INTERIOR NOISE REDUCTION BY ADAPTIVE FEEDBACK VIBRATION CONTROL

KU-FRL-1178-2

November 1998

Final Report
under
Grant number NAG-1-1735
Prepared for
NASA Langley Research Center
Hampton, VA

INTERIOR NOISE REDUCTION BY ADAPTIVE FEEDBACK VIBRATION CONTROL

KU-FRL-1178-2

November 1998

Prepared by: Tae W. Lim

Tae W. Lim (Principal Investigator)

Hadi Alhassani /TWL

Abdulhadi A. Alhassani (Graduate Research Assistant)

Approved by: _____

Richard J. Silcox (Technical Monitor, NASA)

Flight Research Laboratory
The University of Kansas Center for Research, Inc.
2291 Irving Hill Drive
Lawrence, KS 66045-2969

ABSTRACT

The objective of this project is to investigate the possible use of adaptive digital filtering techniques in simultaneous, multiple-mode identification of the modal parameters of a vibrating structure in real-time. It is intended that the results obtained from this project will be used for state estimation needed in adaptive structural acoustics control. The work done in this project is basically an extension of the work on real-time single mode identification, which was performed successfully using a digital signal processor (DSP) at NASA, Langley. Initially, in this investigation the single mode identification work was duplicated on a different processor, namely the Texas Instruments TMS320C40 DSP. The system identification results for the single mode case were very good. Then an algorithm for simultaneous two mode identification was developed and tested using analytical simulation. When it successfully performed the expected tasks, it was implemented in real-time on the DSP system to identify the first two modes of vibration of a cantilever aluminum beam. The results of the simultaneous two mode case were good but some problems were identified related to frequency warping and spurious mode identification.

The frequency warping problem was found to be due to the bilinear transformation used in the algorithm to convert the system transfer function from the continuous-time domain to the discrete-time domain. An alternative approach was developed to rectify the problem. The spurious mode identification problem was found to be associated with high sampling rates. Noise in the signal is suspected to be the cause of this problem but further investigation will be needed to clarify the cause.

For simultaneous identification of more than two modes, it was found that theoretically an adaptive digital filter can be designed to identify the required number of modes, but the algebra became very complex which made it impossible to implement in the DSP system used in this study.

The on-line identification algorithm developed in this research will be useful in constructing a state estimator for feedback vibration control.

Table of Contents

1 Introduction

This chapter details the background and driving force behind the use of adaptive filters in modal parameter estimation. It also gives an overview of the techniques used to achieve this objective and an outline of this document.

1.1 Background and Literature Survey

Active control of a structure that has time-varying plant dynamics requires accurate tracking of such variance, which could be the result of uncertainties in the structure response or noise. In airframes, such variance can also be induced by changes in altitude, speed, and temperature [1]. As a result, some form of on-line, real-time estimation of the modal parameters of the structure becomes essential.

Researchers at the U.S. Naval Center for Space Technologies at the U.S. Naval Research Laboratory have studied the use of adaptive filters and neural networks in developing accurate knowledge of the varying dynamics of large, lightweight, flexible space structures. Such knowledge becomes essential in environments where accurate attitude control and vibration suppression are needed in order to meet the stringent jitter and pointing requirements [2].

Modal parameter identification of linear, time-invariant systems has been an area of extensive research for the past decade. It has been mainly used for off-line applications such as the eigensystem realization algorithm, the poly reference technique, and the observer-Kalman filter identification algorithm. In addition, several recursive or on-line versions of the time-domain algorithms have also been studied for use in on-line identification of time-varying dynamic systems [3].

Adaptive filters have been used extensively in various signal processing applications such as echo cancellation, speech analysis, spectral estimation [4], removal of powerline hum from medical instruments, equalization of troposcatter communication signals [5] and cancellation of maternal heartbeat in fetal electrocardiography [4]. They are also the focus of major current studies that try to use adaptive filtering algorithms to achieve adaptive inverse control of unknown and time varying systems [4].

1.2 Scope of the work

Modal testing is a well established field that has applications in almost every engineering discipline. In most applications, frequency response functions (FRF's), which are functions in the frequency domain that relate the response output to a prescribed input to the structure under consideration, are estimated using various methods of exciting the structure, measuring the response and processing the measured data. Such tests are usually repeated many times to collect as many FRF's as possible. These are then processed and used to determine the modal parameters of the various modes of vibrations of the structure. Many standard tools and methods are practiced in industry to obtain such results. Although the ultimate goal of this study is the same as described above, namely modal parameter estimation, the methods employed and techniques used in this investigation are totally different.

In this project, the modal parameters of a structure are estimated in real-time using adaptive filtering techniques. This information will be used in the next phase of this project to do real-time state-estimation for active control.

An adaptive filter as the name implies is one that has the ability to adapt to and learn changes in the characteristics of a signal. In the control and dynamics discipline, they are mainly used for plant modeling and disturbance cancellation [4]. Particular to this project, they are used for system identification. Such a filter usually has weights that can be changed/updated through an adaptation algorithm to achieve some objective. In this study the adaptive filter is designed to perform system identification of multiple modes of vibration on a structure by means of minimizing the error between the filter output and the plant output, i.e. structure response. When that

happens, then the filter is said to have identified/modeled the plant and this on-line real-time system information can then be used in an active control setting.

1.3 Outline of the Report

The above section has provided an outline of the basic concept and reasoning behind this work. This section on the other hand provides an outline of the various chapters included in this document.

Chapter 2 Theoretical Development

This chapter details the mathematical development and representation of the structure and the adaptive filter. The process usually starts by modeling the structure using a continuous-time transfer function that relates the structure response to the excitation input. The pulsed transfer function of the structure model is obtained using some form of s to z transformation. The method used here is the bilinear transformation. Once the pulsed transfer function of the structure is available, then a discrete adaptive filter is built to perform the system identification. The filter weights are updated through an adaptation algorithm, the one used here is the Widrow-Hoff Delta Rule [4]. This development is done for both single and multiple mode identification models.

Chapter 3 Analytical Simulation

The simulation of the mathematical models developed in the previous chapter are presented here. MATLAB®, which is a high performance numeric computation and visualization software, is used to simulate the system, the adaptive filter and the training process. The structure is modeled using a spring-mass-damping system. Results and implementation problems are discussed along with possible reasons and solutions.

Chapter 4 C40 DSP System

The real-time testing of the adaptive filter and the identification process is implemented on the C40 Digital Signal Processing System. The purpose of this chapter is to familiarize the reader and any future users with such a system. This helps in understanding the real-time identification process and appreciating the problems associated with it.

Chapter 5 Real-Time System Identification

This chapter details the equipment used in the lab, the testing set up and results for the single and two mode real-time system identification. Discussions of problems associated with such testing and possible solutions are embedded in the chapter.

Chapter 6 Conclusions and Recommendations

The most significant conclusions of the project are summarized in this chapter along with recommendations for improvement and future applications.

Appendix A Simulation Code

The MATLAB code used for system simulation and algorithm testing is included in this appendix for both the single and two mode cases.

Appendix B Real-Time Single Mode Code

This appendix details the code used for the real-time system identification of the first mode. It contains the C40 code and the host PC code which are written in C. It also includes the DSPLINK register set up file, the network configuration file, an m-file to interface with MATLAB, the linker command file and an error return subroutine.

Appendix C Real-Time Two Mode Code

Similar to appendix B, but for the real-time simultaneous two mode case.

Appendix D Real-Time Two Mode Code - Filtered

Similar to appendix C, but for the two mode case with band-pass filtering.

2 Theoretical Development

This chapter details the mathematical representation of a vibrating structure and the system identification process using an adaptive filter.

2.1 Mathematical Representation of a Vibrating Structure

A single-input, single-output (SISO) multiple degree of freedom vibrating structure can be modeled with the following continuous transfer function assuming proportional viscous damping [1]:

$$\frac{Y_p(s)}{U_q(s)} = \sum_i \frac{{}_iA_{pq}s^2}{s^2 + 2\zeta_i\omega_i s + \omega_i^2} \qquad (2.1)$$

Where U_q is the excitation applied at point q of the structure and Y_p is the acceleration response at point p (hence the s^2 term in the numerator). The quantities ζ, ω and A are the equivalent viscous damping ratio, natural frequency and modal amplitude, respectively for the ith mode. The mode shape information is embedded in the modal amplitude. The modal amplitude is evaluated using the mode shape coefficients ${}_i\phi_p$ and ${}_i\phi_q$ as follows [6]:

$${}_iA_{pq} = {}_i\phi_p \; {}_i\phi_q \qquad (2.2)$$

The driving point measurement must be taken in order to identify the mode shape coefficients in equation (2.2). For example, consider the first mode of a structure. Assume that the excitation is applied at point 1 and the response is measured at point 2. To solve uniquely for the mode shape coefficients in equation (2.2) which are ${}_1\phi_1$ and ${}_1\phi_2$, equation (2.1) will have to be evaluated at the driving point, i.e. point 1. Thus, at least three sensors are needed (force transducer for input signal, driving point accelerometer and an accelerometer where the mode shape coefficient is to be identified) to fully identify the modal parameters of a mode unless only the mode shape coefficient at the driving point is to be identified. Since the data acquisition system used in this project is a dual-channel system, the luxury of using more than 2 sensors is not available. This limitation hinders the identification of mode shape coefficients other than at the driving point.

2.2 System Identification Process

A SISO dynamic system which has an input or excitation *u(t)* and an output or response *y(t)* as shown below can be thought of as an analog filter. If the input and output of this analog filter are synchronously sampled and used to train an adaptive digital filter driven by some adaptation algorithm, when the weights of the adaptive filter converge and the error is minimized, the adaptive filter transfer function becomes identical to the analog filter pulse transfer function [5]. In other words, applying the same input to both filters will produce the same output, which means that the adaptive filter has accurately identified the unknown dynamic system.

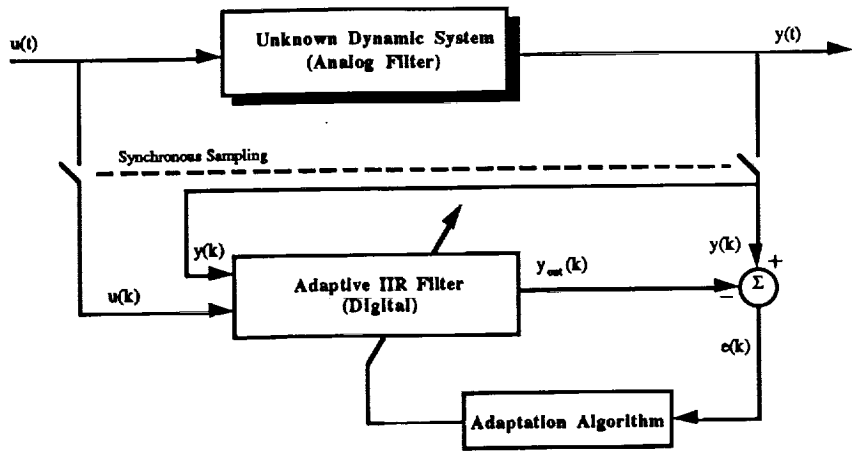

Figure 2.1: Identification of an Unknown Dynamic System Using an Adaptive Digital Filter [4]

2.2.1 The Dynamic System

In our case, the unknown system is a vibrating structure which has the transfer function shown by equation (2.1). In order to obtain the modal parameters of a specific mode, the adaptive filter coefficients have to be related to the modal parameters. This is done by equating the pulsed transfer functions of the filter and the system which means that the continuous transfer function of the structure has to be transformed to get the pulsed transfer function. This is done by applying the bilinear transformation (Tustin approximation):

$$s = 2f_s \frac{z-1}{z+1} \tag{2.3}$$

to equation (2.1), where f_s is the sampling frequency in Hz. We obtain:

$$\frac{Y_p(z)}{U_q(z)} = \sum_i \frac{{}_iA_{pq}\left(2f_s\frac{z-1}{z+1}\right)^2}{\left(2f_s\frac{z-1}{z+1}\right)^2 + 2\zeta_i\omega_i\left(2f_s\frac{z-1}{z+1}\right)+\omega_i^2} \tag{2.4}$$

expanding and collecting terms give:

$$\frac{Y_p(z)}{U_q(z)} = \sum_i \frac{4\,{}_iA_{pq}f_s^2(z^2-2z+1)}{(4f_s^2+4\zeta_i\omega_i f_s+\omega_i^2)z^2+(-8f_s^2+2\omega_i^2)z+(4f_s^2-4\zeta_i\omega_i f_s+\omega_i^2)}$$

$$\frac{Y_p(z)}{U_q(z)} = \sum_i \frac{\frac{4\,{}_iA_{pq}f_s^2}{4f_s^2+4\zeta_i\omega_i f_s+\omega_i^2}(z^2-2z+1)}{z^2+\frac{(-8f_s^2+2\omega_i^2)}{4f_s^2+4\zeta_i\omega_i f_s+\omega_i^2}z+\frac{(4f_s^2-4\zeta_i\omega_i f_s+\omega_i^2)}{4f_s^2+4\zeta_i\omega_i f_s+\omega_i^2}} \tag{2.5}$$

2.2.2 The Adaptive Filter

An adaptive filter is a digital filter that has variable parameters (weights) which can be modified in real-time to achieve a certain objective. While time-invariant, non-adaptive filters are also used to solve many signal processing problems, such filters require a good knowledge of the signal at hand, before they can be efficiently implemented. Uncertainties about the characteristics of a signal or its variation with time can provide a real challenge to a standard filter. This is basically where adaptive filters come into play, their ability to track down changes and learn the characteristics of the signal in real-time makes them a powerful tool in many applications. As a matter of fact, they have been used for well over a decade to solve such problems as echo and noise cancellation, channel equalization, speech analysis and a host of other signal processing applications [4,5].

In this project, adaptive filters are used to identify the modal parameters of a vibrating structure as shown in Figure 2.1. The most common types of adaptive filters used are finite impulse response (FIR) and infinite impulse response (IIR) filters. The application at hand usually determines the type of filter to be used. In this application, an IIR filter is used which has poles and zeros and could model the dynamic system under consideration. In all cases, an adaptive filter has a standard input and output and an error input that is used to update the weights through an adaptation algorithm. There are many adaptation algorithms that can be used with a variety of filters, the one used here is called the α-LMS or Widrow-Hoff delta rule [4]. It is discussed in greater details below.

As mentioned earlier, the unknown dynamic system has poles and zeros and is thus better approximated by an IIR filter. Such a digital filter has the following system pulsed transfer function [7]:

$$H(z) = \frac{Y_{out}(z)}{U(z)} = \frac{\sum_{n=0}^{M} B_n z^{-n}}{1 - \sum_{n=1}^{K} A_n z^{-n}}$$

expanding:

$$\frac{Y_{out}(z)}{U(z)} = \frac{B_0 + B_1 z^{-1} + B_2 z^{-2} + B_3 z^{-3} + B_4 z^{-4} + \cdots\cdots}{1 - A_1 z^{-1} - A_2 z^{-2} - A_3 z^{-3} - A_4 z^{-4} - \cdots\cdots}$$

By dividing numerator and denominator by the highest negative power of z, we can rewrite the above equation as:

$$\frac{Y_{out}(z)}{U(z)} = \frac{B_0 z^4 + B_1 z^3 + B_2 z^2 + B_3 z + B_4 + \cdots\cdots}{z^4 - A_1 z^3 - A_2 z^2 - A_3 z - A_4 - \cdots\cdots}$$

which can be factorized into multiple second order terms that can be expressed in the format given in equation (2.5):

$$\frac{Y_{out}(z)}{U(z)} = \sum_i \frac{b_i (z^2 - 2z + 1)}{z^2 + a_{i1} z + a_{i2}} \qquad (2.6)$$

In general, the adaptive filter can be written as a difference equation:

$$y_{out}(k) = \sum_{n=1}^{K} a_n y_{out}(k-n) + \sum_{n=0}^{M} b_n u(k-n)$$

Such an equation can be easily implemented in a code because it consists of current and past values of the input and output signals multiplied by constant coefficients [7].

2.2.3 The Adaptive Linear Combiner

The implementation of the adaptive filter described above is done by modeling it as an adaptive linear combiner (LC). The basic tenet of an adaptive LC is learning. It processes information and learns through a certain algorithm to achieve a prescribed goal. In our case, this goal is to make the output signal of the filter as close as possible to the response of the structure. The adaptive LC takes in each adaptation cycle an input vector P(k) = [P1(k), P2(k), P3(k), ... Pn(k)], multiplies each input by a weight and takes the sum of all weighted inputs and compares it to a desired output $y_d(k)$. The error, e(k) which is the difference between the adaptive LC output and the desired output (structural response) is used to update the weights. The process continues until weights converge and error is minimized.

The weights are updated in this case by the α-LMS or Widrow-Hoff Rule [4]:

$$W(k+1) = W(k) + \alpha e(k) \frac{P(k)}{P^T(k)P(k)} \qquad (2.7)$$

where α is the learning rate, which can be thought of as a step size. It is basically a factor that determines how fast the weights are updated. It has the recommended range of [4]

$$0.1 < \alpha < 1.0$$

It will be shown later that an even smaller learning rate is sometimes needed to get convergence. If training takes place and the error is minimized, the pulse transfer function of the unknown system, equation (2.5) and that of the adaptive filter, equation (2.6) are identical. Equating coefficients relates the filter parameters and the modal parameters of the structure:

$$b_i = \frac{4_i A_{pq} f_s^2}{4f_s^2 + 4\zeta_i \omega_i f_s + \omega_i^2}$$

$$a_{i1} = \frac{-8f_s^2 + 2\omega_i^2}{4f_s^2 + 4\zeta_i \omega_i f_s + \omega_i^2}$$

$$a_{i2} = \frac{4f_s^2 - 4\zeta_i \omega_i f_s + \omega_i^2}{4f_s^2 + 4\zeta_i \omega_i f_s + \omega_i^2}$$

Thus, the modal parameters are defined in terms of the filter coefficients as:

$$\omega_i = 2f_s \sqrt{\frac{1 + a_{i1} + a_{i2}}{1 - a_{i1} + a_{i2}}} \qquad (2.8)$$

$$\zeta_i = \frac{2f_s(1-a_{i2})}{\omega_i(1-a_{i1}+a_{i2})} \tag{2.9}$$

$${}_iA_{pq} = \frac{4b_i}{1-a_{i1}+a_{i2}} \tag{2.10}$$

Although the mathematical development described above is for a single mode case, it can be extended for multiple mode cases. The following sections describe in detail the modal parameter identification procedures for one, two and larger number of modes.

2.2.4 Case I: Identification of Single Mode

The development of this case is based on work done by Lim, Cabell and Silcox [1].
For a collocated sensor and actuator case, the pulse transfer function of the structure for a single mode from equation (2.5) is:

$$\frac{Y(z)}{U(z)} = \frac{\dfrac{4\phi^2 f_s^2}{4f_s^2+4\zeta\omega f_s+\omega^2}(z^2-2z+1)}{z^2+\dfrac{(-8f_s^2+2\omega^2)}{4f_s^2+4\zeta\omega f_s+\omega^2}z+\dfrac{(4f_s^2-4\zeta\omega f_s+\omega^2)}{4f_s^2+4\zeta\omega f_s+\omega^2}} \tag{2.11}$$

Similarly, the pulse transfer function of the adaptive filter from equation (2.6) becomes for a single mode:

$$\frac{Y_{out}(z)}{U(z)} = \frac{b(z^2-2z+1)}{z^2+a_1z+a_2} \tag{2.12}$$

The above equation can be written in a difference equation format as:

$$y_{out}(k) = -a_1 y_{out}(k-1) - a_2 y_{out}(k-2) + b[u(k)-2u(k-1)+u(k-2)] \tag{2.13}$$

Since this filter is used to identify the system of equation (2.11), its output delay variables, $y_{out}(k-1)$ and $y_{out}(k-2)$ are replaced by the system response delay variables, $y(k-1)$ and $y(k-2)$, which are available through measurement. As a result, equation (2.13) becomes:

$$y_{out}(k) = -a_1 y(k-1) - a_2 y(k-2) + b[u(k)-2u(k-1)+u(k-2)] \tag{2.14}$$

which says that the adaptive filter's output is a function of the system excitation and response delay variables.

Implementation of the filter in equation (2.14) as an adaptive LC is as follows:

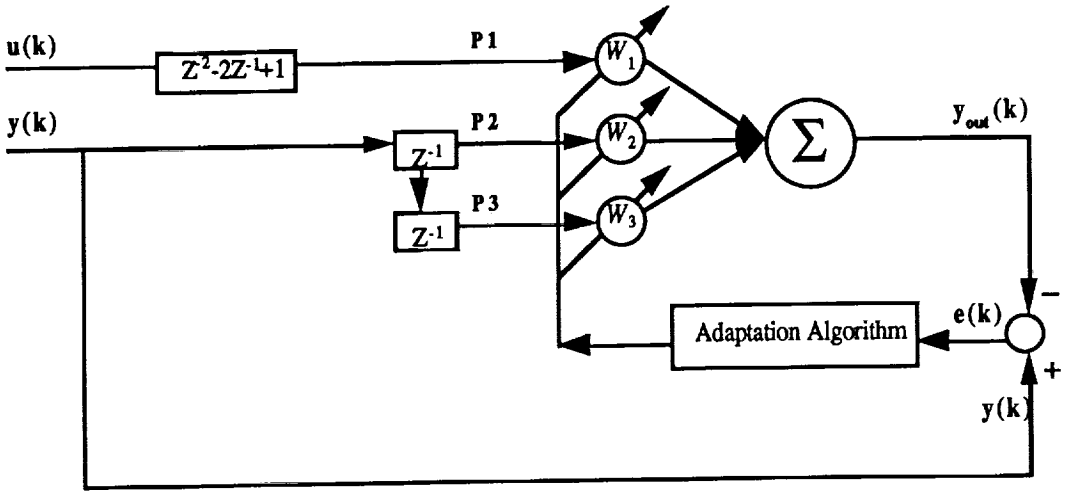

Figure 2.2: Adaptive Linear Combiner to Identify the Modal Parameters of a Single Mode

where:

$$\begin{aligned} P1 &= u(k) - 2u(k-1) + u(k-2) \quad \Rightarrow \quad W_1 = b \\ P2 &= y(k-1) \quad \Rightarrow \quad W_2 = -a_1 \\ P3 &= y(k-2) \quad \Rightarrow \quad W_3 = -a_2 \end{aligned} \tag{2.15}$$

Once the filter coefficients are identified, the modal parameters are calculated using equations (2.8), (2.9) and (2.10). The simulation and real-time implementation of this network are detailed in Chapters 3 and 5, respectively.

2.2.5 Case II: Simultaneous Identification of Two Modes

The pulse transfer function of the first two modes of the structure is obtained from equation (2.5) for I=2 are:

$$\frac{Y(z)}{U(z)} = \frac{\dfrac{4_1 A_{pq} f_s^2}{4f_s^2 + 4\zeta_1\omega_1 f_s + \omega_1^2}(z^2 - 2z + 1)}{z^2 + \dfrac{(-8f_s^2 + 2\omega_1^2)}{4f_s^2 + 4\zeta_1\omega_1 f_s + \omega_1^2} z + \dfrac{(4f_s^2 - 4\zeta_1\omega_1 f_s + \omega_1^2)}{4f_s^2 + 4\zeta_1\omega_1 f_s + \omega_1^2}} + \frac{\dfrac{4_2 A_{pq} f_s^2}{4f_s^2 + 4\zeta_2\omega_2 f_s + \omega_2^2}(z^2 - 2z + 1)}{z^2 + \dfrac{(-8f_s^2 + 2\omega_2^2)}{4f_s^2 + 4\zeta_2\omega_2 f_s + \omega_2^2} z + \dfrac{(4f_s^2 - 4\zeta_2\omega_2 f_s + \omega_2^2)}{4f_s^2 + 4\zeta_2\omega_2 f_s + \omega_2^2}}$$

The pulse transfer function of the adaptive filter is obtained from equation (2.6) as:

$$\frac{Y_{out}(z)}{U(z)} = \frac{b_1(z^2 - 2z + 1)}{z^2 + a_{11}z + a_{12}} + \frac{b_2(z^2 - 2z + 1)}{z^2 + a_{21}z + a_{22}}$$

Combine the two terms on the right hand side to obtain:

$$\frac{Y_{out}(z)}{U(z)} = (z^2 - 2z + 1)\left[\frac{b_1\left(z^2 + a_{21}z + a_{22}\right) + b_2\left(z^2 + a_{11}z + a_{12}\right)}{z^4 + (a_{11}+a_{21})z^3 + (a_{12}+a_{21}a_{11}+a_{22})z^2 + (a_{21}a_{12}+a_{22}a_{11})z + a_{22}a_{12}}\right]$$

Define: $$V(z) = U(z)\left[z^2 - 2z + 1\right] \tag{2.16}$$

and rearrange to arrive at:

$$\frac{Y_{out}(z)}{V(z)} = \frac{(b_1 + b_2)z^2 + (a_{21}b_1 + a_{11}b_2)z + (b_1a_{22} + b_2a_{12})}{z^4 + (a_{11}+a_{21})z^3 + (a_{12}+a_{21}a_{11}+a_{22})z^2 + (a_{21}a_{12}+a_{22}a_{11})z + a_{22}a_{12}}$$

The corresponding difference equation becomes:

$$\begin{aligned} y_{out}(k) = &-(a_{11}+a_{21})y_{out}(k-1) - (a_{12}+a_{21}a_{11}+a_{22})y_{out}(k-2) \\ &- (a_{21}a_{12}+a_{22}a_{11})y_{out}(k-3) - (a_{22}a_{12})y_{out}(k-4) \\ &+ (b_1+b_2)v(k-2) + (a_{21}b_1 + a_{11}b_2)v(k-3) \\ &+ (b_1a_{22} + b_2a_{12})v(k-4) \end{aligned} \tag{2.17}$$

Similar to the single mode case, the adaptive filter's delay variables are replaced by measured structural response variables:

$$\begin{aligned} y_{out}(k) = &-(a_{11}+a_{21})y(k-1) - (a_{12}+a_{21}a_{11}+a_{22})y(k-2) \\ &- (a_{21}a_{12}+a_{22}a_{11})y(k-3) - (a_{22}a_{12})y(k-4) + (b_1+b_2)v(k-2) \\ &+ (a_{21}b_1 + a_{11}b_2)v(k-3) + (b_1a_{22} + b_2a_{12})v(k-4) \end{aligned}$$

where *v(k-2)* is obtained by dividing both sides of equation (2.16) by z^2 and casting in the difference equation form:

$$v(k-2) = u(k) - 2u(k-1) + u(k-2) \tag{2.18}$$

The adaptive filter difference equation above indicates that an adaptive LC with 7 inputs, 1 output and 7 weights is needed as shown below:

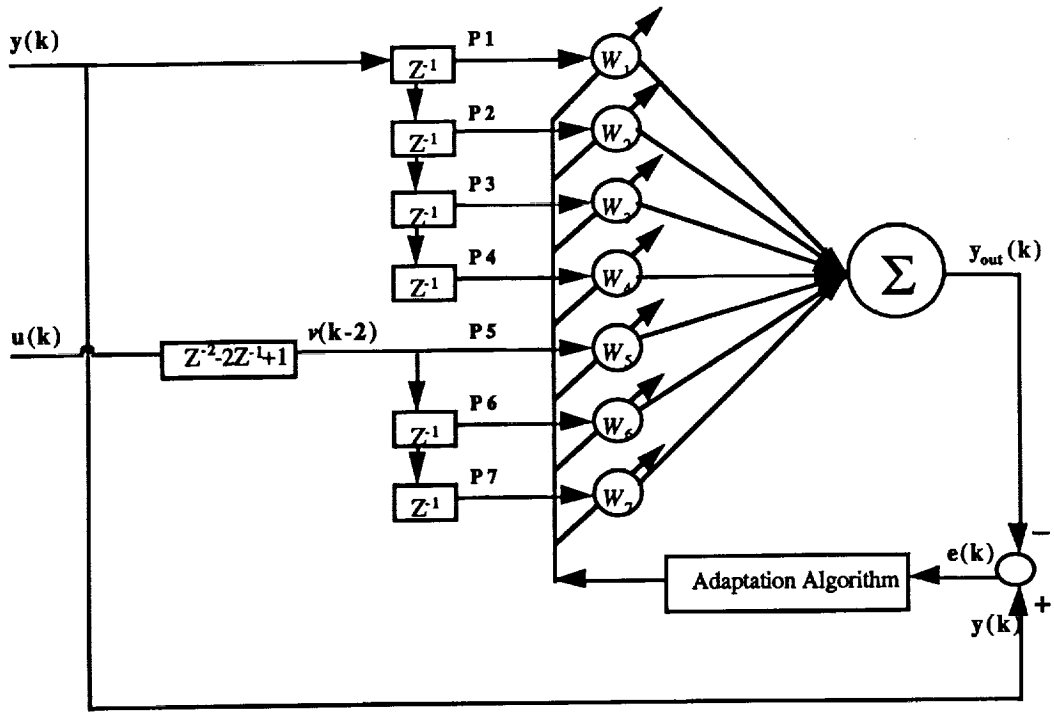

Figure 2.3: Adaptive Linear Combiner to Identify the Modal Parameters of Two Modes

where:

$$
\begin{aligned}
W_1 &= -(a_{11}+a_{21}) && \Rightarrow && P1 = y(k-1)\\
W_2 &= -(a_{12}+a_{21}a_{11}+a_{22}) && \Rightarrow && P2 = y(k-2)\\
W_3 &= -(a_{21}a_{12}+a_{22}a_{11}) && \Rightarrow && P3 = y(k-3)\\
W_4 &= -(a_{22}a_{12}) && \Rightarrow && P4 = y(k-4)\\
W_5 &= +(b_1+b_2) && \Rightarrow && P5 = v(k-2)\\
W_6 &= +(a_{21}b_1+a_1b_2) && \Rightarrow && P6 = v(k-3)\\
W_7 &= +(b_1a_{22}+b_2a_{12}) && \Rightarrow && P7 = v(k-4)
\end{aligned}
\qquad (2.19)
$$

When the weights converge, the filter coefficients can be extracted by solving the seven nonlinear equations above simultaneously. Since the first four equations are functions of four unknown coefficients only as written below, they can be solved simultaneously independent of the other three.

$$a_{11}+a_{21} = -W_1 \qquad (2.20)$$

$$a_{12}+a_{21}a_{11}+a_{22} = -W_2 \qquad (2.21)$$

$$a_{21}a_{12}+a_{22}a_{11} = -W_3 \qquad (2.22)$$

$$a_{22}a_{12} = -W_4 \qquad (2.23)$$

Substituting equations (2.20) and (2.23) into (2.22) gives:

$$-W_3 = a_{12}(-W_1 - a_{11}) + a_1 a_{22}$$

$$a_{11} = \frac{-W_3 + a_{12}W_1}{\left(\frac{-W_4}{a_{12}} - a_{12}\right)} \tag{2.24}$$

Substituting (2.23) into (2.24) and adding to (2.21) gives:

$$-W_2 a_{12} + W_3 a_{11} = a_{12}^2 - W_4 - a_{11}^2 a_{22} \tag{2.25}$$

Substituting (2.23) and (2.24) into (2.25) gives an equation in one variable only:

$$-W_2 a_{12} + W_3 \left[\frac{\frac{-W_3 + a_{12}W_1}{-W_4 - a_{12}^2}}{a_{12}}\right] = a_{12}^2 - W_4 + \left[\frac{\frac{-W_3 + a_{12}W_1}{-W_4 - a_{12}^2}}{a_{12}}\right]\frac{W_4}{a_{12}}$$

After rigorous manipulation, the above equation reduces to the following:

$$\begin{aligned} a_{12}^6 + W_2 a_{12}^5 + (W_1 W_3 + W_4) a_{12}^4 + (2W_2 W_4 - W_3^2 + W_1^2 W_4) a_{12}^3 \\ + (-W_1 W_3 W_4 - W_4^2) a_{12}^2 + (W_2 W_4^2) a_{12} - W_4^3 = 0 \end{aligned} \tag{2.26}$$

The above equation is a 6th order polynomial in one filter coefficient, namely a_{12}. The converged weights will always cause equation (2.26) to produce 2 real roots and 2 pairs of complex conjugates. Either real root produces the solution needed for the modal parameter identification of the two modes. The other coefficients of the filter are solved by substituting back the value of a_{12} into the weight equations:

$$\begin{aligned} a_{22} &= -\frac{W_4}{a_{12}} \\ a_{21} &= \frac{a_{22}W_1 - W_3}{a_{12} - a_{22}} \\ a_{11} &= -W_1 - a_{12} \\ b_2 &= \frac{a_{22}W_5 - W_7}{(a_{22} - a_{12})} \\ b_1 &= W_5 - b_2 \end{aligned} \tag{2.27}$$

Once the filter coefficients are determined, then the modal parameters can be easily calculated using equations (2.8), (2.9) and (2.10). The simulation and real-time implementation of this network are detailed in Chapters 3 and 5, respectively.

