the values of ρ,λ,N and the suitable value for μ will now be developed. A number of assumptions are invoked in the derivation but nevertheless, the result obtained usually gives a good starting point in the choice of μ.

The derivation begins by considering the difference between the exact least squares solution $\underline{H}_{LS}$ and the initial solution, $\underline{H}_{init}$. Since $\underline{H}_{init}$ and $\underline{H}_{LS}$ are both vectors of degenerate intervals, the notation used will be that of single valued real numbers rather than that of interval arithmetic.

Assuming that each coefficient in $\underline{H}_{init}$ differs from $\underline{H}_{LS}$ by a random variable drawn from a uniform distribution between $-\frac{\rho}{2}$ and $\frac{\rho}{2}$, it is possible to obtain

$$E(||\underline{H}_{LS}-\underline{H}_{init}||^2)=\frac{N\rho^2}{12} \qquad [3.18]$$

Next, consider the expected value of the cost function, $J_3(k)$, given in [2.64] as

$$J_3(k)=\sum_{i=0}^{k}\lambda^{k-i}e^2(i) \;+\; \mu\lambda^k||\underline{H}(k)-\underline{H}_{init}||^2$$

To evaluate the expected value of the cost function, it is necessary to use the approximation $\underline{H}(k) = \underline{H}_{LS}$, which is valid except just after reinitialisation,

$$E(J_3(k))=E(\sum_{i=0}^{k}\lambda^{k-i}e^2(i))+E(\mu\lambda^k||\underline{H}_{LS}-\underline{H}_{init}||^2) \qquad [3.19]$$

$$=\frac{1-\lambda^{k+1}}{1-\lambda}\zeta^2+\frac{\mu\lambda^k N\rho^2}{12}$$

where $\zeta^2=E(e^2(i))$.

In a stationary environment, a good solution is to keep $E(J_3(k))$ constant before and after reinitialisation. This corresponds to the correct balance being maintained between the initial solution $\underline{H}_{init}$ and subsequent solutions $\underline{H}(k)$. This is done by setting

$$E(J_3(k))=E(J_3(\infty)) \qquad [3.20]$$

for all k. Therefore

$$\frac{1-\lambda^{k+1}}{1-\lambda}\zeta^2+\frac{\mu\lambda^k N\rho^2}{12}=\frac{\zeta^2}{1-\lambda} \qquad [3.21]$$

giving

$$\frac{\mu\lambda^k N\rho^2}{12}=\frac{\zeta^2\lambda^{k+1}}{1-\lambda}$$

and hence

$$\mu=\frac{12\zeta^2\lambda}{(1-\lambda)N\rho^2} \qquad [3.22]$$

3.8. Conclusions

In this chapter, a new method for detecting the imminent divergence of a fast RLS adaptive algorithm has been proposed. A scheme of arithmetic known as interval arithmetic has been developed. This scheme of arithmetic enables an error analysis to be performed in real time in parallel with solving the least squares adaptive filtering problem. By reinitialising when the accumulation of finite precision errors on the solution exceeds a predefined maximum limit, divergence is prevented.

The penalty for using interval arithmetic is its increased computational complexity compared with single valued real arithmetic. The computational complexity of an interval algorithm is approximately double that of its non interval counterpart. The complexity remains, however, directly proportional to the filter length and so great savings in computation are still obtained compared with the conventional RLS algorithm for moderately long filters. Furthermore, due to the regular structure of the interval operations, it would be possible to construct a dedicated interval arithmetic

processor from a number of real value arithmetic processors, resulting in a similar speed of operation to the non interval algorithm. In this case, the penalty is the increased hardware complexity.

Results will be presented from software simulations in the next chapter, demonstrating the stability of the interval arithmetic methods, using both fixed and floating point arithmetic and in chapter 5, the implementation of the fixed point version of the interval FTF algorithm on a TMS320C25 digital signal processor[90] will be described.

4 Interval Algorithms - Software Simulations

4.1. Introduction

In this chapter, simulation results for the interval fast RLS algorithms will be presented. The aims of these simulations are twofold. Firstly, they will show that the interval algorithms do not diverge over at least one million iterations, whereas the non-interval fast RLS algorithms diverge fairly rapidly. Secondly, the results will demonstrate that the performance of an interval fast RLS adaptive filter is comparable to that obtained using less computationally efficient least squares techniques.

Two different adaptive filtering configurations[4] will be simulated. Adaptive system identification will be considered as an example of direct system modelling[16, 118, 119] and adaptive equalisation will be performed as an example of inverse system modelling[46, 63, 64].

Both the stationary and non-stationary characteristics of the interval algorithms will be considered. The results of the non-stationary simulations are of particular importance, as it is necessary to demonstrate that the tracking capabilities of the interval algorithms are not significantly impaired by the regular reinitialisations, which must be performed to prevent the algorithm from diverging. The example of a non-stationary system which will be simulated is the fading high frequency (HF) channel[120-123] for digital communications and it will be shown that the error rate

which may be achieved using an interval fast RLS adaptive equaliser is similar to that obtained using the conventional RLS algorithm.

Both floating and fixed point arithmetic are simulated. The floating point number system used was 64 bit floating point arithmetic, with a 56 bit mantissa, a 7 bit exponent and a sign bit. The fixed point arithmetic system used 16 bit truncation with the provision of a 32 bit long accumulator, which may be used during the various scalar product operations in the algorithm to achieve greater accuracy. This 16/32 bit fixed point arithmetic system is typical of that available on many current digital signal processors (DSPs)[90-96] and indeed, the fixed point simulations were used as a starting point for a hardware implementation of one of the fast RLS algorithms, which will be described in detail in the next chapter.

4.2. System Identification

The configuration for adaptive system identification is shown in Figure 4.1. The input signal to the adaptive filter is generated by passing Gaussian noise through a prefilter. The purpose of this prefilter is to provide control over the spectral properties of the adaptive filter input signal. This enables various eigenvalue ratios to be obtained for the input autocorrelation matrix defined by

$$\mathbf{R}_{xx} = \begin{bmatrix} E(x(k)x(k)) & \cdots & E(x(k)x(k+N-1)) \\ E(x(k)x(k+1)) & \cdots & E(x(k)x(k+N-2)) \\ E(x(k)x(k+2)) & \cdots & \vdots \\ \vdots & \cdots & \vdots \\ \vdots & \cdots & E(x(k)x(k-2)) \\ \vdots & \cdots & E(x(k)x(k-1)) \\ E(x(k)x(k+N-1)) & \cdots & E(x(k)x(k)) \end{bmatrix} \qquad [4.1]$$

By varying the eigenvalue ratio, the ill conditioning of the adaptive filtering problem is varied[44]. Table 4.1 shows the two prefilters which were used during simulations and the eigenvalues associated with them for a length 5 adaptive filter.

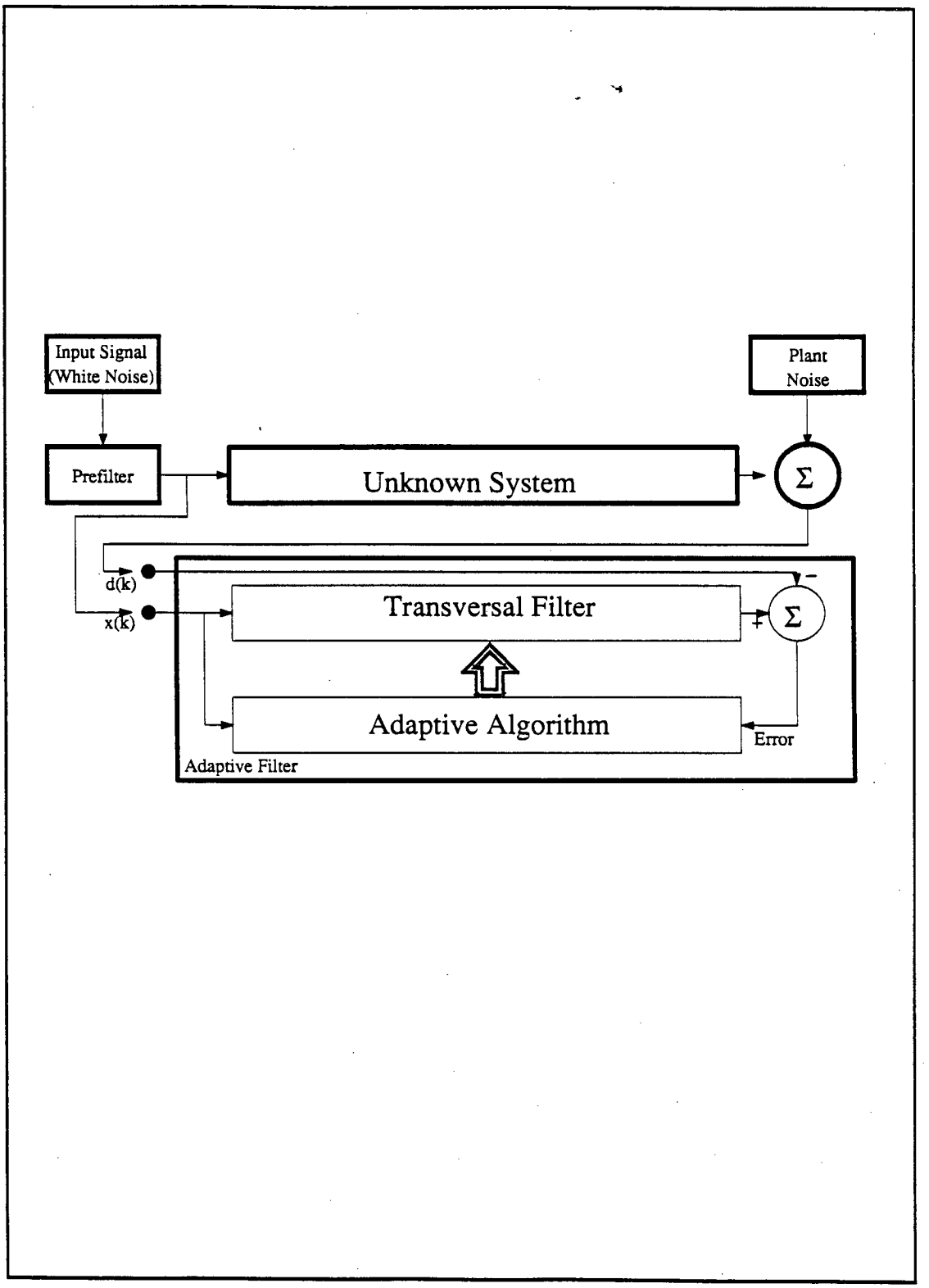


Figure 4.1 Configuration for adaptive system identification

	Prefilter	Eigenvalue ratio †
1	$1.0+0.865z^{-1}$	18.7
2	$1.0+0.600z^{-1}$	7.3

† For a length 5 adaptive filter

Table 4.1 : The eigenvalue ratios obtained using different prefilters for the simulation shown in Figure 4.2.

The input signal to the adaptive filter is also passed through an unknown system, which for all of the system identification simulations in this chapter, was a 5 tap finite impulse response filter with the 5 coefficients randomly chosen between -1 and +1. The output from this unknown system is corrupted by a small amount of additive Gaussian noise. This signal is used as the desired response input for the adaptive filter.

If the adaptive filter is operating correctly, then the output signal from the adaptive filter should be almost equal to the output from the unknown system. If it produces the same output from the same input signal, then it must have the same transfer function as the unknown system, enabling the unknown system to be identified.

The performance of this system is measured by how close the coefficients of the adaptive filter converge to the coefficients of the unknown system. If the coefficients of the unknown filter are denoted by $\underline{H}_{opt}$, then a measure of the performance is given by the norm of the vector of coefficient errors defined as

$$10\times\log_{10}\left\{\frac{||\underline{H}(k)-\underline{H}_{opt}||^2}{||\underline{H}(0)-\underline{H}_{opt}||^2}\right\} \qquad [4.2]$$

in dB.

A large negative value of the performance function of equation [4.2] indicates that the performance of the adaptive filter is good. The level of performance which will be attained after the adaptive filter converges is dependent upon the signal to noise ratio introduced by the noise at the desired response input of the adaptive filter. For simulations, the signal to noise ratio was measured at the desired response input to the adaptive filter.

Figure 4.2 shows the performance of the conventional RLS algorithm when performing system identification. It is included mainly for comparison with the performance of the various fast algorithms in Figures 4.3 - 4.8. The forgetting factor was set at 0.98, the length of the adaptive filter was 5 and a signal to noise ratio of 30dB was used.

It can be seen that the performance measure rapidly drops to below the 30dB noise level at the start of the simulation, as the adaptive filter converges. This rapid initial convergence is typical of a least squares algorithm and it is one of the principal advantages of using least squares techniques. After initial convergence, the solution remains at a low level, as would be expected in a stationary simulation, where the optimum solution does not vary with time.

4.3. Divergence of the FAEST, Fast Kalman and FTF Algorithms

Figures 4.3 - 4.8 show the instability of the fast RLS algorithms. For each of these simulations, the signal to noise ratio was set at 30dB and a forgetting factor of 0.98 was used. The arithmetic system was 64 bit floating point arithmetic and the length of the adaptive filter was 5.

Figures 4.3,4.4 and 4.5 show the numerical instability of the FTF, FAEST and fast Kalman algorithms with an input autocorrelation matrix eigenvalue ratio of 18.3, and Figures 4.6, 4.7 and 4.8 show the same algorithms, but the prefilter has been

changed to yield an eigenvalue ratio of 7.3.

For all of the fast algorithms, it can be seen that the convergence and initial solution are identical to that obtained using the conventional RLS algorithm, but then that the algorithms suddenly diverge and fail to provide a solution which is valid in the least squares sense.

The number of iterations that the algorithm is able to perform before it diverges has been found to vary substantially between different simulation runs, even when using the same algorithms and parameters. This means that comparisons between the different fast RLS algorithms are not particularly easy to perform, but a number of important trends have been noticed.

- The algorithms take longer to diverge at lower eigenvalue ratios. This is as would be expected, since the high eigenvalue ratios result in the least squares filtering problem becoming more ill conditioned, which means that the process is more susceptible to numerical errors.

- The fast Kalman algorithm appears to take longer to diverge than the FTF and FAEST algorithms. This is believed to be due to the increased computational complexity of the fast Kalman algorithm. The additional computations are thought to introduce some redundancy into the algorithm and the errors generated in these redundant calculations tend to cancel each other out to some extent, resulting in smaller errors at each iteration and hence a larger number of iterations can be performed before they accumulate to the extent that divergence occurs.

- The solution after divergence of the fast Kalman algorithm appears to be different from that of the FTF and FAEST algorithms. Although the fast Kalman algorithm no longer produces a useful solution, the filter coefficients appear to be bounded, whereas for the FTF and FAEST algorithms, after divergence,

the coefficients increase without limit.

- All of the algorithms eventually diverge. This is a direct result of the transition matrix eigenvalue analysis presented in section 2.10.

4.4. FTF Algorithm Using Rescue Variable

In the paper in which the FTF algorithm was presented[3], it was suggested that numerical stability could be improved by using a rescue variable. This has not been done in the results of Figures 4.3 and 4.6, so that the results from the various unstabilised algorithms could be compared.

The rescue is performed by reinitialising, using the method described in section 2.10.3. A rescue should be performed if, during any iteration of the algorithm, the quantity

$$[1+e^b(k)\gamma'(k)\delta(k)] \qquad [4.3]$$

is negative. This quantity should be positive at all times, since for an infinite precision implementation,

$$[1+e^b(k)\gamma'(k)\delta(k)]=\frac{\lambda\alpha^b(k-1)}{\alpha^b(k)} \qquad [4.4]$$

From the definition of $\alpha^b(k)$ in equation [2.38], $\alpha^b(k)$ is the minimum value of a sum of squares of backwards prediction errors and so it is a positive quantity. Hence, the ratio in equation [4.4] should be positive at all times.

The results which are obtained using this rescue procedure are shown in Figure 4.9. The rescue procedure gives a worthwhile improvement in the number of iterations for which the FTF algorithm produces a useful solution, but divergence of the algorithm still occurs.

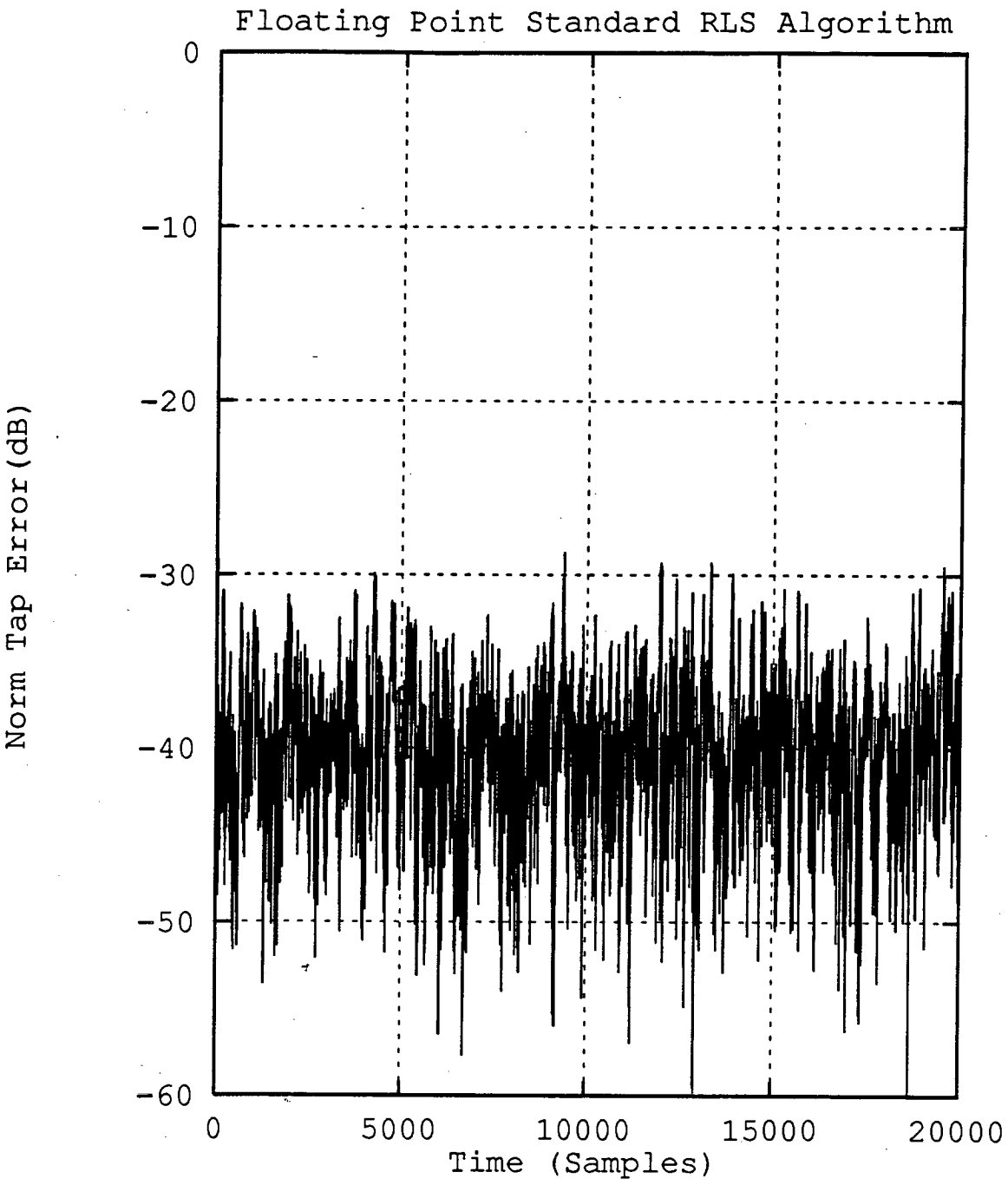


Figure 4.2 Performance of the conventional RLS algorithm in performing stationary system identification. $\lambda = 0.98$, SNR=30dB. input autocorrelation EVR=18.7. After the algorithm converges, it remains at a good solution for the duration of the simulation.

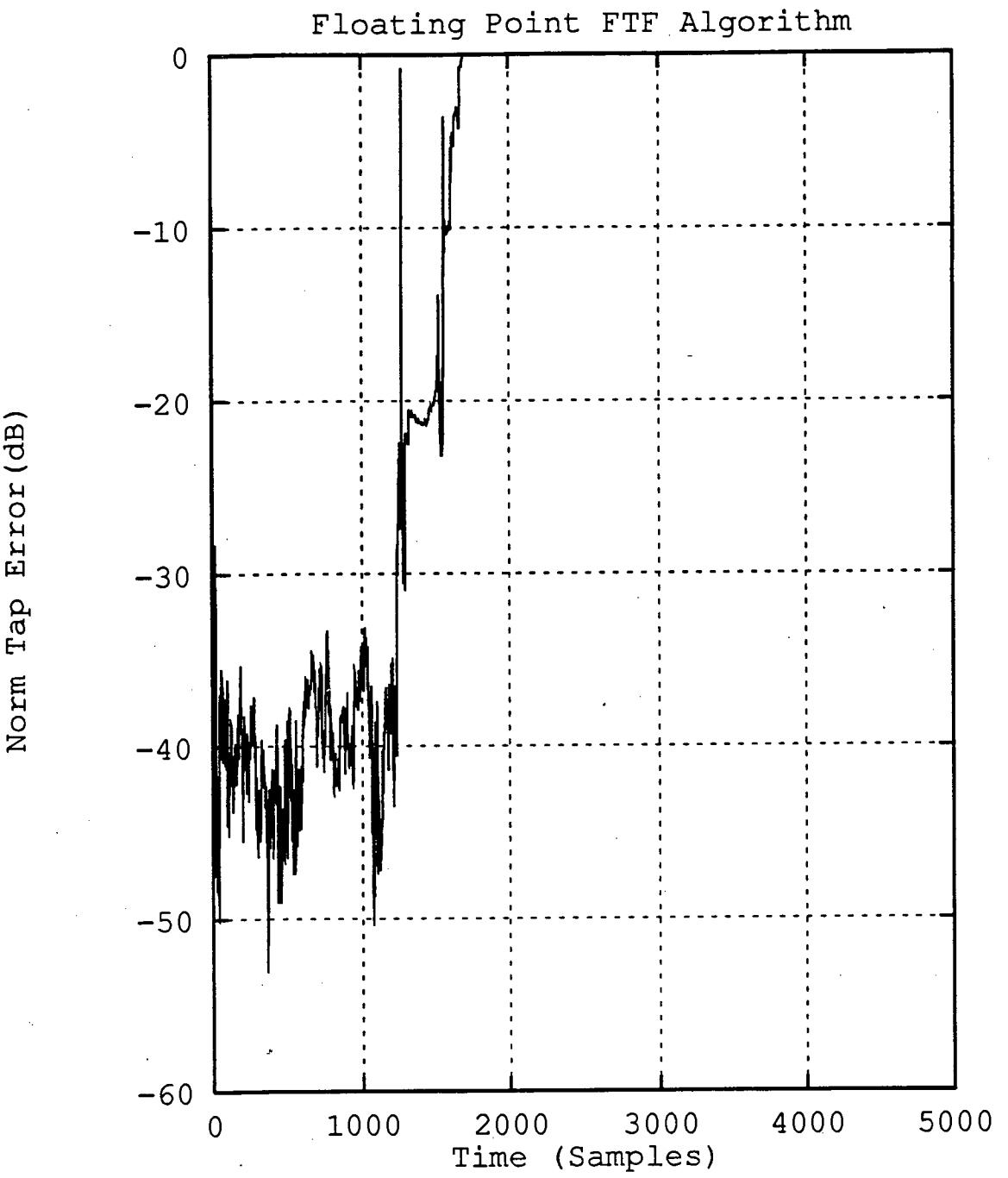


Figure 4.3 Divergence of the FTF fast RLS algorithm due to numerical instability. $\lambda = 0.98$, SNR=30dB, input autocorrelation EVR=18.7.

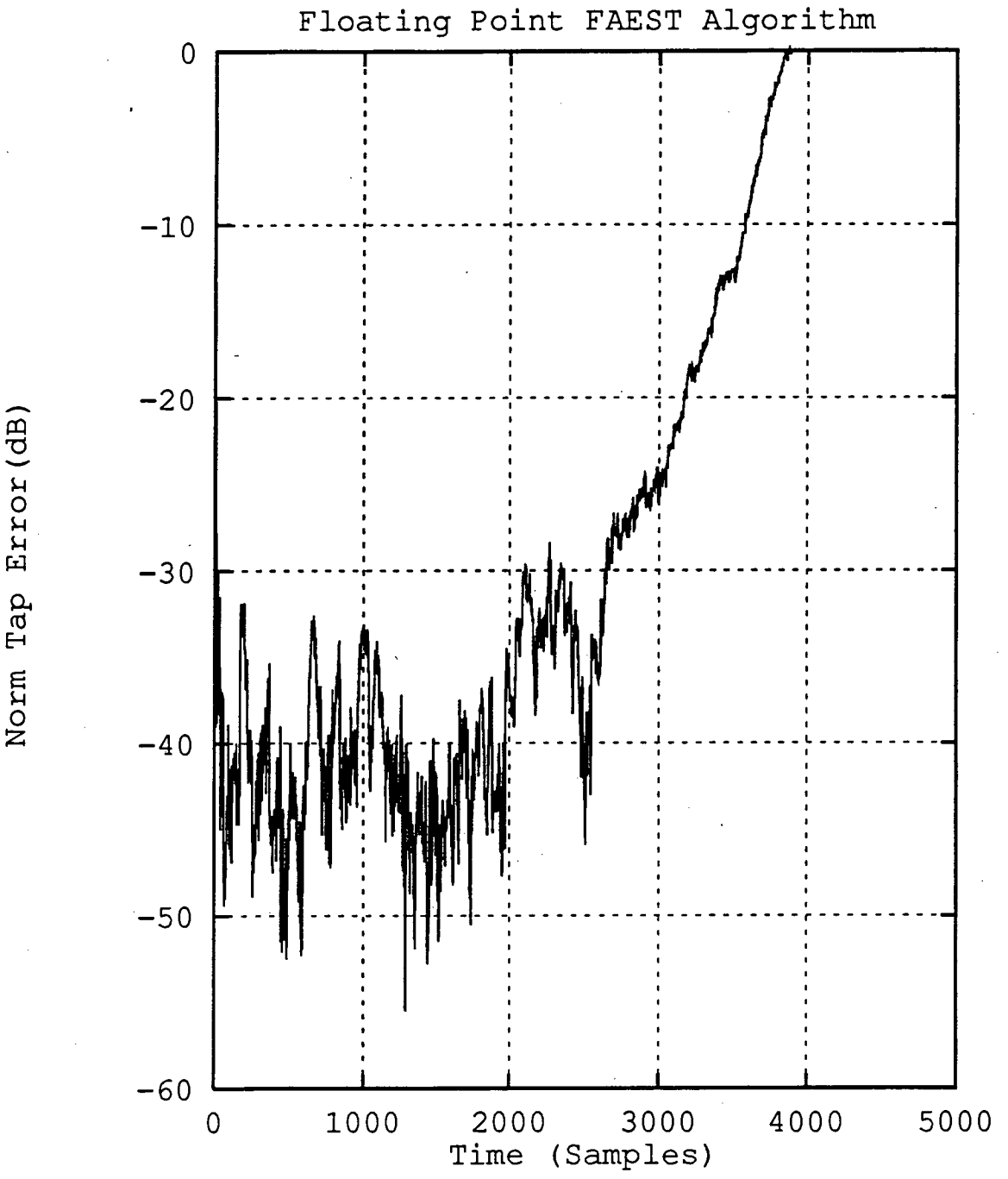


Figure 4.4 Divergence of the FAEST fast RLS algorithm due to numerical instability. $\lambda = 0.98$, SNR=30dB, input autocorrelation EVR=18.7.

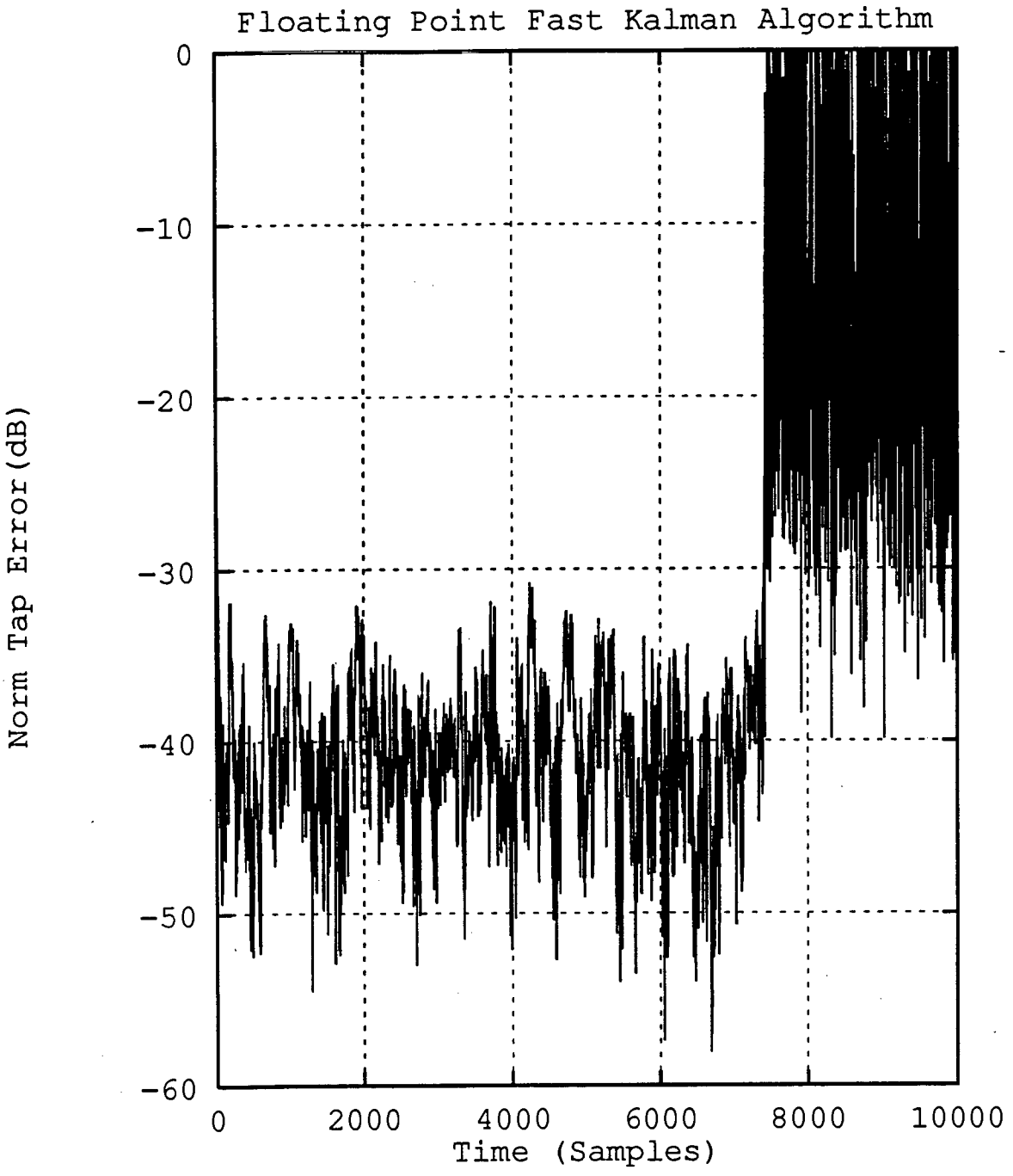


Figure 4.5 Divergence of the fast Kalman fast RLS algorithm due to numerical instability. $\lambda = 0.98$, SNR=30dB, input autocorrelation EVR=18.7.

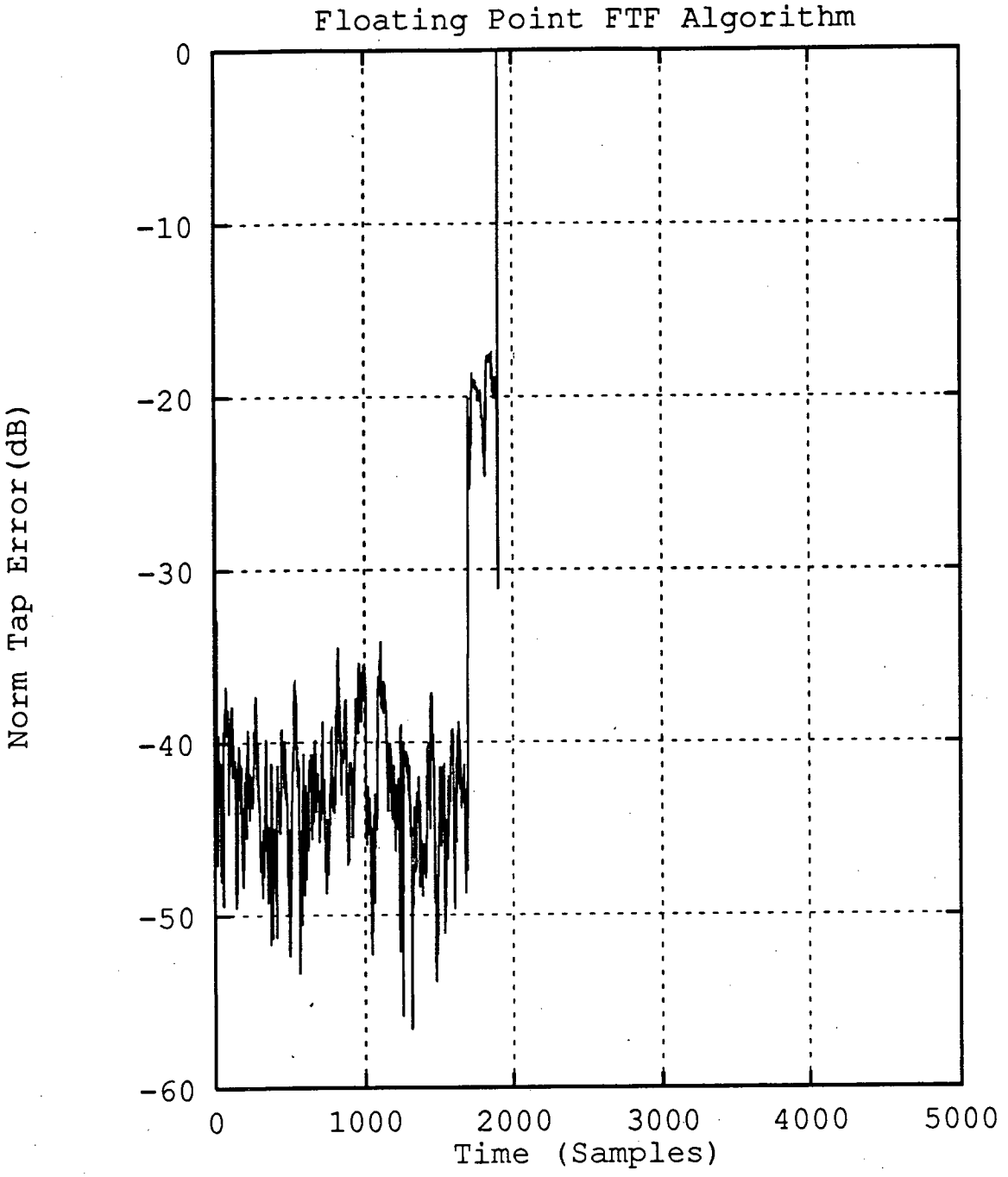


Figure 4.6 Divergence of the FTF fast RLS algorithm due to numerical instability. $\lambda = 0.98$, SNR=30dB, input autocorrelation EVR=7.3.

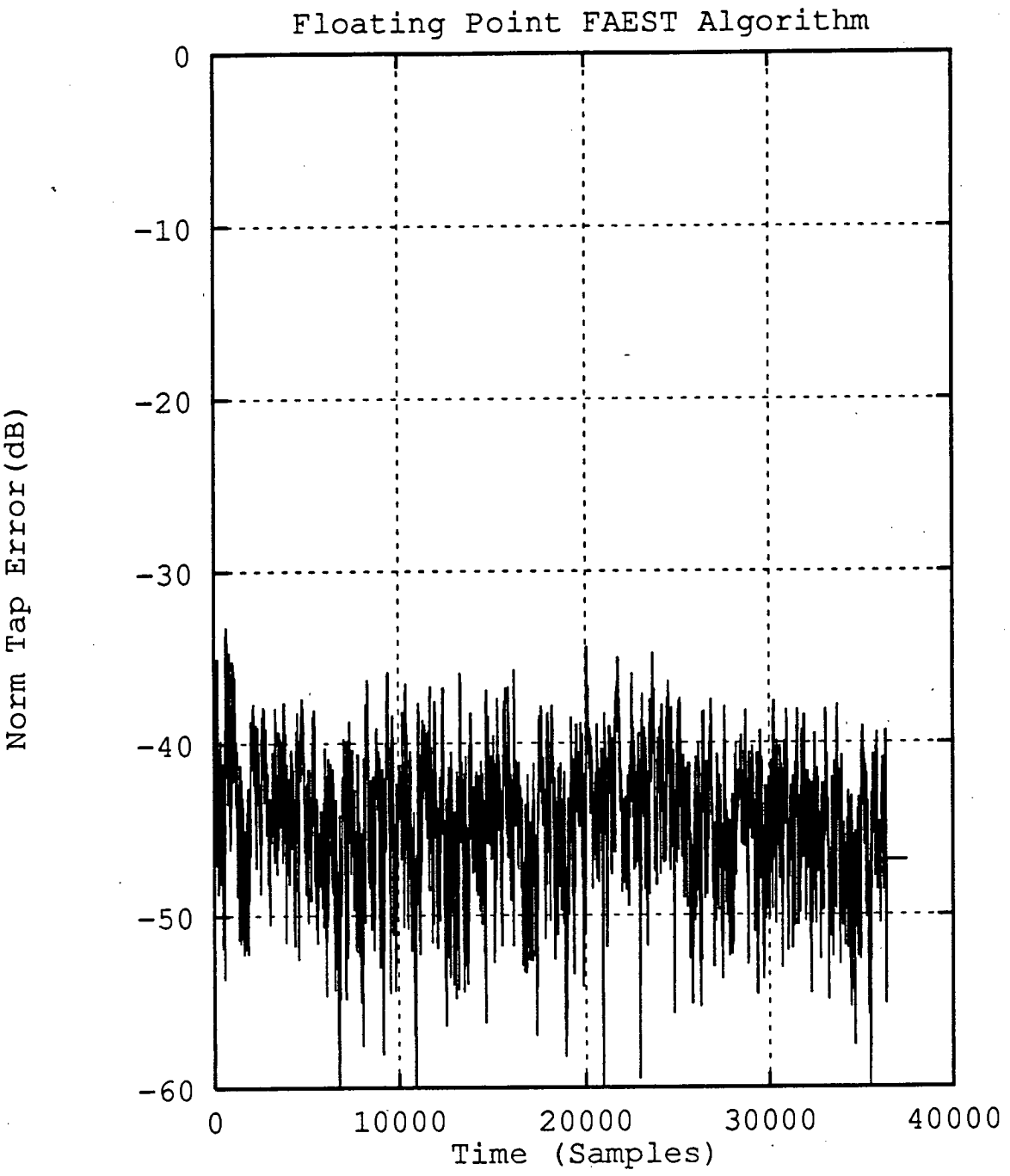


Figure 4.7 Divergence of the FAEST fast RLS algorithm due to numerical instability. $\lambda = 0.98$, SNR=30dB, input autocorrelation EVR=7.3. The algorithm first 'locks up' at a solution and then fails with a division by zero error at around 37,000 iterations.

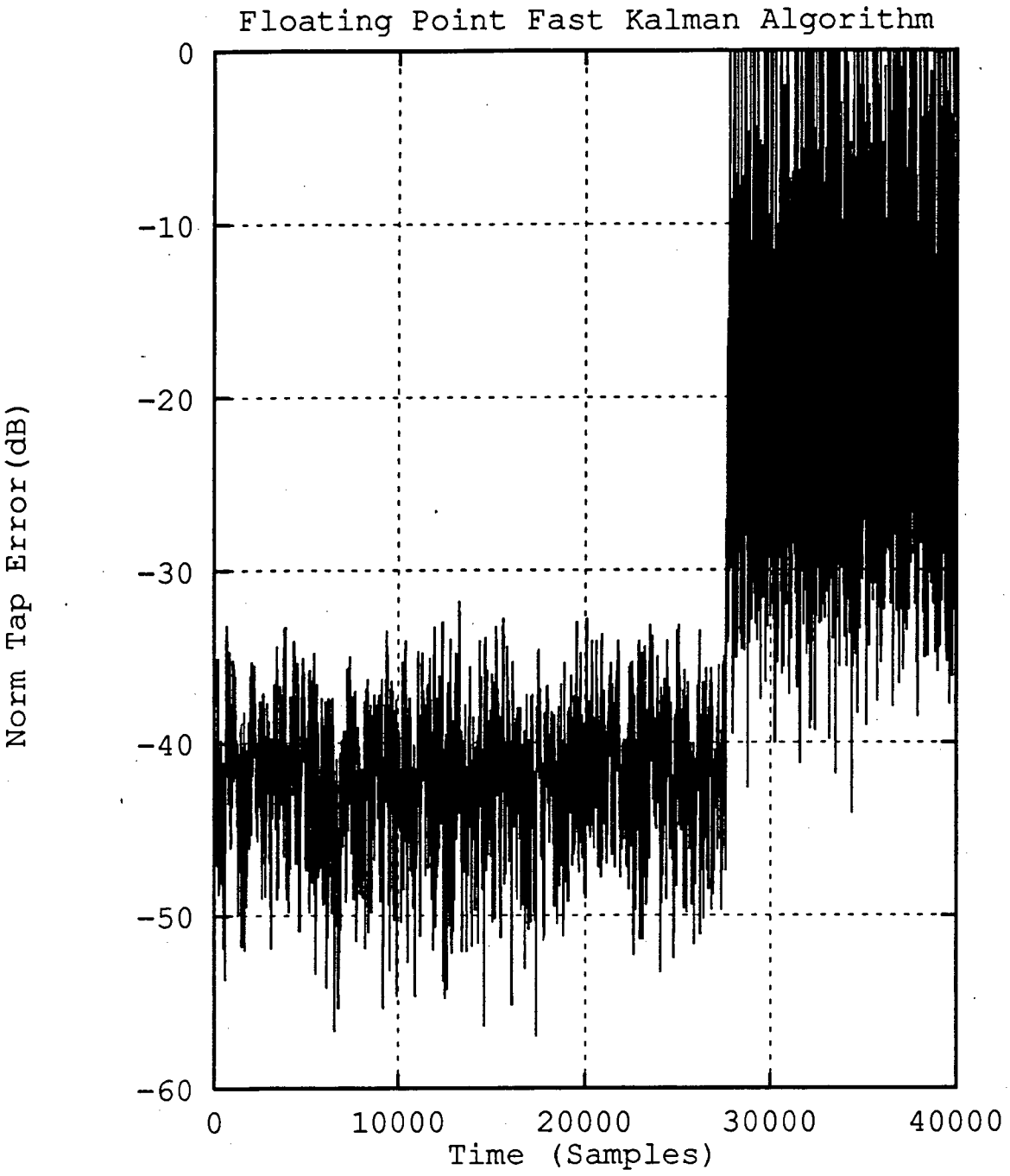


Figure 4.8 Divergence of the fast Kalman fast RLS algorithm due to numerical instability. $\lambda = 0.98$, SNR=30dB, input autocorrelation EVR=7.3.

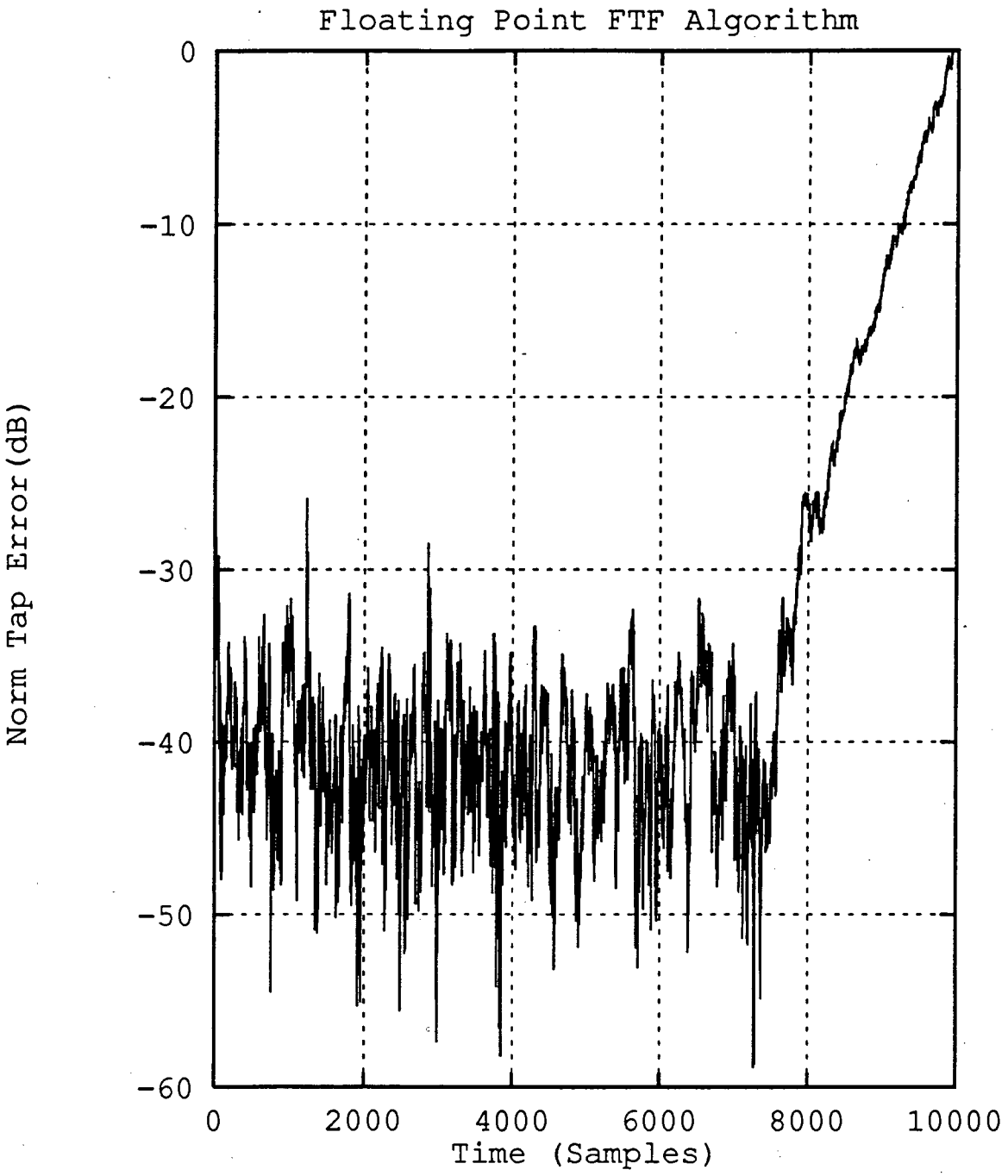


Figure 4.9 Divergence of the FTF algorithm, using a rescue procedure which involves reinitialising the algorithm if $[1+e^b(k)\dot{\gamma}'(k)\delta(k)]\leq 0$ $\lambda=0.98$, μ (reinitialisation soft constraint weight) $=100.0$, SNR=30dB, input autocorrelation EVR=18.7.

4.5. FTF Performance Using Interval Arithmetic

Figure 4.10 illustrates the operation of the interval version of the FTF algorithm. The maximum difference between the upper and lower endpoints of the coefficients of the adaptive filter, denoted by ρ has been deliberately set very large so that the endpoints can differ significantly from each other. The upper and lower endpoints of the first coefficient have been plotted in Figure 4.10 along with the optimum solution, which is for this coefficient to equal 0.9.

From the graph, it can be seen that the upper and lower endpoints of the solution are initially almost identical and both converge close to the optimum solution. As numerical errors accumulate, the two endpoints start to diverge from each other until the difference exceeds the threshold ρ at which a rescue is required. After the rescue, both the upper and lower endpoints are again moved together and they will both track the optimum solution until the next rescue is required.

Figures 4.11 and 4.12 show the performance function for the interval FTF algorithm. Figure 4.11 shows the short term performance of the algorithm and may be compared with Figure 4.2, the performance function for the conventional RLS algorithm. It can be seen that the level of performance which is attained is almost identical to that of the less computationally efficient algorithm for the duration of the simulation.

Figure 4.12 illustrates the long term performance of the FTF algorithm over one million time iterations. During this period, no evidence of divergence is indicated and the interval arithmetic rescue system performs correctly.

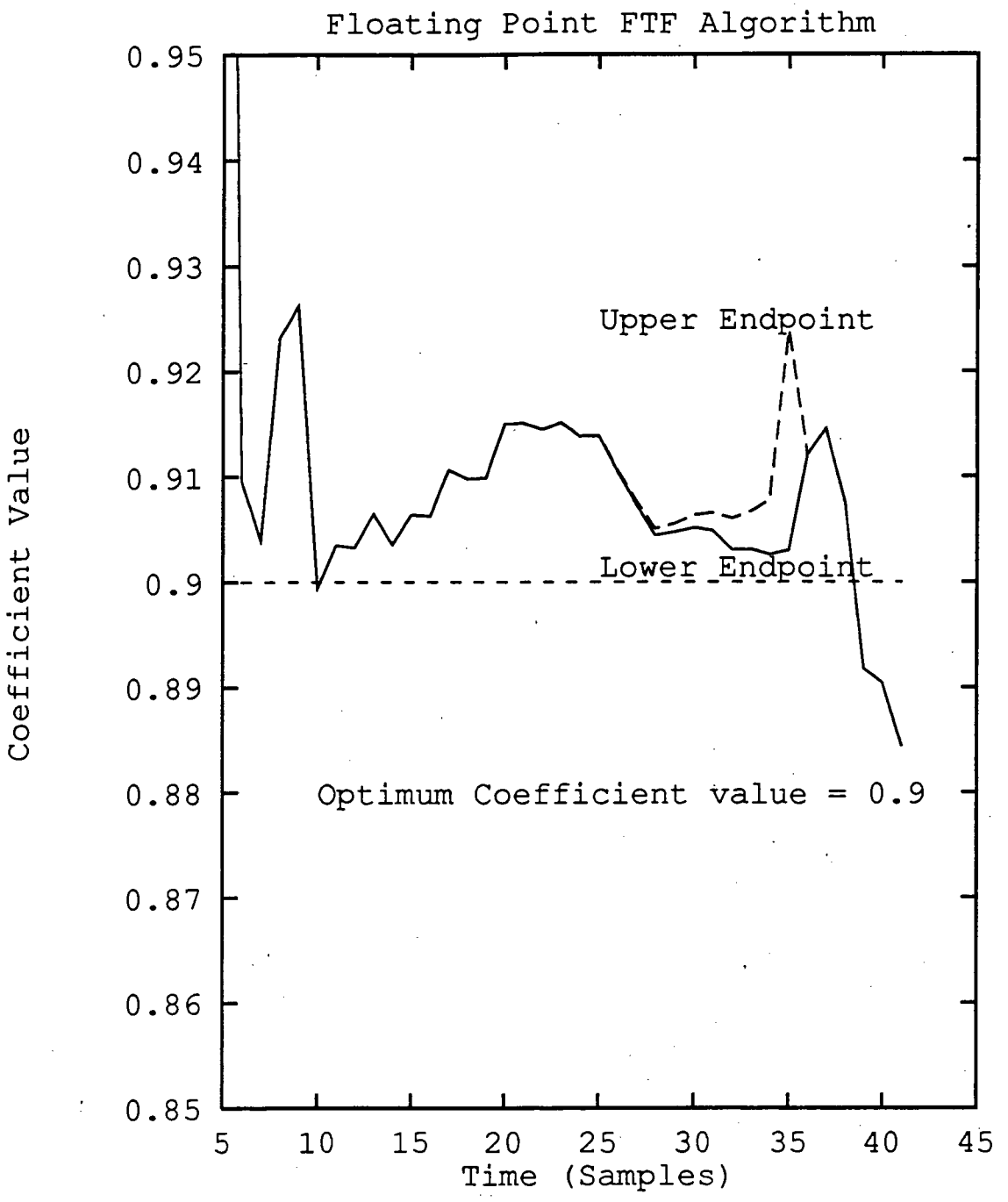


Figure 4.10 A coefficient of the solution calculated by the interval FTF algorithm. Both the upper and lower endpoints of the coefficient are plotted and the figure shows how these begin to differ from each other as numerical errors accumulate and how they are brought back together again by the rescue procedure. $\lambda = 0.98$, $\mu = 1.0$, $\rho = 0.2$, input autocorrelation EVR = 18.7, SNR = 30 dB.

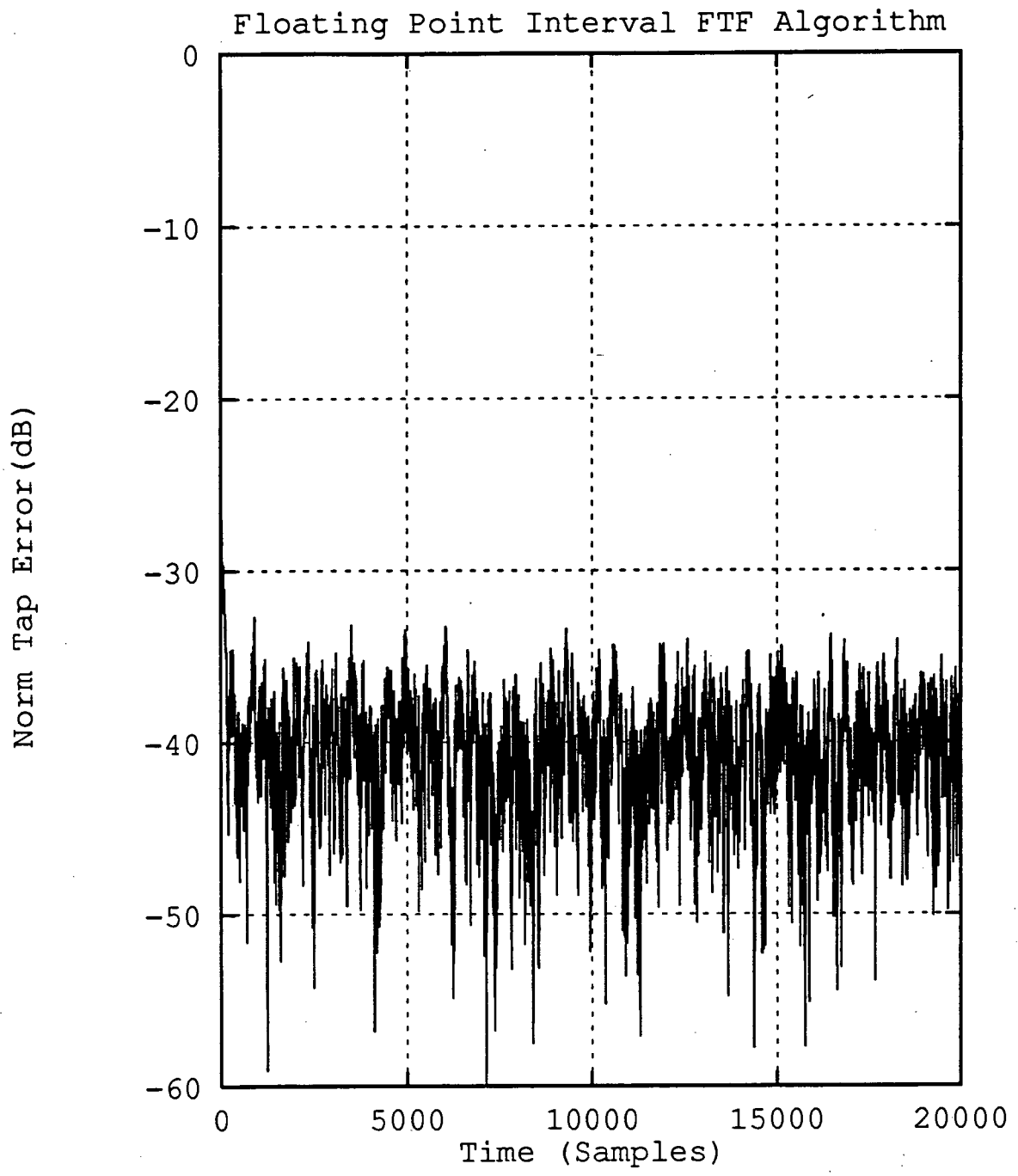


Figure 4.11 Short term performance of the interval FTF algorithm. The algorithm produces a performance level similar to that of the conventional RLS algorithm during the simulation. $\lambda = 0.98$, $\mu = 50.0$, $\rho = 0.004$, SNR=30dB, input autocorrelation EVR=18.7.

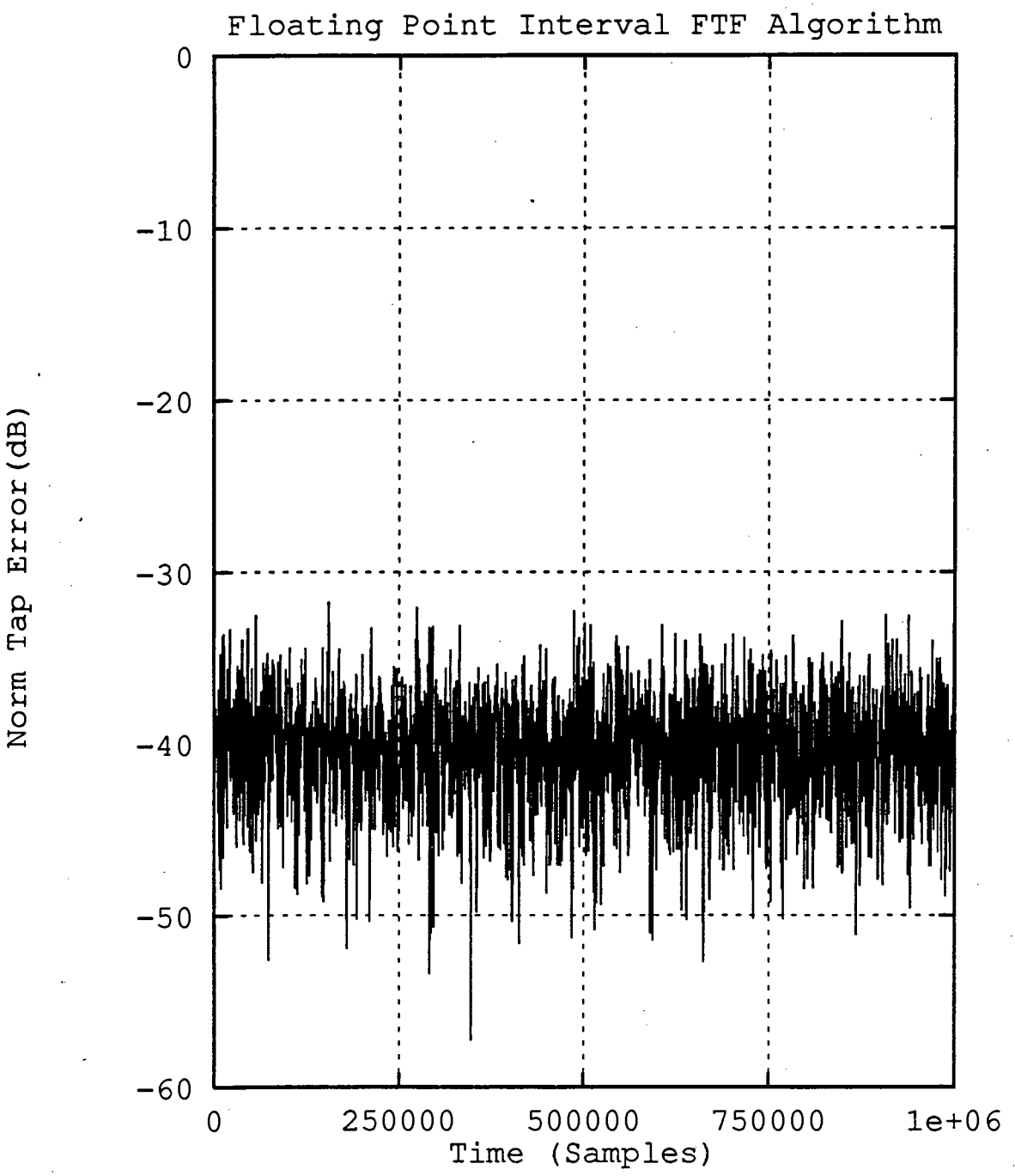


Figure 4.12 Long term performance of the interval FTF algorithm. The algorithm produces a performance level similar to that of the conventional RLS algorithm during the simulation. $\lambda = 0.98$, $\mu = 50.0$, $\rho = 0.004$, SNR=30dB, input autocorrelation EVR=18.7.

4.6. Fixed Point Implementation of the FTF Algorithm

When implementing a fixed point version of any algorithm[124], there are a number of important considerations which will affect the performance obtained. The difficulty in using fixed point arithmetic is the limited dynamic range available. Variables must be represented in such a way that they can be stored to a reasonable level of accuracy, but at the same time, care must be taken to ensure that overflows of the variables are sufficiently unlikely to occur. There is therefore a tradeoff to be made between the accuracy to which a number is represented and the probability of overflow errors.

The problem is to determine for each variable where the binary point should be fixed. The process by which this was done was first to assess the likely range of the variable, using the floating point simulation. A considerable safety margin must then be left, as the maximum values for each quantity may differ considerably between different runs of the same simulation and they are dependent upon the exact data sequence. Having assessed the likely range of each variable, a fixed point simulation can then be developed and the fixed point scale factors can then be further refined.

The ranges and positions of the binary point for each variable in the fixed point implementation of the FTF algorithm are listed in Table 4.2.

Variable	Fixed Point Position	Range	Precision
$\underline{a}(k)$	10	-32 to 31.999023	0.000997
$\underline{X}(k)$	15	-1 to 0.999969	0.000031
$\tilde{\underline{c}}(k),\tilde{\underline{c}}'(k)$	3	-4096 to 4095.875	0.125
$\underline{b}(k)$	15	-1 to 0.999969	0.000031
$\underline{H}(k)$	15	-1 to 0.999969	0.000031
$e^f(k)$	14	-2 to 1.999938	0.000061
$\epsilon^f(k)$	15	-1 to 0.999969	0.000031
$\alpha^f(k)$	15	-1 to 0.999969	0.000031
$\gamma(k),\gamma'(k)$	15	-1 to 0.999969	0.000031
rescue	14	-2 to 1.999938	0.000061
$e^b(k)$	15	-1 to 0.9999690	0.000031
$\epsilon^b(k)$	19	-0.0625 to 0.062498	1.907×10^{-6}
$\alpha^b(k)$	15	-1 to 0.999969	0.000031
$e(k)$	15	-1 to 0.999969	0.000031
$\epsilon(k)$	15	-1 to 0.999969	0.000031

Table 4.2:Scaling used for fixed point FTF

The simulation software enabled two different overflow characteristics to be used. The roll-over characteristic is the simplest, as overflows are simply ignored. This means that it is likely that if overflow occurs in calculating a result which should be positive, a negative result will probably be obtained and vice versa. Hence, the errors which occur using roll-over are very large indeed. The saturation characteristic reduces the errors which occur in the event of overflow. If a result is calculated which exceeds the largest positive representable number, then the result is replaced by the largest positive representable number and similarly, negative overflows are replaced by the largest representable negative number. After the scale factors were

correctly chosen, the overflow mode which was used was found to make no difference to system performance, indicating that overflows rarely occurred.

The performance of the fixed point FTF algorithm is shown by Figures 4.13 and 4.14. Figure 4.13 shows the performance without any rescues being performed while Figure 4.14 shows the performance when the rescue method described in section 4.5 is used.

As would be expected, the 16 bit fixed point implementation has severe problems with numerical instability. After around 500 time iterations, the algorithm diverges. Moreover, the rescue method which was used to improve the stability of the floating point algorithm gives no useful improvement when applied to the fixed point implementation.

4.7. Fixed Point Interval FTF Performance

Interval methods may be applied to the fixed point implementation of the FTF algorithm in a similar way to the floating point algorithm. The maximum value which may be used for the parameter μ is limited, however, as the variables $\alpha^f(-1)$ and $\alpha^b(-1)$ must both be set to the same order of magnitude as μ at reinitialisation. For the scale factors used, this limits the maximum acceptable value of μ to 1.0.

The results from the short and long term simulations of the fixed point interval FTF algorithm are presented in Figures 4.15 and 4.16. Whilst the performance is not quite as good as the floating point conventional RLS solution, it is nevertheless impressive for such a highly limited precision implementation.

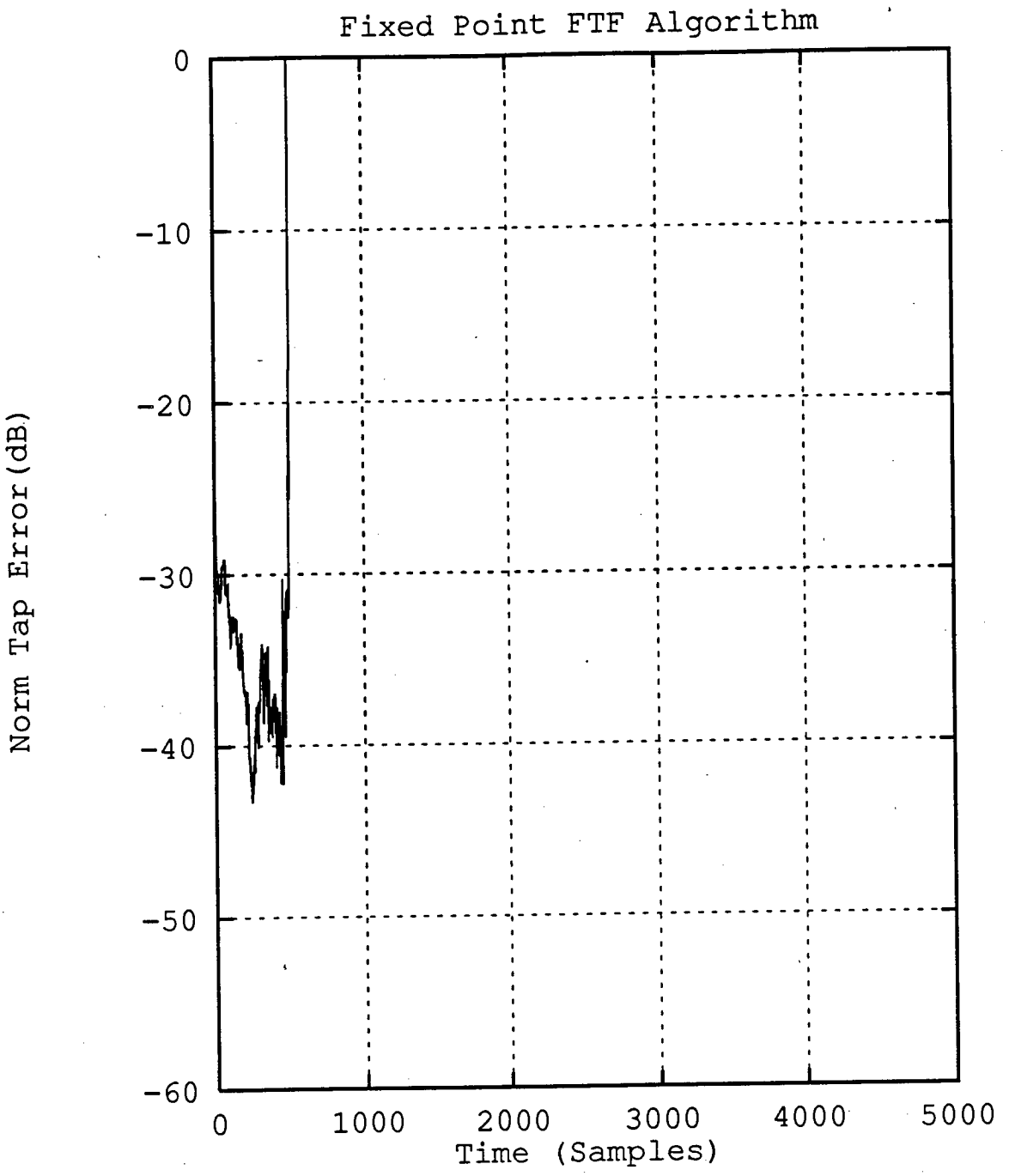


Figure 4.13 Performance of a 16/32 bit fixed point implementation of the FTF algorithm. $\lambda = 0.98$, SNR=30dB, input autocorrelation EVR=18.7.

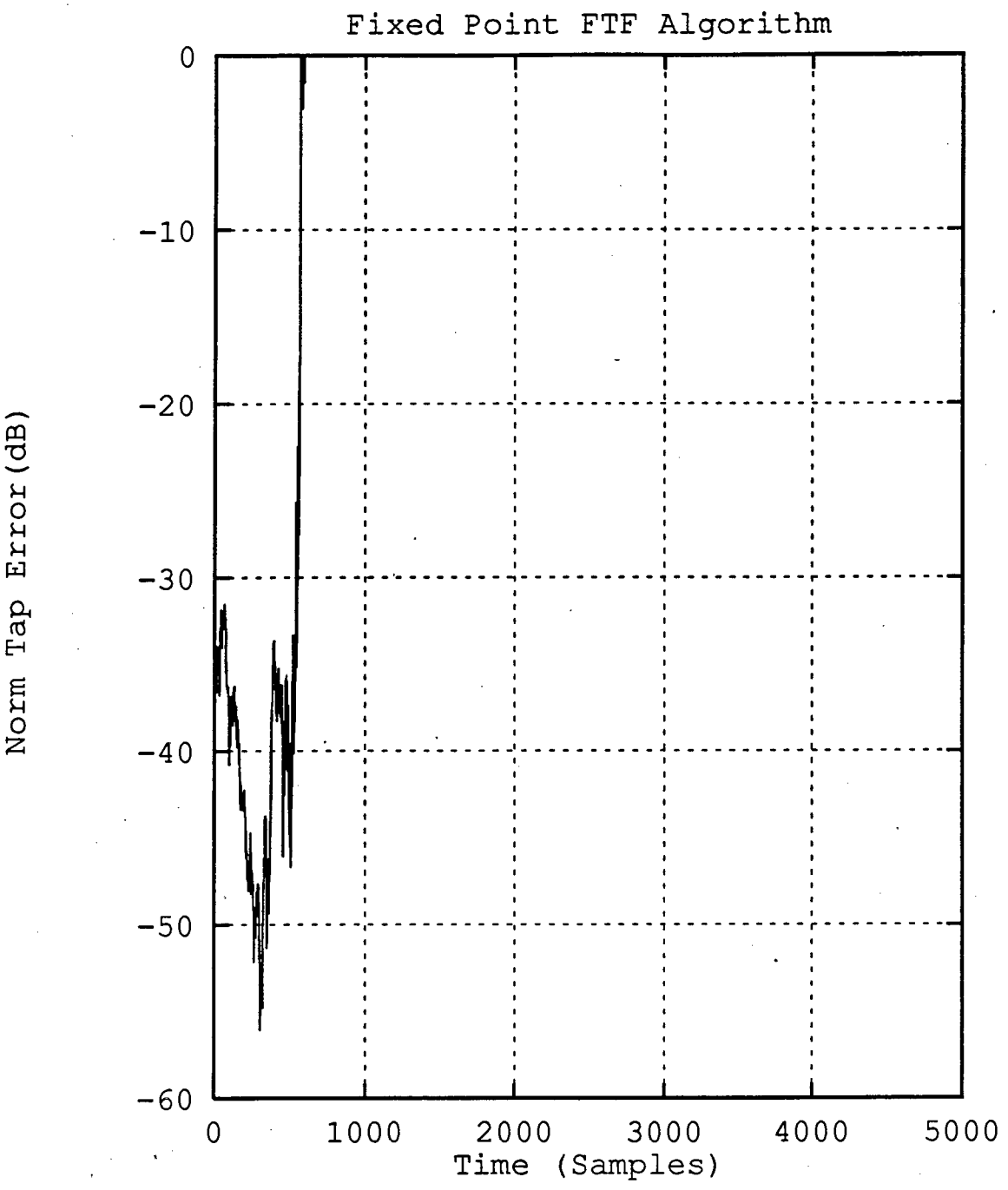


Figure 4.14 Performance of a 16/32 bit fixed point implementation of the FTF algorithm using rescue method of section 4.5 $\lambda = 0.98$, $\mu = 0.25$, $\rho = 0.04$, SNR=30dB, input autocorrelation EVR=18.7.

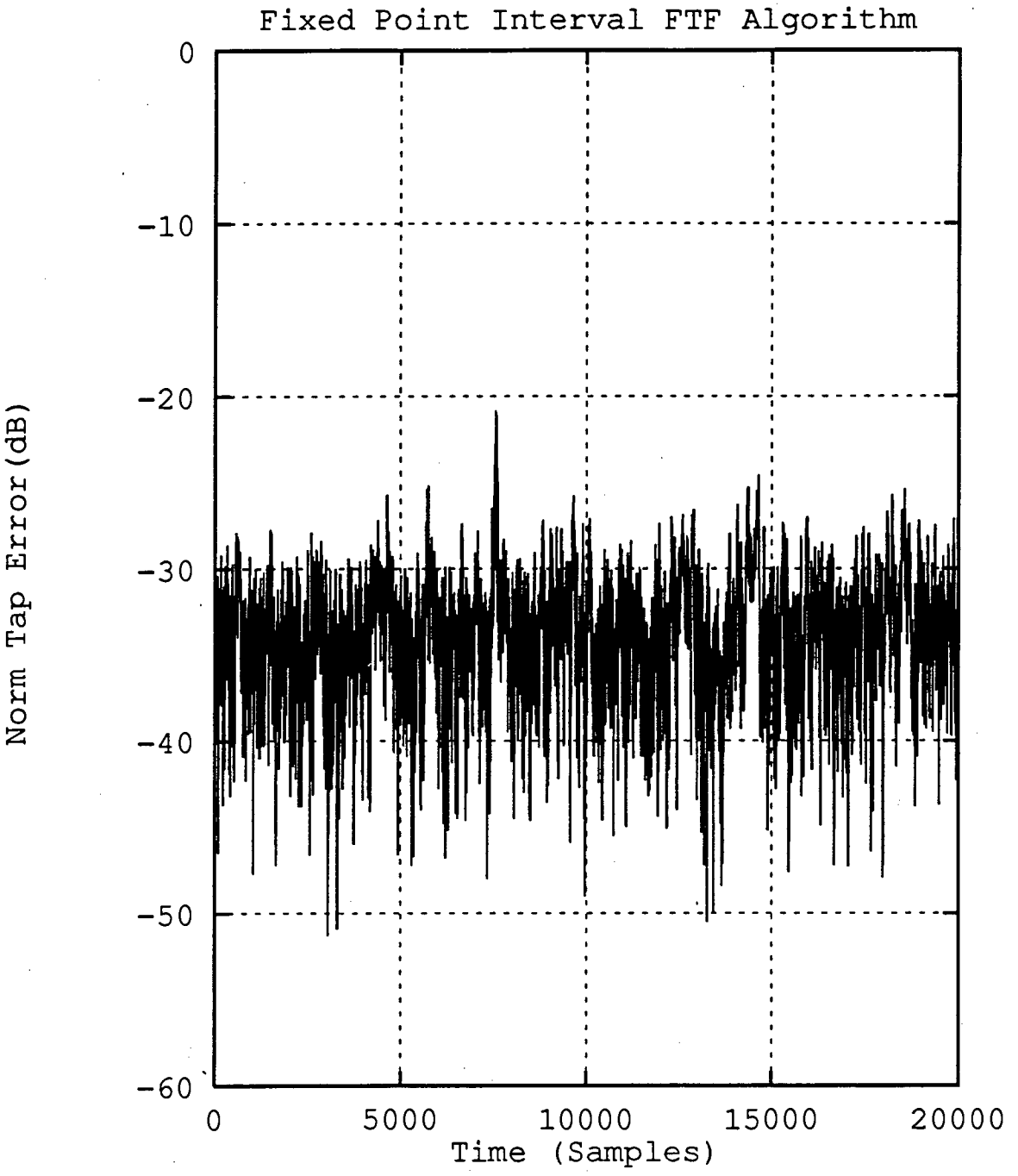


Figure 4.15 Short term performance of a 16/32 bit fixed point implementation of the interval FTF algorithm. $\lambda = 0.98$, $\mu = 0.25$, $\rho = 0.04$, SNR=30dB, input autocorrelation EVR=18.7.

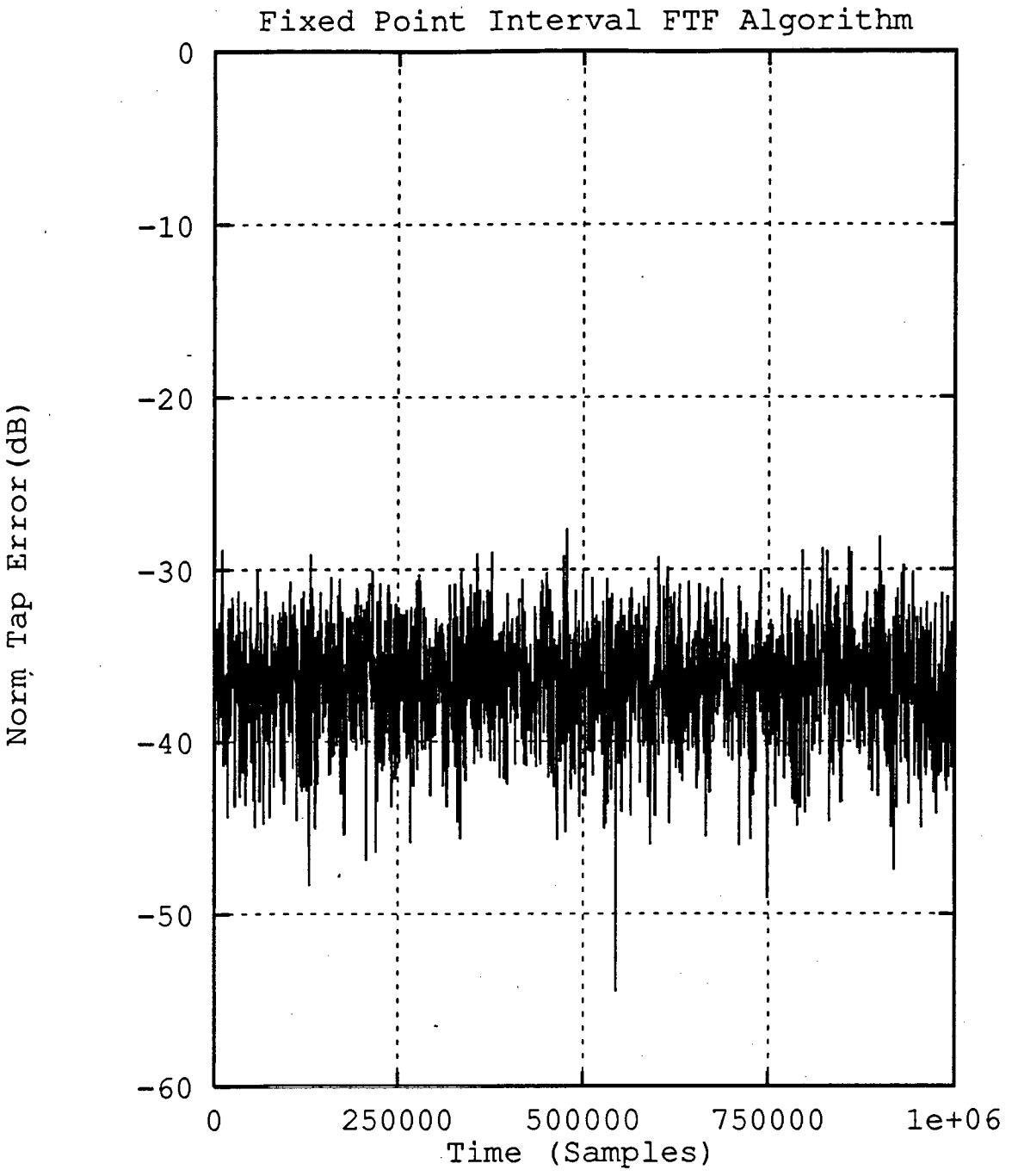


Figure 4.16 Long term performance of a 16/32 bit fixed point implementation of the interval FTF algorithm. $\lambda = 0.98$, $\mu = 0.25$, $\rho = 0.04$, SNR=30dB, input autocorrelation EVR=18.7

4.8. Application of Interval Algorithms to Stationary and Non-Stationary Equalisation

The aims of the simulations in this section are twofold. Firstly, they are intended to demonstrate that the operation of the interval methods is not specific to adaptive system identification and that the interval algorithms may be equally successfully applied to other adaptive filtering problems. Secondly, the results compare the tracking performance of the interval algorithms with that of the conventional RLS algorithm. Some impairment in tracking performance is possible when using the interval algorithms, due to the regular reinitialisations which are being performed and one of the aims in this section is to examine how significant the degradation in tracking performance is.

The adaptive filtering application which is being considered is that of adaptive equalisation for digital communications. The digital communications channel which will be simulated is the HF channel[48, 122], a model[120, 121, 123] of which is shown in Figure 4.17. The channel is represented by a three tap finite impulse response (FIR) filter, the output of which is subject to interference by Gaussian noise. The coefficients of the channel are generated from other Gaussian noise sources, which are passed through low pass filters, so that they have slowly time varying random values.

The physical process which is being modelled by this channel is that of multi-path interference[48, 49] illustrated in Figure 4.18. Signals arrive at the receiver by a number of different paths. As the lengths of the paths are different, the signals are subject to different time delays between the transmitter and receiver and so interference occurs.

Figure 4.19 shows the configuration of an adaptive equaliser. The output from the channel is passed into the adaptive filter input. It is assumed that the transmitted signal is available at the receiver and this is passed into the desired response input. This signal may be generated at the receiver initially by transmitting a known training sequence and after the adaptive filter has converged, by using the actual output from the equaliser, which may be passed through a decision device. This mode of operation is known as decision-directed mode[47-49]. For the purposes of simulation, the effects of decision directed operation were not considered and it was assumed that the transmitted signal was known exactly at the receiver.

A two level baseband signal was simulated. This signal was generated from a pseudo-random source and either had the value -1 or 1 with equal probability. A real communications system would include a modulator at the transmitter and a demodulator at the receiver, but assuming that the modulation process, the channel and the demodulation process are all linear, then the results obtained from a simulation of the baseband system are identical.

The performance measure which was used for all simulations was the probability of error. The output from the adaptive equaliser was passed through a decision process which gave an output of +1 for all positive inputs and an output of -1 for all negative inputs. The number of occasions on which the output from this decision device differed from the transmitted bit was counted over many iterations of the algorithm. A 5 tap equaliser was used with a delay of two bits at the desired response input to enable non-minimum phase channels to be equalised. A 10 bit training period was assumed to be available and errors were only counted after this training sequence was completed.

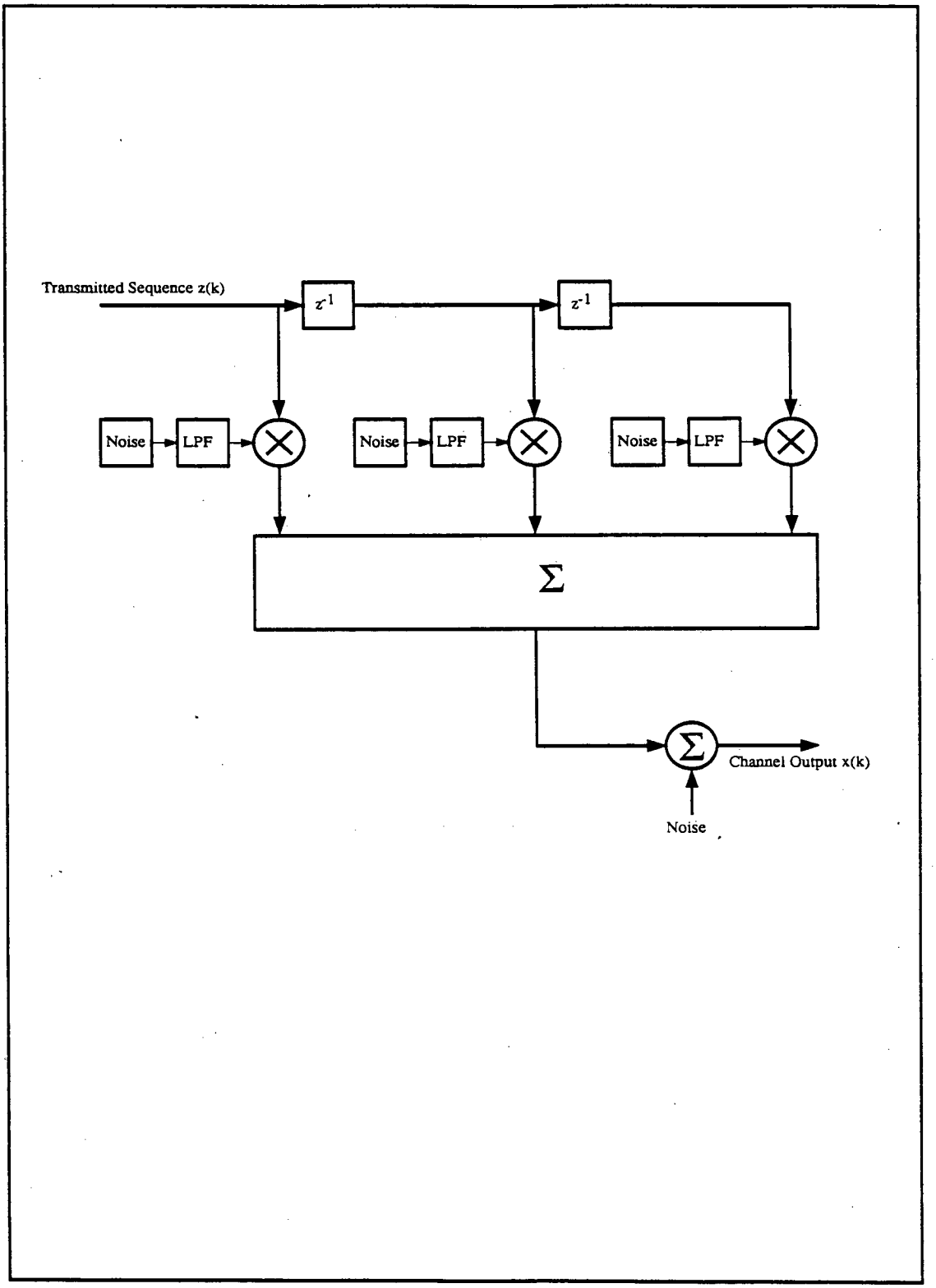


Figure 4.17 Model of a fading HF channel

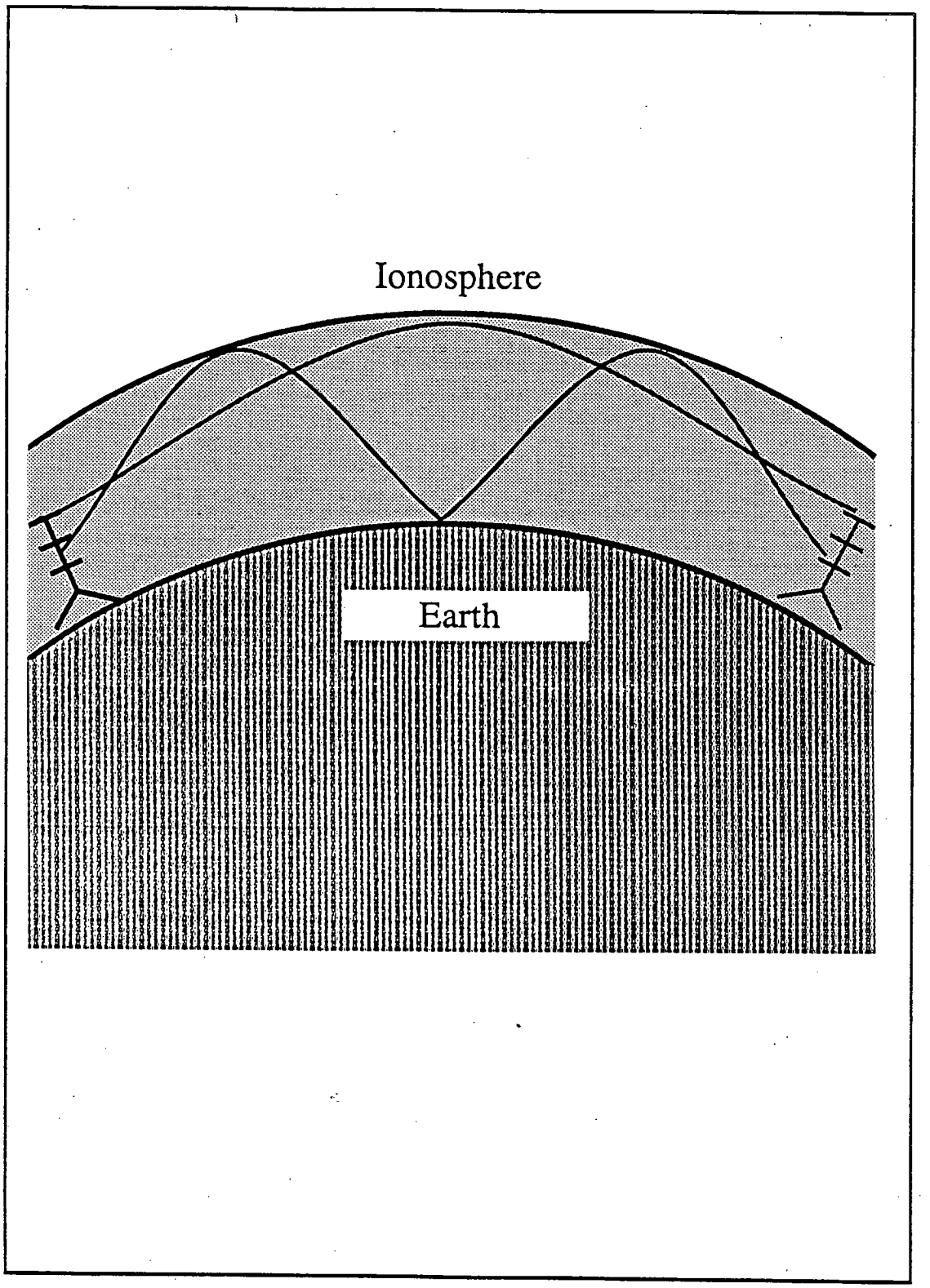
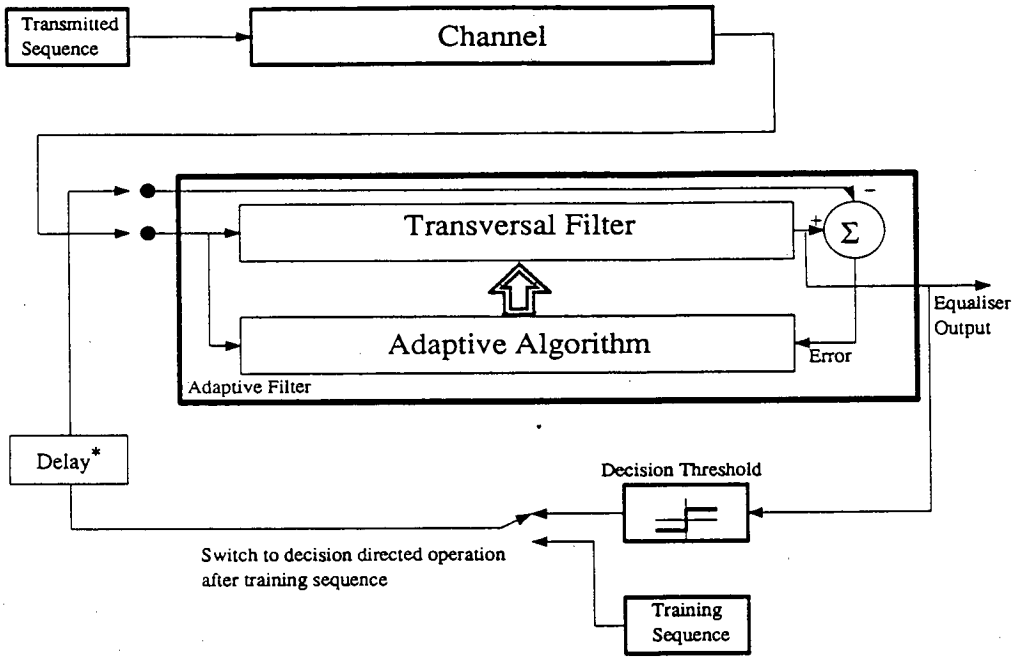


Figure 4.18 The time varying multi-path interference which is being modelled by the channel of figure 4.17



*Required if the channel has zeros outside the unit circle (non-minimum phase conditions)

Figure 4.19 Block Diagram of an Adaptive Equaliser

4.8.1. Performance for a Stationary Channel

Figure 4.20 shows the bit error rates which were obtained using the conventional RLS algorithm and the interval FAEST algorithm as the signal to noise ratio is varied between 0dB and 10dB. The channel coefficients remained fixed during this simulation and the channel impulse response was $h_{chan}(z)=1.0+0.5z^{-1}$.

The bit error rates obtained for the two algorithms are nearly identical, demonstrating that an interval algorithm can offer similar performance to the conventional RLS algorithm within the context of equalisation as well as system identification. The third curve is an optimum lower bound, which is the theoretical probability of error when no multi-path distortion occurs and the only source of interference is additive Gaussian noise.

It can be seen that the performance of both the adaptive equalisers falls far short of the optimum bound. This is due to the limitations of the linear adaptive equaliser, which can only form a linear decision region in the signal space. A number of other structures can offer improved performance, but these are not considered here.

4.8.2. Performance for a Fading Channel

In all of the simulations presented so far, the optimum solution has not varied with time and the tracking performances of the various algorithms have not been compared. In this simulation, a fading channel represented by the model of Figure 4.17 was used. The fade rate was achieved by setting the bandwidth of the low-pass filters to be 0.00016 × bit frequency, corresponding to a moderately severe fading channel.

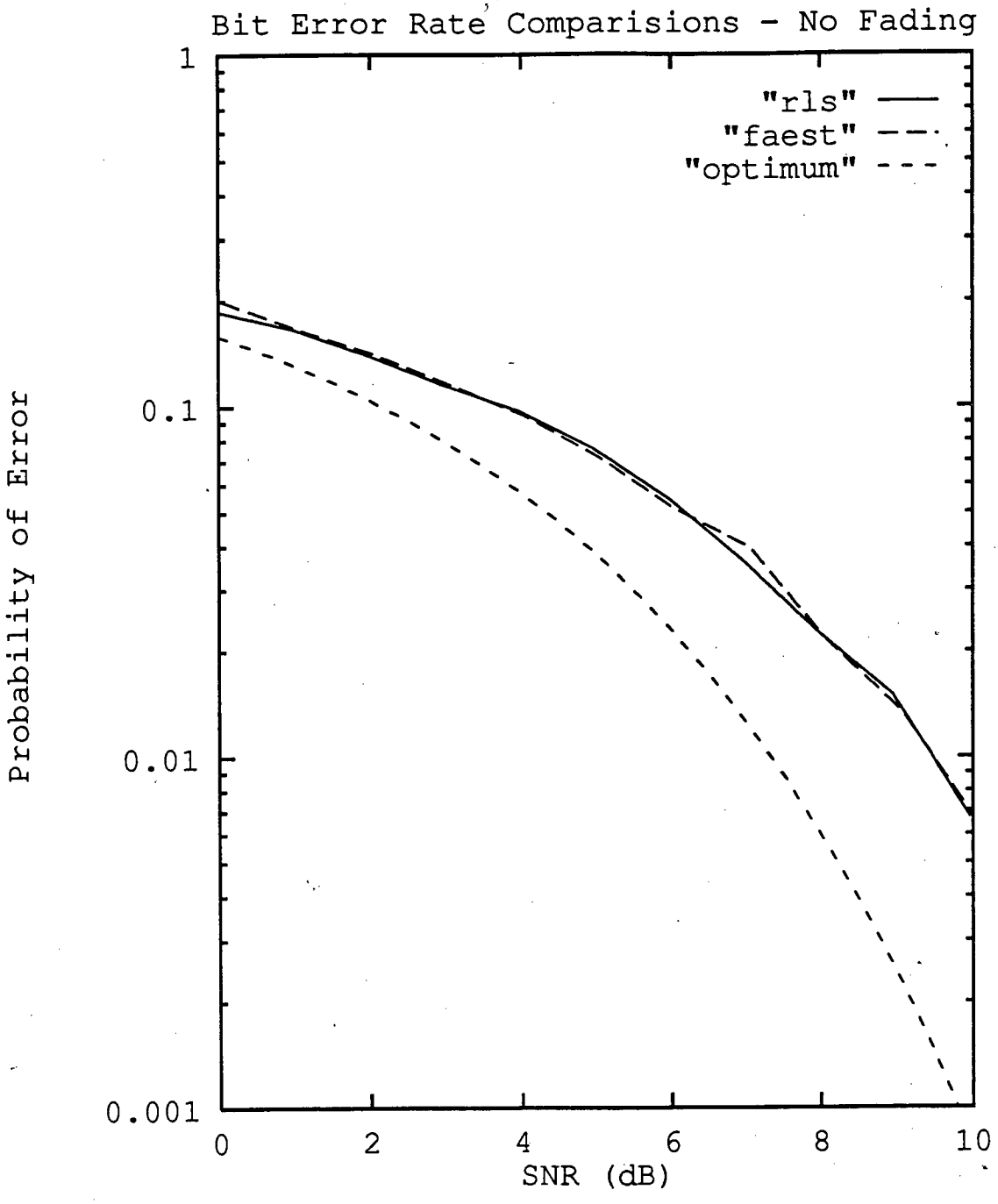


Figure 4.20 Comparison of probability of error for the conventional RLS adaptive equaliser and the interval FAEST adaptive equaliser. The channel being equalised had the impulse response $1.0+0.5z^{-1}$. For the RLS algorithm, $\lambda = 0.98$ For the FAEST algorithm, $\lambda = 0.98$, $\rho = 0.02$ and $\mu = 50.0$

The results for the non-stationary simulation are shown in Figure 4.21. The optimum solution has been plotted along with the performance of the conventional RLS algorithm and the performance of the FAEST algorithm for $\mu=15.0$ and $\mu=50.0$.

It is clear from the results that the FAEST algorithm offers comparable performance to the conventional RLS algorithm with μ set to 15.0, but that there is a slight degradation in performance when the simulation is performed with $\mu=50.0$. This is due to the impairment in tracking caused by setting the reinitialisation soft-constraint to have too much influence. The results for $\mu=15.0$ indicate, however, that tracking performance which is as good as that of the RLS algorithm may be obtained by choosing the reinitialisation parameters correctly.

In this simulation, 100,000 bits were required to provide an accurate estimate of the bit error rate and so, if there had been any problems due to the numerical instability of the FAEST algorithm over this fairly large number of iterations, this would have resulted in a significantly higher bit error rate. This confirms the numerical robustness shown in the system identification simulations.

4.9. Conclusions

A number of important results relating to the performance of the fast RLS algorithms have been suggested by computer simulation.

The numerical instability and divergence of the fast Kalman, FAEST and FTF algorithms have been demonstrated experimentally. The number of iterations before the onset of instability has been demonstrated to depend upon the algorithm being used, the input autocorrelation eigenvalue ratio and the accuracy of the arithmetic being

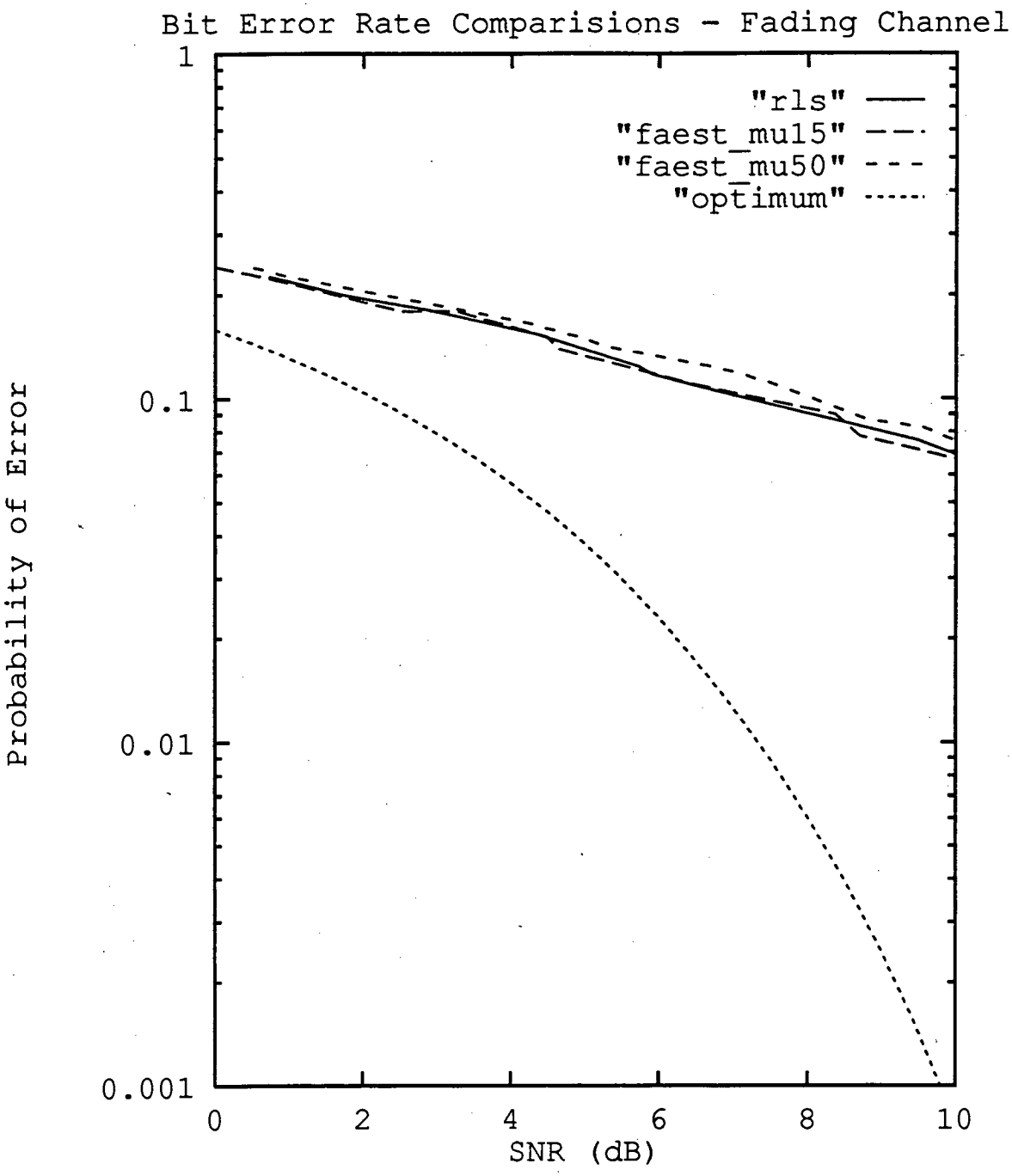


Figure 4.21 Comparison of probability of error for the conventional RLS adaptive equaliser and the interval FAEST adaptive equaliser, for a fading channel. For the RLS algorithm, $\lambda = 0.98$ For the FAEST algorithm, $\lambda = 0.98$, $\rho = 0.02$ and $\mu = 15.0$ or 50.0

used.

Many of the results have related to comparing the interval versions of the fast RLS algorithms with the performance of the conventional RLS algorithm. In all simulations, the interval fast RLS algorithms have been numerically stable and have not exhibited the divergence of their non-interval counterparts. The performance of the stable interval algorithms has been shown to be almost identical to that of less efficient least squares techniques in a number of different simulations involving system identification and HF channel equalisation. The tracking performance of the interval algorithms is comparable to a least squares algorithm which runs continuously.

The performance of the fixed point interval FTF algorithm is of particular importance, as the complexity associated with implementing a fixed point algorithm in hardware is considerably less than that of implementing the same algorithm using floating point arithmetic. The next chapter will consider the implementation of this algorithm on a TMS320C25 digital signal processor. An equaliser similar to that of section 4.8 will be developed, capable of operating at 1200 bits per second. The implementation of faster equalisers will also be considered.

4.10. Frequency of Reinitialisation

In the simulations of this chapter, the interval arithmetic rescue procedure reinitialises the algorithm after approximately every 100 iterations using 64 bit floating point arithmetic, and approximately every 50 iterations using 16 bit fixed point arithmetic. Comparing this with the performance of the unstabilised algorithm (shown in figure 4.3 and figure 4.13), it is apparent that about 10 rescues are performed by the stabilised algorithm in the time that the unstabilised algorithm takes to diverge.

5 Interval Algorithms - Hardware Implementation

5.1. Introduction

Having demonstrated the stable performance of fixed point versions of the interval fast RLS algorithms in software simulations, the next step is to attempt to implement them in a real time hardware system. There are two important reasons for doing this. Firstly, the hardware implementation may be used to confirm the validity of the simulations and to check that there are no factors which were not taken into account during simulations prohibiting the use of the algorithms in practice. Secondly, the hardware implementation provides important information on the speed of operation of a real-time system.

The approach has been to implement the algorithms on a digital signal processor (DSP)[125]. Many of the major semiconductor manufacturers now make DSPs[90-96] which are suitable for implementing the high speed, numerically intensive operations often required to perform signal processing in real time. The many different devices which are now available all have different architectures[126] and instruction sets, but they share a number of common features such as hardware multipliers, rapid multiply and accumulate instructions and separate program and data memory spaces, all of which make them more suitable than a general purpose microprocessor for signal processing applications.

The processor which was chosen for the hardware implementation was the Texas Instruments TMS320C25[90, 95]. This is a second generation device which represents the middle of the range in currently available DSP technology. It uses 16 bit fixed point arithmetic with a 32 bit long accumulator and offers a 100ns instruction cycle time. More sophisticated processors are now available which offer greater speed, floating point arithmetic and a number of other features, but the system which was developed seeks to demonstrate an implementation of the algorithm using the minimum hardware requirements.

The chapter will begin by discussing the implementation of interval arithmetic and the interval FTF algorithm on the TMS320C25. The configuration and circuitry used to generate test signals will then be described and results will be presented for the implementation. The results will show that the TMS320C25 is suitable for implementing medium length ($N=5$) adaptive filters at data rates of up to 1200 bits/s. More rapid implementations would require the use of either a more powerful processor, or even an array of processors performing parallel computations. Alternatively, a dedicated silicon device could be fabricated which would enable operation at high speed. This option is considered in more detail in chapter 6.

5.2. Implementing the Algorithm on a TMS320C25

Due to the high organisational complexity of the FTF algorithm, a very structured approach is required to implement the algorithm successfully using TMS320C25 assembly language. The problem may be subdivided into two areas. The first is to develop a set of assembler macros which enable interval arithmetic to be performed. The second area is to develop a program which makes use of the interval macros to perform the computations of the FTF algorithm and so implement a fixed point interval arithmetic version.

The benefit of this approach is that the interval macros can be tested extensively before the FTF algorithm is developed, allowing many of the errors in the assembly language program to be isolated at an early stage.

Figure 5.1 shows a block diagram of the TMS320C25 processor board and associated hardware. This board was built as a final year honours project[127] in the Department of Electrical Engineering at the University of Edinburgh. The circuit diagram for this board appears in Appendix D. Real signals can be passed into the board using two analogue to digital converters (ADCs), which may be operated at sampling rates of up to 100kHz. A sample clock input is also available to determine the exact time at which conversion will begin. Output from the board is performed by a single digital to analogue converter (DAC). The TMS320C25 board operates using a 20MHz crystal and one wait state for memory access, which gives an operating speed of approximately ¼ of the maximum available using this processor. Program development was done using a personal computer which was connected to the the TMS320 board by an RS-232 serial link and which provides a number of important facilities including a TMS320C25 macro assembler, a TMS320C25 linker, file format conversion, file storage and a terminal emulator for use when debugging programs running on the TMS320C25.

5.2.1. Macros to Perform Interval Arithmetic on a TMS320C25

The macros to implement interval arithmetic are listed in appendix C. They are divided into three subsections - those for performing scalar interval arithmetic, those for performing vector interval arithmetic and those for performing system operations such as reading input values from ADCs, writing outputs to the DAC and synchronising timing.

Table 5.1a and 5.1b give the average execution times for each of the macros in instruction cycles. For the demonstration system, an instruction cycle takes 400ns,

although the processor may be operated with an instruction cycle time of 100ns.

Macro	Function	Number of Instruction Cycles
s_add	Add two intervals	10
s_sub	Subtract two intervals	10
s_neg	Change the sign of interval	10
s_mult	Multiply two intervals	68.5
s_div	Divide two interval	103.5

Table 5.1a : Execution times for scalar interval macros

Macro	Function	Number of Instruction Cycles
msc	Multiply interval vector by scalar	42.5N+8
scprod	Calculate scalar product of two intervals	93N+27.5
v_add	Add two vectors of intervals	7N+5
v_sub	Subtract two vectors of intervals	7N+5

Table 5.1b : Execution times for vector interval macros for a vector length of N.

5.2.2. The FTF Algorithm on a TMS320C25

Having developed macros to perform interval arithmetic, the implementation of the FTF algorithm is fairly simple. The scale factors used are the same as for the fixed point software simulation of the algorithm, described in section 4.6 except for the vector of filter coefficients, $\underline{H}(k)$ which is modified to have its binary point in position 13, to enable filter coefficients of between -4 and +3.99988 to be represented without overflow. It is possible that further improvements in the hardware performance could be obtained by changing the scaling factors, but the difficulty of

detecting overflow in a hardware implementation would require that a considerable amount of experimentation was necessary in choosing the optimum scale factors. The fixed point scaling of any algorithm is a difficult and time consuming task and this is particularly true of an algorithm with the organisational complexity of the FTF algorithm. The use of a floating point DSP could eliminate this requirement. The TMS320C25 was used in its saturation overflow mode to minimise the effects of any overflows or underflows by forcing the accumulator to its largest positive or negative value as required.

The assembly language program to implement the FTF algorithm is given in appendix C.

5.3. Test Configuration

The application in which the hardware implementation of the interval FTF algorithm was tested was that of adaptive equalisation. A board was developed to generate baseband signals similar to those which would be encountered in a real digital communications system. The configuration used is similar to the software simulation described in section 4.8.

5.3.1. Generation of Test Signals

Figure 5.2 shows a block diagram of the board used to generate test signals for the hardware adaptive equaliser. A 31 bit shift register is used to generate a pseudo-random binary sequence[128], which represents the transmitted signal. The final four stages of the shift register are used to represent the transmitted symbol, denoted by $z(k)$ and the transmitted symbol delayed by one, two and three sample periods, $z(k-1)$, $z(k-2)$ and $z(k-3)$.

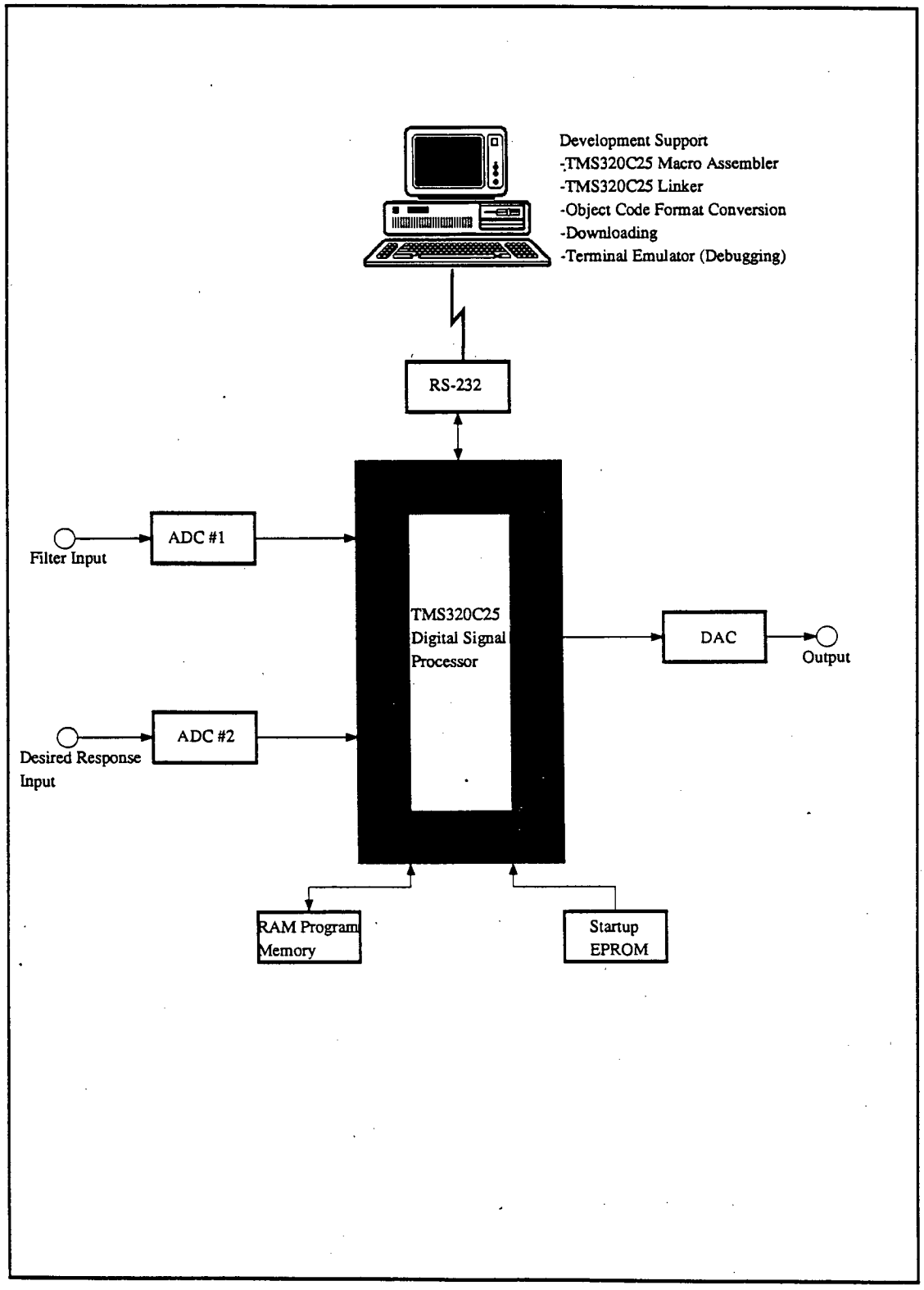


Figure 5.1 Block diagram of the TMS320C25 board and associated hardware

The output of the channel is distorted by intersymbol interference (ISI) and is given by

$$x(k) = h_{chan_0} z(k) + h_{chan_1} z(k-1) + h_{chan_2} z(k-2) + h_{chan_3} z(k-3) + noise \qquad [5.1]$$

As z(k) is a digital signal, it only has the values 0 or 1 and so, the multiplications in equation [5.1] can be performed in hardware using simple switches, rather than expensive analogue multipliers. If a switch is on, this corresponds to the output of the switch being equal to the input multiplied by 1 and if it is off, this corresponds to multiplication by zero.

The outputs from all the switches (multipliers) are added, along with some Gaussian noise, using an analogue summing amplifier, so as to implement the channel described by equation [5.1].

There are three connections from this board to the TMS320C25 processor board. The first forms the desired response input to the adaptive filter. It is generated by passing the pseudo-random binary sequence output, $z(k)$ through some analogue stages which enable the offset and the amplitude of this signal to be controlled. This signal is used as the input to one of the ADCs on the TMS320C25 board. The second ADC is connected to the summing amplifier which gives the channel output as described by equation [5.1]. The input to this ADC represents the primary input to the adaptive filter. The third connection from the test signal generator to the processor board is a bit clock, which is simply the clock signal used to control the shift register in the pseudo-random binary sequence generator. This signal is used by the processor board to trigger the start of conversion (SOC) on the analogue to digital converters, to ensure that the input signals are sampled at the correct time, which is at the centre of each bit. The software achieves synchronisation by waiting for an end of conversion (EOC) to be signalled by both ADCs before attempting to read inputs from them.

5.3.2. Equaliser Arrangement

A five coefficient equaliser was used in the system and no delay was used in the desired response input path, so that only minimum phase channels could be equalised. Although the board described in section 5.3.1 can generate intersymbol interference over four bit periods, only ISI over three bit periods was actually used in hardware tests.

5.3.3. Measurement of Results

Two outputs from the adaptive filter were measured in different experiments. The output from the adaptive filter, denoted by

$$y(k)=\underline{H}^T(k-1)\underline{X}(k) \qquad [5.2]$$

was measured. If the adaptive equaliser performs correctly, this signal should closely approximate to the transmitted signal, $z(k)$. The other output to be measured was the filter error, denoted by

$$e(k)=d(k)-\underline{H}^T(k-1)\underline{X}(k) \qquad [5.3]$$

where $d(k)$ is the desired response of the adaptive filter, which in this case is equal to $z(k)$. After initial convergence, $e(k)$ should remain small, indicating that divergence due to numerical inaccuracies is not occurring.

Both signals were captured using an HP5183 digital storage oscilloscope. This enables results to be displayed, plotted on a pen-plotter and stored.

Figure 5.3 shows a block diagram of the whole system including test signal generation, equalisation and capture of results

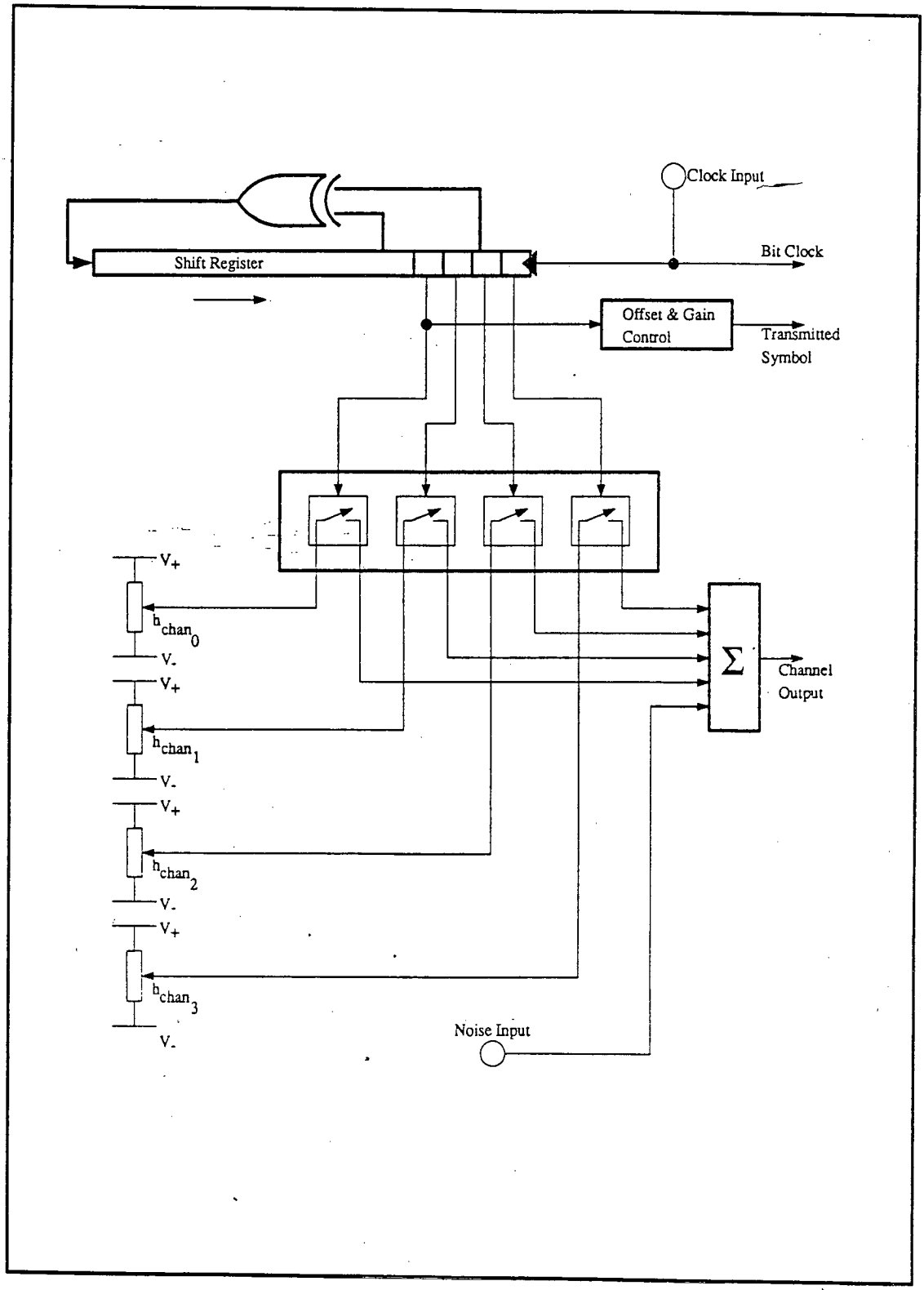


Figure 5.2 Block diagram of the board used to generate test signals for the hardware adaptive equaliser

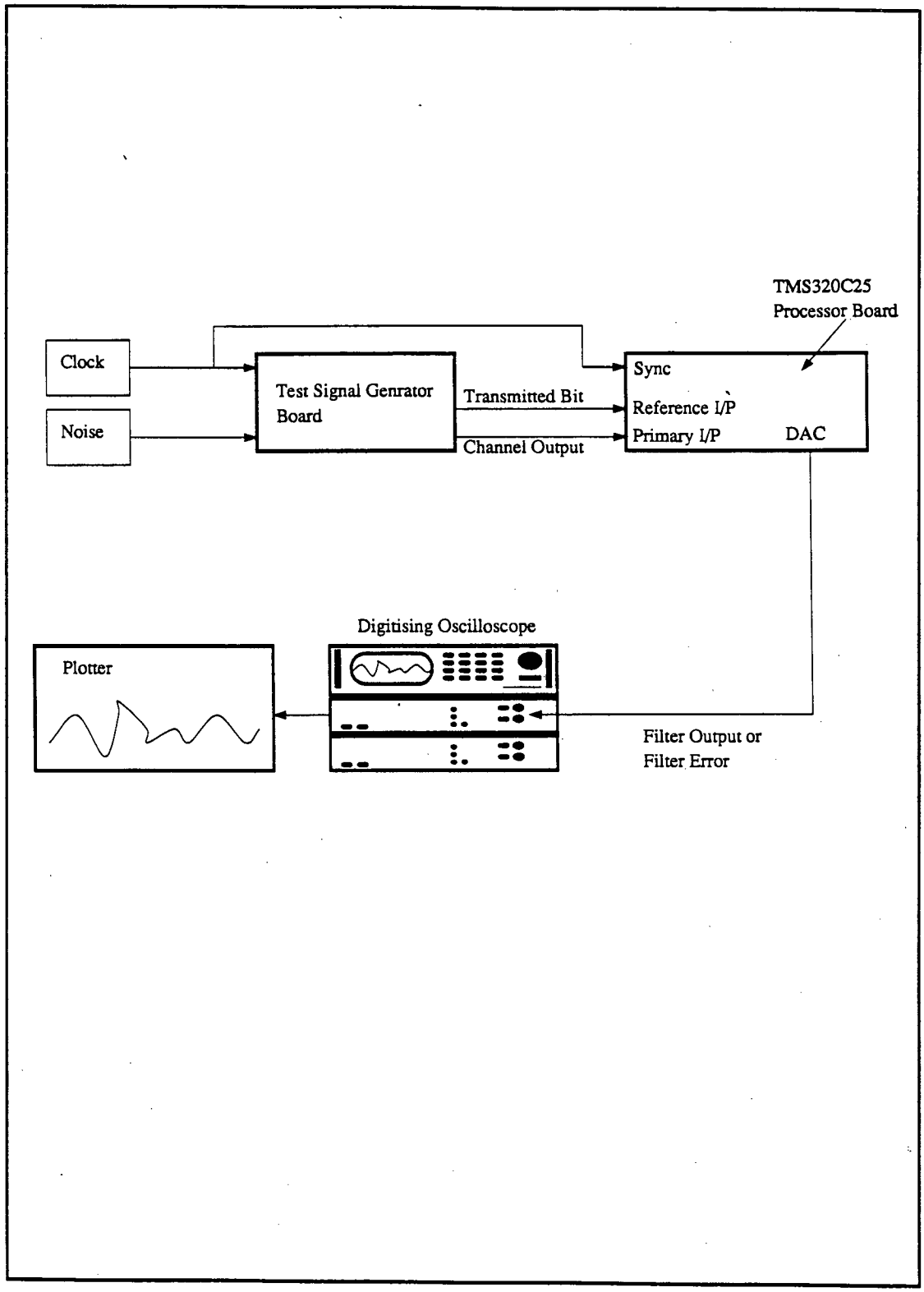


Figure 5.3 Block diagram of the hardware test system including signal generation, equalisation and measurement

5.4. Results

5.4.1. Eye Diagrams

Figures 5.4, 5.5 and 5.6 show various eye diagrams measured in the equaliser system. An eye diagram is simply a trace of all possible values of a signal, formed by recording the signal over a large number of bits and plotting it. An eye pattern which is open in the centre indicates that the signal could be used successfully with a decision device to recover the original binary symbols.

Figure 5.4 shows the eye diagram of the pesudo random binary sequence which forms the desired response input. As would be expected, there are only two distinct levels, corresponding to the transmitted symbols 0 and 1 respectively. The eye pattern is wide open in the centre. This eye diagram represents the ideal pattern for an equaliser which performs perfectly.

Figure 5.5 shows the eye diagram at the output from the channel. Instead of having two distinct levels, it has eight levels due to the intersymbol interference introduced by the channel. No noise was used during this test. The eye pattern is almost closed in the centre, indicating that equalisation is required if the original sequence is to be recovered.

Figure 5.6 shows the eye diagram measured at the output from the interval FTF equaliser. The eye pattern has been opened by the equaliser, which has removed much of the distortion introduced by the channel. The original transmitted sequence could be recovered from this signal, indicating successful equalisation.

5.4.2. Filter Error

Figures 5.7 and 5.8 show the instantaneous square of the filter error. If the algorithm performs correctly, this error should rapidly become very small as the algorithm converges. After convergence, the error should remain small if the algorithm is numerically stable.

Figure 5.7 shows the initial convergence of the algorithm. It can be seen that after a few output samples, the filter error becomes small, once again illustrating the rapid initial convergence of a least squares algorithm.

Figure 5.8 shows the long term error performance of the hardware equaliser. After the spike representing initial convergence, the error remains very small, indicating that the interval arithmetic rescue procedure for the FTF algorithm is working correctly and preventing divergence due to numerical errors.

5.5. Speed of Operation

The maximum speed of operation for the hardware adaptive filter was found to be 300 bits/s. The TMS320C25 processor was being operated at around ¼ of its maximum speed and so the program could be expected to operate at speeds of up to 1200 bits/s using the same processor. This data rate is fairly low although there are applications in telecommunications where equalisation is performed using sampling rates and filter lengths compatible with the performance of this implementation. Nevertheless, it would be desirable be able to operate the algorithm at much higher data rates. The low speed is partly due to inefficiencies in the assembly language program due to the requirement to make it very structured and the use of a set of general macros. It is believed that a modest performance increase could be obtained at the expense of making the assembly language code much more difficult to under-

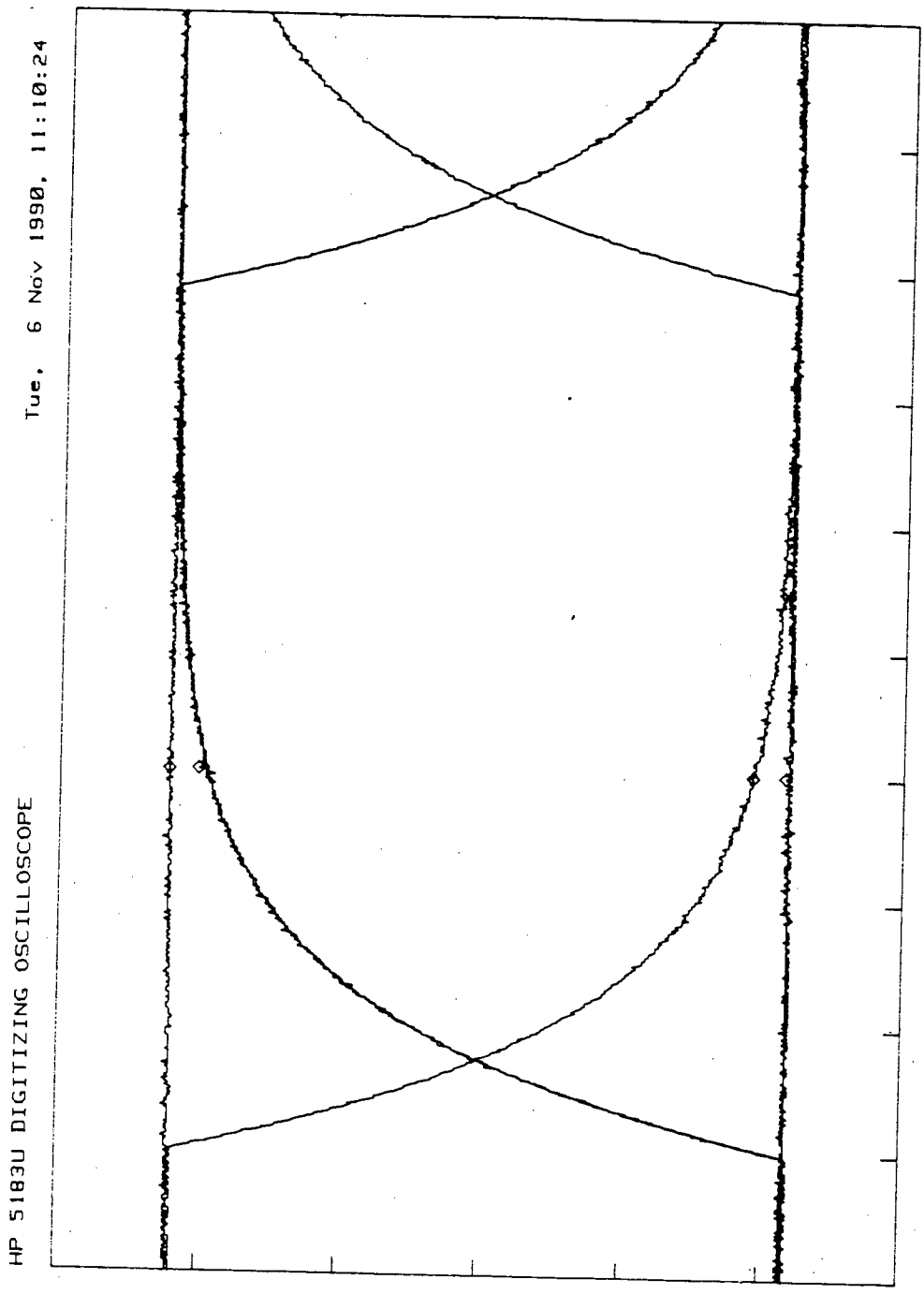


Figure 5.4 Eye diagram measured at the desired response input to the adaptive filter. The pattern is that of an ideal two level eye diagram.

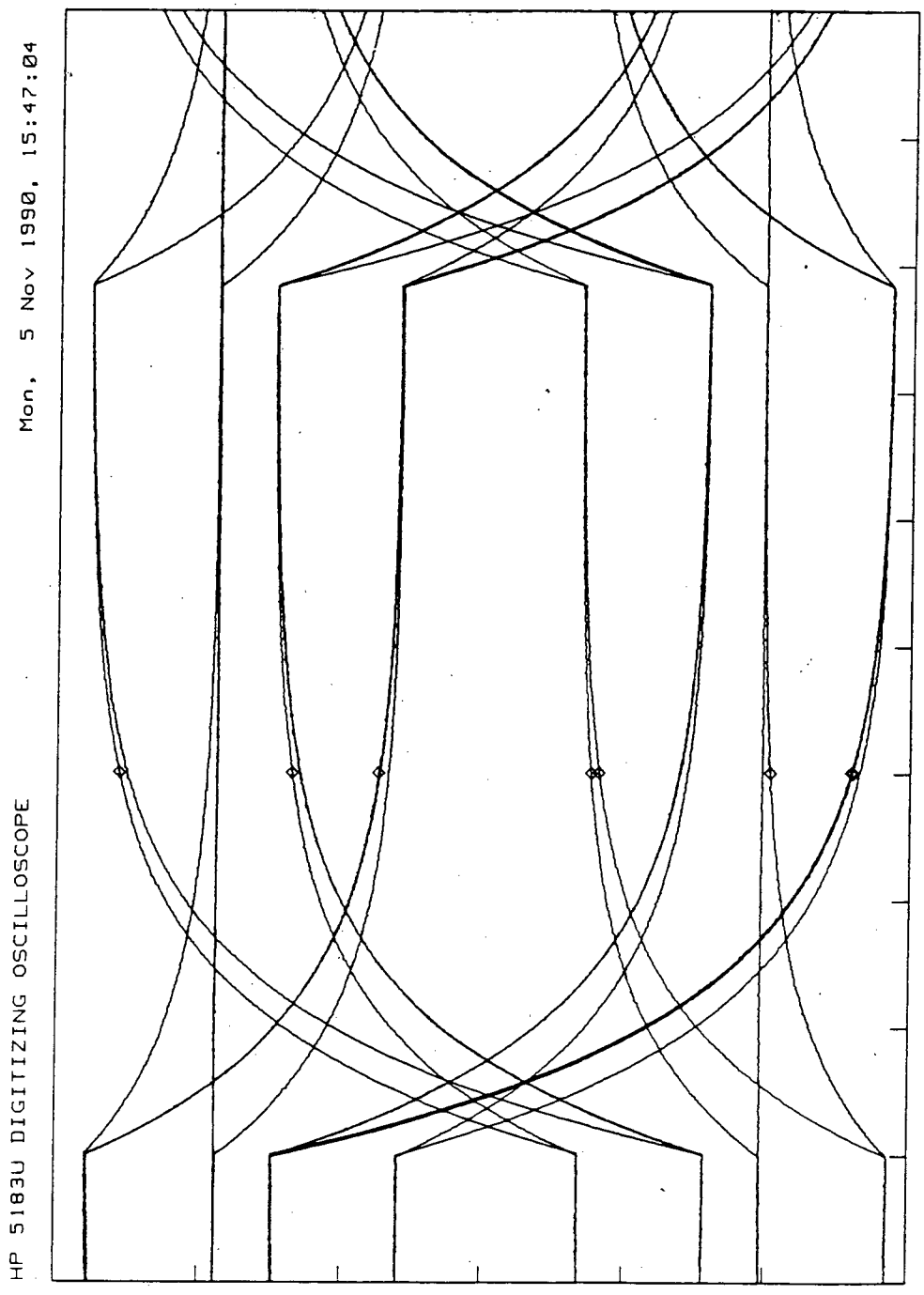


Figure 5.5 Eye diagram measured at the output of the channel. It has eight distinct levels due to the three coefficient channel which introduces intersymbol interference.