2.2.6 Case III: Identification of Three Modes

Similar to the previous 2 cases, the pulse transfer function of the first three modes of the structure is obtained from equation (2.5) for I=3:

$$\frac{Y(z)}{U(z)} = \frac{\dfrac{4_1 A_{pq} f_s^2}{4f_s^2 + 4\zeta_1\omega_1 f_s + \omega_1^2}(z^2 - 2z + 1)}{z^2 + \dfrac{(-8f_s^2 + 2\omega_1^2)}{4f_s^2 + 4\zeta_1\omega_1 f_s + \omega_1^2} z + \dfrac{(4f_s^2 - 4\zeta_1\omega_1 f_s + \omega_1^2)}{4f_s^2 + 4\zeta_1\omega_1 f_s + \omega_1^2}} +$$

$$\frac{\dfrac{4_2 A_{pq} f_s^2}{4f_s^2 + 4\zeta_2\omega_2 f_s + \omega_2^2}(z^2 - 2z + 1)}{z^2 + \dfrac{(-8f_s^2 + 2\omega_2^2)}{4f_s^2 + 4\zeta_2\omega_2 f_s + \omega_2^2} z + \dfrac{(4f_s^2 - 4\zeta_2\omega_2 f_s + \omega_2^2)}{4f_s^2 + 4\zeta_2\omega_2 f_s + \omega_2^2}}$$

$$\frac{\dfrac{4_3 A_{pq} f_s^2}{4f_s^2 + 4\zeta_3\omega_3 f_s + \omega_3^2}(z^2 - 2z + 1)}{z^2 + \dfrac{(-8f_s^2 + 2\omega_3^2)}{4f_s^2 + 4\zeta_3\omega_3 f_s + \omega_3^2} z + \dfrac{(4f_s^2 - 4\zeta_3\omega_3 f_s + \omega_3^2)}{4f_s^2 + 4\zeta_3\omega_3 f_s + \omega_3^2}}$$

Using equations (2.6) and (2.16), the pulse transfer function of the adaptive filter is written as:

$$\frac{Y_{out}(z)}{V(z)} = \frac{b_1}{z^2 + a_{11}z + a_{12}} + \frac{b_2}{z^2 + a_{21}z + a_{22}} + \frac{b_3}{z^2 + a_{31}z + a_{32}}$$

Combine the terms to obtain the common denominator:

$$\begin{aligned} Den(z) = z^6 &+ (a_{11} + a_{21} + a_{31})z^5 + (a_1 a_{21} + a_{11}a_{31} + a_{21}a_{31} + a_{12} + a_{22} + a_{32})z^4 \\ &+ (a_2 a_{12} + a_{11}a_{22} + a_3 a_{12} + a_{31}a_{21}a_{11} + a_{31}a_{22} + a_{32}a_{11} + a_{32}a_{21})z^3 \\ &+ (a_{22}a_{12} + a_{31}a_{21}a_{12} + a_{31}a_{22}a_{11} + a_{32}a_{12} + a_{32}a_2 a_{11} + a_3 a_{22})z^2 \\ &+ (a_3 a_{22}a_{12} + a_{32}a_{21}a_{12} + a_{32}a_{22}a_{11})z + a_{12}a_{22}a_{32} \end{aligned}$$

and the numerator:

$$
\begin{aligned}
Num(z) = {} & (b_1 + b_2 + b_3)z^4 + (b_1a_{21} + b_1a_{31} + b_2a_{31} + b_2a_{11} + b_3a_{11} + b_3a_{21})z^3 \\
& + (b_1a_{22} + b_1a_{21}a_{31} + b_1a_{32} + b_2a_{12} + b_2a_{11}a_{31} + b_2a_{32} + b_3a_{12} + b_3a_{21}a_{11} + b_3a_{22})z^2 \\
& + (b_1a_{21}a_{32} + b_1a_{22}a_{31} + b_2a_{11}a_{32} + b_2a_{12}a_{31} + b_3a_{12}a_{21} + b_3a_{11}a_{22})z \\
& + (b_3a_{22}a_{12} + b_1a_{22}a_{32} + b_2a_{12}a_{32})
\end{aligned}
$$

Thus, we have a difference equation as:

$$
\begin{aligned}
y_{out}(k) = {} & -(a_{11} + a_{21} + a_{31})\, y(k-1) \\
& -(a_{11}a_{21} + a_{11}a_{31} + a_{21}a_{31} + a_{12} + a_{22} + a_{32})\, y(k-2) \\
& -(a_{21}a_{12} + a_{11}a_{22} + a_{31}a_{12} + a_{31}a_{21}a_{11} + a_{31}a_{22} + a_{32}a_{11} + a_{32}a_{21})\, y(k-3) \\
& -(a_{22}a_{12} + a_{31}a_{21}a_{12} + a_{31}a_{22}a_{11} + a_{32}a_{12} + a_{32}a_{21}a_{11} + a_{32}a_{22})y(k-4) \\
& -(a_{31}a_{22}a_{12} + a_{32}a_{21}a_{12} + a_{32}a_{22}a_{11})\, y(k-5) - (a_{12}a_{22}a_{32})y(k-6) \\
& +(b_1 + b_2 + b_3)v(k-2) \\
& +(b_1a_{21} + b_1a_{31} + b_2a_{31} + b_2a_{11} + b_3a_{11} + b_3a_{21})v(k-3) \\
& +(b_1a_{22} + b_1a_{21}a_{31} + b_1a_{32} + b_2a_{12} + b_2a_{11}a_{31} + b_2a_{32} + \\
& \qquad + b_3a_{12} + b_3a_{21}a_{11} + b_3a_{22})v(k-4) \\
& +(b_1a_{21}a_{32} + b_1a_{22}a_{31} + b_2a_{11}a_{32} + b_2a_{12}a_{31} + b_3a_{12}a_{21} + b_3a_{11}a_{22})v(k-5) \\
& +(b_3a_{22}a_{12} + b_1a_{22}a_{32} + b_2a_{12}a_{32})v(k-6)
\end{aligned}
\qquad (2.28)
$$

This difference equation indicates that an adaptive LC with 11 inputs, 11 weights and 1 output is needed. The adaptive LC diagram will be similar to that in Figure 2.3 but with the more inputs and weights where the weights are given by:

$$
\begin{aligned}
W1 &= -(a_{11} + a_{21} + a_{31}) \\
W2 &= -(a_{11}a_{21} + a_{11}a_{31} + a_{21}a_{31} + a_{12} + a_{22} + a_{32}) \\
W3 &= -(a_{21}a_{12} + a_{11}a_{22} + \cdots\cdots + a_{31}a_{22} + a_{32}a_{11} + a_{32}a_{21}) \\
W4 &= -(a_{22}a_{12} + a_{31}a_{21}a_{12} + a_{31}a_{22}a_{11} + a_{32}a_{12} + a_{32}a_{21}a_{11} + a_{32}a_{22}) \\
W5 &= -(a_{31}a_{22}a_{12} + a_{32}a_{21}a_{12} + a_{32}a_{22}a_{11}) \\
W6 &= -(a_{12}a_{22}a_{32}) \\
W7 &= +(b_1 + b_2 + b_3) \\
W8 &= +(b_1a_{21} + b_1a_{31} + b_2a_{31} + b_2a_{11} + b_3a_{11} + b_3a_{21}) \\
W9 &= +(b_1a_{22} + b_1a_{21}a_{31} + \cdots\cdots + b_2a_{32} + b_3a_{12} + b_3a_{21}a_{11} + b_3a_{22}) \\
W10 &= +(b_1a_{21}a_{32} + b_1a_{22}a_{31} + \cdots\cdots + b_3a_{12}a_{21} + b_3a_{11}a_{22}) \\
W11 &= +(b_3a_{22}a_{12} + b_1a_{22}a_{32} + b_2a_{12}a_{32})
\end{aligned}
\qquad (2.29)
$$

and the adaptive LC inputs are:

$$
\begin{aligned}
P1 &= y(k-1) \\
P2 &= y(k-2) \\
P3 &= y(k-3) \\
p4 &= y(k-4) \\
P5 &= y(k-5) \\
P6 &= y(k-6) \\
P7 &= v(k-2) \\
P8 &= v(k-3) \\
P9 &= v(k-4) \\
P10 &= v(k-5) \\
P11 &= v(k-6)
\end{aligned}
\qquad (2.30)
$$

It is evident that at least 6 nonlinear equations need to be solved simultaneously to obtain the filter coefficients. This has proven to be a formidable task. Manual manipulation of these equations in a way similar to the two mode case is almost impossible. A symbolic math manipulator (MAPLE®) on the University of Kansas LARK system was used to tackle these equations. However, the program was unable to produce answers even after running for more than 24 hours. It can safely be concluded that MAPLE is incapable of solving such a system of equations. In a conversation the author had with Dr. Ralph Byers of the KU mathematics department who is an expert on numerical analysis, he recommended that an optimization algorithm be used. In addition, he indicated that obtaining an answer fast enough for this algorithm to still be implemented in real-time is doubtful. As a result, the real-time implementation of the adaptive filter for more than two degrees of freedom systems will not be attempted. An alternative method will have to be considered.

3 Analytical Simulation

The algorithms and mathematical models developed in chapter 2 are simulated and tested in this chapter. This analytical work is done using MATLAB® which provides an excellent platform for simulation. It also provides an excellent way of testing DSP codes that are later adjusted and implemented on the C40 DSP system. This is the reason why the MATLAB codes in Appendix A are written in the long format that closely resemble the C40 code. The vibrating structure is modeled as a spring-mass-damper system connected in series.

3.1 Single DOF Case

The MATLAB simulation program for the SDOF case described in this section is included in Appendix A. Consider a spring-mass-damper system shown in Figure 3.1. The dynamics of the system is represented by the following equation of motion:

$$m\ddot{x}(t)+c\dot{x}(t)+kx(t)=F(t)$$

where m is mass, c damping coefficient, k spring constant, $x(t)$ displacement response, and $F(t)$ the excitation applied to the system.

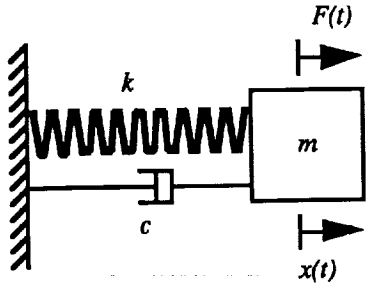

Figure 3.1: Single DOF Spring-Mass-Damper System

The modal parameters of such a system are chosen to be similar to those of the aluminum beam to be tested in the lab:

natural frequency:	$\omega = 11$ Hz,
damping ratio:	$\zeta = 0.01$,
modal amplitude:	$A_{pq} = 1$

where p is the response measurement point and q is the excitation point. In the single DOF case, p and q are the same point.

The system is implemented in MATLAB in the continuous transfer function format of equation (2.1):

$$\frac{Y_p(s)}{U_q(s)}=\frac{A_{pq}\,s^2}{s^2+2\zeta\omega\,s+\omega^2}$$

where $U_q(s)$ is a normally distributed random excitation applied to the system, and $Y_p(s)$ is the system acceleration response. The system is converted to discrete time (pulse transfer function) using the continuous-to-discrete command c2dm with the bilinear transformation option tustin in

MATLAB at a specific sampling rate. Once the pulse transfer function is determined, the system response to the random excitation is simulated using the discrete linear simulation command dlsim. Figure 3.2 shows the time histories of a sample excitation and response signals and the FFT of the system response which clearly shows the single mode.

The system excitation and response are then used to form the adaptive linear combiner inputs P1, P2 and P3. Off-line values of the weights can be obtained by solving the following linear equation which relates the output of the adaptive LC to its inputs:

$$\mathbf{Y}_{out}(k) = \mathbf{P}(k)\mathbf{W}(k) \tag{3.1}$$

where: $\mathbf{Y}_{out}(k) = \begin{bmatrix} y_{out}(1) \\ y_{out}(2) \\ \vdots \\ y_{out}(k) \end{bmatrix}$, $\mathbf{P}(k) = \begin{bmatrix} P1(1) & P2(1) & P3(1) \\ P1(2) & P2(2) & P3(2) \\ P1(3) & P2(3) & P3(3) \\ \vdots & \vdots & \vdots \\ P1(k) & P2(k) & P3(k) \end{bmatrix}$ and $\mathbf{W}(k) = \begin{bmatrix} W_1(k) \\ W_2(k) \\ W_3(k) \end{bmatrix}$

By solving for the weights, we obtain

$$\mathbf{W}(k) = \mathbf{P}^{*}(k)\mathbf{Y}_{out}(k)$$

where $\mathbf{P}^{*}$ is a pseudo-inverse defined by

$$\mathbf{P}^{*} = (\mathbf{P}^{T}\mathbf{P})^{-1}\mathbf{P}^{T}$$

The computation is done in MATLAB using the pinv command. It is based on singular value decomposition.

In order to identify the off-line weights, the output of the adaptive LC, $\mathbf{Y}_{out}(k)$, is replaced by the system response vector $\mathbf{Y}(k)$, which is available through measurements. These off-line values are used to check the on-line weights which should converge to the exact off-line values provided that the system is time-invariant. Thus, the 3x1 off-line weights vector is obtained by

$$\mathbf{W}_{off} = \mathbf{P}^{*}(k)\mathbf{Y}(k)$$

Note that no training is taking place and, as a result, the weights are not updated every time step. These in turn are used to determine the off-line modal parameters of the system. This off-line test provides a way of determining the sanity of the system response obtained through simulation or actual data acquisition. Possible problems could include inaccurate simulation, bad sampling, low signal-to-noise ratio, damaged sensors, etc. In addition, the off-line method provides a way of testing the adaptive LC architecture and helps find errors before the real-time implementation of the identification algorithm. It is hard to determine sources of error once the adaptation process is started because there are many variables involved. But by eliminating problems associated with the adaptation process, one can easily track down sources of error.

If the spring-mass-damper system was chosen to have the modal parameters described previously, and the response signal was sampled at 32 Hz, then the off-line weights obtained using the above procedure are

$$\begin{aligned} W_{1off} &= 0.4571 \\ W_{2off} &= -0.152 \\ W_{3off} &= -0.903 \end{aligned}$$

The MATLAB code for this case is given in Appendix A.

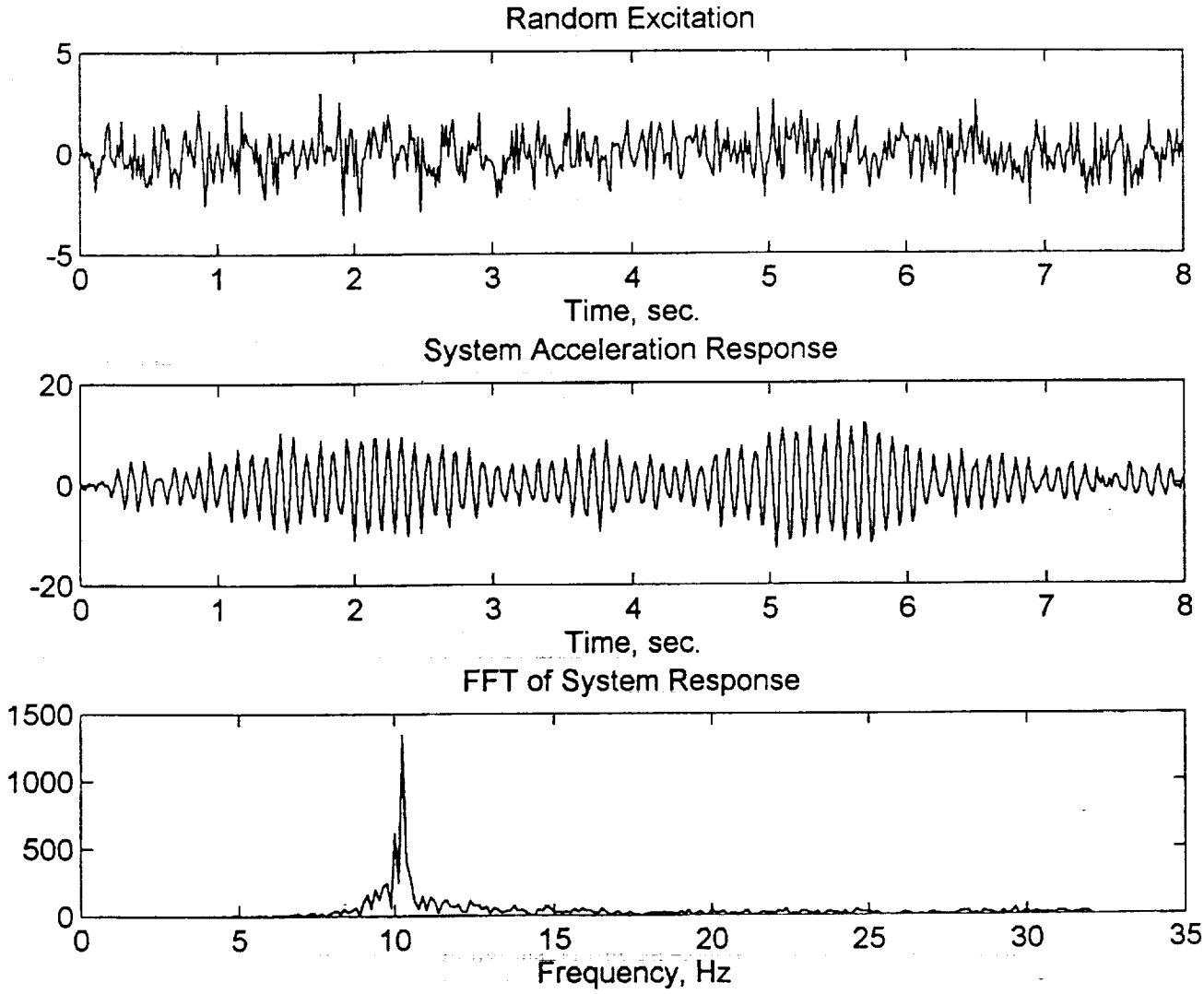

Figure 3.2: Time Histories of the Single DOF System Excitation and Response Signals and the FFT of the Response (f_s = 64 Hz)

Corresponding modal parameters of the system are then obtained using equations (2.8)-(2.10) and (2.15) as

off-line natural frequency: $\omega_{off} = 11$ Hz,
off-line damping ratio: $\zeta_{off} = 0.01$,
off-line modal constant: $A_{pq\text{-}off} = 1.0$

During the simulation, if the off-line modal parameters come out exactly the same as the actual system parameters, one can assume that the mathematical development of the adaptive LC equations is correct.

At this point, one can proceed safely with the next step, which is testing the adaptation algorithm using the on-line method that also simulates the real-time implementation. First, the delay variables *y(k-1), y(k-2), u(k-1) and u(k-2)* which are needed to construct the adaptive LC inputs are initialized along with the weights and learning rate. A for statement is used to implement a loop that runs for a specified number of iterations.

As discussed in Chapter 2, the adaptive LC inputs are simply:

$$P1 = u(k) - 2u(k-1) + u(k-2)$$
$$P2 = y(k-1)$$
$$P3 = y(k-2)$$

Using these inputs and the weights the adaptive LC can be built using the linear relationship of equation (3.1). The error $e(k)$ at time k, is taken as the difference between the actual system response, $y(k)$ and the adaptive LC output, $y_{out}(k)$ at k (see Figure 2.2). This error is then fed into the adaptation algorithm of equation (2.7) to update the weights. The on-line modal parameters are calculated during every iteration.

The results are presented in a graphical format for various sampling rates as shown in Figures 3.3 to 3.8. The plots show that the modal parameters converge to the actual system values in a very short time depending on the sampling rate. A higher sampling rate as shown usually means quicker convergence because more data points are available for a given length of time. Figure 3.3 ($f_s = 32$ Hz) shows that the adaptive LC weights converge to the off-line values as stated above within 1 second, whereas Figure 3.8 ($f_s = 1024$ Hz) shows an almost instantaneous convergence. The error goes to zero at the same rate indicating accurate system identification. The convergence rate can also be modified by adjusting the learning rate. A learning rate of 1.0 means total error correction, any value greater than 1.0 means that the error would be overcorrected and should not be used [4]. In this case, a learning rate of 1.0 was used and found to work well. On the other hand, such a high learning rate can cause problems for the adaptation process in some cases, therefore a smaller value must be used. This problem becomes evident in the two mode identification process, where in some cases a learning rate as small as 5% can be too high and produce poor results.

As shown in the plots, for all cases, system identification took place without difficulty even at sampling rates higher than 1kHz. Thus, it can be concluded that for the single DOF case, the adaptation algorithm is not adversely affected by the sampling rate. This as it turns out, is not necessarily true for the two DOF case as will be shown in the next section, where the adaptation process is significantly affected by the sampling frequency.

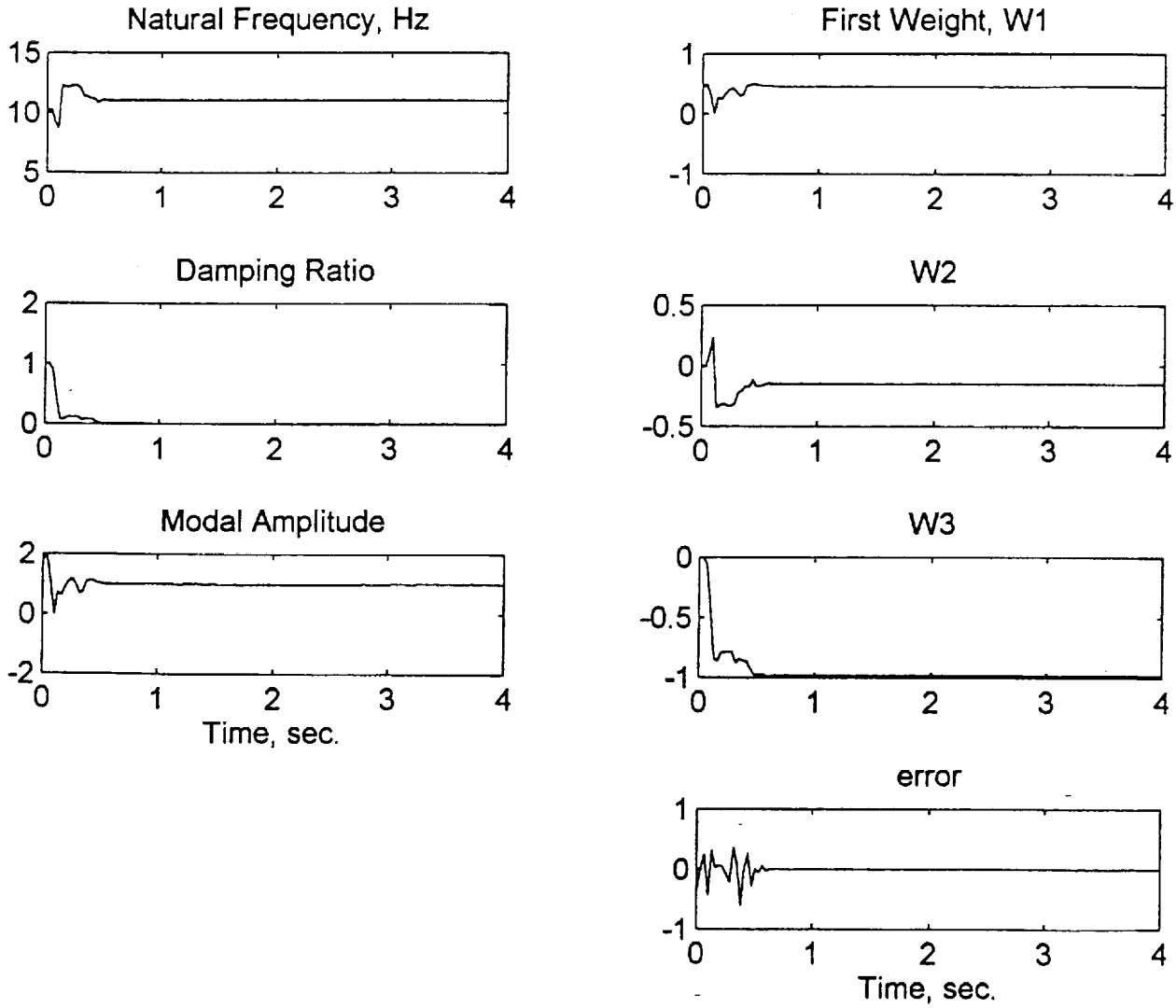

Figure 3.3: On-Line Modal Parameters, Weights and Error for the Single Mode Identification in MATLAB
(f_s = 32 Hz, α = 1.0)

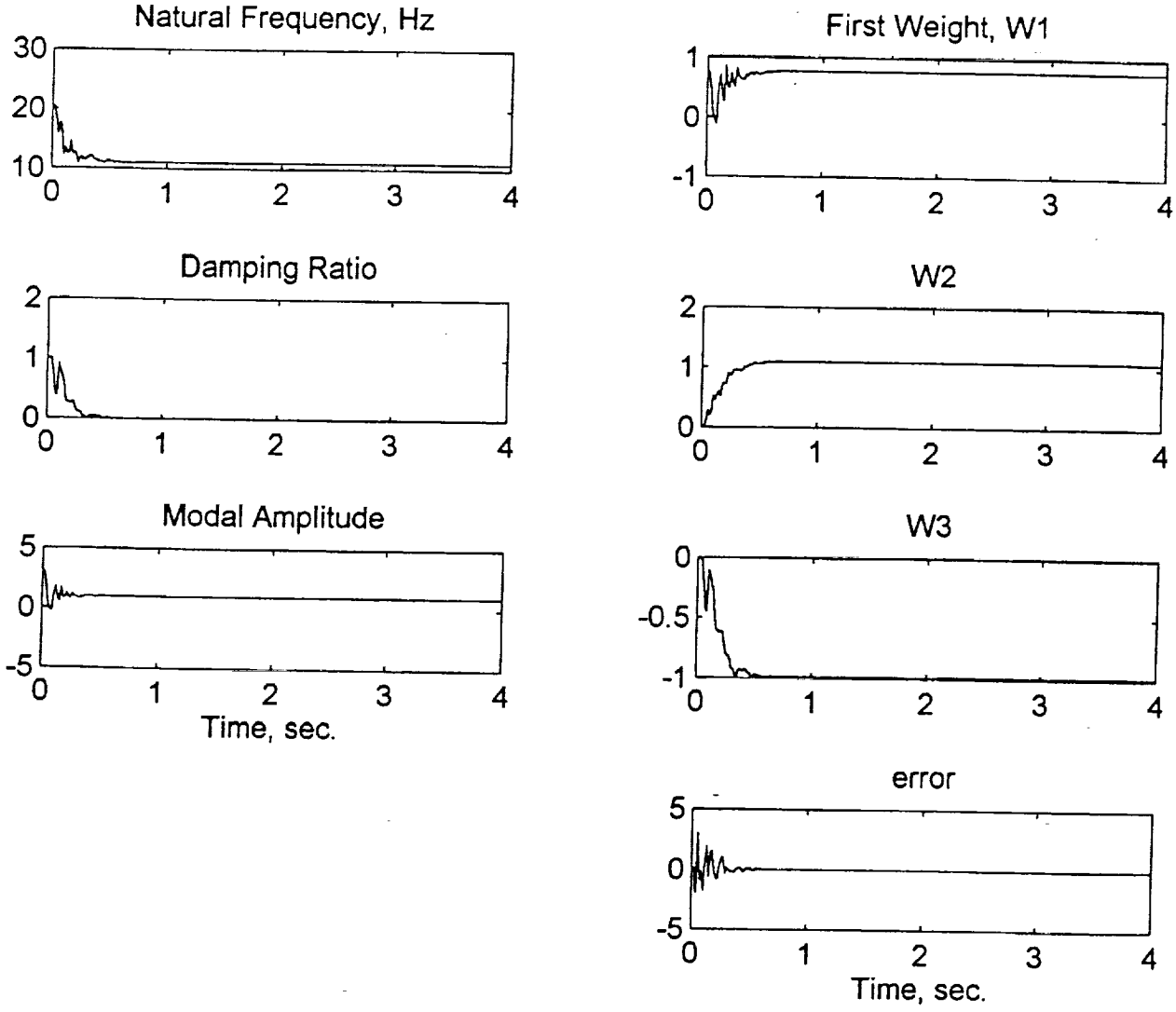

Figure 3.4: On-Line Modal Parameters, Weights and Error for the Single Mode Identification in MATLAB
(f_s = 64 Hz, α = 1.0)

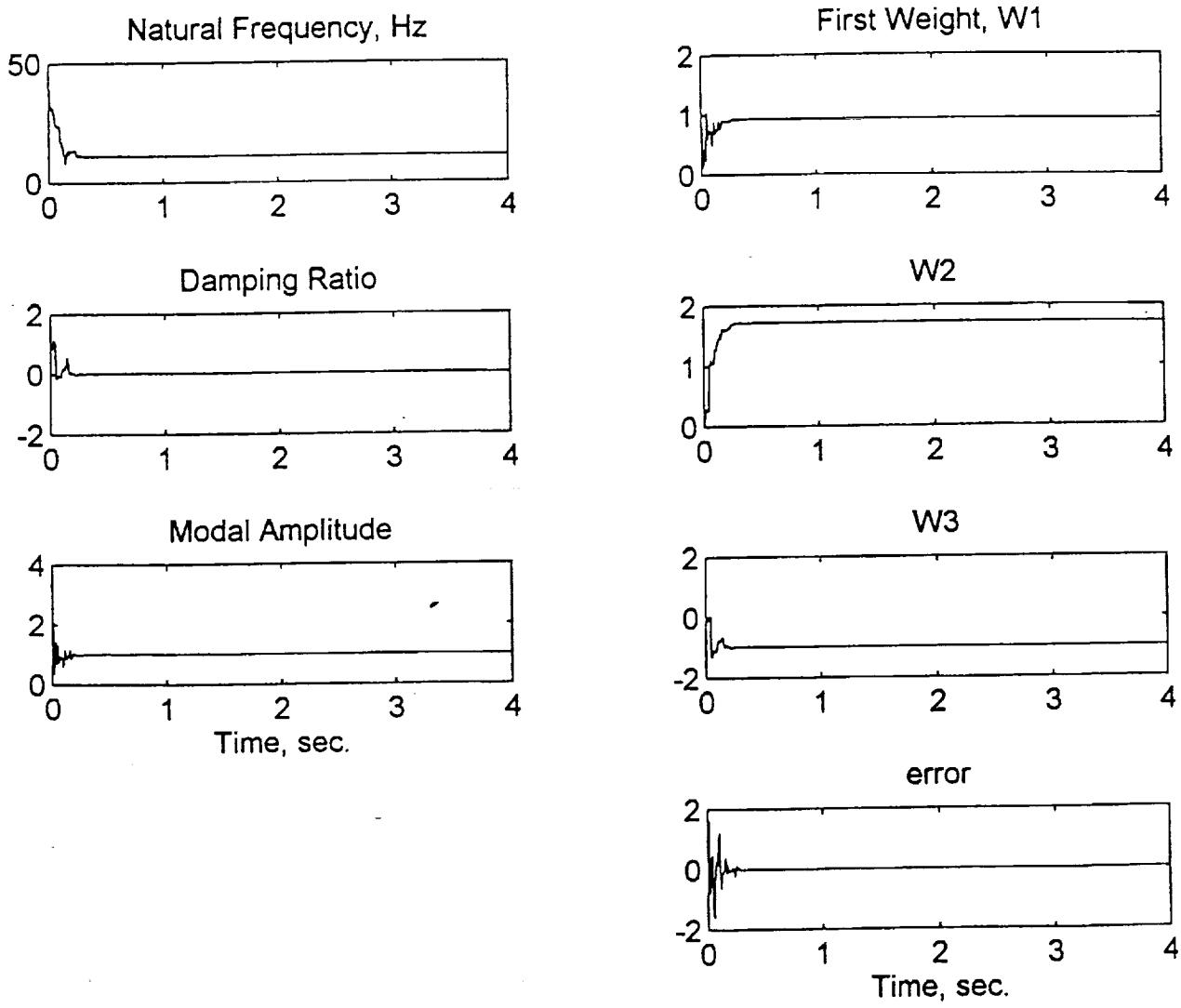

Figure 3.5: On-Line Modal Parameters, Weights and Error for the Single Mode Identification in MATLAB
(f_s = 128 Hz, α = 1.0)

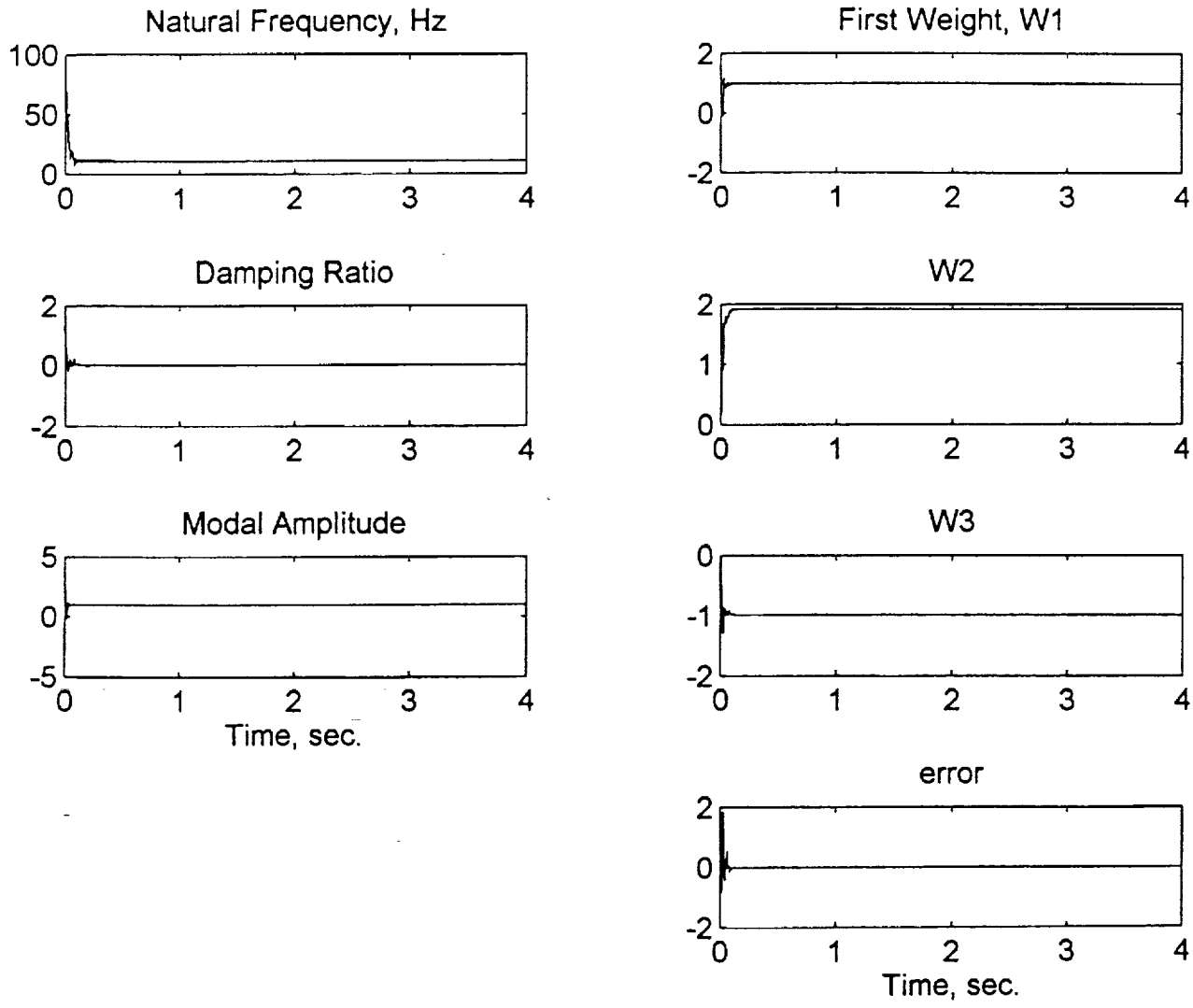

Figure 3.6: On-Line Modal Parameters, Weights and Error for the Single Mode Identification in MATLAB
(f_s = 256 Hz, α = 1.0)

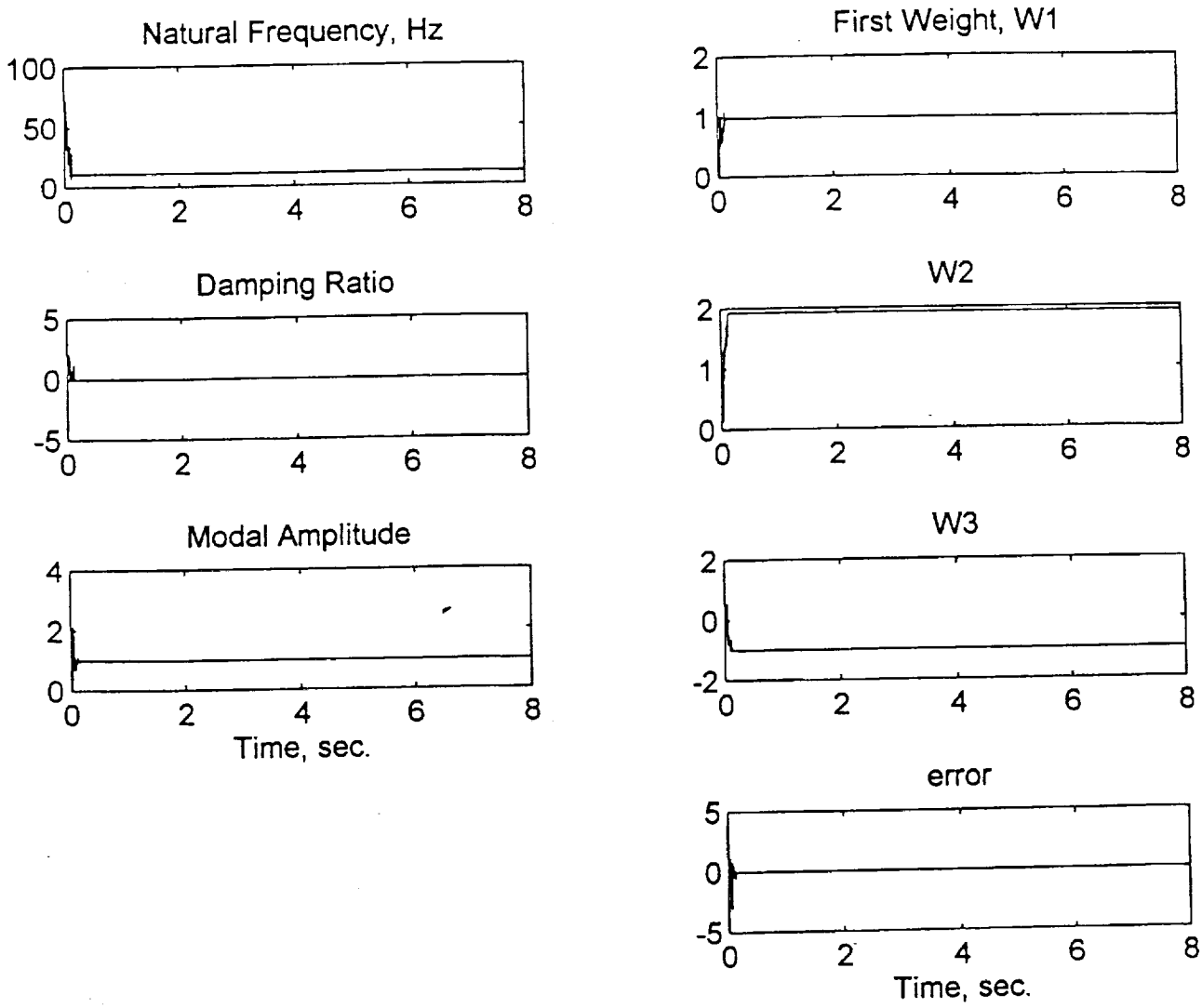

Figure 3.7: On-Line Modal Parameters, Weights and Error for the Single Mode Identification in MATLAB
(f_s = 512 Hz, α = 1.0)

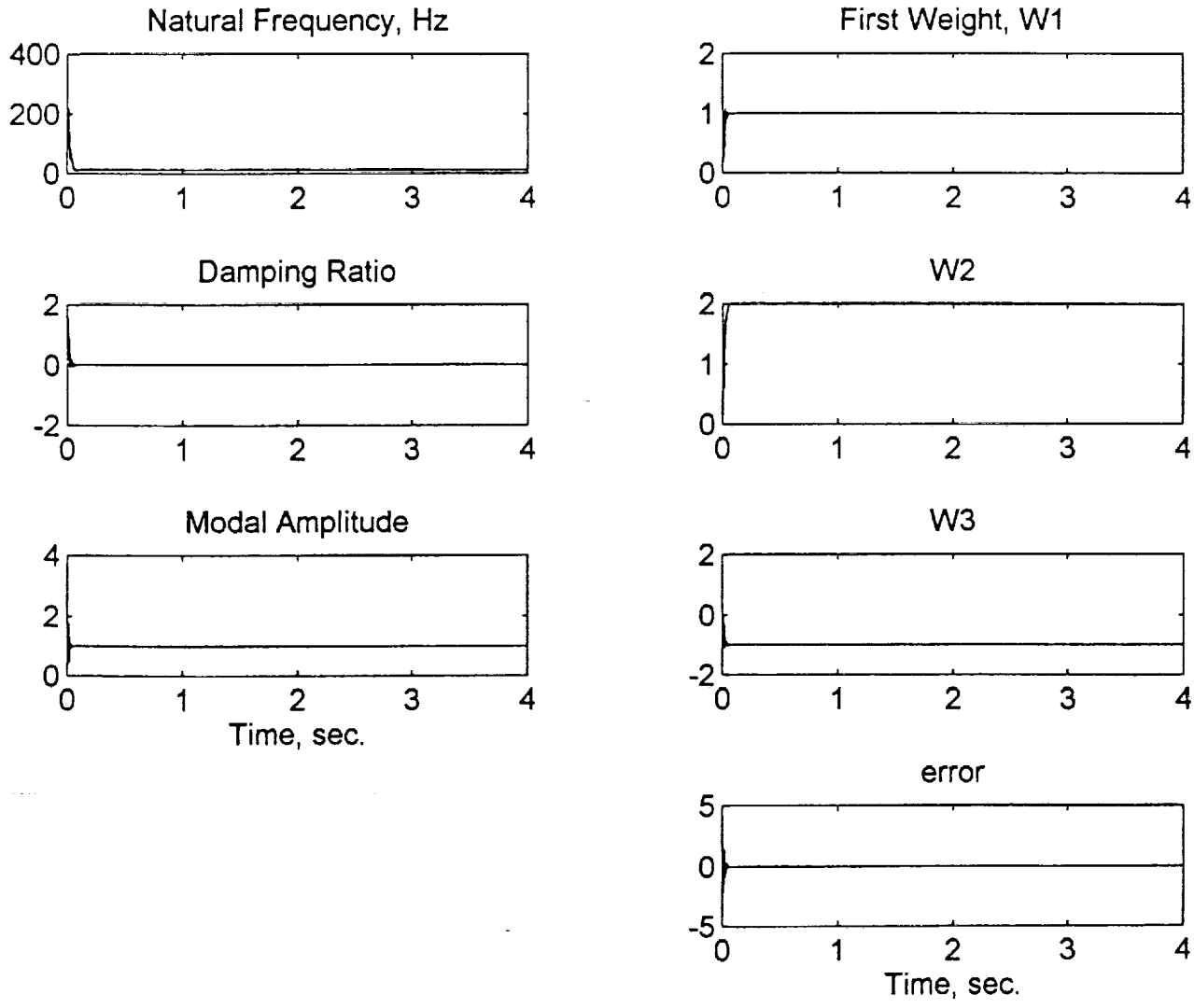

Figure 3.8: On-Line Modal Parameters, Weights and Error for the Single Mode Identification in MATLAB
(f_s = 1024 Hz, α = 1.0)

3.2 Two DOF Case

The MATLAB simulation program for the two DOF case described in this section is included in Appendix A. As shown in Figure 3.9, the two DOF spring-mass-damper system driven by external forces is represented by the following equation of motion in matrix form [8]:

$$\begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix}\begin{bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{bmatrix}+\begin{bmatrix} c_1+c_2 & -c_2 \\ -c_2 & c_2 \end{bmatrix}\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix}+\begin{bmatrix} k_1+k_2 & -k_2 \\ -k_2 & k_2 \end{bmatrix}\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix}=F\begin{bmatrix} f_1(t) \\ f_2(t) \end{bmatrix} \quad (3.2)$$

where $f(t)$ is an excitation applied to a mass, and $x(t)$ is a displacement response of a mass.

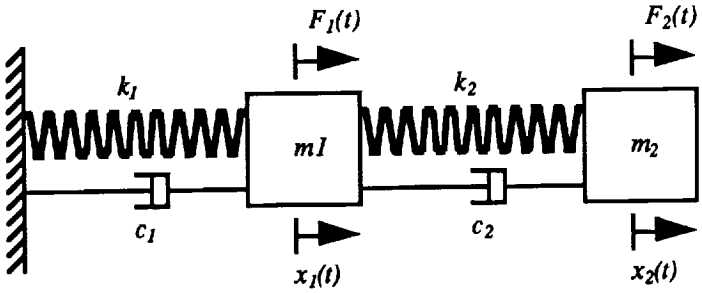

Figure 3.9: Two DOF Spring-Mass-Damper System

The modal coordinate transformation is employed to decouple the equations of motion for a multiple DOF system as

$$\mathbf{x} = M^{-\frac{1}{2}}\mathbf{q} = M^{-\frac{1}{2}}E\mathbf{r} \quad (3.3)$$

where:

$$\mathbf{x}=\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix}, \quad M=\begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix}, \quad \mathbf{q}=\begin{bmatrix} q_1(t) \\ q_2(t) \end{bmatrix}, \quad \mathbf{r}=\begin{bmatrix} r_1(t) \\ r_2(t) \end{bmatrix}$$

and E is a matrix that contains the normalized eigenvectors:

$$E = [\mathbf{v}_1\ \mathbf{v}_2\ \mathbf{v}_3 \ldots .\ \mathbf{v}_n] \quad (3.4)$$

where n is the number of modes (here n = 2). The mode shapes can be extracted from the eigenvectors using the following equation:

$$\Phi = M^{-1/2}\mathbf{v}$$

Substituting equation (3.3) into equation (3.2) and multiplying by $M^{-\frac{1}{2}}$ gives

$$M^{-\frac{1}{2}}MM^{-\frac{1}{2}}\ddot{\mathbf{q}} + M^{-\frac{1}{2}}CM^{-\frac{1}{2}}\dot{\mathbf{q}} + M^{-\frac{1}{2}}KM^{-\frac{1}{2}}\mathbf{q} = M^{-\frac{1}{2}}F\mathbf{f}$$

C and K are the damping and stiffness matrices, respectively. F is a 2x2 identity matrix. The above equation is simplified to

$$I\ddot{\mathbf{q}} + \tilde{C}\dot{\mathbf{q}} + \tilde{K}\mathbf{q} = M^{-\frac{1}{2}}F\mathbf{f}$$

where

$$M^{-\frac{1}{2}}MM^{-\frac{1}{2}} = I \quad , \quad \tilde{C} = M^{-\frac{1}{2}}CM^{-\frac{1}{2}} \quad , \quad \tilde{K} = M^{-\frac{1}{2}}KM^{-\frac{1}{2}}$$

In the modal coordinate system, the equation becomes:

$$E\ddot{\mathbf{r}} + \tilde{C}E\dot{\mathbf{r}} + \tilde{K}E\mathbf{r} = M^{-\frac{1}{2}}F\mathbf{f}$$

$$E^T E\ddot{\mathbf{r}} + E^T\tilde{C}E\dot{\mathbf{r}} + E^T\tilde{K}E\mathbf{r} = E^T M^{-\frac{1}{2}}F\mathbf{f}$$

where

$$E^T E \cong \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad , \quad E^T\tilde{C}E = \begin{bmatrix} 2\zeta_1 w_1 & 0 \\ 0 & 2\zeta_2\omega_2 \end{bmatrix} \quad , \quad E^T\tilde{K}E = \begin{bmatrix} \omega_1^2 & 0 \\ 0 & \omega_2^2 \end{bmatrix}$$

The equation can also be cast into a state-space format:

$$\begin{bmatrix} \dot{r}_1 \\ \dot{r}_2 \\ \ddot{r}_1 \\ \ddot{r}_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\omega_1^2 & 0 & -2\zeta_1\omega_1 & 0 \\ 0 & -\omega_2^2 & 0 & -2\zeta_2\omega_2 \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \\ \dot{r}_1 \\ \dot{r}_2 \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ E^T M^{-\frac{1}{2}}F \end{bmatrix} \mathbf{f}$$

and the corresponding acceleration output is:

$$\begin{bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{bmatrix} = M^{-\frac{1}{2}}E\begin{bmatrix} \ddot{r}_1 \\ \ddot{r}_2 \end{bmatrix} = \left[M^{-\frac{1}{2}}E\left[diag(-\omega_i^2)\right] \quad M^{-\frac{1}{2}}E\left[diag(-2\zeta_i\omega_i)\right]\right] \begin{bmatrix} r_1 \\ r_2 \\ \dot{r}_3 \\ \dot{r}_4 \end{bmatrix}$$

$$+ \left[M^{-\frac{1}{2}}EE^T M^{-\frac{1}{2}}F\right]$$

This state-space model is used in MATLAB to represent the two DOF system with the following values:

$m_1 = m_2 = 3$ kg
$k_1 = 35$ kN/m
$k_2 = 150$ kN/m
$C = 0.0001*K$ N·s/m

The resulting modal parameters are:

first mode natural frequency:	ω_1 = 11.8 Hz (74 rad/sec)
first mode damping ratio:	ζ_1 = 0.004

first mode shape: $\Phi_1 = \begin{bmatrix} {}_1\phi_1 \\ {}_1\phi_2 \end{bmatrix} = \begin{bmatrix} 0.384 \\ 0.431 \end{bmatrix}$

second mode natural frequency: $\omega_2 = 52$ Hz (326 rad/sec)

second damping ratio: $\zeta_2 = 0.016$

second mode shape: $\Phi_2 = \begin{bmatrix} {}_2\phi_1 \\ {}_2\phi_2 \end{bmatrix} = \begin{bmatrix} -0.431 \\ 0.384 \end{bmatrix}$

A random excitation can be applied to either m_1 or m_2 depending on the user's choice. The two DOF program given in Appendix A, requests an input for an excitation point. Similarly, the user is asked to enter a response point. A same point excitation and response measurement must be taken to solve for the mode shape coefficients. If the mode shape coefficients at both of the masses are desired, the simulation program will have to be run twice. Figure 3.10 shows the random excitation signal, the system acceleration response (f_s = 200 Hz) and the fast fourier transform (FFT) of the response signal. Note that the FFT shows the first mode at the expected frequency (11.8 Hz), whereas the second mode is shown at a lower frequency (around 45 Hz instead of the expected 52Hz). This is the result of a phenomenon associated with the bilinear transformation called frequency warping, which will be discussed in detail in Chapter 5.

Once the system response is obtained, the rest of the simulation program is similar to the single DOF case with the exception of the adaptive filter coefficients extraction process from the weights. In the SDOF case, the filter coefficients and the weights have the simple relationships shown in equation (2.15). However, for the two DOF case, the relationship between the weights and the filter coefficients is more complicated as shown in equation (2.19). A sixth order polynomial equation (2.26) must be solved before any coefficients can be extracted. This is done using the Newton root approximation algorithm. The MATLAB roots command which solves for the roots of a polynomial by finding the eigenvalues of the associated companion matrix could have been used in this case, which would have reduced the amount of code and iteration time tremendously. However, the reason this easy route was not taken is due to the fact that the author wishes to make the simulation code as close to the real-time code (C code) in chapter 5 as possible to facilitate troubleshooting. The Newton iteration method solves the equation $f(x) = 0$, where f is assumed to be continuous and differentiable. Using an approximate value of the root x_n, a new value, x_{n+1} that is closer to the root (assuming convergence) is found by the following [9]:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

This is done using a for loop as shown in the simulation program (DDOFSIM.M) in Appendix A. It was found that the Newton method converges to a root after only a few iterations with an initial value, x_o=1. Since only one real root is needed to solve equation (2.26), which is continuous and differentiable, this turned out to be a good estimate. Once equation (2.26) is solved for one root (coefficient), equation (2.27) is solved simultaneously for the rest of the filter coefficients. The modal parameters can then be extracted using equations (2.8)-(2.10).

Here are the results of a sample run (f_s = 128, α = 1.0):

<u>First Run (excitation at point 1, response at point 2)</u>

off-line weights: $W_{1off} = 1.2213$
$W_{2off} = -1.1787$

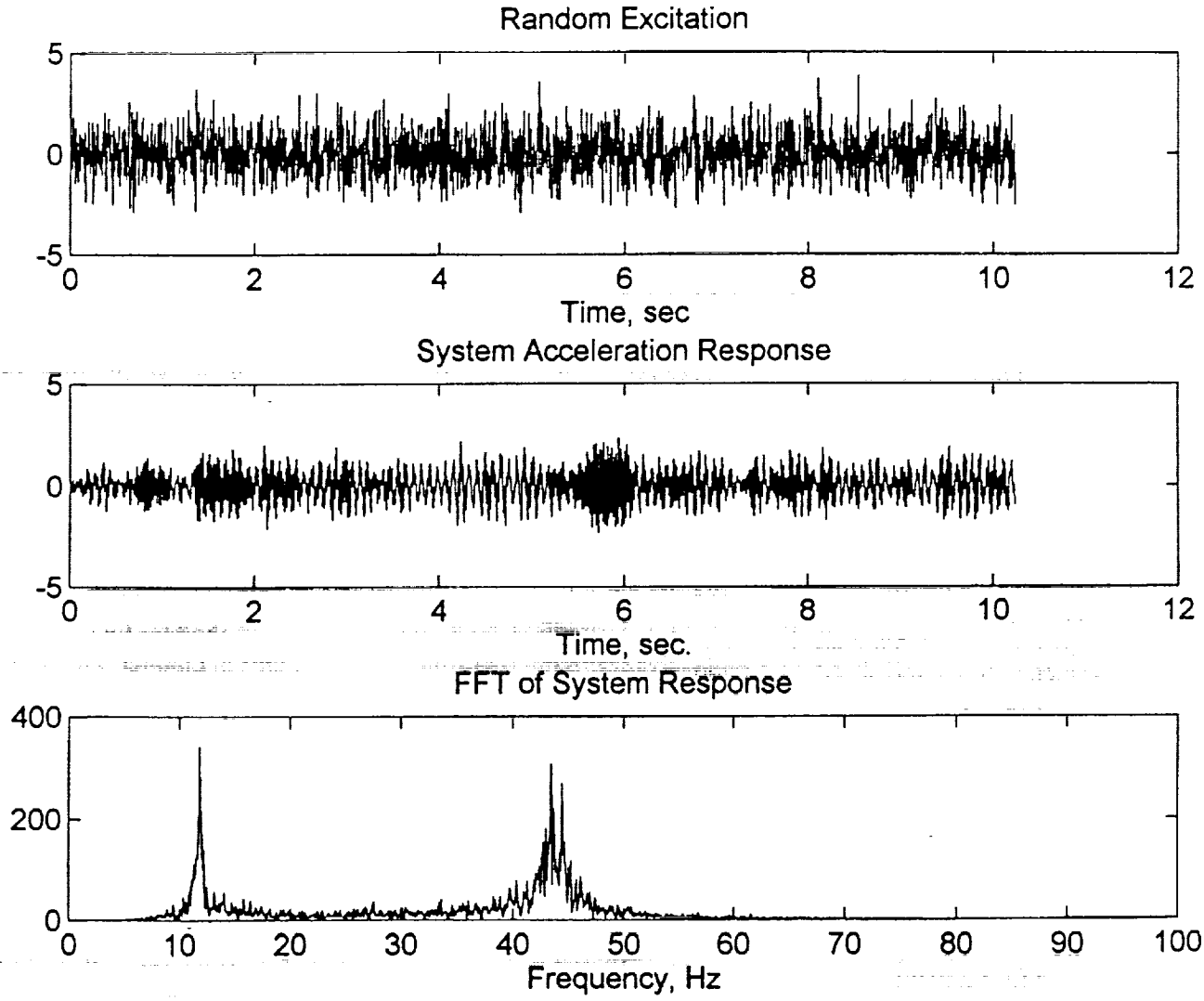

Figure 3.10: Time Histories of the Two DOF System Excitation and Acceleration Response Signals and the Fast Fourier Transform of the System Response (f_s = 200Hz)

$$W_{3off} = 1.1706$$
$$W_{4off} = -0.9650$$
$$W_{5off} = 0.0902$$
$$W_{6off} = 0.1760$$
$$W_{7off} = 0.0857$$

which give the following off-line modal parameters:

first mode off-line natural frequency: $\omega_{1off} = 74.1$ rad/sec (11.8 Hz),
. . . . . damping ratio: $\zeta_{1off} = 0.004$,
. . . . . modal constant: ${}_1A_{21\text{-}off} = 0.1655$
second mode off-line natural frequency: $\omega_{2off} = 326$ rad/sec (52 Hz),
. . . . . damping ratio: $\zeta_{2off} = 0.016$,
. . . . . modal constant: ${}_2A_{21\text{-}off} = -0.1655$

Second Run (same point excitation and response @ point 1)

off-line weights:
$$W_{1off} = 1.2213$$
$$W_{2off} = -1.1787$$
$$W_{3off} = 1.1706$$
$$W_{4off} = -0.9650$$
$$W_{5off} = 0.2056$$
$$W_{6off} = -0.0547$$
$$W_{7off} = 0.2011$$

which give the following off-line modal parameters:

first mode off-line natural frequency: $\omega_{1off} = 74.1$ rad/sec (11.8 Hz),
. . . . . damping ratio: $\zeta_{1off} = 0.004$,
. . . . . modal constant: ${}_1A_{11\text{-}off} = 0.1474$
second mode off-line natural frequency: $\omega_{2off} = 326$ rad/sec (52 Hz),
. . . . . damping ratio: $\zeta_{2off} = 0.016$,
. . . . . modal constant: ${}_2A_{11\text{-}off} = 0.186$

Using equation (2.2), the modal constants are solved and the following mode shapes are produced

$$\Phi_1 = \begin{bmatrix} {}_1\phi_1 \\ {}_1\phi_2 \end{bmatrix} = \begin{bmatrix} 0.384 \\ 0.431 \end{bmatrix} \qquad \Phi_2 = \begin{bmatrix} {}_2\phi_1 \\ {}_2\phi_2 \end{bmatrix} = \begin{bmatrix} -0.484 \\ 0.342 \end{bmatrix}$$

which are fairly close to the actual values shown earlier (page 41).