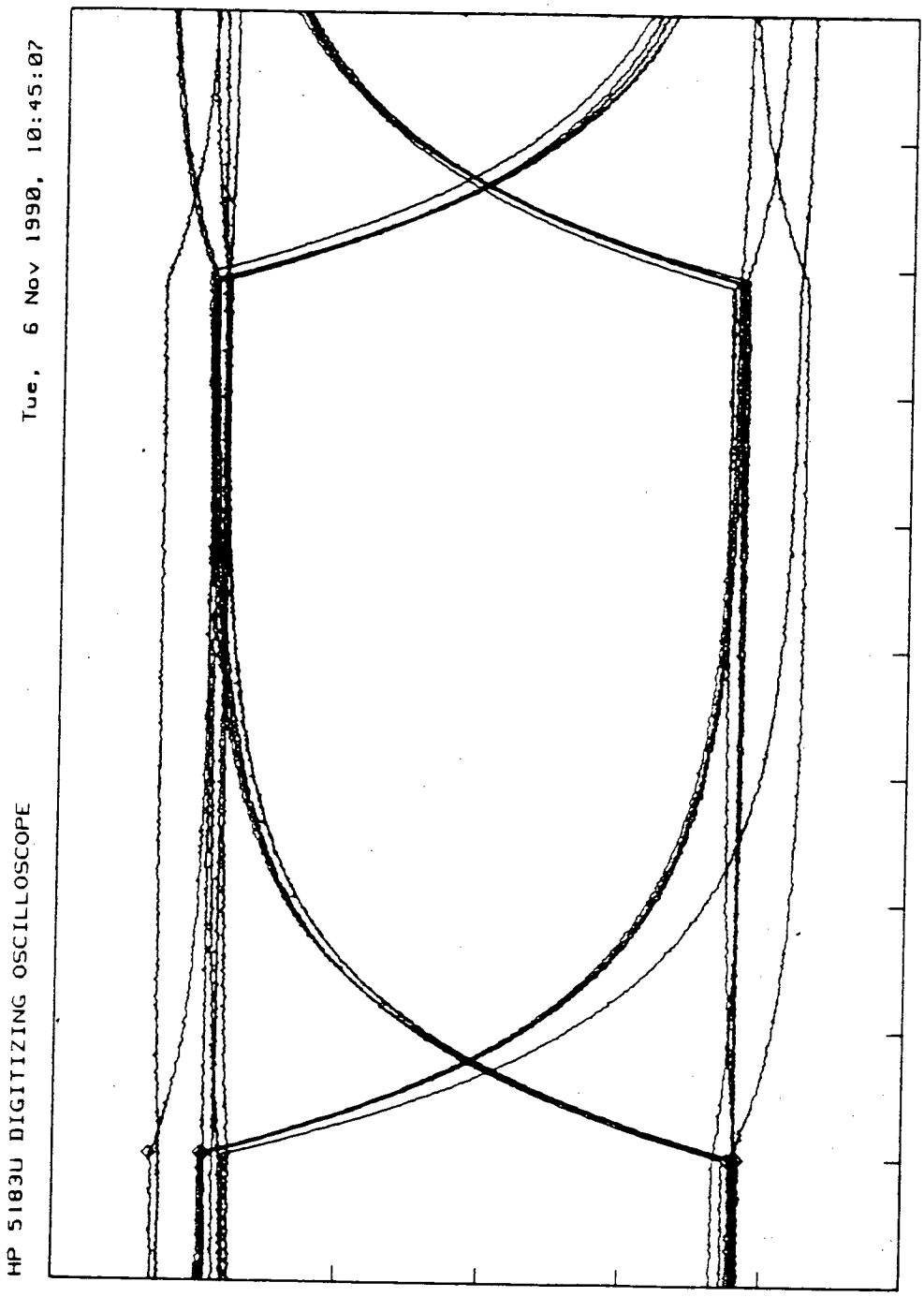
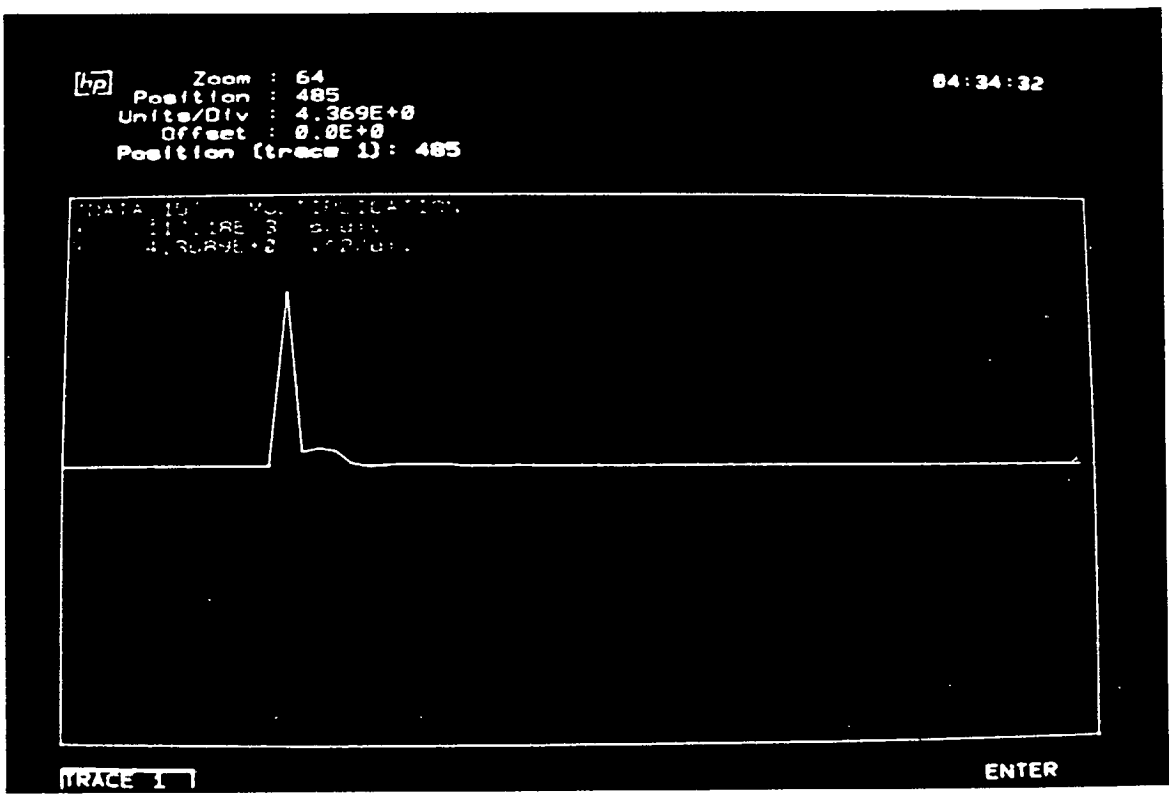


Figure 5.6 Eye diagram measured at the output of the adaptive equaliser. It is much closer to the ideal pattern of Figure 5.4 than the distorted pattern of Figure 5.5, indicating that the interval FTF adaptive equaliser has removed much of the intersymbol interference. $\lambda = 0.999969$, $\mu = 0.015258$, $\rho = 0.009155$



Adaptive filter starts

Figure 5.7 Graph of filter error squared against time for the interval FTF adaptive equaliser. The graph shows the rapid initial convergence of the filter error to a small value. $\lambda = 0.999969$, $\mu = .015258$, $\rho = 0.009155$

This graph is an expanded version of figure 5.8

stand and debug. A more significant and fundamental reason, however, is that the architecture of the TMS320C25 and other digital signal processors is not particularly suitable for the implementation of interval arithmetic.

5.6. Conclusions

The results presented in this chapter for the hardware implementation of the interval FTF algorithm are an important confirmation of the validity of the software simulation results. They demonstrate the feasibility of the interval fast RLS algorithms for application to real time systems

When considering the performance of the hardware implementation, there are two important aspects - the long term stability of the solution and the short term accuracy of it, both of which are influenced by the limited precision of the arithmetic used. From the results, the long term stability of the algorithm is good. After convergence of the algorithm, the filter error remains at a low level for hundreds of thousands of samples, indicating that interval arithmetic successfully prevents the divergence of the algorithm due to numerical errors. The short term accuracy of the solution is also acceptable, resulting in an eye diagram which shows an 'open' pattern, indicating successful equalisation. It is believed, however, that the accuracy of the solution could be improved by changing some of the scale factors used in the fixed point implementation. The choice of an optimum set of scale factors is difficult, but one of the major successes of this implementation has been to demonstrate that through the use of interval arithmetic, very limited precision fixed point implementations of the fast RLS algorithms are possible.

One important result obtained from the hardware implementation which could not have been obtained by software simulation of the algorithm is the maximum speed of operation. The maximum speed of operation of around 1200 bits/s for a TMS320C25 implementation is suitable for some equalisation applications in

telecommunications, but higher speeds would be desirable to increase the number of applications to which the system could be applied and gain the full advantages of using a fast RLS algorithm.

Two main reasons for the fairly low speed of operation have been identified. Firstly, the assembly language program has not been optimised for maximum speed of operation. This is because the joint requirements of producing structured code and minimising the number of instructions used are, to an extent, incompatible. It is believed that optimisation of the program could result in speed increases of the order of up to 50%, but would certainly not provide the large increase in speed required for many applications.

The second reason for the low speed of operation is that there is a mismatch between the architecture of the TMS320C25 DSP and the algorithm which is to be implemented on it. One of the most important instructions on any DSP is the multiply and accumulate instruction. This operation is extremely common in many signal processing techniques, such as convolution, correlation and recursive and non-recursive filtering. It is also important in the fast RLS algorithms which rely on implementing non-recursive filters to calculate forward and backward prediction errors, as well as to perform the filtering of input data. Unfortunately, as the TMS320C25 has only a single accumulator, a single cycle multiply and accumulate instruction cannot be performed using interval arithmetic and a very significant overhead is incurred in swapping the accumulator to temporary storage in data memory and performing multiplications and 32 bit long additions as separate instructions. The interval multiply and accumulate operation, which is performed as part of the macro "scprod" requires 82 instruction cycles and represents a significant contribution to the total time required to perform one iteration of the algorithm, as it is performed from within various loops.

It is of interest to note that this problem is not only relevant to interval arithmetic.

Any application which requires the use of complex numbers will be similarly affected and so this problem is of considerable importance.

Two solutions to the architectural problem are proposed : the use of a twin processor system which would make available two accumulators or the use of an interval coprocessor with a conventional DSP chip to perform the computations. The feasibility of the coprocessor is considered in detail in the next chapter.

6 An Interval Arithmetic Coprocessor for the TMS320C25

6.1. Introduction

One way of improving the performance of the DSP implementation of the interval arithmetic fast RLS algorithm presented in the previous chapter would be to develop a coprocessor device to provide hardware support for the interval arithmetic operations. In this chapter, a design for such a device will be examined and the feasibility of the design and likely performance will be discussed.

The coprocessor chip is designed to connect to the DSP address and data buses and to appear to the processor like a number of input and output ports. As the coprocessor is accessed using the processor's IN and OUT instructions, only one 16 bit transfer of data either to or from the coprocessor is possible in a single instruction cycle. For this reason, it is not worthwhile to implement operations such as interval addition and subtraction on the coprocessor, as they can be performed more rapidly using the main DSP. The design philosophy has, therefore, been to provide hardware support for interval multiplication and in particular, to develop a hardware architecture which provides for the rapid multiply and accumulate operations using interval arithmetic to implement a fast RLS algorithm.

An advanced software package was used to develop the design for the coprocessor. The package was developed under the Silicon Architectures Research Initiative

(SARI) programme[129] at the University of Edinburgh. It enables the designer to develop rapdily digital very large scale integration (VLSI) technology devices. The tools allow a designer to proceed automatically from a description of the behaviour required of a device to a gate level description of the structure required to implement the device. The tools allow the designer to have a large degree of control over the translation from behavioural description, which is a high level language description of the functions required of the device to structural layout to enable the design to be optimised in different ways. Starting with the same behavioural description, it is possible to use this flexibility to develop a device with the minimum possible number of gates, or the maximum speed of operation, or with respect to any other optimum criteria set by the designer. The tools automatically ensure the logical correctness of the design and carefully check that timing specifications for each component in the resulting structure are met, eliminating many of the errors which would be generated by a manual design process. The tools also support hierarchical designs and a design may be structured in a manner analogous to structured software programming.

In this chapter, two levels of the design of the coprocessor chip will be considered. The lower level of the design involves the development of an interval multiplier, which is simply a component which takes the endpoints of two intervals as inputs and gives the endpoints of the product of these intervals as its output. The higher level of the design handles all of the communications with the TMS320C25. It provides all of the registers and logic required to interface with the DSP address and data buses, has two accumulator structures and also has, as one of its components, the interval multiplier, which is synthesised at the lower level.

6.2. The SARI Toolset

The central aim of the SARI toolset[130] is to enable the designer of a complicated VLSI system to proceed rapidly from an algorithmic description of the computations and functionality required to a gate-level structural configuration. In so doing, the toolset ensures that a logically correct design is produced, assuming that the original algorithmic description is error-free.

It is clear that for any algorithmic description of a process, there will be a great number of possible structures, all of which perform the process. The choice of an optimal structure depends to a large extent upon the particular compromises and constraints which are necessary. Depending upon the application, a designer may wish to generate a design which minimises component cost, maximises processing speed, or which has the lowest possible power consumption. The toolset, therefore, allows the designer a large amount of freedom in the design process, rather than attempting to automate it completely. Hence, there is a high degree of interaction during the task of conversion from an algorithmic to a structural description, in which many of the creative design decisions have to be made by the designer. Moreover, the netlist which is finally produced by the tools is technology independent. It is simply a description of a digital logic configuration which is one implementation of the algorithm required. The conversion from this netlist to a physical silicon layout can be performed by a conventional silicon layout package, allowing complete freedom in the technology of the actual gates used to realise the netlist structural description.

The design process begins with the development of an algorithmic description. The description is written in the VHSIC hardware description language (VHDL)[131] which was developed as part of the very high speed integrated circuit (VHSIC) programme by the United States Department of Defence. The VHDL language supports both behavioural and structural descriptions of electronic hardware, but for

algorithmic specification, only the behavioural parts of the language need be used. The VHDL description for the coprocessor is listed in appendix E. Around 200 lines of VHDL source code were developed to describe the coprocessor and interval multiplication. The benefit of specifying the algorithm in VHDL is that it is possible to simulate directly from the VHDL program, using the Standard VHDL 1076 Support Environment (VSE)[132] The correctness of the algorithmic statement of the problem can, therefore, be checked and the performance of the device can be verified before the design process proceeds.

After simulation, the next stage to be undertaken when using the SARI toolset is translation from VHDL to an intermediate SARI language known as Babble. This intermediate representation lists all of the operations which have to be performed so as to carry out the computations of the algorithm, as well as the signals that each of these operations must use as inputs and generate as outputs. The translation from VHDL to Babble is performed completely automatically. The Babble representation is understood by the SARI synthesis tool, which is used to perform the rest of the design process.

The first stage of this process is resource selection, in which each of the operations in the algorithm is matched to a physical resource such as an adder or multiplier. The matching may be performed either automatically, or the user may manually match operations to specific resources, so as to meet the design goals.

Having defined which resource is used for each operation, the next stage is to schedule the use of the resources. The main tool in doing this is the resource time (R/T) graph. This is a chart in which the horizontal axis identifies each of the available resources and the vertical axis represents time. A shaded area on the R/T graph indicates that a resource is in use at a given time. An important design rule is that only one operation may be scheduled on a single resource at any one time. The SARI synthesis tool enforces this and ensures that the scheduling is valid. It sup-

ports a comprehensive model of time which enables setup, hold and reuse times for resources to be specified. For pipelined resources, the reuse time may be less than the execution time. Initially, the resources are allocated so that each operation is given a different resource and ASAP (as soon as possible) scheduling is used, so that as soon as all of the signals required to carry out an operation are available, it will be performed. This allocation generally gives a configuration which will operate very rapidly, but which would require an unreasonably large area of silicon, as it uses a very large number of resources. The design process can then proceed by binding several operations to a single resource. The resource may, therefore, be reused, resulting in a design which uses fewer resources, but which may not operate as quickly. When the binding is altered in this way, the design must be rescheduled to ensure that valid timing is again obtained.

When the designer is satisfied with the resource binding and scheduling, the next stage is memory synthesis, in which the various storage requirements of the algorithm are created. The memory resources are random access memories (RAM), which may have multiple ports. A location in a RAM resource must not be reused until the value stored in it has been read and is no longer required. Moreover, each data port on the RAM will only support one access (read or write) at any clock cycle.

Having generated and allocated memory components, the designer proceeds by communications planning and synthesis. This stage involves connecting the resources together, using wires, multiplexers and tri-state buffers as required. This process may be performed automatically by the toolset.

At this stage of the design process, a fully connected data processing network has been completed, but two design steps remain. The first is address generation, in which local address generators are designed, so as to ensure that the memory which has been synthesised has the correct locations accessed at the correct times. The

second requirement is that of control synthesis, in which a finite state machine controller is generated, to provide control signals for the various components used in the design, such as tri-state buffers and multiplexers, as well as more complicated components, such as arithmetic logic units (ALUs), which may have simple operations such as additions bound to them. Unfortunately, version 4.2 of the SARI toolset does not support all of these design steps and they would either have to be performed manually by the designer, or by a later version of the SARI tools.

6.3. Functions of the Coprocessor

Table 6.1 and Figure 6.1 show the coprocessor as it appears to the DSP. The coprocessor has a chip select input, CS , which must be high to access the device. The $\bar{R}/W$ input to it must be low during read operations and high when writing to the device. It also has an address bus, used by the coprocessor to determine which interval register is to be accessed and input and output data buses, "w_data" and "r_data" respectively. For a practical design, these buses would be combined on to a single bidirectional data bus, using the $\bar{R}/W$ signal to determine the direction of data transfer.

Internally, the coprocessor has a number of registers. By writing to locations 0 to 3, it is possible to load the 16 bit wide registers "op1_reg", "op2_reg", "op3_reg" and "op4_reg", which contain the endpoints of the interval to be multiplied. The shift control register, "shift_reg", is located at address 5 and a value may be written to it to determine the shift which will be applied to the accumulators so as to form the results. A control register at location 4 enables various functions to be selected on the coprocessor. Writing the value '1' to the control register zeros both of the accumulators and writing the value '2' to it causes the device to perform an interval multiplication and add the result to the existing contents of the accumulator. The accumulator contents are then transferred to the result registers "res_0reg" and

"res_1reg", using a left shift controlled by the contents of the "res_shift" register. The results can be accessed by reading from the locations 0 and 1, which give the contents of "res_0reg" and "res_1reg" respectively.

6.4. Design of the Interval Multiplier

Figure 6.2 shows the control flow graph for the interval multiplier. It is very simple, consisting of a single block of computation which takes four inputs and generates the interval product of these inputs, giving two outputs. This simplicity in control flow is because the behavioural description of the procedure does not contain any conditional statements or loops.

The diagrams in this chapter are produced directly from screen displays which were generated while using the SARI toolset.

Figure 6.3 shows the structure which is initially associated with the interval multiply behaviour before synthesis of memory locations, address generators, type converters, communications and control components. The figure shows the structure as consisting of four multipliers and two components, designated "find_min" and "find_max". The behavioural code describes interval multiplication, using

$$[a^l,a^u]\times[b^l,b^u]=[\min(a^lb^l,a^lb^u,a^ub^l,a^ub^u), \quad \max(a^lb^l,a^lb^u,a^ub^l,a^ub^u)] \qquad [6.1]$$

directly, the four multipliers being required to calculate the four products a^lb^l, a^lb^u, a^ub^l and a^ub^u, and the two components "find_min" and "find_max" being required to choose the largest and the smallest of the products respectively. These components represent lower levels of the design hierarchy, which consist of a number of comparison operations.

A resource/time graph for the computational block of the interval multiplication design may also be generated. Figure 6.4 shows the initial allocation of resources for

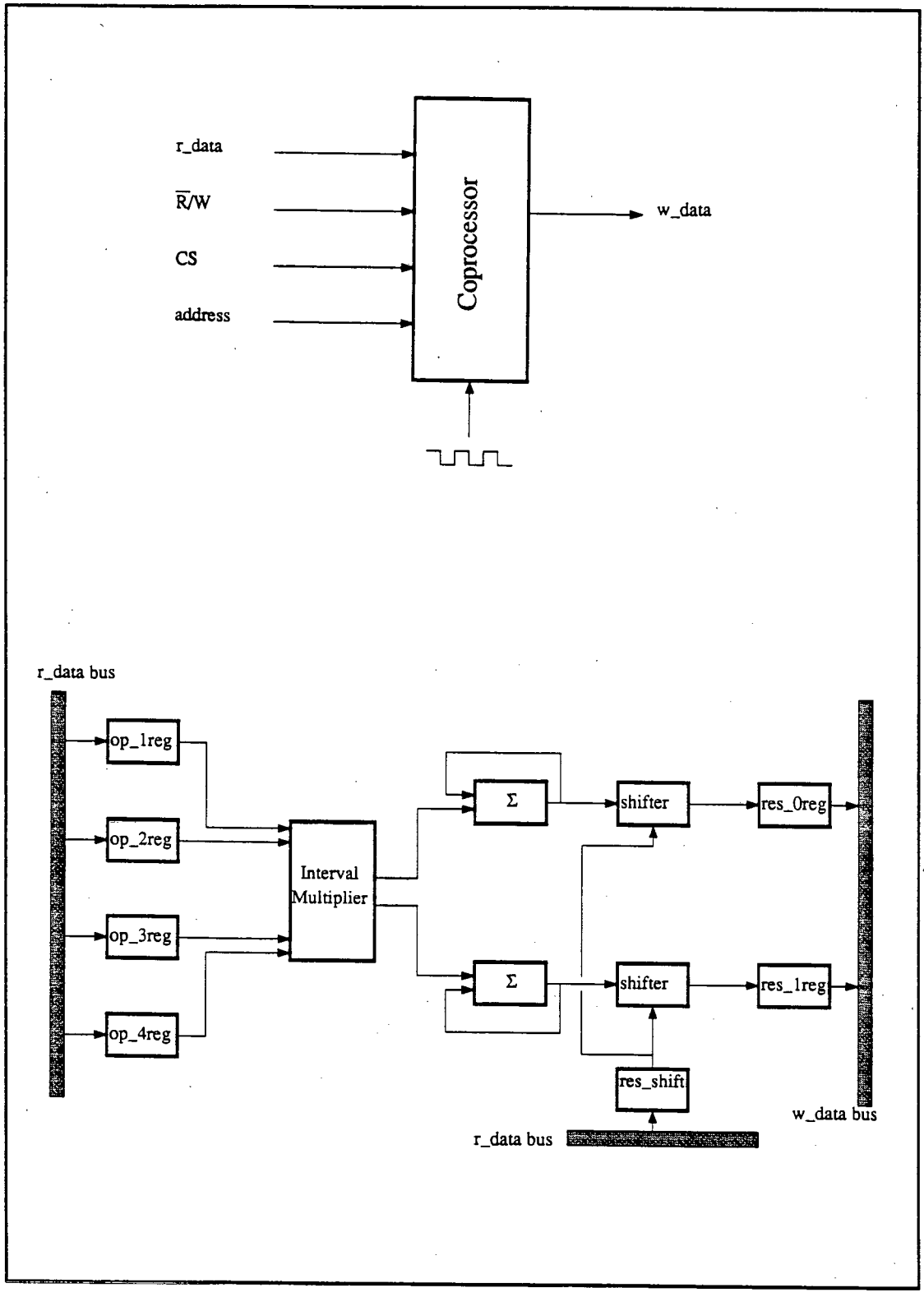


Figure 6.1 Data flow path for the coprocessor.

Location	Read		Write	
	Register Name	Function	Register Name	Function
0	res_0reg	Lower endpoint of result	op1_reg	operand 1 lower endpoint
1	res_1reg	Upper endpoint of result	op2_reg	operand 1 upper endpoint
2	-	-	op3_reg	operand 2 lower endpoint
3	-	-	op4_reg	operand 2 upper endpoint
4	-	-	control	'1'=zero accumulator '2'=multiply & accumulate 'other'=no operation
5	-	-	res_shift	left shift for calculating results

Table 6.1 Operation of the interval arithmetic coprocessor.

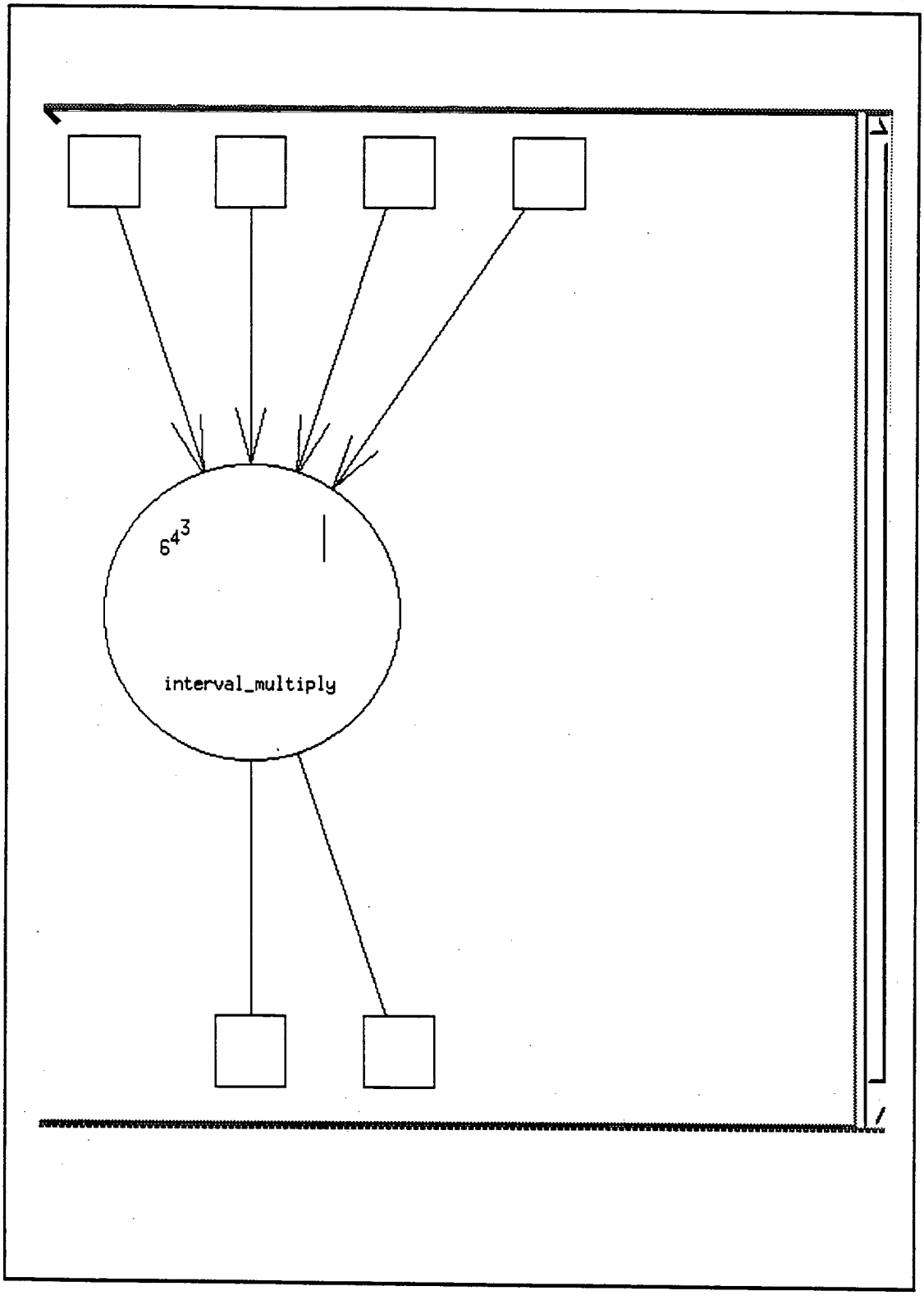


Figure 6.2 Control flow graph for the interval multiplication procedure

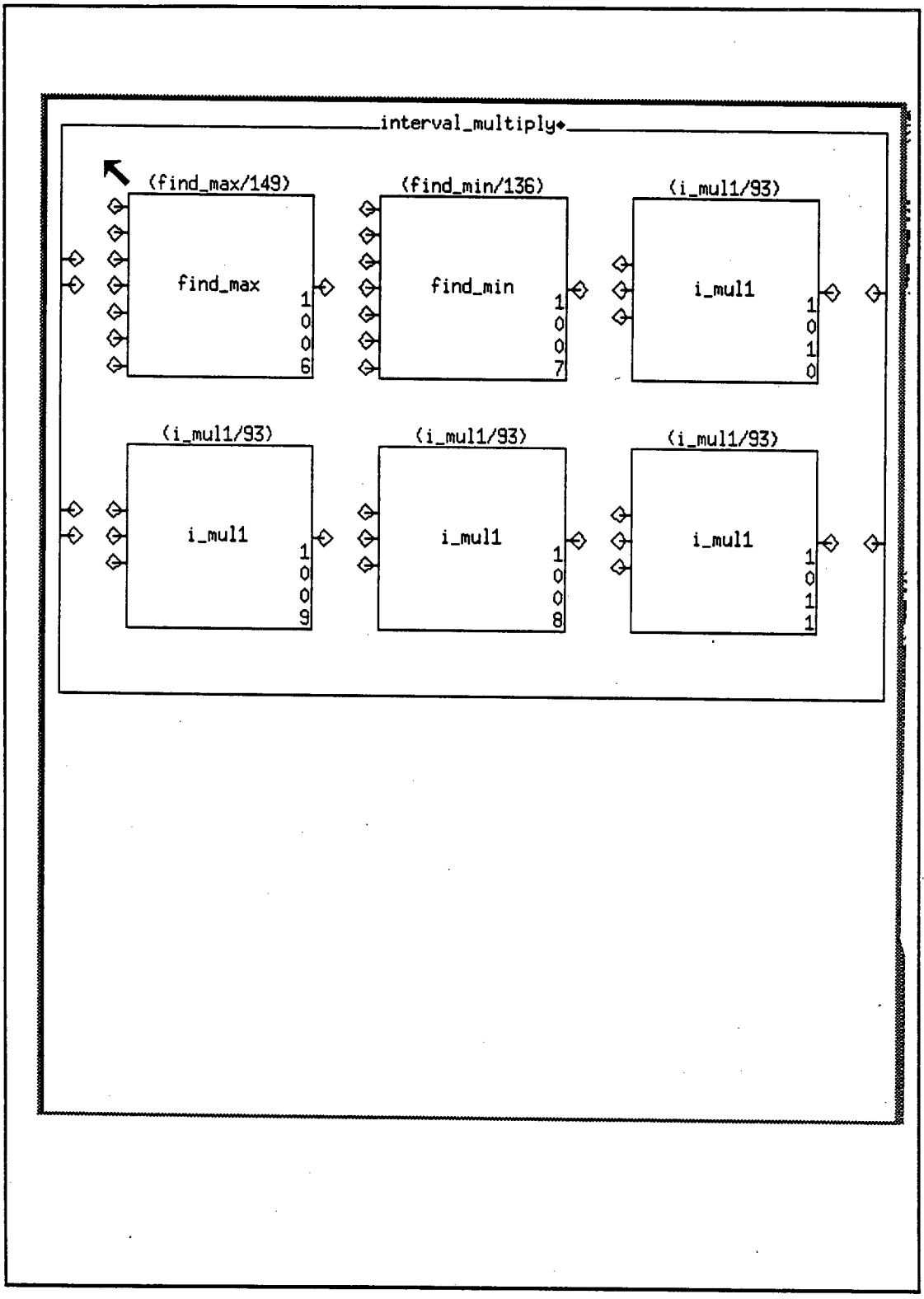


Figure 6.3 Initial structure of the interval multiply component

this block. Four separate multipliers are assumed to be available and the four products required are formed by a parallel process. This may be too costly in terms of hardware utilisation and so, the four multiplication operations may be scheduled to occur serially on a single multiplier. Figure 6.5 shows the R/T graph in these circumstances, noting that the processing operation now takes a considerably longer time.

Having bound the operations to the resources as required, it is now possible to synthesise the memory components, address generators, type converters, communications and control circuitry required to complete the structure. Figure 6.6 shows the structure which is obtained using the four multiplier R/T graph of Figure 6.4. The diagram shows the four integer multipliers denoted by the boxes marked "i_mult1" and also the componenets "find_min" and "find_max", which find the minimum and the maximum of their four inputs. The SARI toolset has added a number of components including two registers, which store the result of each interval multiplication opeartion and a number of type converters, denoted by the boxes marked "etc" to enable the various components to be connected together.

Figure 6.7 shows the structure resulting from synthesis of the design with the single multiplier R/T graph of Figure 6.3. A number of additional registers have been allocated to the design, so that all four multiplications which are required to be performed can be scheduled onto a single physical multiplier. The result of each of the multiplication operations is stored until all four products are available. The components "find_min" and "find_max" are then used to compute the interval arithmetic result, which is then stored by reusing two of the registers.

This completes the structural design of the interval multiplier. This component is used as part of the higher level design, which includes all of the registers and other components shown in Figure 6.1.

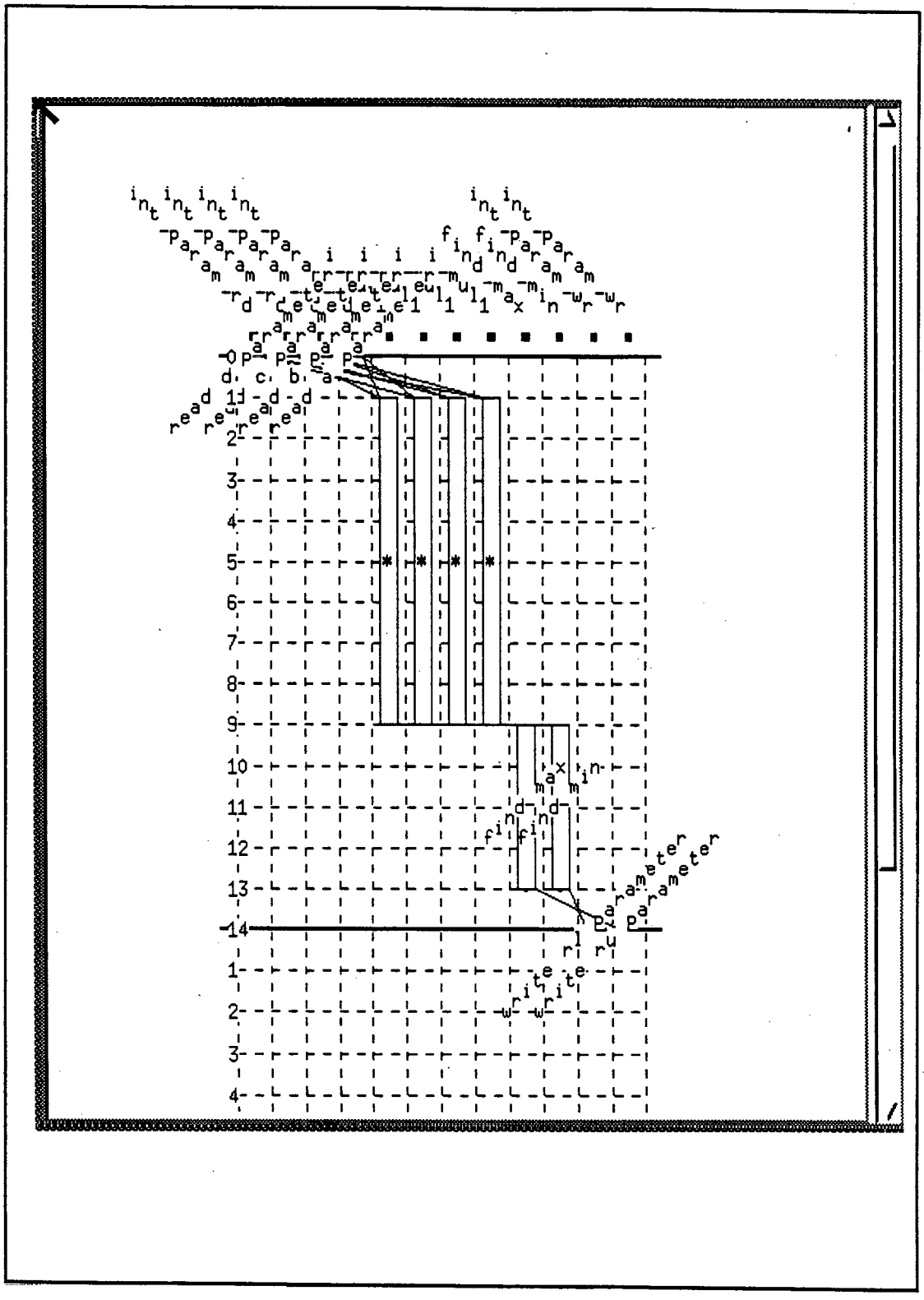


Figure 6.4 Initial resource time graph for the interval multiplier

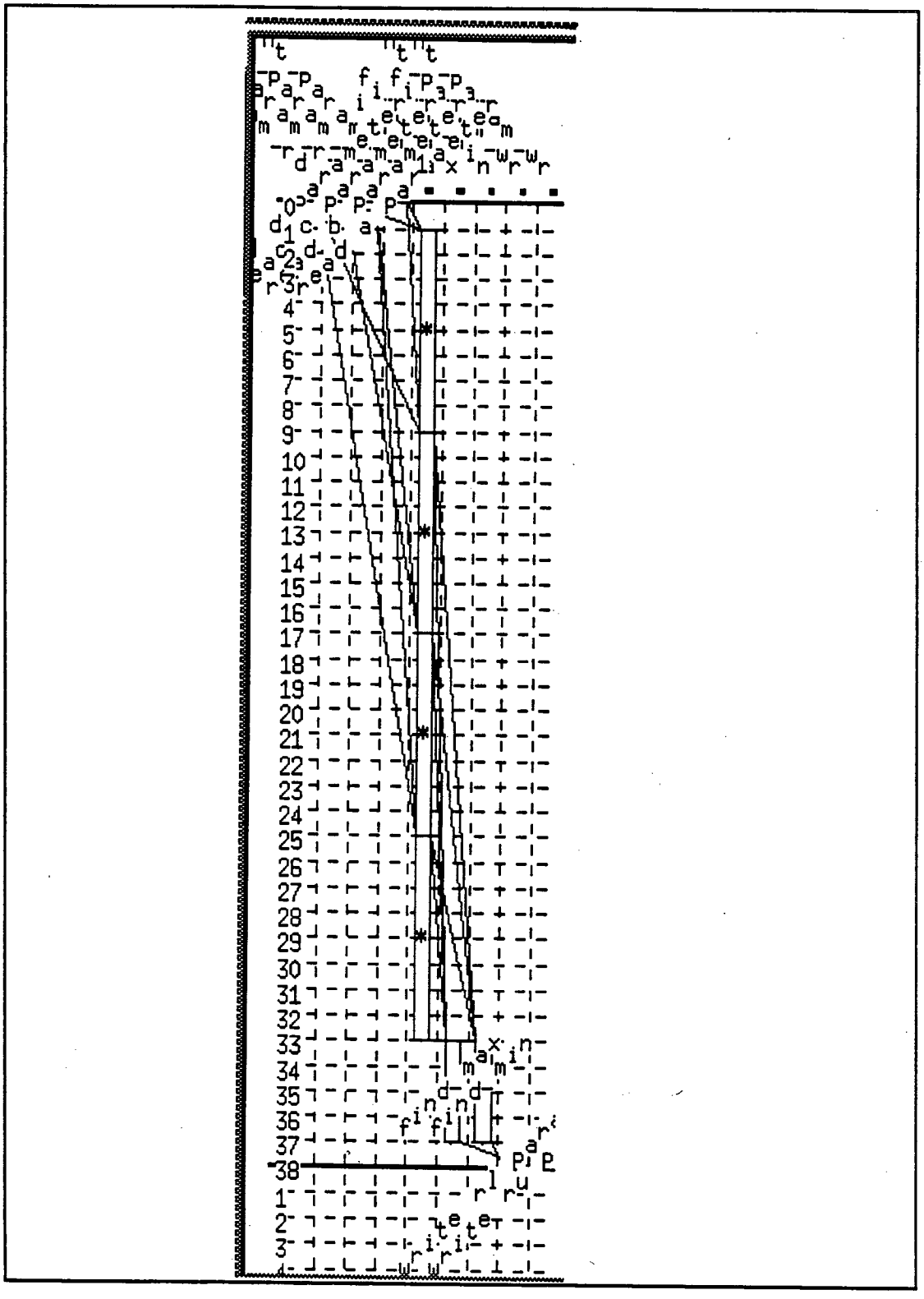


Figure 6.5 Resource/time graph after rebinding to schedule multiplications to occur sequentially on a single multiplier.

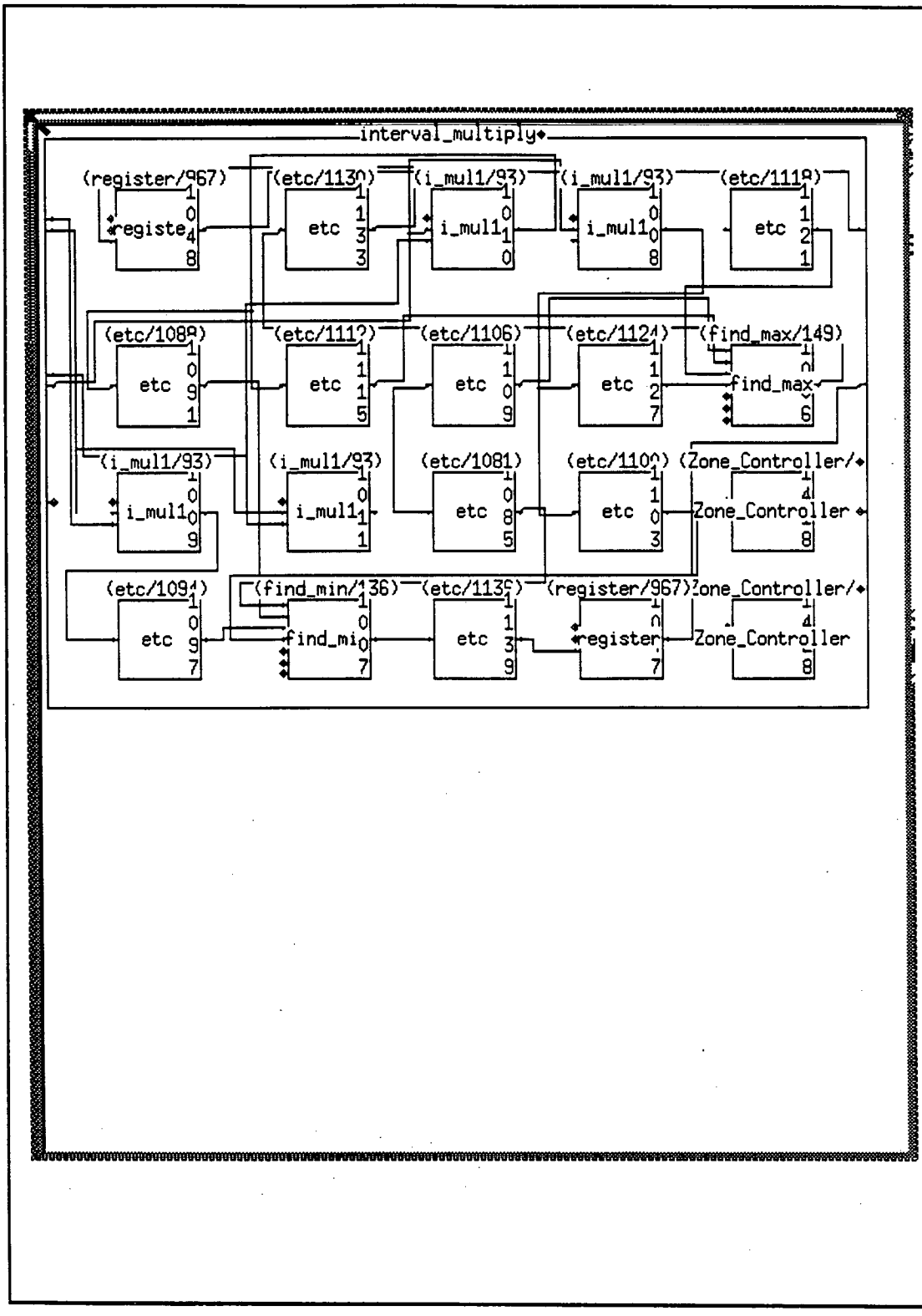


Figure 6.6 Structure of the interval multiplier, using 4 integer multipliers.

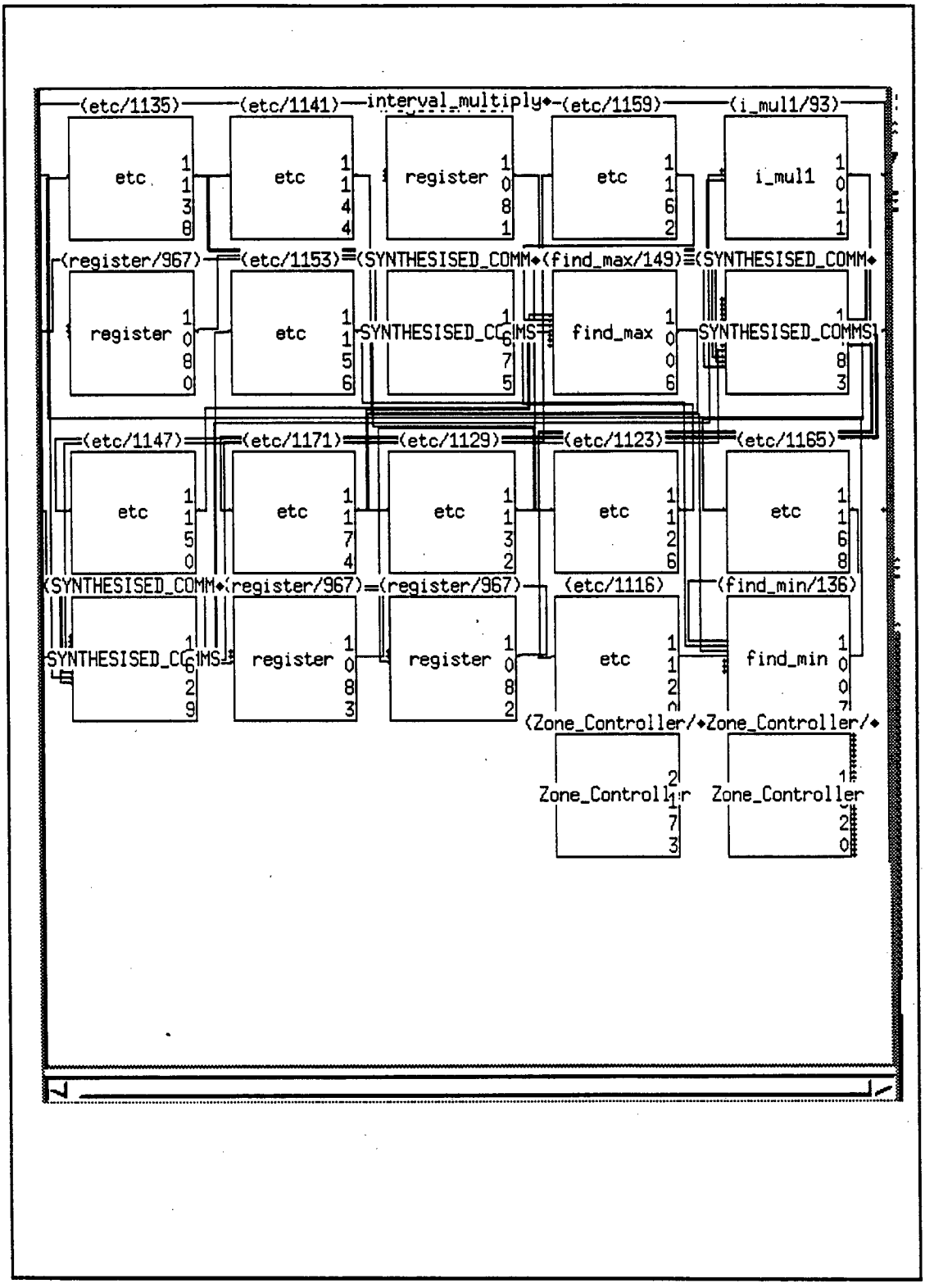


Figure 6.7 Structure of the interval multiplier, using 1 integer multiplier.

6.5. Top Level Design of the Coprocessor

The control flow graph for the top level of the coprocessor architecture is shown in Figure 6.8. It consists of a number of computational blocks corresponding to different conditional instructions in the description of the behaviour of the coprocessor device, which in turn correspond to the different operations available using the coprocessor chip, shown in Table 6.1.

Each of the computational blocks has an R/T graph, but due to the large number of blocks, these graphs are not shown. Most of the blocks consist of a single operation and therefore, no rescheduling of resources can be performed. The exception to this is the computational block which adds to the existing values in the accumulator and which left shifts the new accumulator values, which are then transferred to the result registers. This computation requires two add and two shift operations, each of which could either be performed using a single resource sequentially, or by using two identical resources in parallel. In this implementation, parallel resources were used for greater speed of operation.

Memory, address, type converter communications and control synthesis may be performed, yielding a structural implementation for the coprocessor device.

6.6. Feasibility of the Design

To yield a worthwhile increase in speed, the coprocessor would have to be capable of forming an interval product in around 10 instruction cycles. This would correspond to an increase in speed of around 6-10 times as compared with the software arithmetic routines "s_mult", "scprod" and "msc" described in the previous

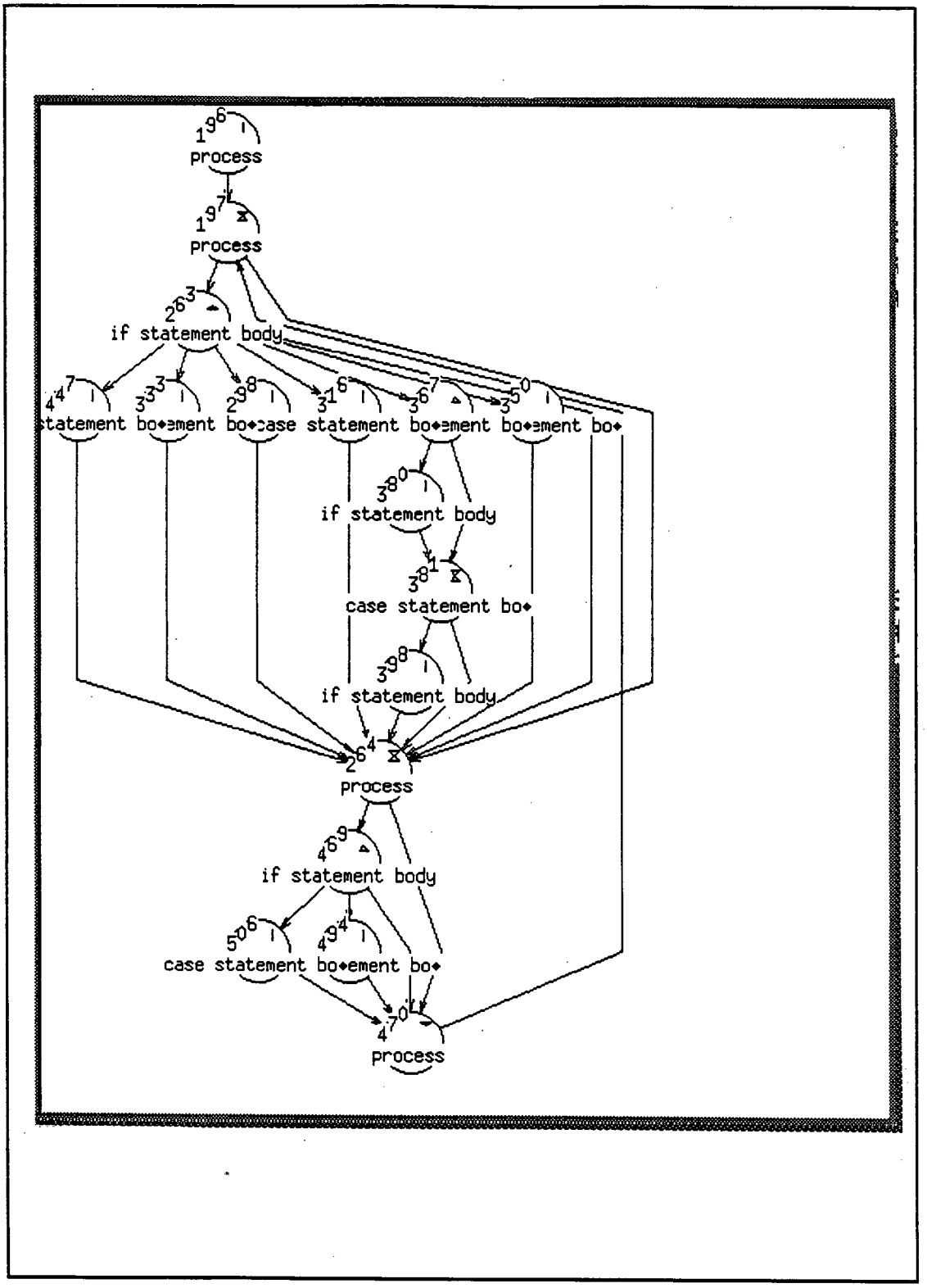


Figure 6.8 Control flow graph for the coprocessor chip.

chapter. If it is assumed that each register transfer to or from the coprocessor requires one instruction cycle, then four instruction cycles, or 400ns are required to load the four operand registers prior to performing the interval multiplication. This leaves a period of 600ns to form the interval product and add it to the contents of the two accumulators. There would also be an additional overhead of 3 instructions per scalar product operation, to load the shift control register and to read the two result registers. Therefore, the total time to perform an N point scalar product operation would be around $10N+3$ instruction cycles

Multipliers are available which can form a product in less than 100ns, such as the multiplier built in to the TMS320C25 which uses 1.5μm CMOS technology and so, the single multiplier version of the design, shown in the R/T graph of Figure 6.5 would yield the speed necessary. Alternatively, it would be possible to construct the four multiplier version of the coprocessor, which uses the R/T graph of Figure 6.4. This device could probably perform an interval multiplication in around $6N+3$ instruction cycles, including data transfers to the coprocessor.

The design could, therefore, yield a very worthwhile increase in speed for the interval FTF algorithm. It seems likely that the maximum data rate would be increased by a factor of 5-8, making the hardware adaptive filter suitable for a wider range of applications.

6.7. Conclusions

The feasibility of developing a vector product coprocessor for a digital signal processor using VLSI technology has been demonstrated. Advanced software tools have greatly speeded up the design process and have made it easier to proceed from a description of the behaviour of the coprocessor to a structural implementation.

Initially, it had been hoped to develop a dedicated device for the entire interval

arithmetic fast RLS algorithm. Unfortunately, the resulting design proved to be very complicated and although the SARI toolset could have synthesised a structure, it would probably not have been possible to implement it using current technology. The coprocessor presented in this chapter gives some hardware support for the interval arithmetic algorithm, but does not go to the extreme of attempting to implement the interval arithmetic FTF algorithm entirely in hardware. It is, therefore, a sensible compromise, given the current level of VLSI technology.

7 Conclusions

7.1. Achievements of the Work

The significant and original contribution of the work presented in this thesis has been the development of a new method by which the fast RLS algorithms may be stabilised, making use of interval arithmetic. This has complemented the range of existing stabilisation techniques.

The stabilisation of the fast RLS algorithms is by no means an easy task. Firstly, the finite precision errors which cause the divergence are the result of a non-linear truncation process. For this reason, even a probabilistic analysis of the output from a single fixed point multiplier is difficult. The organisational complexity of the fast RLS algorithms contributes further to the difficulties encountered in developing and analysing a suitable stabilisation procedure. Due to these difficulties, many existing stabilisation procedures do not offer any guarantee of absolute stability. Simulation results for the existing methods will generally demonstrate a very worthwhile improvement in robustness for particular input signals, but cannot offer proof of stability. It is partly for this reason that few practical adaptive filtering systems have been developed which have made use of the fast RLS adaptive algorithms. System designers are understandably unwilling to make use of any procedure to stabilise the fast RLS algorithms unless they can be certain about its effectiveness.

The appeal of interval arithmetic is that, due to the endpoint rounding scheme, the interval calculated will contain the infinite precision result of any calculation. This means that the interval technique can be guaranteed to give numerically stable performance and provided that the design constants associated with the rescue procedure are correctly chosen, the algorithms will also give useful performance. This guarantee is the main advance which has been gained by the use of interval arithmetic. Most other stabilisation procedures which have been proposed have relied on simulation results to demonstrate that more stable performance is obtained. Whilst many of the improvements in stability demonstrated in this way are very worthwhile, it is by no means certain that divergence will never occur. Using interval arithmetic, however, a guaranteed maximum error limit can be attained.

Simulation results have been presented which have confirmed the stability of the interval methods and have also demonstrated that there is no significant degradation in performance when the interval fast RLS solution is compared with the conventional RLS solution, which is assumed to give an exact least squares solution to the problem. A number of important configurations with practical applications have been demonstrated, including adaptive system identification and adaptive equalisation. Both time varying and non time varying problems were considered, so as to ascertain that both the tracking performance and the steady state accuracy of the algorithms are not significantly affected by interval arithmetic and the associated reinitialisation process.

Of particular importance to cost sensitive applications is that the interval FTF algorithm may be implemented using low accuracy fixed point arithmetic and will still give acceptable performance. While the 16 bit implementations gave good performance, it is believed that this is close to the minimum accuracy at which the interval FTF algorithms could be realised. It is likely that a 24 or 32 bit wordlength would yield excellent performance, with the potential to be even better than 32 bit floating point arithmetic, provided that the scale factors are appropriately chosen.

Interestingly, the Motorola DSP 56000 processor offers 24 bit fixed point arithmetic and so may be better suited to this particular application than the Texas Instruments TMS320C25 processor used for the hardware realisation. However, the use of fixed point arithmetic with fast RLS algorithms has not been previously documented and so the 16 bit implementation is particularly significant.

7.2. Limitations and Areas for Future Work

Perhaps the most serious limitation to the use of interval arithmetic is its increased computational complexity compared with single valued real number arithmetic. Interval addition and subtraction require two real number operations to be performed. If the algorithm of[117] is used, then interval multiplication may be performed with an average of 2.4 real number multiplications, but there is an additional overhead involved in making the decisions for the conditional part of this algorithm. The penalty for obtaining stable performance using the fast RLS algorithms is, therefore, considerably increased computational complexity. The complexity remains, however, linearly dependent upon the length of the adaptive filter and so, the advantages of using a fast algorithm are not lost.

Another possible criticism of the interval arithmetic stabilisation method is that it reinitialises the algorithm on the basis of a pessimistic worst case error analysis. This means that reinitialisation takes place considerably more frequently than may be strictly necessary. One alternative which could be considered is to replace the rounding procedure for the endpoints with a probabilistic one, in which there is a small probability that an endpoint will actually be rounded in the wrong direction. This could lead to a more realistic model of the truncation errors, but it also introduces two additional problems. Firstly, the more complicated rounding procedure adds further to the complexity of the interval algorithm. Secondly, the guarantees associated with using a worst case analysis are lost and so, it becomes difficult to be

certain of the absolute stability of the algorithm. Furthermore, the simulation results have demonstrated that there is little degradation in performance caused by the regular reinitialisation of the existing interval method and so, there would be little to be gained by this more complicated rounding arrangement.

One area for further work which would be worthwhile would be a comparative study of the various stabilisation methods. Such a study would have to compare the complexity of the different methods, the relative accuracy of the solution produced by each algorithm and quantify how stable the different procedures are. It would also be of interest to see how suitable each of the stabilisation procedures is for hardware implementation, particularly when using fixed point arithmetic. Few results have been published regarding fixed point implementations of the fast RLS transversal filter algorithms, but this information is necessary so as to develop cost effective realisations of the algorithms.

Once these comparisons have been made and the characteristics of the different ways of implementing fast RLS algorithms are better understood, then practical applications for these highly efficient algorithms should become more widespread.

References

1. Ljung, L., Morf, M., and Falconer, D., "Fast calculation of gain matrices for recursive estimation schemes," *Int. J. Control*, vol. 27, pp. 1 - 19, 1978.

2. Carayannis, G., Manolakis, D. G., and Kalouptsidis, N., "A Fast Sequential Algorithm for Least Squares Filtering and Prediction," *IEEE Trans. Acoust, Speech, Signal Process.*, vol. ASSP-31, No 6., pp. 1394 - 1402, 1983.

3. Cioffi, J. M. and Kailath, T., "Fast, Recursive Least Squares Transversal Filters for Adaptive Filtering," *IEEE Trans. Acoust, Speech, Signal Process.*, vol. ASSP-32, No 2., pp. 304 - 337, 1984.

4. Widrow, B. and Stearns, S., *Adaptive Signal Processing,* Prentice-Hall, Englewood Cliffs, 1985.

5. Cowan, C. F.N. and Grant, P. M., *Adaptive Filters,* Prentice Hall, Englewood Cliffs, 1985.

6. Mulgrew, B. and Cowan, C.F.N., *Adaptive Filters and Equalisers,* Kluwer Academic Publishers, Norwell, Mass., 1988.

7. Koford, J.S. and Groner, G.F., "The use of an adaptive threshold element to design a linear optimal pattern classifier," *IEEE Trans. Info. Theory*, vol. IT-12, pp. 42-50, Jan 1966.

8. Dillon, L.S., *Principles of Animal Biology,* pp. 275-278, Macmillan, New York, 1965.

9. Zohar, S., "New hardware realisations of non-recursive digital filters," *IEEE Trans. Comput.*, vol. CT-22, pp. 328-347, April 1973.

10. Herrman, O. and Schussler, H.W., "Design of non-recursive digital filters with minimum phase.," *Electron. Lett.*, vol. 6, pp. 329-330, 1970.

11. McLellan, J.H. and Parks, T.W., "A unified approach to the design of optimum FIR linear phase digital filters ," *IEEE Trans. Comput.*, vol. CT-20, pp. 697-701, Nov 1973.

12. Rabiner, L.R., "Linear program design of Finite Impulse Response digital filters," *IEEE Trans. Audio Electroacoust.*, vol. AU-20, pp. 280-288, Oct 1977.

13. Roberts, R.A. and Mullis, C.T., *Digital Signal Processing,* Addison Wesley, Reading, Mass., 1987.

14. Gray, A.H. and Markel, J.D., "Digital Lattice and Ladder Filter Synthesis," *IEEE Trans. Audio Electroacoust.*, vol. AV-21(6), pp. 491-500, 1973.

15. Friedlander, B., "Lattice filters for adaptive processing," *Proc. IEEE.*, vol. 70, pp. 829-867, August 1982.

16. Honig, M.L. and Messerschmitt, D.G., *Adaptive Filters:Structures, Algorithms and Applications,* Kluwer Academic Publishers, Norwell, MA., 1984.

17. Ahmed, N., Solon, D.L., Hummels, D.R., and Parikh, D.D., "Sequential regression considerations of adaptive filtering," *Electron. Lett.*, vol. 13, pp. 446-447, July 1977..

18. Parikh, D. and Ahmed, N., "A sequential regression algorithm for recursive filters," *Electron. Lett.*, vol. 14, pp. 266-268, April 1978.

19. Johnson, C.J., Larimore, M.G., Treichler, J.R., and Anderson, B.D.O, "SHARF Convergence Properties," *IEEE Trans. Circ. Syst.*, vol. CAS-28, pp. 499-510, June 1981.

20. Larimore, M.G., Treichler, J.R., and Johnson, C.R., "SHARF:An algorithm for adapting IIR digital filters," *IEEE Trans. Audio Speech Signal Process.*, vol. ASSP-28, pp. 1622-1624, Nov 1976.

21. Feintuch, P.L., "An adaptive recursive LMS filter," *Proc. IEEE*, vol. 64(3), pp. 1622-1624, Nov 1976.

22. Hecht-Nielson, R., *Neurocomputing,* Addison-Wesley, Reading, Mass., 1990.

23. Ruck, D., Rogers, S.K., Kabrinsky, M., Oxley, M.E., and Suter, B.W., "The multilayer perceptron as an approximation to the Bayes optimal discriminant function," *IEEE Trans. Neural Net.*, vol. 1, pp. 296-298, Dec 1990.

24. Gallant, S.I., "Preceptron based learning algorithms," *IEEE Trans. Neural Net.*, vol. 1, pp. 171-191, June 1990.

25. Shynk, J., "Performance surface of a single layer perceptron," *IEEE Trans. Neural Net.*, vol. 1, pp. 268-274, Sept 1990.

26. Watterson, J.W., "An optimal multilayer perceptron neural receiver for signal detection," *J IEEE Trans. Neural Net.*, vol. 1 , pp. 298-300, Dec 1990.