Note that the first four weights are the same in both cases whereas the last three are different. This is due to the fact that, as shown in equation (2.19), the first four weight equations are functions of the filter denominator coefficients (a's) only. These influence all modal parameters of the specific mode according to equations (2.8)-(2.10). Whereas, the filter numerator coefficients (b's) only influence the modal shape as shown in equation (2.10). As a result, equation (2.19) shows that the b-coefficients appear in the last three weights only.

The development of the on-line part of the code for this case is similar to that of the SDOF case. The results are discussed next.

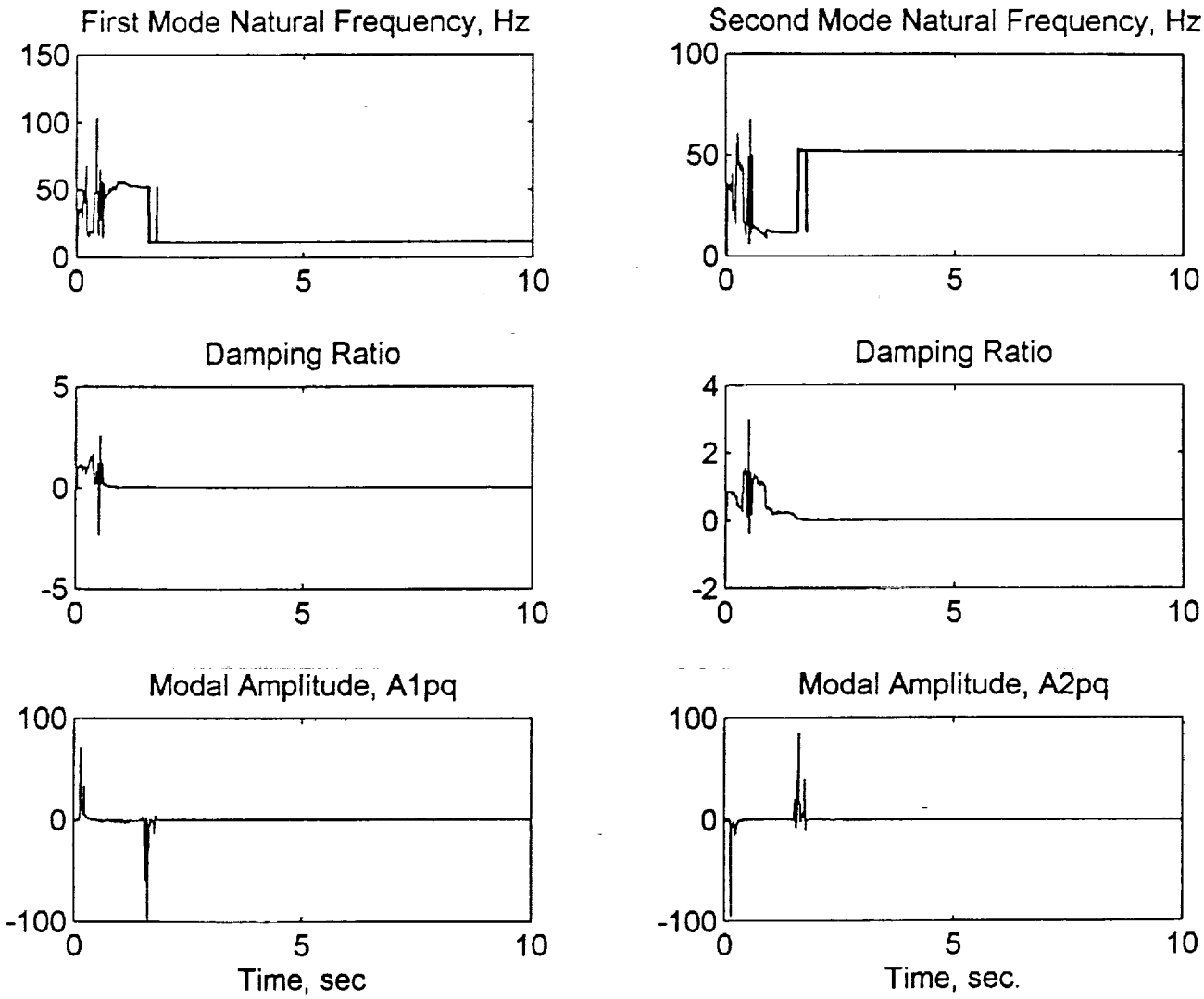

Figure 3.11a: On-Line Modal Parameters of the Simultaneous Two Mode Identification done in MATLAB
(Case 1: f_s = 110 Hz, α = 1.0)

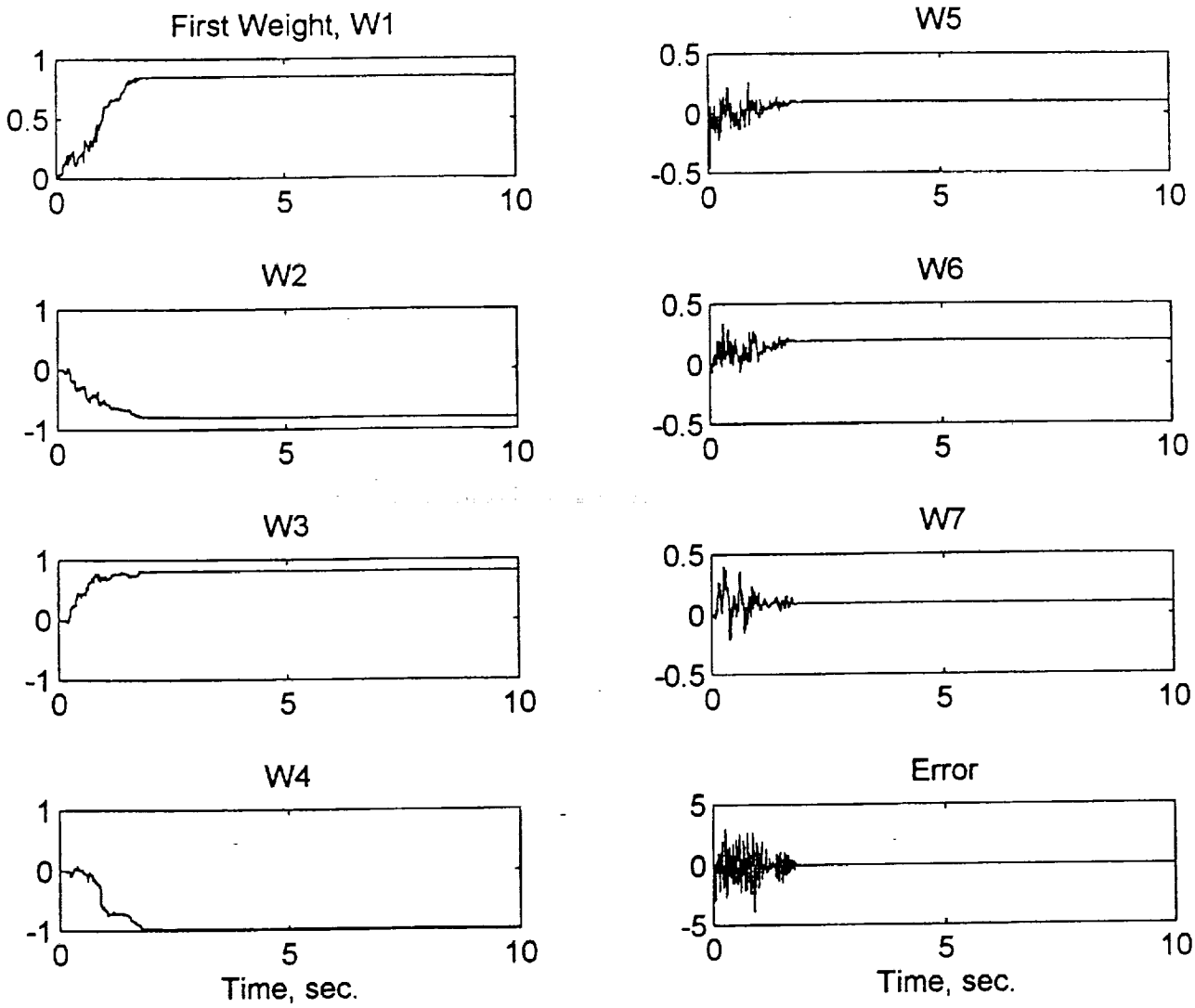

Figure 3.11b: Adaptive LC Weights and Error
(Case 1: f_s = 110 Hz, α = 1.0)

Case 1 **(Figure 3.11a,b)** Sampling Rate = 110 Hz
Learning Rate = 1.0

The sampling rate is slightly higher than twice the second mode natural frequency. A learning rate of 1.0 is used for total error correction. The first and second mode parameters converge to the exact off-line values within 2 seconds. The weights also converge to the off-line values and the error goes to zero within the same time indicating full identification.

Case 2 **(Figure 3.12a,b)** Sampling Rate = 128 Hz
Learning Rate = 1.0

Slightly increasing the sampling rate to 128 Hz does not have much of an effect on the results. The modal parameters are accurately identified within approximately 2.5 sec.

Case 3 **(Figure 3.13a,b)** Sampling Rate = 256 Hz
Learning Rate = 1.0

Increasing the sampling rate further to 256 Hz and using total error correction ($\alpha = 1.0$) produces poor results as shown. Even though the adaptive filter fails to accurately identify both modes, Figure 3.13b shows that the algorithm has tried to minimize the error. Allowing the adaptation process to run for a longer period of time did not produce better results. Possible explanation is given in the next section.

Case 4 **(Figure 3.14a,b)** Sampling Rate = 256 Hz
Learning Rate = 0.05

Reducing the learning rate significantly produced better results than the previous case. But the adaptive filter still fails to accurately identify both modes. As shown in Figure 3.14a, the first mode is totally skipped, the second mode appears as the first mode, and in place of the second mode the filter tried to identify an adjacent component that is not an actual mode. The following section contains possible explanation for this mode-skipping phenomenon.

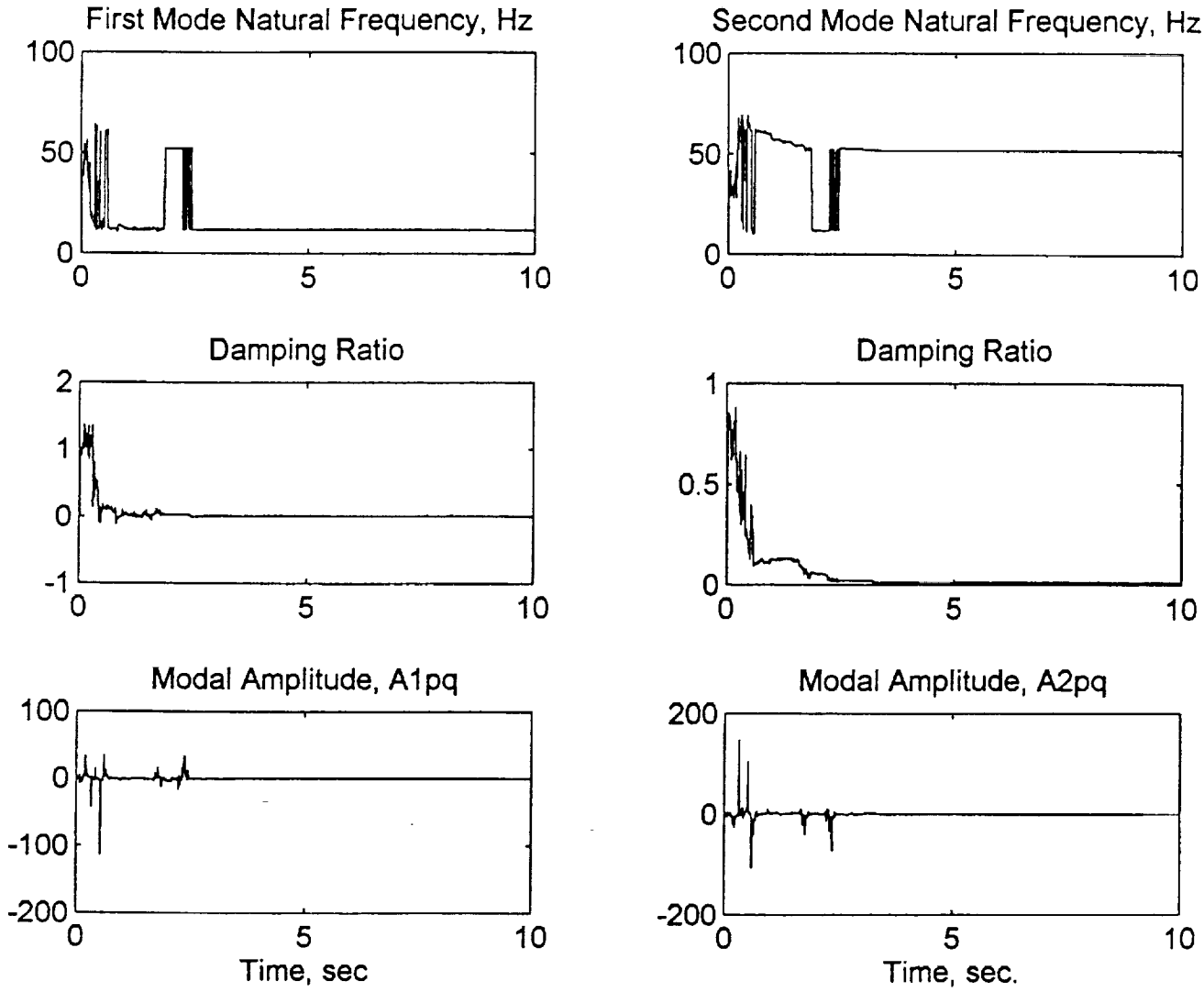

Figure 3.12a: On-Line Modal Parameters of the Simultaneous Two Mode Identification done in MATLAB
(Case 2: f_s = 128 Hz, α = 1.0)

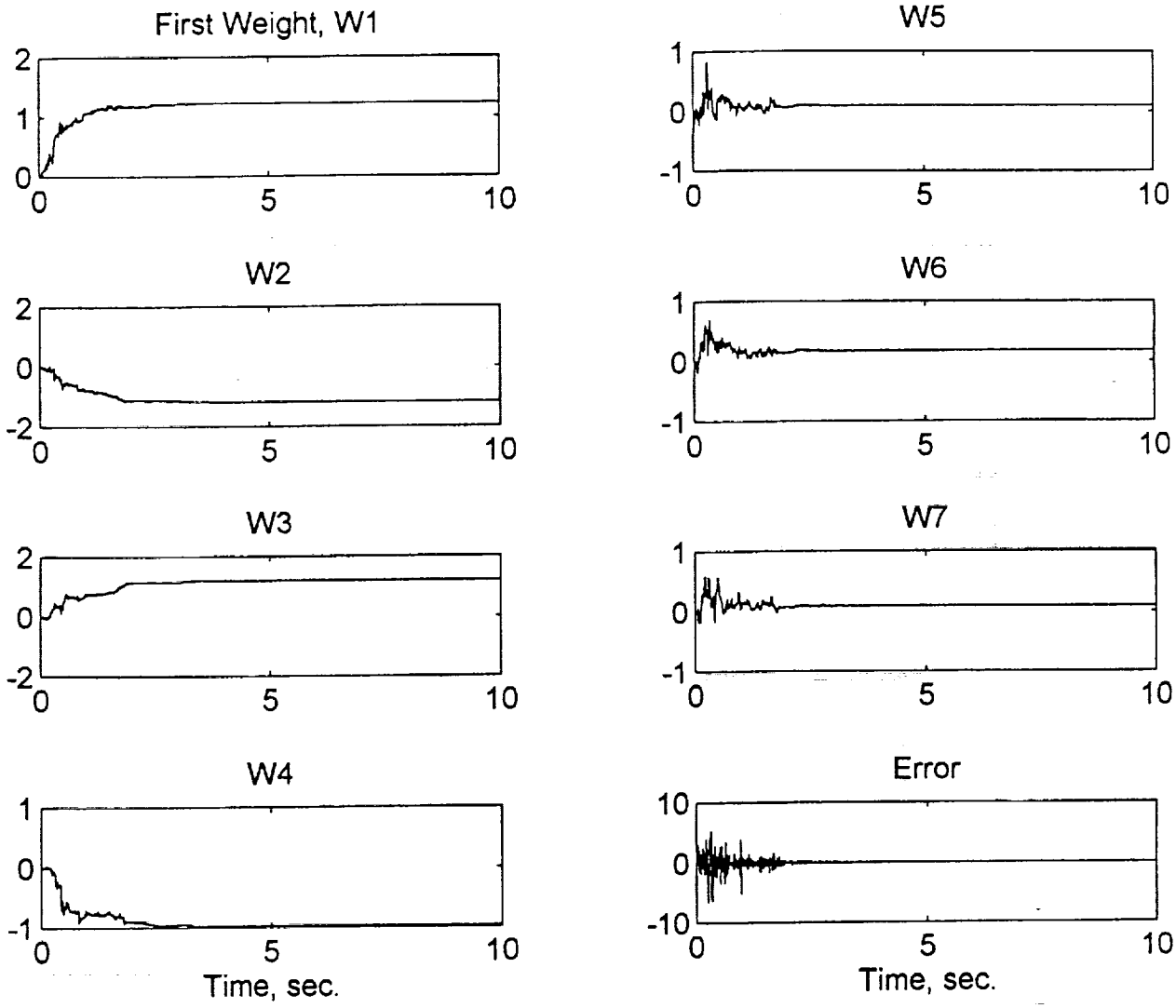

Figure 3.12b: Adaptive LC Weights and Error
(Case 2: f_s = 128 Hz, α = 1.0)

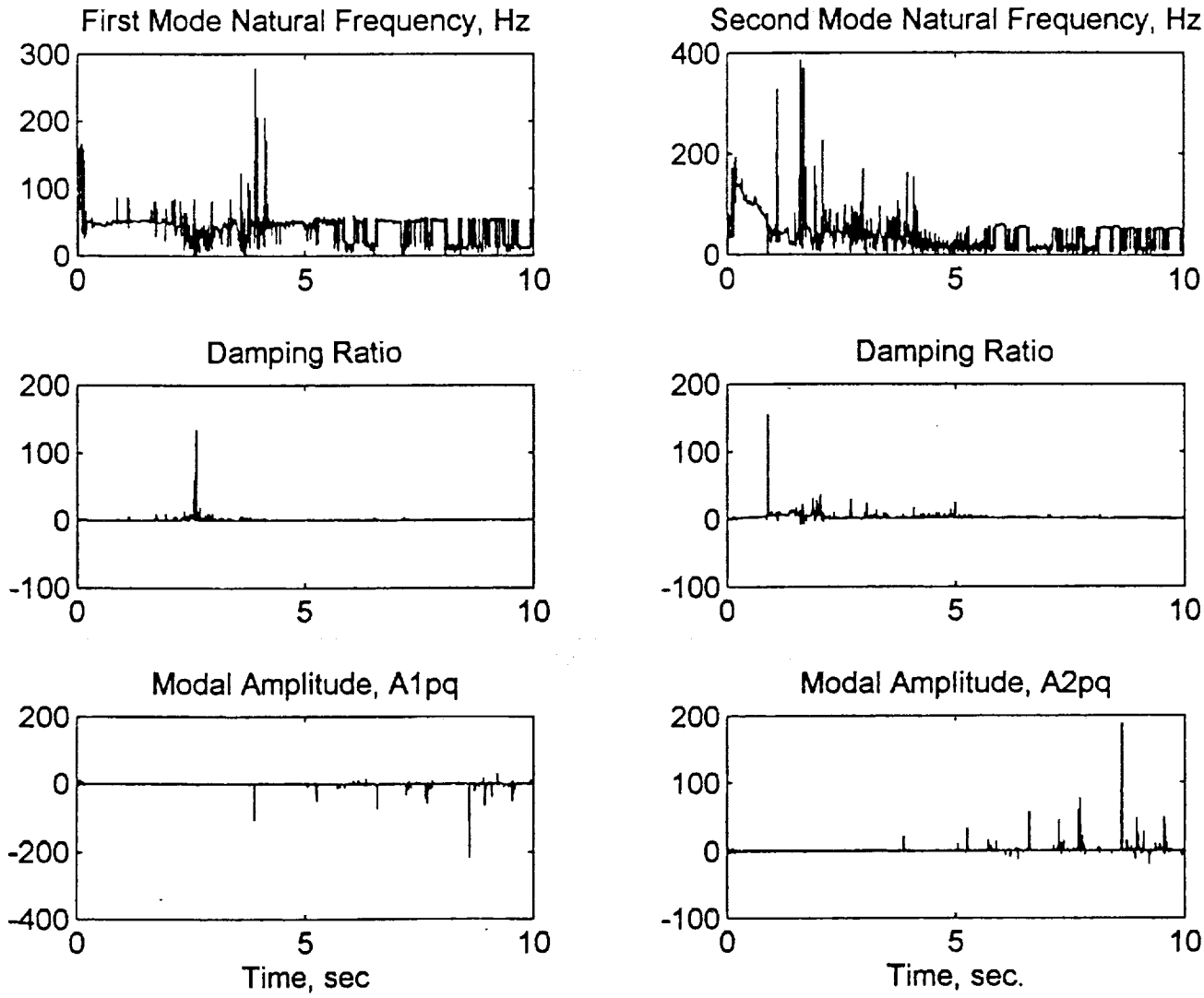

Figure 3.13a: On-Line Modal Parameters of the Simultaneous Two Mode Identification done in MATLAB
(Case 3: f_s = 256 Hz, α = 1.0)

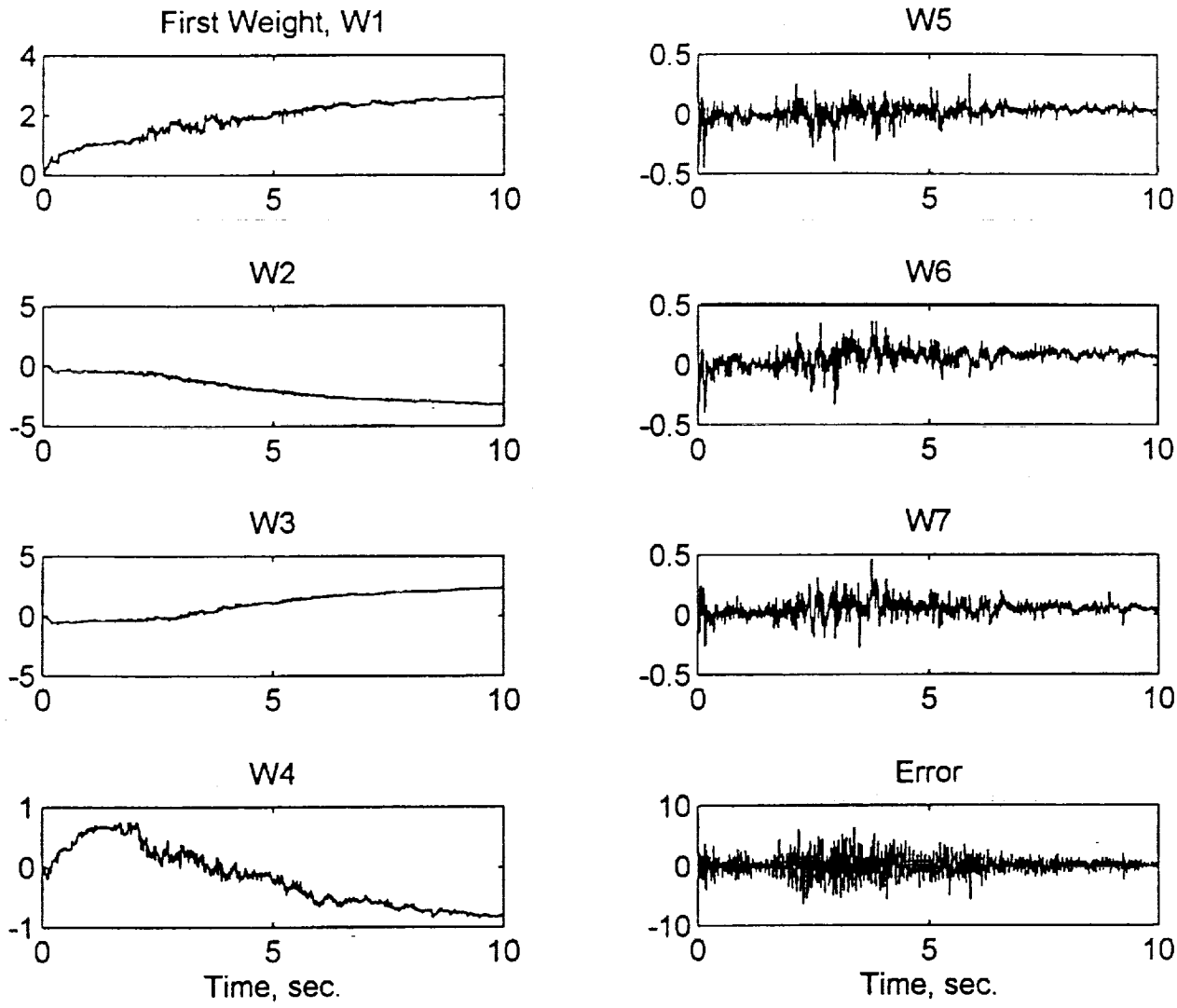

Figure 3.13b: Adaptive LC Weights and Error
(Case 3: f_s = 256 Hz, α = 1.0)

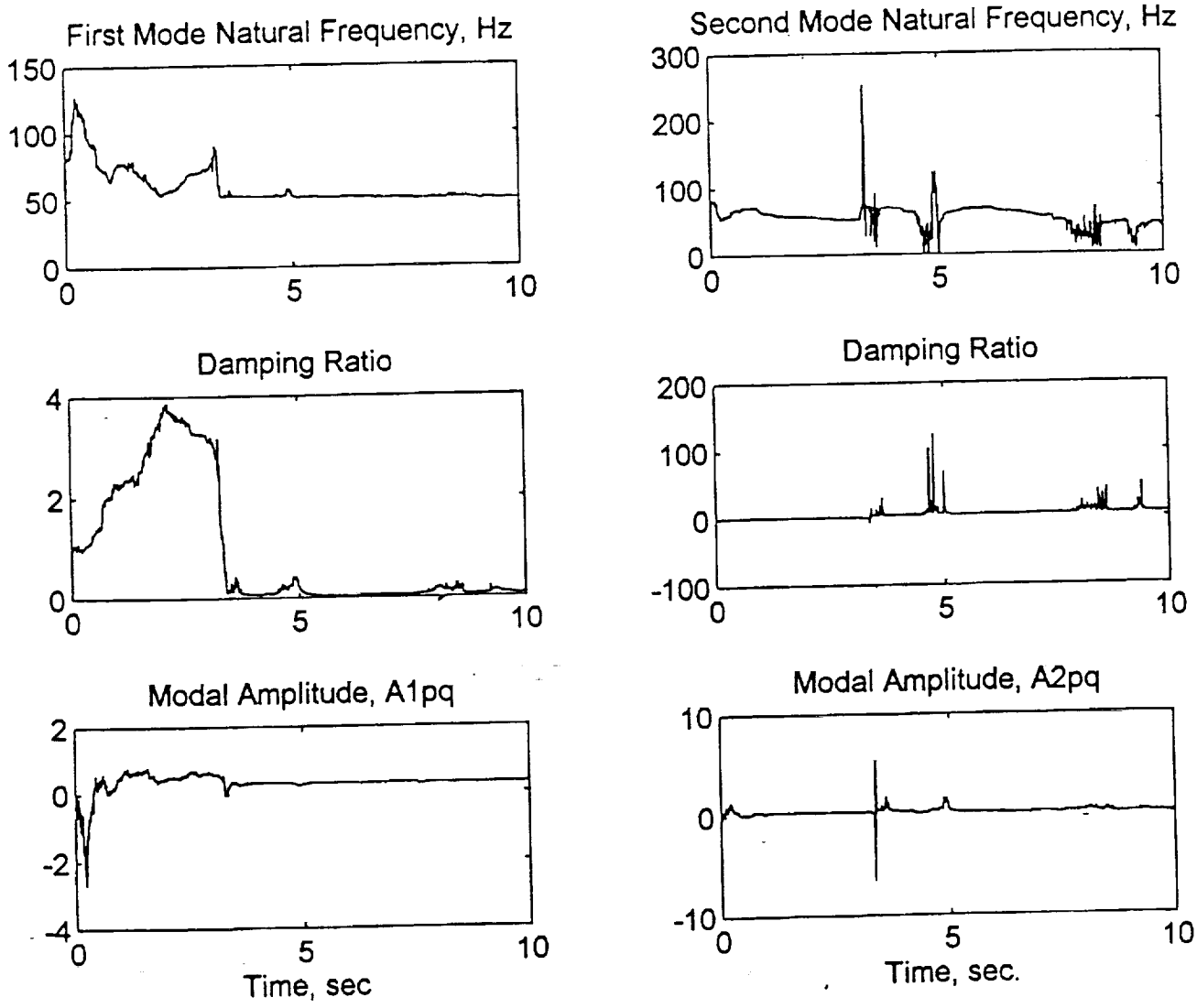

Figure 3.14a: On-Line Modal Parameters of the Simultaneous Two Mode Identification done in MATLAB
(Case 4: $f_s = 256$ Hz, $\alpha = 0.05$)

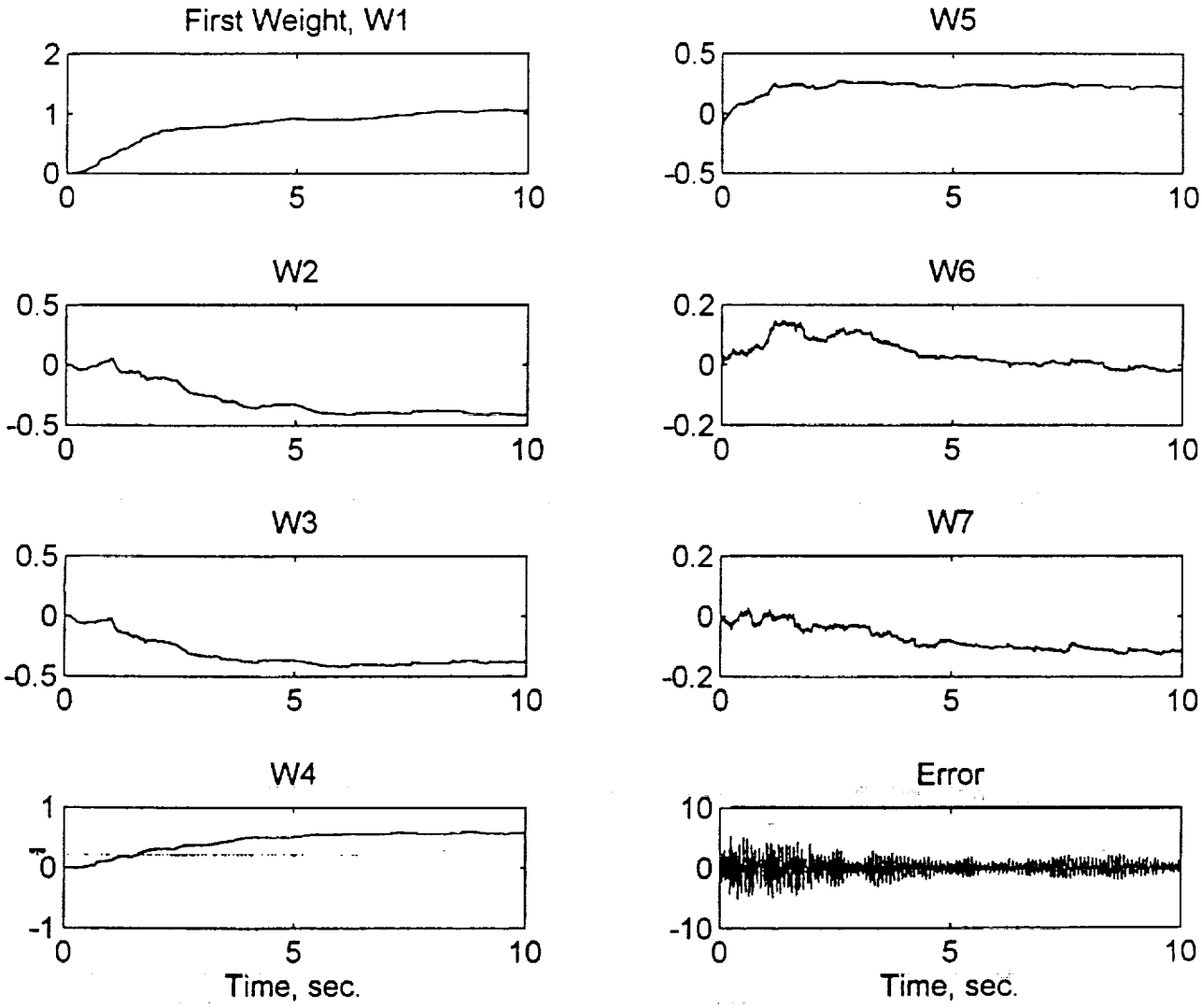

Figure 3.14b: Adaptive LC Weights and Error
(Case 4: f_s = 256 Hz, α = 0.05)

Case 5 (Figure 3.15a,b) Sampling Rate = 1024 Hz
Learning Rate = 0.05

Increasing the sampling rate to more than 1kHz produces worse results. Also, the mode skipping phenomenon still occurs where the actual second mode appears as the first mode and in place of the second mode, the adaptive filter tries to identify a false mode.

Note that the adaptation process duration for this case is 5 seconds only. This is done to save time and prevent the computer from running into memory limitations. However, in order to ensure that the adaptation process is allowed to run for a sufficient amount of time before coming to a final conclusion, the MATLAB program was slightly modified to save frequency information only and was run for an adaptation time of 20 seconds. The results were very similar to those shown in Figure 3.15a with no different behavior seen after the first five seconds. The following discussion tries to provide an explanation for such behavior.

Two Mode ID and Sampling Rate

It has been found in simulation and verified by real-time testing in the lab (Chapter 5) that unlike the single mode case, the simultaneous two mode identification process is sensitive to the sampling rate. When the test is run at a sampling rate that is slightly higher than twice the second mode natural frequency, then the algorithm performs well and full system identification can take place as shown in Figures 3.11, 3.12. As the sampling rate increases, system identification becomes harder to achieve. For example, in the specific case discussed here, it has been found that the algorithm is able to successfully drive the adaptive filter to fully identify the system in the frequency range of 110 Hz to somewhere slightly above 200 Hz. This is not an exact range because each test run is unique and sometimes a slight improvement can be seen at the high end of frequency by manipulating the learning rate. However, there always exists a sampling rate associated with each specific case beyond which there is no hope of accurate system identification. Such a sampling rate is hard to pinpoint and can only be determined by trial and error during testing. As a matter of fact, even under the exact same conditions, this sampling rate can differ from one test to the next. In general, it has been found that in most cases, if a sampling rate higher than 4-5 times the second mode natural frequency is used, then there is a high possibility of inaccurate results.

Many possible causes for this problem were considered and they were all fully investigated. Some of these included manipulating the learning rate, changing the damping and magnitude of the signals, using filtered excitation signals, increasing the number of data points, remodeling the continuous system (using a transfer function approach instead of the state-space model), and using the exact root finding command in MATLAB instead of the Newton algorithm. Most of these possible causes were not expected to be the source of the problem because the off-line modal parameters always came out to be exact regardless of the sampling rate. In addition, system identification was successful at low sampling rates.

For a while, the bilinear transformation was suspected as the culprit but it was hard to eliminate this possibility in simulation because the discrete system is obtained using this particular method and using any other method, such as zero-order hold, did not work at all because the basic idea is different. However, this possibility was eliminated during the real-time testing as discussed in Chapter 5. In the real-time testing the discrete signal is obtained by sampling an actual dynamic system response, so the bilinear transformation does not come into play until the training is done and then its inverse is used to show the results in the continuous time domain (equations 2.8, 2.9, 2.10). Therefore, the error term and the weights in the real-time testing are not influenced by the bilinear transformation. When these were investigated, it was found that at the higher sampling rates, the weights did not converge to any reasonable values and the error was not minimized as it should be, at this point the bilinear transformation was eliminated as a possible cause. This has left one last option, that the problem is actually in the adaptation algorithm itself.

The adaptation algorithm has proven to work well for the single DOF case at any sampling rate and for a limited range of sampling rates in the two DOF case. This is enough proof that the

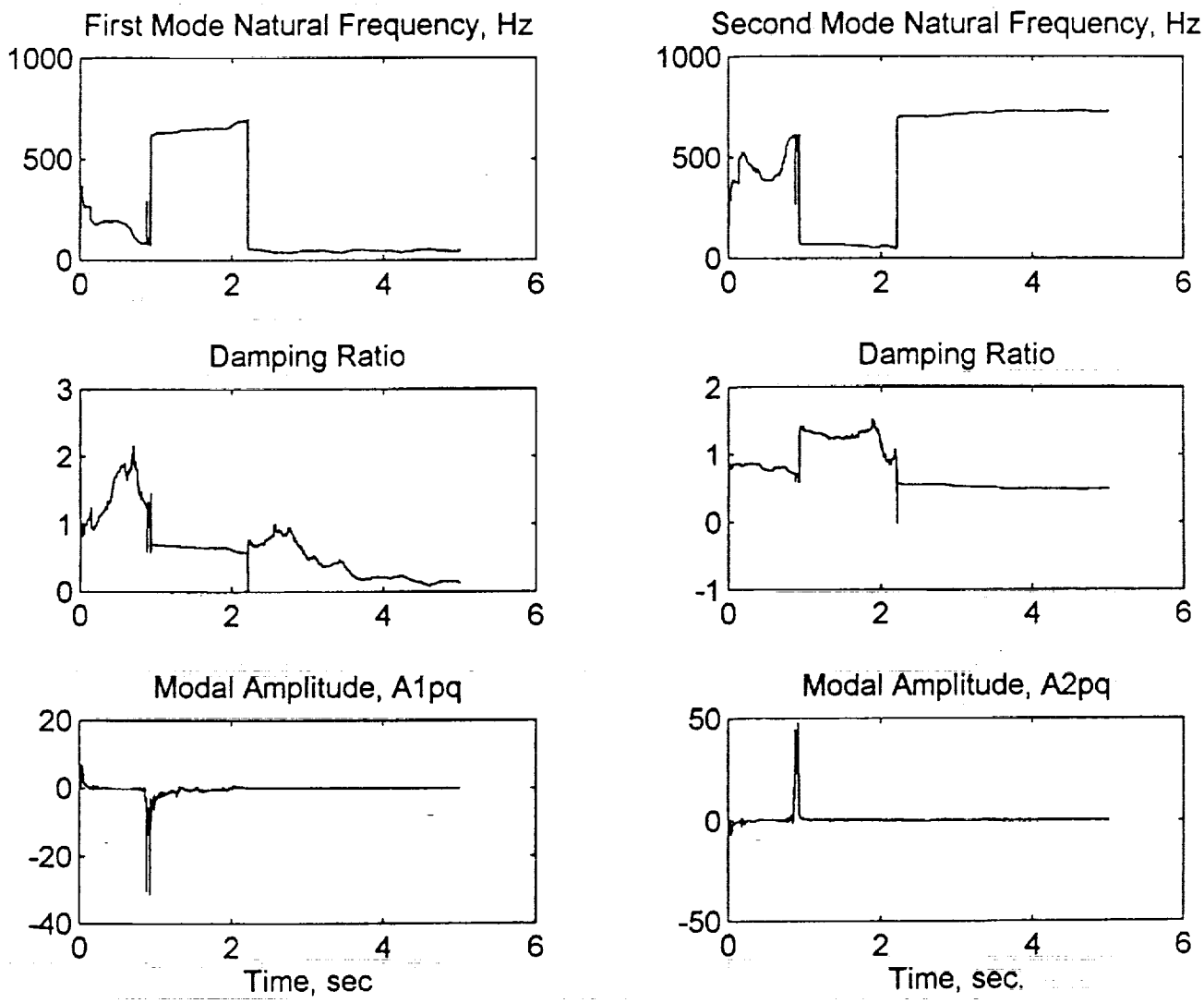

Figure 3.15a: On-Line Modal Parameters of the Simultaneous Two Mode Identification done in MATLAB
(Case 5: f_s = 1024 Hz, α = 0.05)

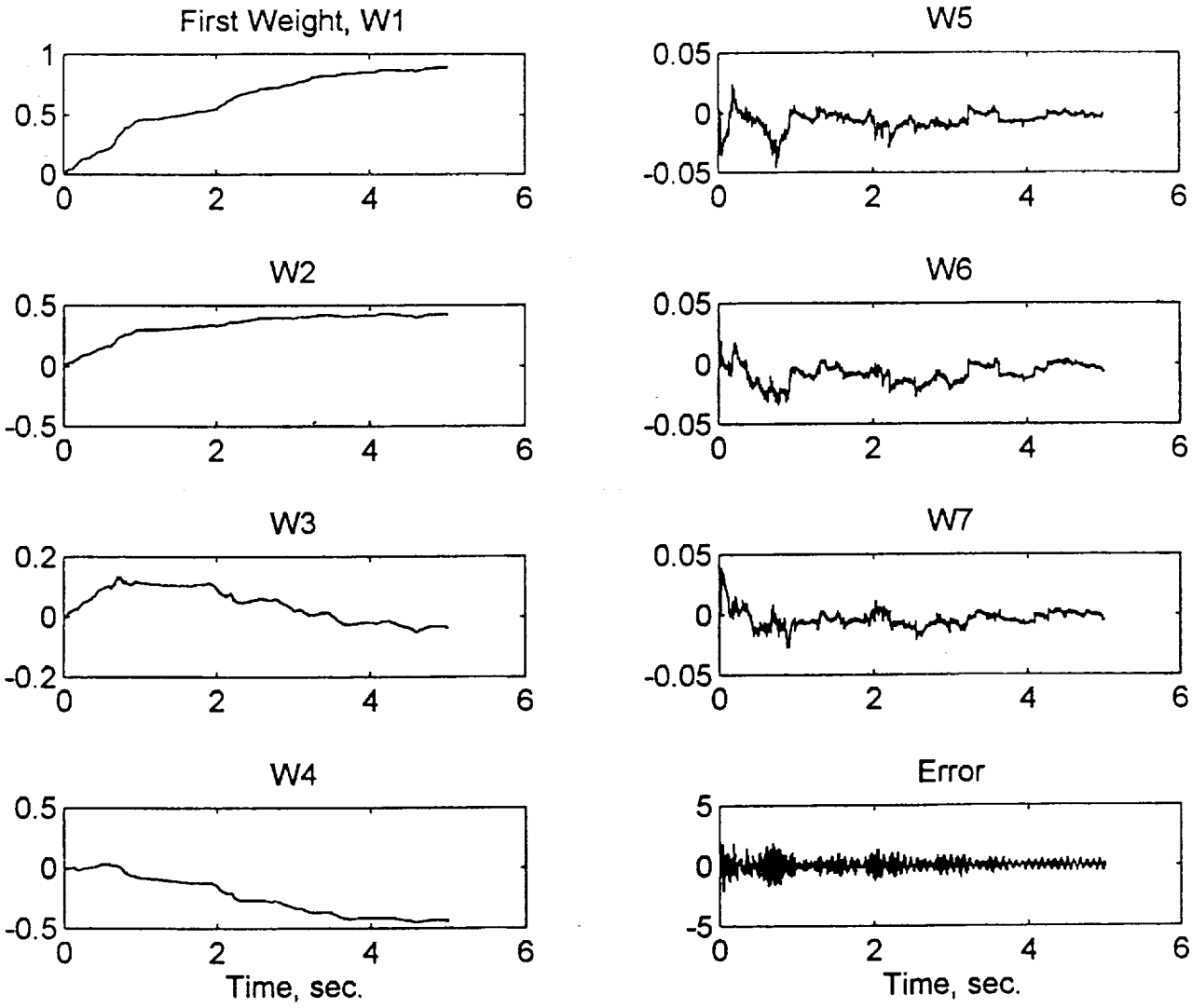

Figure 3.15b: Adaptive LC Weights and Error
(Case 5: f_s = 1024 Hz, α = 0.05)

basic formulation of the algorithm and the way it is applied is correct. However, it has been noticed that at the higher sampling rates, the algorithm in most cases totally skips the first mode, tries to identify the second mode and an additional mode that does not exist. For example, Figure 3.14a shows that the algorithm has identified the second mode natural frequency as the first mode and then tried to identify some higher mode as the second mode but finally came back to the actual second mode or something in its vicinity. It has totally skipped the first mode and assumed that the second mode is actually the first. In addition, Figure 3.14b shows that the algorithm did try to minimize the error especially at the end, but was unsuccessful mainly because it skipped the most obvious mode. Figures 3.15a,b show similar behavior. This phenomenon had been seen time and time again in simulation and in real-time testing which leads the author to the following possible conclusion.

The FFT plots of the simulated and actual system response always show the second mode not as a sharp clearly defined single frequency peak similar to the first mode, but as multiple peaks in the vicinity of the second mode frequency (Figure 3.10). Therefore, it can be assumed that once the sampling rate reaches a high enough value for the small magnitude frequency components in the vicinity of the second mode to become visible, then the algorithm totally ignores the first mode and tries to identify the second mode and something adjacent to it. The algorithm does not have the built-in decision making capability that could enable it to tell the difference between actual vibration modes and simple noise. When this mode-skipping phenomenon was noted, it was thought that if the above explanation is true, then bringing the two modes closer together should solve the problem, or at least improve the sensitivity of the algorithm to the sampling rate. When this was tested in simulation and in real-time, the results did not show any improvement and the same problem did occur.

A possible problem with the explanation given above as the cause of the mode skipping phenomenon is the single DOF case. In the on-line testing of the single DOF case, system identification took place without problems at sampling frequencies higher than 100 times the actual mode natural frequency. At such high sampling rates, the algorithm did not skip the first mode and identify a noise component in its place. However, this apparent contradiction might work in favor of the given explanation, because in the two DOF case, the adaptive filter has two coupled modes that influence and interact with each other and if the algorithm identifies only one of them, then the error would be reduced significantly which might mislead it into assuming that the goal is achieved. But the single DOF case has only one mode, which makes it difficult for the algorithm to try and minimize the error if this mode is totally ignored. As a result, the explanation given above seems like the most logical one. A possible solution to this problem could be the use of an adaptive learning rate (variable step size). This is discussed in the recommendations section (6.2).

4 The C40 Digital Signal Processing System*

The C40 DSP system used in the data acquisition and the real-time algorithm implementation is discussed in some details in this chapter. Most of the information contained in this chapter is taken out of the user's guides and manuals provided by Texas instruments and the third party vender, Spectrum Signal Processing Inc. [10-21]. As a result, the authors obtained written permission from both parties mentioned above to reproduce a limited amount of text, tables and figures.

**Reprinted by Permission of Texas Instruments and Spectrum Signal Processing Inc.*

4.1 Hardware

This section describes the various hardware components and connections that make up the C40 system used in this project. Figure 4.1 shows the two main boards that comprise the C40 system hardware, the QPC/C40B DSP Board showing a single TMS320C40 DSP processor in site A, and the PC/DMCB (smaller board) which has a single daughter module in site A. The 50-way ribbon cable connecting the two boards provides the DSPLINK interface. These are discussed in details in this section.

4.1.1 TMS320C40 Processor

The Texas Instrument's' TMS320C40 is a parallel floating point digital signal processor which is capable of 25 Million Instructions per Second (MIPS) performance if operating from a 50 MHz clock. It can attain a peak arithmetic performance of 275 Million Operations Per Second (MOPS). The total memory reach of the C40 is 4 Gbytes which contains program memory and registers affecting timers, communication ports and DMA channels. Memory space allocation will be discussed in more details in coming sections. Some of the main features of the C40 include [17]:

- Six identical communication ports for high-speed interprocessor communication which are capable of 20 Megabytes per second asynchronous transfer rate at each port, simple processor to processor communication, and bi-directional transfers for maximum flexibility. The TMS320C40 Parallel Runtime Support Library (PRSL) described in the software section (4.2.4) has several functions that allow the user to perform either synchronous communication port transfers which use the CPU to transfer data between memory and the six C40 communication ports or asynchronous communication port transfers that use direct memory access (DMA) autoinitialization and communication port flag synchronization mode for concurrent data input/output and CPU computation [20]. The asynchronous data transfer functions are the ones used mainly in the C code of chapter 5 to transfer data from the C40 to the PC.
- Direct memory access (DMA) coprocessor that supports six DMA channels that perform data transfers within the C40 memory map. This provides the advantage of moving data to and from the C40 memory without any CPU intervention. Each DMA channel is controlled by nine registers that are mapped in the C40 peripheral address space. The six channels transfer data in a sequential time-slice fashion rather than simultaneously due to that fact that they share common buses on the DMA coprocessor. The PRSL of section 4.2.4 also has several DMA functions written to assist the programmer is setting up the DMA channels for different transfer tasks. None of these functions are used in this project.
- High-performance CPU has a 40-ns instruction cycle time with 40/32-bit single-cycle floating point multiplier for high performance in computationally intensive algorithms.
- Two identical external data and address buses supporting shared memory systems and high data rate, single-cycle transfers.

Reference 15 is a very extensive user's guide of the C40 processor that has an overwhelming amount of information that can help any user utilize the full potential of this powerful processor.

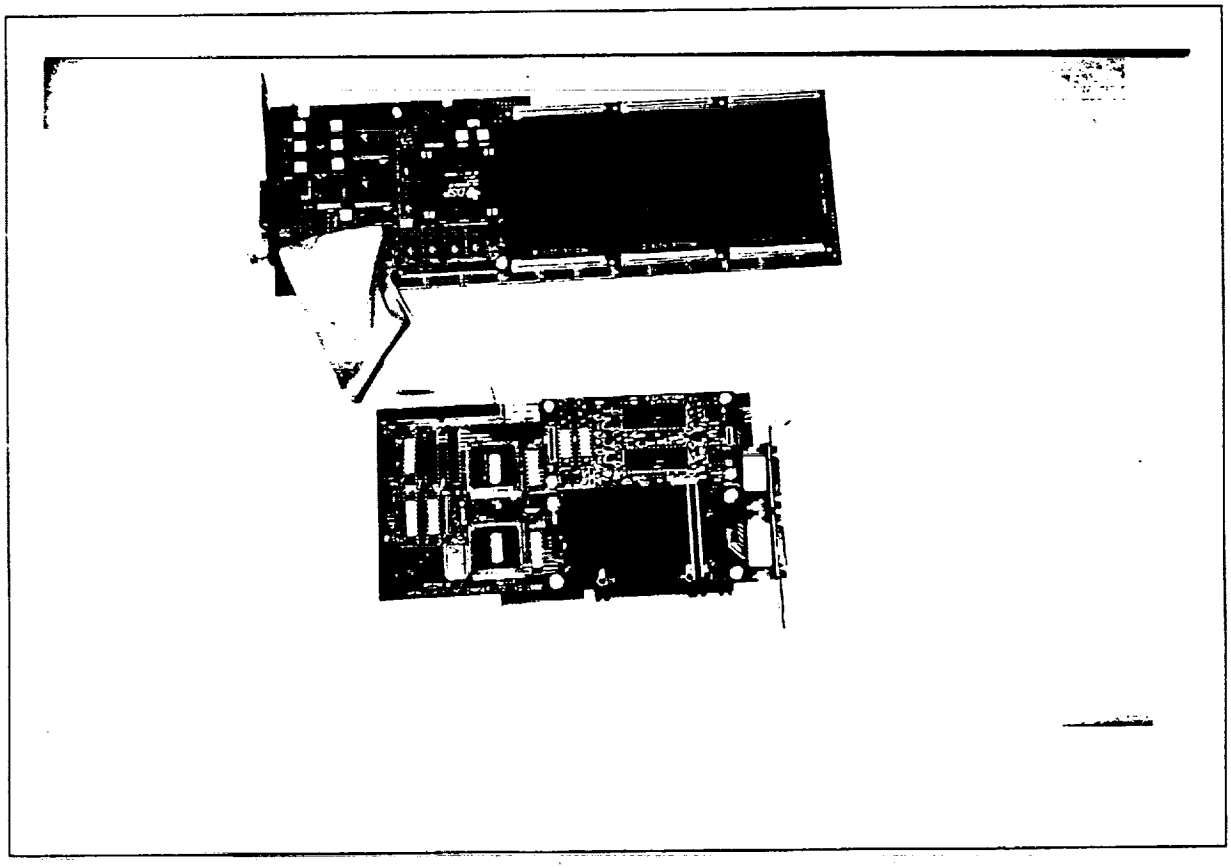

Figure 4.1: C40 DSP System Used in the Real-Time System Identification Test

However, the processor is used in this project in a very limited sense and a detailed discussion of its characteristics and possible applications is beyond the scope of this thesis. Any user of the C40 processor is highly encourage to refer to the user's guide in order to familiarize themselves with the potential power of this device. The third party documentation do not provide such a perspective.

4.1.2 MDC40S1 Tim-40 Module

The Texas Instruments' Tim-40 module is a standard hardware platform that consists of a TMS320C40 processor, memory and/or peripherals. The memory and peripherals are implemented according to the specific application at the discretion of the third party vendor, in this case the third party vender is Spectrum Signal Processing. However, the module's physical and electrical characteristics must conform to a standard specification. Such a module is the Loughborough Sound Images (LSI) MDC40S1 module which adheres to Texas Instruments' Tim-40 specification. It consists of one TMS320C40 processor operating from a 50 MHz clock, three banks (Bank 0, 1 and 2) of external 32K x 32 zero wait state Static RAM (SRAM), a 32K x 8 Programmable Erasable ROM (PEROM) to provide the identification ROM required by the Tim-40 module specifications and a clock oscillator. Figure 4.2a,b show the module layout and block diagram. The memory map of the module is shown in Figure 4.3. Banks 0 and 1 of the external zero wait state SRAM are located in the local memory of the module which along with the internal C40 Blocks 0 and 1 make up one continuous block in the memory map. This gives the user the power to work within a single memory block if the program is larger than any one individual block without having to jump from one block to another. This has proven to be a very useful tool in this project where some of the programs were required to store large arrays of data. In order to make this feasible, the External SRAM bank 1 (a.k.a. ERAM1) is not configured in the linker command file (Appendices B, C and D). Instead, the SRAM bank 0 (a.k.a. ERAM0) is configured as a 64K bank instead of the usual 32K to accommodate the large arrays of data. This means that in Figure 4.3, the ERAM0 bank will extend from address 0030 8000h to 0031 0000h. Then in the linker command file the **.data** section is forwarded to ERAM0, meaning that the arrays will be stored in that memory block.

The third external zero wait state SRAM bank (bank 2) is used as the on-module global memory. It is recommended that for optimum module performance, data should be restricted to local memory and executable code restricted to global memory. The TMS320 Floating Point C compiler produces six relocatable blocks of code and data, which are called sections. One of these sections is the **.text** section which is an initialized section that contains all the executable code as well as floating point constants. Therefore, in the linker command file, the **.text** section is sent to the global memory (ERAM2) as shown in Appendices B, C and D. Another section that has already been mentioned above, is the **.data** section which according to the recommendations must be restricted to local memory.

4.1.3 QPC/C40B QUAD C40 Board

The Loughborough Sound Images (LSI) QuadPC/C40B board shown in Figure 4.1 is designed to accommodate up to four Tim-40 modules for fast parallel processing and real-time embedded applications. It has a digital system expansion interface known as DSPLINK and a 16 bit PC interface making it suitable for PC AT and compatibles. Figure 4.4 shows a block diagram of the QPC/C40B board.