27. Barto, A., Sutton, R., and Anderson, C., "Neuron-like adaptive elements that can solve difficult learning control problems," *IEEE Trans. Syst. Man & Cybernet.*, vol. SMC-13, pp. 834-846, 1983.

28. Leitmann, G., *Mathematics in Science and Engineering vol 5: Optimization Techniques,* Academic Press, New York, 1962.

29. Leitmann, G., *Mathematics in Science and Engineering vol 31 : Topics in Optimization,* 31, Academic Press, New York, 1967.

30. Luenberger, D., *Optimization by Vector Space methods,* J. Wiley & Sons, New York, 1969.

31. Pontryagin, L.S., Boltyanskii, V.G., Gamkrelidze, R.V., and Mishcehenko, E.F., *The mathematical theory of optimal processes (English translation from Russian),* J. Wiley & Sons., New York, 1962.

32. Widrow, B. and McCool, J., "A comparison of adaptive algorithms based on the methods of steepest descent and random search," *IEEE. Trans. Ant. Propag.*, vol. AP-24, pp. 615-637, Sept. 1976.

33. Darwin, C., *The descent of man and selection in relation to sex,* Murray, London, 1882.

34. Darwin, C., *On the origin of species,* Murray, London, November 1859.

35. Widrow, B. and et, al, "Adaptive Noise Cancellation:Principles and Applications," *Proc. IEEE*, vol. 63, pp. 1692-1716, Dec. 1975.

36. Clark, G.A., Mitra, S.K., and Parker, S.R., "Block Implementations of adaptive digital filters," *IEEE Trans. Circ. Syst.*, vol. CAS-28, pp. 584-592, June

1981.

37. Widrow, B. and et, al, "Stationary and Non-Stationary learning Characteristics of the LMS adaptive Filter," *Proc. IEEE*, vol. 64, pp. 1151-1162, 1976.

38. Lawson, C.L. and Hanson, R.J., *Solving Least Squares Problems,* Prentice-Hall, 1974.

39. Godard, D., "Channel Equalisation using a Kalman Filter for Fast Data Transmission," *IBM J. Res. Develop.*, vol. 18(3), pp. 267-273, May 1974.

40. Cowan, C.F.N., "Performance comparison of finite linear adaptive filters," *IEE Proceedings Part F*, vol. 134(3), pp. 211-216, June 1987.

41. Faden, D., "Tracking properties of adaptive signal processing algorithms," *IEEE Trans. Acoust. Speech Signal Process.*, vol. ASSP-29, p. 439, June 1981.

42. Bershad, N. and Macchi, O., "Comparison of RLS and LMS algorithms for tracking a chirped signal," *Proc. ICASSP 89*, Glasgow, May, 1989.

43. McLaughlin, S., "Adaptive Estimation and Equalisation of the High Frequency communications channel," *Ph.D. Thesis*, vol. University of Edinburgh, 1990.

44. Cioffi, J. M., "Limited Precision Effects in Adaptive Filtering," *IEEE Trans. Circuits Syst.*, vol. CAS-34 No 7., pp. 821 - 833, 1987.

45. Lucky, R., "Automatic equalisation for digital communication," *Bell Syst. Tech J.*, vol. 44, pp. 547-588, April 1965.

46. Gersho, A., "Adaptive equalisation of highly dispersive channels for data transmission," *Bell Syst. Tech. J.*, vol. 48, pp. 55-70, Jan 1969.

47. Proakis, J.G. and Miller, J.H., "An adaptive receiver for digital signaling through channels with inter-symbol interference," *IEEE Trans. Info. Theory*, vol. IT-15, pp. 484-497, June 1967.

48. Proakis, J.G., *Digital Communications,* McGraw-Hill, Singapore, 1983.

49. Stremler, F.G., *Introduction to Communications Systems,* Addison Wesley, Reading, Mass., 1982.

50. Duttweiler, D.L., "A twelve channel digital echo canceller," *IEEE Trans. Comm.*, vol. COMM-26, pp. 647-653, May 1978.

51. Glover, J.R., "Adaptive noise cancelling applied to sinusoidal interferences," *IEEE Trans. Acoust. Speech Signal Process.*, vol. ASSP-35, pp. 481-491, Dec 1977.

52. Atal, B.S. and Schroeder, M.R., "Adaptive predictive coding of speech signals," *Bell Syst. Tech J.*, vol. 49, pp. 1973-1986, Oct 1970.

53. Atal, B.S. and Schroeder, M.R., "Predictive coding of speech and subjective error criteria," *IEEE Trans Acoust. Speech Signal Process.*, vol. ASSP-27, pp. 247-254, June 1979.

54. Asher, R.B., Andrisani, D., and Dorato, P., "Bibliography on Adaptive Control Systems," *Proc. IEEE*, vol. 64, p. 1266, August 1976.

55. Mishkin, E. and Braun, L., *Adaptive Control Systems,* McGraw-Hill, New York, 1961.

56. Belman, R., *Adaptive Control Processes:A guided tour,* Princeton University Press, Princeton, NJ, 1961.

57. Sworder, D., *Optimal Adaptive Control Systems,* Academic Press, New York, 1966.

58. Anstorom, K.J. and Wittenmark, B., *Adaptive Control,* Addison-Wesley, Reading, Mass., 1989.

59. Honig, M.L., "Echo cancellation of voiceband data signals using RLS and stochastic gradient algorithms," *IEEE Trans Comm.*, vol. COMM-33(1), pp. 65-73, Jan 1984.

60. Hunta, J.C. and Webster, J.G., "60Hz interference in electrocardiography," *IEEE Trans. Biomed. Eng.*, vol. BME-20, pp. 91-101, March 1973.

61. Harrison, W.A., Lim, J.S., and Singer, E., "A new application of adaptive noise cancellation," *IEEE Trans. Acoust Speech Signal Process.*, vol. ASSP-34, pp. 21-27, Jan 1986.

62. Forney, G.R., "Maximum Likelihood Sequence Estimation of digital sequences in the presence of intersymbol interference," *IEEE Trans. Info. Theory*, vol. IT-18, pp. 363-387, May 1972.

63. Nissen, C.W., "Automatic channel equalisation algorithms," *Proc. IEEE*, vol. 55, p. 698, May 1967.

64. Proakis, J.G., "Adaptive digital filters for equalisation of telephone channels," *IEEE Trans. Audio Electroacoust.*, vol. AU-18, pp. 195-200, June 1970.

65. Sorenson, H.W., "Least-squares estimation from Gauss to Kalman," *IEEE Spectrum*, vol. 7, pp. 63-68, July, 1970.

66. Horowitz, L.L. and Senne, K.D., "Performance advantage of complex LMS for controlling narrow band adaptive arrays," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-29, pp. 722-736, June 1981..

67. Ljung, L., "Convergence of Recursive Estimators," *Proceedings 5th IFAC Symposium on identification and system parameter identification*, Darmstadt, 1979.

68. Ljung, L., "Analysis of a generalised recursive prediction algorithm," *Automatica*, vol. 17, pp. 88-99, Jan 1981.

69. Ljung, L. and Sodstorm, T., *Theory and Practice of recursive identification.*

70. Falconer, D.D. and Ljung, L., "Application of Fast Kalman estimation to adaptive equalisation," *IEEE Trans. Comm.*, vol. COMM-26, pp. 1439-1446, Oct 1978.

71. Morf, M., Dickinson, B., Kailaith, T., and Vieira, A., "Efficient Solutions of Covariance Equations for Linear Prediction," *IEEE Trans. Acoust. Speech, Signal Process.*, vol. ASSP-25, pp. 429-435, Oct 1977.

72. Samson, C., "A unified treatment of Fast Kalman algorithms for identification," *Int. J. Control.*, vol. 31, pp. 909-934, May, 1982.

73. Kalouptsidis, N., Carayannis, G., and Manolakis, D., "A Fast Covariance Algorithm for Sequential Least Squares Filtering and Prediction," *IEEE Trans. Auto. Control*, vol. AC-29, pp. 752-755, Aug. 1984.

74. Kalouptsidis, N., Carayannis, G., and Manolakis, D., "A fast covariance type algorithm for sequential least squares filtering and prediction," *Proc. IEEE Conf. Decision Contr.*, San Antonio, 1983.

75. Carayannis, G., Manolakis, D., and Kalouptsidis, N., "Fast Kalman type algorithms for sequential signal processing," *Proc ICASSP 83*, Boston, April 1983.

76. Carayannis, G., Kalouptsidis, N., and Manolakis, D., "Fast recursive algorithms for a class of linear equations," *IEEE Trans Acoust. Speech and Signal Process.*, vol. ASSP-20, pp. 227-239, April 1982.

77. Cioffi, J.M. and Kailath, K., "Windowed Fast Transversal Filter Adaptive Algorithms with Normalisation," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-33, pp. 607-625, June 1985.

78. Cioffi, J.M., "The Block Processing FTF adaptive algorithm," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-34, pp. 77-90, Feb 1986.

79. Wilkinson, J.H., *Rounding Errors in Algebraic Processes,* H.M. Stationary Office, London, 1963.

80. Barnes, C.W., Tran, B.N., and Leung, S.H., "On the statistics of fixed point roundoff error," *IEEE Trans Acoust. Speech Signal Process.*, vol. ASSP-33, pp. 595-606, June 1985.

81. Mulcahy, L.P., "On fixed point roundoff error analysis," *IEEE Trans. Acoust. Speech Signal Process.*, vol. ASSP-37, p. 1623, October, 1989.

82. Press, W.H., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T., *Numerical Recipes in C - The art of scientific computing,* Cambridge University Press, 1988.

83. Levinson, N., "The Wiener RMS (Root Mean Square) Error Criterion in Filter Design and Prediction," *J. Math. Phys.*, vol. Vol 25, pp. 261-278, Jan 1947.

84. Kailaith, T., "A view of three decades of linear filtering theory," *IEEE Trans. Info. Theory.*, vol. IT-20, pp. 146-181, March 1974.

85. Broyden, C.G., *Basic Matrices,* Macmillan Press, London, 1975.

86. Astorom, K.J. and Wittenmark, B., *Adaptive Control,* Addison Wesley, Reading, Mass., 1989.

87. Mendel, J.M., *Discrete Techniques of parameter estimation,* Marcel Dekker, New York, 1973.

88. Westlare, J.R., *A handbook of numerical matrix inversion and solution of linear equations,* John Wiley, New York, 1968.

89. Ayres, F., *Theory and problems of matrices,* McGraw Hill, New York, 1962.

90. *Second Generation TMS320 Users Guide,* Texas Instruments, Dallas, Texas, 1988.

91. Papamichalis, P. and Simar, R., "The TMS320C30 digital signal processor," *IEEE Micro Magazine*, vol. 8, pp. 13-29, Dec 1988.

92. Fuccio, M.L and et, al, "The DSP32C : AT&T's second generation floating point digital signal processor," *IEEE Micro Magazine*, vol. 8, pp. 30-48, Dec 1988.

93. Sohie, G.R.L. and Klonker, K.L., "A digital signal processor with IEEE floating point arithmetic," *IEEE Micro Magazine*, vol. 8 , pp. 49-67, Dec 1988.

94. Klonker, K.L., "The Motorola DSP56000 digital signal processor," *IEEE Micro Magazine*, vol. 6, pp. 29-48, Dec 1986.

95. Lin, K.S., Frantz, G.A., and Simar, R., "The TMS320 family of digital signal processors," *Proc. IEEE*, vol. 75, pp. 1143-1159, September 1987.

96. *TMS320 Family Development Support,* Texas Instruments, Dallas, Texas.

97. Lin, D.W., "On the Digital Implementation of the Fast Kalman Algorithm," *IEEE. Trans. Acoust. Speech Signal Process.*, vol. ASSP-32, pp. 998-1005,

1984.

98. Lee, D.T.L, Morf, M., and Friedlander, B., "Recursive Least Squares Ladder Estimation Algorithms," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-29(3), pp. 627-641, June 1981.

99. Porat, B., Friedlander, B., and Morf, M., "Square Root Covariance Ladder Algorithms," *IEEE Trans. Automatic Control*, vol. AC-27(4), pp. 813-829, August 1982.

100. Shensa, M.J., "Recursive Least Squares Lattice Algorithms - a Geometric Approach," *IEEE Trans. Automatic Control*, vol. AC-26(3), pp. 675-702, June 1981.

101. Ling, F., Manolakis, D., and Proakis, J.G., "Numerically Robust Least Squares Lattice Ladder Algorithms with Direct Updating of the Reflection Coefficients," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-34(4), pp. 837-845, Aug 1986.

102. Ling, F. and Proakis, J.G., "A generalised multi-channel LS lattice algorithm based on sequential processing stages," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-32, pp. 381-389, April 1984.

103. Cioffi, J.M., "The fast update adaptive rotors RLS algorithm," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-38, pp. 631-653, April 1990.

104. Ling, F., "Efficient least squares lattice algorithms based on Givens rotation with systolic array implementation," *Proc. ICASSP'89*, Glasgow, May 1989..

105. Gentleman, W.M., "Least squares computation by Givens transformation without square roots," *J. Inst. Maths. Applications*, vol. 12, pp. 329-336, 1973.

106. Hariharan, S. and Clark, A.P., "HF Channel estimation using a Fast Transversal Filter Algorithm," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-38, pp. 1353-1362, Aug 1990.

107. Kim, D. and Alexander, W.E., "Stability Analysis of the Fast RLS Adaptation Algorithm," *Proceedings ICASSP 88*, vol. 3, pp. 1361-1364, New York, April 1988.

108. Benallal, A. and Gilloire, A., "A New Method to Stabilize Fast RLS Algorithms based on a First Order Model of the Propagation of Numerical Errors," *Proceedings ICASSP 88*, vol. 3, pp. 1373-1376, New York, April 1988.

109. Slock, D.T. and Kailath, T., "Numerically Stable Fast Recursive Least Squares Transversal Filters," *Proceedings ICASSP 88*, vol. 3, pp. 1365-1368, New York, April 1988.

110. Botto, J.L. and Moustakides, G.V., "Stabilising the Fast Kalman Algorithm," *IEEE Trans. Acoust., Speech, Signal Process*, vol. ASSP-38, pp. 1342-1348, Sept 1989.

111. Moore, R. E., *Interval Analysis*, Prentice-Hall, Englewood Cliffs, 1966.

112. Callender, C.P. and Cowan, C.F.N., "Numerically Robust Implementations of Fast RLS Adaptive Algorithms using Interval Arithmetic," *Proceedings EUSIPCO 90*, pp. 173-176, Barcelona, Sept. 1990.

113. Callender, C.P. and Cowan, C.F.N., "Numerically Stable Fast Recursive Least Squares Algorithms for Adaptive Filtering using Interval Arithmetic," *Proceedings 10th IEE Saraga Colloquium on Digital and Analogue filters and filtering systems*, pp. 5/1 - 5/3, London, May 1990.

114. Callender, C.P. and Cowan, C.F.N., "Numerically robust implementations of the Fast RLS adaptive algorithms using Interval Arithmetic," *Signal Processing*, Submitted Jan 1991.

115. Knuth, D.E., *The Art of Computer Programming*, 2:Seminumerical Algorithms, p. Chapter 4, Addison Wesley, Reading, Mass, 1969.

116. Kulish, U.W. and Miranker, W.L., *A New Approach to Scientific Computation*, New York Academic Press, New York, 1983.

117. Gibb, A., "Procedures for Range Arithmetic (Algorithm 61)," *Comm. Assoc. Comp. Mach.*, vol. 4:7, pp. 319 - 320, 1961.

118. Kailath(ed), T., "Special Issue on system identification and time series analysis," *IEEE Trans. Automat. Contr.*, vol. AC-19, Dec 1974.

119. Luders, G. and Narenda, K.S., "Stable adaptive schemes for state estimation and identification of linear systems," *IEEE Trans. Automat. Contr.*, vol. AC-19, Dec 1974.

120. Watterston, C.C. and et, al, "Experimental confirmation of an HF channel model," *IEEE Trans. Communications*, vol. COM-18, pp. 792-803, 1970.

121. Shaver, and et, al, "Evaluation of a Gaussian HF channel model," *IEEE Trans. Communications*, vol. COM-18, pp. 77-88, 1967.

122. Shepherd, R.A. and Lomax, J.B., "Frequency spread in ionospheric radio propagation," *IEEE Trans. Communications*, vol. COM-18, pp. 268-275, 1967.

123. Ehrman, L., Bates, L.B., Eschle, J.F., and Kates, J.M., "Real time software simulation of the HF radio channel," *IEEE Trans. Communications*, vol. COM-30, pp. 1809-1817, Aug 1980.

124. Lin(ed.), K.S., *Digital signal processing applications with the TMS320 family,* Prentice Hall/Texas Instruments, Englewood Cliffs, N.J., 1987.

125. Aliphas, A. and Feldman, J.A., "The versatility of digital signal processing chips," *IEEE Spectrum*, vol. 24, pp. 40-45, June 1987.

126. Allen, J., "Computer architecture for digital signal processing," *Proc. IEEE*, vol. 73, pp. 852-873, 1986.

127. Macdonald, A., "Real Time Hardware Simulator based on the TMS320C25 digital signal processor," *B.Eng(Hons) Project Report HSP743*, May, 1990.

128. Horowitz, P. and Hill, W., *The art of electronics,* Cambridge University Press, Cambridge, 1980.

129. Grant, P.M., "The DTI-industry sponsored silicon architectures research initiative," *IEE Electronics & Communication Engineering Journal*, pp. 102-108, June 1990.

130. *Sage User Manual,* Silicon Architectures Research Initiative, Edinburgh, 1990.

131. Lipsett, R., Schaefer, C., and Ussery, C., *VHDL: Hardware Description and Design,* Kluwer Academic Publishers, Norwell, Mass., 1989.

132. *The Standard VHDL 1076 Support Environment (VSE) User's Manual - Version 2.1 (Unix),* Intermetrics Inc., IR-MD-124, Feb 1990.

Appendix A

Original Publications

[1] Callender, C.P and Cowan, C.F.N. "Numerically Robust Implementations of Fast RLS Adaptive Algorithms using Interval Arithmetic" pp173-176, Proceedings EUSIPCO 90, Barcelona, 1990.

[2] Callender, C.P. and Cowan C.F.N. "Numerically robust implementations of the Fast RLS adaptive algorithms using Interval Arithmetic" pp5/1-5/3, Colloquium Digest, 10th IEE Saraga Colloquium on Digital and Analogue filters and filtering systems, London, May 1990.

[3] Callender, C.P. and Cowan C.F.N. "Numerically robust implementations of the Fast RLS adaptive algorithms using Interval Arithmetic", submitted to Signal Processing, January 1991.

SIGNAL PROCESSING V: Theories and Applications
L. Torres, E. Masgrau, and M.A. Lagunas (eds.)

Numerically Robust Implementations of Fast RLS Adaptive Algorithms using Interval Arithmetic

Christopher P. Callender

Colin F.N. Cowan

Department of Electrical Engineering,
University of Edinburgh,
Edinburgh EH9 3JL,
Scotland, UK.

Abstract

In this paper, a new approach is presented to the stabilisation of the Fast Recursive Least Squares adaptive filter algorithms. Using the new method, the accumulation of numerical errors is monitored by using interval arithmetic, rather than real number arithmetic to perform all the computations. If the numerical error is found to be too large, then the algorithm is reinitialised to prevent divergence. Results demonstrate the stable performance of the Fast Transversal Filters (FTF) algorithm, using both floating point and fixed point interval arithmetic.

1. Introduction

Fast RLS (Recursive Least Squares) algorithms for adaptive filtering, such as the FTF algorithm [1], the FAEST algorithm [2], and the Fast Kalman Algorithm [3], offer the same rapid initial convergence properties as the standard RLS algorithm [4], but offer low computational requirements. The fast algorithms are characterised by requiring $O(N)$ additions and multiplies per time sample, as compared with $O(N^2)$ for standard RLS. Their low computational load is of the same order as the popular Least Mean Squares (LMS) algorithm [5].

The reason that they have not achieved the same widespread application in high speed real time systems is that they suffer from numerical instability when implemented on either a fixed or floating point processor. Small numerical errors accumulate at every iteration of the algorithm [6] until the solution diverges, often very rapidly, from that which is correct.

Attempts to stabilise the algorithms have been proposed with varying degrees of success. Basically, the stabilisation procedures can be divided into two categories - those in which the algorithm is reinitialised [7] and those in which modifications are made to the algorithm [8].

Reinitialisation involves resetting certain internal variables, hopefully before divergence occurs. The solution of the adaptive filter just before reinitialisation is passed forward using a soft constraint, so that the adaptive algorithm does not need to reconverge after it is reset. A design constant, normally denoted by μ controls the balance between the initial soft-constraint and the least squares solution. The difficulty is to predict when the algorithm requires reinitialisation. It is this problem which is addressed by the new methods introduced in this paper.

Modifications to the algorithms typically take the form of introducing 'leakage' factors into the update equations in order to prevent the otherwise unconstrained growth of numbers due to limited precision arithmetic. Problems with this method are that computational efficiency is reduced, biases are introduced to the solution, and it is often difficult to prove that the modifications provide a sufficient condition for stability.

In this paper, a new form of arithmetic is presented [9], which enables an error analysis to be performed whilst the algorithm is running. If the results of this analysis indicate that numerical problems are becoming significant, the algorithm automatically reinitialises.

2. Theory

2.1. Interval Arithmetic (Infinite Precision)

An interval number is a range of real numbers, bounded by lower and upper endpoints. The notation used is to write an interval in the form [a,b]

which is taken to mean

$$[a,b]=\left\{x \mid a \le x \le b, x \in R\right\} \quad (1)$$

Thus [a,b] means all real numbers which lie between a and b.

Arithmetic operations on intervals are then defined by:

$$[a,b]+[c,d]=[a+c,b+d] \quad (2a)$$

$$[a,b]+[c,d]=[a-d,b-c] \quad (2b)$$

$$[a,b].[c,d]=[min(ac,ad,bc,bd), \quad (2c)$$
$$max(ac,ad,bc,bd)]\dagger$$

$$[a,b]/[c,d]=\left[a,b\right].\left[\frac{1}{d},\frac{1}{c}\right] \quad (2d)$$

unless 0 $\in$ [c,d] in which case division is undefined.

† More efficient methods for interval multiplication exist. The signs of a,b,c and d are examined and normally only two real multiplies are then performed [10].

2.2. Interval Arithmetic (Finite Precision)

All real variables in the Fast RLS algorithms may be replaced by intervals in such a way as to ensure that the interval contains the *exact* value of the variable. The way in which this is performed is processor dependent. Arithmetic is implemented using equations (2a) - (2d), but care is taken over the direction of rounding of the endpoints, to guarantee that the finite precision interval contains the whole of the infinite precision interval, and often slightly more.

If this is done, the filter taps will also become intervals. The difference between the upper and lower endpoints, or width of the interval represents the extent to which finite precision errors have been accumulated. If any of these widths are too large the algorithm may be reinitialised. It is also necessary to reinitialise the algorithm if any division is attempted in which the divisor is an interval which contains zero.

3. Computational Efficiency

It is clear from equation (2) that the computational requirement for each interval operation is 2 real operations, except for multiplication. The multiplication algorithm of [10] normally requires only 2 real multiplies, but in one case 4 are necessary. The computation of lower and upper endpoints may, however, be shared between two processors, resulting in the same speed of operation as for non-interval Fast RLS, but with increased hardware complexity.

4. Choice of Design Parameters

The choice of the design parameters, ρ, the maximum difference between the upper and lower endpoint of each filter coefficient, and μ, the parameter for the soft-constrained reinitialisation of the algorithm may be chosen as:

$$\rho^2 < MMSE \quad (3)$$

where MMSE is the minimum mean square error for the adaptive filter, and

$$\mu = \frac{MMSE\ \lambda}{N\rho^2(1-\lambda)} \quad (4)$$

where λ is the RLS forgetting factor, and N is the filter length.

5. Results

All simulations involved using the FTF algorithm to perform system identification (Figure 1). The 'unknown' system was an FIR filter of length 5, and both it and the adaptive filter were excited by coloured Gaussian noise (eigenvalue ratio ~ 20). The output from the 'unknown' system was corrupted by small amounts of additive white Gaussian noise. This signal was the desired response input to the adaptive system.

In all graphs, the tap weights were found by taking the mean of the upper and lower endpoints of the tap intervals. The norm of the tap error vector was then calculated. This was converted to a dB scale.

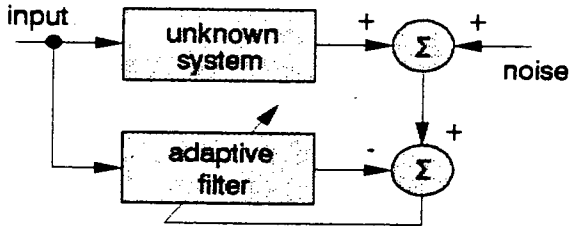
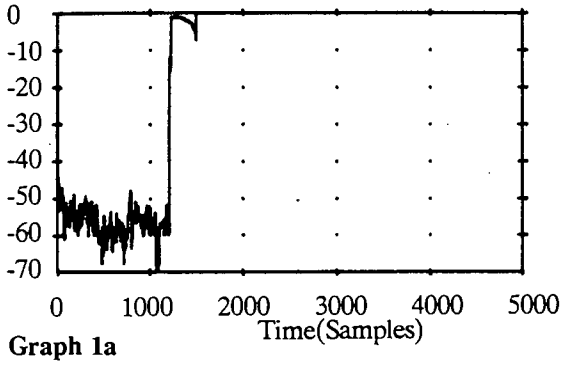


Figure 1: System identification using the FTF adaptive algorithm

Graph 1a and 1b show the performance of the FTF algorithm, with no form of stabilisation using 64 bit floating point and 16 bit fixed point arithmetic respectively. The fixed point simulations used a 32 bit long accumulator for intermediate results, as is common on many 16 bit DSP chips.

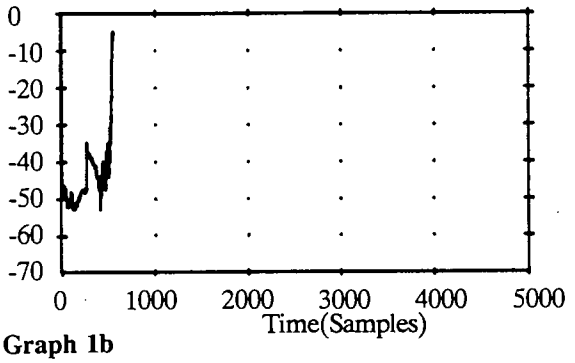
Norm Tap Error (dB) - Floating Point



Graph 1a

No rescues
λ=0.98, SNR=45dB, 5,000 iterations.

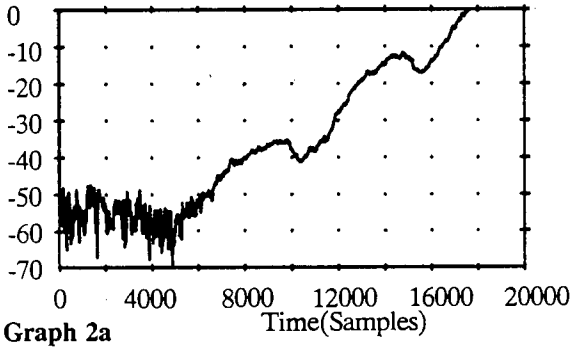
Norm Tap Error (dB) - Fixed Point



Graph 1b

No rescues
λ=0.98, SNR=45dB, 5,000 iterations.

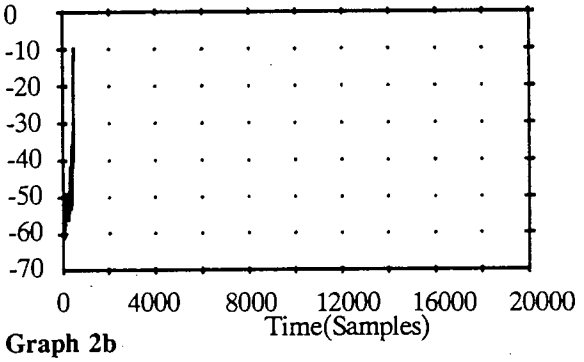
Norm Tap Error (dB) - Floating Point



Graph 2a

Rescued if rescue variable is negative [1].
λ=0.98, μ=1.0, SNR=45dB, 20,000 iterations.

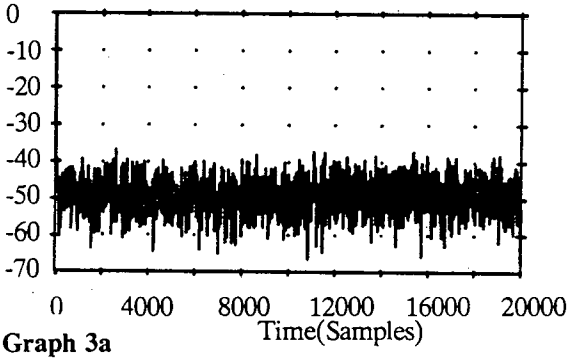
Norm Tap Error (dB) - Fixed Point



Graph 2b

Rescued if rescue variable is negative [1].
λ=0.98, μ=0.5, SNR=45dB, 20,000 iterations.

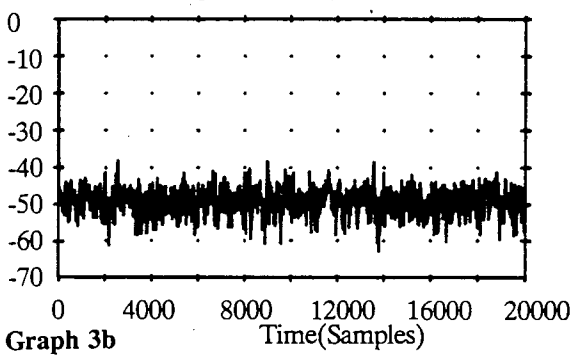
Norm Tap Error (dB) - Floating Point



Graph 3a

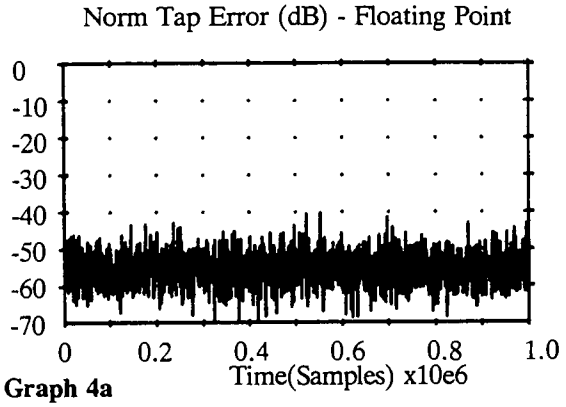
Interval FTF
ρ=0.005, λ=0.98, μ=1.0, SNR=45dB, 20,000 iterations.

Norm Tap Error (dB) - Fixed Point



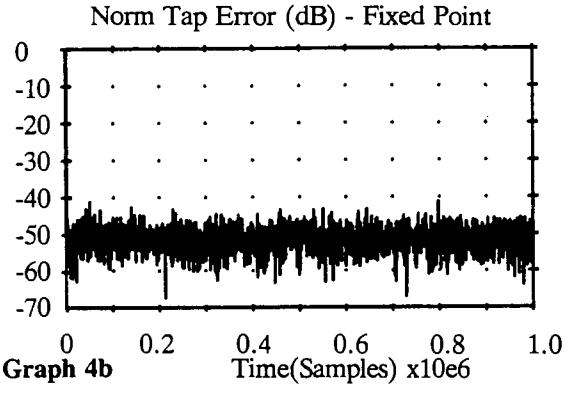
Graph 3b

Interval FTF
ρ=0.005, λ=0.98, μ=0.5, SNR=45dB, 20,000 iterations.



Graph 4a

Long term performance of Interval FTF. One million iterations (every 500th point plotted). ρ=0.005, λ=0.98, μ=1.0, SNR=45dB.



Graph 4b

Long term performance of Interval FTF. One million iterations (every 500th point plotted). 0.005, λ=0.98, μ=0.5, SNR=45dB.

Graphs 2a and 2b demonstrate the use of the rescue method outlined[1] to stabilise the algorithm, again using 64 bit floating point and 16 bit fixed point numbers. Little improvement is apparent on the 16 bit results, and even the 64 bit floating point version eventually diverges.

Graphs 3a and 3b show the interval method, which was applied by reinitialising if the width of any of the tap intervals was greater than 0.005. The performance of the 16 bit fixed point algorithm and the 64 bit floating point algorithm is almost identical.

Finally graphs 4a and 4b illustrate the long term stability of the interval method using both floating and fixed point arithmetic for one million iterations.

Other simulations have demonstrated the successful application of interval techniques to the Fast Kalman algorithm.

5. Conclusions

Interval arithmetic provides a way to monitor the accumulation of numerical errors. This may be used to reinitialise the Fast RLS algorithms before divergence occurs, yielding numerically stable performance.

The increased computation of the interval methods is a disadvantage, but the number of operations is still proportional to the filter length.

References

1. Cioffi, John M. and Kailath, Thomas, "Fast, Recursive Least Squares Transversal Filters for Adaptive Filtering," *IEEE Trans. Acous, Speech, Signal Process.*, vol. ASSP-32, No 2., pp. 304 - 337, 1984.
2. Carayannis, George, Manolakis, Dimitris G., and Kalouptsidis, Nicholas, "A Fast Sequential Algorithm for Least Squares Filtering and Prediction," *IEEE Trans. Acous, Speech, Signal Process.*, vol. ASSP-31, No 6., pp. 1394 - 1402, 1983.
3. Ljung, Lennart, Morf, Martin, and Falconer, David, "Fast calculation of gain matrices for recursive estimation schemes," *Int. J. Control*, vol. 27, pp. 1 - 19, 1978.
4. Cowan, Colin F.N. and Grant, Peter M., *Adaptive Filters*, Prentice Hall, Englewood Cliffs, 1985.
5. Widrow, B. and Stearns, S., *Adaptive Signal Processing*, Prentice-Hall, Englewood Cliffs, 1985.
6. Cioffi, John M., "Limited Precision Effects in Adaptive Filtering," *IEEE Trans. Circuits Syst.*, vol. CAS-34 No 7., pp. 821 - 833, 1987.
7. Lin, D.W., "On the Digital Implementation of the Fast Kalman Algorithm," *IEEE. Trans. Acoust. Speech Signal Process.*, vol. ASSP-32, pp. 98-1005, 1984.
8. Slock, D.T.M. and Kailath, T., "Numerically Stable Fast Recursive Least Squares Transversal Filters," *Proc. ICASSP 88 Conf.*, vol. 3, pp. 1365 -1368, 1988.
9. Moore, Ramon E., *Interval Analysis*, Prentice-Hall, Englewood Cliffs, 1966.
10. Gibb, Allan, "Procedures for Range Arithmetic (Algorithm 61)," *Comm. Assoc. Comp. Mach.*, vol. 4:7, pp. 319 - 320, 1961.

Numerically Stable Fast Recursive Least Squares Algorithms for Adaptive Filtering using Interval Arithmetic

Christopher P. Callender[†]

Colin F.N. Cowan[†]

Introduction

Fast Recursive Least Squares algorithms such as the Fast Kalman algorithm[1], the FAEST algorithm[2], and the FTF algorithm[3] perform least squares adaptive filtering with low computational complexity, which is directly proportional to the filter length. Unfortunately, these highly efficient algorithms suffer from severe numerical instability when implemented using either fixed or floating point digital arithmetic. Small numerical errors due to the finite precision of the computations at each iteration of the algorithm are propagated and accumulate[4]. Eventually the algorithm diverges from the correct solution, often very suddenly. In this paper a new approach is used to perform stabilisation. Interval Arithmetic[5] is used to provide an upper and a lower bound to the solution produced by the adaptive algorithm, allowing for the possible effects of finite precision arithmetic. If the difference between the upper and the lower bounds becomes excessively large, then the Fast RLS algorithm may be reinitialised[6], preventing divergence.

Interval Numbers and Interval Arithmetic

An interval number is simply a range of real numbers. An interval number may be written in the form [a,b], which is taken to mean all real numbers between lower endpoint a and upper endpoint b, or

$$[a,b]=\left\{x \mid a \le x \le b, x \in R\right\} \tag{1}$$

Having defined what is meant by an interval number, we may now proceed to define the arithmetic operations $\div$, $-$, $*$ and $\div$ for the interval number system.

$$[a,b]+[c,d]=[a+c,b+d] \tag{2a}$$

$$[a,b]-[c,d]=[a-d,b-c] \tag{2b}$$

$$[a,b]*[c,d]=[min(ac,ad,bc,bd),max(ac,ad,bc,bd)] \tag{2c}$$

$$[a,b]\div[c,d]=[a,b]*\left[\frac{1}{d},\frac{1}{c}\right] \tag{2d}$$

unless $0 \in [c,d]$ in which case the results of division are undefined.

These operation may be implemented on a digital processor, provided that care is taken over the rounding directions of the calculated results[5]. Lower endpoints should be rounded in the direction of $-\infty$ and upper endpoints in the direction of $+\infty$. More efficient methods of interval multiplication and division exist, which give the same results as equations (2c) and (2d)[7].

Application to the FTF algorithm

To use interval arithmetic with the FTF algorithm[3], every number in the algorithm is converted to an interval number and every arithmetic operation is converted to an interval operation. When this is done, the filter coefficients calculated by the FTF algorithm also become intervals, and the difference between the upper and lower endpoints of each of these coefficients represents the extent to which the solution has accumulated numerical error. Real valued filter inputs may be represented by degenerate intervals of the form [a,a] and for the purposes of obtaining real valued outputs, the centre of the interval given by

$$centre([a,b])=\frac{1}{2}\left(a+b\right) \tag{3}$$

[†]Department of Electrical Engineering, Univeristy of Edinburgh, Edinburgh EH9 3JL.

may be used.

To make the algorithm numerically stable, reinitialisation using a soft constrained initial solution[36] must be performed if any of the differences between the upper and lower endpoints of the filter coefficients exceeds a design constant denoted by ρ or if division is attempted by an interval [c,d] such that $0 \in [c,d]$.

Results

To illustrate the stability of the interval FTF algorithm, an adaptive system identification experiment was performed. A noise sequence was input to a FIR filter with unknown coefficients and the response of this filter used to train the adaptive filter using the FTF algorithm.

The norm of the filter coefficient error vector in deciBels was plotted against time to illustrate the performance of the adaptive system. In all simulations, 16 bit fixed point arithmetic was used, with the provision of a 32 bit long accumulator. The signal to noise ratio was 40dB.

Figure 2 illustrates the divergence of the algorithm using non interval arithmetic after only a few hundred iterations

Figure 3 illustrates the stable performance of the FTF algorithm using interval arithmetic. A good solution is obtained for the entire duration of the simulation.

Figure 4 illustrates the long term performance of the algorithm. One million iterations were performed, and the error plotted on every 500th iteration. A stable solution is again obtained for the whole simulation.

Other simulation results have demonstrated that the technique is equally applicable to floating point digital arithmetic, and to other Fast RLS adaptive algorithms such as the Fast Kalman algorithm[1].

Conclusions

Interval arithmetic provides a method for monitoring the accumulaion of numerical error, and determining whether a Fast RLS adaptive algorithm requires reinitialisation. The interval algorithms have a computational complexity which is approximately double that of their non-interval counterparts, but which still remains directly proportional to the filter length.

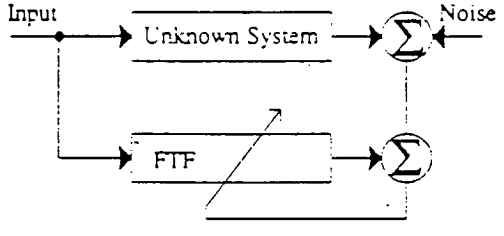


Figure 1:Simulation configuration for adaptive system identification.

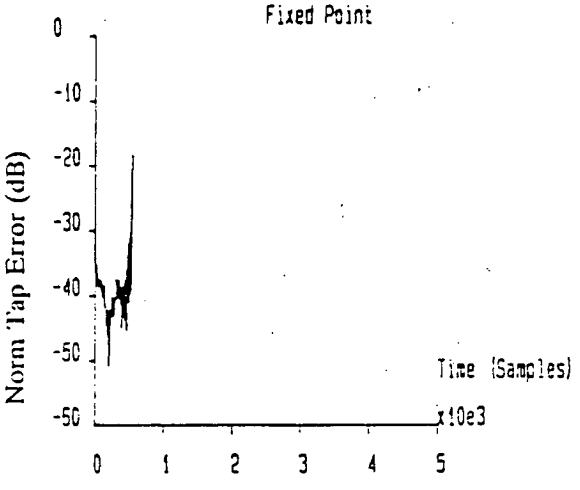


Figure 2:Divergence of FTF algorithm $\lambda = 0.98$, SNR = 40dB

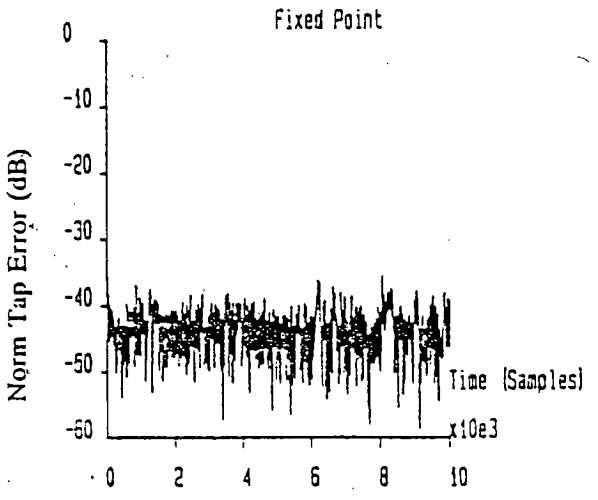


Figure 3:Interval FTF
$\rho = 0.002$, $\lambda = 0.98$, $\mu = 0.25$, SNR = 40dB,
10,000 iterations.

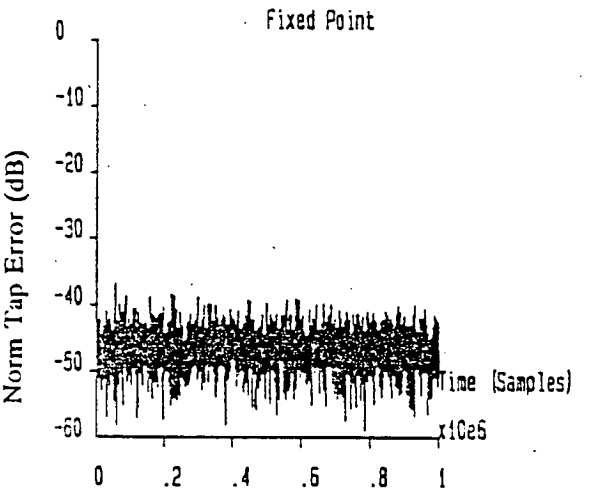


Figure 4:Interval FTF
$\rho = 0.002$, $\lambda = 0.98$, $\mu = 0.25$, SNR = 40dB,
1,000,000 iterations.

References

1. Ljung, L., Morf, M., and Falconer, D., "Fast calculation of gain matrices for recursive estimation schemes," *Int. J. Control*, vol. 27, pp. 1 - 19, 1978.
2. Carayannis, G., Manolakis, D. G., and Kalouptsidis, N., "A Fast Sequential Algorithm for Least Squares Filtering and Prediction," *IEEE Trans. Acous, Speech, Signal Process.*, vol. ASSP-31, No 6., pp. 1394 - 1402, 1983.
3. Cioffi, J. M. and Kailath, T., "Fast, Recursive Least Squares Transversal Filters for Adaptive Filtering," *IEEE Trans. Acous, Speech, Signal Process.*, vol. ASSP-32, No 2., pp. 304 - 337, 1984.
4. Cioffi, J. M., "Limited Precision Effects in Adaptive Filtering," *IEEE Trans. Circuits Syst.*, vol. CAS-34 No 7., pp. 821 - 833, 1987.
5. Moore, R. E., *Interval Analysis*, Prentice-Hall, Englewood Cliffs, 1966.
6. Lin, D.W., "On the Digital Implementation of the Fast Kalman Algorithm," *IEEE. Trans. Acoust. Speech Signal Process.*, vol. ASSP-32, pp. 98-1005, 1984.
7. Gibb, A., "Procedures for Range Arithmetic (Algorithm 61)," *Comm. Assoc. Comp. Mach.*, vol. 4:7, pp. 319 - 320, 1961.

Numerically robust implementations of the Fast RLS adaptive algorithms using Interval Arithmetic

Christopher P Callender
Colin F.N. Cowan

Department of Electrical Engineering,
University of Edinburgh,
Kings Buildings,
Mayfield Road,
Edinburgh EH9 3JL,
Scotland.

Abstract

Fast Recursive Least Squares Algorithms have been developed which perform least squares adaptive filtering in a computationally efficient manner. Unfortunately, these algorithms also suffer from severe numerical instability due to the accumulation of finite precision errors. In this paper a new approach to stabilisation is presented. Upper and lower bounds for all of the quantities involved in the algorithm are calculated in such a way as to guarantee that the infinite precision value is contained within the range of values. The difference between the upper and lower bounds represents the extent to which finite precision errors have accumulated. If these errors are unacceptable, the algorithm may be reinitialised. Results are presented which demonstrate the stability of the new method applied to the Fast Transversal Filters (FTF) Algorithm using both floating and fixed point arithmetic. Results from a hardware implementation of a communications equaliser are also presented.

1. Introduction

Several algorithms have been developed which perform Recursive Least Squares (RLS) adaptive filtering[1, 2] in a highly computationally efficient manner. These algorithms include the Fast Kalman (FK) algorithm [3], the Fast *a Posteriori* Error

Sequential Technique (FAEST)[4, 5] and the Fast Transversal Filters (FTF) algorithm[6-8]. Table I compares the computational complexities of these and other popular algorithms for adaptive Finite Impulse Response (FIR) filters.

Algorithm(Exponentially Windowed, Unnormalised)	Complexity
Least Mean Squares	2N or 3N
Standard Recursive Least Squares	$2.5N^2+4N$
Fast Kalman	10N
Fast a-Posteriori Error Sequential Technique	7N
Fast Transversal Filters	7N

Table I: Comparison of computational complexity of various adaptive algorithms. The complexity is given as the number of multiplications per iteration required to implement an adaptive filter of order N.

It can be seen that all of the Fast RLS algorithms are characterised by a complexity which depends linearly upon the filter order, and which compares favourably with the popular Least Mean Squares (LMS) algorithm[9, 10]. The principal advantage of a Least Squares algorithm over the LMS gradient search algorithm is its greatly improved initial convergence properties, particularly when the spectral conditioning of the input signal is poor[11]

Unfortunately, all of the Fast RLS adaptive algorithms are numerically unstable. This means that when they are implemented using limited precision arithmetic[12] as would be the case with any practical implementation, the small errors[13] at each iteration of the algorithm accumulate until rapid divergence occurs, and the algorithm no longer provides a valid solution.

It has been shown[12] that this instability is introduced by an unstable transformation which underlies all of the transversal Fast RLS algorithms. The matrix associated with this transformation has eigenvalues larger in magnitude than untity. The effect of this is to amplify any errors which exist in the algorithmic quantities at time k to produce larger errors at time k+1. The eventual divergence of the algorithm is therefore inevitable, and so it is not possible to use any of these algorithms continuously without some form of modification.

Various Fast RLS algorithms[14-17] have been discovered for lattice filtering[18] The algorithm of[17] is numerically stable except in the case of very high filter order, N. However, all of the Fast RLS lattice algorithms are characterised by considerably increased complexity when compared with their transversal filter counterparts. Furthermore, in many applications involving system identification or channel estimation, it is the filter coefficients which are of interest, and not the filter output. A complicated transformation[19] is required to convert from lattice coefficients to transversal coefficients.

Various solutions to the instability problem have been proposed. Generally, they can be divided into two categories - those in which the algorithm is regularly reinitialised to prevent divergence[20-23] and those in which the algorithm runs continuously with certain modifications which are designed to provide stability[24-26].

The reinitialisation methods involve resetting various algorithmic variables, hopefully before divergence occurs. Reinitialisation may be performed either periodically in time as in[20], or when certain conditions are violated[6, 21, 22]. In either case, a soft-constrained initial solution is used so that the algorithm does not have to reconverge after reinitialisation. Using reinitialisation to stabilise the algorithms has the advantage of adding little or nothing to the computational complexity, but obviously care must be taken to ensure that reinitialisation occurs sufficiently frequently that divergence does not occur. It is this problem which is addressed by the methods of

this paper.

Other stabilisation procedures, in which modifications are made to the algorithm will generally have increased computational complexity compared to the unstabilised algorithm. It is also very difficult to prove the absolute stability of the modified algorithms, and whilst simulations have demonstrated very worthwhile improvements in performance over their unstabilised counterparts, it is difficult to guarantee their correct operation in all circumstances.

The stabilisation procedure presented in this paper falls into the first category, in which the algorithm is reinitialised. A method of monitoring the accumulation of numerical errors using interval arithmetic is proposed[27]. Lower and upper bounds on the results of all calculations are evaluated, and if the difference between the bounds of the solution is excessive, then the algorithm may be reinitialised, resulting in stable performance.

2. Theory

2.1. Least Squares Adaptive Filtering

A transversal Finite Impulse Response (FIR) filter is one in which the input sequence,

$$\underline{X}(k)=\begin{pmatrix} x_k \\ \cdot \\ \cdot \\ x_{k-N+1} \end{pmatrix} \tag{1}$$

is filtered to produce an output given by

$$y(k)=\underline{X}(k).\underline{H}^T(k) \tag{2}$$

where $\underline{H}(k)$ is a vector of N coefficients, known as tap weights.

To make the filter adaptive, an algorithm must be developed, which finds the

optimum value for $\underline{H}(k)$. This is done by introducing a desired response signal $d(k)$. The filter error is given by

$$e(k)=d(k)-y(k) \tag{3}$$

The exponentially windowed least squares solution to this problem is the one which minimises:

$$J_1(k)=\sum_{i=0}^{k}\lambda^{k-i}e^2(k) \tag{4}$$
$$=\sum_{i=0}^{k}\lambda^{k-i}\left(d(i)-\underline{X}(i).\underline{H}(i)\right)^2$$

λ is a forgetting factor, slightly less than 1, used to enable the adaptive algorithm to track time variant solutions for $\underline{H}(k)$.

The solution which minimises (4) is found to be

$$\underline{H}(k)=\left[\sum_{i=0}^{k}\lambda^{k-i}\underline{X}(i)\underline{X}^T(i)\right]^{-1}\sum_{i=0}^{k}\lambda^{k-i}\underline{X}(i)d(i) \tag{5}$$
$$=r_{xx}^{-1}r_{xd}$$

In principle a least squares adaptive algorithm could be implemented using (5) to calculate the optimum filter coefficients. It should be noted, however, that the first term, r_{xx}^{-1} of this expression involves the inversion of an N x N matrix, which requires order N^3 operations per iteration, using the Gaussian Elimination technique.

The standard Recursive Least Squares (RLS) algorithm is derived by developing equations to update $r^{xx\,-1}(k-1)$, using the new data at time k, so as to give $r_{xx}^{-1}(k)$. This requires only order N^2 operations per iteration.

The Fast RLS algorithms depend upon the shifting properties of the input data vector, $\underline{X}(k)$ with time. This results in a complexity of only order N. The derivation of each of the Fast RLS algorithms is complicated, and will not be repeated here. The

FTF algorithm is listed in appendix I.

2.2. Interval Arithmetic

An interval number[28] is a range of real numbers. Intervals are written using the notation [a,b], which is taken to mean all real numbers between lower endpoint a and upper endpoint b. In set notation

$$[a,b]=\left\{x \mid a \leq x \leq b, x \in R\right\} \tag{6}$$

The arithmetic operator $\bullet$, where $\bullet$ is one of $+$, $-$, , $\div$ may be defined by

$$[a,b]\bullet[c,d]=\left\{x \bullet y \mid a \leq x \leq b, c \leq y \leq d, x \in R, y \in R\right\} \tag{7}$$

That is to say that the result of operation $\bullet$ is the range of all possible results when each of the intervals being operated upon takes all of its possible values. The operation $\div$ cannot be defined for ($c \leq 0$ and $d \geq 0$).

Functions to implement the four operations $+$, $-$, , and $\div$[29] are given in appendix II. When implementing the operations on a limited precision processor, particular care must be taken over the rounding directions of the endpoints of the results. If care is taken to ensure that all lower endpoints are rounded in the direction of $-\infty$ and all upper endpoints in the direction of $+\infty$, then the range of the finite precision interval is guaranteed to contain all of the infinite precision interval.

If the processor being used implements finite precision arithmetic using truncation then it will sometimes round in the correct direction and sometimes it will not. The results must therefore be corrected after calculation.

2.3. Using Interval Arithmetic with the Fast RLS algorithms

Having devised procedures to perform the interval arithmetic operations, it is now a

simple matter to use them with a Fast RLS algorithm. First, all of the scalar variables in the algorithm are converted to interval quantities and the vectors are converted to vectors of intervals. The interval procedures described in § 2.2 are then used to perform all of the computations of the algorithm.

If this is done, then the filter coefficients calculated by the algorithm, $\underline{H}(k)$, will also become intervals. The difference between their upper and lower endpoints indicates the extent to which the solution has been corrupted by numerical errors. If the difference between any of these endpoints exceeds a certain predefined limit, ρ, then the algorithm must be reinitialised to prevent divergence, using the reinitialisation procedure in appendix I.

Reinitialisation is also required if any division operation is about to be preformed by an interval of the form [c,d] where $c \leq 0$ and $d \geq 0$, as this cannot be defined.

The filter and desired response inputs to the algorithm may be represented by degenerate intervals of the form [a,a], which is equivalent to the single real value a.

To obtain non-interval outputs, we may use the centre of the output interval, given by

$$centre\left(\left[a,b\right]\right) = \frac{1}{2}\left(a+b\right) \tag{8}$$

Alternatively, to reduce computation either of the endpoints may be used as an approximation to (8) instead.

2.4. The Reinitialisation Procedure

It is obviously undesirable for the algorithm to have to reconverge every time that it is reinitialised. To prevent this, the algorithm is given an initial solution be means of a 'soft constraint'[6, 20]. This corresponds to modifying the algorithm to minimise the cost function

$$J_2(k)=\sum_{i=0}^{k}\lambda^{k-i}e^2(k)+\mu\lambda^k||\underline{H}(k)-\underline{H}(0)||^2 \qquad (9)$$

where the time index, k, is modified so that k=0 corresponds to the moment of reinitialisation. $\underline{H}\{0\}$ is the initial solution for the filter coefficients and μ is the soft constraint parameter which controls the balance between the two terms of equation (9). If it is too small, the initial condition, $\underline{H}\{0\}$ will be ignored, and the algorithm will have to reconverge. If it is too large, the algorithm will remain close to the possibly incorrect solution $\underline{H}\{0\}$ and will not adapt. The choice of a correct value for μ is therefore of importance in obtaining good performance.

The filter coefficients just before reinitialisation, denoted by $\underline{H}\{-1\}$ are used to obtain the initial solution $\underline{H}\{0\}$.

$$\underline{H}\{0\}=centre\{H\{-1\}\} \qquad (10)$$

where the centre operation (equation 8) is performed on a coefficient by coefficient basis to the vector $\underline{H}\{-1\}$.

The reinitialisation procedure of appendix I table 3 may now be used.

2.5. Choice of the design constants μ and ρ

The choice of the design constants μ, the reinitialisation soft constraint parameter, and ρ, the maximum tolerable width of the filter coefficients is clearly of great importance to the performance of the interval algorithm.

The value for ρ depends upon the level of noise in the system, and if ρ is chosen sufficiently small, then the error in the solution due to arithmetic errors will be of the same order of magnitude as the error in the solution due to the noise present. ρ is therefore chosen to be of the same order as the expected filter error, after the RLS algorithm has converged.

Having selected ρ, it is now possible to find the correct value for μ. It has already been noted that μ controls the balance between the initial condition and the normal RLS solution. If μ is chosen too small, the algorithm will have to reconverge after reinitialisation, and a series of 'spikes' in the solution will be seen at each time that reinitialisation occurs. Too large a value for μ will result in incorrect initial conditions causing the algorithm to give an incorrect solution for some time after reinitialisation occurs.

If we assume that each filter coefficient differs from the infinite precision solution by a random variable from a uniform distribution between $-\rho$ and ρ, just before reinitialisation, then

$$E([Norm\ Tap\ Error]^2) = E(||\underline{h}(i) - \underline{h}_{init}||^2) = \frac{N\rho^2}{3} \qquad 11$$

If we then calculate the expected value of the cost function $J_2(K)$ after the reinitialisation takes place, then

$$E(J_2(k)) = E(\mu\lambda^k ||\underline{h}(i) - \underline{h}_{init}) + E(\sum_{i=0}^{k} \lambda^{k-i} e^2(i)) \qquad 12$$

where $e(i) = d(k) - \underline{h}'(k)\underline{x}(i)$

Hence, using (11) and (12), and expanding the geometric series.

$$E(J_2(k)) = \mu \frac{\lambda^k N\rho^2}{3} + E(e^2(i)) \frac{1-\lambda^{k+1}}{1-\lambda} \qquad 13$$

For a good balance between initial conditions and subsequent adaptation, assuming that the system is stationary, we impose the condition that $E(J_2(i)) = E(J_2(\infty))$ for all i, that is to say that the expected value of the cost function, which is directly related to the filter error is constant for all time after the reinitialisation. Imposing this condition,

$$\frac{\mu\lambda^k N\rho^2}{3} + E(e^2(i)) \frac{1-\lambda^{k+1}}{1-\lambda} = \frac{E(e^2(i))}{1-\lambda} \qquad 14$$

from which we obtain

$$\mu = \frac{3E(e^2(i))\lambda}{N\rho^2(1-\lambda)}. \qquad 15$$

In practice, the initial assumption that the error distribution is uniform between $-\rho$ and $+\rho$ may not be strictly valid, but simulation results have demonstrated that (15) provides a useful starting point for the choice of μ and that operation of the interval algorithm is relatively insensitive to the value chosen for μ.

3. Simulation Results

To illustrate the stability of the new methods, simulations have been performed using an adaptive filter in the system identification configuration as shown in Figure 1.

The signal y(k) is input to both the adaptive filter and to some unknown system which is to be identified. The response of the unknown system is summed with a small amount of noise and forms the desired response of the adaptive filter. The adaptive system converges to have a response close to that of the unknown system. It is then possible to extract the filter coefficients of the FIR adaptive system, which will give approximately the transfer function of the unknown system.

All graphs were obtained by plotting the Euclidian norm of the tap error vector in decibels versus time. The availability of a tap error vector as a reliable performance indicator was the major motivation for using the system identification configuration for simulations. Norm tap error in decibels is given by:

$$NTE(k) = 10\log_{10}\left[\frac{||\underline{H}_{opt}(k) - \underline{H}(k)||^2}{||\underline{H}_{opt}(0)||^2}\right] \qquad 16$$

where

$$\left|\left|\begin{bmatrix} a_0 \\ a_1 \\ . \\ . \\ a_N \end{bmatrix}\right|\right| = \sqrt{a_0^2 + a_1^2 + \cdots + a_N^2}$$

and $\underline{H}_{opt}(k)$ is the vector of optimum tap weights at time k. If the unknown system is actually an FIR filter, then the elements of $H_{opt}(k)$ are the tap weights of this FIR filter.

For all simulations, both the adaptive filter and the unknown system were FIR filters of length 5. The coefficients of the unknown system were

$$\underline{H}_{opt}(k) = \begin{bmatrix} 0.9 \\ 0.3 \\ -0.3 \\ -0.7 \\ 0.1 \end{bmatrix}$$

The input signal was obtained by filtering white gaussian noise with a FIR filter with coefficients [1.0,0.865] which gave an eigenvalue ratio of around 18.

All simulations were performed using the unnormalised FTF algorithm as this is the most computationally efficient RLS algorithm available.

Two arithmetic schemes were tested - 64 bit floating point arithmetic and 16 bit fixed point arithmetic with the provision of a 32 bit long accumulator for the storage of intermediate results during scalar product calculations. The fixed point system is typical of the minimum level of facilities provided by most digital signal processors.

A. FTF with no Stabilisation

Figures 2a and 2b show the sudden divergence of the FTF algorithm when no stabilisation procedure is used. Figure 2a shows the 64 bit floating point implementation and figure 2b shows the 16/32 bit fixed point implementation The fixed point simulation was always found to produce a division by zero error soon after divergence occurred.

It is clear from these results that the numerical stability of the FTF algorithm is unacceptable, unless some form of stabilisation procedure is introduced. The graphs

also demonstrate that increasing the accuracy of the arithmetic will only succeed in delaying the onset of divergence.

B. Reinitialisation if the FTF rescue variable is negative

In[6], the original paper which introduced the FTF algorithm, a variable was identified, which should always be positive. If at any stage this variable becomes negative, then the algorithm should be rescued by reinitialisation.

Figure 3a and 3b show the effects of applying this stabilisation procedure. It can be seen that the floating point version remains stable for longer but that divergence still occurs eventually. The stability of the fixed point version is not significantly improved by this procedure.

The results demonstrate that although it is a necessary condition for the rescue variable to remain positive, it is by no means sufficient if long term stability is required.

C. Interval FTF

Figure 4a and 4b show the results of applying the new interval methods to the FTF algorithm. The results show that numerical stability is greatly improved, and that there is essentially no difference between the fixed and floating point implementations. Care must be taken over the fixed point scale factors, to ensure that overflows do not occur.

The results for the fixed point implementation are particularly impressive, indicating that the interval technique enables the Fast RLS algorithms to be implemented on low cost fixed point digital signal processors.

D. Long Term Stability

Figure 5a and 5b show the long term stability of the new methods for both 64 floating point and 16/32 bit fixed point arithmetic. 1 million time iterations are performed and again no divergence of the algorithm is apparent.

4. Hardware Tests

The interval FTF algorithm was implemented on a TMS320C25 digital signal processor[30]. A number of assembler macros were developed to perform the interval operations of § 2.2, as well as a macro for calculating the interval scalar product of two vectors, taking advantage of the 32 bit long accumulator, and an efficient macro for multiplying a vector by a scalar using interval arithmetic.

The algorithm was used for adaptive equalisation. Figure 6 shows the configuration. A pseudo random binary sequence generator produces a signal which is passed through an FIR channel, producing multi-path interference at the input to the adaptive filter. The purpose of the adaptive system is to converge to the inverse of the FIR channel, removing the multi-path interference. To enable it to do this, the desired response input to the adaptive equaliser is the original pseudo-random binary signal. In practice, the system could be switched to *decision directed mode* [31] after a training sequence. The principal advantage of using a Least Squares algorithm is to minimise the length of this training period.

The data rate in the hardware experiments was 300 bits per second (bps), using an adaptive filter of length 5. The TMS320C25 processor was operating at $\frac{1}{4}$ of its maximum speed, so the current implementation could be used at speeds up to 1200bps. For significantly faster operation, it would be necessary either to develop a multi-processor configuration, or to design a dedicated interval arithmetic co-processor to operate along with the digital signal processor.

Figure 7 shows the various eye diagrams measured using the system. Figure 7a was measured at the output from the channel, and shows the eight different signal levels

introduced by the three tap channel. The eye pattern is not widely open, suggesting that in the presence of noise the bit error rate would be high if this signal was used as the input to a decision device. Figure 7b is the eye diagram after equalisation. It shows two distinct levels, and is widely open, indicating that this signal could be received with a much lower error than the one at the input to the equaliser.

Figure 8 is a trace of filter error squared against time and it shows the stability of the interval arithmetic algorithm. After the spike at the left hand side of the trace, corresponding to the initial convergence, it can be seen that the square of the error remains small for the remainder of the experiment. This confirms the numerical stability of the algorithm in the hardware implementation.

5. Conclusions

The rapid, data independent initial convergence of the RLS algorithms makes them well suited to applications such as echo cancellation for modems, and to channel equalisation for digital radio and telephone communications as they enable a shorter training sequences to be used. The lower computational complexity of the fast algorithms makes them well suited to applications in which a high data rate is also required.

The new interval algorithms operate using interval quantities which are guaranteed to contain the infinite precision solution and the problems of finite precision error are circumvented. An error analysis of the algorithm is performed in real time by the new methods and numerical errors within the algorithm are strictly limited. If the limits are exceeded, then the algorithm is reinitialised, resulting in numerical stability. The stability is independent of the precision of arithmetic which is being used and it has been shown that performance using 16/32 bit fixed point arithmetic is very similar to that obtained using the much more accurate 64 bit floating point arithmetic scheme.

The only disadvantage of the interval methods is their increased computational complexity. The complexity, however, remains directly proportional to filter length, and so for long adaptive filters, a considerable saving in complexity over the standard RLS algorithm will still be obtained.

References

1. Godard, D., "Channel Equalisation using a Kalman Filter for Fast Data Transmission," *IBM J. Res. Develop.*, vol. 18(3), pp. 267-273, May 1974.

2. Honig, M.L. and Messerschmitt, D.G., *Adaptive Filters:Structures, Algorithms and Applications,* Kluwer Academic Publishers, Norwell, MA., 1984.

3. Ljung, L., Morf, M., and Falconer, D., "Fast calculation of gain matrices for recursive estimation schemes," *Int. J. Control*, vol. 27, pp. 1 - 19, 1978.

4. Carayannis, G., Manolakis, D. G., and Kalouptsidis, N., "A Fast Sequential Algorithm for Least Squares Filtering and Prediction," *IEEE Trans. Acoust, Speech, Signal Process.*, vol. ASSP-31, No 6., pp. 1394 - 1402, 1983.

5. Kalouptsidis, N., Carayannis, G., and Manolakis, D., "A Fast Covariance Algorithm for Sequential Least Squares Filtering and Prediction," *IEEE Trans. Auto. Control*, vol. AC-29, pp. 752-755, Aug. 1984.

6. Cioffi, J. M. and Kailath, T., "Fast, Recursive Least Squares Transversal Filters for Adaptive Filtering," *IEEE Trans. Acoust, Speech, Signal Process.*, vol. ASSP-32, No 2., pp. 304 - 337, 1984.

7. Cioffi, J.M. and Kailath, K., "Windowed Fast Transversal Filter Adaptive

Algorithms with Normalisation," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-33, pp. 607-625, June 1985.

8. Cioffi, J.M., "The Block Processing FTF adaptive algorithm," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-34, pp. 77-90, Feb 1986.

9. Widrow, B. and et, al, "Stationary and Non-Stationary learning Characteristics of the LMS adaptive Filter," *Proc. IEEE*, vol. 64, pp. 1151-1162, 1976.