4.1.3.1 PC Interface

The PC interface is facilitated by two LSI C40 Network API libraries, which are high level language interface routines that allow the user to download and run code on the C40 from a PC program without having to deal with the low level details of the interface. This means that the user does not need to write code to access the PC interface directly [11]. The C40 NetAPI libraries are

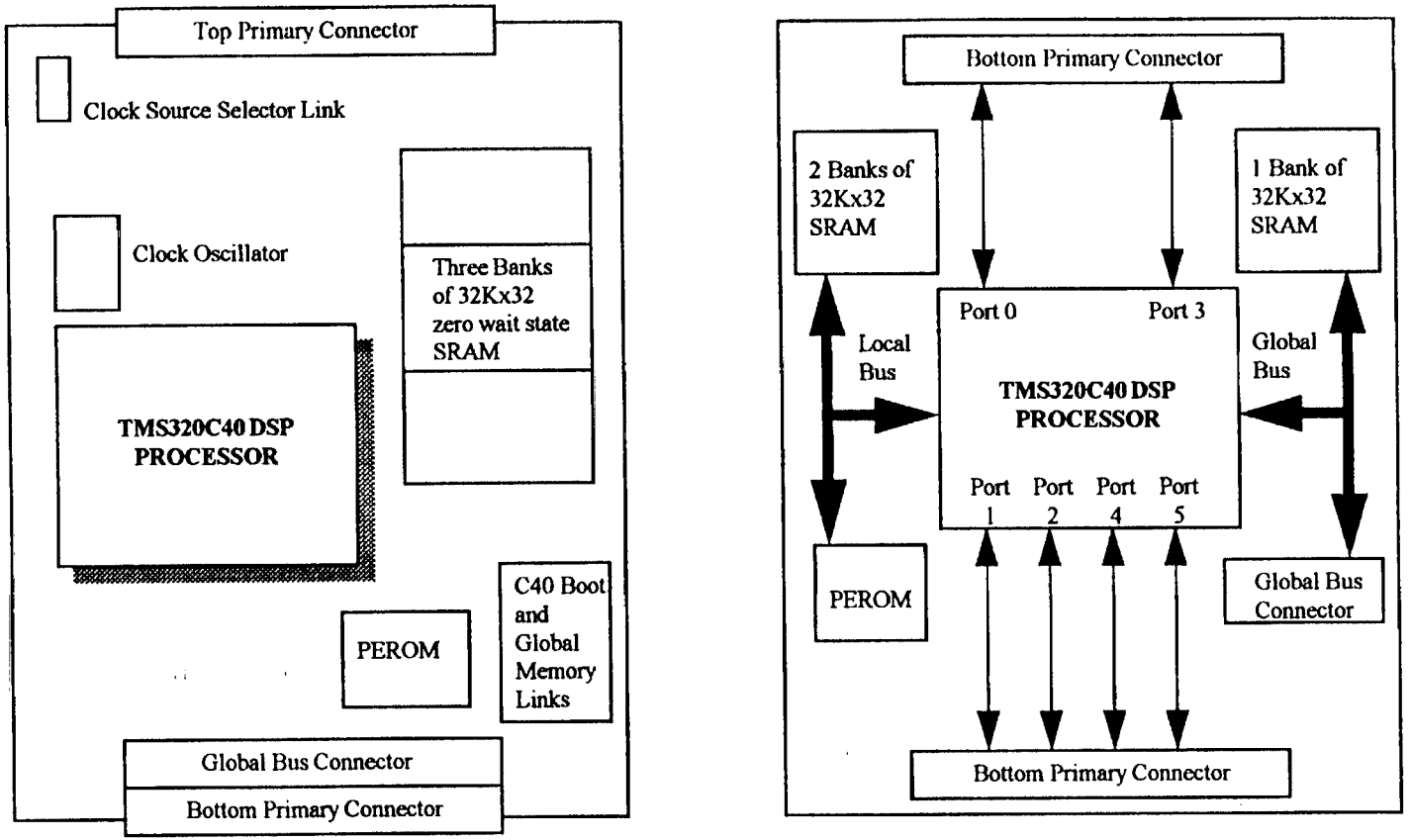

Figure 4.2: TIM-40 Module Layout and Block Diagram

(Reprinted by Permission of Spectrum Signal Processing, Inc. (Reference 10))

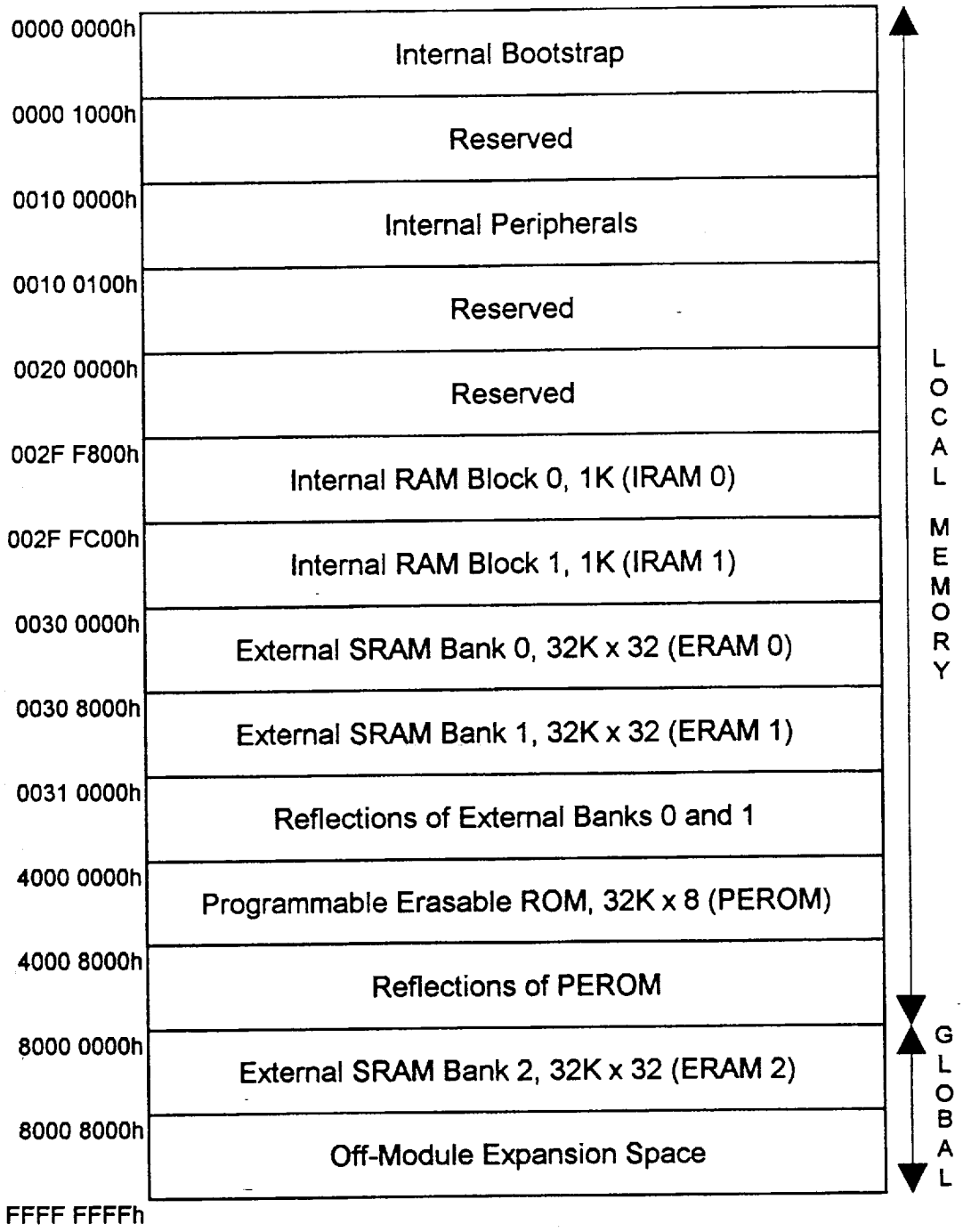

Figure 4.3: TIM-40 Module Local and Global Memory Map
(Reprinted by Permission of Spectrum Signal Processing, Inc. (Reference 10))

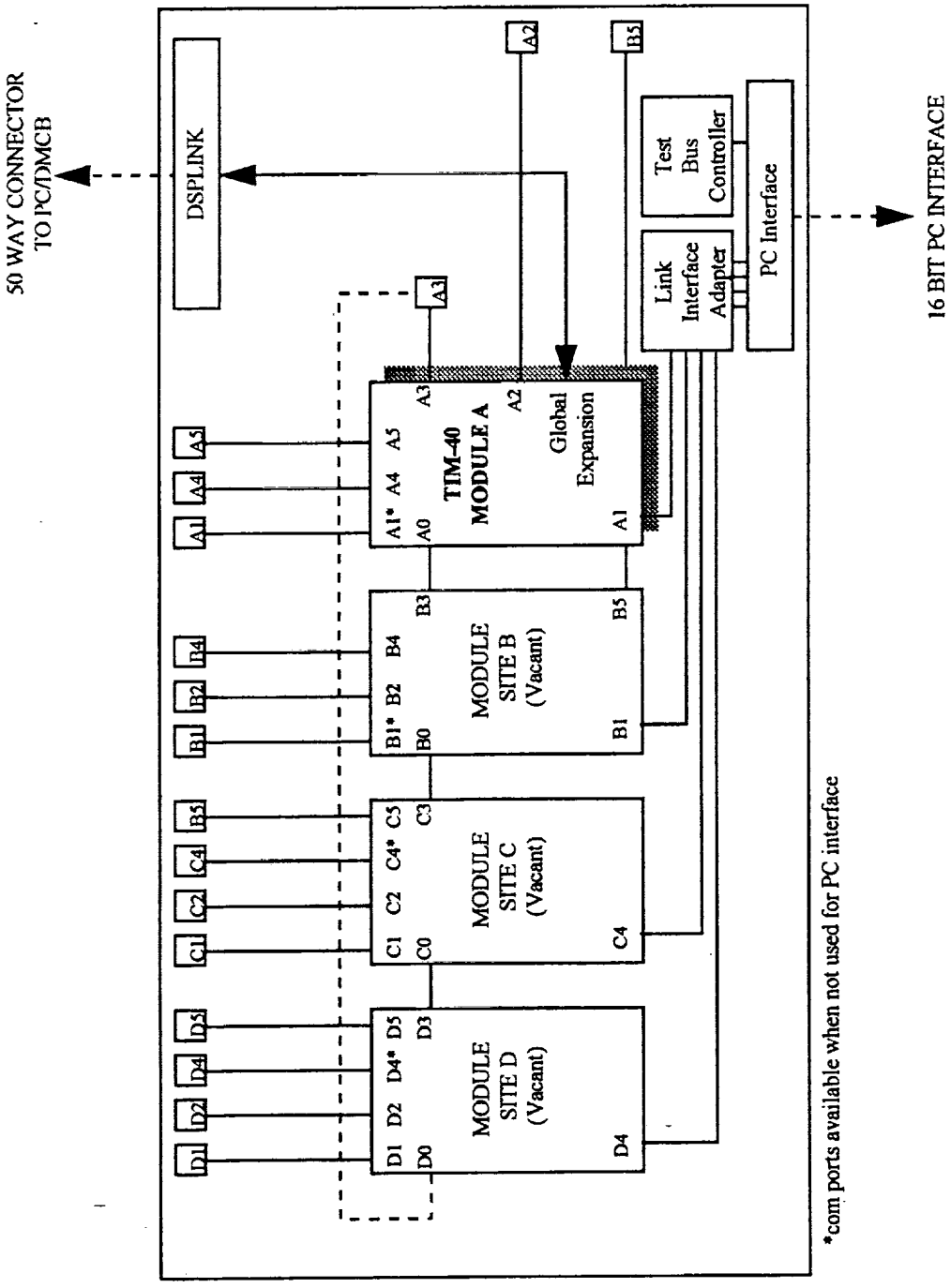

Figure 4.4: Quad PC/C40B Board Block Diagram As Used in This Project

(Reprinted by Permission of Spectrum Signal Processing, Inc. (Reference 11))

discussed in greater details in section 4.2.3. The PC interface is enabled/disabled by setting a jumper on a hardware link on the board, if the enable option is chosen then the communication ports A1, B1, C4 and D4 are used for PC interface and hence can not be used for interprocessor communication as shown in Figure 4.4. Disabling the PC interface means that the QPC/C40B board can be operated independently of the PC, which means that in a multiboard system, only the master board needs to be mapped in the PC I/O map and control of the other boards takes place through this master board. The communication between the QPC/C40B board and the PC takes place via three main routes:

1) Link Interface Adapter (LIA)
The LIA provides the main data exchange mechanism between the Tim-40 modules on the QPC/C40B board and the host PC. LIA is enabled by a hardware link on the board. When the PC interface is enabled as discussed in the previous section, communication ports A1, B1, C4 and D4 are routed directly through the LIA circuitry to the PC bus as shown in Figure 4.4. The LIA emulates the operation of the C40 communication ports allowing direct communication between the host PC and the Tim-40 modules [11]. Each of the Network API libraries contains several functions that allow data transfer to/from the C40 via the LIA. Some of these functions are mentioned in section 4.2.3.

2) PC Bus
The PC bus is the main control interface that provides access to various board functions such as interrupts and resets through the Control Register which is a software programmable register.

3) Test Bus Controller (TBC)
The TBC is an interface to the C40's JTAG-based scan path circuitry and functions as a full emulation system. It is used to implement the DB40 debugger and the Network API libraries. Accessing the TBC directly is not recommended [11], instead the user is encouraged to use the Network API libraries. JTAG is a debug port for the C40 that allows the debugger to peak/poke the memory without interfering with the CPU.

4.1.3.2 Digital System Parallel Expansion (DSPLINK)

DSPLINK is a high speed, bi-directional bus that allows data transfers to/from the C40 processor without using the input/output bus on the PC. It is mapped into the global memory map of the Tim-40 module in site A on the QPC/C40B board as shown in the block diagram of the board in Figure 4.4. The DSPLINK interface provides a high bandwidth, 32 bit, memory-mapped parallel expansion capability. The QPC/C40B implements the full 32 data lines and supports interrupt and wait signals on the DSPLINK 50 way connector. A slave board (PC/DMCB described below) communicates with QPC/C40B board via a 50-way shrouded connector. The DSPLINK interface is mapped into four spaces in the global memory map of the Tim-40 module in site A. Space 1 is accessed from addresses B000 0000h, Space 2 from B000 0100h, Space 3 from B000 0200h and Space 4 from B000 0300h (note that DSPLINK has a base address of B000 0000h). The four spaces are given to allow the user to operate the slave board at different speeds, Space 1 allows the fastest access and Space 4 the slowest. Also, in a multiple slave board configuration, each board must be located in a different area of the DSPLINK I/O space to prevent contention. In this project, Space 4 is used even though it is the slowest because it is the only one guaranteed to operate correctly with any LSI slave board. Each space consists of 256 locations [11]. A header file (**carrier.h**) is supplied with the QPC/C40B board that defines pointers to all of the DSPLINK interface registers for the PC/DMCB board. This header file (Appendix B) shows that Space 4 in the global memory map is being used. Note that the PC/DMCB slave board is a 16 bit peripheral whereas the C40 processor has a 32 bit data bus, so the slave board communicates via the upper half of the data bus (D16-D31). As a result, the data received by the processor has to be shifted down by 16 bits in order that it is read correctly by the C40. In the programming code, this is done using the >> operator right after the data is read from the channels. The code in Appendices

B, C and D shows such an operation at the beginning of the interrupt service routine (ISR). On the other hand, in order to configure the slave board (PC/DMCB) for operation by the C40 processor, a set of registers must be set up. The values for such registers must be shifted up 16 bits in order for the slave board to be able to read them. These registers and their functions are discussed in greater details in sections 4.1.4-6.

4.1.4 Crystal Analog Daughter Module (DM)

The Crystal DM is a 16 bit dual-channel delta-sigma I/O module that has a maximum sampling rate of 48 kHz. It is compatible with any LSI AMELIA-sited carrier board. In this case, it is fitted on a PC/DMCB in site A. The analog input/output signals are routed to/from the DM via the DM connector located at the backplate of the carrier board. The analog inputs to both channels have a maximum voltage span of +/- 2 volts, with 10 kΩ input impedance. Each input is referred to a separate ground to ensure signal integrity [13]. The analog inputs are inverted through a unity gain stage before being presented to the analog-to-digital converter. As a result, the values read by the C40 have to be inverted back (multiply by -1) to get the correct sign. This is done in the Interrupt Service Routine (ISR) after the 16 bit shift mentioned earlier. The analog outputs have a maximum output voltage span of +/- 2 volts and are referred to separate grounds. They are also inverted through a unity gain stage [13]. Both input channels of the DM are sampled at the same rate (synchronous sampling) which is derived directly from the system clock. Any of the following clocks can be chosen to be the system clock:

DMCLK_0 DM factory-fitted resident clock set at 12.288 MHz
DMCLK_1 . . . . . 6.144 MHz
MCLK_0 Clock derived from TCLK_0 PC/DMCB carrier board factory-fitted resident clock set at 6.144 MHz.
MCLK_1 Clock derived from TCLK_1 PC/DMCB carrier board factory-fitted resident clock set at 12.288 MHz.
EXTCLK_0 External user-specific clock routed to the DM through pin 8 of the DM connector.
EXTCLK_1 External user-specific clock routed to the DM through pin 9 of the DM connector.

The process of selecting the desired sampling rate for the DM is tedious and confusing. It involves setting up certain DSPLINK registers to configure the DM in order to select the type of system clock used, the prescaler, the prescale factors and finally the sampling rate. This process has been simplified by developing the final register configurations that are needed to produce certain sampling rates. These are listed below. Note that the possible sampling rates that can be generated on the DM without using an external clock source are 48, 44.1, 32, 29.4, 24, 22.05, 16, 14.7, 12, 11.025, 8, 7.35, 6, 5.5125, 4 and 3.675 kHz [13]. Some of these sampling rates might seem strange, but they are selected this way for a reason. Apparently, the Crystal Daughter Module was designed for audio applications. The sampling rates were selected to be the Nyquist frequencies for some of these applications. For example, the 44.1 kHz sampling rate is used for audio compact disks and the 48 kHz is used for digital audio tapes. For the purpose of this project, the registers were set up to produce selected rates that can be easily decimated down to desired values. These are shown as follows (Note that 0x in C code means a hexadecimal number).

Sampling rate = 4 kHz
DM Route Register: *DM1_ROUTE = 0x0000
AMELIA Control Register: *DM1_AMELIA_CONTROL = 0xB3 0000
and: *DM1_AMELIA_CONTROL = 0xF3 0000
User Control Register: *DM1_USER_CONTROL = 0xA8E0 0000

6 kHz
DM Route Register: *DM1_ROUTE = 0x0000
AMELIA Control Register: *DM1_AMELIA_CONTROL = 0xA3 0000
and: *DM1_AMELIA_CONTROL = 0xE3 0000
User Control Register: *DM1_USER_CONTROL = 0xA8E0 0000

8 kHz
DM Route Register: *DM1_ROUTE = 0x0000
AMELIA Control Register: *DM1_AMELIA_CONTROL = 0xB3 0000
and: *DM1_AMELIA_CONTROL = 0xF3 0000
User Control Register: *DM1_USER_CONTROL = 0xA8A0 0000

12 kHz
DM Route Register: *DM1_ROUTE = 0x0000
AMELIA Control Register: *DM1_AMELIA_CONTROL = 0xA3 0000
and: *DM1_AMELIA_CONTROL = 0xE3 0000
User Control Register: *DM1_USER_CONTROL = 0xA8A0 0000

16 kHz
DM Route Register: *DM1_ROUTE = 0x0000
AMELIA Control Register: *DM1_AMELIA_CONTROL = 0xB3 0000
and: *DM1_AMELIA_CONTROL = 0xF3 0000
User Control Register: *DM1_USER_CONTROL = 0xA860 0000

24 kHz
DM Route Register: *DM1_ROUTE = 0x0000
AMELIA Control Register: *DM1_AMELIA_CONTROL = 0xA3 0000
and: *DM1_AMELIA_CONTROL = 0xE3 0000
User Control Register: *DM1_USER_CONTROL = 0xA860 0000

32 kHz
DM Route Register: *DM1_ROUTE = 0x0000
AMELIA Control Register: *DM1_AMELIA_CONTROL = 0xB3 0000
and: *DM1_AMELIA_CONTROL = 0xF3 0000
User Control Register: *DM1_USER_CONTROL = 0xA820 0000

48 kHz
DM Route Register: *DM1_ROUTE = 0x0000
AMELIA Control Register: *DM1_AMELIA_CONTROL = 0xA3 0000
and: *DM1_AMELIA_CONTROL = 0xE3 0000
User Control Register: *DM1_USER_CONTROL = 0xA820 0000

Where

- **DM Route Register** is a 4 bit register. Each bit controls the direction and routing of one of the four possible system clock signals to and from the DM.
- **AMELIA Control Register** is an 8 bit register that is used to set up and control the Application Module Link Interface Adapter (AMELIA). These bits are used to select the system clock, reset/calibrate the DM, set the board in Master/Slave mode and select sample rate.
- **User Control Register** is a 16 bit register that is used to select clock source for prescaler, select prescale factor and enable system clock.

In the C40 code shown in the Appendices, these registers are set up before globally enabling the interrupts. The other DSPLINK registers that are listed in the **carrier.h** header file and that are used to configure the DM and the carrier board are [13,15]:

- **Reset Register** is a 16 bit register which when read resets all logic on the DM. This halts the operation of the DM immediately. So in order to resume operation on the DM module, it must be fully reconfigured which means that this register must be read before any of the other registers are set up (see code in Appendices).
- **Interrupt Mask Register** is a 3 bit register the once configured, allows the DM to interrupt the C40 via the DSPLINK interface under specific conditions. For example, setting bit 0 of this register will allow the DM to interrupt the C40 whenever the Input Data Registers are full, meaning a sample of data can be read. Similarly, setting bit 1 will cause interrupts to be generated when the Output Data Registers are empty, meaning a data sample has been delivered and the registers are ready for another sample to be output.
- **Interrupt Status Register** is a 3 bit read only register that displays the status of the pending interrupts. It must be read at the beginning of the Interrupt Service Routine (ISR) to clear pending interrupts. So when an interrupt is received by the C40, the first step in the C40 ISR must be to read the interrupt Status Register. Bit 0 will be high when the Input Data Registers are full and the data can be read. Bit 1 will be high to signal that the Output Data Registers are empty and the next data to be output may be written. Bit 2 should always be set 0.
- **Board Interrupt Status Register** is a 2 bit register for the carrier board that defines the interrupt status of each DM in a board with two DM's placed on sites A and B. This register is not used in this project since only one DM is available on site A of the carrier board.
- **Configuration Register** is 16 bit register that is used to unlock AMELIA and initiate a valid communication protocol between the carrier board and the DM. This is done by writing a configuration word (key) to this register. The KEY is B390h.
- **Timer1 Register**. A 16 bit, write only register that is generally used to define sample/timer options. This register is not used by this DM or carrier board.
- **Channel 0/1 Input Registers** are two 16 bit, read only data registers residing in the AMELIA. They are usually setup in the ISR to read data from the input channels 0/1 respectively every time the ISR is executed which is at the sampling rate. The contents of these registers can be obtained by dumping them into two separate variables using the assignment operator '='.
- **Channel 0/1 Output Registers** are two 16 bit, write only AMELIA data registers that deliver samples to the DM output channels 0/1 respectively.
- **Channel 2/3 Input and Output Registers**. Since the Crystal DM is a dual-channel board, these channels are not available.

Bit maps for these registers are documented in References [13] and [14] and they should be referred to if the user wishes to have a different sampling rate that is not listed above.

4.1.5 Application ModulE Link Interface Adapter (AMELIA)

The AMELIA is a programmable ASIC chip sited on the carrier board for each DM to provide an interface between the DM and the DPSLINK interface (Figure 4.5). The registers specific for this chip are discussed in the previous section.

4.1.6 PC Daughter Module Carrier Board (PC/DMCB)

The PC/DMCB is a general purpose DM carrier board that has two DM sites, A and B which can be used with a range of LSI's DSP boards supporting DSPLINK. Each DM site is supported by an AMELIA chip. External signals are routed to and from each DM site via its 26-way high

density DM connector on the carrier board. The DSPLINK interface to the C40 carrier board is provided with a 50 way connector as shown in Figure 4.5. Each DM is fully controlled and programmed from the C40 board via DSPLINK. If two DMs are available, then they can be operated either asynchronously (both DMs configured as masters), or synchronously (Master/Slave mode). However, for the purpose of this project, only one daughter module residing in site A is available. Figure 4.5 shows a board layout of the PC/DMCB. The 26-way high density DM connector has the pinout shown in Table 4.1. The PC/DMCB has two factory-fitted clocks, TCLK_0 and TCLK_1. These clocks are routed to the AMELIA chip on the board. They are software controlled and can be prescaled to generate master clock outputs, MCLK_0 and MCLK_1. The carrier board can be located in any of four different areas of the DSPLINK I/O space. This is achieved by setting a hardware link on the carrier board (LK3) to a specific base address. In this case it has been given a base offset address of 300h which will map it to space 4 on DSPLINK. The Carrier Board base address is determined by adding the DSPLINK base address in the C40 memory map and the carrier board offset address set by link LK3. This is shown in the carrier board memory map in Table 4.2 which also shows the control and data transfer registers for the DM.

4.2 Software

The software supplied by the C40 manufacturer and the third party vender that is used to configure and operate the C40 system is discussed in this section.

4.2.1 System Configuration

Once the C40 system is installed in the PC and all the necessary hardware links and connections are set up, the system is ready to be software configured before it is ready and operational. There are two configuration files that need to be generated, and they are the system configuration file (**netapi.cfg**) and the JTAG configuration file (**board000.cfg**). Reference [16] has detailed step by step procedure on the installation process of these files.

1) System Configuration File (netapi.cfg)
This is a text file that is generated by the user with the help of a utility program provided by the manufacturer. It contains a description of the C40 system used. This may be a simple single processor single board set up, or it could be a complex multi-board network. The information in the system configuration file is used to initialize various addresses and registers in the system. The DB40 debugger and the C40 NETAPI libraries obtain information about the system from this file. The file created for this project is shown in Appendix B and it contains information about the host system, the board type, host connection, control register, JTAG base address, LIA base address, module type, processor site, processor type, processor speed and processor memory map.

2) JTAG Configuration File (board000.cfg)
Each system that uses the JTAG emulation system either directly, via the DB40 debugger, or the functions in the development library must have one or more JTAG configuration file associated with it [16]. Each of these files describes a JTAG scan chain which has a listing of all the processors in the system or board that are connected in chain. Since only one board with a single processor is used in this project, the JTAG configuration file contains a single scan chain for a single processor, namely:

"CPU_A" TI320C40

4.2.2 C and Assembly Code Debugger (DB40)

The DB40 is LSI's version of the TI TMS320C40 C Source Debugger used to debug C40 C code, Assembly code or both. This is a very powerful and user friendly software that enables the user to

Table 4.1: Crystal Daughter Module Connector Pinout

Pin	Assignment	Description
1	GIN_B	Channel 1 input ground signal
2	GIN_A	Channel 0 input ground signal
3	GOUT_B	Channel 1 output ground signal
4	GOUT_A	Channel 0 output ground signal
5	UC1	Not Used
6	UC4	Not Used
7	COMMON	Common
8	EXTCLK_0	External user-generated clock signal
9	EXTCLK_1	External user-generated clock signal
10	AIN_B	Channel 1 analog input signal
11	AIN_A	Channel 0 analog input signal
12	AOUT_B	Channel 1 analog output signal
13	AOUT_A	Channel 0 analog output signal
14	UC0	Not Used
15	UC3	Not Used
16	COMMON	Common
17	DGND	Digital ground pin
18	EXTCONV_0	Clock signal used for slave DMs
19	COMMON	Common
20	COMMON	Common
21	COMMON	Common
22	COMMON	Common
23	UC2	Not Used
24	UC5	Not Used
25	DGND	Digital ground pin
26	EXTCONV_1	Clock signal used for slave DMs

Reprinted by permission of Spectrum Signal Processing, Inc. (Reference 14)

Table 4.2: Carrier Board Memory Map for Site A Daughter Module

CB Offset Address	DSPLINK2 Base Address	CB Base Address *	Read Register	Write Register
300h	B000 0000h	B000 0300h	Ch0 Data	Ch0 Data
301h	B000 0000h	B000 0301h	Reset	Timer1
302h	B000 0000h	B000 0302h	Ch2 Data	Ch2 Data
303h	B000 0000h	B000 0303h	Interrupt Status	Interrupt Mask
304h	B000 0000h	B000 0304h	Ch1 Data	Ch1 Data
305h	B000 0000h	B000 0305h	AMELIA Status	AMELIA Control
306h	B000 0000h	B000 0306h	Ch3 Data	Ch3 Data
307h	B000 0000h	B000 0307h	Not Used	Route/User Control/ Configuration

* Pointers to these locations are shown in the Carrier.h file in Appendix B

Reprinted by permission of Spectrum Signal Processing, Inc.(Reference 14)

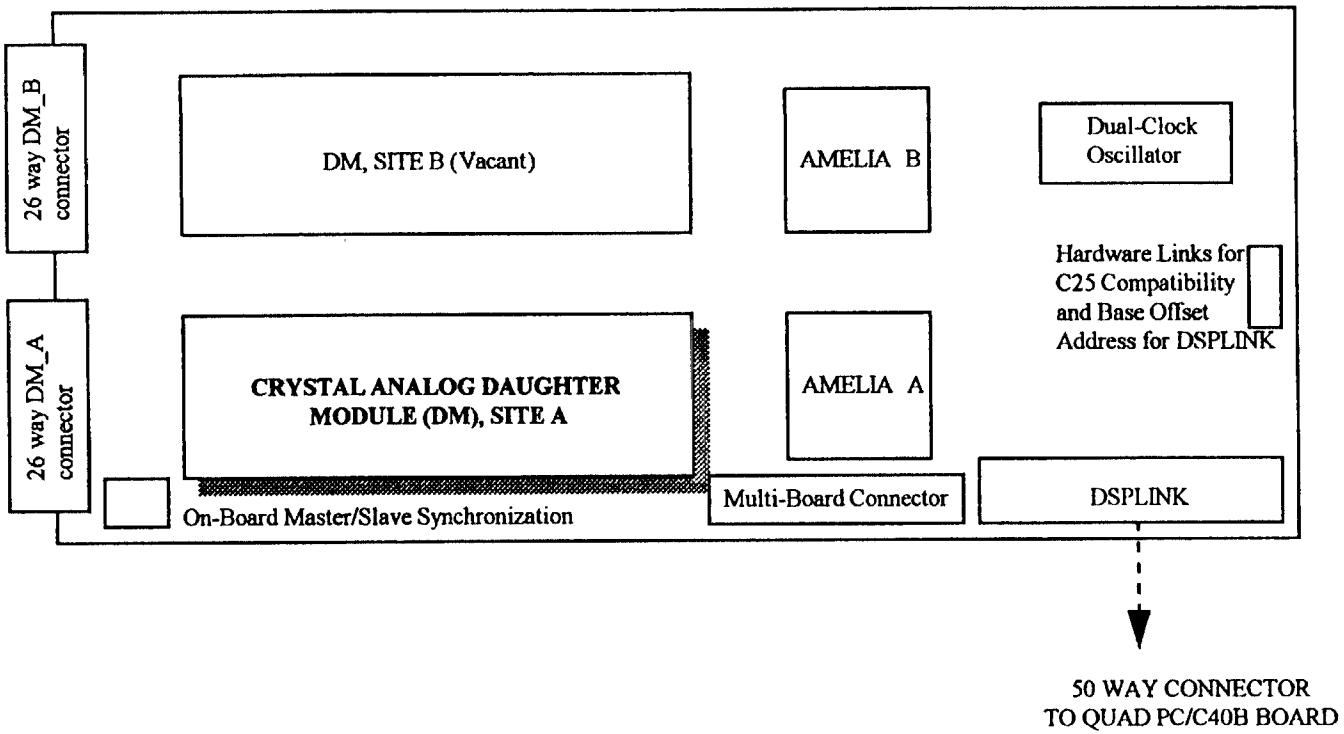

Figure 4.5: PC/DMCB Board Layout As Used in This Project

(Reprinted by Permission of Spectrum Signal Processing, Inc. (Reference 14))

download a program on the C40 system and perform high level as well as low level debugging operations. In addition to the programming code, CPU register contents and memory contents can optionally appear in separate adjustable windows. The comprehensive data display allows the user to create windows for displaying and editing the values of variables, arrays, structures and pointers in their natural format, whether float, integer, character or pointer [19]. Commands can either be entered at the prompt or using the mouse and menu at the top of the screen as shown in Figure 4.6. The process that a program has to go through in preparation for the debugger is simple. Once the code is written, it must be compiled using the **-g** compiler option which tells the compiler to produce symbolic debugging information and the **-v40** option which identifies the code as a C40 program and ensures that the intermediate assembly language code is produced for the right processor. Once the program is compiled, it must be linked using ,among other options, the **-o** option which tells the linker to create and name an output file (**filename.out**). This is the file that is loaded onto the C40 system either via the interface libraries of the host PC or in this case the debugger. At this point, the code is loaded into the DB40 software and is ready for debugging. Any variables or arrays that need to be monitored are called using the watch command (**wa**) or the display command (**disp**) respectively. Breakpoints can be set to control the code execution. The code can be run for a specific time, number of steps or up to a breakpoint. The latter is a very useful option that gives the user a lot of power in concentrating on the trouble spots in the code. If the step by step command is used, the debugger steps through the code one assembly instruction at a time, every time an instruction is executed, corresponding registers and variables are highlighted and updated accordingly. The code can be restarted as many times as necessary without leaving the software, all code initializations are done automatically every time the user requires a restart. The TMS320C4x C Source Debugger User's Guide [19] is an extensive documentation of this software and has description and examples of most of the commands and instructions needed to provide an extensive on-processor debugging operations. The discussion has been limited here due to the fact that the software is easy to use once the code is prepared and loaded into the debugger.

4.2.3 Network API Libraries

These are two high level language libraries, namely the development library and the application library, that contain a host of interface routines [16]. The C40 code development cycles usually involve using the Texas Instruments floating point tools to produce an executable program (**.out**) file which can be tested using the DB40 debugger as discussed above and downloaded to the C40 system using a host resident program. This PC resident program contains the interface routines of either library that can download the C40 code, start and halt it and allow C40-host interaction and data exchange.

4.2.3.1 The Development Library

In addition to the above features, the development library uses the JTAG system via the on-board Test Bus Controller to communicate with individual C40 registers and memory blocks. This library is provided in the file **c4xdev.lib** and can be compiled using either the Microsoft C Version 8.0 large memory model compiler or the Borland C Versions 3.1 and 4.0 compilers. A header file that contains declarations and definitions needed by the library such as structures, unions, enumerations and function prototypes is provided in the file **c4xdev.h** which must be included in the host program. However, since the extra features of this library are not needed for this project, its smaller and faster sister, the application library is used.

4.2.3.2 The Application Library

The application library is optimized for use in a host application similar to the development library and provides functions to allow code to be downloaded and run on any target processor in a multiboard system [16]. It also provides hand optimized routines for transferring data between the

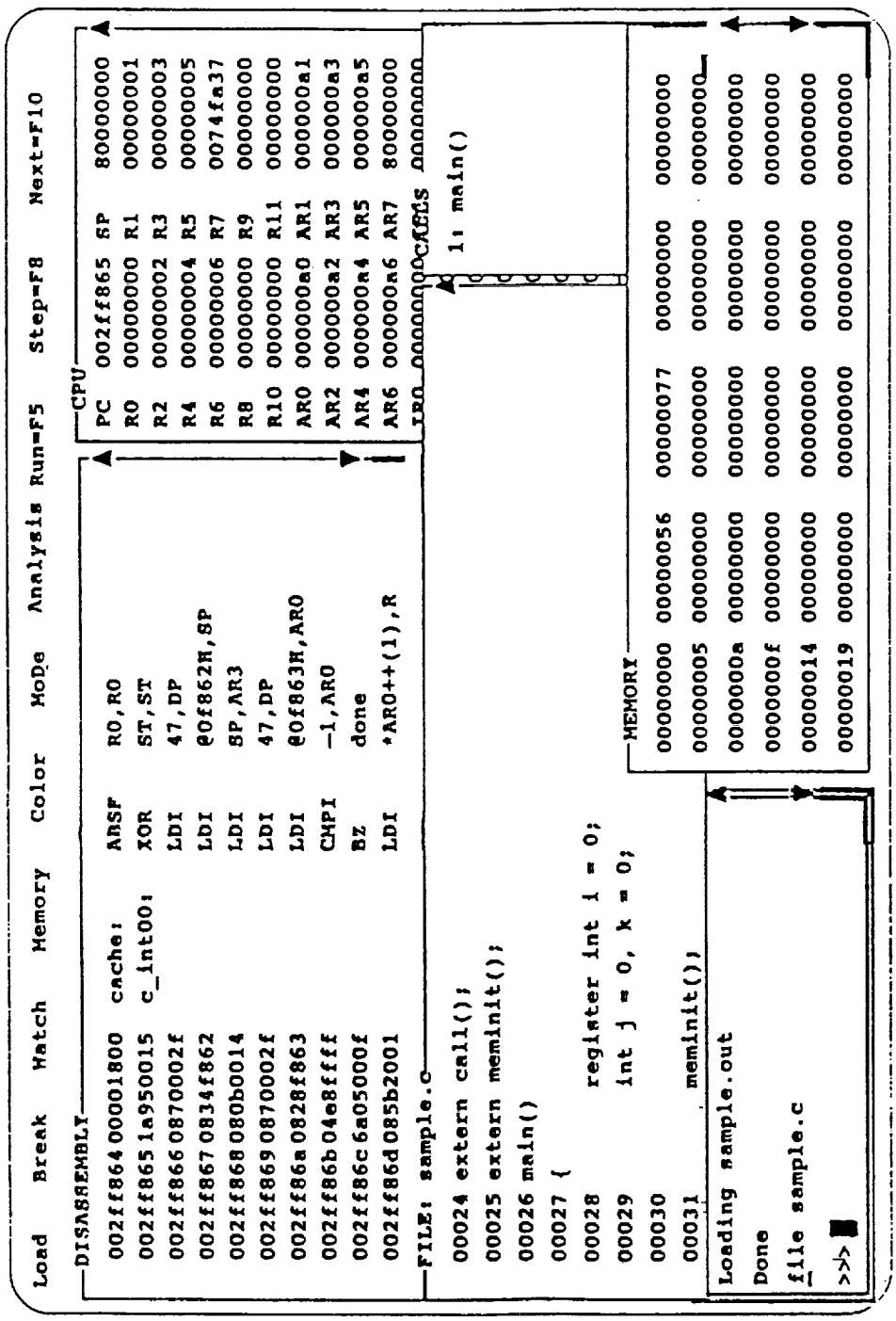

Figure 4.6: TI TMS320C4x C Source Debugger Sample Window in Mixed Debug Mode
(Reprinted by Permission of Texas Instruments, (Ref. 17))

host application and target processor. The code comes in the Microsoft C and Borland C versions. In this project, the Microsoft C compiler is used. As well as the standard DOS form, this library is also available in two Windows 3.1 compatible forms. The library is available in the **c4xapp.lib** and the associated header file is called **c4xapp.h**. The routines in this library provide access to the board registers and the Link Interface Adapter discussed above. The C40 executable (**.out**) file can be downloaded and executed. Easy C40-host interaction is also provided. Before using the Application library, the following preparations have to be done [16]:

- The system configuration file, **netapi.cfg** discussed in 4.2.1 must be present in the same directory as the host program.
- Some other files needed by the application library must also be included in this working directory. These are **edboot.out, edload.rom, c4xload.rom** and **boot.out** and are provided in the software package included with the C40 system.
- The application library header file **c4xapp.h** which declares all library functions using function prototypes must be included in the host PC code.

Once the above steps are completed, the final step after writing the host PC code is to compile and link it with the configuration file (**netapi.cfg**), the reader library file (**lmcfglib.lib**) and the Microsoft C version of the application library.
The application library has 20 different control, access and utility functions. The most used in this project are outlined below:

- **Utility Functions**
  These are common to both the development and application libraries.

  Global_Network_Reboot
  Used to set the carrier board and the Tim-40 module into a known state. This must be the first function called from the library and it must be followed by a function that loads the C40 code onto the processor. In this case, the **Load_And_Run_File_LIA** function is used.
  Open_Processor_ID
  Creates a processor handle for the specified processor. This handle is then used by the other functions within the library to access this processor.
  Close_Processor_ID
  Clears all memory taken by the processor specific information. The memory must be cleared at the end of the each session.
  Clear_All_Lib_Memory
  Clears all memory allocated during the work session by calling the Global_Network_Reboot function.

- **Link Access Functions**
  Once communication with the C40 processor is established and the processor is rebooted and is ready for operation, the following functions download and run the C40 code and provide the C40-host interaction [16]:

  Load_And_Run_File_LIA
  Loads an executable object file (**.out**) to the processor specified in the function call parameters. It then sets up the internal C40 Global and Local Memory Interface Control Register values before starting the program.
  Read_LIA_Words_32
  Reads a block of 32 bit words from the C40 via the LIA. This function is used here in junction with the **send_msg** function which is discussed below in the PRSL (section 4.2.4). It is used to read the size of the data array that will be sent by the **send_msg**

function from the processor. A function that reads a block of 32 bit floating-point numbers from the C40 must be used right after this function to receive the array sent by the **send_msg** function whose size has already been read by **Read_LIA_Words_32.**

Read_LIA_Floats-32
Reads a block of 32 bit floating-point numbers form the C40 via the LIA and places it in temporary storage. This function has the added useful feature of automatically converting the data from the C40 format to IEEE format [16]. This function is used here to read a fixed size array sent by the **send_msg** function from the C40.

Examples of how these functions are used are shown in the host code in Appendices A, B and C. The return code error messages for these functions are included in the **CHKERROR.C** file shown in Appendix B.

4.2.4 TMS320C40 Parallel Runtime Support Library

This library provides support and standard method programming for the C40 digital signal processor peripherals such as the six direct memory access channels and the byte-wide communication ports via the C programming language. These peripherals are controlled through memory-mapped registers that are accessed easily through assembly or C language. The PRSL contains well over 100 functions and macros which are categorized as communication port, DMA, interrupt, multiprocessor and timer functions and macros. The library itself is provided as a source code along with a header files and must be built before files can be linked to it [20]. Several building options are available and the one chosen for this project is the small memory model option using the stack-passing parameter convention with a level 2 optimization and header file installation [16]. The C40 code is linked to the PRSL in the linker command file (**.cmd**) as shown in the Appendices. The appropriate header files, such as **intpt40.h** or **compt40.h**, must be included in the C40 code before any functions from that specific section of the library can be called. In this application, only some of the communication port and the interrupt functions are used. These are listed below:

- **Interrupt Functions**
  Used to tell the processor programs how to handle various tasks according to their priority. They provide access to the vector table and the CPU interrupt registers.

  INT_DISABLE
  This is a macro that resets bit 14 (Global Interrupt Enable, GIE) of the C40 status register (ST) to globally disable all C40 interrupts.
  set_ivtp
  Sets up the interrupt-vector-table pointer (IVTP) register to the address specified in the function argument. An interrupt table contains interrupt vectors. An interrupt vector is an address of an interrupt service routine that should start executing when an interrupt is received. If the DEFAULT option is used, then the IVTP pointer is set to the **.vector** section in the linker command file.
  install_int_vector
  This function sets up the interrupt service routine address into the section where the IVTP register points. So if the **.vector** section in the linker command file is allocated a default address, then the **set_ivtp** function sets the IVTP to point to the top of the stack and the **install_int_vector** function puts the timer 0 interrupt service routine address in that memory location plus a user specified offset. Therefore, when timer 0 interrupt occurs, the processor branches off to the interrupt service routine. In this application, the C40 system interfaces with the PC/DMCB and as a result the interrupts are provided by the PC/DMCB system as discussed above at the desired sampling rate. In other words, the interrupt service routine which in this case is a subroutine that contains the data acquisition and system

identification code, is executed at the sampling rate set by the PC/DMCB system to provide real-time system identification.

load_iif
Sets the IIOF1 pin to be level trigger interrupt. The status of the external pins IIOF(3-1) indicates where to find the source program to be loaded (memory or communication ports)

INT_ENABLE
A macro that sets bit 14 (GIE) of the C40 status register (ST) to globally enable all interrupts.

- **Communication Port Functions**
  Used mainly for data transfers between memory and the six C40 communication ports or the DMA autoinitialization [20].

  send_msg
  This function sets up a DMA to send a word array that is pointed to by a data array pointer to a specified communication port channel. The operation of this function is asynchronous to the CPU operations after the setup, meaning that the CPU can be used in parallel with the data transfer. It is important to note that this function sends the number of data words to be delivered as the first item in the stream. Therefore, if the host program is expecting an array from a specific communication port, it must first read the number of words to be read, then the actual data array. This has been mentioned in the Application Library section above.

  out_msg
  Similar to the send_msg above but is CPU controlled.

The C40 code in Appendices B, C and D contain examples of these functions and macros and the context in which they are used.

4.2.5 Compiling and Linking the C40 and Host C Code

Two batch files have been written to compile and link the C40 and host C codes. These are named **comp_dsp.bat** and **comp_pc8.bat** respectively and they are available in the working directory for the C40 system.

4.2.5.1 C40 Code

The C40 code is compiled and linked using the following commands and options. These and other compiler and linker options can be found in Reference [18] in more details:

```
cl30 -s -g -v40 -alxs file.C
lnk30 file.CMD
```

cl30 is the command that invokes the compiler and assembler

-s invokes the interlist utility, which interlists C source statements into the compiler's assembly language output. This option automatically invokes the **-k** option which instructs the shell to keep the assembly language file (**filename.asm**). This assembly language file becomes useful if the assembly code for any C statement needs to be inspected or if the number of assembly instructions in the interrupt service routine (ISR) needs to be counted to ensure that the total number of clock cycles does not exceed the time between interrupts. The latter option is important if the code that needs to be executed in the ISR is long. The user needs to ensure that this code can be executed entirely before the next interrupt comes in, which would be skipped if the code is still running, and the result will be a false sampling rate. For example if the PC/DMCB was set to sample at 4 kHz, which means that the entire ISR code would be executed 4000 times per second, then the total time

allowed for the ISR to execute before the next interrupt comes in is 250µs. Since this particular processor has a 50 MHz, then the duration of a clock cycle is 0.02µs, which means that the total number of clock cycles the ISR is allowed to have is 12500 cycles. This number should not be exceeded, otherwise a false sampling rate would result. There is an easier way of approximating the number of clock cycles a portion of C code takes, and that is by using the **?clk** command in the DB40 Debugger. However, this command does not necessarily give the correct number, and if the code is large, it is safer to manually count the assembly instructions in the **.asm** file.

-g tells the compiler to generate symbolic debugging directives that are used by the DB40 Debugger.

-v40 specifies the target processor, in this case the TMS320C40 processor.

-alxs an assembler option that invokes the assembler to produce an assembly listing file, produce a symbolic cross-reference in the listing file and retain labels.

file.C the file that contains the C40 source code.

-lnk30 the command that invokes the linker.

file.CMD the linker command file that contains the linker options, standard memory configuration and section allocations into memory (see **.cmd** files in Appendices B, C and D). These are:

-x forces rereading of libraries if unresolved symbols are not found.

-c enables ROM autoinitialization.

-o *file* generates the executable **.out** file that is loaded onto the C40.

-m *file* generates a map file of the input and output sections.

-i *dir* directs the library search algorithm to look in *dir* for the Run Time Support Library.

-l *lib* links Parallel Runtime Support Library that is located in *dir*.

-e *global_symbol*
defines a *global_symbol* that specifies the primary entry point for the output module. Must be c_int00 if the DSP code is C source files.

MEMORY {}
the linker, not the compiler, specifies the standard memory configuration (memory map) for the C40 carrier board. See section 4.1.2 for more details.

SECTIONS the compiler usually produces several blocks (sections) of code and data that are relocatable and can be allocated in memory in various ways to conform to the user specified application. Once these sections are created by the compiler, the linker takes over and allocates these sections into target memory. For example, the linker can be instructed to place all global variables (**.bss** section) into fast internal RAM or allocate executable code into internal ROM. The following sections are used in this application [21].

.text contains all the executable code and floating-point constants.

.data contains tables of data or preinitialized variables.

.vectors contains the interrupt vector table which must lie on a 512-word boundary (see linker command files).

.cinit contains tables with the values for initializing variables and constants.

.bss reserves space for global and static variables. At program startup, the C boot routine copies data out of the **.cinit** section and stores it here.

.stack allocates memory for the system stack which is used to pass arguments to functions and to allocate local variables [18].

Other sections are discussed in Reference [18].

4.2.5.2 PC Host Code

The PC host code is compiled and linked using the following command and options [22].

cl /W4 /O1 /AL /F 4000 file.C netapi.lib readlib.lib

cl	invokes the Microsoft C Version 8.0 C compiler and linker.
/W4	Sets warning level 4. Tells the compiler to display the least severe level of warning messages.
/O1	This option affects the optimizing procedure that the compiler performs. The **/O1** option produces very small executable files that will run on most machines.
/AL	Selects large memory model. A program code and data are stored in blocks, the memory model of the program determines the organization of these blocks. In addition, the memory model determines the type of executable file that is generated by the compiler, the large memory model generates an **.exe** file [22].
/F *size*	Sets the program stack size to the number of bytes specified by size, where size is a hexadecimal number in the range 0001 to FFFF. For programs in this application, a value of 4000hex (16K) is used. If the program issues a stack-overflow error, then the user might try to increase the stack size. Reference [22] indicates that the maximum stack size accepted is for greater than 64K.
file.C	C source file that contains the host PC code.
netapi.lib	Links the code with the specified NetAPI library. In this case the application library is used, so the **c4xapp.lib** is linked.
readlib.lib	links the code with the reader library associated with the specific NetAPI library used. In this case, the **lmcfglib.lib** is used.

4.3 Remarks

This chapter is meant to provide a basic description of the specific C40 system used in this project and its operation to help the reader understand the process involved in the real-time code implementation and consequently appreciate the problems associated with such an undertaking. In addition, it is hoped that this chapter will provide the next user of the C40 system a good starting point and aid them in picking up the learning curve in a short period of time. This could save a great deal of time because the C40 system used here consists of modules and components from two different manufacturers and a third party vendor. Each module and piece of software comes with its own documentation in a fragmented format to accommodate a wide range of applications. As a result, there are 12 manuals and user guides that came with this system which had to be related to one another to provide the knowledge required to use the system accurately and efficiently, and that is what this chapter is designed to deliver. It is by no means a comprehensive description of the entire system and should not be taken as such, but rather a starting point that could make the learning process a lot smoother. The author feels that the documentation provided here along with the example programs included in the appendices is sufficient to use the C40 system in this type of application. If different applications or an extension of this application are required, then the user is advised to refer to the original manuals.

5 Real-Time System Identification

The adaptive filter and algorithm that have been developed in chapter 2 and simulated analytically in chapter 3 are tested in real-time in this chapter. The first two sections describe the equipment used in the lab and the experimental setup. The last section details the real-time system identification process and the results obtained.

5.1 Equipment Used

This section details the equipment used in the lab to test the system identification algorithm in real-time.

5.1.1 Computers

Two computers are used. A Compaq Deskpro 575 PC (computer #1) with a 75 MHz pentium processor and 16MB RAM is the main computer used for the experiment. It houses the C40 system and its accompanying software. A Microtech PC (computer #2) that has a LabView input/output AT-MIO-16X I/O board is used to provide the excitation signal for the shaker. The AT-MIO-16X board is a 16 channel, high-performance, multi-function analog/digital input/output board [23]. It is driven by the LabView Virtual Instrumentation software (VI). A VI is created to serve as a random signal generator. This is used to provide the driving signal for the shaker via the LING STAR amplifier.

5.1.2 LING STAR 1.0 Power Amplifier

The STAR 1.0 is an audio amplifier that has been reconfigured for vibration and modal testing use. It has an output power of 1.4 kVA, output voltage 120 V_{rms} (nominal), output current 12 A_{rms} (nominal), frequency response 2 Hz - 20 kHz and voltage gain of 56 [24]. The power amplifier is used to drive the shaker which delivers the excitation force to the structure.

5.1.3 LING LMT-50 Modal Shaker

The LING shaker is an electrodynamic transducer capable of producing a total sine vector force rating of 50 pounds [25]. It has a useful frequency range of DC to 2 kHz and a maximum rated stroke limit displacement of 1.2 inches peak-to-peak. It is capable of an 80g maximum acceleration and 60 in/s maximum velocity. Its fundamental resonance frequency is at 3500 Hz.

5.1.4 PCB Piezotronics Model 280A02 Force Transducer

The PCB force transducer is designed to measure compressive, tensile and impact forces over a dynamic range of 10 to 500 lbs. It can withstand a maximum compression of 100 lbs, maximum tension of 500 lbs and has a resolution of 0.002 lbs [26]. It has resonant frequency at 70 kHz and a 10 μsec rise time.

5.1.5 PCB Piezotronics Model Q353B33 Quartz Shear mode ICP Accelerometer

The PCB 353 quartz shear mode accelerometer has a quartz sensing element housed in a titanium casing and weighs only 25 grams. It offers high performance for precision acceleration measurements and utilizes an integrated circuit piezoelectric electronics [27]. It has a frequency range of 0.07 to 7000 Hz, a mounted resonance frequency of 22 kHz, an axial maximum shock

limit of +/- 2000g and a -65 to 250 °F operating temperature range. The accelerometer and force transducer must be powered with a PCB approved constant current power/signal conditioners, such as the one listed next.

5.1.6 PCB Piezotronics Model 483B18 Six Channel Line Power Voltage Amplifier

This is a well regulated 24 VDC or 105-125 VAC driven six channel power unit with a continuous gain range of 1 - 100. It provides constant current excitation to the ICP transducers (in this case, the force transducer and accelerometer mentioned above). The front panel has the gain adjustment dials and the fault monitoring switch. The back panel has BNC jacks for both input and output connections. It has a low frequency response of 0.05 Hz and a high frequency response of up to 200 kHz. The output voltage is +/- 10 volts and the output current is +/- 1 mA with an output impedance of 50 Ohms [28].

5.1.7 Test Structure

The main structure used in testing is an aluminum beam with a uniform rectangular cross section. It has the following properties:

Overall Length:	L = 48 in
Cross Section Area:	A = 0.75 in^2 (0.5x1.5 in)
Modulus of Elasticity:	E = 10.3x106 lb/in^2
Mass Density:	ρ = 0.098 lb/in^3
Moment of Inertia:	I = 0.0156 in^4

5.1.8 The Texas Instrument/Spectrum C40 System

This system and the accompanying software were discussed in detail in chapter 4.

5.2 Experimental Setup

The equipment used is laid out as shown in Figures 5.1 and 5.2. A connection is made from the AT-MIO-16X board to the back of the LING STAR 1.0 power amplifier. This connection carries the random signal generated by the LabView VI to the amplifier. The amplifier has two modes of operation, a Voltage mode and a Current mode. The Current mode was selected because it is recommended by the manufacturer for modal analysis and modal excitation [24]. A power cable connects the amplifier to the shaker which is mounted vertically on a wooden workbench. A wire stringer is attached to the shaker at one end and at the other end it has the force transducer which in turn is securely attached to the structure. The structure is clamped at one end to an adjustable steel platform as shown in Figure 5.2 and is free to vibrate at the other end. The accelerometer can be attached to the structure at any point desired. Two standard 10-32 (model 002C10) general purpose coaxial cables are used to connect the sensors to the signal conditioner. These cables have a coaxial plug at one end that connects to the sensor [27], and terminate in a BNC plug that hooks up to the back of the signal conditioner (input channel). Two oscilloscope probes are used to take the output of the signal conditioner to a breadboard that in turn is attached to the input/output channels of the C40 system by a ribbon cable.