10. Widrow, B. and Stearns, S., *Adaptive Signal Processing,* Prentice-Hall, Englewood Cliffs, 1985.

11. Mulgrew, B. and Cowan, C.F.N., *Adaptive Filters and Equalisers,* Kluwer Academic Publishers, Norwell, Mass., 1988.

12. Cioffi, J. M., "Limited Precision Effects in Adaptive Filtering," *IEEE Trans. Circuits Syst.*, vol. CAS-34 No 7., pp. 821 - 833, 1987.

13. Wilkinson, J.H., *Rounding Errors in Algebraic Processes,* H.M. Stationary Office, London, 1963.

14. Lee, D.T.L, Morf, M., and Friedlander, B., "Recursive Least Squares Ladder Estimation Algorithms," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-29(3), pp. 627-641, June 1981.

15. Porat, B., Friedlander, B., and Morf, M., "Square Root Covariance Ladder Algorithms," *IEEE Trans. Automatic Control*, vol. AC-27(4), pp. 813-829, August 1982.

16. Shensa, M.J., "Recursive Least Squares Lattice Algorithms - a Geometric Approach," *IEEE Trans. Automatic Control*, vol. AC-26(3), pp. 675-702, June

1981.

17. Ling, F., Manolakis, D., and Proakis, J.G., "Numerically Robust Least Squares Lattice Ladder Algorithms with Direct Updating of the Reflection Coefficients," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-34(4), pp. 837-845, Aug 1986.

18. Gray, A.H. and Markel, J.D., "Digital Lattice and Ladder Filter Synthesis," *IEEE Trans. Audio Electroacoust.*, vol. AV-21(6), pp. 491-500, 1973.

19. Cowan, C. F.N. and Grant, P. M., *Adaptive Filters*, Prentice Hall, Englewood Cliffs, 1985.

20. Lin, D.W., "On the Digital Implementation of the Fast Kalman Algorithm," *IEEE. Trans. Acoust. Speech Signal Process.*, vol. ASSP-32, pp. 998-1005, 1984.

21. Hariharan, S. and Clark, A.P., "HF Channel estimation using a Fast Transversal Filter Algorithm," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-38, pp. 1353-1362, Aug 1990.

22. Kim, D. and Alexander, W.E., "Stability Analysis of the Fast RLS Adaptation Algorithm," *Proceedings ICASSP 88*, vol. 3, pp. 1361-1364, New York, April 1988.

23. Elefteriou, E. and Falconer, D., "Restart Methods for Stabilising FRLS Adaptive Equaliser Filters in Digital HF Transmission," *Globecom*, Atlanta, Dec 1984.

24. Benallal, A. and Gilloire, A., "A New Method to Stabilize Fast RLS Algorithms based on a First Order Model of the Propagation of Numerical Errors,"

Proceedings ICASSP 88, vol. 3, pp. 1373-1376, New York, April 1988.

25. Slock, D.T. and Kailath, T., "Numerically Stable Fast Recursive Least Squares Transversal Filters," *Proceedings ICASSP 88*, vol. 3, pp. 1365-1368, New York, April 1988.

26. Botto, J.L. and Moustakides, G.V., "Stabilising the Fast Kalman Algorithm," *IEEE Trans. Acoust., Speech, Signal Process*, vol. ASSP-38, pp. 1342-1348, Sept 1989.

27. Callender, C.P. and Cowan, C.F.N., "Numerically Robust Implementations of Fast RLS Adaptive Algorithms using Interval Arithmetic," *Proceedings EUSIPCO 90*, pp. 173-176, Barcelona, Sept. 1990.

28. Moore, R. E., *Interval Analysis*, Prentice-Hall, Englewood Cliffs, 1966.

29. Gibb, A., "Procedures for Range Arithmetic (Algorithm 61)," *Comm. Assoc. Comp. Mach.*, vol. 4:7, pp. 319 - 320, 1961.

30. *Second Generation TMS320 Users Guide*, Texas Instruments, Dallas, Texas, 1988.

31. Stremler, F.G., *Introduction to Communications Systems*, Addison Wesley, Reading, Mass., 1982.

Appendix I:The unnormalised FTF Algorithm

Fast Exact Initialisation
$T=0$: $A_{0,0}=B_{0,0}=1$, $C_{0,0}=0$(zero dimension) $H_{1,0}=\frac{-d(0)}{y(0)}$, $\gamma_0(0)=1$, $\alpha_0(0)=y(0)^2$
$1\leq T\leq N$: $e_{f-1}^p(T)=A_{T-1,T-1}\left[y(T),\ldots,y(1)\right]'$ $A_{T,T}^T=\left[A^{T-1,T-1}, \frac{-e_{f-1}^p(T)}{y(0)}\right]$ $e_{T-1}(T)=e_{f-1}^p(T)\gamma_{T-1}(T-1)$ $\alpha_T(T)=\lambda\alpha_{T-1}(T-1)$ $\alpha_{T-1}(T)=\alpha_T(T)+e_{f-1}^p(T)e_{T-1}(T)$ $\gamma_T(T)=\gamma_{T-1}(T-1)\frac{\alpha_T(T)}{\alpha_{T-1}(T)}$ $\tilde{C}_{T,T}=\left[0\ \tilde{C}_{T-1,T-1}\right]-\frac{e_{T-1}(T)}{\alpha_T(T)}A_{T-1,T-1}$ $B_{T,T}=\left[\left(y(0)\gamma_T(T)\tilde{C}_{T,T}\right)\ 1\right]$ (Only when T=N) $\beta_{T,T}=y(0)^2\gamma_T(T)$ (Only when T=N) $\epsilon_f^p(T)=d(T)+H_{T,T-1}Y_T(T)$ $\epsilon_T(T)=\epsilon_f^p(T)\gamma_T(T)$ if T<N, $H_{T+1,T}=\left[H_{T,T-1}\ \ \frac{-\epsilon_f^p(T)}{y(0)}\right]$ if T=N, $H_{N,T}=H_{T,T-1}+\epsilon_T(T)\tilde{C}_{T,T}$

Table I:The Fast Exact Initialisation Procedure for the FTF algorithm

Unnormalised FTF Algorithm	
1	$e_N^p(T) = A_{N,T-1} Y_{N+1}(T)$
2	$e_N(T) = e_N^p(T)\gamma_N(T-1)$
3	$\alpha_N(T) = \lambda\alpha_N(T-1) + e_N^p(T)e_N(T)$
4	$\gamma_{N+1}(T) = \dfrac{\lambda\alpha_N(T-1)}{\alpha_N(T)}\gamma_N(T-1)$
5	$\tilde{C}_{N+1,T} = \begin{bmatrix} 0 & \tilde{C}_{N,T-1} \end{bmatrix} - \dfrac{1}{\lambda} e_N^p(T)\alpha_N^{-1}(T-1)A_{N,T-1}$
6	$A_{N,T} = A_{N,T-1} + e_N(T)\begin{bmatrix} 0 & \tilde{C}_{N,T-1} \end{bmatrix}$
7	$r_N^p(T) = -\lambda\beta_N(T-1)\tilde{C}_{N+1,T}^N$
8	$\gamma_N(T) = \left[1 + r_N^p(T)\gamma_{N+1}(T)\tilde{C}_{N+1,T}^N\right]^{-1}\gamma_{N+1}(T)$
	rescue variable † $= \left[1 + r_n^p(T)\gamma_{N+1}(T)\tilde{C}_{N+1,T}^N\right]$
9	$r_N(T) = r_N^p(T)\gamma_N(T)$
10	$\beta_N(T) = \lambda\beta_N(T-1) + r_N^p(T)r_N(T)$
11	$\begin{bmatrix} \tilde{C}_{N,T} & 0 \end{bmatrix} = \tilde{C}_{N+1,T} - \tilde{C}_{N+1,T}^N B_{N,T-1}$
12	$B_{N,T} = B_{N,T-1} + r_N(T)\begin{bmatrix} \tilde{C}_{N,T} & 0 \end{bmatrix}$
13	$\epsilon_N^p(T) = d(T) + H_{N,T-1}Y_N(T)$
14	$\epsilon_N(T) = \epsilon_N^p(T)\gamma_N(T)$
15	$H_{N,T} = H_{N,T-1} + \epsilon_N(T)\tilde{C}_{N,T}$

† Rescue if rescue variable is negative (see table III).

Table II:The steady state FTF algorithm.

FTF Reinitialisation
$A_{N,-1} = \left[1,0,\ldots,0\right]$
$B_{N,-1} = \left[0,\ldots,0,1\right]$
$C_{N,-1} = \left[0,\ldots,0\right]$
$\alpha_N(-1) = \lambda^N \mu$
$\beta_N(-1) = \mu$
$\gamma_N(-1) = 1$
$W_{N,-1} = W_0$

Table III:The rescue procedure for the FTF algorithm. μ is a soft constraint which determines the influence of the solution before the reinitialisation, W_0, on future solutions.

Appendix II:Efficient interval arithmetic procedures.

RANGE_ADD(a,b,c,d)

```
/*A procedure to calculate the result [e,f]=[a,b] + [c,d] */

e=a+c;
f=b+d;
```

End of procedure.

RANGE_SUBTRACT(a,b,c,d)

```
/*A procedure to calculate the result [e,f]=[a,b] - [c,d] */

e=a-d;
f=b-c;
```

End of procedure.

RANGE_DIVIDE(a,b,c,d)

```
/* A procedure to calculate the result [e,f]=[a,b] / [c,d] */

if (c≤0 and d≥0)  {
                  print 'Division by zero error"
                  exit
                  }
if (c<0) {
      if (b>0) e=b/d; else e=b/c;
      if (a>=0) f=a/c; else f=a/d;
      }
else {
      if (a<0) e=a/c; else e=a/d;
      if (b>0) f=b/c; else f=b/c;
      }
```

End of procedure.

RANGE_MULTIPLY(a,b,c,d)

```
/* Procedure to calculate the result [e,f]=[a,b] * [c,d] */

if (a<0 && c>=0) {
        temp=a;
        a=c;
        c=temp;
        temp=b;
        b=d;
        d=temp;
        }
if (a>=0) {
        if (c>=0) {
                e=a*c;
                f=b*d;
                }
        else {
        e=b*c;
        if (d>=0) f=b*d; else f=a*d;
        }
}
else {
        if (b>0) {
                if (d>0) {
                        e=min(a*d,b*c);
                        f=max(a*c,b*d);
                        }
                else  {
                        e=b*c;
                        f=a*c;
                }
        }
        else {
                f=a*c;
                if (d<=0) e=b*d; else e=a*d;
        }
}
```

End of procedure.

Captions

Figure 1

Configuration used for all computer simulations. Adaptive system identification is performed by connecting the output of an unknown system to the desired response input of an adaptive filter. The adaptive system converges to produce the same response as the unknown system, and if the unknown system is an FIR filter, the adaptive system will then have the same coefficients.

Figure 2a

Performance of FTF adaptive algorithm with no stabilisation using 64 bit floating point arithmetic.
$\lambda = 0.98$
Signal to Noise Ratio=40dB

Figure 2b

Performance of FTF adaptive algorithm with no stabilisation using 16/32 bit fixed point arithmetic.
$\lambda = 0.98$
Signal to Noise Ratio=40dB

Figure 3a

Performance of FTF adaptive algorithm, reinitialising if rescue variable is negative using 64 bit floating point arithmetic.
$\lambda = 0.98$
$\mu = 1.0$
Signal to Noise Ratio=40dB

Figure 3b

Performance of FTF adaptive algorithm, reinitialising if rescue variable is negative using 16/32 bit fixed point arithmetic.
$\lambda = 0.98$
$\mu = 1.0$
Signal to Noise Ratio=40dB

Figure 4a

Performance of the Interval FTF algorithm, using 64 bit floating point arithmetic.
$\lambda = 0.98$
$\mu = 627.2$
$\rho = 0.00225$
Signal to Noise Ratio=40dB

Figure 4b

Performance of the Interval FTF algorithm, using 16/32 bit fixed point arithmetic.
$\lambda = 0.98$
$\mu = 0.5$
$\rho = 0.00225$
Signal to Noise Ratio = 40dB

Figure 5a

Long term performance of the Interval FTF algorithm, using 64 bit floating point arithmetic.
$\lambda = 0.98$
$\mu = 627.2$
$\rho = 0.00225$
Signal to Noise Ratio = 40dB

Figure 5b

Long term performance of the Interval FTF algorithm, using 16/32 bit fixed point arithmetic.
$\lambda = 0.98$
$\mu = 0.5$
$\rho = 0.00225$
Signal to Noise Ratio = 40dB

Figure 6

Hardware configuration for performing adaptive equalisation. A pseudo-random sequence representing the transmitted signal is generated, and input to a FIR filter which represents the transmission channel. The output from the channel is the input to an adaptive filter, which is given the transmitted sequence as its desired response. It therefore converges to the inverse of the FIR channel, allowing the original transmitted signal to be recovered.

Figure 7a

Eye diagram measured at the input to the adaptive equaliser. It shows the eight distinct levels introduced by the three tap FIR channel. The narrow, partially closed 'eye' indicates that the signal is not suitable for use without equalisation in the presence of high levels of noise.

Figure 7b

Eye diagram measured at the output from the adaptive equaliser. It shows that two distinct levels have been almost recovered, and that the signal is suitable for use as the input to a decision device.

Figure 8

Graph of squared filter error against time for the hardware system. After the spike representing initial convergence, it can be seen that the error remains small, confirming the numerical stability of the

interval arithmetic algorithm.

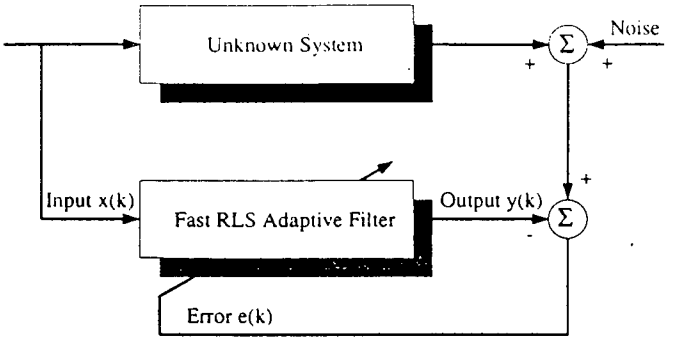
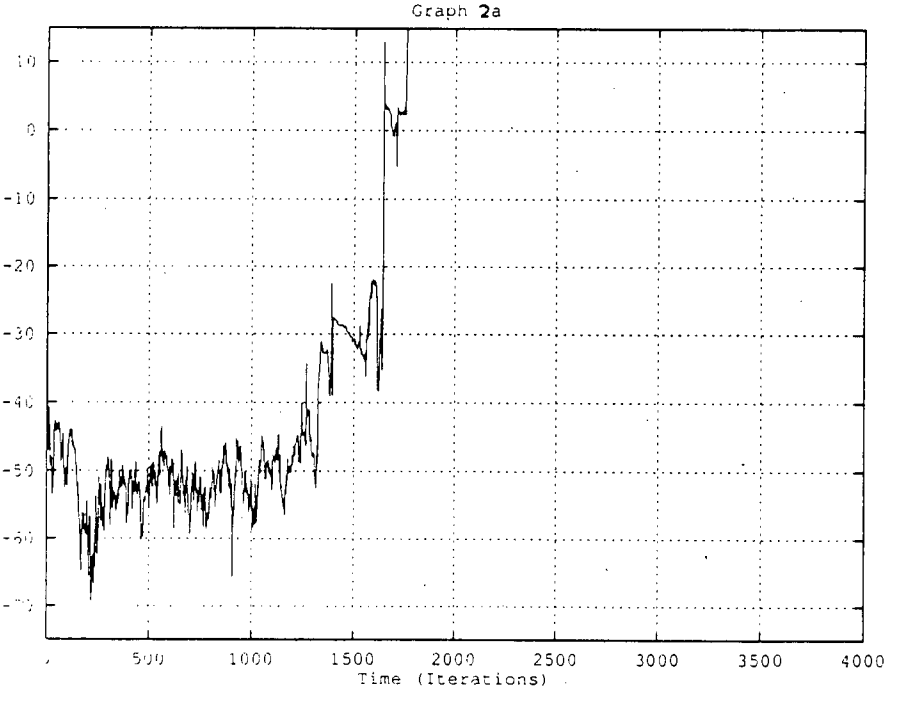
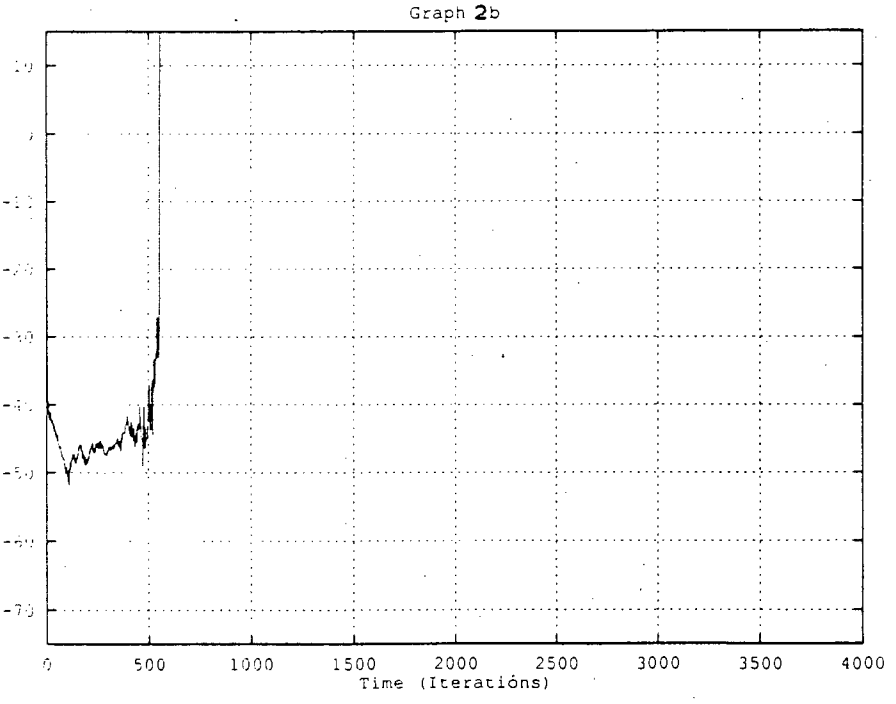
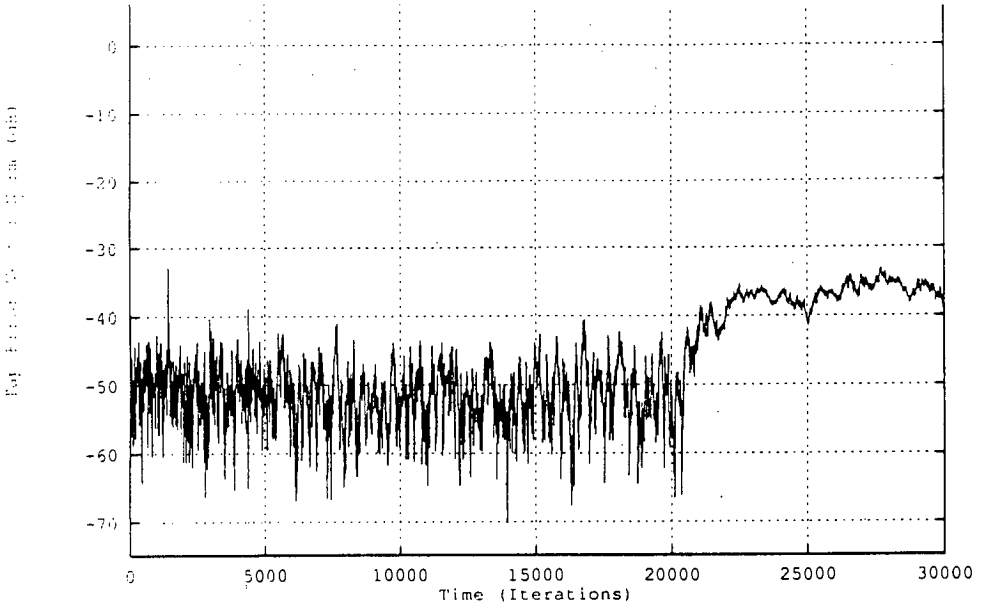


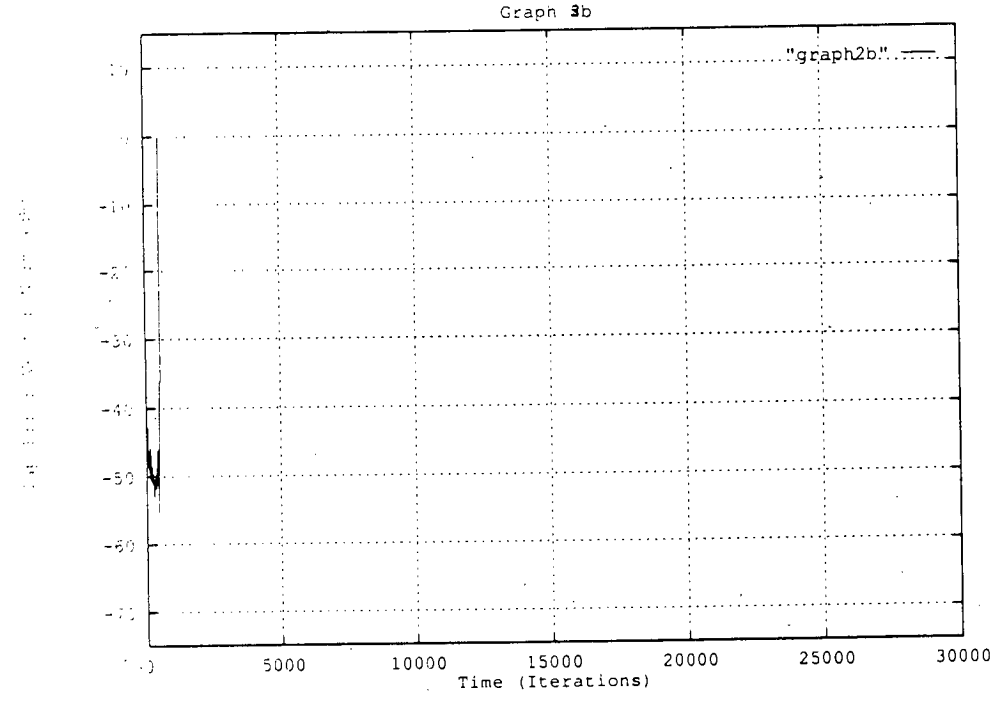
Figure 1



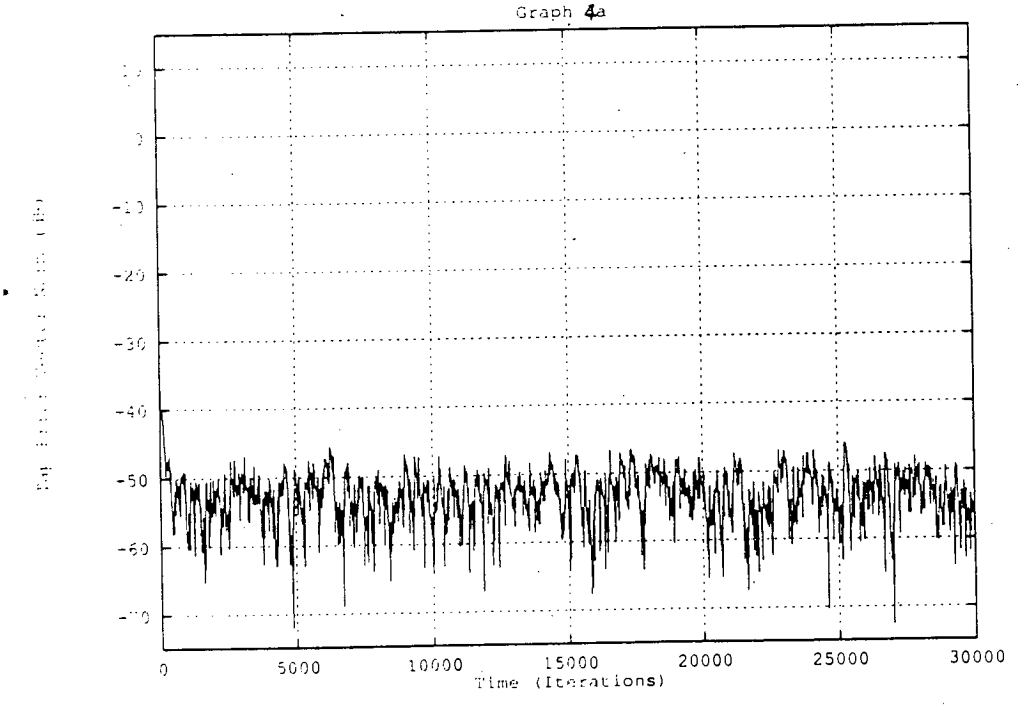


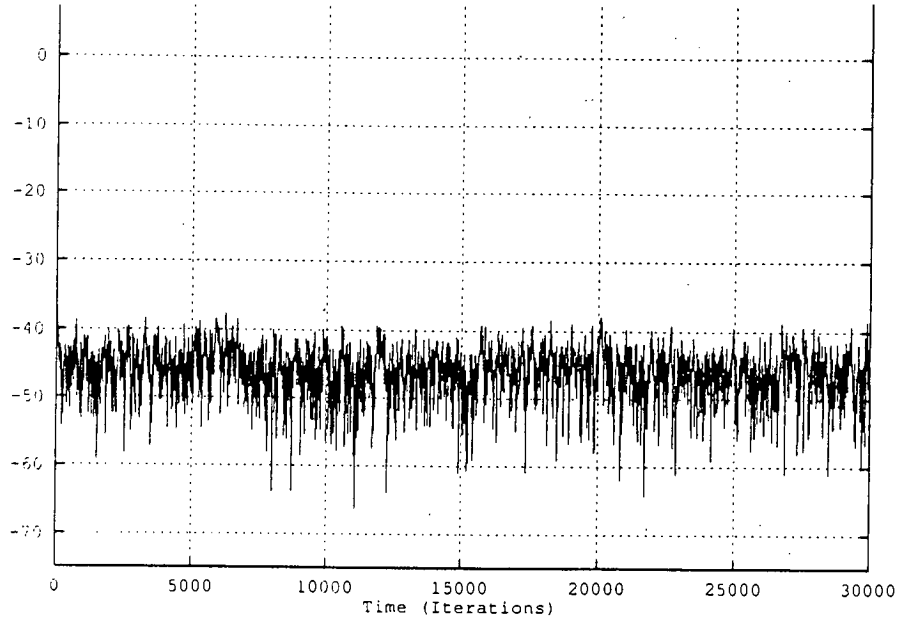


Graph 3b

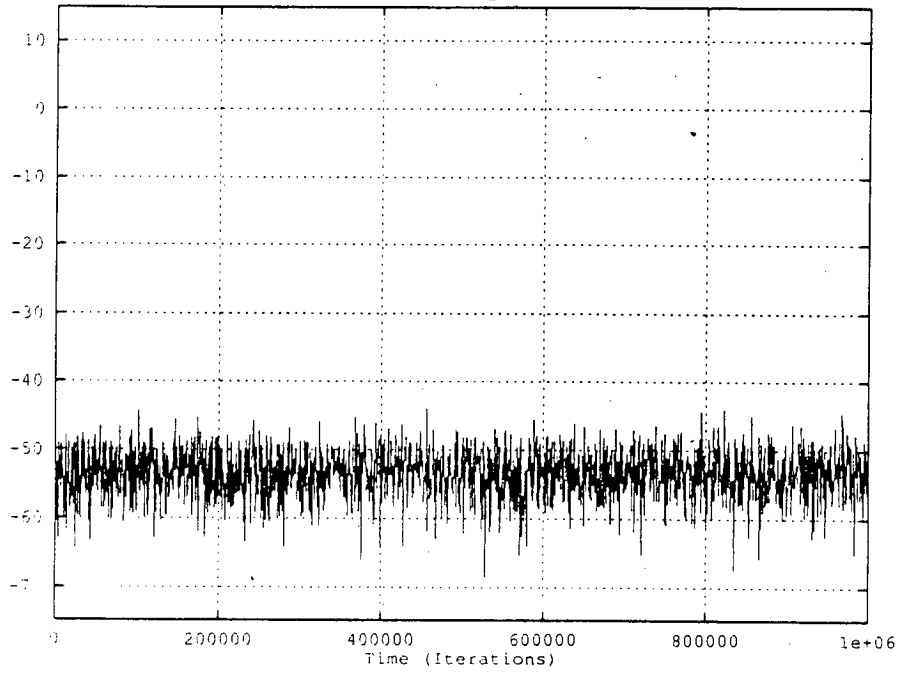


Graph 4a

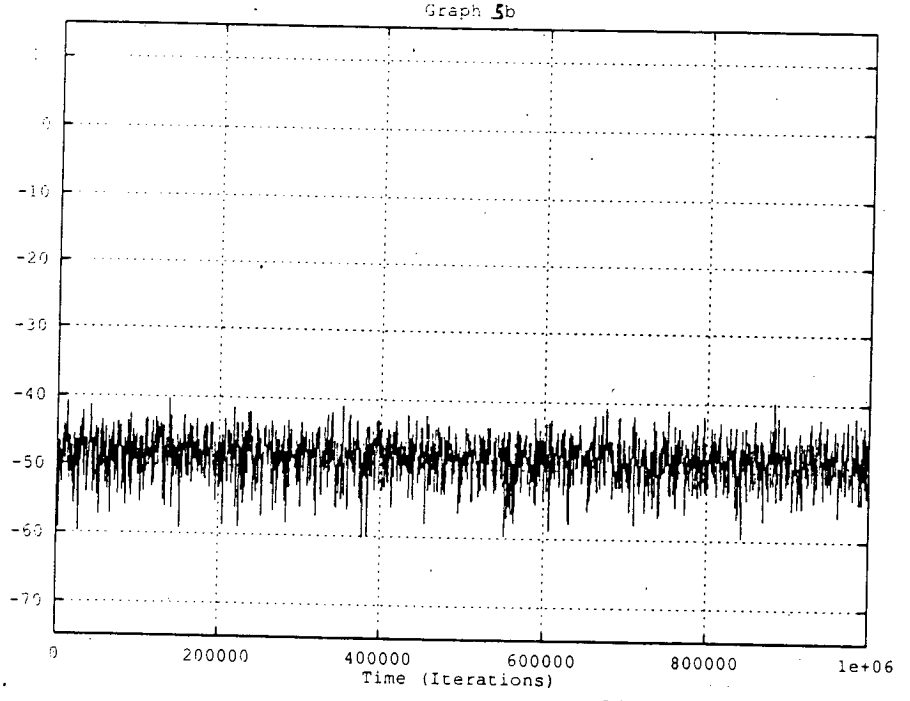




Graph 5a



Graph 5b



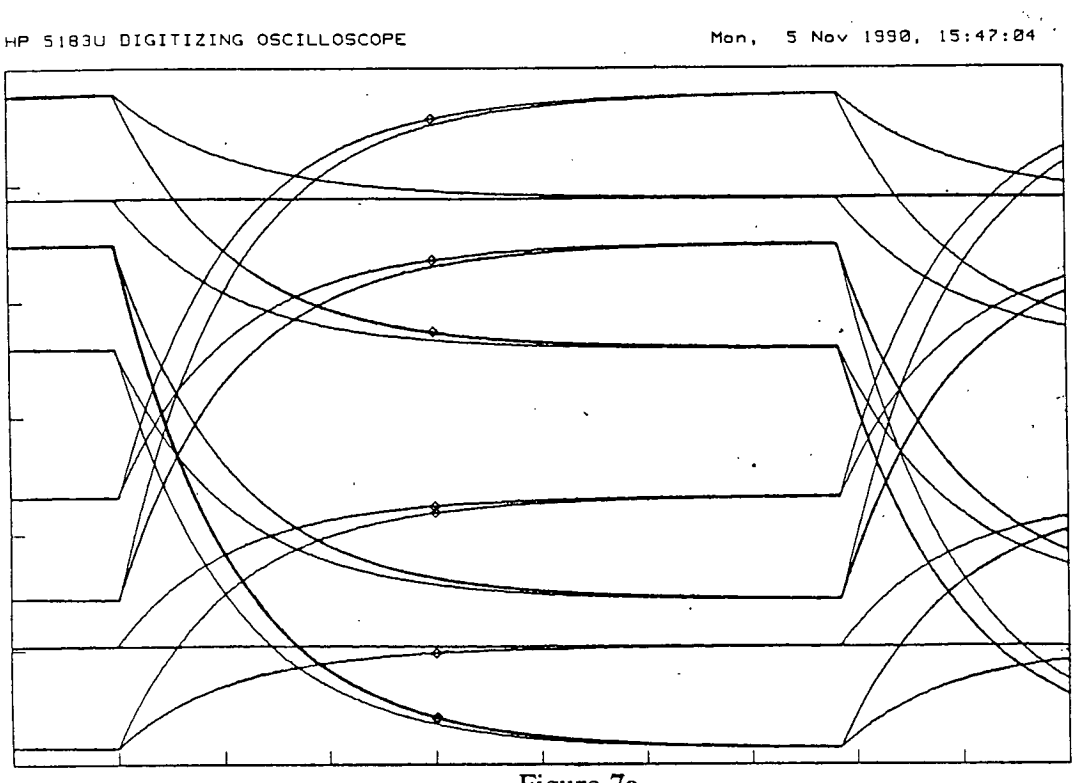


Figure 7a

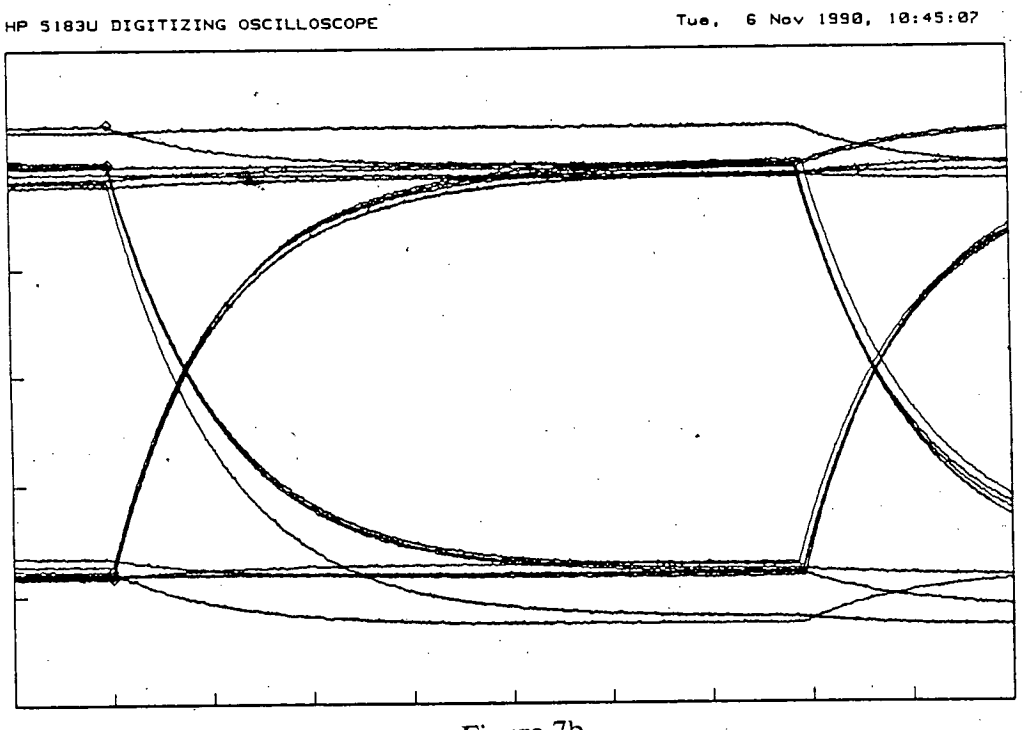


Figure 7b

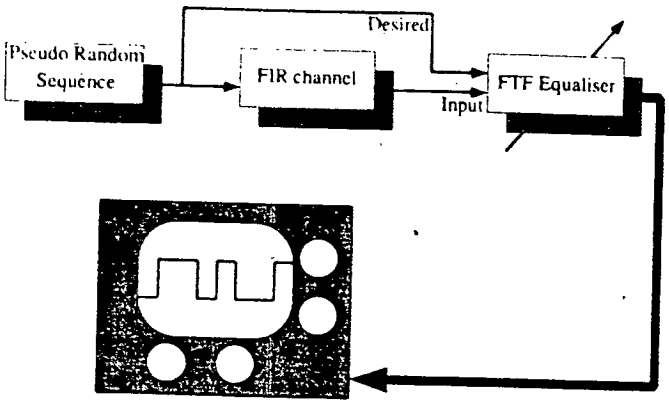


Figure 6

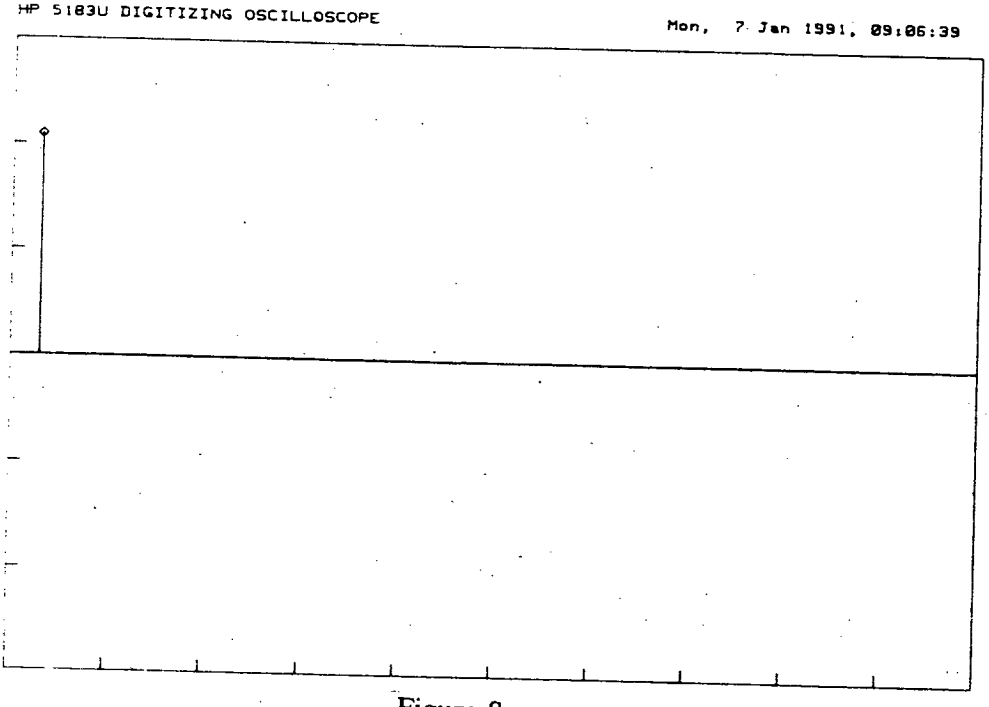


Figure 8

Appendix B

Simulation Software

```
#include <stdio.h>

void msc();
double radd();
double rmul();
double rdiv();
void cadd();
void cmul();
void cinv();
void error();

struct SPVAR {
        int  rsize;
        int csize;
        double *element;
        };

double *m_realloc(ptr,sz)
double *ptr;
int sz;
{
if (ptr==0) ptr=(double *) malloc(sz);

else ptr=(double *)realloc(ptr,sz);
return(ptr);
}

void dbx_disp(me)
struct SPVAR me;
{
          int       j, k,r,c;
           double   *adr;

           r=me.rsize;
           c=me.csize;
           adr=me.element;
          for (j = 0; j != r; j++) {

                    for (k = 0; k != c; k++) {

                              if (*(adr + 1) < 0)

                                        printf("%8.2e - %8.2ej   ", *adr,
                                             (-*(adr + 1)));
                              else
                                        printf("%8.2e + %8.2ej   ", *adr,
                                             *(adr + 1));
                              adr++;
                              adr++;
                    }
                    printf("\n");
          }
                  printf("\n");
}

displaymatrix(adr, r, c,fp)
int       r, c;
```

displaymatrix

```
FILE *fp;
double  *adr;
{
        int     j, k;
        for (j = 0; j != r; j++) {

                for (k = 0; k != c; k++) {

                        if (*(adr + 1) < 0)

                                fprintf(fp,"%8.2e - %8.2ej   ", *adr,
                                    (-*(adr + 1)));
                        else
                                fprintf(fp,"%8.2e + %8.2ej   ", *adr,
                                    *(adr + 1));
                        adr++;
                        adr++;
                }
                fprintf(fp,"\n");
        }
        fprintf(fp,"\n");
}

void addmatrix(adr1, adr2, adr3, r1, c1, r2, c2, r3, c3)
int     r1, c1, r2, c2, r3, c3;
double  *adr1, *adr2, *adr3;
{
        int     j, k;
        if (r1 != r2)
                error("Unable to add matrices of different sizes");
        if (c1 != c2)
                error("Unable to add matrices of different sizes");
        if (r1 != r3)
                error("Result matrix of incorrect size in add");
        if (c1 != c3)
                error("Result matrix of incorrect size in add");
        for (j = 0; j != r1; j++) {

                for (k = 0; k != c1; k++) {

                        cadd(*adr1, *(adr1 + 1), *adr2, *(adr2 +
                            1), adr3);
                        adr1 = adr1 + 2;
                        adr2 = adr2 + 2;
                        adr3 = adr3 + 2;
                }
        }
}

void iden(adr, r, c)

double  *adr;
int     r, c;
{
        int     j, k;
        if (r != c)
                error("Non square matrix cannot be set to identity");
        for (j = 0; j != r; j++) {

                for (k = 0; k != c; k++) {

                        if (j == k) {
                                *adr = 1;
                                *(adr + 1) = 0;

                        }
                        if (j != k) {
```

```
                                        *adr = 0;
                                        *(adr + 1) = 0;
                                }
                                adr = adr + 2;
                        }
                }
}

void zer(adr, r, c)

double  *adr;
int     r, c;
{
        int     j, k;
        for (j = 0; j != r; j++) {
                for (k = 0; k != c; k++) {
                        *adr = 0;
                        *(adr + 1) = 0;
                        adr = adr + 2;
                }
        }
}

void setel(adr, r, c, x, y, vr, vi)
double  vr, vi;
double  *adr;
int     r, c, x, y;
{
        if (y > c)
                error("Setelement out of bounds");
        if (x > r)
                error("Setelement out of bounds");
        if (y < 1)
                error("Setelement out of bounds");
        if (x < 1)
                error("Setelement out of bounds");
        x--;
        y--;
        *(adr + (c * x * 2) + y * 2) = vr;
        *(adr + (c * x * 2) + y * 2 + 1) = vi;
}

void mult(adr1, adr2, adr3, r1, c1, r2, c2, r3, c3)
double  *adr1, *adr2, *adr3;
int     r1, c1, r2, c2, r3, c3;
{
        int     j, k, l;
        double  total[2], ar, ai, br, bi, t[2];
        double  *temp;
        if (r1 == 1 && c1 == 1) {
                msc(adr2,adr3,r2,c2,r3,c3,*adr1,*(adr1+1));
                goto SKIP;
                }
        if (r2 == 1 && c2 == 1) {
                msc(adr1,adr3,r1,c1,r3,c3,*adr2,*(adr2+1));
                goto SKIP;
                }
        if (r2 != c1)
                error("Unable to multiply matrices - dimensions incorrect");
        if (c2 != c3)
                error("Result matrix of incorrect size in multiply");
```

```
        if (r1 != r3)
                error("Result matrix of incorrect size in multiply");
        temp = (double *)malloc(sizeof(double)*r3 * c3 * 2);
        if (temp == 0)

                error("Out of Memory Error");
        for (j = 0; j != r3; j++) {

                for (k = 0; k != c3; k++) {

                        total[0] = 0;

                        total[1] = 0;

                        for (l = 0; l != c1; l++) {

                                ar = (*(adr1 + (c1 * j * 2) + l
                                    *2));
                                ai = (*(adr1 + (c1 * j * 2) + l
                                    *2 + 1));
                                br = (*(adr2 + (c2 * l * 2) + k
                                    *2));
                                bi = (*(adr2 + (c2 * l * 2) + k
                                    *2 + 1));
                                cmul(ar, ai, br, bi, t);
                                cadd(t[0], t[1], total[0], total[1],

                                    total);
                        }
                        *(temp + (c3 * j * 2) + k * 2) = total[0];

                        *(temp + (c3 * j * 2) + k * 2 + 1) = total[1];
                }
        }
        for (j = 0; j != r3; j++) {

                for (k = 0; k != c3; k++) {

                        ar = (*(temp + (c3 * j * 2) + k * 2));
                        ai = (*(temp + (c3 * j * 2) + k * 2 + 1));
                        *(adr3 + (c3 * j * 2) + k * 2) = ar;
                        *(adr3 + (c3 * j * 2) + k * 2 + 1) = ai;
                }
        }
        free(temp);
SKIP: ;
}

number_of_rows(r1,c1,r2,c2)                                    number_of_rows
int r1,c1,r2,c2;
{
if (r1==1 && c1==1) return(r2);
return(r1);
}

number_of_columns(r1,c1,r2,c2)                                 number_of_columns
int r1,c1,r2,c2;
{
if (r2==1 && c2==1) return(c1);
return(c2);
}

void getel(adr, r, c, x, y, v)
double  *v;
double  *adr;
int     r, c, x, y;
{
```

```
        if (y > c)
                error("Getelement out of bounds");
        if (x > r)
                error("Getelement out of bounds");
        if (y < 1)
                error("Getelement out of bounds");
        if (x < 1)
                error("Getelement out of bounds");
        x--;
        y--;
        (*v) = (*(adr + (c * x * 2) + y * 2));
        (*(v + 1)) = (*(adr + (c * x * 2) + y * 2 + 1));
}

void submatrix(adr1, adr2, adr3, r1, c1, r2, c2, r3, c3)
int     r1, c1, r2, c2, r3, c3;
double  *adr1, *adr2, *adr3;
{
        int     j, k;
        double  a, b, d;
        if (r1 != r2)
                error("Unable to subtract matrices of different sizes");
        if (c1 != c2)
                error("Unable to subtract matrices of different sizes");
        if (r1 != r3)
                error("Result matrix of incorrect size in subtract");
        if (c1 != c3)
                error("Result matrix of incorrect size in subtract");
        for (j = 0; j != r1; j++) {

                for (k = 0; k != c1; k++) {

                        cadd(*adr1, *(adr1 + 1), -(*adr2), -(*(adr2
                                + 1)), adr3);
                        adr1 = adr1 + 2;
                        adr2 = adr2 + 2;
                        adr3 = adr3 + 2;
                }
        }
}

void trans(adr1, adr2, r1, c1, r2, c2)
double  *adr1, *adr2;
int     r1, c1, r2, c2;
{
        double  ar, ai;
        double  *temp;
        int     j, k;
        if (r1 != c2)
                error("Result matrix of incorrect size in transpose");
        if (c1 != r2)
                error("Result matrix of incorrect size in transpose");
        temp = (double *)malloc(sizeof(double)*r2 * c2 * 2);
        if (temp == 0)

                error("Out of memory error");
        for (j = 0; j != r1; j++) {

                for (k = 0; k != c1; k++) {

                        ar = (*(adr1 + (j * c1 * 2) + k * 2));
                        ai = ((*(adr1 + (j * c1 * 2) + k * 2 +
                                1)));
                        *(temp + (k * r1 * 2) + j * 2) = ar;
                        *(temp + (k * r1 * 2) + j * 2 + 1) = ai;
                }
        }
```

```
        for (j = 0; j != r2; j++) {

                for (k = 0; k != c2; k++) {

                        ar = (*(temp + (c2 * j * 2) + k * 2));
                        ai = (*(temp + (c2 * j * 2) + k * 2 + 1));
                        *(adr2 + (c2 * j * 2) + k * 2) = ar;
                        *(adr2 + (c2 * j * 2) + k * 2 + 1) = ai;
                }
        }
        free(temp);
}

void transp(adr1, adr2, r1, c1, r2, c2)
double  *adr1, *adr2;
int     r1, c1, r2, c2;
{
        double  ar, ai;
        double  *temp;
        int     j, k;
        if (r1 != c2)
                error("Result matrix of incorrect size in transpose");
        if (c1 != r2)
                error("Result matrix of incorrect size in transpose");
        temp = (double *)malloc(sizeof(double)*r2 * c2 * 2);
        if (temp == 0)

                error("Out of memory error");
        for (j = 0; j != r1; j++) {

                for (k = 0; k != c1; k++) {

                        ar = (*(adr1 + (j * c1 * 2) + k * 2));
                        ai = (-(*(adr1 + (j * c1 * 2) + k * 2 +
                            1)));
                        *(temp + (k * r1 * 2) + j * 2) = ar;
                        *(temp + (k * r1 * 2) + j * 2 + 1) = ai;
                }
        }
        for (j = 0; j != r2; j++) {

                for (k = 0; k != c2; k++) {

                        ar = (*(temp + (c2 * j * 2) + k * 2));
                        ai = (*(temp + (c2 * j * 2) + k * 2 + 1));
                        *(adr2 + (c2 * j * 2) + k * 2) = ar;
                        *(adr2 + (c2 * j * 2) + k * 2 + 1) = ai;
                }
        }
        free(temp);
}

void msc(adr1, adr2, r1, c1, r2, c2, vr, vi)
double  *adr1, *adr2;
double  vr, vi;
int     r1, c1, r2, c2;
{
        int     j, k;
        double  ar, ai;
        if (r1 != r2)
                error("Result matrix of incorrect size in multscal");
        if (c1 != c2)
                error("Result matrix of incorrect size in multscal");
        for (j = 0; j != r1; j++) {

                for (k = 0; k != c1; k++) {
```

```
                        ar = (*adr1);
                        ai = (*(adr1 + 1));
                        cmul(ar, ai, vr, vi, adr2);
                        adr1 = adr1 + 2;
                        adr2 = adr2 + 2;
                }
        }
}

void cpy(adr1, adr2, r1, c1, r2, c2)
double  *adr1, *adr2;
int     r1, c1, r2, c2;
{
        int     j, k;
        double  ar, ai;
        if (r1 != r2)
                error("Result matrix of incorrect size in copy");
        if (c1 != c2)
                error("Result matrix of incorrect size in copy");
        for (j = 0; j != r1; j++) {

                for (k = 0; k != c1; k++) {

                        ar = (*adr1);
                        ai = (*(adr1 + 1));
                        *adr2 = ar;
                        *(adr2 + 1) = ai;
                        adr1 = adr1 + 2;
                        adr2 = adr2 + 2;
                }
        }
}

void ups(adr, r, c)
double  *adr;
int     r, c;
{
        int     j, k;
        double  ar, ai;
        if (r != 1 && c != 1)
                error("Unable to upshift a matrix");
        if (r == 1 && c == 1)
                error("Unable to upshift a scalar");
        if (r == 1)
                j = c;
        if (c == 1)
                j = r;
        for (k = j - 1; k != 0; k--) {

                ar = (*(adr + k * 2 - 2));
                ai = (*(adr + k * 2 - 1));
                *(adr + k * 2) = ar;
                *(adr + k * 2 + 1) = ai;
        }
        *adr = 0;

        *(adr + 1) = 0;

}

void dos(adr, r, c)
double  *adr;
int     r, c;
{
        int     j, k;
        double  ar, ai;
        if (r != 1 && c != 1)
```

```
                error("Unable to downshift a matrix");
        if (r == 1 && c == 1)
                error("Unable to downshift a scalar");
        if (r == 1)
                j = c;
        if (c == 1)
                j = r;
        for (k = 0; k != j - 1; k++) {

                ar = (*(adr + k * 2 + 2));
                ai = (*(adr + k * 2 + 3));
                *(adr + k * 2) = ar;
                *(adr + k * 2 + 1) = ai;
        }
        *(adr + j) = 0;

        *(adr + j + 1) = 0;

}

void inv(adr1, adr2, r1, c1, r2, c2)
double  *adr1, *adr2;
int     r1, r2, c1, c2;
{
        int     n, m, j,ll,l,piv;
        double  factor[2], a[2], t[2];
        double  *temp,big,mod,tmp;
        if (r1 != c1)
                error("Cannot invert a non-square matrix");
        if (r2 != r1)
                error("Result matrix of incorrect size in inverse");
        if (c2 != c1)
                error("Result matrix of incorrect size in inverse");
    if (c1 == 1 && r1==1) {
        cinv(adr1[0],adr1[1],adr2);

        return;
        }
        temp = (double *)malloc(sizeof(double)*r1 * c1 * 2);
        if (temp == 0)

                error("Out of Memory Error");
        for (n = 0; n != r1; n++) {

                for (m = 0; m != c1; m++) {

                        *(temp + (r1 * m * 2) + n * 2) = (*(adr1
                                + (r1 * m * 2) + n * 2));
                        *(temp + (r1 * m * 2) + n * 2 + 1) = (*(adr1
                                + (r1 * m * 2) + n * 2 + 1));
                }
        }
        iden(adr2, r2, c2);
        for (n = 0; n != r1; n++) {


        /*Find a partial pivot*/
                big=0.0;

                for(l=n;l!=r1;l++) {
                    mod=*(temp+
                        l*r1*2)+n*2) * (*(temp + (l*r1*2)+n*2));
                    mod+=*(temp+
                        l*r1*2)+n*2+1) * (*(temp + (l*r1*2)+n*2+1));
                      if (mod>big) {
                                big=mod;
                                piv=l;
```

```
                    }
                }
if (big==0.0) {
                    error("Singular Matrix Error");
                    free(temp);
                    }
/*Now we need to swap rows n and piv of the matrix*/
if (piv!=n) {
        for(ll=0;ll!=c1;ll++) {
                    tmp=*(temp+(piv*r1*2)+ll*2);
                    *(temp+(piv*r1*2)+ll*2)=(*(temp+(n*r1*2)+ll*2));
                    *(temp+(n*r1*2)+ll*2)=tmp;
                    tmp=*(temp+(piv*r1*2)+ll*2+1);
                    *(temp+(piv*r1*2)+ll*2+1)=
                                        *(temp+(n*r1*2)+ll*2+1));
                    *(temp+(n*r1*2)+ll*2+1)=tmp;
                    tmp=*(adr2+(piv*r1*2)+ll*2);
                    *(adr2+(piv*r1*2)+ll*2)=(*(adr2+(n*r1*2)+ll*2));
                    *(adr2+(n*r1*2)+ll*2)=tmp;
                    tmp=*(adr2+(piv*r1*2)+ll*2+1);
                    *(adr2+(piv*r1*2)+ll*2+1)=
                                        (*(adr2+(n*r1*2)+ll*2+1));
                    *(adr2+(n*r1*2)+ll*2+1)=tmp;
        }
}
        cinv(*(temp + (n * c1 * 2) + n * 2), *(temp + (n
             *c1 * 2) + n * 2 + 1), factor);
        for (j = 0; j != c1; j++) {
                    cmul(factor[0], factor[1], *(temp + (n *
                         c1 * 2) + j * 2), *(temp + (n * c1 * 2)
                          + j * 2 + 1), temp + (n * c1 * 2) + j *
                         2);
                    cmul(factor[0], factor[1], *(adr2 + (n *
                         c2 * 2) + j * 2), *(adr2 + (n * c2 * 2)
                          + j * 2 + 1), adr2 + (n * c2 * 2) + j *
                         2);
}
        for (m = 0; m != r1; m++) {
                    if (m != n) {
                            factor[0] = (*(temp + (m * c1 *
                                 2) + n * 2));
                            factor[1] = (*(temp + (m * c1 *
                                 2) + n * 2 + 1));
                            for (j = 0; j != c1; j++) {
                                    a[0] = (*(temp + (n * c1
                                         *2) + j * 2));
                                    a[1] = (*(temp + (n * c1
                                         *2) + j * 2 + 1));
                                    cmul(a[0], a[1], factor[0],
                                         factor[1], t);
                                    cadd(*(temp + (m * c1 *
                                         2) + j * 2), *(temp + (m
                                         *c1 * 2) + j * 2 + 1), -t[0],
                                         -t[1], temp + (m * c1 *
```

```
                                                    2) + j  * 2);
                                                a[0] = (*(adr2 + (n  * c2

                                                    *2) + j  * 2));
                                                a[1] = (*(adr2 + (n  * c2
                                                    *2) + j  * 2 + 1));
                                                cmul(a[0], a[1], factor[0],

                                                    factor[1], t);
                                                cadd(*(adr2 + (m  * c2  *
                                                    2) + j  * 2), *(adr2 + (m
                                                    *c2  * 2) + j  * 2 + 1), -t[0],

                                                    -t[1], adr2 + (m  * c2  *
                                                    2) + j  * 2);
                                        }
                                }
                        }
                }
free(temp);
}

void input(filename, ptr)
char    *filename[];
double  *ptr;
{
        static char     fname[10][20];

        static FILE *fopen(), *fpointer[10];

        float   c, d;
        int     a, f = 0, b = 10;

        if (strcmp(filename, "close") == 0) {

                for (a = 0; a != 10; a++) {

                        if (fname[a][0] != 0) {

                                fclose(fpointer[a]);
                                fname[a][0] = 0;

                        }
                }
                return;
        }

        for (a = 9; a != -1; a--) {
                if (strcmp(filename, fname[a]) == 0) {

                        f = 1;
                        b = a;
                } else {
                        if (fname[a][0] == 0)

                                b = a;
                }
        }
        if (b == 10)

                error("Too many files open (maximum of 10)");

        if (f == 0) {

                strcpy(fname[b], filename);
                fpointer[b] = fopen(fname[b], "r");
                if (fpointer[b] == 0)
```

```
                        error("Unable to open file for input");
                }
                fscanf(fpointer[b], "%f %f", &c, &d);
                *ptr = c;
                *(ptr + 1) = d;
}

void output(filename, ptr)
char    *filename[];
double  *ptr;
{
        static char     oname[10][20];

        static FILE *fopen(), *opointer[10];

        int     a, f = 0, b = 10;

        if (strcmp(filename, "close") == 0) {

                for (a = 0; a != 10; a++) {

                        if (oname[a][0] != 0) {

                                fclose(opointer[a]);
                                oname[a][0] = 0;

                        }
                }
                return;
        }

        for (a = 9; a != -1; a--) {
                if (strcmp(filename, oname[a]) == 0) {

                        f = 1;
                        b = a;
                } else {
                        if (oname[a][0] == 0)

                                b = a;
                }
        }
        if (b == 10)

                error("Too many files open (Maximum of 10)");

        if (f == 0) {

                strcpy(oname[b], filename);
                opointer[b] = fopen(oname[b], "w");
                if (opointer[b] == 0)

                        error("Unable to open file for output");
        }
        fprintf(opointer[b], "%f %f\n", (*ptr), (*(ptr + 1)));
        return;
}


void error(message)
char    *message[];
{
        int *t;
        t=0;

        fprintf(stderr, "***DSPSIM Runtime Error***\n");
        fprintf(stderr, "%s\n\n", message);
```

```
        exit(0);
}


void cadd(a, b, c, d, e)
double  a, b, c, d, *e;
{
        (*e) = radd(a, c);
        (*(e + 1)) = radd(b, d);
}

void cmul(a, b, c, d, e)
double  a, b, c, d, *e;
{
        (*e) = radd(rmul(a, c), -rmul(b, d));
        (*(e + 1)) = radd(rmul(b, c), rmul(d, a));
}


void cinv(a, b, c)
double  a, b, *c;
{
        (*c) = rdiv(a, radd(rmul(a, a), rmul(b, b)));
        (*(c + 1)) = rdiv(-b, radd(rmul(a, a), rmul(b, b)));
}


double  radd(a, b)
double  a, b;
{
        double  result;
        result = a + b;
        return (result);
}


double  rmul(a, b)
double  a, b;
{
        double  result;
        result = a * b;
        return (result);
}

double  rdiv(a, b)
double  a, b;
{
        double  result;
        result = a / b;
        return (result);
}
```

Preprocessor for matrix operations

```
/*
 * A pre-preprocessor for 'C' which converts matrix expressions into /*
 * /*sptools.h code
 */

#include <ctype.h>
#include <stdio.h>
#define TRUE 1
#define FALSE 0

void            error();
void            lerror();
void            process();
void            extract();
void            strip();
void            mul();
int             line_number = 0;

int             floating_flag = 1;
int             level;

main(argc, argv)                                                    main
        char            *argv[];
        int             argc;
{
        char            out_name[40];

        char            linebuffer[256], ch;
        int             j, k;
        FILE            *fopen(), *fp_in, *fp_out;
        if (argc != 2)
                error("ppr usage incorrect ... use ppr filename");
        fp_in = fopen(argv[1], "r");
        if (fp_in == 0)

                error("file does not exist");
        strcpy(out_name, argv[1]);
        k = 0;

        while (*(out_name + k) != 0)

                k++;
        k--;
        if (*(out_name + k) != 'p')
                error("File to be processed must be a .p file");
        *(out_name + k) = 'c';
        fp_out = fopen(out_name, "w");
        if (fp_out == 0)

                error("Unable to open a .c file for output");
        k = 0;

        while (!feof(fp_in)) {
                fscanf(fp_in, "%c", &ch);
                if (ch != '\n') {
```

```
                linebuffer[k] = ch;
                k++;
            } else {
                linebuffer[k] = '\0';

                line_number++;
                if (linebuffer[0] != '$') {

                    fprintf(fp_out, "%s\n", linebuffer);
                    if (contains(linebuffer, "#include") && contains(linebuffer, "tools.h")) {
                        fprintf(fp_out, "spvar TEMP0,TEMP10,TEMP20;\n");

                        fprintf(fp_out, "spvar TEMP1,TEMP11,TEMP21;\n");
                        fprintf(fp_out, "spvar TEMP2,TEMP12,TEMP22;\n");
                        fprintf(fp_out, "spvar TEMP3,TEMP13,TEMP23;\n");
                        fprintf(fp_out, "spvar TEMP4,TEMP14,TEMP24;\n");
                        fprintf(fp_out, "spvar TEMP5,TEMP15,TEMP25;\n");
                        fprintf(fp_out, "spvar TEMP6,TEMP16,TEMP26;\n");
                        fprintf(fp_out, "spvar TEMP7,TEMP17,TEMP27;\n");
                        fprintf(fp_out, "spvar TEMP8,TEMP18,TEMP28;\n");
                        fprintf(fp_out, "spvar TEMP9,TEMP19,TEMP29;\n");
                    }
                    if (contains(linebuffer, "#define")
                        && contains(linebuffer, "FIXED"))
                        floating_flag = 0;

                } else {
                    linebuffer[k] = '\0';

                    strip(linebuffer);
                    process(linebuffer, strlen(linebuffer), fp_out);
                }
                k = 0;

            }
        }

        fclose(fp_in);
        fclose(fp_out);
}

void
error(mesg)                                                    error
        char            *mesg[];
{
        fprintf(stderr, "%s\n", mesg);
        exit(0);

}

void
lerror(mesg)                                                   lerror
        char            *mesg[];
{
        fprintf(stderr, "%s at line %d\n", mesg, line_number);
        exit(0);

}

void
process(strptr, length, fp)                                    process
        char            *strptr;
        int             length;
        FILE            *fp;
{
        int             res_start = -1, res_finish = -1;
        char            result[256], ev[256];
```

```
        int                     k;
        k = 1;
        while (*(strptr + k) != '=') {
            if (*(strptr + k) != ' ' && res_start == -1)
                res_start = k;
            if (*(strptr + k) != ' ' && res_start != -1)
                res_finish = k;
            ++k;
            if (k == length)
                lerror("No equals sign in expression");
        }
        if (res_start == -1 || res_finish == -1)
            lerror("No result variable in expression");
        res_finish++;
        if (!(isvarch(*(strptr + res_finish + 1))
                || *(strptr + res_finish + 1) == '('))
            lerror("Bad expression");

        extract(strptr, result, res_start, res_finish);
        extract(strptr, ev, res_finish + 1, length);
        level = -1;
        eval(ev, fp);
        fprintf(fp, "/*%d*/ copy (%s,%s);\n", line_number, ev, result);
}

void                                                          extract
extract(strptr1, strptr2, start, finish)
        char                    *strptr1, *strptr2;
        int                     start, finish;
{
        int                     k;
        if (start < 0)

            start = 0;

        if (finish < 0)

            finish = 0;

        if (finish > strlen(strptr1))
            finish = strlen(strptr1);
        for (k = 0; k != finish - start; k++) {

            *(strptr2 + k) = *(strptr1 + k + start);
        }
        *(strptr2 + finish - start) = '\0';

}

void                                                          strip
strip(strptr)
        char                    *strptr;
{
        char                    stripped[256];
        int                     k, l;
        k = 0;

        l = 0;

        while (*(strptr + k) != '\0') {

            if (!isspace(*(strptr + k))) {
                stripped[l] = *(strptr + k);
                l++;
            }
            k++;
        }
```

```
        stripped[l] = '\0';

        strcpy(strptr, stripped);
}

eval(strptr, fp)                                                    eval
        char            *strptr;
        FILE            *fp;
{
        char            local_copy[256], temp[256], temp1[256], temp2[256];
        int             l, l1, k0, k1, k2, k3, k4, k5, k6, brac_count;

        strcpy(local_copy, strptr);

SEARCH_LOOP:k1 = search(local_copy, '(', 0);

        k2 = search(local_copy, '\'', 0);

        k0 = search(local_copy, '"', 0);

        k3 = search(local_copy, '#', 0);

        k4 = search(local_copy, '*', 0);

        k5 = search(local_copy, '+', 0);

        k6 = search(local_copy, '-', 0);


        if (level == 30)

                error("Expression too complicated");
        if (k0 == -1 && k1 == -1 && k2 == -1 &&

                k3 == -1 && k4 == -1 && k5 == -1 && k6 == -1) {
                if (search(local_copy, ')', 0) != -1)

                        lerror("Unbalanced brackets");
                else {
                        strcpy(strptr, local_copy);
                        return (0);

                }
        }
        if (k1 != -1) {
                brac_count = 1;
                l = k1 + 1;
                while (brac_count != 0 && local_copy[l] != '\0') {

                        if (local_copy[l] == '(')
                                brac_count++;
                        if (local_copy[l] == ')')
                                brac_count--;
                        l++;
                }
                if (brac_count != 0)

                        lerror("Unbalanced brackets");
                extract(local_copy, temp, k1 + 1, l - 1);
                eval(temp, fp);         /* The recursive bit to deal with brackets! */
                extract(local_copy, temp1, 0, k1);

                extract(local_copy, temp2, l, 1 + strlen(local_copy));
                strcat(temp, temp2);
                strcat(temp1, temp);
                strcpy(local_copy, temp1);
                goto SEARCH_LOOP;
```

```
}
if (k2 != -1) {
     level++;
     sprintf(temp1, "TEMP%d\0", level);

     l = k2 - 1;
     while (l != -1 && isvarch(local_copy[l]))
          l--;
     extract(local_copy, temp, l + 1, k2);
     if (strlen(temp) == 0)

          lerror("Bad expression");
     if (floating_flag)
          fprintf(fp, "/*%d*/ resize(%s,%s.csize,%s.rsize);\n"
               ,line_number, temp1, temp, temp);
     if (!floating_flag)
          fprintf(fp, "/*%d*/ resize(%s,%s.csize,%s.rsize,%s.format);\n"
               ,line_number, temp1, temp, temp, temp);
     fprintf(fp, "/*%d*/ transpconj(%s,%s);\n", line_number, temp, temp1);
     extract(local_copy, temp2, 0, l + 1);

     strcat(temp2, temp1);
     extract(local_copy, temp1, k2 + 1, 1 + strlen(local_copy));
     strcat(temp2, temp1);
     strcpy(local_copy, temp2);
     goto SEARCH_LOOP;
}
if (k0 != -1) {

     level++;
     sprintf(temp1, "TEMP%d\0", level);

     l = k0 - 1;

     while (l != -1 && isvarch(local_copy[l]))
          l--;
     extract(local_copy, temp, l + 1, k0);

     if (strlen(temp) == 0)

          lerror("Bad expression");
     if (floating_flag)
          fprintf(fp, "/*%d*/ resize(%s,%s.csize,%s.rsize);\n"
               ,line_number, temp1, temp, temp);
     if (!floating_flag)
          fprintf(fp, "/*%d*/ resize(%s,%s.csize,%s.rsize,%s.format);\n"
               ,line_number, temp1, temp, temp, temp);
     fprintf(fp, "/*%d*/ transpose(%s,%s);\n", line_number, temp, temp1);
     extract(local_copy, temp2, 0, l + 1);

     strcat(temp2, temp1);
     extract(local_copy, temp1, k0 + 1, 1 + strlen(local_copy));

     strcat(temp2, temp1);
     strcpy(local_copy, temp2);
     goto SEARCH_LOOP;
}
if (k3 != -1) {
     level++;
     sprintf(temp1, "TEMP%d\0", level);

     l = k3 - 1;
     while (l != -1 && isvarch(local_copy[l]))
          l--;
     extract(local_copy, temp, l + 1, k3);
     if (strlen(temp) == 0)
```

```
                lerror("Bad expression");
            if (strcmp(temp, temp1) == 0)

                sprintf(temp, "TEMP1_%d\0", level);

            if (floating_flag)
                fprintf(fp, "/*%d*/ resize(%s,%s.rsize,%s.csize);\n"
                    ,line_number, temp1, temp, temp, temp);
            if (!floating_flag)
                printf(fp, "/*%d*/ resize(%s,%s.rsize,%s.csize,%s.format);\n"
                    ,line_number, temp1, temp, temp, temp);
            fprintf(fp, "/*%d*/ inverse(%s,%s);\n", line_number, temp, temp1);
            extract(local_copy, temp2, 0, l + 1);

            strcat(temp2, temp1);
            extract(local_copy, temp1, k3 + 1, 1 + strlen(local_copy));
            strcat(temp2, temp1);
            strcpy(local_copy, temp2);
            goto SEARCH_LOOP;
        }
        if (k4 != -1) {
            level++;
            sprintf(temp2, "TEMP%d\0", level);

            l = k4 - 1;
            l1 = k4 + 1;
            while (l != -1 && (isvarch(local_copy[l]) || local_copy[l] == '.'))
                l--;
            while (local_copy[l1] != '\0' && (isvarch(local_copy[l1])
                                    || local_copy[l1] == '.'))
                l1++;
            extract(local_copy, temp, l + 1, k4);
            extract(local_copy, temp1, k4 + 1, l1);
            if (strlen(temp) == 0 || strlen(temp1) == 0)

                lerror("Bad expression");
            if (test_number(temp) || test_number(temp1))
                goto scalar_multiply;
            if (floating_flag)
                fprintf(fp, "/*%d*/ resize(%s,number_of_rows(%s.rsize,%s.csize,%s.rsize,%s.c
size),number_of_columns(%s.rsize,%s.csize,%s.rsize,%s.csize));\n", line_number,
  temp2, temp, temp, temp1, temp1, temp, temp, temp1, temp1);
            if (!floating_flag)
                fprintf(fp, "/*%d*/ resize(%s,number_of_rows(%s.rsize,%s.csize,%s.rsize,%s.c
size),number_of_columns(%s.rsize,%s.csize,%s.rsize,%s.csize),%s.format);\n", li
ne_number, temp2, temp, temp, temp1, temp1, temp, temp, temp1, temp1, temp1);
            fprintf(fp, "/*%d*/ multiply(%s,%s,%s);\n", line_number, temp, temp1, temp2);

            extract(local_copy, temp1, 0, l + 1);

            strcat(temp1, temp2);
            extract(local_copy, temp2, l1, 1 + strlen(local_copy));
            strcat(temp1, temp2);
            strcpy(local_copy, temp1);
            goto SEARCH_LOOP;

scalar_multiply:
            if (test_number(temp) && test_number(temp1)) {
                mul(temp, temp1, temp2);
                extract(local_copy, temp1, 0, l + 1);

                strcat(temp1, temp2);
                extract(local_copy, temp2, l1, 1 + strlen(local_copy));
                strcat(temp1, temp2);
                strcpy(local_copy, temp1);
                goto SEARCH_LOOP;
```

```
            }
            if (test_number(temp)) {
                if (floating_flag)
                    fprintf(fp, "/*%d*/ resize(%s,%s.rsize,%s.csize);\n", line_number, temp2, t
emp1, temp1);
                if (!floating_flag)
                    fprintf(fp, "/*%d*/ resize(%s,%s.rsize,%s.csize,%s.format);\n", line_number
, temp2, temp1, temp1, temp1);
                fprintf(fp, "/*%d*/ multscal(%s,%s,0.0,%s);\n", line_number, temp1, temp, te
mp2);
            }
            if (test_number(temp1)) {
                if (floating_flag)
                    fprintf(fp, "/*%d*/ resize(%s,%s.rsize,%s.csize);\n", line_number, temp2, t
emp, temp);
                if (!floating_flag)
                    fprintf(fp, "/*%d*/ resize(%s,%s.rsize,%s.csize,%s.format);\n", line_number
, temp2, temp, temp, temp);
                fprintf(fp, "/*%d*/ multscal(%s,%s,0.0,%s);\n", line_number, temp, temp1, te
mp2);
            }
            extract(local_copy, temp1, 0, l);

            strcat(temp1, temp2);
            extract(local_copy, temp2, l1, 1 + strlen(local_copy));
            strcat(temp1, temp2);
            strcpy(local_copy, temp1);
            goto SEARCH_LOOP;
        }
        if (k5 != -1) {
            level++;
            sprintf(temp2, "TEMP%d\0", level);

            l = k5 - 1;
            l1 = k5 + 1;
            while (l != 0 && isvarch(local_copy[l]))

                l--;
            while (local_copy[l1] != '\0' && isvarch(local_copy[l1]))

                l1++;
            extract(local_copy, temp, l, k5);
            extract(local_copy, temp1, k5 + 1, l1);
            if (strlen(temp) == 0 || strlen(temp1) == 0)

                lerror("Bad expression");

            if (floating_flag)
                fprintf(fp, "/*%d*/ resize(%s,%s.rsize,%s.csize);\n", line_number, temp2, te
mp, temp);
            if (!floating_flag)
                fprintf(fp, "/*%d*/ resize(%s,%s.rsize,%s.csize,%s.format);\n", line_number,
 temp2, temp, temp, temp);
            fprintf(fp, "/*%d*/ add(%s,%s,%s);\n", line_number, temp, temp1, temp2);
            extract(local_copy, temp1, 0, l);

            strcat(temp1, temp2);
            extract(local_copy, temp2, l1, 1 + strlen(local_copy));
            strcat(temp1, temp2);
            strcpy(local_copy, temp1);
            goto SEARCH_LOOP;
        }
        if (k6 != -1) {
            level++;
            sprintf(temp2, "TEMP%d\0", level);

            l = k6 - 1;
```

```
            l1 = k6 + 1;
            while (l != 0 && isvarch(local_copy[l]))

                l--;
            while (local_copy[l1] != '\0' && isvarch(local_copy[l1]))

                l1++;
            extract(local_copy, temp, l, k6);
            extract(local_copy, temp1, k6 + 1, l1);
            if (strlen(temp) == 0 || strlen(temp1) == 0)

                lerror("Bad expression");
            if (floating_flag)
                fprintf(fp, "/*%d*/ resize(%s,%s.rsize,%s.csize);\n", line_number, temp2, te
mp, temp);
            if (!floating_flag)
                fprintf(fp, "/*%d*/ resize(%s,%s.rsize,%s.csize,%s.format);\n", line_number,
 temp2, temp, temp, temp);
            fprintf(fp, "/*%d*/ subtract(%s,%s,%s);\n", line_number, temp, temp1, temp2);

            extract(local_copy, temp1, 0, l);

            strcat(temp1, temp2);
            extract(local_copy, temp2, l1, 1 + strlen(local_copy));
            strcat(temp1, temp2);
            strcpy(local_copy, temp1);
            goto SEARCH_LOOP;
        }
}

int
search(strptr, ch, k)                                          search
        char            *strptr, ch;
        int              k;
{
        int              l;
        l = -1;
        while (*(strptr + k) != '\0') {

            if (l == -1 && *(strptr + k) == ch)
                l = k;
            k++;
        }
        return (l);
}

contains(strptr1, strptr2)                                     contains
        char            *strptr1, *strptr2;
{
        int              p, flag = 0;

        if (strlen(strptr2) > strlen(strptr1))
            return (flag);
        for (p = 0; p != 1 + strlen(strptr1) - strlen(strptr2); p++)

            if (strncmp(strptr1 + p, strptr2, strlen(strptr2)) == 0)

                flag = 1;
        return (flag);
}

test_number(strptr)                                            test_number
        char            *strptr;
{
        int              k, flag = 1;
        for (k = 0; k != strlen(strptr); k++)
```

```
            if (!(isdigit(*(strptr + k)) || *(strptr + k) == '.'))
                flag = 0;

        return (flag);
}

void
mul(strptr1, strptr2, strptr3)                                    mul
        char            *strptr1, *strptr2, *strptr3;
{
        double          x, y;
        sscanf(strptr1, "%lf", &x);
        sscanf(strptr2, "%lf", &y);
        sprintf(strptr3, "%lf\0", x * y);

}

isvarch(ch)                                                      isvarch
        char            ch;
{
        if (isalnum(ch))
            return (1);
        if (ch == '_')
            return (1);
        else
            return (0);

}
```

Conventional RLS simulation

```
/*RLS Algorithm*/

#define FLOATING
#define MANTISSA_LENGTH 56
#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include < /u4 /call /lib /COMPLEX_src /sptools.h>

#define MAXRND 2147483647.0

#define UNKNOWN_LENGTH 5

double calcnte();
double rnum();
double gauss();
double inp,desired;
double XW[UNKNOWN_LENGTH];
double Weights[UNKNOWN_LENGTH]={0.9,0.3,-0.3,0.7,0.1};

double FEED_FORWARD[2]={1.0,0.865};

double XK[2];
float NOISE;
int N=0;



#define sigma 0.01


main()                                                    main
{
spvar LAM,X,P,W,K,D,E;
struct complex cn;

FILE *fopen(),*fp;
double nte,gain_factor=0.0,*average;

float lambda=0.0,SNR=-1;

int n,s,p_10,p,ensemble=-1,ens;

char clear_screen=12;
char up_line=11;
char *command,*arg1,*arg2,*arg3,*arg4;
char o_file[20];


/*Initialisation*/

fp=fopen("NORMTAPERROR.DAT","w");


srandom(1);
```

```
printf("%c",clear_screen);
printf("Simulation of Standard RLS Algorithm\n\n");
printf("by Chris Callender, 1989\n\n\n\n");

printf("Floating Point Mantissa Length = %d\n\n",MANTISSA_LENGTH);
while(N<1) {
        printf("Filter Length:");
        scanf("%d",&N);
}
cvector(X,N);
matrix(P,N,N);
cvector(K,N);
cvector(W,N);
scalar(D);
scalar(E);
scalar(LAM);

printf("\n\n");
while (lambda<0.8 || lambda>1.0)

{
printf("Please enter a value for lambda between 0.8 and 1.0: ");

scanf("%f",&lambda);
}
setscalar(LAM,lambda,0.0);


while (SNR<0 || SNR>120) {

printf("\n\nPlease enter SIGNAL/NOISE ratio in dB (0 - 120db): ");

scanf("%f",&SNR);
}

while (ensemble<1) {
printf("\n\nHow many runs to make ensemble average: ");
scanf("%d",&ensemble);
}

for(n=0;n!=N;n++) gain_factor=gain_factor+Weights[n]*Weights[n];

gain_factor=gain_factor+FEED_FORWARD[0]*FEED_FORWARD[0]+FEED_FORWARD[1]*FEED_FO
RWARD[1];
gain_factor=sqrt(gain_factor);

NOISE=gain_factor/exp10(SNR/20.0);


printf("\n\nHow many data samples per run: ");
scanf("%d",&s);
fprintf(fp,"%d\n\n",s);
average=(double *)malloc(sizeof(double)*(s+1));
if (average==0) {

        fprintf(stderr,"RLS Runtime error...out of memory");
        exit(1);
        }

printf("%cRLS Simulation Running\n\n\n",clear_screen);
p_10=s/10;

for (ens=1;ens!=ensemble+1;ens++) {

for(n=0;n!=N;n++) XW[n]=0.0;

for(n=0;n!=2;n++) XK[n]=0.0;
```

```
zero(W);
zero(X);
identity(P);
multscal(P,1.0/sigma,0.0,P);

p=p_10;


nte=calcnte(W.element);
*average=*average+nte/ensemble;

for(n=1;n!=s+1;n++) {

if (n==p) {
        printf("%cRun #%d:Status %d%%\n",up_line,ens,(p*10)/p_10);

        p=p+p_10;

        }

        /*if (n==s/2) {
              Weights[0]=0.3;

              Weights[1]=0.7;

              Weights[2]=-0.6;

              Weights[3]=0.2;

              Weights[4]=-1.2;
              }*/

        upshift(X);
        makedata();
        setcvector(X,1,inp,0.0);



        setscalar(D,desired,0.0);

$       E=D - X' * W
$       K=P*X*((LAM+X'*P*X)#)
$       P=(P - K*X'*P)*LAM#
$       W=W + K * E

*(average+n)=*(average+n)+calcnte(W.element)/ensemble;
        }
printf("\n");
}
for(n=0;n!=s+1;n++) {

        fprintf(fp,"%20.16e\n",*(average+n));

        }

fclose(fp);
makedB(s,average);

fp=fopen("gplottext.tmp","w");
fprintf(fp,"Fl. Point\n");
fprintf(fp,"%d bits\n\n",MANTISSA_LENGTH);
fprintf(fp,"Fil Len=%d\n",N);
fprintf(fp,"lam=%3.2f\n",lambda);
fprintf(fp,"SNR=%4.2f\n",SNR);
fprintf(fp,"%d runs\n",ensemble);
fclose(fp);
```

```
}

double calcnte(ptr)
double *ptr;
{
int k;
double nte;
struct complex Weight;


for (k=0;k!=N;k++) {

        Weight.real=((*ptr));
        ptr++;
        Weight.imaginary=((*ptr));
        ptr++;
        nte=nte+(Weight.real-Weights[k])*(Weight.real-Weights[k]);
        }
return(nte);
}


makedata()
{
int j;

        XK[1]=XK[0];

        XK[0]=gauss();

        inp=XK[0]*FEED_FORWARD[0]+XK[1]*FEED_FORWARD[1];

        for (j=UNKNOWN_LENGTH-1;j!=0;j--) {

                XW[j]=XW[j-1];
                }
        XW[0]=inp;

        desired=0.0;

        for (j=0;j!=UNKNOWN_LENGTH;j++) desired=desired+XW[j]*Weights[j];

        desired=desired+(gauss())*NOISE;
}


double rnum()
{
return ((random()/MAXRND));
}

double gauss()
{
double a,b;
double result;
a=rnum();
b=rnum();
result=sqrt(-2*log(a))*cos(2*3.141592654*b);
return(result);
}


makedB(s,data)
int s;
double *data;
{
FILE *fopen(),*fp2;
```

```
float se;
float init;
double p;
int k;

fp2=fopen("ERRdB.DAT","w");


for (k=0;k!=s+1;k++) {

        se=*(data+k);
        if (k==0) init=se;

        p=10*log10(se/init);

        fprintf(fp2,"%f\n",p);
        }

fclose(fp2);
}
```

Fast Transversal Filters simulation

```
/*FTF Algorithm*/

#define FLOATING
#define MANTISSA_LENGTH 56
#include </u4/call/lib/COMPLEX/sptools.h>
#include <math.h>
#include <stdio.h>

#define MAXRND 2147483647.0


int    N;
double calcnte();
double rnum();
double gauss();
void change_weights();
double inp,desired;
double X[16];
double Weights[16];
double FEED_FORWARD[3]={1.0,0.600};

double XK[3];
float NOISE;



main(argc,argv)
int argc;
char *argv[];
{
spvar A,rescue,y0,alpham1,eNp,eN,gammaN,gammaNp1,epsilon,epsilonp;

spvar tempscal1,tempscal2,rN,rNp,beta,Y,YNp1,C,Cex,B,tempN,W,alpha,CNp1;
struct complex cn;
FILE *fopen(),*fp;
double nte.gain_factor=0.0,*average,temp,MU=-1.0;

float lambda=0.0,SNR=-1;

int k,n,s,p_10,p,ensemble=-1,ens,res_flag;

char clear_screen=12;
char up_line=11;

/*Initialisation*/
if (argc!=2) res_flag = 1;
else {
      if (strcmp(argv[1],"-on")==0) res_flag=1;

      if (strcmp(argv[1],"-ON")==0) res_flag=1;

      if (strcmp(argv[1],"-off")==0) res_flag=0;

      if (strcmp(argv[1],"-OFF")==0) res_flag=0;

      }
```

```
printf("%c",clear_screen);
printf("Simulation of FTF Algorithm\n\n");
printf("by Chris Callender, 1989\n\n\n\n");
if (res_flag) printf("\n\nRescue=ON, Floating Point Mantissa Length = %d\n\n",M
ANTISSA_LENGTH);
if (!res_flag) printf("\n\nRescue=OFF Floating Point Mantissa Length = %d\n\n",
MANTISSA_LENGTH);
printf("Filter Length:");
scanf("%d",&N);
for (k=0;k!=N;k++) Weights[k]=rnum()*2.0-1.0;

if (N<16) for (k=N;k!=16;k++) Weights[k]=0.0;

rvector(A,(N+1));
scalar(rescue);
scalar(y0);

scalar(alpham1);
scalar(eNp);
scalar(eN);
scalar(gammaN);
scalar(gammaNp1);
scalar(alpha);
scalar(epsilon);
scalar(epsilonp);
scalar(tempscal1);
scalar(tempscal2);
scalar(rN);
scalar(rNp);
scalar(beta);
cvector(Y,N);
cvector(YNp1,(N+1));
rvector(C,N);
rvector(Cex,(N+1));
rvector(CNp1,(N+1));
rvector(B,(N+1));
rvector(tempN,N);
rvector(W,N);

fp=fopen("NORMTAPERROR.DAT","w");

srandom(time(0));


while (lambda<0.8 || lambda>1.0)

{
printf("\n\nPlease enter a value for lambda between 0.8 and 1.0: ");

scanf("%f",&lambda);
}

while(MU<0 && res_flag) {

printf("\n\nPlease enter a value for soft constraint parameter MU:");
scanf("%lf",&MU);
}

while (SNR<0 || SNR>120) {

printf("\n\nPlease enter SIGNAL/NOISE ratio in dB (0 - 120db): ");

scanf("%f",&SNR);
}

while (ensemble<1) {
```

```
printf("\n\nHow many runs to make ensemble average: ");
scanf("%d",&ensemble);
}

for(n=0;n!=N;n++) gain_factor=gain_factor+Weights[n]*Weights[n];

gain_factor=gain_factor*(FEED_FORWARD[0]*FEED_FORWARD[0]+FEED_FORWARD[1]*FEED_F
ORWARD[1]+FEED_FORWARD[2]*FEED_FORWARD[2]);
gain_factor=sqrt(gain_factor);

NOISE=gain_factor/exp10(SNR/20.0);


printf("\n\nHow many data samples per run: ");
scanf("%d",&s);
fprintf(fp,"%d\n\n",s);
average=(double *)malloc(sizeof(double)*(s+1));
if (average==0) {

        fprintf(stderr,"FTF Runtime error...out of memory");
        exit(1);
        }
printf("%cFTF Simulation Running\n\n\n",clear_screen);
p_10=s/10;

for (ens=1;ens!=ensemble+1;ens++) {
for(n=0;n!=N;n++) X[n]=0.0;

for(n=0;n!=3;n++) XK[n]=0.0;



setrvector(A,1,1.0,0.0);

setrvector(B,1,1.0,0.0);

zero(C);
zero(W);
nte=calcnte(W.element);
*average=*average+nte/ensemble;
makedata();
setscalar(y0,inp,0.0);

copy(y0,tempscal1);

setscalar(tempscal2,-desired,0.0);

copy(tempscal1,alpha);
multiply(alpha,alpha,alpha);
copy(alpha,alpham1);
inverse(tempscal1,tempscal1);.
multiply(tempscal1,tempscal2,tempscal1);
getscalar(tempscal1,cn);
setrvector(W,1,cn.real,cn.imaginary);
setscalar(gammaN,1.0,0.0);

p=p_10;



for(n=1;n!=N+1;n++) {
        nte=calcnte(W.element);
        *(average+n)=*(average+n)+nte/ensemble;
        makedata();
        upshift(Y);
        upshift(YNp1);
        setcvector(Y,1,inp,0.0);
```

```
setcvector(YNp1,1,inp,0.0);


multiply(A,YNp1,eNp);

for(k=1;k!=N+1;k++) {
      getrvector(A,k,cn);
      setrvector(tempN,k,cn.real,cn.imaginary);
      }
inverse(y0,tempscal1);

multiply(eNp,tempscal1,tempscal1);
getscalar(tempscal1,cn);
setrvector(A,n+1,-cn.real,-cn.imaginary);

multiply(eNp,gammaN,eN);

multscal(alpha,lambda,0.0,alpha);


multiply(eNp,eN,tempscal1);
add(alpha,tempscal1,alpham1);

inverse(alpham1,tempscal1);
multiply(tempscal1,alpha,tempscal1);
multiply(gammaN,tempscal1,gammaN);

upshift(C);
inverse(alpha,tempscal1);
multiply(tempscal1,eNp,tempscal1);
getscalar(tempscal1,cn);
multscal(tempN,cn.real,cn.imaginary,tempN);
subtract(C,tempN,C);

if (n==N) {
      copy(C,tempN);
      getscalar(y0,cn);

      multscal(tempN,cn.real,cn.imaginary,tempN);
      getscalar(gammaN,cn);
      multscal(tempN,cn.real,cn.imaginary,tempN);
      for(k=1;k!=N+1;k++) {
            getrvector(tempN,k,cn);
            setrvector(B,k,cn.real,cn.imaginary);
            }
      setrvector(B,N+1,1.0,0.0);


      copy(gammaN,beta);
      multiply(beta,y0,beta);

      multiply(beta,y0,beta);


      }

setscalar(tempscal1,desired,0.0);

multiply(W,Y,tempscal2);
add(tempscal1,tempscal2,epsilonp);

multiply(epsilonp,gammaN,epsilon);

if (n<N) {
      inverse(y0,tempscal1);

      multiply(tempscal1,epsilonp,tempscal1);
```

```
            getscalar(tempscal1,cn);
            setrvector(W,n+1,-cn.real,-cn.imaginary);
            }
        if (n==N) {
            getscalar(epsilon,cn);
            multscal(C,cn.real,cn.imaginary,tempN);
            add(W,tempN,W);
            }
        }

for(n=1;n!=s+1;n++) {

RE_START:
if (n==p) {
        printf("%cRun #%d:Status %d%%\n",up_line,ens,(p*10)/p_10);

        p=p+p_10;

        }


        makedata();
        upshift(Y);
        upshift(YNp1);
        setcvector(Y,1,inp,0.0);

        setcvector(YNp1,1,inp,0.0);



        /*    #1        */
        multiply(A,YNp1,eNp);

        /*    #2        */
        multiply(eNp,gammaN,eN);

        /*    #3        */
        copy(alpha,tempscal2);
        multiply(eNp,eN,tempscal1);
        multscal(alpha,lambda,0.0,alpha);

        add(alpha,tempscal1,alpha);


        /*    #4        */
        inverse(alpha,tempscal1);
        multiply(gammaN,tempscal1,gammaNp1);
        multiply(gammaNp1,tempscal2,gammaNp1);
        multscal(gammaNp1,lambda,0.0,gammaNp1);



        /*    #5        */
        for (k=1;k!=N+1;k++) {
            getrvector(C,k,cn);
            setrvector(Cex,k+1,cn.real,cn.imaginary);
            }
        setrvector(Cex,1,0.0,0.0);

        inverse(tempscal2,tempscal2);
        multiply(tempscal2,eNp,tempscal2);
        multscal(tempscal2,1/lambda,0.0,tempscal2);

        getscalar(tempscal2,cn);
        multscal(A,-cn.real,-cn.imaginary,CNp1);
        add(CNp1,Cex,CNp1);
```

```
/*    #6        */
getscalar(eN,cn);
multscal(Cex,cn.real,cn.imaginary,Cex);
add(Cex,A,A);

/*    #7        */
getrvector(CNp1,N+1,cn);
setscalar(rNp,-cn.real,-cn.imaginary);
multiply(rNp,beta,rNp);
multscal(rNp,lambda,0.0,rNp);


/*    #8        */
getrvector(CNp1,N+1,cn);
setscalar(tempscal1,cn.real,cn.imaginary);
multiply(tempscal1,gammaNp1,tempscal1);
multiply(tempscal1,rNp,tempscal1);
setscalar(tempscal2,1.0,0.0);

add(tempscal1,tempscal2,tempscal1);
copy(tempscal1,rescue);
inverse(tempscal1,tempscal1);
multiply(tempscal1,gammaNp1,gammaN);


getscalar(rescue,cn);
if (cn.real<0.0 && res_flag==1) {

      zero(A);
      setrvector(A,1,1.0,0.0);

      zero(B);
      setrvector(B,N+1,1.0,0.0);

      zero(C);
      temp=pow(lambda,(double )N)*MU;
      setscalar(alpha,temp,0.0);

      temp=MU;
      setscalar(beta,temp,0.0);

      setscalar(gammaN,1.0,0.0);

      goto RE_START;
      }

/*    #9        */
multiply(rNp,gammaN,rN);

/*    #10       */

multscal(beta,lambda,0.0,beta);

multiply(rNp,rN,tempscal1);
add(tempscal1,beta,beta);

/*    #11       */
getrvector(CNp1,N+1,cn);
multscal(B,-cn.real,-cn.imaginary,Cex);
add(CNp1,Cex,Cex);

for(k=1;k!=N+1;k++) {
getrvector(Cex,k,cn);
setrvector(C,k,cn.real,cn.imaginary);
}
```

```
        setrvector(Cex,N+1,0.0,0.0);


        /*      #12     */
        getscalar(rN,cn);
        multscal(Cex,cn.real,cn.imaginary,Cex);
        add(Cex,B,B);

        /*      #13     */
        setscalar(tempscal1,desired,0.0);

        multiply(W,Y,tempscal2);
        add(tempscal1,tempscal2,epsilonp);

        /*      #14     */
        multiply(epsilonp,gammaN,epsilon);

        /*      #15     */
        getscalar(epsilon,cn);
        multscal(C,cn.real,cn.imaginary,tempN);
        add(W,tempN,W);
        nte=calcnte(W.element);
        *(average+n)=*(average+n)+nte/ensemble;
}
}
for(n=0;n!=s+1;n++) {

        fprintf(fp,"%20.16e\n",*(average+n));

        }

fclose(fp);
makedB(s,average);

}

double calcnte(ptr)
double *ptr;
{
int k;
double nte=0;

struct complex Weight;


for (k=0;k!=N;k++) {

        Weight.real=((*ptr));
        ptr++;
        Weight.imaginary=((*ptr));
        ptr++;
        nte=nte+(Weight.real+Weights[k])*(Weight.real+Weights[k]);
        }
return(nte);
}


makedata()
{
int j;

        XK[2]=XK[1];
        XK[1]=XK[0];

        XK[0]=gauss();

        inp=XK[0]*FEED_FORWARD[0]+XK[1]*FEED_FORWARD[1]+XK[2]*FEED_FORWARD[2];
```

```
        for (j=N-1;j!=0;j--) {

              X[j]=X[j-1];
             }
        X[0]=inp;

        desired=0.0;

        for (j=0;j!=N;j++) desired=desired+X[j]*Weights[j];

        desired=desired+(gauss())*NOISE;
}


double rnum()
{
return ((random()/MAXRND));
}

double gauss()
{
double a,b;
double result;
a=rnum();
b=rnum();
result=sqrt(-2*log(a))*cos(2*3.141592654*b);
return(result);
}


makedB(s,data)
int s;
double *data;
{
FILE *fopen(),*fp2;
float se;
float init;
double av_level=0;

double p;
int k;

fp2=fopen("ERRdB.DAT","w");


for (k=0;k!=s+1;k++) {

        se=*(data+k);
        if (k==0) init=se;

        p=10*log10(se/init);

        if (k>4*N) av_level+=p;
        fprintf(fp2,"%f\n",p);
        }

printf("Average performance level=%lf\n",av_level/(s-4*N));
fclose(fp2);
}

void change_weights(t,file_ptr)
FILE *file_ptr;
int t;
{
}
```

Fast Kalman simulation

```
/* Simulation of the Fast Kalman Algorithm */
#define FLOATING
#define MANTISSA_LENGTH 56
#include </u4/call/lib/COMPLEX_src/sptools.h> /*tools.h*/
#include <math.h>
#include <stdio.h>

#define MAXRND 2147483647.0


int    N;
double calcnte();
double rnum();
double gauss();
void change_weights();
double inp,desired;
double X[5];
double Weights[5]={0.9,0.3,-0.3,0.7,0.1};

double FEED_FORWARD[2]={1.0,0.600};

double XK[2];
float NOISE;



main(argc,argv)
int argc;
char *argv[];
{
/* All of the spvar definitions should go in here*/
spvar Xnm1,enm1,a,c,en,epsilon,Cex,m,mu,r,b,w,err,xn,dn,forget,temp,temp1,y;
struct complex cn1,cn2;
FILE *fopen(),*fp;
double delta=-1.0,nte,gain_factor=0.0,*average;

float lambda=0.0,SNR=-1;

int k,n,s,p_10,p,ensemble=-1,ens,res_flag;

char clear_screen=12;
char up_line=11;



/*Initialisation*/
if (argc!=2) res_flag = 1;
else {
      if (strcmp(argv[1],"-on")==0) res_flag=1;

      if (strcmp(argv[1],"-ON")==0) res_flag=1;

      if (strcmp(argv[1],"-off")==0) res_flag=0;

      if (strcmp(argv[1],"-OFF")==0) res_flag=0;
```

```
        }
printf("%c",clear_screen);
printf("Simulation of Fast Kalman Algorithm\n\n");
printf("by Chris Callender, 1989\n\n\n\n");
if (res_flag) printf("\n\nRescue=ON, Floating Point Mantissa Length = %d\n\n",M
ANTISSA_LENGTH);
if (!res_flag) printf("\n\nRescue=OFF Floating Point Mantissa Length = %d\n\n",
MANTISSA_LENGTH);
printf("Filter Length:");
scanf("%d",&N);

temp.rsize=1; temp.csize=1; temp.element=(double *)malloc(sizeof(double)*2); if
 (temp.element==0) error("Out of Memory");

rvector(Xnm1,N);
scalar(temp);
cvector(temp1,N);
scalar(forget);
scalar(xn);
scalar(dn);
scalar(enm1);
scalar(en);
cvector(a,N);
cvector(b,N);
cvector(c,N);
cvector(w,N);
scalar(err);
scalar(r);
scalar(mu);
cvector(m,N);
cvector(Cex,(N+1));
scalar(epsilon);
scalar(y);

fp=fopen("NORMTAPERROR.DAT","w");


srandom(1);


while (lambda<0.8 || lambda>1.0)

{
printf("Please enter a value for lambda between 0.8 and 1.0: ");

scanf("%f",&lambda);
}

while (delta<0.0)

{
printf("Please enter a small positive value for delta: ");
scanf("%lf",&delta);
}

while (SNR<0 || SNR>120) {

printf("\n\nPlease enter SIGNAL/NOISE ratio in dB (0 - 120db): ");

scanf("%f",&SNR);
}

while (ensemble<1) {
printf("\n\nHow many runs to make ensemble average: ");
scanf("%d",&ensemble);
}
```

```
for(n=0;n!=N;n++) gain_factor=gain_factor+Weights[n]*Weights[n];

gain_factor=gain_factor*(FEED_FORWARD[0]*FEED_FORWARD[0]+FEED_FORWARD[1]*FEED_F
ORWARD[1]);
gain_factor=sqrt(gain_factor);

NOISE=gain_factor/exp10(SNR/20.0);


printf("\n\nHow many data samples per run: ");
scanf("%d",&s);
fprintf(fp,"%d\n\n",s);
average=(double *)malloc(sizeof(double)*(s+1));
if (average==0) {

        fprintf(stderr,"FK Runtime error...out of memory");
        exit(1);
        }
printf("%cFK Simulation Running\n\n\n",clear_screen);
p=p_10=s/10;


zero(w);
nte=calcnte(w.element);
*average=*average+nte/ensemble;
zero(a);
zero(b);
zero(c);
setscalar(epsilon,delta,0.0);

setscalar(forget,lambda,0.0);


for (ens=1;ens!=ensemble+1;ens++) {

for(n=1;n!=s+1;n++) {
if (n==p) {
        printf("%cRun #%d:Status %d%%\n",up_line,ens,(p*10)/p_10);

        p=p+p_10;

        }
        /*The algorithm goes in here!*/


        /*(K1)*/

        makedata();
        setscalar(xn,inp,0.0);

        setscalar(dn,desired,0.0);

$       enml=xn-(Xnml)*a

        /*(K2)*/

$       a=a+c*enml

        /*(K3)*/

$       en=xn-(Xnml)*a

        /*(K4)*/
$       epsilon=forget*epsilon+en*enml


        /*(K5)*/
```

```
$       temp=en * (epsilon#)
        getscalar(temp,cn1);
        setrvector(Cex,1,cn1.real,cn1.imaginary);
$       temp1=c - a * temp
        for(k=2;k!=N+2;k++) {
              getcvector(temp1,k-1,cn1);
              setcvector(Cex,k,cn1.real,cn1.imaginary);
              }

        /*(K6)*/
        getcvector(Cex,N+1,cn1);
        setscalar(mu,cn1.real,cn1.imaginary);

        for(k=1;k!=N+1;k++) {
              getcvector(Cex,k,cn1);
              setcvector(m,k,cn1.real,cn1.imaginary);
              }

        /*(K7)*/

        getrvector(Xnm1,N,cn1);
        setscalar(temp,cn1.real,cn1.imaginary);

        upshift(Xnm1);
        setrvector(Xnm1,1,inp,0.0);

$       r=temp - Xnm1 * b

        /*(K8)*/
        setscalar(temp,1.0,0.0);


$       b=(b + m * r) * ((temp-mu*r)#)

        /*(K9)*/

$       c=m + b * mu

$       y=Xnm1 * w
$       err=dn - y
$       w=w + c * err
        nte=calcnte(w.element);
            *(average+n)=*(average+n)+nte/ensemble;
        }
}
for(n=0;n!=s+1;n++) {

        fprintf(fp,"%20.16e\n",*(average+n));

        }
fclose(fp);
makedB(s,average);

}

double calcnte(ptr)
double *ptr;
{
int k;
double nte;
struct complex Weight;


nte=0.0;

for (k=0;k!=N;k++) {
```

```
        Weight.real=((*ptr));
        ptr++;
        Weight.imaginary=((*ptr));
        ptr++;
        nte=nte+(Weight.real-Weights[k])*(Weight.real-Weights[k]);
        }
return(nte);
}


makedata()
{
int j;

        XK[1]=XK[0];

        XK[0]=gauss();

        inp=XK[0]*FEED_FORWARD[0]+XK[1]*FEED_FORWARD[1];

        for (j=4;j!=0;j--) {

              X[j]=X[j-1];
              }
        X[0]=inp;

        desired=0.0;

        for (j=0;j!=N;j++) desired=desired+X[j]*Weights[j];

        desired=desired+(gauss())*NOISE;
}


double rnum()
{
return ((random()/MAXRND));
}

double gauss()
{
double a,b;
double result;
a=rnum();
b=rnum();
result=sqrt(-2*log(a))*cos(2*3.141592654*b);
return(result);
}


makedB(s,data)
int s;
double *data;
{
FILE *fopen(),*fp2;
float se;
float init;
double p;
int k;

fp2=fopen("ERRdB.DAT","w");


for (k=0;k!=s+1;k++) {

        se=*(data+k);
        if (k==0) init=se;
```

```
        p=10*log10(se/init);

        fprintf(fp2,"%f\n",p);
        }

fclose(fp2);
}

void change_weights(t,file_ptr)
FILE *file_ptr;
int t;
{
}
```

FAEST simulation

```
/*Floating point simulation of FAEST algorithm*/

#define FLOATING
#define MANTISSA_LENGTH 56
#include </u4/call/lib/COMPLEX/sptools.h>
#include <stdio.h>
#include <math.h>

#define MAXRND 2147483647.0


int    N;
double calcnte();
double rnum();
double gauss();
void change_weights();
double inp,desired;
double X[5];
double Weights[5]={0.9,0.3,-0.3,0.7,0.1};

double FEED_FORWARD[2]={1.0,0.600};

double XK[2];
float NOISE;



main(argc,argv)
int argc;
char *argv[];
{
/* All of the spvar definitions should go in here*/
spvar XN,w,wmp1,temp1mp1,temp2mp1,a,b,zog,alphaf,alphafold,alphab,alpha;
spvar c,xn,z,ef,eb,e,epsilon,epsilonf,epsilonb,delta,d,aold,forget;
struct complex cn;
FILE *fopen(),*fp;
double sigma=-1.0,nte,gain_factor=0.0,*average,temp;

double lambda=0.0;

float SNR=-1;
int k,n,s,p_10,p,ensemble=-1,ens,res_flag;

char clear_screen=12;
char up_line=11;

/*Initialisation*/
if (argc!=2) res_flag = 1;
else {
      if (strcmp(argv[1],"-on")==0) res_flag=1;

      if (strcmp(argv[1],"-ON")==0) res_flag=1;

      if (strcmp(argv[1],"-off")==0) res_flag=0;

      if (strcmp(argv[1],"-OFF")==0) res_flag=0;
```

```
        }
printf("%c",clear_screen);
printf("Simulation of FAEST Algorithm\n\n");
printf("by Chris Callender, 1989\n\n\n\n");
if (res_flag) printf("\n\nRescue=ON, Floating Point Mantissa Length = %d\n\n",M
ANTISSA_LENGTH);
if (!res_flag) printf("\n\nRescue=OFF Floating Point Mantissa Length = %d\n\n",
MANTISSA_LENGTH);
printf("Filter Length:");
scanf("%d",&N);
/*All dimensions of matrices should be set here */


rvector(XN,N);
cvector(w,N);
cvector(wmp1,N+1);
cvector(temp1mp1,(N+1));
cvector(temp2mp1,(N+1));
cvector(a,N);
cvector(aold,N);
cvector(b,N);
scalar(zog);
scalar(alphaf);
scalar(alphafold);
scalar(alphab);
scalar(alpha);
cvector(c,N);
scalar(xn);
scalar(z);
scalar(ef);
scalar(eb);
scalar(e);
scalar(epsilon);
scalar(epsilonf);
scalar(epsilonb);
scalar(delta);
scalar(forget);
cvector(d,N);

fp=fopen("NORMTAPERROR.DAT","w");


srandom(1);


while (lambda<0.8 || lambda>1.0)

{
printf("Please enter a value for lambda between 0.8 and 1.0: ");

scanf("%lf",&lambda);
}
setscalar(forget,lambda,0.0);

while (sigma < 0.0)

{
printf("Please enter a small positive value for sigma: ");
scanf("%lf",&sigma);
}


while (SNR<0 || SNR>120) {

printf("\n\nPlease enter SIGNAL/NOISE ratio in dB (0 - 120db): ");

scanf("%f",&SNR);
```

```
}

while (ensemble<1) {
printf("\n\nHow many runs to make ensemble average: ");
scanf("%d",&ensemble);
}

for(n=0;n!=N;n++) gain_factor=gain_factor+Weights[n]*Weights[n];

gain_factor=gain_factor*(FEED_FORWARD[0]*FEED_FORWARD[0]+FEED_FORWARD[1]*FEED_F
ORWARD[1]);
gain_factor=sqrt(gain_factor);

NOISE=gain_factor/exp10(SNR/20.0);


printf("\n\nHow many data samples per run: ");
scanf("%d",&s);
fprintf(fp,"%d\n\n",s);
average=(double *)malloc(sizeof(double)*(s+1));
if (average==0) {

        fprintf(stderr,"FAEST Runtime error...out of memory");
        exit(1);
        }
printf("%cFAEST Simulation Running\n\n\n",clear_screen);
p_10=s/10;


for (ens=1;ens!=ensemble+1;ens++) {
p=p_10;

for(n=0;n!=N;n++) X[n]=0.0;

for(n=0;n!=2;n++) XK[n]=0.0;


zero(zog);
zero(XN);
zero(a);
zero(b);
zero(c);
zero(w);
zero(wmp1);

nte=calcnte(c.element);
*average=*average+nte/ensemble;
temp=sigma*pow((double)lambda,(double)N);
setscalar(alphaf,temp,0.0);

setscalar(alphab,sigma,0.0);

setscalar(alpha,1.0,0.0);


for(n=1;n!=s+1;n++) {
if (n==p) {
        printf("%cRun #%d:Status %d%%\n",up_line,ens,(p*10)/p_10);

        p=p+p_10;

        }
        /*The algorithm goes in here!*/

        makedata();
        setscalar(xn,inp,0.0);
```

```
        setscalar(z,desired,0.0);


$       ef=xn + XN * a

$       epsilonf=ef * (alpha#)

        copy(a,aold);
$       a=a+w * epsilonf

$       alphafold=forget * alphaf
$       alphaf=alphafold+ef*epsilonf

        for(k=1;k!=N+1;k++) {
              getcvector(w,k,cn);
              setcvector(temp1mp1,k+1,cn.real,cn.imaginary);
              getcvector(aold,k,cn);
              setcvector(temp2mp1,k+1,cn.real,cn.imaginary);
              }
        setcvector(temp1mp1,1,0.0,0.0);

        setcvector(temp2mp1,1,1.0,0.0);

$       wmp1=temp1mp1 - (ef * (alphafold#)) * temp2mp1

        /*Partitioning*/

        for(k=1;k!=N+1;k++) {
              getcvector(wmp1,k,cn);
              setcvector(d,k,cn.real,cn.imaginary);
              }
        getcvector(wmp1,(N+1),cn);
        setscalar(delta,cn.real,cn.imaginary);

$       eb=zog-delta * alphab * forget

$       w=d - delta * b

$       alpha = alpha + ( ef * alphafold #) * ef + delta * eb

$       epsilonb = eb * (alpha #)

$       alphab = forget * alphab + eb * epsilonb

$       b=b+w * epsilonb

        upshift(XN);
        setrvector(XN,1,inp,0.0);


        /* Time update the LS FIR Filter */

$       e=z + XN * c

$       epsilon = e * (alpha#)

$       c=c + w * epsilon

        nte=calcnte(c.element);
        *(average+n)=*(average+n)+nte/ensemble;
        }
}
for(n=0;n!=s+1;n++) {

        fprintf(fp,"%20.16e\n",*(average+n));
```

```
        }
fclose(fp);
makedB(s,average);

}

double calcnte(ptr)
double *ptr;
{
int k;
double nte=0.0;

struct complex Weight;


for (k=0;k!=N;k++) {

        Weight.real=((*ptr));
        ptr++;
        Weight.imaginary=((*ptr));
        ptr++;
        nte=nte+(Weight.real+Weights[k])*(Weight.real+Weights[k]);
        }
return(nte);
}


makedata()
{
int j;

        XK[1]=XK[0];

        XK[0]=gauss();

        inp=XK[0]*FEED_FORWARD[0]+XK[1]*FEED_FORWARD[1];

        for (j=4;j!=0;j--) {

              X[j]=X[j-1];
              }
        X[0]=inp;

        desired=0.0;

        for (j=0;j!=N;j++) desired=desired+X[j]*Weights[j];

        desired=desired+(gauss())*NOISE;
}


double rnum()
{
return ((random()/MAXRND));
}

double gauss()
{
double a,b;
double result;
a=rnum();
b=rnum();
result=sqrt(-2*log(a))*cos(2*3.141592654*b);
return(result);
}
```

```
makedB(s,data)
int s;
double *data;
{
FILE *fopen(),*fp2;
float se;
float init;
double p;
int k;

fp2=fopen("ERRdB.DAT","w");


for (k=0;k!=s+1;k++) {

        se=*(data+k);
        if (k==0) init=se;

        p=10*log10(se/init);

        fprintf(fp2,"%f\n",p);
        }

fclose(fp2);
}

void change_weights(t,file_ptr)
FILE *file_ptr;
int t;
{
}
```

Fixed point FTF simulation

```
/*Fixed point simulation of FTF algorithm*/

#define MAXRND 2147483647.0


#include <math.h>
#include <stdio.h>
typedef short int VAR;
double X[5];
double XK[2];
double FEED_FORWARD[2]={0.15,0.12975};

double Weights[5]={0.9,0.3,-0.3,-0.7,0.1};

double calcnte();
double gauss();
VAR div();
int mul();
VAR add();
VAR scalar_product();
int N;
double NOISE=0.001;

VAR inp,des,sat_flag;

main(argc,argv)
int argc;
char *argv[];
{
double nte,d_lambda=0.0,d_MU=-1.0,SNR=-1.0,gain_factor,*average;

int s,seed,ens,ensemble=0;

int long_accumulator;
VAR *A,*Y,*YNp1,*C,*Cex,*CNp1,*B,*W;
VAR index,t,lambda,mu;
VAR rescue,y0,alpham1,eNp,eN,gammaN,alpha,alphaold,epsilon;

VAR gammaNp1,epsilonp,rN,rNp,beta;
FILE *fopen(),*fp;


if (argc!=2) sat_flag = 0;

else {
      if (strcmp(argv[1],"-sat")==0) sat_flag=1;

      if (strcmp(argv[1],"-SAT")==0) sat_flag=1;

      }
fp=fopen("NORMTAPERROR.DAT","w");

printf("Simulation of FTF Algorithm\n\n");
printf("by Chris Callender, 1989\n\n\n\n");
printf("\n\n16 Bit Fixed Point\n\n");
printf("Filter Length:");
scanf("%d",&N);
```

```
while (d_lambda<0.8 || d_lambda>1.0)

{
printf("Please enter a value for lambda between 0.8 and 1.0: ");

scanf("%lf",&d_lambda);
}

while(d_MU<0) {

printf("\n\nPlease enter a value for soft constraint parameter MU:\n ");
printf("or MU=0.0 to disable rescues\n");

scanf("%lf",&d_MU);
}


while (SNR<0 || SNR>220) {

printf("\n\nPlease enter SIGNAL/NOISE ratio in dB (0 - 220db): ");

scanf("%lf",&SNR);
}

while (ensemble<1) {
printf("\n\nHow many runs to make ensemble average: ");
scanf("%d",&ensemble);
}

for(t=0;t!=N;t++) gain_factor=gain_factor+Weights[t]*Weights[t];

gain_factor=gain_factor*(FEED_FORWARD[0]*FEED_FORWARD[0]+FEED_FORWARD[1]*FEED_F
ORWARD[1]);
gain_factor=sqrt(gain_factor);

NOISE=gain_factor/exp10(SNR/20.0);


printf("\n\nHow many data samples per run: ");
scanf("%d",&s);
average=(double *)malloc(sizeof(double)*(s+1));
if (average==0) {

      fprintf(stderr,"FTF Runtime error...out of memory");
      exit(1);
      }
/*First, allocate memory for vectors*/

A=(VAR *)malloc(sizeof(VAR)*(N+1)); /*Scale factor will be 1024*/

Y=(VAR *)malloc(sizeof(VAR)*N); /*Scale Factor will be 32768*/
YNp1=(VAR *)malloc(sizeof(VAR)*(N+1)); /*Scale factor will be 32768*/
C=(VAR *)malloc(sizeof(VAR)*N); /*Scale Factor will be 8*/
CNp1=(VAR *)malloc(sizeof(VAR)*(N+1)); /*Scale Factor will be 8*/
B=(VAR *)malloc(sizeof(VAR)*(N+1)); /*Scale Factor will be 32768*/
W=(VAR *)malloc(sizeof(VAR)*N); /*Scale Factor will be 32768*/

seed=time(0);

srandom(seed);
printf("\n\n");
lambda=32768*d_lambda;
mu=32768*d_MU;

for (ens=1;ens!=ensemble+1;ens++) {
      for(t=0;t!=N;t++) X[t]=0.0;
```

```
        for(t=0;t!=2;t++) XK[t]=0.0;


/* First the Fast Exact Initialisation Routine*/
A[0]=1024;

B[0]=32767;

for (index=1;index!=N+1;index++) {
        A[index]=0;

        B[index]=0;

        }
for (index=0;index!=N;index++) {

        C[index]=0;

        W[index]=0;

        Y[index]=0;

        }
for (index=0;index!=N+1;index++) YNp1[index]=0;

nte=calcnte(W);
fprintf(fp,"%20.16e\n",nte);

inp=0;

while (abs(inp)<9000) makedata(0);

y0=inp;

alpha=mul(y0,y0,15);

alpham1=alpha;
W[0]=-div(des,y0,15);

gammaN=32767;



for(t=1;t!=N+1;t++) {
        nte=calcnte(W);
        fprintf(fp,"%20.16e\n",nte);

        makedata(t);
        for(index=N+1;index!=0;index--) {

                YNp1[index]=YNp1[index-1];
                if (index!=N+1) Y[index]=Y[index-1];
                }
        YNp1[0]=inp;

        Y[0]=inp;


        eNp=scalar_product(A,YNp1,N+1,11);

        A[t]=-div(eNp,y0,11);


        eN=mul(eNp,gammaN,14);

        alpha=mul(lambda,alpha,15);
```

```
        alpham1=add(alpha,mul(eNp,eN,14));

        gammaN=mul(gammaN,div(alpha,alpham1,15),15);

        for(index=t;index!=0;index--) {

              C[index]=C[index-1];
              }
        C[0]=0;

        for(index=0;index!=t;index++) {

              C[index]=add(C[index],-div(mul(eNp,A[index],10),alpha,4));

              }

        if (t==N) {
              for(index=0;index!=N;index++) {

                    B[index]=mul(mul(y0,gammaN,15 ),C[index],3) ;

                    }
              B[N]=32767;

              beta=mul(mul(y0,y0,15),gammaN,9);

              }

        epsilonp=add(scalar_product(Y,W,N,15),des);

        epsilon=mul(epsilonp,gammaN,15);

        if (t<N) W[t]=-div(epsilonp,y0,15);


        if (t==N) {
              for(index=0;index!=N;index++) {

                    W[index]=add(W[index],mul(epsilon,C[index],4));
                    }
              }
        }
/* Now the FTF Algorithm proper */

for (t=N+1;t!=s+1;t++) {
        makedata(t);
        for(index=N+1;index!=0;index--) {

              YNp1[index]=YNp1[index-1];
              if (index!=N+1) Y[index]=Y[index-1];
              }
        YNp1[0]=inp;

        Y[0]=inp;


        RE_START:
        /*#1*/

        eNp = scalar_product(A,YNp1,N+1,11);

        /*#2*/

        eN=mul(eNp,gammaN,14);

        /*#3*/
        alphaold=alpha;
```

```
        alpha=add(mul(lambda,alpha,15),mul(eNp,eN,14));

        /*#4*/
        gammaNp1 = mul(mul(lambda,div(alphaold,alpha,10),15),gammaN,10);


        /*#5*/
        CNp1[0]=-mul(div(eNp,mul(alphaold,lambda,15),5),A[0],11) ;

        for (index=1;index!=N+1;index++) {
              CNp1[index]=add(C[index-1],-mul(div(eNp,mul(alphaold,lambda,15),5),A[index],1
1));
              }
        /*#6*/
        for (index=1;index!=N+1;index++) {
              A[index]=add(A[index],mul(eN,C[index-1],8));
              }
        /*#7*/
        rNp=mul(mul(-lambda,beta,15),CNp1[N],11);

        rescue=add(16384,mul(mul(rNp,gammaNp1,15),CNp1[N],4));
        if (rescue<0 && d_MU!=0.0) {

              for (index=0;index!=N+1;index++) {

                    A[index]=0;

                    B[index]=0;

                    if (index!=N) C[index]=0;

                    }
                    A[0]=1024;

                    B[N]=32767;
                    alpha=mu;
                    for (index=1;index!=N;index++) alpha=mul(alpha,lambda,15);
                    beta=mu << 3;
                    gammaN=32767;
                    goto RE_START;
        }


        /*#8*/
        gammaN=div(gammaNp1,rescue,14);

        /*#9*/
        rN=mul(rNp,gammaN,11) ;

        /*#10*/

        beta=add(mul(beta,lambda,15),mul(rNp,rN,11));

        /*#11*/
        for(index=0;index!=N+1;index++) {

              C[index]=add(CNp1[index],-mul(CNp1[N],B[index],15));
        }

        /*#12*/
        for(index=0;index!=N+1;index++) {

              B[index]=add(B[index],mul(rN,C[index],7));
              }

        /*#13*/
        epsilonp=add(scalar_product(Y,W,N,15),des);
```

```
        /*#14*/
        epsilon = mul(epsilonp,gammaN,15);

        /*#15*/

        for(index=0;index!=N+1;index++) {

        W[index]=add(W[index],mul(epsilon,C[index],3));
        }
        nte=calcnte(W);
        fprintf(fp,"%20.16e\n",nte);

        }
}
fclose(fp);
}


double calcnte(ptr)
VAR *ptr;
{
int k;
double nte,W;


nte=0.0;

for (k=0;k!=N;k++) {

        W=((*ptr)/32768.0);

        ptr++;
        nte=nte+(W+Weights[k])*(W+Weights[k]);
        }
return(nte);
}


makedata(t)
int t;
{
double m_inp,m_des;
int j;

        XK[1]=XK[0];

        if (t==0) XK[0]=-4.0;

        else XK[0]=gauss();

        m_inp=XK[0]*FEED_FORWARD[0]+XK[1]*FEED_FORWARD[1];

        for (j=4;j!=0;j--) {

              X[j]=X[j-1];
              }
        X[0]=m_inp;

        m_des=0.0;

        for (j=0;j!=N;j++) m_des=m_des+X[j]*Weights[j];

        m_des=m_des+(gauss())*NOISE;
        /*Most ADCs saturate so...*/
        if (m_inp>1.0) m_inp=0.99969482;

        if (m_des>1.0) m_des=0.99969482;
```

```
        des=m_des*32768;
        inp=m_inp*32768;
}


double rnum()
{
return ((random()/MAXRND));
}

double gauss()
{
double a,b;
double result;
a=rnum();
b=rnum();
result=sqrt(-2*log(a))*cos(2*3.141592654*b);
return(result);
}

VAR div(a,b,res_shift)
int a,b,res_shift;
{
int c;
c=a;
c=c << res_shift;
if (b==0) {

        fprintf(stderr,"Algorithm fails...division by zero");
        exit(1);
        }
c=c/b;
/* A good idea to saturate division, in case 1/1 is calculated, q15*/
if (c>32767) c=32767;
if (c<-32768) c=-32768;
a=c;
return(a);
}

int mul(a,b,res_shift)
VAR a,b,res_shift;
{
int c;
c=a * b;
if (res_shift >=0) c=c >> res_shift;

if (res_shift <0) c=c << -res_shift;

#ifdef DEBUG
if (c<-32768 || c>32767) {
        fprintf(stderr,"Overflow Warning\n");
        }
#endif
if (sat_flag!=0) {

        if (c>32767) c=32767;
        if (c<-32768) c=32768;
        }
return(c);
}


VAR add(a,b)
VAR a,b;
{
int c;
c=a+b;
```

```
#ifdef DEBUG
if (c<-32768 || c>32767) {
                fprintf(stderr,"Overflow Warning\n");
                }
#endif
if (sat_flag!=0) {

        if (c>32767) c=32767;
        if (c<-32768) c=32768;
        }
return(c);
}

VAR scalar_product(a,b,len,res_shift)
VAR *a,*b;
int len;
VAR res_shift;
{
VAR index;
int long_accumulator;
long_accumulator=0;


for(index=0;index!=len;index++) {

        long_accumulator = long_accumulator + (*a) * (*b);
        a++;
        b++;
        }
long_accumulator = long_accumulator >> res_shift;
#ifdef DEBUG
if (long_accumulator<-32768 || long_accumulator>32767) {
        fprintf(stderr,"Overflow Warning\n");
        }
#endif
if (sat_flag!=0) {

        if (long_accumulator>32767) long_accumulator=32767;
        if (long_accumulator<-32768) long_accumulator=32768;
        }
return(long_accumulator);
}
```

Floating point interval arithmetic functions

```
/* #include file intools.h

Version 2.2

Written by Chris Callender, January 1989
Last Update:27/6/89*/


#include <stdio.h>
#include <math.h>
struct interval {
      double     lower_endpoint;
      double     upper_endpoint;
};
void        imult();
void        iadd();
void        idiv();

/*Begin by defining the structures assocaited with scalars, column
vectors, row vectors, and matrices. They are all basically matrices, but
a scalar has one row and one column, a column vector has one column and a
row vector has one row*/

#define spvar          static struct SPVAR

struct      SPVAR {
      int rsize;
      int csize;
      struct interval *element;
      };


#define scalar(name)              name.rsize=1;\
                name.csize=1;\
                name.element=(struct interval *)malloc(sizeof(struct interval))

#define cvector(name,row)               name.rsize=row;\
                        name.csize=1;\
                        name.element=(struct interval *)malloc(sizeof(struct interval)*row)

#define rvector(name,column)                  name.rsize=1;\
                        name.csize=column;\
                        name.element=(struct interval *)malloc(sizeof(struct interval)*column)

#define matrix(name,row,column)               name.rsize=row;\
                        name.csize=column;\
                        name.element=(struct interval *)malloc(sizeof(struct interval)*column*row);
\
                        if (name.element==0) error("Out of Memory")



#define resize(name,r,c)                  name.rsize=r; \
                        name.csize=c; \
                        free(name.element);\
                        name.element=m_realloc(name.element,sizeof(struct interval)*r*c);\
                        if (name.element==0) error("Out of Memory")
```

```
struct interval *m_realloc(ptr,sz)
struct interval *ptr;
int sz;
{
if (ptr==0) ptr=(struct interval *) malloc(sz);

else ptr=(struct interval *)realloc(ptr,sz);
return(ptr);
}

/*A display function prints the matrix on standard output. It will
  work with scalars, row and column vectors, and matrices. Use
       display(MATRIXNAME);
  in the program to display the current value of MATRIXNAME*/

#define display(name) displaymatrix(name.element,name.rsize,name.csize)

displaymatrix(adr, r, c)
int     r, c;
double      *adr;
{
       int   j, k;
       for (j = 0; j != r; j++) {

            for (k = 0; k != c; k++) {

                 printf("[%lf,%lf] ",*adr,*(adr+1));
                 adr++;
                 adr++;
            }
            printf("\n");
       }
       printf("\n");
return(0);

}


/*Matrix addition routine. Works automatically with scalars, vectors and
   matrices. Use the command:-
       add(MATRIX1,MATRIX2,RESULT);
   to make the matrix RESULT equal to MATRIX1+MATRIX2
   matrices. */

#define add(name1,name2,name3) addmatrix(name1.element,name2.element,\
name3.element,name1.rsize,name1.csize,name2.rsize,name2.csize,name3.rsize,\
name3.csize)

void addmatrix(adr1, adr2, adr3, r1, c1, r2, c2, r3, c3)

int     r1, c1, r2, c2, r3, c3;
double      *adr1, *adr2, *adr3;
{
       int   j, k;
       if (r1 != r2)
            error("Unable to add matrices of different sizes");
       if (c1 != c2)
            error("Unable to add matrices of different sizes");
       if (r1 != r3)
            error("Result matrix of incorrect size in add");
       if (c1 != c3)
            error("Result matrix of incorrect size in add");
       for (j = 0; j != r1; j++) {

            for (k = 0; k != c1; k++) {

                 cadd(*adr1, *(adr1 + 1), *adr2, *(adr2 +
```

```
                    1), adr3);
                adr1 = adr1 + 2;
                adr2 = adr2 + 2;
                adr3 = adr3 + 2;
            }
        }
}


/*Identity sets a matrix, vector or scalar to the multiplication identity.
  It is useful in the initialisation of matrices. Correct syntax is:
     identity(MATRIXNAME);*/

#define identity(name) iden(name.element,name.rsize,name.csize)

void iden(adr, r, c)

double      *adr;
int     r, c;
{
        int   j, k;
        if (r != c)
            error("Non square matrix cannot be set to identity");
        for (j = 0; j != r; j++) {

            for (k = 0; k != c; k++) {

                if (j == k) {
                    *adr = 1.0;

                    *(adr + 1) = 1.0;

                }
                if (j != k) {
                    *adr = 0.0;

                    *(adr + 1) = 0.0;

                }
                adr = adr + 2;
            }
        }
}


/*Similar to identity, this sets a matrix, vector or scalar to the
  addition identity element (zero)*/

#define zero(name) zer(name.element,name.rsize,name.csize)

void zer(adr, r, c)

double      *adr;
int     r, c;
{
        int   j, k;
        for (j = 0; j != r; j++) {

            for (k = 0; k != c; k++) {

                *adr = 0;

                *(adr + 1) = 0;

                adr = adr + 2;
            }
        }
```

```
}

/*Setmatrix is used to set one element of a matrix to a specified
  value--it takes the form:
     setmatrix(MATRIXNAME,columnnumber,rownumber,value) to set
  the element[columnnumber][rownumber] of MATRIXNAME to the value
  value.*/

#define setmatrix(name,x,y,rvalue,imvalue) setel(name.element,name.rsize,\
name.csize,x,y,rvalue,imvalue)

/*These help with the setting of column and row vectors and scalars*/
#define setcvector(name,x,rvalue,imvalue) setmatrix(name,x,1,rvalue,imvalue)
#define setrvector(name,y,rvalue,imvalue) setmatrix(name,1,y,rvalue,imvalue)
#define setscalar(name,rvalue,imvalue) setmatrix(name,1,1,rvalue,imvalue)

void setel(adr, r, c, x, y, vr, vi)
double      vr, vi;
double      *adr;
int     r, c, x, y;
{
      if (y > c)
           error("Setelement out of bounds");
      if (x > r)
           error("Setelement out of bounds");
      if (y < 1)
           error("Setelement out of bounds");
      if (x < 1)
           error("Setelement out of bounds");
      x--;
      y--;
      *(adr + (c * x * 2) + y * 2) = vr;
      *(adr + (c * x * 2) + y * 2 + 1) = vi;
}


/*Matrix multiplication routine. Will work with vectors, scalars and
  matrices. Use the usual:
        multiply(MATRIX1,MATRIX2,RESULT);
  to make the matrix RESULT equal to MATRIX1 multiplied by MATRIX2*/

#define multiply(name1,name2,name3) mult(name1.element,name2.element,\
name3.element,name1.rsize,name1.csize,name2.rsize,name2.csize,\
name3.rsize,name3.csize)

void mult(adr1, adr2, adr3, r1, c1, r2, c2, r3, c3)
double      *adr1, *adr2, *adr3;
int     r1, c1, r2, c2, r3, c3;
{
      int   j, k, l;
      double     total[2], ar, ai, br, bi, t[2];
      double     *temp;
      if (r1 == 1 && c1 == 1) {
                    msc(adr2,adr3,r2,c2,r3,c3,*adr1,*(adr1+1));
                    goto SKIP;
                    }
      if (r2 == 1 && c2 == 1) {
                    msc(adr1,adr3,r1,c1,r3,c3,*adr2,*(adr2+1));
                    goto SKIP;
                    }
      if (r2 != c1)
           error("Unable to multiply matrices - dimensions incorrect");
      if (c2 != c3)
           error("Result matrix of incorrect size in multiply");
      if (r1 != r3)
           error("Result matrix of incorrect size in multiply");
```

```
        temp = (double *)malloc(sizeof(double)*r3 * c3 * 2);
        if (temp == 0)

             error("Out of Memory Error");
        for (j = 0; j != r3; j++) {

             for (k = 0; k != c3; k++) {

                  total[0] = 0;

                  total[1] = 0;

                  for (l = 0; l != c1; l++) {

                       ar = (*(adr1 + (c1 * j * 2) + l
                            *2));
                       ai = (*(adr1 + (c1 * j * 2) + l
                            *2 + 1));
                       br = (*(adr2 + (c2 * l * 2) + k
                            *2));
                       bi = (*(adr2 + (c2 * l * 2) + k
                            *2 + 1));
                       cmul(ar, ai, br, bi, t);
                       cadd(t[0], t[1], total[0], total[1],

                            total);
                  }
                  *(temp + (c3 * j * 2) + k * 2) = total[0];

                  *(temp + (c3 * j * 2) + k * 2 + 1) = total[1];
             }
        }
        for (j = 0; j != r3; j++) {

             for (k = 0; k != c3; k++) {

                  ar = (*(temp + (c3 * j * 2) + k * 2));
                  ai = (*(temp + (c3 * j * 2) + k * 2 + 1));
                  *(adr3 + (c3 * j * 2) + k * 2) = ar;
                  *(adr3 + (c3 * j * 2) + k * 2 + 1) = ai;
             }
        }
        free(temp);
SKIP: ;
}

number_of_rows(r1,c1,r2,c2)
int r1,c1,r2,c2;
{
if (r1==1 && c1==1) return(r2);
return(r1);
}

number_of_columns(r1,c1,r2,c2)
int r1,c1,r2,c2;
{
if (r2==1 && c2==1) return(c1);
return(c2);
}

/*The opposite of setmatrix, getmatrix returns the value of an element of
   a matrix*/
#define getmatrix(name,x,y,var) getel(name.element,name.rsize,\
name.csize,x,y,&var)

/*Opposites of setcvector,setscalar and setrvector*/
#define getcvector(name,x,var) getmatrix(name,x,1,var)
```

```
#define getrvector(name,y,var) getmatrix(name,1,y,var)
#define getscalar(name,var) getmatrix(name,1,1,var)

void getel(adr, r, c, x, y, v)
double      *v;
double      *adr;
int     r, c, x, y;
{
        if (y > c)
            error("Getelement out of bounds");
        if (x > r)
            error("Getelement out of bounds");
        if (y < 1)
            error("Getelement out of bounds");
        if (x < 1)
            error("Getelement out of bounds");
        x--;
        y--;
        (*v) = (*(adr + (c * x * 2) + y * 2));
        (*(v + 1)) = (*(adr + (c * x * 2) + y * 2 + 1));
}


/*Use subtract exactly as add, but result is MATRIX1-MATRIX2*/
#define subtract(name1,name2,name3) submatrix(name1.element,name2.element,\
name3.element,name1.rsize,name1.csize,name2.rsize,name2.csize,name3.rsize,\
name3.csize)

void submatrix(adr1, adr2, adr3, r1, c1, r2, c2, r3, c3)

int     r1, c1, r2, c2, r3, c3;
double      *adr1, *adr2, *adr3;
{
        int   j, k;
        double     a, b, d;
        if (r1 != r2)
            error("Unable to subtract matrices of different sizes");
        if (c1 != c2)
            error("Unable to subtract matrices of different sizes");
        if (r1 != r3)
            error("Result matrix of incorrect size in subtract");
        if (c1 != c3)
            error("Result matrix of incorrect size in subtract");
        for (j = 0; j != r1; j++) {

            for (k = 0; k != c1; k++) {

                cadd(*adr1, *(adr1 + 1), -(*(adr2+ 1)),-(*(adr2)), adr3);
                adr1 = adr1 + 2;
                adr2 = adr2 + 2;
                adr3 = adr3 + 2;
            }
        }
}


/*Transpose calculates the transpose of a matrix. Use
   transpose(MATRIX,RESULT); to set RESULT equal to the transpose of MATRIX*/

#define transpose(name1,name2) transp(name1.element,name2.element,name1.rsize,\

name1.csize,name2.rsize,name2.csize)

transp(adr1, adr2, r1, c1, r2, c2)
double      *adr1, *adr2;
int     r1, c1, r2, c2;
{
```

```
        double    ar, ai;
        double    *temp;
        int  j, k;
        if (r1 != c2)
            error("Result matrix of incorrect size in transpose");
        if (c1 != r2)
            error("Result matrix of incorrect size in transpose");
        temp = (double *)malloc(sizeof(double)*r2 * c2 * 2);
        if (temp == 0)

            error("Out of memory error");
        for (j = 0; j != r1; j++) {

            for (k = 0; k != c1; k++) {

                ar = (*(adr1 + (j * c1 * 2) + k * 2));
                ai = (*(adr1 + (j * c1 * 2) + k * 2 +
                    1));
                *(temp + (k * r1 * 2) + j * 2) = ar;
                *(temp + (k * r1 * 2) + j * 2 + 1) = ai;
            }
        }
        for (j = 0; j != r2; j++) {

            for (k = 0; k != c2; k++) {

                ar = (*(temp + (c2 * j * 2) + k * 2));
                ai = (*(temp + (c2 * j * 2) + k * 2 + 1));
                *(adr2 + (c2 * j * 2) + k * 2) = ar;
                *(adr2 + (c2 * j * 2) + k * 2 + 1) = ai;
            }
        }
        free(temp);
}


/*Multiply a matrix by a scalar. Use the command:
    multscal(MATRIX,kreal,kimaginary,RESULT);
    to make RESULT equal to k*MATRIX*/
#define multscal(name1,vreal,vimag,name2) msc(name1.element,name2.element,\
name1.rsize,name1.csize,name2.rsize,name2.csize,vreal,vimag)

msc(adr1, adr2, r1, c1, r2, c2, vr, vi)
double    *adr1, *adr2;
double    vr, vi;
int   r1, c1, r2, c2;
{
        int  j, k;
        double    ar, ai;
        if (r1 != r2)
            error("Result matrix of incorrect size in multscal");
        if (c1 != c2)
            error("Result matrix of incorrect size in multscal");
        for (j = 0; j != r1; j++) {

            for (k = 0; k != c1; k++) {

                ar = (*adr1);
                ai = (*(adr1 + 1));
                cmul(ar, ai, vr, vi, adr2);
                adr1 = adr1 + 2;
                adr2 = adr2 + 2;
            }
        }
}
```

```
/*Copy one matrix to another. Syntax is:
copy(MATRIX,RESULT); which makes RESULT equal to MATRIX. Equivalent to
multscal(MATRIX,1,RESULT); but clearer, faster and more readable.*/

#define copy(name1,name2) cpy(name1.element,name2.element,\
name1.rsize,name1.csize,name2.rsize,name2.csize)

cpy(adr1, adr2, r1, c1, r2, c2)
double      *adr1, *adr2;
int     r1, c1, r2, c2;
{
        int   j, k;
        double      ar, ai;
        if (r1 != r2)
            error("Result matrix of incorrect size in copy");
        if (c1 != c2)
            error("Result matrix of incorrect size in copy");
        for (j = 0; j != r1; j++) {

            for (k = 0; k != c1; k++) {

                ar = (*adr1);
                ai = (*(adr1 + 1));
                *adr2 = ar;
                *(adr2 + 1) = ai;
                adr1 = adr1 + 2;
                adr2 = adr2 + 2;
            }
        }
}


/*Upshift only applies to vectors, and shifts each element up one place in
   the vector. Will work automatically with either column or row vectors.
   Use:
   upshift(VECTORNAME);*/

#define upshift(name) ups(name.element,name.rsize,name.csize)

ups(adr, r, c)
double      *adr;
int     r, c;
{
        int   j, k;
        double      ar, ai;
        if (r != 1 && c != 1)
            error("Unable to upshift a matrix");
        if (r == 1 && c == 1)
            error("Unable to upshift a scalar");
        if (r == 1)
            j = c;
        if (c == 1)
            j = r;
        for (k = j - 1; k != 0; k--) {

            ar = (*(adr + k * 2 - 2));
            ai = (*(adr + k * 2 - 1));
            *(adr + k * 2) = ar;
            *(adr + k * 2 + 1) = ai;
        }
        *adr = 0;

        *(adr + 1) = 0;

}
```

```
#define downshift(name) dos(name.element,name.rsize,name.csize)

dos(adr, r, c)
double      *adr;
int     r, c;
{
        int   j, k;
        double      ar, ai;
        if (r != 1 && c != 1)
            error("Unable to downshift a matrix");
        if (r == 1 && c == 1)
            error("Unable to downshift a scalar");
        if (r == 1)
            j = c;
        if (c == 1)
            j = r;
        for (k = 0; k != j - 1; k++) {

            ar = (*(adr + k * 2 + 2));
            ai = (*(adr + k * 2 + 3));
            *(adr + k * 2) = ar;
            *(adr + k * 2 + 1) = ai;
        }
        *(adr + j) = 0;

        *(adr + j + 1) = 0;

}


#define inverse(name1,name2) inv(name1.element,name2.element,name1.rsize,\
name1.csize,name2.rsize,name2.csize)

inv(adr1, adr2, r1, c1, r2, c2)
double      *adr1, *adr2;
int     r1, r2, c1, c2;
{
        int   n, m, j;
        double      factor[2], a[2], t[2];
        double      *temp;
        if (r1 != c1)
            error("Cannot invert a non-square matrix");
        if (r2 != r1)
            error("Result matrix of incorrect size in inverse");
        if (c2 != c1)
            error("Result matrix of incorrect size in inverse");
        temp = (double *)malloc(sizeof(double)*r1 * c1 * 2);
        if (temp == 0)

            error("Out of Memory Error");
        for (n = 0; n != r1; n++) {

            for (m = 0; m != c1; m++) {

                *(temp + (r1 * m * 2) + n * 2) = (*(adr1
                      + (r1 * m * 2) + n * 2));
                *(temp + (r1 * m * 2) + n * 2 + 1) = (*(adr1
                      + (r1 * m * 2) + n * 2 + 1));
            }
        }
        iden(adr2, r2, c2);
        for (n = 0; n != r1; n++) {

            iinv(*(temp + (n * c1 * 2) + n * 2), *(temp + (n
                  *c1 * 2) + n * 2 + 1), factor);
            for (j = 0; j != c1; j++) {
```

```
                cmul(factor[0], factor[1], *(temp + (n *
                    c1 * 2) + j * 2), *(temp + (n * c1 * 2)
                     + j * 2 + 1), temp + (n * c1 * 2) + j *
                    2);
                cmul(factor[0], factor[1], *(adr2 + (n *
                    c2 * 2) + j * 2), *(adr2 + (n * c2 * 2)
                     + j * 2 + 1), adr2 + (n * c2 * 2) + j *
                    2);
            }
            for (m = 0; m != r1; m++) {
                if (m != n) {
                    factor[0] = (*(temp + (m * c1 *
                        2) + n * 2));
                    factor[1] = (*(temp + (m * c1 *
                        2) + n * 2 + 1));
                    for (j = 0; j != c1; j++) {
                        a[0] = (*(temp + (n * c1
                            *2) + j * 2));
                        a[1] = (*(temp + (n * c1
                            *2) + j * 2 + 1));
                        cmul(a[0], a[1], factor[0],
                            factor[1], t);
                        cadd(*(temp + (m * c1 *
                            2) + j * 2), *(temp + (m
                            *c1 * 2) + j * 2 + 1), -t[0],
                            -t[1], temp + (m * c1 *
                            2) + j * 2);
                        a[0] = (*(adr2 + (n * c2
                            *2) + j * 2));
                        a[1] = (*(adr2 + (n * c2
                            *2) + j * 2 + 1));
                        cmul(a[0], a[1], factor[0],
                            factor[1], t);
                        cadd(*(adr2 + (m * c2 *
                            2) + j * 2), *(adr2 + (m
                            *c2 * 2) + j * 2 + 1), -t[0],
                            -t[1], adr2 + (m * c2 *
                            2) + j * 2);
                    }
                }
            }
        }
    free(temp);
}

void input(filename, ptr)
char        *filename[];
double      *ptr;
{
    static char     fname[10][20];

    static FILE *fopen(), *fpointer[10];

    float       c, d;
    int   a, f = 0, b = 10;
```

```
        if (strcmp(filename, "close") == 0) {

                for (a = 0; a != 10; a++) {

                        if (fname[a][0] != 0) {

                                fclose(fpointer[a]);
                                fname[a][0] = 0;

                        }
                }
                return;
        }

        for (a = 9; a != -1; a--) {
                if (strcmp(filename, fname[a]) == 0) {

                        f = 1;
                        b = a;
                } else {
                        if (fname[a][0] == 0)

                                b = a;
                }
        }
        if (b == 10)

                error("Too many files open (maximum of 10)");

        if (f == 0) {

                strcpy(fname[b], filename);
                fpointer[b] = fopen(fname[b], "r");
                if (fpointer[b] == 0)

                        error("Unable to open file for input");
        }
        fscanf(fpointer[b], "%f %f", &c, &d);
        *ptr = c;
        *(ptr + 1) = d;
}


void output(filename, ptr)
char        *filename[];
double      *ptr;
{
        static char     oname[10][20];

        static FILE *fopen(), *opointer[10];

        int   a, f = 0, b = 10;

        if (strcmp(filename, "close") == 0) {

                for (a = 0; a != 10; a++) {

                        if (oname[a][0] != 0) {

                                fclose(opointer[a]);
                                oname[a][0] = 0;

                        }
                }
                return;
        }
```

```
        for (a = 9; a != -1; a--) {
            if (strcmp(filename, oname[a]) == 0) {

                f = 1;
                b = a;
            } else {
                if (oname[a][0] == 0)

                    b = a;
            }
        }
        if (b == 10)

            error("Too many files open (Maximum of 10)");

        if (f == 0) {

            strcpy(oname[b], filename);
            opointer[b] = fopen(oname[b], "w");
            if (opointer[b] == 0)

                error("Unable to open file for output");
        }
        fprintf(opointer[b], "%f %f\n", (*ptr), (*(ptr + 1)));
        return;
}


error(message)
char        *message[];
{
        int *t;
        t=0;

        fprintf(stderr, "***DSPSIM Runtime Error***\n");
        fprintf(stderr, "%s\n\n", message);
        exit(0);

}



cadd(a, b, c, d, e)
double      a, b, c, d, *e;
{
struct interval temp1,temp2;
temp1.lower_endpoint=a;
temp1.upper_endpoint=b;
temp2.lower_endpoint=c;
temp2.upper_endpoint=d;
iadd(&temp1,&temp2,e);
}


cmul(a, b, c, d, e)
double      a, b, c, d, *e;
{
struct interval temp1,temp2;
temp1.lower_endpoint=a;
temp1.upper_endpoint=b;
temp2.lower_endpoint=c;
temp2.upper_endpoint=d;
imult(&temp1,&temp2,e);
}


iinv(a,b,c)
```

```
double a,b,*c;
{
struct interval temp1,temp2;
temp1.lower_endpoint=1.0;

temp1.upper_endpoint=1.0;

temp2.lower_endpoint=a;
temp2.upper_endpoint=b;
idiv(&temp1,&temp2,c);
}

void iadd(inter1,inter2,interres)
struct interval *inter1,*inter2,*interres;
{
double upper,lower;
lower=(inter1->lower_endpoint)+(inter2->lower_endpoint);
upper=(inter1->upper_endpoint)+(inter2->upper_endpoint);


if (lower<0) lower=nextafter(lower,-(infinity()));

if (upper>0) upper=nextafter(upper,infinity());


if (upper<lower) error("Upper less than lower");

interres->lower_endpoint=lower;
interres->upper_endpoint=upper;
}

void isub(inter1,inter2,interres)
struct interval *inter1,*inter2,*interres;
{
double upper,lower;
lower=(inter1->lower_endpoint)-(inter2->upper_endpoint);
upper=(inter1->upper_endpoint)-(inter2->lower_endpoint);


if (upper>0) upper=nextafter(upper,infinity());

if (lower<0) lower=nextafter(lower,-(infinity()));


if (upper<lower) error("Upper less than lower");
interres->lower_endpoint=lower;
interres->upper_endpoint=upper;
}

void imult(inter1,inter2,interres)
struct interval *inter1,*inter2,*interres;
{
double r[4];
double min,max;
int k;

r[0]=(inter1->lower_endpoint)*(inter2->lower_endpoint);

r[1]=(inter1->lower_endpoint)*(inter2->upper_endpoint);
r[2]=(inter1->upper_endpoint)*(inter2->lower_endpoint);
r[3]=(inter1->upper_endpoint)*(inter2->upper_endpoint);

max=r[0];

min=r[0];
```

```
for(k=1;k!=4;k++) {
        if (r[k]>max) max=r[k];
        if (r[k]<min) min=r[k];
        }


if (min<0) min=nextafter(min,-(infinity()));

if (max>0) max=nextafter(max,infinity());


if (max<min) error("Upper less than lower");
interres->lower_endpoint=min;
interres->upper_endpoint=max;
}

void idiv(inter1,inter2,interres)
struct interval *inter1,*inter2,*interres;
{
struct interval temp;
if (inter2->upper_endpoint>0 && inter2->lower_endpoint<0)

            error("Divison by zero error.\n");
temp.lower_endpoint=1/(inter2->upper_endpoint);
temp.upper_endpoint=1/(inter2->lower_endpoint);
imult(inter1,&temp,interres);
}
```

Interval arithmetic FTF algorithm

```
/*FTF Algorithm*/

#define FLOATING
#define MANTISSA_LENGTH 56
#include </u4/call/lib/INTERVAL_ANALYSIS_src/intools.h>
#include <math.h>

#define MAXRND 2147483647.0


int    N;
double calcnte();
double rnum();
double gauss();
double inp,desired;
double X[5];
double Weights[5]={0.9,0.3,-0.3,-0.7,0.1};

double FEED_FORWARD[2]={1.0,0.865};

double XK[2];
float NOISE;



main(argc,argv)
int argc;
char *argv[];
{
spvar A,rescue,y0,alpham1,eNp,eN,gammaN,gammaNp1,epsilon,epsilonp;

spvar tempscal1,tempscal2,rN,rNp,beta,Y,YNp1,C,Cex,B,tempN,W,alpha,CNp1;
struct interval cn;
FILE *fopen(),*fp,*fp1,*fdiag1;
double nte,gain_factor=0.0,*average,temp,MU;

double absolute_error=-1.0,mean,width;

float lambda=0.0,SNR=-1000.0;

int i,k,n,s,p_10,p,ensemble=-1,ens,res_flag,l=0;

char clear_screen=12;
char up_line=11;
char *command,*arg1,*arg2,*arg3,*arg4;
arg4="graph";

/*Initialisation*/
if (argc!=2) res_flag = 1;
else {
        if (strcmp(argv[1],"-on")==0) res_flag=1;

        if (strcmp(argv[1],"-ON")==0) res_flag=1;

        if (strcmp(argv[1],"-off")==0) res_flag=0;
```

```
        if (strcmp(argv[1],"-OFF")==0) res_flag=0;

        }
printf("%c",clear_screen);
printf("Simulation of FTF Algorithm\n\n");
printf("by Chris Callender, 1989\n\n\n\n");
if (res_flag) printf("\n\nRescue=ON, Floating Point Mantissa Length = %d\n\n",M
ANTISSA_LENGTH);
if (!res_flag) printf("\n\nRescue=OFF Floating Point Mantissa Length = %d\n\n",
MANTISSA_LENGTH);
printf("Filter Length:");
scanf("%d",&N);
rvector(A,(N+1));
scalar(rescue);
scalar(y0);

scalar(alpham1);
scalar(eNp);
scalar(eN);
scalar(gammaN);
scalar(gammaNp1);
scalar(alpha);
scalar(epsilon);
scalar(epsilonp);
scalar(tempscal1);
scalar(tempscal2);
scalar(rN);
scalar(rNp);
scalar(beta);
cvector(Y,N);
cvector(YNp1,(N+1));
rvector(C,N);
rvector(Cex,(N+1));
rvector(CNp1,(N+1));
rvector(B,(N+1));
rvector(tempN,N);
rvector(W,N);

fp=fopen("NORMTAPERROR.DAT","w");
fdiag1=fopen("DIAGNOSTIC.DAT","w");
fp1=fopen("FTF_RESCUE_STATS","w");

command="/bin/csh";
arg1="/u4/call/shellscripts/plotshell";
arg3="IFTF ALgoritrhm";
arg2="gplottext.tmp";

srandom(0);


while (lambda<0.8 || lambda>1.0)

{
printf("Please enter a value for lambda between 0.8 and 1.0: ");

scanf("%f",&lambda);
}


while (SNR<-120 || SNR>120) {

printf("\n\nPlease enter SIGNAL/NOISE ratio in dB (-120 - 120db): ");

scanf("%f",&SNR);
}
```

```
while (ensemble<1) {
printf("\n\nHow many runs to make ensemble average: ");
scanf("%d",&ensemble);
}

for(n=0;n!=N;n++) gain_factor=gain_factor+Weights[n]*Weights[n];

gain_factor=gain_factor+FEED_FORWARD[0]*FEED_FORWARD[0]+FEED_FORWARD[1]*FEED_FO
RWARD[1];
gain_factor=sqrt(gain_factor);

NOISE=gain_factor/exp10(SNR/20.0);


absolute_error=0.125*NOISE;

MU=(NOISE*NOISE*lambda)/(N*absolute_error*absolute_error*(1-lambda));

printf("Enter value for MU:[%lf]\n",MU);
scanf("%lf",&MU);

printf("Enter value for absolute error rho:[%lf]\n",absolute_error);
scanf("%lf",&absolute_error);

printf("\n\nHow many data samples per run: ");
scanf("%d",&s);
fprintf(fp,"%d\n\n",s);
average=(double *)malloc(sizeof(double)*(s+1));
if (average==0) {

        fprintf(stderr,"FTF Runtime error...out of memory");
        exit(1);
        }
printf("%cFTF Simulation Running\n\n\n",clear_screen);

p_10=s/10;

for (ens=1;ens!=ensemble+1;ens++) {

for(n=0;n!=N;n++) X[n]=0.0;

for(n=0;n!=2;n++) XK[n]=0.0;


l=0;

setrvector(A,1,1.0,1.0);

setrvector(B,1,1.0,1.0);

zero(C);
zero(W);
nte=calcnte(W.element);
*average=*average+nte/ensemble;
makedata();
setscalar(y0,inp,inp);

copy(y0,tempscal1);

setscalar(tempscal2,-desired,-desired);
copy(tempscal1,alpha);
multiply(alpha,alpha,alpha);
copy(alpha,alpham1);
inverse(tempscal1,tempscal1);
multiply(tempscal1,tempscal2,tempscal1);
getscalar(tempscal1,cn);
setrvector(W,1,cn.upper_endpoint,cn.lower_endpoint);
```

```
setscalar(gammaN,1.0,1.0);

p=p_10;


for(n=1;n!=N+1;n++) {
        nte=calcnte(W.element);
        *(average+n)=*(average+n)+nte/ensemble;
        makedata();
        upshift(Y);
        upshift(YNp1);
        setcvector(Y,1,inp,inp);
        setcvector(YNp1,1,inp,inp);

        multiply(A,YNp1,eNp);

        for(k=1;k!=N+1;k++) {
              getrvector(A,k,cn);
              setrvector(tempN,k,cn.upper_endpoint,cn.lower_endpoint);
              }
        inverse(y0,tempscal1);

        multiply(eNp,tempscal1,tempscal1);
        getscalar(tempscal1,cn);
        setrvector(A,n+1,-cn.lower_endpoint,-cn.upper_endpoint);

        multiply(eNp,gammaN,eN);

        multscal(alpha,lambda,lambda,alpha);

        multiply(eNp,eN,tempscal1);
        add(alpha,tempscal1,alpham1);

        inverse(alpham1,tempscal1);
        multiply(tempscal1,alpha,tempscal1);
        multiply(gammaN,tempscal1,gammaN);

        upshift(C);
        inverse(alpha,tempscal1);
        multiply(tempscal1,eNp,tempscal1);
        getscalar(tempscal1,cn);
        multscal(tempN,cn.lower_endpoint,cn.upper_endpoint,tempN);
        subtract(C,tempN,C);

        if (n==N) {
              copy(C,tempN);
              getscalar(y0,cn);

              multscal(tempN,cn.lower_endpoint,cn.upper_endpoint,tempN);
              getscalar(gammaN,cn);
              multscal(tempN,cn.lower_endpoint,cn.upper_endpoint,tempN);
              for(k=1;k!=N+1;k++) {
                    getrvector(tempN,k,cn);
                    setrvector(B,k,cn.lower_endpoint,cn.upper_endpoint);
                    }
              setrvector(B,N+1,1.0,1.0);


              copy(gammaN,beta);
              multiply(beta,y0,beta);

              multiply(beta,y0,beta);


              }
```

```
        setscalar(tempscal1,desired,desired);
        multiply(W,Y,tempscal2);
        add(tempscal1,tempscal2,epsilonp);

        multiply(epsilonp,gammaN,epsilon);

        if (n<N) {
              inverse(y0,tempscal1);

              multiply(tempscal1,epsilonp,tempscal1);
              getscalar(tempscal1,cn);
              setrvector(W,n+1,-cn.upper_endpoint,-cn.lower_endpoint);
              }
        if (n==N) {
              getscalar(epsilon,cn);
              multscal(C,cn.lower_endpoint,cn.upper_endpoint,tempN);
              add(W,tempN,W);
              }
        /*fprintf(fdiag1,"%20.16lf %20.16lf\n",W.element[0].upper_endpoint,W.element[0
].lower_endpoint);*/
        l++;
        }

for(n=N+1;n!=s+1;n++) {

RE_START:
if (n==p) {
        printf("%cRun #%d:Status %d%%\n",up_line,ens,(p*10)/p_10);

        p=p+p_10;

        }


        makedata();
        upshift(Y);
        upshift(YNp1);
        setcvector(Y,1,inp,inp);
        setcvector(YNp1,1,inp,inp);


        /*    #1        */
        multiply(A,YNp1,eNp);

        /*    #2        */
        multiply(eNp,gammaN,eN);

        /*    #3        */
        copy(alpha,tempscal2);
        multiply(eNp,eN,tempscal1);
        multscal(alpha,lambda,lambda,alpha);
        add(alpha,tempscal1,alpha);


        /*    #4        */
        inverse(alpha,tempscal1);
        multiply(gammaN,tempscal1,gammaNp1);
        multiply(gammaNp1,tempscal2,gammaNp1);
        multscal(gammaNp1,lambda,lambda,gammaNp1);



        /*    #5        */
        for (k=1;k!=N+1;k++) {
              getrvector(C,k,cn);
              setrvector(Cex,k+1,cn.lower_endpoint,cn.upper_endpoint);
              }
```

```
setrvector(Cex,1,0.0,0.0);

inverse(tempscal2,tempscal2);
multiply(tempscal2,eNp,tempscal2);
multscal(tempscal2,1/lambda,1/lambda,tempscal2);
getscalar(tempscal2,cn);
multscal(A,-cn.upper_endpoint,-cn.lower_endpoint,CNp1);
add(CNp1,Cex,CNp1);

/*    #6      */
getscalar(eN,cn);
multscal(Cex,cn.lower_endpoint,cn.upper_endpoint,Cex);
add(Cex,A,A);

/*    #7      */
getrvector(CNp1,N+1,cn);
setscalar(rNp,-cn.upper_endpoint,-cn.lower_endpoint);
multiply(rNp,beta,rNp);
multscal(rNp,lambda,lambda,rNp);

/*    #8      */
getrvector(CNp1,N+1,cn);
setscalar(tempscal1,cn.lower_endpoint,cn.upper_endpoint);
multiply(tempscal1,gammaNp1,tempscal1);
multiply(tempscal1,rNp,tempscal1);
setscalar(tempscal2,1.0,1.0);

add(tempscal1,tempscal2,tempscal1);
copy(tempscal1,rescue);
getscalar(rescue,cn);
if (cn.lower_endpoint<0.0 && res_flag) {

     /*fprintf(fp1,"Division by zero problem at t=%d\n",n);*/
     zero(A);
     setrvector(A,1,1.0,1.0);

     zero(B);
     setrvector(B,N+1,1.0,1.0);

     zero(C);
     temp=pow(lambda,(double )N)*MU;
     setscalar(alpha,temp,temp);
     temp=MU;
     setscalar(beta,temp,temp);
     setscalar(gammaN,1.0,1.0);

     for(k=1;k!=N+1;k++) {
          setrvector(W,k,(W.element[k-1].lower_endpoint+W.element[k-1].upper_endpoint)
/2.0,(W.element[k-1].lower_endpoint+W.element[k-1].upper_endpoint)/2.0);

     }
     l=0;

     goto RE_START;
     }
inverse(tempscal1,tempscal1);
multiply(tempscal1,gammaNp1,gammaN);



/*    #9      */
multiply(rNp,gammaN,rN);

/*    #10     */

multscal(beta,lambda,lambda,beta);
multiply(rNp,rN,tempscal1);
```

```
        add(tempscal1,beta,beta);

        /*    #11      */
        getrvector(CNp1,N+1,cn);
        multscal(B,-cn.upper_endpoint,-cn.lower_endpoint,Cex);
        add(CNp1,Cex,Cex);

        for(k=1;k!=N+1;k++) {
        getrvector(Cex,k,cn);
        setrvector(C,k,cn.lower_endpoint,cn.upper_endpoint);
        }

        setrvector(Cex,N+1,0.0,0.0);


        /*      #12      */
        getscalar(rN,cn);
        multscal(Cex,cn.lower_endpoint,cn.upper_endpoint,Cex);
        add(Cex,B,B);

        /*      #13      */
        setscalar(tempscal1,desired,desired);
        multiply(W,Y,tempscal2);
        getscalar(tempscal2,cn);
        add(tempscal1,tempscal2,epsilonp);

        /*      #14      */
        multiply(epsilonp,gammaN,epsilon);

        /*      #15      */
        getscalar(epsilon,cn);
        multscal(C,cn.lower_endpoint,cn.upper_endpoint,tempN);
        add(W,tempN,W);
        for (k=1;k!=N+1;k++) {
        getrvector(W,k,cn);
        if ((cn.upper_endpoint-cn.lower_endpoint)>absolute_error) {
              /*fprintf(fp1,"Output too wide at t=%d\n",n);*/
              zero(A);
              setrvector(A,1,1.0,1.0);

              zero(B);
              setrvector(B,N+1,1.0,1.0);

              zero(C);
              temp=pow(lambda,(double )N)*MU;
              setscalar(alpha,temp,temp);
              temp=MU;
              setscalar(beta,temp,temp);
              setscalar(gammaN,1.0,1.0);

              for(k=1;k!=N+1;k++) {
                    setrvector(W,k,(W.element[k-1].lower_endpoint+W.element[k-1].upper_endpoint)
/2.0,(W.element[k-1].lower_endpoint+W.element[k-1].upper_endpoint)/2.0);

              }
              l=0;

              goto RE_START;
        }
}

        /*fprintf(fdiag1,"%20.16lf %20.16lf\n",W.element[0].upper_endpoint,W.element[0
].lower_endpoint);*/
        nte=calcnte(W.element);
        *(average+n)=*(average+n)+nte/ensemble;
l++;
}
```

```
}
for(n=0;n!=s+1;n++) {

        fprintf(fp,"%20.16e\n",*(average+n));

        }

fclose(fp);
fclose(fp1);
fclose(fdiag1);

makedB(s,average);

fp=fopen("gplottext.tmp","w");
fprintf(fp,"Floating Point\n");
fprintf(fp,"Mantissa Length %d bits\n\n",MANTISSA_LENGTH);
fprintf(fp,"Filter Length=%d\n",N);
fprintf(fp,"MU=%lf\n",MU);
fprintf(fp,"lambda=%f\n",lambda);
fprintf(fp,"rho=%lf",absolute_error);
fprintf(fp,"SNR=%f\n",SNR);
fprintf(fp,"Ensemble of %d runs\n",ensemble);
fclose(fp);

}

double calcnte(ptr)
double *ptr;
{
int k;
double nte=0.0,mean;

struct interval Weight;


for (k=0;k!=N;k++) {

        Weight.lower_endpoint=((*ptr));
        ptr++;
        Weight.upper_endpoint=((*ptr));
        ptr++;
        mean=(Weight.lower_endpoint+Weight.upper_endpoint)/2;
        nte=nte+(Weights[k]+mean)*(Weights[k]+mean);
        }
return(nte);
}


makedata()
{
int j;

        XK[1]=XK[0];

        XK[0]=gauss();

        inp=XK[0]*FEED_FORWARD[0]+XK[1]*FEED_FORWARD[1];

        for (j=4;j!=0;j--) {

              X[j]=X[j-1];
              }
        X[0]=inp;

        desired=0.0;

        for (j=0;j!=N;j++) desired=desired+X[j]*Weights[j];
```

```
        desired=desired+(gauss())*NOISE;
}


double rnum()
{
return ((random()/MAXRND));
}

double gauss()
{
double a,b;
double result;
a=rnum();
b=rnum();
result=sqrt(-2*log(a))*cos(2*3.141592654*b);
return(result);
}


makedB(s,data)
int s;
double *data;
{
FILE *fopen(),*fp2;
float se;
float init;
double av_level=0;

double p;
int k;

fp2=fopen("ERRdB.DAT","w");


for (k=0;k!=s+1;k++) {

        se=*(data+k);
        if (k==0) init=se;

        p=10*log10(se/init);

        if (k>4*N) av_level+=p;
        fprintf(fp2,"%f\n",p);
        }

printf("Average performance level=%lf\n",av_level/(s-4*N));
fclose(fp2);
}
```

Interval arithmetic fast Kalman algorithm

```
/* Simulation of the Fast Kalman Algorithm */
#define FLOATING
#define MANTISSA_LENGTH 56
#include </u4/call/lib/INTERVAL_ANALYSIS_src/intools.h>
#include <math.h>
#include <stdio.h>

#define MAXRND 2147483647.0


int    N;
double calcnte();
double rnum();
double gauss();
void change_weights();
double inp,desired;
double X[5];
double Weights[5]={0.9,0.3,-0.3,0.7,0.1};

double FEED_FORWARD[2]={1.0,0.865};

double XK[2];
float NOISE;



main(argc,argv)
int argc;
char *argv[];
{
/* All of the spvar definitions should go in here*/
spvar Xnml,enml,a,c,en,epsilon,Cex,m,mu,r,b,w,err,xn,dn,forget,temp,temp1,y;
spvar f,kappa,d,one,x1,W,gamma;
struct interval cn1;
FILE *fopen(),*fp;
double delta=-1.0,nte,gain_factor=0.0,*average;

float lambda=0.0,SNR=-1;

int k,n,s,p_10,p,ensemble=-1,ens,res_flag,rescue_flag=0;

char clear_screen=12;
char up_line=11;



/*Initialisation*/
if (argc!=2) res_flag = 1;
else {
      if (strcmp(argv[1],"-on")==0) res_flag=1;

      if (strcmp(argv[1],"-ON")==0) res_flag=1;

      if (strcmp(argv[1],"-off")==0) res_flag=0;

      if (strcmp(argv[1],"-OFF")==0) res_flag=0;
```

```
        }
printf("%c",clear_screen);
printf("Simulation of Covariance Fast Kalman Algorithm\n\n");
printf("by Chris Callender, 1989\n\n\n\n");
if (res_flag) printf("\n\nRescue=ON, Floating Point Mantissa Length = %d\n\n",M
ANTISSA_LENGTH);
if (!res_flag) printf("\n\nRescue=OFF Floating Point Mantissa Length = %d\n\n",
MANTISSA_LENGTH);
printf("Filter Length:");
scanf("%d",&N);

rvector(Xnm1,N);
rvector(x1,N);
scalar(temp);
cvector(temp1,N);
scalar(forget);
scalar(xn);
scalar(dn);
scalar(enm1);
scalar(en);
cvector(a,N);
cvector(b,N);
cvector(c,N);
cvector(w,N);
scalar(err);
scalar(r);
scalar(mu);
cvector(m,N);
cvector(Cex,(N+1));
scalar(epsilon);
scalar(y);
cvector(f,N);
scalar(kappa);
scalar(gamma);
scalar(one);
cvector(d,N);
matrix(W,N,N);

fp=fopen("NORMTAPERROR.DAT","w");


srandom(1);


while (lambda<0.8 || lambda>1.0)

{
printf("Please enter a value for lambda between 0.8 and 1.0: ");

scanf("%f",&lambda);
}

while (delta<0.0)

{
printf("Please enter a small positive value for delta: ");
scanf("%lf",&delta);
}

while (SNR<0 || SNR>120) {

printf("\n\nPlease enter SIGNAL/NOISE ratio in dB (0 - 120db): ");

scanf("%f",&SNR);
}

while (ensemble<1) {
```

```
printf("\n\nHow many runs to make ensemble average: ");
scanf("%d",&ensemble);
}

for(n=0;n!=N;n++) gain_factor=gain_factor+Weights[n]*Weights[n];

gain_factor=gain_factor*(FEED_FORWARD[0]*FEED_FORWARD[0]+FEED_FORWARD[1]*FEED_F
ORWARD[1]);
gain_factor=sqrt(gain_factor);

NOISE=gain_factor/exp10(SNR/20.0);


printf("\n\nHow many data samples per run: ");
scanf("%d",&s);
fprintf(fp,"%d\n\n",s);
average=(double *)malloc(sizeof(double)*(s+1));
if (average==0) {

        fprintf(stderr,"FK Runtime error...out of memory");
        exit(1);
        }
printf("%cFK Simulation Running\n\n\n",clear_screen);
p=p_10=s/10;


zero(Xnm1);
zero(x1);
zero(w);
nte=calcnte(w.element);
*average=*average+nte/ensemble;
zero(a);
zero(b);
setscalar(one,1.0,1.0);

setscalar(epsilon,delta*lambda,delta*lambda);
setscalar(gamma,delta,delta);
setscalar(forget,lambda,lambda);
zero(W);
for(n=1;n!=N+1;n++) setmatrix(W,n,n,pow((double)lambda,(double)(1-n)),pow((doub
le)lambda,(double)(1-n)));
$       c=((gamma + x1 * W# *x1^)#*W#*x1^)
copy(c,d);

for (ens=1;ens!=ensemble+1;ens++) {

for(n=1;n!=s+1;n++) {
if (n==p) {
        printf("%cRun #%d:Status %d%%\n",up_line,ens,(p*10)/p_10);

        p=p+p_10;

        }
        /*The algorithm goes in here!*/


        RE_START:
        if (rescue_flag==1) {
                copy(Xnm1,x1);
                zero(a);
                zero(b);
                setscalar(epsilon,10.0*lambda,10.0*lambda);

                setscalar(gamma,10.0,10.0);

$               c=((gamma + x1 * W# *x1^)#*W#*x1^)
                copy(c,d);
```

```
            for(k=1;k!=N+1;k++) {
                                    setcvector(w,k,(w.element[k-1].lower_endpoint+w.element
[k-1].upper_endpoint)/2.0,(w.element[k-1].lower_endpoint+w.element[k-1].upper_e
ndpoint)/2.0);

                        }

            rescue_flag=0;

            printf("Rescued at t=%d\n\n",n);
            }

        /*(K1)*/

        makedata();
        setscalar(xn,inp,inp);
        setscalar(dn,desired,desired);
$       enm1=xn-(Xnm1)*a

        /*(K2)*/

$       a=a+c*enm1

        /*(K3)*/

$       en=xn-(Xnm1)*a

        /*(K4)*/
$       epsilon=forget*epsilon+en*enm1


        /*(K5)*/
        if (epsilon.element->lower_endpoint<0.0 && epsilon.element->upper_endpoint>0.0
) {
            rescue_flag=1;
            goto RE_START;
            }

$       temp=en * (epsilon#)
        getscalar(temp,cn1);
        setrvector(Cex,1,cn1.lower_endpoint,cn1.upper_endpoint);
$       temp1=c - a * temp
        for(k=2;k!=N+2;k++) {
            getcvector(temp1,k-1,cn1);
            setcvector(Cex,k,cn1.lower_endpoint,cn1.upper_endpoint);
            }

        /*(K6)*/
        getcvector(Cex,N+1,cn1);
        setscalar(mu,cn1.lower_endpoint,cn1.upper_endpoint);

        for(k=1;k!=N+1;k++) {
            getcvector(Cex,k,cn1);
            setcvector(m,k,cn1.lower_endpoint,cn1.upper_endpoint);
            }

        /*(K7)*/

        getrvector(Xnm1,N,cn1);
        setscalar(temp,cn1.lower_endpoint,cn1.upper_endpoint);

        upshift(Xnm1);
        setrvector(Xnm1,1,inp,inp);
$       r=temp - Xnm1 * b

        /*(K8)*/
```

```
$       temp=one-mu*r
        if (temp.element->lower_endpoint<0.0 && temp.element->upper_endpoint>0.0) {

             rescue_flag=1;
             goto RE_START;
             }

$       b=(b + m * r) * (temp#)

$       f=m + b * mu

$       temp=one - Xnm1 *f * Xnm1 *d
        if (temp.element->lower_endpoint<0.0 && temp.element->upper_endpoint>0.0) {

             rescue_flag=1;
             goto RE_START;
             }

$       kappa=(one - Xnm1 * f *Xnm1 *d)#
        if (kappa.element->lower_endpoint<0.0 && kappa.element->upper_endpoint>0.0) {

             rescue_flag=1;
             goto RE_START;
             }

$       d=kappa#*(d - f*(Xnm1*d))



$       c=f - d*(x1*f)

$       y=Xnm1 * w
$       err=dn - y
$       w=w + c * err
        for(k=1;k!=(N+1);k++) {
             getcvector(w,k,cn1);
             if ((cn1.upper_endpoint-cn1.lower_endpoint)>0.5*NOISE) {

                  rescue_flag=1;
                  goto RE_START;
                  }
             }
        nte=calcnte(w.element);
            *(average+n)=*(average+n)+nte/ensemble;
        }
}
for(n=0;n!=s+1;n++) {

        fprintf(fp,"%20.16e\n",*(average+n));

        }
fclose(fp);
makedB(s,average);

}

double calcnte(ptr)
double *ptr;
{
int k;
double nte,mean;


nte=0.0;

for (k=0;k!=N;k++) {
```

```
        mean=0.5*((*ptr)+(*(ptr+1)));

        ptr+=2;
        nte=nte+(Weights[k]-mean)*(Weights[k]-mean);
        }
return(nte);
}


makedata()
{
int j;

        XK[1]=XK[0];

        XK[0]=gauss();

        inp=XK[0]*FEED_FORWARD[0]+XK[1]*FEED_FORWARD[1];

        for (j=4;j!=0;j--) {

              X[j]=X[j-1];
              }
        X[0]=inp;

        desired=0.0;

        for (j=0;j!=N;j++) desired=desired+X[j]*Weights[j];

        desired=desired+(gauss())*NOISE;
}


double rnum()
{
return ((random()/MAXRND));
}

double gauss()
{
double a,b;
double result;
a=rnum();
b=rnum();
result=sqrt(-2*log(a))*cos(2*3.141592654*b);
return(result);
}


makedB(s,data)
int s;
double *data;
{
FILE *fopen(),*fp2;
float se;
float init;
double p;
int k;

fp2=fopen("ERRdB.DAT","w");


for (k=0;k!=s+1;k++) {

        se=*(data+k);
        if (k==0) init=se;
```

```
        p=10*log10(se/init);

        fprintf(fp2,"%f\n",p);
        }

fclose(fp2);
}

void change_weights(t,file_ptr)
FILE *file_ptr;
int t;
{
}
```

Interval arithmetic FAEST algorithm

```
/*Simulation of FAEST algorithm using floating point interval arithmetic*/

#define FLOATING
#define MANTISSA_LENGTH 56
#include </u4/call/lib/INTERVAL_ANALYSIS_src/intools.h>
#include <stdio.h>
#include <math.h>

#define MAXRND 2147483647.0


int    N;
double calcnte();
double rnum();
double gauss();
void change_weights();
double inp,desired;
double X[5];
double Weights[5]={0.9,0.3,-0.3,0.7,0.1};

double FEED_FORWARD[2]={1.0,0.865};

double XK[2];
float NOISE;



main(argc,argv)
int argc;
char *argv[];
{
/* All of the spvar definitions should go in here*/
spvar XN,w,wmp1,temp1mp1,temp2mp1,a,b,zog,alphaf,alphafold,alphab,alpha;
spvar c,xn,z,ef,eb,e,epsilon,epsilonf,epsilonb,delta,d,aold,forget;
struct interval cn;
FILE *fopen(),*fp;
double sigma=-1.0,nte,gain_factor=0.0,*average,temp;

double lambda=0.0,mean;

float SNR=-1;
int k,n,s,p_10,p,ensemble=-1,ens,res_flag,res_rqd=0;

char clear_screen=12;
char up_line=11;

/*Initialisation*/
if (argc!=2) res_flag = 1;
else {
      if (strcmp(argv[1],"-on")==0) res_flag=1;

      if (strcmp(argv[1],"-ON")==0) res_flag=1;

      if (strcmp(argv[1],"-off")==0) res_flag=0;

      if (strcmp(argv[1],"-OFF")==0) res_flag=0;
```

```
        }
printf("%c",clear_screen);
printf("Simulation of FAEST Algorithm\n\n");
printf("by Chris Callender, 1989\n\n\n\n");
if (res_flag) printf("\n\nRescue=ON, Floating Point Mantissa Length = %d\n\n",M
ANTISSA_LENGTH);
if (!res_flag) printf("\n\nRescue=OFF Floating Point Mantissa Length = %d\n\n",
MANTISSA_LENGTH);
printf("Filter Length:");
scanf("%d",&N);
/*All dimensions of matrices should be set here */


rvector(XN,N);
cvector(w,N);
cvector(wmp1,N+1);
cvector(temp1mp1,(N+1));
cvector(temp2mp1,(N+1));
cvector(a,N);
cvector(aold,N);
cvector(b,N);
scalar(zog);
scalar(alphaf);
scalar(alphafold);
scalar(alphab);
scalar(alpha);
cvector(c,N);
scalar(xn);
scalar(z);
scalar(ef);
scalar(eb);
scalar(e);
scalar(epsilon);
scalar(epsilonf);
scalar(epsilonb);
scalar(delta);
scalar(forget);
cvector(d,N);

fp=fopen("NORMTAPERROR.DAT","w");


srandom(1);


while (lambda<0.8 || lambda>1.0)

{
printf("Please enter a value for lambda between 0.8 and 1.0: ");

scanf("%lf",&lambda);
}
setscalar(forget,lambda,lambda);
while (sigma < 0.0)

{
printf("Please enter a small positive value for sigma: ");
scanf("%lf",&sigma);
}


while (SNR<0 || SNR>120) {

printf("\n\nPlease enter SIGNAL/NOISE ratio in dB (0 - 120db): ");

scanf("%f",&SNR);
}
```

```
while (ensemble<1) {
printf("\n\nHow many runs to make ensemble average: ");
scanf("%d",&ensemble);
}

for(n=0;n!=N;n++) gain_factor=gain_factor+Weights[n]*Weights[n];

gain_factor=gain_factor*(FEED_FORWARD[0]*FEED_FORWARD[0]+FEED_FORWARD[1]*FEED_F
ORWARD[1]);
gain_factor=sqrt(gain_factor);

NOISE=gain_factor/exp10(SNR/20.0);


printf("\n\nHow many data samples per run: ");
scanf("%d",&s);
fprintf(fp,"%d\n\n",s);
average=(double *)malloc(sizeof(double)*(s+1));
if (average==0) {

        fprintf(stderr,"FAEST Runtime error...out of memory");
        exit(1);
        }
printf("%cFAEST Simulation Running\n\n\n",clear_screen);
p_10=s/10;


for (ens=1;ens!=ensemble+1;ens++) {
p=p_10;

for(n=0;n!=N;n++) X[n]=0.0;

for(n=0;n!=2;n++) XK[n]=0.0;


zero(zog);
zero(XN);
zero(a);
zero(b);
zero(c);
zero(w);
zero(wmp1);

nte=calcnte(c.element);
*average=*average+nte/ensemble;
temp=sigma*pow((double)lambda,(double)N);
setscalar(alphaf,temp,temp);
setscalar(alphab,sigma,sigma);
setscalar(alpha,1.0,1.0);


for(n=1;n!=s+1;n++) {
RE_START:
if (res_rqd) {
        res_rqd=0;

        fprintf(stderr,"Rescued at %d\n\n",n);
        zero(a);
        zero(b);
        zero(w);
        temp=100.0*pow((double)lambda,(double)N);

        setscalar(alphaf,temp,temp);
        setscalar(alphab,100.0,100.0);

        setscalar(alpha,1.0,1.0);
```

```
        for(k=0;k!=N;k++) {

              mean=0.5*(c.element[k].lower_endpoint+c.element[k].upper_endpoint);

              fprintf(stderr,"%lf\n",mean);
              setcvector(c,k+1,mean,mean);
              }
        }


if (n==p) {
        printf("%cRun #%d:Status %d%%\n",up_line,ens,(p*10)/p_10);

        p=p+p_10;

        }
        /*The algorithm goes in here!*/

        makedata();
        setscalar(xn,inp,inp);
        setscalar(z,desired,desired);


$       ef=xn + XN * a

        if (alpha.element->lower_endpoint<0.0 && alpha.element->upper_endpoint>0.0) {

              res_rqd=1;
              goto RE_START;
              }
$       epsilonf=ef * (alpha#)

        copy(a,aold);
$       a=a+w * epsilonf

$       alphafold=forget * alphaf
$       alphaf=alphafold+ef*epsilonf

        for(k=1;k!=N+1;k++) {
              getcvector(w,k,cn);
              setcvector(temp1mp1,k+1,cn.lower_endpoint,cn.upper_endpoint);
              getcvector(aold,k,cn);
              setcvector(temp2mp1,k+1,cn.lower_endpoint,cn.upper_endpoint);
              }
        setcvector(temp1mp1,1,0.0,0.0);

        setcvector(temp2mp1,1,1.0,1.0);


        if (alphafold.element->lower_endpoint<0.0 && alphafold.element->upper_endpoint
>0.0) {

              res_rqd=1;
              goto RE_START;
              }
$       wmp1=temp1mp1 - (ef * (alphafold#)) * temp2mp1

        /*Partitioning*/

        for(k=1;k!=N+1;k++) {
              getcvector(wmp1,k,cn);
              setcvector(d,k,cn.lower_endpoint,cn.upper_endpoint);
              }
        getcvector(wmp1,(N+1),cn);
        setscalar(delta,cn.lower_endpoint,cn.upper_endpoint);
```

```
$       eb=zog-delta * alphab * forget

$       w=d - delta * b

$       alpha = alpha + ( ef * alphafold #) * ef + delta * eb

        if (alpha.element->lower_endpoint<0.0 && alpha.element->upper_endpoint>0.0) {

             res_rqd=1;
             goto RE_START;
             }
$       epsilonb = eb * (alpha #)

$       alphab = forget * alphab + eb * epsilonb

$       b=b+w * epsilonb

        upshift(XN);
        setrvector(XN,1,inp,inp);

        /* Time update the LS FIR Filter */

$       e=z + XN * c

$       epsilon = e * (alpha#)

$       c=c + w * epsilon

        nte=calcnte(c.element);
        *(average+n)=*(average+n)+nte/ensemble;
        }
}
for(n=0;n!=s+1;n++) {

        fprintf(fp,"%20.16e\n",*(average+n));

        }
fclose(fp);
makedB(s,average);

}

double calcnte(ptr)
double *ptr;
{
int k;
double nte=0.0;

double mean;

for (k=0;k!=N;k++) {

        mean=0.5*((*ptr)+(*(ptr+1)));

        ptr+=2;
        nte=nte+(mean+Weights[k])*(mean+Weights[k]);
        }
return(nte);
}


makedata()
{
int j;

        XK[1]=XK[0];
```

```
        XK[0]=gauss();

        inp=XK[0]*FEED_FORWARD[0]+XK[1]*FEED_FORWARD[1];

        for (j=4;j!=0;j--) {

              X[j]=X[j-1];
              }
        X[0]=inp;

        desired=0.0;

        for (j=0;j!=N;j++) desired=desired+X[j]*Weights[j];

        desired=desired+(gauss())*NOISE;
}


double rnum()
{
return ((random()/MAXRND));
}

double gauss()
{
double a,b;
double result;
a=rnum();
b=rnum();
result=sqrt(-2*log(a))*cos(2*3.141592654*b);
return(result);
}


makedB(s,data)
int s;
double *data;
{
FILE *fopen(),*fp2;
float se;
float init;
double p;
int k;

fp2=fopen("ERRdB.DAT","w");


for (k=0;k!=s+1;k++) {

        se=*(data+k);
        if (k==0) init=se;

        p=10*log10(se/init);

        fprintf(fp2,"%f\n",p);
        }

fclose(fp2);
}

void change_weights(t,file_ptr)
FILE *file_ptr;
int t;
{
}
```

Fixed point interval arithmetic routines

```
/* Rountines for 16 bit interval multiplication, addition, and division */

/* See Gibb, A., "Algorithm 61 - Procedures for range arithmetic", Comm. ACM,
Vol 4:7, July 1961.*/

/*First define the structure for an interval number*/

struct INTERVAL {
      short int lower_ep; /*16 bit lower endpoint */
      short int upper_ep; /*16 bit upper endpoint */
      };

#define interval struct INTERVAL

void add(range_a,range_b,range_res)
interval *range_a,*range_b,*range_res;
{
int e,f;
if (range_a -> lower_ep > range_a -> upper_ep || range_b -> lower_ep > range_b
-> upper_ep) {
      fprintf(stderr,"Range endpoint error");
      exit(1);
      }
e=(range_a -> lower_ep) + (range_b -> lower_ep);
f=(range_a -> upper_ep) + (range_b -> upper_ep);
/* No need to correct as fixed point addition is exact */
if (e>32767) e=32767;
if (f>32767) f=32767;
if (e<-32768) e=-32768;
if (f<-32768) f=-32768;
range_res -> lower_ep=e;
range_res -> upper_ep=f;
}

void neg(range)
interval *range;
{
short int a;
if (range -> lower_ep > range -> upper_ep) {
      fprintf(stderr,"Range endpoint error");
      exit(1);
      }
a=range -> upper_ep;
range -> upper_ep =-(range -> lower_ep);
range -> lower_ep =-a;
}

short int div(range_a,range_b,range_res,res_shift)
interval *range_a,*range_b,*range_res;
{
int e,f;
int a,b,c,d;
if (range_a -> lower_ep > range_a -> upper_ep || range_b -> lower_ep > range_b
-> upper_ep) {
      fprintf(stderr,"Range endpoint error");
      exit(1);
      }
```

```
a=(range_a -> lower_ep) << res_shift;
b=(range_a -> upper_ep) << res_shift;
c=range_b -> lower_ep;
d=range_b -> upper_ep;

if (c<=0 && d>=0) return(1);


if (c<0) {

        if (b>0) e=b/d; else e=b/c;

        if (a>=0) f=a/c; else f=a/d;

        }
else {
        if (a<0) e=a/c; else e=a/d;

        if (b>0) f=b/c; else f=b/d;

        }
--e;
++f;
if (e>32767) e=32767;
if (f>32767) f=32767;
if (e<-32768) e=-32768;
if (f<-32768) f=-32768;
range_res -> lower_ep = e;
range_res -> upper_ep = f;
return(0);

}


int min(n1,n2)
int n1,n2;
{
if (n1<=n2) return(n1);
if (n2<n1) return(n2);
}

int max(n1,n2)
int n1,n2;
{
if (n1>=n2) return(n1);
if (n2>n1) return(n2);
}

void mul(range_a,range_b,range_res,res_shift)
interval *range_a,*range_b,*range_res;
short int res_shift;
{
int e,f,mask,error;
short int a,b,c,d,temp;

if (range_a -> lower_ep > range_a -> upper_ep || range_b -> lower_ep > range_b
-> upper_ep) {
        fprintf(stderr,"Range endpoint error");
        exit(1);
        }

a=range_a -> lower_ep;
b=range_a -> upper_ep;
c=range_b -> lower_ep;
d=range_b -> upper_ep;

if (a<0 && c>=0) {
```

```
        temp=a;
        a=c;
        c=temp;
        temp=b;
        b=d;
        d=temp;
        }
if (a>=0) {

        if (c>=0) {

             e=a*c;
             f=b*d;
             }
        else {
        e=b*c;
        if (d>=0) f=b*d; else f=a*d;

        }
}
else {
        if (b>0) {

             if (d>0) {

                  e=min(a*d,b*c);
                  f=max(a*c,b*d);
                  }
             else      {
                  e=b*c;
                  f=a*c;
             }
        }
        else {
             f=a*c;
             if (d<=0) e=b*d; else e=a*d;

        }
}
mask=(1 << res_shift)-1;
error=e & mask;
if (error!=0 && e<=0) e=(e >> res_shift)-1;

else e=e >> res_shift;
error=f & mask;
if (error!=0 && f>=0) f=(f >> res_shift)+1;

else f=f >> res_shift;

if (e>32767) e=32767;
if (f>32767) f=32767;
if (e<-32768) e=-32768;
if (f<-32768) f=-32768;
range_res -> lower_ep = e;
range_res -> upper_ep = f;
}

scalar_product(range_a,range_b,range_res,length,res_shift)
interval *range_a,*range_b,*range_res;
short int length,res_shift;
{
short int a,b,c,d,temp;
int index;
int lower_long_accumulator,upper_long_accumulator,mask,error;

lower_long_accumulator=0;
```

```
upper_long_accumulator=0;


for (index=0;index!=length;index++) {

        a=range_a -> lower_ep;
        b=range_a -> upper_ep;
        c=range_b -> lower_ep;
        d=range_b -> upper_ep;
if (a>b || c>d) {
        fprintf(stderr,"Range endpoint error");
        exit(1);
        }
if (a<0 && c>=0) {

        temp=a;
        a=c;
        c=temp;
        temp=b;
        b=d;
        d=temp;
        }
if (a>=0) {

        if (c>=0) {

                lower_long_accumulator=lower_long_accumulator+a*c;
                upper_long_accumulator=upper_long_accumulator+b*d;
                }
        else {
        lower_long_accumulator=lower_long_accumulator+b*c;
        if (d>=0) upper_long_accumulator=upper_long_accumulator+b*d; else upper_long_a
ccumulator=upper_long_accumulator+a*d;
        }
}
else {
        if (b>0) {

                if (d>0) {

                        lower_long_accumulator=lower_long_accumulator+min(a*d,b*c);
                        upper_long_accumulator=upper_long_accumulator+max(a*c,b*d);
                        }
                else {
                        lower_long_accumulator=lower_long_accumulator+b*c;
                        upper_long_accumulator=upper_long_accumulator+a*c;
                        }
        }
        else {
                upper_long_accumulator=upper_long_accumulator+a*c;
                if (d<=0) lower_long_accumulator=lower_long_accumulator+b*d; else lower_long_
accumulator=lower_long_accumulator+a*d;
        }
}
        ++range_a;
        ++range_b;
}

mask=(1 << res_shift)-1;
error=lower_long_accumulator & mask;
if (error!=0 && lower_long_accumulator<=0) lower_long_accumulator=(lower_long_a
ccumulator >> res_shift)-1;
else lower_long_accumulator=lower_long_accumulator >> res_shift;
error=upper_long_accumulator & mask;
if (error!=0 && upper_long_accumulator>=0) upper_long_accumulator=(upper_long_a
ccumulator >> res_shift)+1;
else upper_long_accumulator=upper_long_accumulator >> res_shift;
```

```
if (lower_long_accumulator>32767) lower_long_accumulator=32767;
if (upper_long_accumulator>32767) upper_long_accumulator=32767;
if (lower_long_accumulator<-32768) lower_long_accumulator=-32768;
if (upper_long_accumulator<-32768) upper_long_accumulator=-32768;
range_res -> lower_ep = lower_long_accumulator;
range_res -> upper_ep = upper_long_accumulator;
}
```

Fixed point interval FTF simulation

```
/*Fixed point simulation of FTF algorithm*/

#define MAXRND 2147483647.0


#include <math.h>
#include <stdio.h>
#include "oldrangeroutines.h"
double X[5];
double XK[2];
double FEED_FORWARD[2]={0.15,0.12975};

double Weights[5]={0.9,0.3,-0.3,-0.7,0.2};

double calcnte();
double gauss();
void reinit();
int N;
double NOISE=0.001;

short int inp,des,sat_flag;

main(argc,argv)
int argc;
char *argv[];
{
double nte,d_lambda=0.0,d_MU=-1.0,SNR=-1.0,gain_factor,*average,width;

int s,seed,ens,ensemble=0;

int long_accumulator;
interval *A,*Y,*YNp1,*C,*Cex,*CNp1,*B,*W;
short int index,t,i_width;
interval lambda,mu;
interval rescue,y0,alpham1,eNp,eN,gammaN,alpha,alphaold,epsilon;

interval gammaNp1,epsilonp,rN,rNp,beta,temp,temp1;
FILE *fopen(),*fp;


if (argc!=2) sat_flag = 0;

else {
        if (strcmp(argv[1],"-sat")==0) sat_flag=1;

        if (strcmp(argv[1],"-SAT")==0) sat_flag=1;

        }
fp=fopen("NORMTAPERROR.DAT","w");

printf("Simulation of FTF Algorithm\n\n");
printf("by Chris Callender, 1989\n\n\n\n");
printf("\n\n16 Bit Fixed Point\n\n");
printf("Filter Length:");
scanf("%d",&N);
while (d_lambda<0.8 || d_lambda>1.0)
```

```
{
printf("Please enter a value for lambda between 0.8 and 1.0: ");

scanf("%lf",&d_lambda);
}

while(d_MU<0) {

printf("\n\nPlease enter a value for soft constraint parameter MU:");
scanf("%lf",&d_MU);
}
printf("\n\nPlease enter maximum width of taps:");
scanf("%lf",&width);
i_width=width*32768;

while (SNR<0 || SNR>220) {

printf("\n\nPlease enter SIGNAL/NOISE ratio in dB (0 – 220db): ");

scanf("%lf",&SNR);
}


while (ensemble<1) {
printf("\n\nHow many runs to make ensemble average: ");
scanf("%d",&ensemble);
}

for(t=0;t!=N;t++) gain_factor=gain_factor+Weights[t]*Weights[t];

gain_factor=gain_factor*(FEED_FORWARD[0]*FEED_FORWARD[0]+FEED_FORWARD[1]*FEED_F
ORWARD[1]);
gain_factor=sqrt(gain_factor);

NOISE=gain_factor/exp10(SNR/20.0);


printf("\n\nHow many data samples per run: ");
scanf("%d",&s);
average=(double *)malloc(sizeof(double)*(s+1));
if (average==0) {

      fprintf(stderr,"FTF Runtime error...out of memory");
      exit(1);
      }
/*First, allocate memory for vectors*/

A=(interval *)malloc(sizeof(interval)*(N+1)); /*Scale factor will be 1024*/

Y=(interval *)malloc(sizeof(interval)*N); /*Scale Factor will be 32768*/
YNp1=(interval *)malloc(sizeof(interval)*(N+1)); /*Scale factor will be 32768*/

C=(interval *)malloc(sizeof(interval)*N); /*Scale Factor will be 8*/
CNp1=(interval *)malloc(sizeof(interval)*(N+1)); /*Scale Factor will be 8*/
B=(interval *)malloc(sizeof(interval)*(N+1)); /*Scale Factor will be 32768*/
W=(interval *)malloc(sizeof(interval)*N); /*Scale Factor will be 32768*/

seed=time(0);

srandom(seed);
printf("\n\n");
lambda.lower_ep=lambda.upper_ep=32768*d_lambda;
mu.lower_ep=mu.upper_ep=32768*d_MU;

for (ens=1;ens!=ensemble+1;ens++) {
      for(t=0;t!=N;t++) X[t]=0.0;
```

```
        for(t=0;t!=2;t++) XK[t]=0.0;


/* First the Fast Exact Initialisation Routine*/
A[0].lower_ep=A[0].upper_ep=1024;

B[0].lower_ep=B[0].upper_ep=16384;

for (index=1;index!=N+1;index++) {
        A[index].lower_ep=A[index].upper_ep=0;

        B[index].lower_ep=B[index].upper_ep=0;

        }
for (index=0;index!=N;index++) {

        C[index].lower_ep=C[index].upper_ep=0;

        W[index].lower_ep=W[index].upper_ep=0;

        Y[index].lower_ep=Y[index].upper_ep=0;

        }
for (index=0;index!=N+1;index++) YNp1[index].lower_ep=YNp1[index].upper_ep=0;

nte=calcnte(W);
*average=*average+nte/ensemble;
makedata(0);

y0.lower_ep=y0.upper_ep=inp;

mul(&y0,&y0,&alpha,15);

alpham1.lower_ep=alpha.lower_ep;
alpham1.upper_ep=alpha.upper_ep;
temp.lower_ep=temp.upper_ep=des;
if (div(&temp,&y0,&temp,15)==1) {

                fprintf(stderr,"Algorithm failed during initialisation");
                exit(1);
                }
neg(&temp);
W[0].lower_ep=temp.lower_ep;

W[0].upper_ep=temp.upper_ep;

gammaN.lower_ep=gammaN.upper_ep=32767;

YNp1[0].lower_ep=Y[0].lower_ep=y0.lower_ep;

YNp1[0].upper_ep=Y[0].upper_ep=y0.upper_ep;



for(t=1;t!=N+1;t++) {
        nte=calcnte(W);
        *(average+t)=*(average+t)+nte/ensemble;
        makedata(t);
        for(index=N+1;index!=0;index--) {

                YNp1[index].lower_ep=YNp1[index-1].lower_ep;
                YNp1[index].upper_ep=YNp1[index-1].upper_ep;
                if (index!=N+1) {
                        Y[index].lower_ep=Y[index-1].lower_ep;
                        Y[index].upper_ep=Y[index-1].upper_ep;
                        }
                }
```

```
YNp1[0].lower_ep=YNp1[0].upper_ep=inp;

Y[0].lower_ep=Y[0].upper_ep=inp;


scalar_product(A,YNp1,&eNp,N+1,11);

if (div(&eNp,&y0,&temp,11)==1) {

      fprintf(stderr,"Algorithm failed during initialisation");
      exit(1);
      }

neg(&temp);
A[t].lower_ep=temp.lower_ep;
A[t].upper_ep=temp.upper_ep;

mul(&eNp,&gammaN,&eN,14);

mul(&lambda,&alpha,&alpha,15);

mul(&eNp,&eN,&temp,14);
add(&alpha,&temp,&alpham1);

if (div(&alpha,&alpham1,&temp,15)==1) {
                        fprintf(stderr,"Algorithm failed during initialisation");
                        exit(1);
                        }
mul(&temp,&gammaN,&gammaN,15);

for(index=t;index!=0;index--) {

      C[index]=C[index-1];
      }
C[0].lower_ep=C[0].upper_ep=0;

for(index=0;index!=t;index++) {

      mul(&eNp,(A+index),&temp,10);

      if (div(&temp,&alpha,&temp,4)==1) {
                        fprintf(stderr,"Algorithm failed during initialisation");
                        exit(1);
                        }
      neg(&temp);
      add((C+index),&temp,(C+index));
      }

if (t==N) {
      for(index=0;index!=N;index++) {

            mul(&y0,&gammaN,&temp,15);

            mul(&temp,C+index,B+index,4);
            }
      B[N].lower_ep=B[N].upper_ep=16384;

      mul(&y0,&y0,&temp,15);

      mul(&temp,&gammaN,&beta,9);
      }

epsilonp.lower_ep=epsilonp.upper_ep=des;
scalar_product(Y,W,&temp,N,15);
add(&temp,&epsilonp,&epsilonp);

mul(&epsilonp,&gammaN,&epsilon,15);
```

```
        if (t<N) {
            if (div(&epsilonp,&y0,&temp,15)==1) {

                                fprintf(stderr,"Algorithm failed during initialisation");
                                exit(1);
                                }
            neg(&temp);
            W[t].lower_ep=temp.lower_ep;
            W[t].upper_ep=temp.upper_ep;
            }

        if (t==N) {
            for(index=0;index!=N;index++) {

                mul(&epsilon,C+index,&temp,4);
                add(W+index,&temp,W+index);
                }
            }
        }
/*
/* Now the FTF Algorithm proper */

for (t=N+1;t!=s+1;t++) {
        makedata(t);
        for(index=N+1;index!=0;index--) {

            YNp1[index].lower_ep=YNp1[index-1].lower_ep;
            YNp1[index].upper_ep=YNp1[index-1].upper_ep;
            if (index!=N+1) Y[index].lower_ep=Y[index-1].lower_ep;
            if (index!=N+1) Y[index].upper_ep=Y[index-1].upper_ep;
            }
        YNp1[0].lower_ep=YNp1[0].upper_ep=inp;

        Y[0].lower_ep=Y[0].upper_ep=inp;


        RE_START:
        /*#1*/

        scalar_product(A,YNp1,&eNp,N+1,11);

        /*#2*/

        mul(&eNp,&gammaN,&eN,14);

        /*#3*/
        alphaold.lower_ep=alpha.lower_ep;
        alphaold.upper_ep=alpha.upper_ep;
        mul(&lambda,&alpha,&temp,15);
        mul(&eNp,&eN,&temp1,14);
        add(&temp,&temp1,&alpha);

        /*#4*/
        if (div(&alphaold,&alpha,&temp,10)==1) {

            reinit(A,B,C,W,&alpha,&beta,&gammaN,&mu,&lambda);
            goto RE_START;
            }
        mul(&lambda,&temp,&temp,15);
        mul(&temp,&gammaN,&gammaNp1,10);


        /*#5*/
        mul(&alphaold,&lambda,&temp,15);
        if (div(&eNp,&temp,&temp,5)==1) {
            reinit(A,B,C,W,&alpha,&beta,&gammaN,&mu,&lambda);
            goto RE_START;
```

```
        }
mul(&temp,A,CNp1,11);
neg(CNp1);
for (index=1;index!=N+1;index++) {
        mul(&alphaold,&lambda,&temp,15);
        if (div(&eNp,&temp,&temp,5)==1) {
                reinit(A,B,C,W,&alpha,&beta,&gammaN,&mu,&lambda);
                goto RE_START;
                }
        mul(&temp,A+index,&temp,11);
        neg(&temp);
        add(&temp,C+index-1,CNp1+index);
        }
/*#6*/
for (index=1;index!=N+1;index++) {
        mul(&eN,C+index-1,&temp,8);
        add(A+index,&temp,A+index);
        }
/*#7*/
temp.lower_ep=lambda.lower_ep;
temp.upper_ep=lambda.upper_ep;
neg(&temp);
mul(&temp,&beta,&temp,15);
mul(&temp,CNp1+N,&rNp,11);

mul(&rNp,&gammaN,&temp,15);
mul(&temp,CNp1+N,&temp,4);
temp1.lower_ep=temp1.upper_ep=16384;
add(&temp1,&temp,&rescue);


/*#8*/
if (div(&gammaNp1,&rescue,&gammaN,14)==1) {
                reinit(A,B,C,W,&alpha,&beta,&gammaN,&mu,&lambda);
                goto RE_START;
                }

/*#9*/
mul(&rNp,&gammaN,&rN,11) ;

/*#10*/

mul(&beta,&lambda,&temp,15);
mul(&rNp,&rN,&temp1,11);
add(&temp,&temp1,&beta);

/*#11*/
for(index=0;index!=N+1;index++) {

        mul(CNp1+N,B+index,&temp,14);
        neg(&temp);
        add(CNp1+index,&temp,C+index);
}

/*#12*/
for(index=0;index!=N+1;index++) {

        mul(&rN,C+index,&temp,8);
        add(B+index,&temp,B+index);
        }

/*#13*/
epsilonp.lower_ep=epsilonp.upper_ep=des;
scalar_product(Y,W,&temp,N,15);
if ((temp.upper_ep - temp.lower_ep)>i_width) {
                reinit(A,B,C,W,&alpha,&beta,&gammaN,&mu,&lambda);
                goto RE_START;
```

```
                }

        add(&temp,&epsilonp,&epsilonp);

        /*#14*/
        mul(&epsilonp,&gammaN,&epsilon,15);

        /*#15*/

        for(index=0;index!=N+1;index++) {

        mul(&epsilon,C+index,&temp,3);
        add(W+index,&temp,W+index);
        }
        nte=calcnte(W);
        *(average+t)=*(average+t)+nte/ensemble;
        }
}
for(t=0;t!=s+1;t++) {

        fprintf(fp,"%20.16e\n",*(average+t));

            }

}


double calcnte(ptr)
interval *ptr;
{
int k;
double nte,W;


nte=0.0;

for (k=0;k!=N;k++) {

        W=((ptr -> lower_ep)/32768.0)+((ptr -> upper_ep)/32768.0);

        W=W/2.0;

        ptr++;
        nte=nte+(W+Weights[k])*(W+Weights[k]);
        }
return(nte);
}


makedata(t)
int t;
{
double m_inp,m_des;
int j;

        XK[1]=XK[0];

        if (t==0) XK[0]=-2.0;

        else XK[0]=gauss();

        m_inp=XK[0]*FEED_FORWARD[0]+XK[1]*FEED_FORWARD[1];

        for (j=4;j!=0;j--) {

            X[j]=X[j-1];
            }
```

```
        X[0]=m_inp;

        m_des=0.0;

        for (j=0;j!=N;j++) m_des=m_des+X[j]*Weights[j];

        m_des=m_des+(gauss())*NOISE;
        /*Most ADCs saturate so...*/
        if (m_inp>1.0) m_inp=0.99969482;

        if (m_des>1.0) m_des=0.99969482;

        des=m_des*32768;
        inp=m_inp*32768;
}


double rnum()
{
return ((random()/MAXRND));
}

double gauss()
{
double a,b;
double result;
a=rnum();
b=rnum();
result=sqrt(-2*log(a))*cos(2*3.141592654*b);
return(result);
}

void reinit(A,B,C,W,alpha,beta,gammaN,mu,lambda)
interval *A,*B,*C,*W,*alpha,*beta,*gammaN,*mu,*lambda;
{
short int index,mean;
for (index=0;index!=N+1;index++) {

        A[index].lower_ep=A[index].upper_ep=0;

        B[index].lower_ep=B[index].upper_ep=0;

        if (index!=N) {
            C[index].lower_ep=C[index].upper_ep=0;

            mean=(W[index].lower_ep+W[index].upper_ep) >> 1;
            W[index].lower_ep=W[index].upper_ep=mean;
            }
        }
A[0].lower_ep=A[0].upper_ep=1024;

B[N].lower_ep=B[N].upper_ep=16384;
alpha -> lower_ep=mu -> lower_ep;
alpha -> upper_ep=mu -> upper_ep;
for (index=1;index!=N+1;index++) mul(alpha,lambda,alpha,15);
beta -> lower_ep=mu -> lower_ep;
beta -> upper_ep=mu -> upper_ep;
gammaN -> lower_ep=gammaN -> upper_ep=32767;
}
```

Appendix C

TMS320C25 Assembly Language Software

crossv2.5.1.asm

TMS320C25 processor board initialisation and RS232 comms

```
        idt     'cross v2.51'
        b       start
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        b       dumpstate
start:  call    boardinit
        call    memorycheck
        call    loaddata
        b       4096
```

```
boardinit:
        cnfd
        ldpk  0
        lack  00h
        sacl  4       ;Mask out all interrupts

        lack  255
        sacl  60h
        out   60h,4   ;ACIA Master RESET
        lack  55h
        sacl  60h
        out   60h,4   ; Set 8 bit, odd parity, no stop bits
        rsxm
        ret



memorycheck:
        zac
        sacl  64h
        lack  1
        sacl  60h
        lack  170     ;Checkerboard bit pattern
        sacl  61h
        lalk  4096    ;First program memory location
memloop:tblr  62h
        tblw  61h
        tblr  63h
        tblw  62h
        add   60h
        push
        lac   63h
        sub   61h
        bz    locok   ;If equal to zero memory location is OK
        lack  255
        sacl  64h     ;This sets the memory faulty flag
locok:  pop
        bnz   memloop
        lac 64h
        bz    memok   ;If it is zero, memory is good
        call  print
        .string "M e m o r y  F a u l t"
        .word 0dh,00
        idle
memok:  call print
        .string "T M S 3 2 0 C 2 5  P r o c e s s o r  B o a r d"
        .word  0dh
        .string "6 4 k  P r o g r a m  M e m o r y"
        .word  0dh,0dh,0
        ret
```

```
loaddata:
        call    print
        .string "C R O S S    v2.51"
        .word 0dh
        .string "(c) Chris Callender, 1990"
        .word   0dh,0dh,0dh,0
        lack 1
        sacl 65h
        lalk    4095
loadloop:
        add     65h
        push
        call    gethex
        sacl    61h
        lac     7fh
        bnz     cleanup
        pop
        tblw    61h
        bnz     loadloop
        call    print
        .string "Out of memory error"
        .word   0dh,00
        idle
cleanup:
        pop
        ret


getchar:
        lack    15h
        sacl    70h
        out     70h,4
        in      70h,4
        lac     70h
        andk    1
        bz      getchar
        lack    55h
        sacl    70h
        out     70h,4
        in      71h,5
        zals    71h
        andk    7fh
        ret


putchar:
        sacl 71h
```

```
pl:     in      70h,4
        lac     70h
        andk    2
        bz      pl
        out     71h,5
        zals    71h
        ret



gethex:
        zac
        sacl    7fh
        lark    0,3
        lark    1,60h
        larp    1
hexloop:
        call    getchar
        call    putchar
        call    asciitohex
        larp    1
        sacl    *+,0,0
        larp    0
        banz    hexloop,*-
        zac
        add     60h,12
        add     61h,8
        add     62h,4
        add     63h,0
        ret


puthex:
        sacl    60h
        andk    61440
        sacl    61h
        lac             61h,4
        sach    61h
        zals    61h
        call    hextoascii
        call    putchar
        zals    60h
        andk    0f00h
        sacl    61h
        lac             61h,8
        sach    61h
        zals    61h
        call    hextoascii
        call    putchar
        zals    60h
        andk    00f0h
```

```
        sacl    61h
        lac             61h,12
        sach    61h
        zals    61h
        call    hextoascii
        call    putchar
        zals    60h
        andk    000fh
        call    hextoascii
        call    putchar
        ret


asciitohex:
        andk    7fh
        sacl    7ch
        sblk    48
        blz     invalid
        sblk    9
        blez    digit
        lac     7ch
        sblk    65
        blz     invalid
        lac     7ch
        sblk    71
        bgez    invalid
        lac     7ch
        sblk    55
        ret
digit:  adlk    9
        ret
invalid:
        lack    255
        sacl    7fh
        zac
        ret


hextoascii:
        andk    0fh
        sar     0,7ch
        sar     1,7dh
        adlk    48
        sacl    7eh
        lar     1,7eh
        lark    0,57
        larp    1
        cmpr    2
        bbz finished
        adlk    7
finished:
```

```
        lar     0,7ch
        lar     1,7dh
        ret


print:
        lack    1
        sacl    60h
stringloop:
        pop
        tblr 61h
        add 60h
        push
        lac     61h
        bz      fprint
        call    putchar
        b       stringloop
fprint:
        ret




dumpstate:

;First store processor state
;locations x'60,x'61,x'70,x'71 will be corrupted, so do not use them.

        sst             62h
        ldpk             0
        sst1            63h     ;Store status registers
        sach            64h
        sacl            65h     ;Store 32 bit accumulator
        spm              0
        pac
        sach            66h     ;Store P register
        sacl            67h
        mpyk             1
        pac
        sacl            68h     ;Store T register
        popd            69h
        popd            6ah
        popd            6bh
        popd            6ch
        popd            6dh
        popd            6eh
```

```
        popd    6fh
        popd    72h     ;Save the stack
        sar     0,73h   ;Finally save the aux. registers
        sar     1,74h
        sar     2,75h
        sar     3,76h
        sar     4,77h
        sar     5,78h
        sar     6,79h
        sar     7,7ah

;Now print ACC=accumval, P=pval, T=tval, PC=pcoldval


        call    print
        .word   0dh,0dh
        .string "ACC="
        .word   0
        zals    64h
        call    puthex
        zals    65h
        call    puthex

        call    print
        .string ",  P="
        .word   0
        zals    66h
        call    puthex
        zals    67h
        call    puthex

        call    print
        .string ",  T="
        .word   0
        zals    68h
        call    puthex

        call    print
        .string ",  PC="
        .word   0
        zals    69h
        sblk    1
        call    puthex



        call    print
        .word   0dh,0dh
        .string "Stack[0]="
        .word   0
        zals    69h
        call    puthex
```

```
call        print
.string     "    S t a c k [ 1 ] ="
.word       0
zals        6ah
call        puthex

call        print
.string     "    S t a c k [ 2 ] ="
.word       0
zals        6bh
call        puthex

call        print
.string     ".   S t a c k [ 3 ] ="
.word       0
zals        6ch
call        puthex

call        print
.word       0dh
.string     "S t a c k [ 4 ] ="
.word       0
zals        6dh
call        puthex

call        print
.string     "    S t a c k [ 5 ] ="
.word       0
zals        6eh
call        puthex

call        print
.string     "    S t a c k [ 6 ] ="
.word       0
zals        6fh
call        puthex

call        print
.string     "    S t a c k [ 7 ] ="
.word       0
zals        72h
call        puthex



call        print
.word       0dh,0dh
.string     "S S T 0 ="
.word       0
zals        62h
call        puthex
```

```
call        print
.string     "    S S T 1 ="
.word       0
zals        63h
call        puthex

call        print
.word       0dh,0dh
.string     " A R [ 0 ] ="
.word       0
zals        73h
call        puthex

call        print
.string     "    A R [ 1 ] ="
.word       0
zals        74h
call        puthex

call        print
.string     "    A R [ 2 ] ="
.word       0
zals        75h
call        puthex

call        print
.string     "    A R [ 3 ] ="
.word       0
zals        76h
call        puthex

call        print
.word       0dh
.string     " A R [ 4 ] ="
.word       0
zals        77h
call        puthex

call        print
.string     "    A R [ 5 ] ="
.word       0
zals        78h
call        puthex

call        print
.string     "    A R [ 6 ] ="
.word       0
zals        79h
call        puthex

call        print
```

```
        .string     "    A R [ 7 ] ="
        .word       0
        zals        7ah
        call        puthex

        call        print
        .word       0dh,0dh,0dh
        .string     "P r e s s  a n y  k e y  t o  c o n t i n u e"
        .word       0
        call        getchar
        subk        20h
        bnz         dumpfin
        call        ddump   ;Dump data memory

dumpfin:
        zals        65h
        addh        64h     ;Load 32 bit accumulator
        lt          67h
        mpyk        1
        lt          68h
        lph         66h
        pshd        72h
        pshd        6fh
        pshd        6eh
        pshd        6dh
        pshd        6ch
        pshd        6bh
        pshd        6ah
        pshd        69h
        lar         0,73h   ;Finally save the aux. registers
        lar         1,74h
        lar         2,75h
        lar         3,76h
        lar         4,77h
        lar         5,78h
        lar         6,79h
        lar         7,7ah

        lst         62h
        lst1        63h     ;Load status registers
        ret

ddump:  lack    0dh
        call    putchar
        call    print
        .string "E n t e r  S t a r t  A d d r e s s :"
        .word   0000h
        call    gethex
        sacl    68h
        lar     0,68h
        lack    0dh
        call    putchar
```

```
        call    putchar

        lark    2,15
loop:   lark    1,7
        sar     0,60h
        zals    60h
        call    puthex
        lack    58
        call    putchar
loop1:  larp    0
        zals    *+
        call    puthex
        lack    20h
        call    putchar
        larp    1
        banz    loop1,*-
        lack    0dh
        call    putchar
        larp    2
        banz    loop,*-
        zals    7fh
        bz      ddump
        ret
        nop
        nop
```

scalars.lib

Assembler macros for scalar interval operations

```
DIV   $MACRO  NAR,DAR,QAR,TEMSGN,SHIFT

      LARP    :NAR:
      PSHD    *,:DAR:
      PSHD    *,:NAR:
      LT      *,:DAR:
      MPY     *,:DAR:
      PAC
      SACH    :TEMSGN:
      LAC     *,0,:DAR:
      ABS
      SACL    *,0,:NAR:
      LAC     *,:SHIFT:,:DAR:
      ABS
      RPTK    15
      SUBC    *,:DAR:
      LARP    :QAR:
      SACL    *,0,:QAR:
      LAC     :TEMSGN:
      BGEZ    DONE?,*,:QAR:
      ZAC
      SUB     *,0,:QAR:
      SACL    *
DONE?
      LARP    :DAR:
      POPD    *,:NAR:
      POPD    *
      $ENDM

S_ADD   $MACRO  src1,src2,res

      LRLK    0,:src1:
      LRLK    1,:src2:
      LRLK    2,:res:

      LARP    0

AL?   LAC     *+,0,1
      ADD     *+,0,2
      SACL    *+,0,0
      LAC     *+,0,1
      ADD     *+,0,2
      SACL    *+,0
      $ENDM

;Macro to perfrom res=src1*2**shift/src2 where src1,src2,res are
;Intervals
```

```
S_DIV   $MACRO  src1,src2,res,scratch,shift
        LRLK    0,:src1:
        LRLK    1,:src1:+1
        LRLK    2,:src2:
        LRLK    3,:src2:+1
        LRLK    4,:res:
        LRLK    5,:res:+1

        LARP    2
        LAC     *
        BGZ     S5?
        BZ      ERROR?
        LARP    3
        LAC     *
        BLZ     S1?

ERROR?  STC
        B       END?


S1?     LARP    1
        LAC     *
        BLEZ    S2?
        DIV     1,3,4,:scratch:,:shift:
        B       S3?
S2?     DIV     1,2,4,:scratch:,:shift:
S3?     LARP    0
        LAC     *
        BLZ     S4?
        DIV     0,2,5,:scratch:,:shift:
        B       S9?
S4?     DIV     0,3,5,:scratch:,:shift:
        B       S9?
S5?     LARP    0
        LAC     *
        BGEZ    S6?
        DIV     0,2,4,:scratch:,:shift:
        B       S7?
S6?     DIV     0,3,4,:scratch:,:shift:
S7?     LARP    1
        LAC     *
        BLEZ    S8?
        DIV     1,2,5,:scratch:,:shift:
        B       S9?
S8?     DIV     1,3,5,:scratch:,:shift:
S9?
        SEPCOR  4,5
        RTC
END?
        $ENDM
```

```
S_MULT  $MACRO  src1,src2,res,temp,res_shift

        LRLK    0,2
        LRLK    1,:src1:
        LRLK    2,:src1:+1
        LRLK    3,:src2:
        LRLK    4,:src2:+1
        LRLK    5,:res:
        LRLK    6,:res:+1


MCLOOP? SSMULT  1,3,:temp:,:res_shift:
        SSMULT  1,4,:temp:+1,:res_shift:
        SSMULT  2,3,:temp:+2,:res_shift:
        SSMULT  2,4,:temp:+3,:res_shift:

        SSMIN_4 :temp:
        LARP    5
        SACL    *,0,6
        SSMAX_4 :temp:
        SACL    *,0,5
        SEPCOR  5,6
        $ENDM

S_SUB   $MACRO  src1,src2,res

        LRLK    0,:src1:
        LRLK    1,:src2:
        LRLK    2,:res:

        LARP    0

SL?     LAC     *+,0,1
        SUB     *+,0,2
        SACL    *+,0,0
        LAC     *+,0,1
        SUB     *+,0,2
        SACL    *+,0
        $ENDM

SEPCOR  $MACRO  arp1,arp2
        LARP    :arp1:
        LAC     *,0,:arp1:
        BGZ     ok1?,*,:arp2:
        SUBK    1
        LARP    :arp1:
        SACL    *,0,:arp2:
ok1?
        LAC     *,0,:arp2:
        BLZ     ok2?
        ADDK    1
        SACL    *,0,:arp2:
```

```
ok2?
      $ENDM

SSMAX_2 $MACRO  addr
      LAC     :addr:
      SUB     :addr:+1
      BGEZ    MX?
      LAC     :addr:+1
      B       END?
MX?   LAC     :addr:
END?
      $ENDM

SSMAX_4 $MACRO  addr
      SSMAX_2 :addr:
      SACL    :addr:+4
      SSMAX_2 :addr:+2
      SACL    :addr:+5
      SSMAX_2 :addr:+4
      $ENDM

SSMIN_2 $MACRO  addr
      LAC     :addr:
      SUB     :addr:+1
      BGEZ    MN?
      LAC     :addr: ;The one at (addr) is smaller.
      B       END?
MN?   LAC     :addr:+1 ;The one at (addr+1) is smaller.
END?
      $ENDM

SSMULT  $MACRO  arp1,arp2,res,res_shift
      LARP    :arp1:
      LT      *,:arp2:
      MPY     *
      PAC
TEST? .SET    :res_shift:>8
      $IF     TEST?
      SACH    :res:,16-:res_shift:
      $ELSE
      RPTK    8-:res_shift:
      SFL
      SACH    :res:,7
      $ENDIF
      $ENDM

S_NEG   $MACRO  addr

      LRLK    0,:addr:
      LARP    0
      LAC     *+,0,0
      PUSH
```

```
LAC     *-,0,0
NEG
SACL    *+,0,0
POP
NEG
SACL    *,0,0
$ENDM
```

vectors.lib

Assembler macros for vector interval operations

```
;16 bit multiplication macro. Multiplies arp1 * arp 2 and stores result
;at res

SMULT   $MACRO  arp1,arp2,res,res_shift
        LARP    :arp1:
        LT      *,:arp2:
        MPY     *
        PAC
TEST?   .set    :res_shift:>8
        $IF TEST?
        SACH    :res:,16-:res_shift:
        $ELSE
        RPTK    8-:res_shift:
        SFL
        SACH    :res:,7
        $ENDIF
        $ENDM

;Minimum of 2 numbers. Looks at the 16 bit numbers at
;(addr) and (addr+1) and sets the accumulator
;to the smaller of them.

SMIN_2  $MACRO  addr
        LAC     :addr:
        SUB     :addr:+1
        BGEZ    MN?
        LAC     :addr: ;The one at (addr) is smaller.
        B       END?
MN?     LAC     :addr:+1 ;The one at (addr+1) is smaller.
END?
        $ENDM

SMAX_2  $MACRO  addr
        LAC     :addr:
        SUB     :addr:+1
        BGEZ    MX?
        LAC     :addr:+1
        B       END?
MX?     LAC     :addr:
END?
        $ENDM

;Macro to find the smallest of (addr), (addr+1)
;(addr+2), (addr+3). Uses addr+4,addr+5
;as temporary storage.

SMIN_4  $MACRO  addr
        SMIN_2  :addr:
        SACL    :addr:+4
        SMIN_2  :addr:+2
```

```
        SACL    :addr:+5
        SMIN_2  :addr:+4
        $ENDM

SMAX_4  $MACRO  addr
        SMAX_2  :addr:
        SACL    :addr:+4
        SMAX_2  :addr:+2
        SACL    :addr:+5
        SMAX_2  :addr:+4
        $ENDM

MSC     $MACRO  src1,src2,res,size,temp,res_shift

        LRLK    0,2
        LRLK    1,:src1:
        LRLK    2,:src1:+1
        LRLK    3,:src2:
        LRLK    4,:src2:+1
        LRLK    5,:res:
        LRLK    6,:res:+1
        LRLK    7,:size:-1


MCLOOP? SMULT   1,3,:temp:,:res_shift:
        SMULT   1,4,:temp:+1,:res_shift:
        SMULT   2,3,:temp:+2,:res_shift:
        SMULT   2,4,:temp:+3,:res_shift:

        SMIN_4  :temp:
        LARP    5
        SACL    *,0,6
        SMAX_4  :temp:
        SACL    *,0,1

        VEPCOR  5,6
        LARP    1
        MAR     *+,1
        MAR     *+,2
        MAR     *+,2
        MAR     *+,5
        MAR     *+,5
        MAR     *+,6
        MAR     *+,6
        MAR     *+,7
        BANZ    MCLOOP?,*-

        $ENDM

;32 bit multiplication macro. Multiplies arp1 * arp 2 and stores result
;at res
```

```
LMULT  $MACRO  arp1,arp2,res
       LARP    :arp1:
       LT      *,:arp2:
       MPY     *
       PAC
       SACL    :res:
       SACH    :res:+1
       $ENDM

V_SHIFT $MACRO  src,size

WORDS? .set    (:size:)*2

       LRLK    0,:src:+WORDS?-2
       LARP    0
       RPTK    WORDS?-2
       DMOV    *-,0
       LRLK    0,:src:+WORDS?-2
       RPTK    WORDS?-3
       DMOV    *-,0
       $ENDM

;Minimum of 2 numbers. Looks at the 32 bit numbers at
;(addr,addr+1) and (addr+2,addr+3) and sets the accumulator
;to the smaller of them.

LMIN_2 $MACRO  addr
       ZALS    :addr:
       ADDH    :addr:+1
       SUBS    :addr:+2
       SUBH    :addr:+3
       BGEZ    MN?
       ZALS    :addr:
       ADDH    :addr:+1 ;The one at (addr,addr+1) is smaller.
       B       END?
MN?    ZALS    :addr:+2
       ADDH    :addr:+3 ;The one at (addr+2,addr+3) is smaller.
END?
       $ENDM

;Macro to find the smallest of (addr,addr+1), (addr+2,addr+3)
;(addr+4,addr+5), (addr+6,addr+7). Uses addr+8,addr+9,addr+10,addr+11
;as temporary storage.

LMIN_4 $MACRO  addr
       LMIN_2  :addr:
       SACL    :addr:+8
       SACH    :addr:+9
       LMIN_2  :addr:+4
       SACL    :addr:+10
       SACH    :addr:+11
       LMIN_2  :addr:+8
```

```
      $ENDM

LMAX_4 $MACRO addr
      LMAX_2  :addr:
      SACL    :addr:+8
      SACH    :addr:+9
      LMAX_2  :addr:+4
      SACL    :addr:+10
      SACH    :addr:+11
      LMAX_2  :addr:+8
      $ENDM

SCPROD $MACRO src1,src2,res,size,temp,res_shift

      LRLK    0,2
      LRLK    1,:src1:
      LRLK    2,:src1:+1
      LRLK    3,:src2:
      LRLK    4,:src2:+1
      LRLK    5,:res:
      LRLK    6,:res:+1
      LRLK    7,:size:-1

      ZAC
      SACL    :temp:
      SACL    :temp:+1
      SACL    :temp:+2
      SACL    :temp:+3

SPLOOP? LMULT   1,3,:temp:+4
      LMULT   1,4,:temp:+6
      LMULT   2,3,:temp:+8
      LMULT   2,4,:temp:+10

      LMIN_4  :temp:+4
      ADDS    :temp:
      ADDH    :temp:+1
      SACL    :temp:
      SACH    :temp:+1
      LMAX_4  :temp:+4
      ADDS    :temp:+2
      ADDH    :temp:+3
      SACL    :temp:+2
      SACH    :temp:+3

      RC
      LARP    1
      MAR     *+,1
      MAR     *+,2
      MAR     *+,2
      MAR     *+,3
      MAR     *+,3
```

```
      MAR    *+,4
      MAR    *+,4
      MAR    *+,7
      BANZ   SPLOOP?,*-
      ZALS   :temp:
      ADDH   :temp:+1
      LARP   5
      SACH   *,16-:res_shift:,6
      ZALS   :temp:+2
      ADDH   :temp:+3
      SACH   *,16-:res_shift:,5
      VEPCOR 5,6
      $ENDM

VEPCOR $MACRO arp1,arp2
      LARP   :arp1:
      LAC    *,0,:arp1:
      BGZ    ok1?,*,:arp2:
      SUBK   1
      LARP   :arp1:
      SACL   *,0,:arp2:
ok1?
      LAC    *,0,:arp2:
      BLZ    ok2?
      ADDK   1
      SACL   *,0,:arp2:
ok2?
      $ENDM

V_ADD  $MACRO src1,src2,res,size

      LRLK   0,:src1:
      LRLK   1,:src2:
      LRLK   2,:res:
      LRLK   3,:size:-1

      LARP   0

AL?   LAC    *+,0,1
      ADD    *+,0,2
      SACL   *+,0,0
      LAC    *+,0,1
      ADD    *+,0,2
      SACL   *+,0,3
      BANZ   AL?,*-,0
      $ENDM

V_SUB  $MACRO src1,src2,res,size

      LRLK   0,:src1:
      LRLK   1,:src2:+1
      LRLK   2,:res:
```

```
        LRLK    3,:size:-1

        LARP    0

SL?     LAC     *+,0,1
        SUB     *-,0,2
        SACL    *+,0,0
        LAC     *+,0,1
        SUB     *+,0,1
        MAR     *+,1
        MAR     *+,2
        SACL    *+,0,3
        BANZ    SL?,*-,0
        $ENDM
```

system.lib

Assembler macros for input/output

```
INADC1  $MACRO  addr
        in              0:addr:
        $ENDM

INADC2  $MACRO  addr
                in              1,:addr:
        $ENDM

OUTDAC  $MACRO  addr
                out             2,:addr:
        $ENDM

TRIGSYNC        $MACRO
         WAIT?bioz WAIT?
        $ENDM
```

ftf.asm

Assembler program for 16 bit fixed point interval FTF adaptive filter

```
        .mlib   b
        .mlib   b
        .mlib   b ; Link in appropriate macros
        .mnolist


;Design Constants
;****************
;

lambda_v .set   32767           ;lambda*32768
mu_v    .set    500             ;mu*32768
rho_v   .set    300             ;rho*32768
N       .set    5               ;Filter length
startmu .set    100             ;Initialisation mu
startal .set    100             ;startmu*lamndaˆN
reinal  .set    500             ;mu*lamndaˆN



;Data memory assignments
;***********************
;

;System

INBUF   .set    60h
DESBUF  .set    61h
OUTBUF  .set    62h
SCRATCH .set    60h


;Scalars

lambda          .set    200h
mu              .set    202h
rescue          .set    204h
y0              .set    206h
alpham1         .set    208h
eNp             .set    20ah
eN              .set    20ch
gammaN          .set    20eh
alpha           .set    210h
alphaold        .set    212h
epsilon         .set    214h
gammaNp1        .set    216h
epsilonp        .set    218h
rN              .set    21ah
rNp             .set    21ch
beta            .set    21eh
t               .set    220h
```

```
;Vectors

A           .set   224h
B           .set   A+(2*(N+1))
Y           .set   B+(2*(N+1))
C           .set   Y+(2*(N+1))
CNp1          .set   C+2*N
W           .set   CNp1+(2*(N+1))
TEMP          .set   W+2*N
TEMP1         .set   TEMP+2*N+2



;Initialisation of values

      nop
      ssxm
      sovm
      ldpk   0

      lalk   lambda_v
      lrlk   0,lambda
      larp   0
      sacl   *+,0,0
      sacl   *,0,0          ;Set lambda=[lambda_v,lambda_v]

      lalk   mu_v
      lrlk   0,mu
      sacl   *+,0,0          ;Set mu=[mu_v,mu_v]
      sacl   *,0,0

      lalk   1024
      lrlk   0,A
      sacl   *+,0,0
      sacl   *,0,0          ;Set A[0]=[1.0,1.0]

      lalk   16384
      lrlk   0,B+2*N
      sacl   *+,0,0
      sacl   *              ;Set B[N]=[1.0,1.0]

      lrlk   0,2*N-1
      lrlk   1,A+2
      lrlk   2,B
      lrlk   3,C
      lrlk   4,W

      zac
      larp   1
LP1   sacl   *+,0,2          ;Set A[1...N+1]=[0,0]
```

```
        sacl    *+,0,3          ;Set B[0...N]=[0,0]
        sacl    *+,0,4          ;Set C[0...N]=[0,0]
        sacl    *+,0,0          ;Set W[0...N]=[0,0]
        banz    LP1,*-,1

        lrlk    0,2*N+1
        lrlk    1,Y
        larp    1
LP2     sacl    *+,0,0          ;Set Y[0...N+1]=[0,0]
        banz    LP2,*-,1

        lrlk    0,gammaN        ;Set gammaN = 1.0
        lalk    32767
        larp    0
        sacl    *+,0,0
        sacl    *

        lrlk    0,alpha         ;Set alpha=startal
        lalk    startal
        larp    0
        sacl    *+,0,0
        sacl    *

        lrlk    0,beta
        lalk    startmu
        larp    0
        sacl    *+,0,0
        sacl    *

        lrlk    0,t
        lack    N
        larp    0
        sacl    *


mainloop
        v_shift Y,(N+1)
        trigsync
        inadc1  INBUF
        inadc2  DESBUF
        lrlk    0,Y
        lrlk    1,INBUF
        larp    1
        lac     *,0,0
        sacl    *+,0,0
        sacl    *,0,0
        lrlk    0,epsilonp
        lrlk    1,DESBUF
        larp    1
        lac     *,0,0
        sacl    *+,0,0
        sacl    *,0,0
```

```
restart

      ;Eq 1
      scprod  A,Y,eNp,N+1,SCRATCH,11

      ;Eq 2
      s_mult  eNp,gammaN,eN,SCRATCH,14

      ;Eq 3
      lrlk    0,alpha
      lrlk    1,alphaold
      larp    0
      lac     *+,0,1
      sacl    *+,0,0
      lac     *,0,1
      sacl    *,0,0
      s_mult  lambda,alpha,temp,SCRATCH,15
      s_mult  eNp,eN,temp+2,SCRATCH,14
      s_add   temp,temp+2,alpha

      ;Eq 4
      s_div   alphaold,alpha,temp,SCRATCH,10
      larp    0
      lrlk    0,t
      lac     *
      bgz     okeq4
      bbnz    reinit
okeq4 s_mult  lambda,temp,temp+2,SCRATCH,15
      s_mult  temp+2,gammaN,gammaNp1,SCRATCH,10


      ;Eq 5
      s_mult  alphaold,lambda,temp,SCRATCH,15
      s_div   eNp,temp,temp+2,SCRATCH,5
      larp    0
      lrlk    0,t
      lac     *
      bgz     okeq5
      bbnz    reinit
okeq5 s_mult  temp+2,A,CNp1,SCRATCH,11
      s_neg   CNp1
      msc     A+2,temp+2,temp1,N,SCRATCH,11
      v_sub   C,temp1,CNp1+2,N

      ;Eq 6

      msc     C,eN,temp+2,N,SCRATCH,8
      zac
      lrlk    0,temp
      larp    0
      sacl    *+,0,0
```

```
        sacl    *,0,0
        v_add   A,temp,A,N+1


        ;Eq 7
        lrlk    1,temp
        lrlk    0,lambda
        larp    0
        lac     *+,0,1
        sacl    *+,0,0
        lac     *,0,1
        sacl    *,0,0
        s_neg   temp
        s_mult  temp,beta,temp+2,SCRATCH,15
        s_mult  temp+2,CNp1+2*N,rNp,SCRATCH,11

        ;Eq 8
        s_mult  rNp,gammaN,temp,SCRATCH,15
        s_mult  temp,CNp1+2*N,temp+2,SCRATCH,4
        lalk    16384
        lrlk    0,temp1
        larp    0
        sacl    *+,0,0
        sacl    *,0,0
        s_add     temp+2,temp1,rescue


        ;Eq 8
        s_div   gammaNp1,rescue,gammaN,SCRATCH,14
        lrlk    0,t
        larp    0
        lac     *
        bgz     okeq8
        bbnz    reinit
okeq8

        ;Eq 9
        s_mult  rNp,gammaN,rN,SCRATCH,11

        ;Eq 10
        s_mult  beta,lambda,temp,SCRATCH,15
        s_mult  rNp,rN,temp1,SCRATCH,11
        s_add   temp,temp1,beta

        ;Eq 11
        msc     B,CNp1+2*N,temp,N,SCRATCH,14
        v_sub   CNp1,temp,C,N

        ;Eq 12
        msc     C,rN,temp,N,SCRATCH,8
        v_add   B,temp,B,N
```

```
      ;Joint process extension

      ;Eq 13
      scprod  Y,W,temp,N,SCRATCH,12
      lrlk    0,temp+1
      larp    0
      lac     *-,0,0
      sub     *,0,0
      subk    rho_v
      blz     okeq13
      lrlk    0,t
      lac     *
      bz      reinit
okeq13 lrlk   0,temp
      larp    0
      lac     *+,0,0
      add     *-,0,0
      sfr
      sacl    OUTBUF
      outdac  OUTBUF

      s_add   epsilonp,temp,epsilonp

      ;Eq 14
      s_mult  epsilonp,gammaN,epsilon,SCRATCH,15

      ;Eq 15
      msc     C,epsilon,temp,N,SCRATCH,6
      v_add   W,temp,W,N


      lrlk    0,t
      larp    0
      lac     *
      bz      mainloop
      subk    1
      sacl    *
      b       mainloop


reinit
      larp    0
      lalk    1024
      lrlk    0,A
      sacl    *+,0,0
      sacl    *,0,0          ;Set A[0]=[1.0,1.0]

      lalk    16384
      lrlk    0,B+2*N
      sacl    *+,0,0
      sacl    *,0,1
```

```
        lrlk    0,2*N-1
        lrlk    1,A+2
        lrlk    2,B
        lrlk    3,C

        zac
RLP1    sacl    *+,0,2
        sacl    *+,0,3
        sacl    *+,0,0
        banz    RLP1,*-,1

        lrlk    0,gammaN
        lalk    32767
        larp    0
        sacl    *+,0,0
        sacl    *,0,0

        lrlk    0,alpha
        lalk    reinal
        sacl    *+,0,0
        sacl    *,0,0

        lrlk    0,beta
        lalk    mu_v
        sacl    *+,0,0
        sacl    *,0,0

        lrlk    0,W
        lrlk    1,W
        lrlk    2,(N-1)
MEANLP  lac     *+,0,0
        add     *+,0,1
        sfr
        sacl    *+,0,1
        sacl    *+,0,2
        banz    MEANLP,*-,0


        b       restart
```

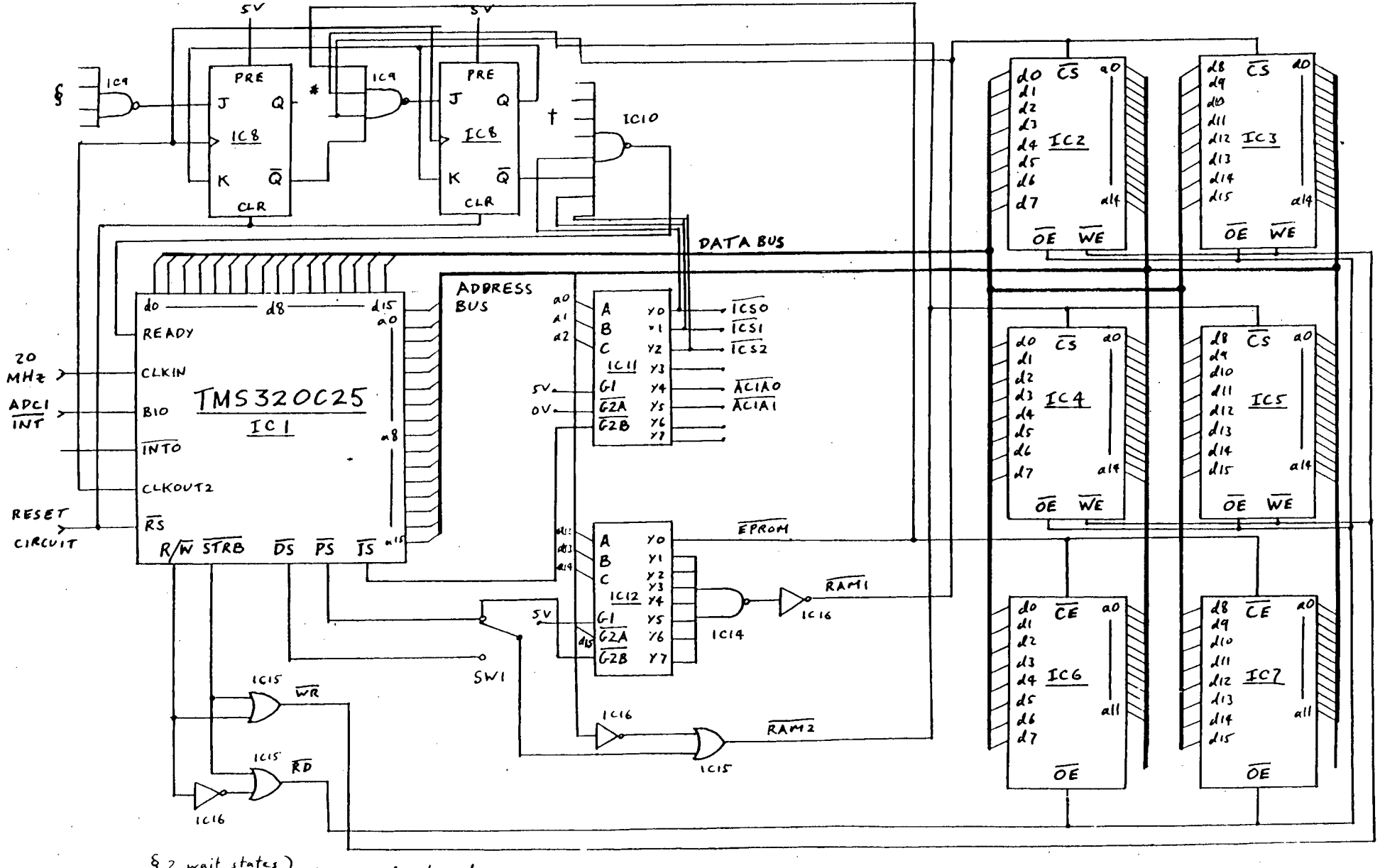
Appendix D

Circuit Diagrams

Processor Board

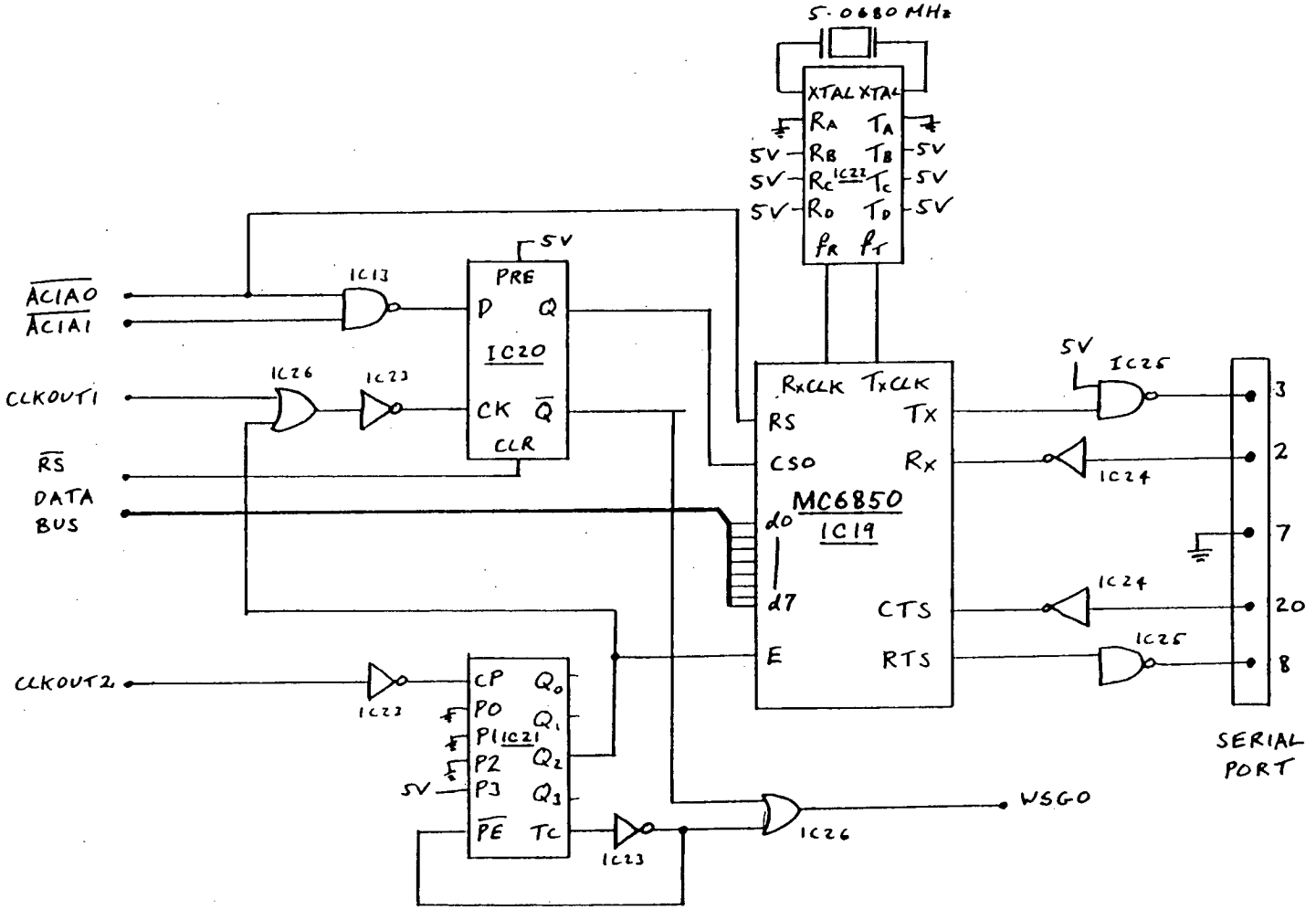
The Processor Board circuits are included on the following pages. IC numbers and part descriptions are listed below.

IC	Part	Description
1	TMS320C25GBL	DSP Microprocessor
2,3,4,5	MS62256L	32K x 8 bit RAM
6,7	TMS2732A	4K x 8 EPROM
8	74LS114AN	Dual J-K Flip Flop
9	74ALS20AN	Dual 4-input NAND
10	74ALS30N	8-input NAND
11,12	74ALS138N	3 x 8 Line Decoder
13	74S00N	Quad 2-input NAND
14	74LS30N	8-input NAND
15	74LS32N	Quad 2-input OR
16	74LS04N	Hex Inverter
17,18	74LS373N	D-Type Octal Transparent Latch
19	MC6850P	Asynchronous Communications Interface Adaptor
20	74LS74	D-type +ve edge Triggered Flip Flop
21	74ALS161	4-bit Binary Counter
22	COM8116P	Clock Generator
23	74LS04	Hex Inverter
24	DS1489	Line Receiver
25	DS1488	Line Driver
26	74LS32	Quad 2-input OR

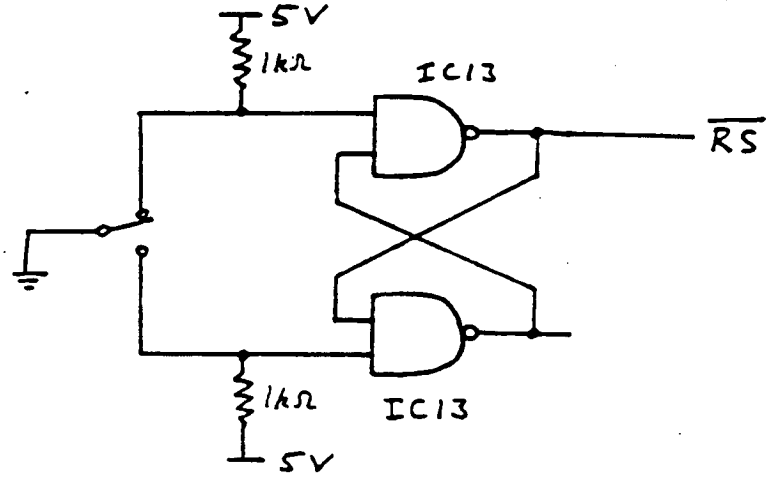


§ 2 wait states
* 1 wait states
† 0 wait states
} tie to 5V if not used

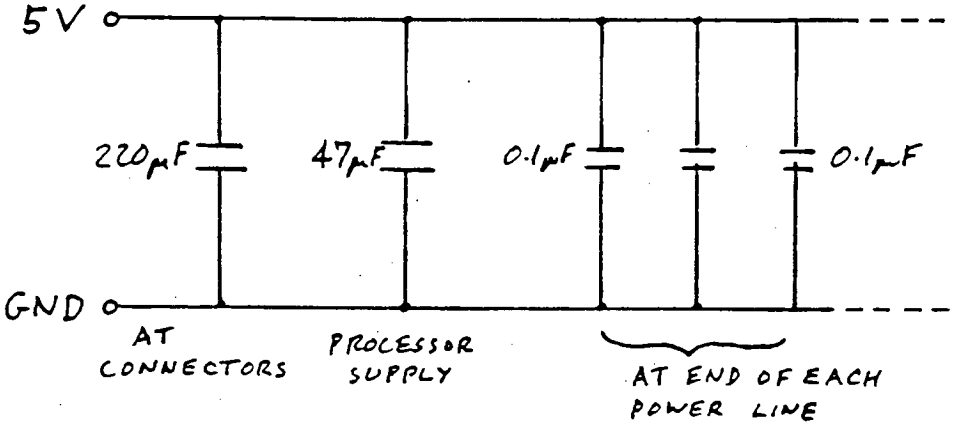
PROCESSOR BOARD CIRCUIT DIAGRAM



SERIAL PORT CIRCUIT DIAGRAM



Reset Debounce Circuit Diagram

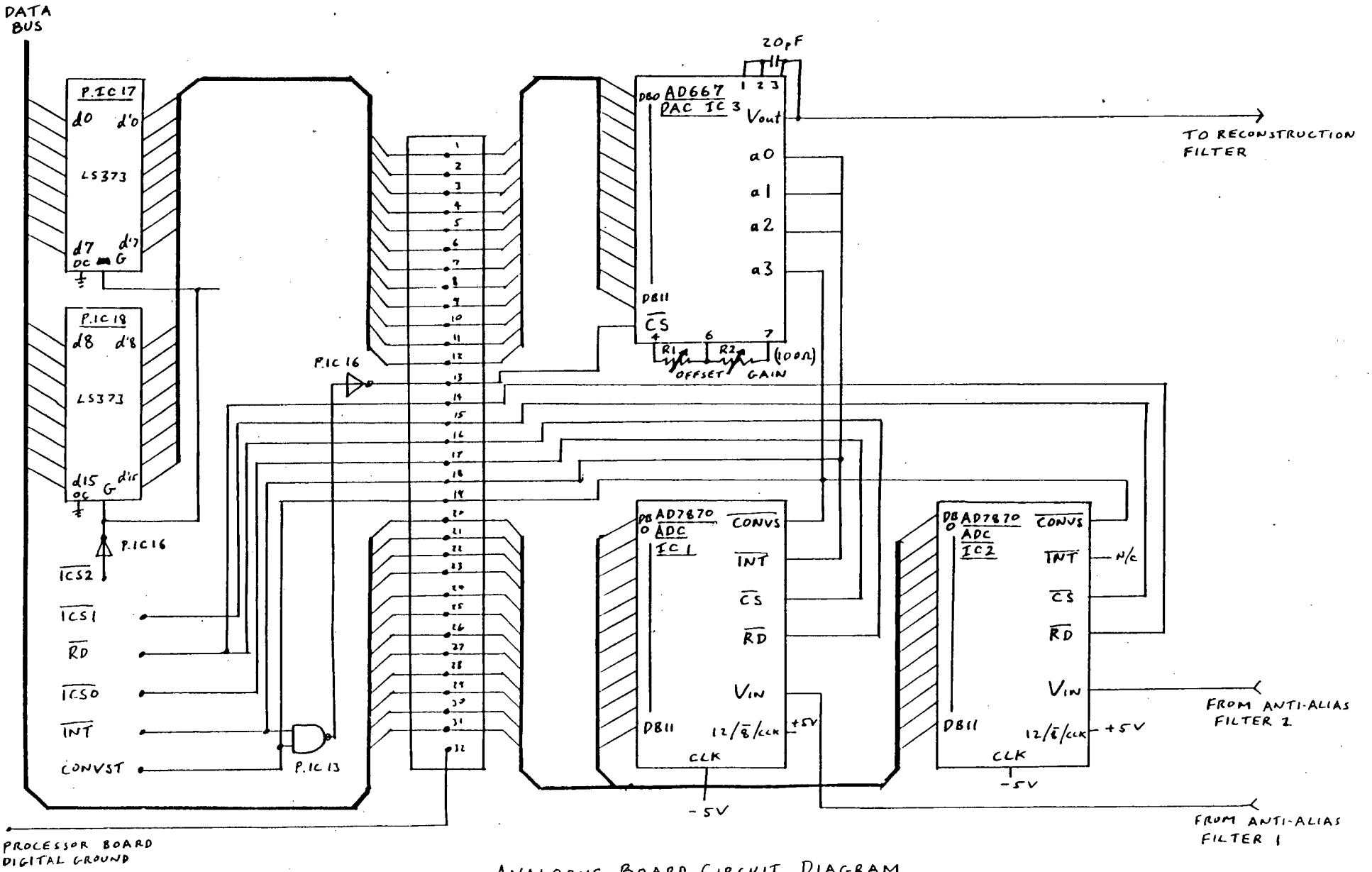


Processor Board Power Supply

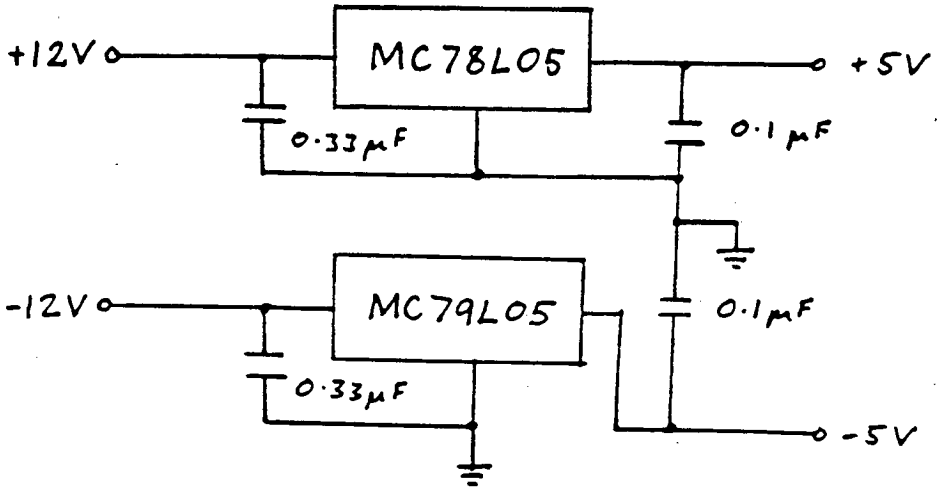
Analogue Board

The Analogue Board circuits are included on the following pages. IC numbers and part descriptions are listed below.

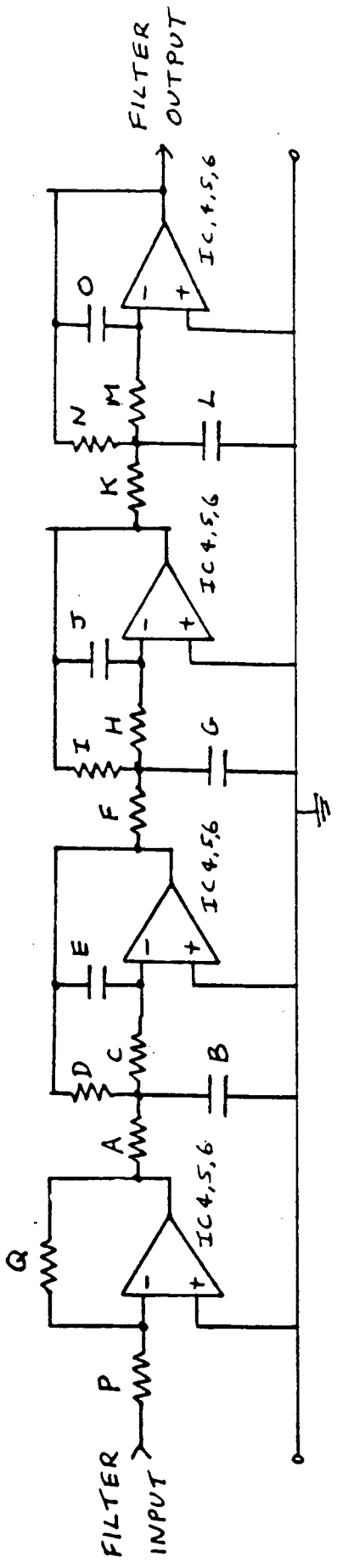
IC	Part	Description
1,2	AD7870	12-bit Analogue To Digital Converter
3	AD667	12-bit Digital To Analogue Converter
4,5,6	LS324	Quad Op-Amp



ANALOGUE BOARD CIRCUIT DIAGRAM



Analogue Board Power Supply

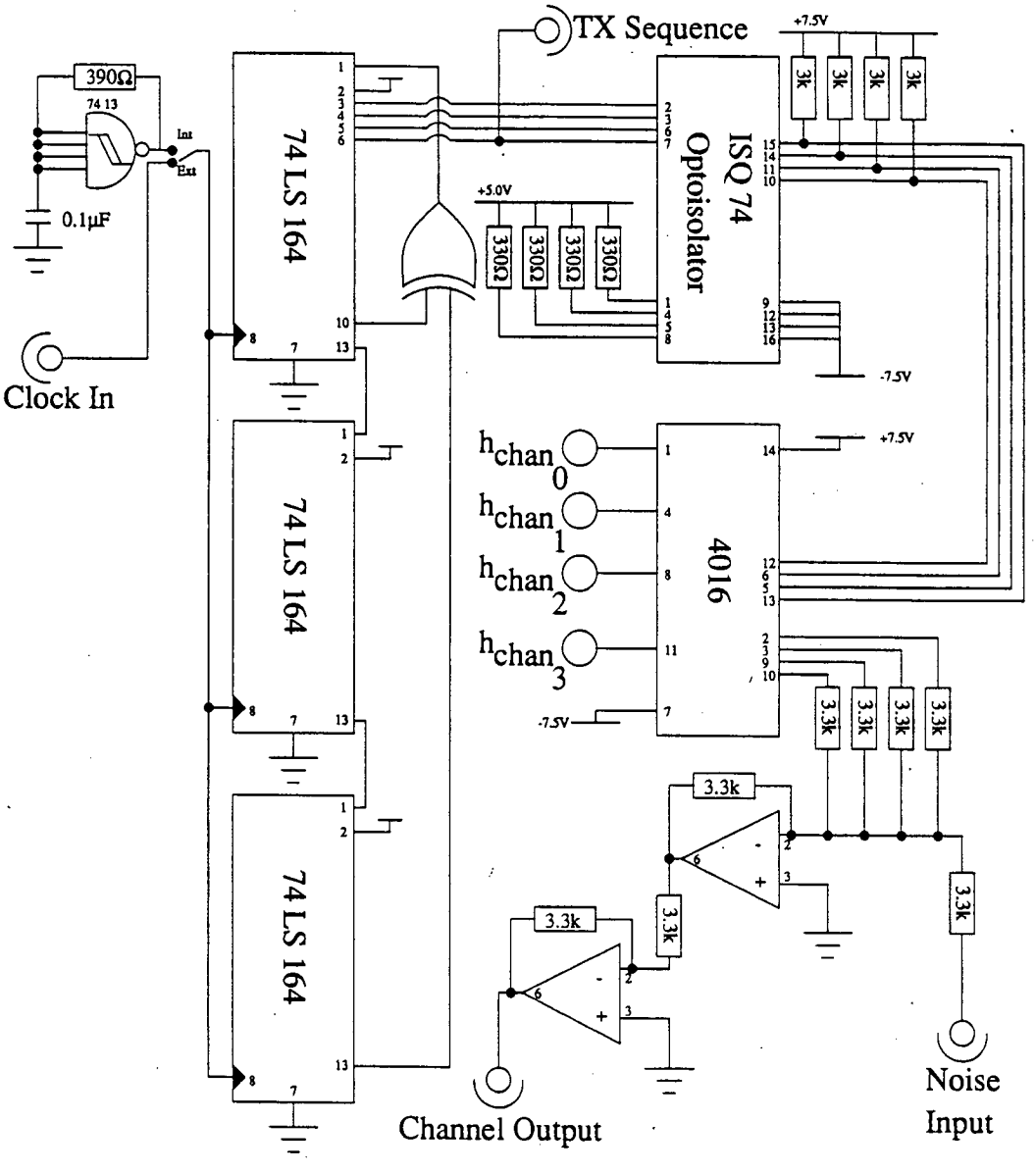


Analogue Board Filter Circuit

	1.7 Khz	4.7 Khz	7.7 Khz	10 Khz	12 Khz	16 Khz	20 Khz
A	2.588	2.588	2.588	2.588	2.588	2.588	2.588
B	.280	.101	.0617	.0475	.0396	.0297	.0238
C	1.294	1.294	1.294	1.294	1.294	1.294	1.294
D	2.588	2.588	2.588	2.588	2.588	2.588	2.588
E	.00936	.00339	.00207	.00160	.00133	.000995	.000796
F	7.071	7.071	7.071	7.071	7.071	7.071	7.071
G	.0375	.0136	.00827	.00637	.00531	.00398	.00318
H	3.536	3.536	3.536	3.536	3.536	3.536	3.536
I	7.071	7.071	7.071	7.071	7.071	7.071	7.071
J	.00936	.00339	.00207	.00160	.00133	.000995	.000796
K	9.659	9.659	9.659	9.659	9.659	9.659	9.659
L	.0201	.00726	.00443	.00341	.00284	.00213	.00171
M	4.830	4.830	4.830	4.830	4.830	4.830	4.830
N	9.659	9.659	9.659	9.659	9.659	9.659	9.659
O	.00936	.00339	.00207	.00160	.00133	.000995	.000796

Alternative Filter Component Values

Note: The unit for resistance is K Ohms.
The unit for capacitor is Micro Farads.
The above values are not industry standard values.
On-board filters are initially configured for 1.7 KHz.


TX Sequence
+7.5V
390Ω
74 13
Int
Ext
0.1µF
Clock In
74 LS 164
74 LS 164
74 LS 164
+5.0V
330Ω
330Ω
330Ω
330Ω
ISQ 74
Optoisolator
3k
3k
3k
3k
-7.5V
hchan0
hchan1
hchan2
hchan3
4016
+7.5V
-7.5V
3.3k
3.3k
3.3k
3.3k
3.3k
3.3k
3.3k
Channel Output
Noise
Input

Appendix E

VHDL source code for coprocessor design

multiply.vhdl

VHDL behavioral source code for interval multiplication

```
package mult_types is

        type mlong is range -2147483648 to 2147483647;
        subtype mshort is mlong range -32768 to 32767;
        subtype bitpos is mshort range 0 to 31;

end mult_types;

-------------------------------------------------
use work.mult_types.all;

package mult_functions is

        procedure interval_multiply(a : in mshort;
                                                                b : in mshort;
                                                                c : in mshort;
                                                                d : in mshort;
                                                                rl : out mlong;
                                                                ru : out mlong);

        function find_min(x1 : in mlong;
                                                        x2 : in mlong;
                                                        x3 : in mlong;
                                                        x4 : in mlong) return mlong;

        function find_max(x1 : in mlong;
                                                        x2 : in mlong;
                                                        x3 : in mlong;
                                                        x4 : in mlong) return mlong;

        function shift(x: in mlong;
                                                s: in bitpos) return mshort;

end mult_functions;
-------------------------------------------------
package body mult_functions is

                procedure interval_multiply(a : in mshort;
                        b : in mshort;
                        c : in mshort;
                        d : in mshort;
                        rl : out mlong;
                        ru : out mlong)

                is
                variable p1,p2,p3,p4:mlong:=0;

                begin
                        p1:=a*c;
                        p2:=a*d;
                        p3:=b*c;
                        p4:=b*d;
```

```
                rl:=find_min(p1,p2,p3,p4);
                ru:=find_max(p1,p2,p3,p4);
        end interval_multiply;


        function find_min(x1 : in mlong;
         x2 : in mlong;
         x3 : in mlong;
         x4 : in mlong) return mlong is

        variable m1,m2:mlong:=0;

        begin

        if (x1<x2) then m1:=x1; else m1:=x2; end if;
        if (x3<x4) then m2:=x3; else  m2:=x4; end if;
        if (m1<m2) then return(m1); else return(m2); end if;

        end find_min;

        function find_max(x1 : in mlong;
         x2 : in mlong;
         x3 : in mlong;
         x4 : in mlong) return mlong is

variable m1,m2:mlong:=0;

begin

if (x1>x2) then m1:=x1; else  m1:=x2; end if;
if (x3>x4) then m2:=x3; else  m2:=x4; end if;
if (m1>m2) then return(m1); else return(m2); end if;

        end find_max;

        function shift(x: in mlong;
                                        s: in bitpos) return mshort is
        variable pow2:mlong:=1;
        begin
                pow2:=1;
                powlp:
                for i in 1 to s loop
                        pow2:=2*pow2;
                end loop powlp;
                return(x/pow2);
        end shift;


end mult_functions;
```

multiply.vhdl

VHDL behavioral source code for interval multiplication

---

coprocessor.vhdl

VHDL behavioral descrition of coprocessor chip

```
use work.mult_functions.all;
use work.mult_types.all;

entity coprocessor is
port(address_bus: in mshort;
                rdata_bus:in mshort;
                wdata_bus:out mshort;
                r_w : in bit;
                cs : in bit;
           CLOCK: in bit);
end coprocessor;
-----------------------------------------------------------------
architecture chip of coprocessor is
begin
        process
        variable op_1reg : mshort:=0;
        variable op_2reg : mshort:=0;
        variable op_3reg : mshort:=0;
        variable op_4reg : mshort:=0;
        variable i_reg : mshort:=0;
        variable res_shift : bitpos:=0;
        variable res_0reg : mshort:=0;
        variable res_1reg : mshort:=0;
        variable outbuf : mshort:=0;
        variable inbuf : mshort:=0;
        variable adbuf : mshort:=0;
        variable low_accumulator: mlong:=0;
        variable high_accumulator: mlong:=0;
        variable h: mlong:=0;
        variable l: mlong:=0;

        begin


                inbuf:=rdata_bus;
                adbuf:=address_bus;


        if (r_w='1' and cs='1') then     --This part for read operations
        case adbuf is
                when 0 =>
                        op_1reg:=inbuf;
                when 1 =>
                        op_2reg:=inbuf;
                when 2 =>
                        op_3reg:=inbuf;
                when 3 =>
                        op_4reg:=inbuf;
                when 4 =>
                        if (inbuf=1) then
```

```
                        low_accumulator:=0;
                        high_accumulator:=0;
                end if;
                if (inbuf=2) then
                        interval_multiply(op_1reg,op_2reg,op_3reg,op_4reg,l,h);
                        low_accumulator:=low_accumulator+l;
                        high_accumulator:=high_accumulator+h;
                        res_0reg:=shift(low_accumulator,res_shift);
                        res_1reg:=shift(high_accumulator,res_shift);
                end if;
        when 5=>
                res_shift:=inbuf;
        when others=>

        end case;
end if;

if (r_w='0' and cs='1') then
case adbuf is
        when 0 =>
                outbuf:=res_0reg;
        when 1 =>
                outbuf:=res_1reg;
        when others=>

end case;
end if;

wdata_bus<=outbuf;


end process;


end chip;
```