5.3 Testing Procedure

The following steps describe the testing procedure used in the lab

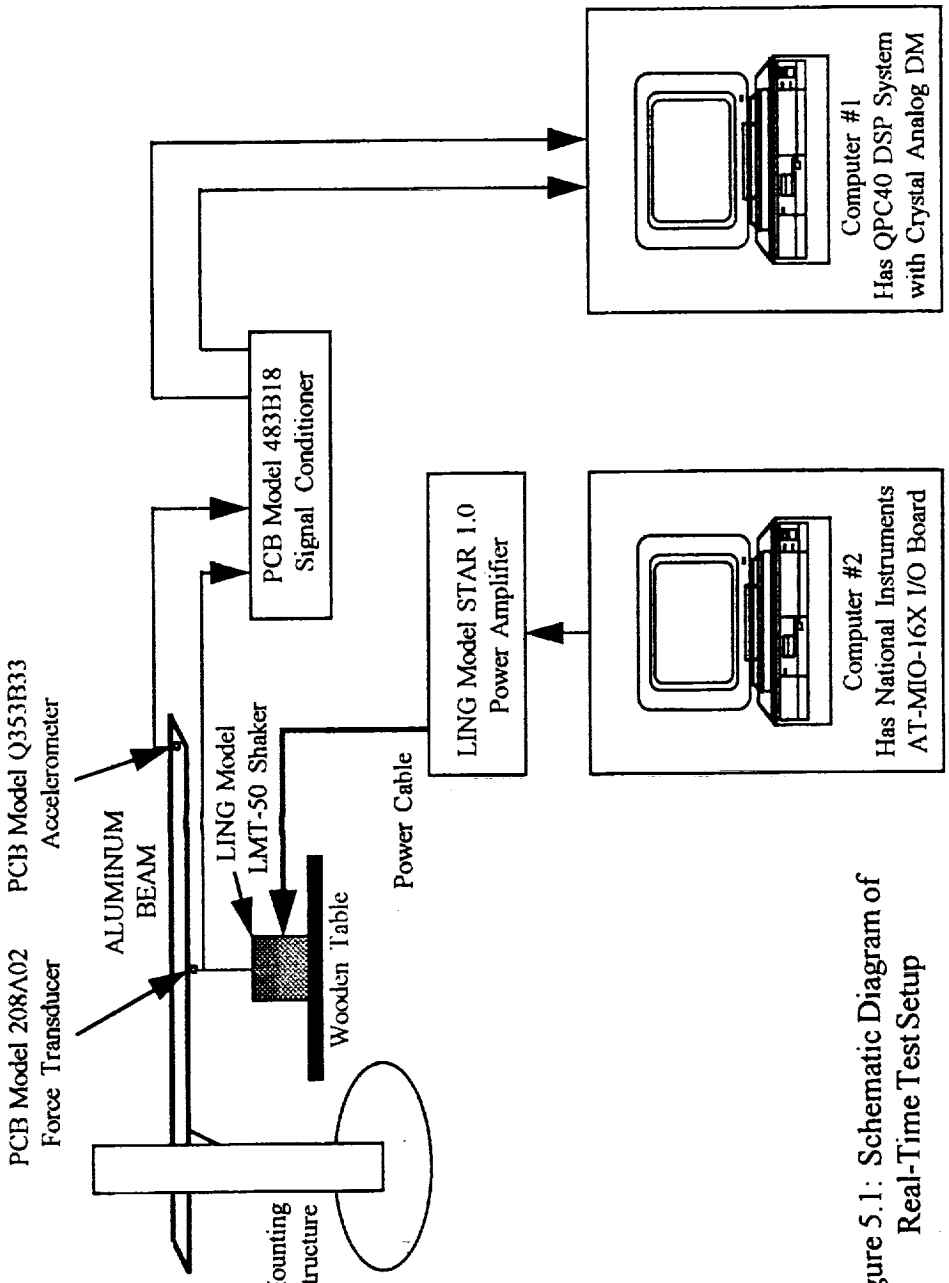

Figure 5.1: Schematic Diagram of Real-Time Test Setup

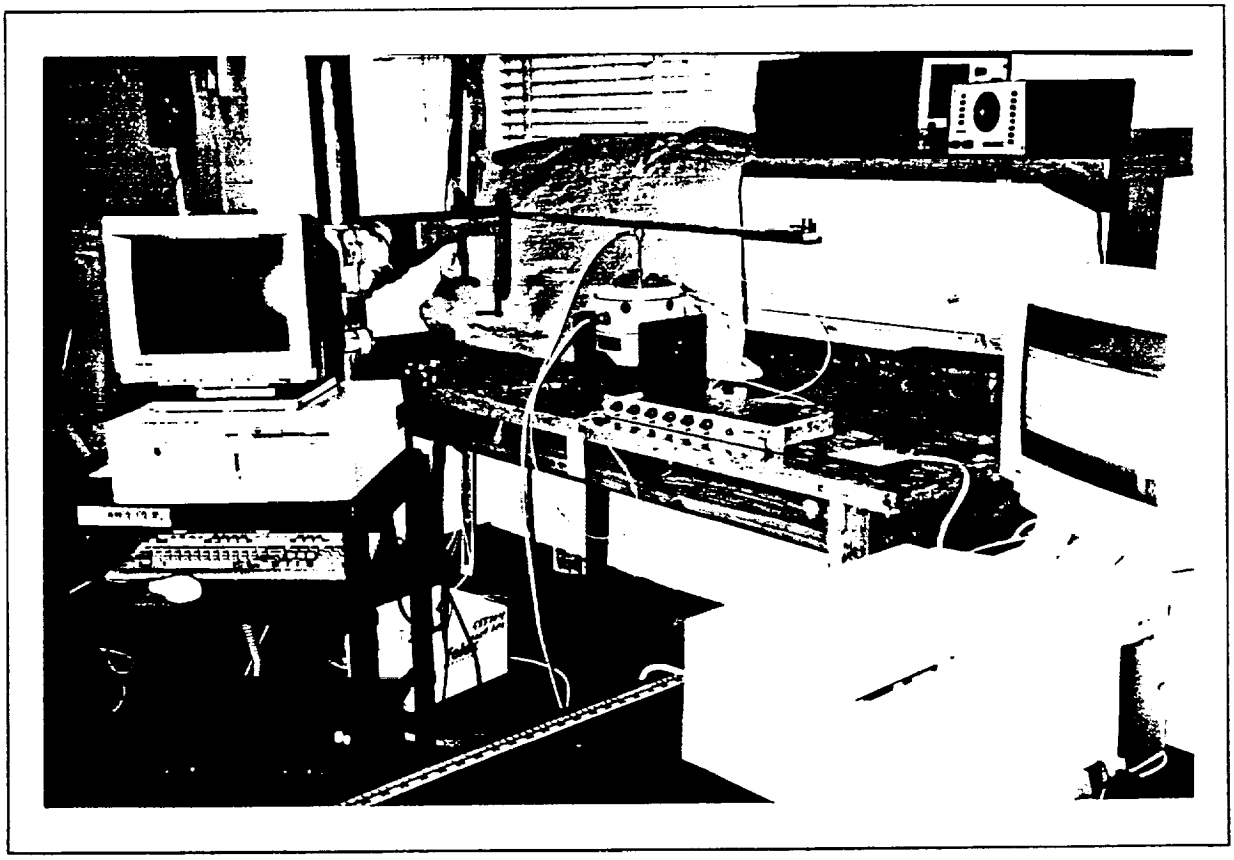

Figure 5.2: Equipment Setup Used in the Real-Time System Identification Tests

1. Write, compile and link C40 and host PC code on computer #1.
2. Prepare the C40 code for the DB40 debugger and load into debugger.
3. Mount the structure on the platform using clamps at the desired length. Load the random signal VI (**UWNoise.VI**) on computer #2, then select the desired sampling frequency and number of points. Also choose the amplitude of the random signal and whether a low-pass filter is to be applied to the signal or not. If the user wishes to only excite certain modes of the structure, then it is recommended that the low-pass filter is activated in the VI and a cutoff frequency selected as needed.
4. Ensure that the gain knob on the STAR power amplifier is turned off. Failing to do this could damage the shaker and the wire stringer.
5. Turn the power amplifier on, and run the random signal VI. Then slowly turn the gain on and increase until the shaker is imparting the desired force to the structure.
6. In the DB40 debugger, choose the **Run** command to download and start the C40 code on the processor. Halt the program, set break points and step through the commands to debug the program and ensure that it is performing the expected tasks. If not, then halt the debugging operation, free the processor using the **runf** command, quit the debugger and reduce the gain on the power amplifier to stop the shaker. Go back to the code and make the necessary changes then repeat the above steps until the debugging operation is complete.
7. Quit the debugger and modify the C40 code to interface with the host PC code.
8. Repeat steps 4 and 5, but this time download the C40 code onto the DSP using the host PC code. The system identification process will terminate when the arrays containing the results are filled up. This time duration depends on the size of the arrays, the sampling rate and whether time decimation is used or not.
9. Start MATLAB and run the M-file (e.g. **C40SDOF.M**) that is associated with the specific testing done. MATLAB will display the final system identification results graphically.

5.4 Real-Time System Identification

This section details the results of the real-time testing of the system identification algorithm developed in Chapter 2 using the equipment and procedures described above. But first a discussion on the problem of aliasing and possible solutions are presented.

Aliasing

In general, minimizing the sampling rate of a discrete signal means minimizing the arithmetic involved. In addition, for this project minimizing the sampling rate also means more code can be executed in the ISR before the next sample comes through, allowing the user more time to perform many tasks. However, if the input signals are not band-limited as in most real-life cases, then a lower sampling rate will cause the higher frequencies to be aliased down and appear in the lower frequency band [7]. This problem is usually avoided by the use of an anti-aliasing filter which is an analog low-pass filter that usually has its cutoff frequency at one-half the desired sampling frequency, thus forcing the input signals to be band-limited in the that region. As a result, any higher frequency components are filtered out and do not alias down to the lower frequency band after sampling.

Unlike most data acquisition systems used nowadays, the C40 system does not come with a built-in anti-aliasing filter. Therefore, an analog low-pass filter was designed and built to be used as an anti-aliasing filter but turned out to be a very crude device that did not deliver the satisfactory response. A recommendation was made by an Electrical Engineering Professor at the School to use the oversample-filter-decimate method [29]. The basic idea is to sample at a high frequency that will capture the entire frequency band where significant information is present, then use a low-pass digital filter to isolate the band of interest, and finally decimate to get the needed sampling frequency for the computations. Decimation is basically throwing away every other sample to bring down the sampling rate to a desired value.

Theoretically, it is assumed that a structure has an infinite number of vibration modes that could appear in the response signal if excited. This assumption invalidates the oversample-filter-decimate method because no matter how high the sampling rate is, there are always higher modes which will alias down into the lower band. Fortunately, in reality this effect can be reduced in magnitude if the higher modes are not well excited. Then, if a sampling frequency at least two times higher than the highest frequency mode that is significantly excited is used, then the entire frequency band of significance is captured and little aliasing takes place. Reducing excitation of higher modes is done by applying a low-pass digital filter on the excitation signal so that only the modes of interest are well excited. Digital filtering after sampling the signals can then be used to isolate these modes. But, low magnitude noise at higher frequencies is always present in the signal and gets aliased down into the lower band. Unfortunately, without an analog anti-aliasing filter, nothing can be done to eliminate this problem and it has to be tolerated. Tests have shown that good results can still be obtained even though such noise is always present in the signals.

Although the oversample-filter-decimate method has been found to work well as shown in the following sections, it does have a disadvantage associated with it. A higher sampling frequency will always mean less code in the interrupt service routine (ISR), thus limiting the number of tasks that can be done. In this project, a sampling frequency of 4 kHz has been found to be high enough to capture the significant information in the signals and avoid aliasing, yet it is still low enough to allow significant amount of computation in the ISR.

Digital Low-Pass Butterworth Filter

The input signals in most test runs in this project were sampled at 4kHz or higher, and a fourth order low-pass digital Butterworth filter has been used to filter out the higher unwanted frequencies. Butterworth filters are Infinite Impulse Response (IIR) filters, that have a maximally flat amplitude response in the pass-band [7]. A fourth order filter has the following pulse transfer function:

$$\frac{Y(z)}{X(z)} = \frac{c_0 + c_1 z^{-1} + c_2 z^{-2} + c_3 z^{-3} + c_4 z^{-4}}{1 - d_1 z^{-1} - d_2 z^{-2} - d_3 z^{-3} - d_4 z^{-4}}$$

Cross multiplying and taking the inverse z transform gives the filter difference equation:

$$y(k) = d_1 y(k-1) + d_2 y(k-2) + d_3 y(k-3) + d_4 y(k-4) + c_0 x(k) + c_1 x(k-1) + c_2 x(k-2) + c_3 x(k-3) + c_4 x(k-4)$$

where x is the filter input (unfiltered signal), and y is the filter output (filtered signal); $x(k)$ is the present sample of the signal and $x(k-1)$ is the previous sample (one tap delay) and so on. The filter coefficients ($d_1, \ldots, d_4; c_0, \ldots, c_4$) are obtained using the butter command in MATLAB which designs a digital Butterworth filter of a desired order given a sampling frequency and a cutoff frequency. A sample fourth order filter is shown below, which was designed at a sampling rate of 4 kHz and has a cutoff frequency of 75 Hz to isolate the first two modes of the beam. The filter is implemented in the ISR as shown below. This is part of a sample code given in Appendix C.

- Once the input signals A (accelerometer), F (force transducer) are sampled, the tap delays (previous values) for the first signal are set up.

1) First Signal, A

```
xa4=xa3;
xa3=xa2;
xa2=xa1;
```

```
xa1=xa;
xa=A;
ya4=ya3;
ya3=ya2;
ya2=ya1;
ya1=FA;
```

- Then the filter is applied to the first signal as follows:

```
FA=3.692234261*ya1- 5.123180777*ya2 + 3.165651468*ya3 -
0.734870850*ya4 + 1.0e-04*( 0.103686192*xa + 0.414744770*xa1 +
0.622117156*xa2 + 0.414744770 *xa3 + 0.1036861926 *xa4);
```

- and the tap delays for the second signal are

2) Second Signal, F

```
xf4=xf3;
xf3=xf2;
xf2=xf1;
xf1=xf;
xf=F;
yf4=yf3;
yf3=yf2;
yf2=yf1;
yf1=FF;
```

- now the filter is applied to the second signal,

```
FF= 3.692234261*ya1- 5.123180777*ya2 + 3.165651468*ya3 -
0.734870850*ya4 + 1.0e-04*( 0.103686192*xa + 0.414744770*xa1 +
0.622117156*xa2 + 0.414744770 *xa3 + 0.1036861926 *xa4);
```

Figures 5.3 and 5.4 show the beam acceleration response signal without and with filtering, respectively. The FFT plot in Figure 5.4 shows how the filter has effectively eliminated all modes and noise components that are higher than 75 Hz, thus isolating the first two modes of vibration.

System Identification

Once the excitation and response signals are sampled and filtered, a counter is set to decimate the signals in time to easily establish an effective sampling rate without having to reconfigure the DM registers or introduce an external clock source. This is done for two reasons: the C40 preset sampling rates are limited and too high for this application, so time decimation allows the user to select a lower rate by simply selecting a desired counter value. The counter is used in an if statement to throw out (decimate) unwanted samples. The second reason for time decimation is the oversample-filter-decimate method discussed above. The system ID code is included within the if statement as shown in the sample codes in Appendices B, C and D.

5.4.1 Single DOF Case (Single Mode)

For the single mode ID, the beam was clamped to the steel platform at a 40 inch length. The excitation point was chosen to be near the center of the beam and the response was taken at the tip as shown in Figure 5.1. The excitation point was chosen so that at least the first two modes will be well excited.

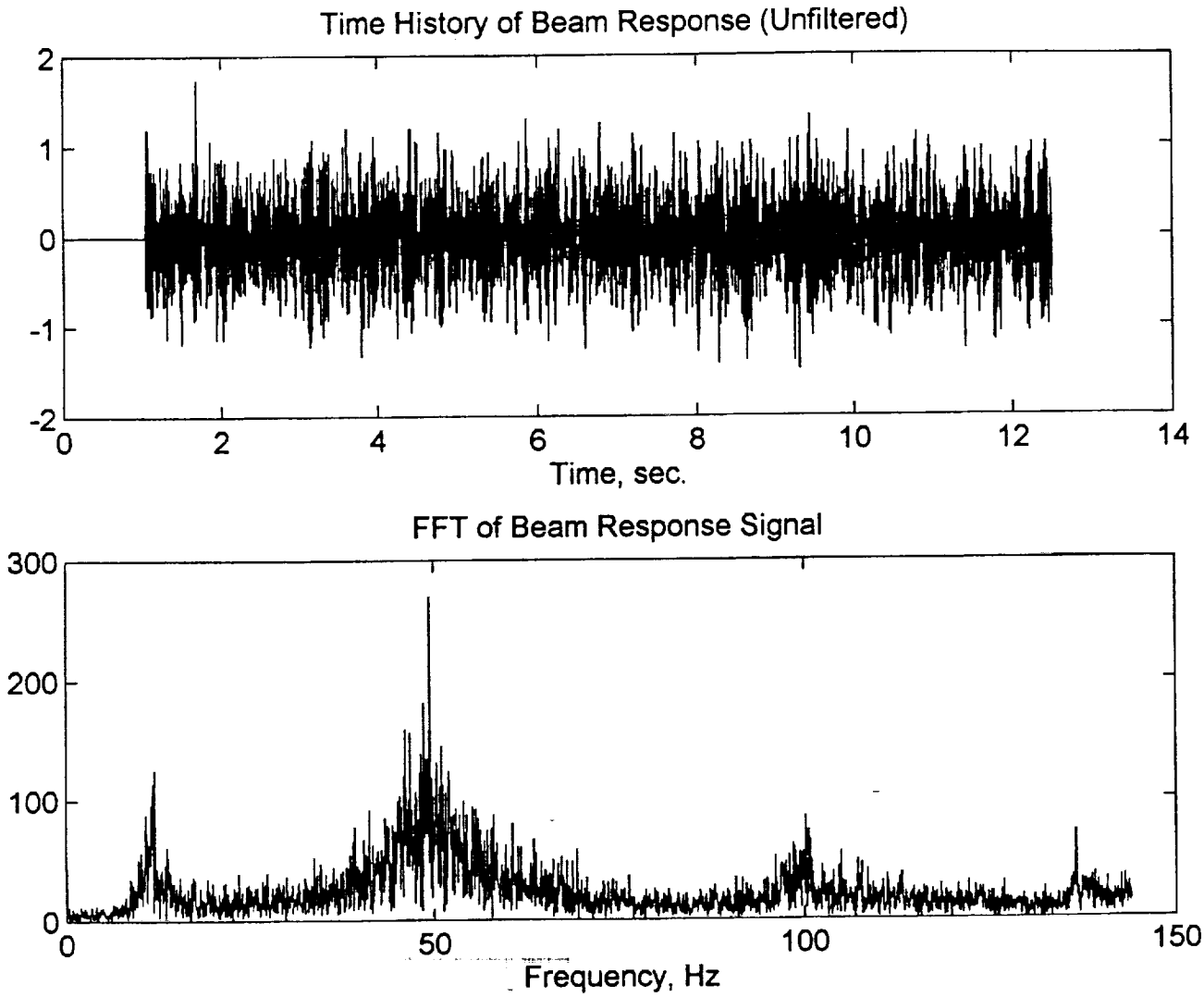

Figure 5.3: Aluminum Beam Unfiltered Acceleration Response and its FFT (f_s = 500 Hz)

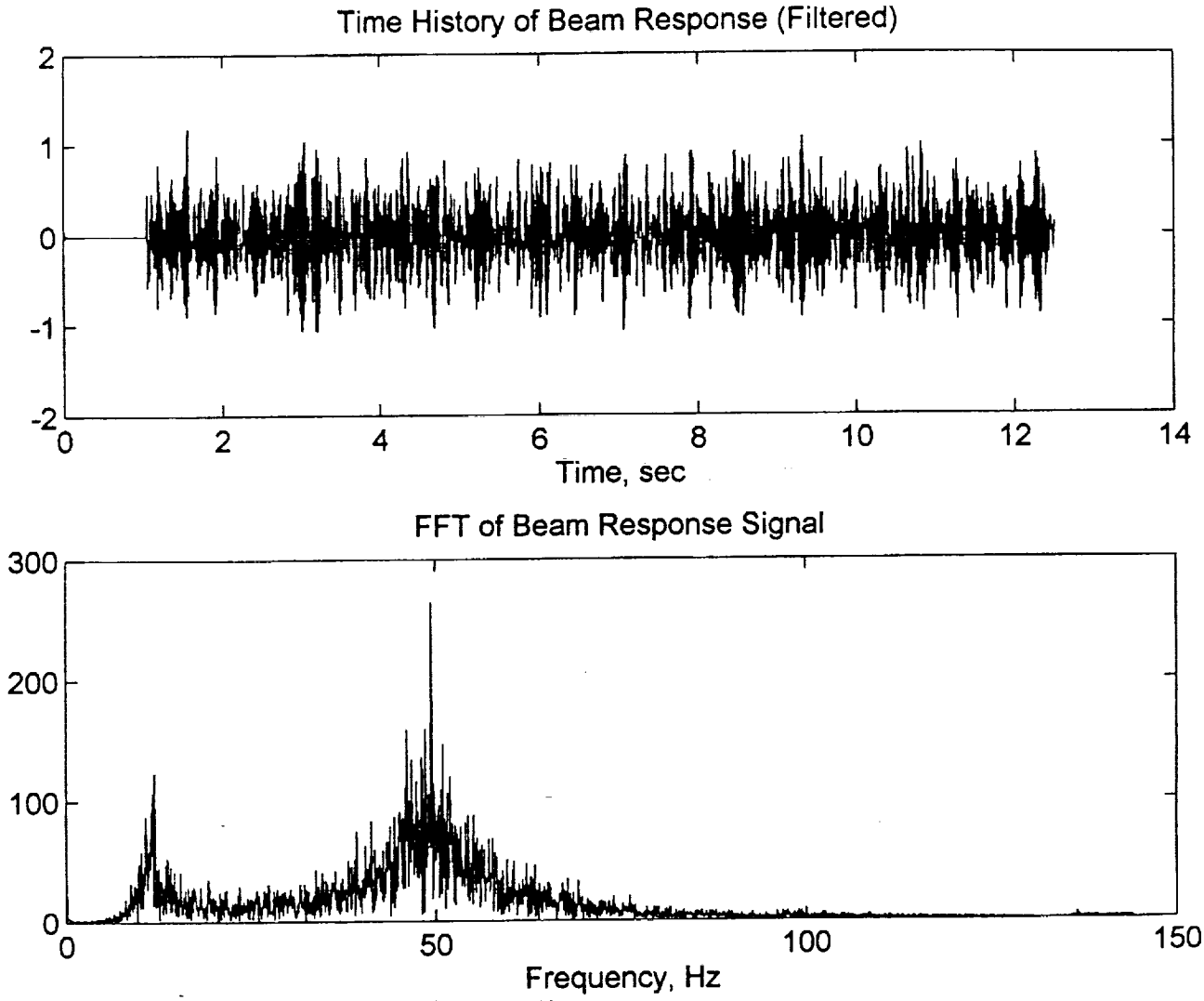

Figure 5.4: Aluminum Beam Filtered Acceleration Response and its FFT (f_s = 500 Hz)

5.4.1 Single DOF Case (Single Mode)

For the single mode ID, the beam was clamped to the steel platform at a 40 inch length. The excitation point was chosen to be near the center of the beam and the response was taken at the tip as shown in Figure 5.1. The excitation point was chosen so that at least the first two modes will be well excited.

The first and second theoretical natural frequencies of the beam in this configuration (clamped-free @ L = 40 in) are 10.2 and 64 Hz, respectively [8]. Prior to the actual testing, a random unfiltered excitation was applied to the beam and the response was sampled, collected and transferred to MATLAB by the C40 system. The fast fourier transform (FFT) of the response (Figure 5.3) is calculated in MATLAB to determine empirically the first and second natural frequencies of the beam in this setup. It was found that the first frequency appears at approximately 11 Hz and the second frequency appears in the vicinity of 50 Hz. The FFT plot shows that the first mode appears as a fairly sharp component whereas the second mode is not as sharp and is therefore hard to pinpoint exactly. However, 50 Hz is a good estimate. The first mode frequency is fairly close to the theoretical value of 10.2 Hz, but the second mode frequency appears at a significantly lower value than that of the theoretical 64 Hz. The difference can be attributed to the fact that the beam is fairly stiff and the mounting steel structure is not absolutely rigid. Depending on the magnitude of the excitation force, the entire mounting structure can shake noticeably. In addition, the wooden workbench on which the shaker is placed vibrates slightly and introduces more error into the signals.

In this section, only the first mode is considered. Therefore, during the actual system identification experiment, the random shaker signal is constructed such that only the first mode is mainly excited which is done by applying a low-pass digital filter (ω_c = 15 Hz) on the signal coming out of computer #2. However, this process does not totally prevent the next two or three modes from being partially excited. This is where the low-pass digital Butterworth filter comes into play. The signals are sampled at 4 kHz which ensures that all the modes that could be excited are picked up at the correct frequencies and do not alias down to the lower band, these are then filtered out using the digital low-pass filter before the training process starts. For this test, the Butterworth filter was designed to have a 20 Hz cutoff frequency which effectively isolates the first mode.

A large number of tests were conducted in this arrangement to examine the influence of the sampling frequency, the learning rate and the low-pass filter on the system identification process. Among those, ten case studies are selected and presented herein. Sample code is shown in the file **C40SDOF.C** in Appendix B. The test cases are summarized in Table 5.1.

Table 5.1: Summary of the Single DOF Real-Time Test Cases

Case No.	Sampling Frequency, (Hz)	Learning Rate	Anti-Aliasing Digital Filter?
1	100	0.01	Yes
2	100	0.01	No
3	100	0.03	Yes
4	100	0.03	No
5	100	0.1	Yes
6	100	0.2	Yes
7	200	0.01	Yes
8	200	0.1	Yes
9	200	0.2	Yes
10	400	0.017	Yes

Case 1 (Figure 5.5): effective sampling rate: $f_s = 100$ Hz
learning rate: $\alpha = 0.01$

Results show that the first mode of vibration of the beam is identified as expected. The natural frequency converges to the expected value of 11 Hz and the damping ratio converges to an average value of 0.011. The modal constant should have converged to a value near 1, but the plot shows that it is slowly moving towards that value. This is due to the low learning rate used. Figure 5.5 shows the linear combiner weights. The second and third weights seem to converge rapidly, but the first weight does not. This is expected because the first weight is the one that influences the mode shape. The error plot shows that it has been reduced significantly but did not go to zero. Actually, unlike simulation, none of the errors in the real-time testing ever converge to zero even if the system identification process is completed. This is mainly due to anomalies such as high frequency noise and non-linearities in the input signals. Such problems are always present in a real-life testing environment. If the modal parameters and weight plots are examined closely, one can see that they are not entirely smooth, which means that the algorithm is actually trying to deal with the anomalies of the input signals. This as it turns out is one of the most powerful features of adaptive filters, their ability to learn the complex behavior of signals.

It must be noted that there exists a lag of almost one second between the moment the C40 executes the code and starts the training process and the actual arrival of the first samples of data. This delay is always present in the system and according to the third party vendor, it is a built-in problem that comes with the A/D converters. Most of the plots show such an effect, which appears mostly as a flat region at the beginning of the time history. When the sampling rate is high, the problem becomes more apparent.

Case 2 (Figure 5.6): effective sampling rate: $f_s = 100$ Hz
learning rate: $\alpha = 0.01$
WITHOUT ANTI-ALIASING FILTER

This case is the same as the one above, except that the low-pass Butterworth filter that deals with the aliasing problem is not applied to the input signals. Consequently, the results did not come out as good as the filtered case above. The error time history shows how the algorithm becomes far more susceptible to the aliased higher modes and high frequency noise in the signal, thus producing a large error.

Case 3 (Figure 5.7): effective sampling rate: $f_s = 100$ Hz
learning rate: $\alpha = 0.03$

The anti-aliasing filter is applied once again, but the learning rate is increased by a factor of three. The results are better than those in case 1. Total system identification time is cut by almost a half and the modal constant does converge, but still sluggish. As expected, the first weight which influences the mode shape converges faster in comparison to case 1. The error is minimized effectively.

Case 4 (Figure 5.8): effective sampling rate: $f_s = 100$ Hz
learning rate: $\alpha = 0.03$
WITHOUT ANTI-ALIASING FILTER

Similar to case 3, but with the aliasing problem. Results are definitely worse. Cases 2 and 4 show clearly the significant effect of aliasing and prove beyond any doubt that some form of anti-aliasing should always be used in order to get the best possible results. Since this point has been clearly illustrated, the low-pass Butterworth filter will be used to combat the aliasing problem from this point on.

Case 5 (Figure 5.9): effective sampling rate: $f_s = 100$ Hz
learning rate: $\alpha = 0.1$

The learning rate is increased by an order of magnitude compared to case 1. Results show a much faster convergence. However, the plots indicate increased susceptibility of the algorithm to high frequency noise in the signal due to a higher learning rate in comparison to case 1 and case 3. Error is minimized almost immediately and maintains a very low value throughout the remainder of the process.

Case 6 (Figure 5.10): effective sampling rate: $f_s = 100$ Hz
learning rate: $\alpha = 0.2$

Learning rate is increased once again and the results show even more improvement in terms of convergence, almost instantaneous, especially in the mode shape constant which converges in case 5 within 10 seconds, but converges here in less than one second. However, the results indicate that the algorithm becomes more influenced by the noise in the signal due to the increased learning rate. Cases 5 and 6 prove that the algorithm has a wide range of learning rates that will produce stable results.

Case 7 (Figure 5.11): effective sampling rate: $f_s = 200$ Hz
learning rate: $\alpha = 0.01$

Learning rate is set to the original value of 0.01 and the sampling rate is doubled. In comparison to case 1 which has same learning rate, it is apparent that the modal constant rate of convergence became worse. Frequency and damping convergence are about the same. The error reduction is also slow, but does move towards a smaller value.

Case 8 (Figure 5.12): effective sampling rate: $f_s = 200$ Hz
learning rate: $\alpha = 0.1$

Same as case 7 but with a higher learning rate. Comparison with case 3 (same learning rate but twice the sampling rate) shows very similar behavior which would indicate that increasing the sampling rate did not have much influence on the learning process. Note that the time scale is different between the two cases.

Case 9 (Figure 5.13): effective sampling rate: $f_s = 200$ Hz
learning rate: $\alpha = 0.2$

This case is included to illustrate a couple of points. The natural frequency convergence is instantaneous similar to case 6, but the plots show a very unusual behavior. As shown in Figure 5.13, the damping ratio plot has many pulses with an order of 10^{38} magnitude. These are the results of dividing by zero which causes an overflow error in the C40 system. According to Reference [17], the most positive extended precision floating point number that the C40 DSP processor has is 3.402×10^{38}. Any higher number causes an overflow and the C40 always returns the largest FPN. It can also be noted that these pulses correspond to values in the natural frequency plot of zero. Since the natural frequency appears in the denominator of the damping ratio term (equation 2.9), then when the natural frequency is zero, the damping ratio term overflows. The natural frequency usually goes to zero when the term under the square root in equation 2.8 is negative. The C40 system sets the whole square root to zero if the term under the

square root is negative. If this becomes a problem in future applications, then it can be easily solved by a simple if statement.

Case 10 (Figure 5.14): effective sampling rate: $f_s = 400$ Hz
learning rate: $\alpha = 0.017$

In order to verify the explanation given at the end of Chapter 3 regarding the effect of sampling rate on the adaptation algorithm performance, high sampling rates were used to study their effect on the single DOF case. Figure 5.14 shows that even at a sampling rate 40 times the natural frequency of the single DOF case, the algorithm does manage to correctly identify the mode. However, choosing an optimum learning rate that is high enough for a reasonable time to convergence, but does not cause an overflow problem as seen in case 9 above, becomes rather difficult. The results in this case were obtained after 14 trials. During these trials, the natural frequency does converge to the expected value but it is usually jagged and goes to zeros many times causing persistent overflow problems. So, even though the actual identification process is not affected significantly by the increase in the sampling rate, finding an appropriate learning rate becomes more difficult.

Remarks

It has been shown that the SDOF system identification process can be influenced by the learning rate and the sampling rate as well as aliasing. Signals that were not processed by an anti-aliasing filter were difficult to identify (cases 2 and 4). In addition, sampling very fast or using a high learning rate can give the algorithm stability problems as seen in cases 6, 9 and 10. In general, the author has found that there usually exists a combination of learning rate and sampling rate that produces the best results for the case at hand. Such a combination is usually determined by trial and error. Please note that the term 'best' is loosely used here, because there is usually a trade-off between stability and speed. Cases 3 and 6 illustrate this point very well. Case 3 is slower to converge and case 6 is more susceptible to noise, thus becomes less stable. The balance between speed and stability will usually depend on the application at hand.

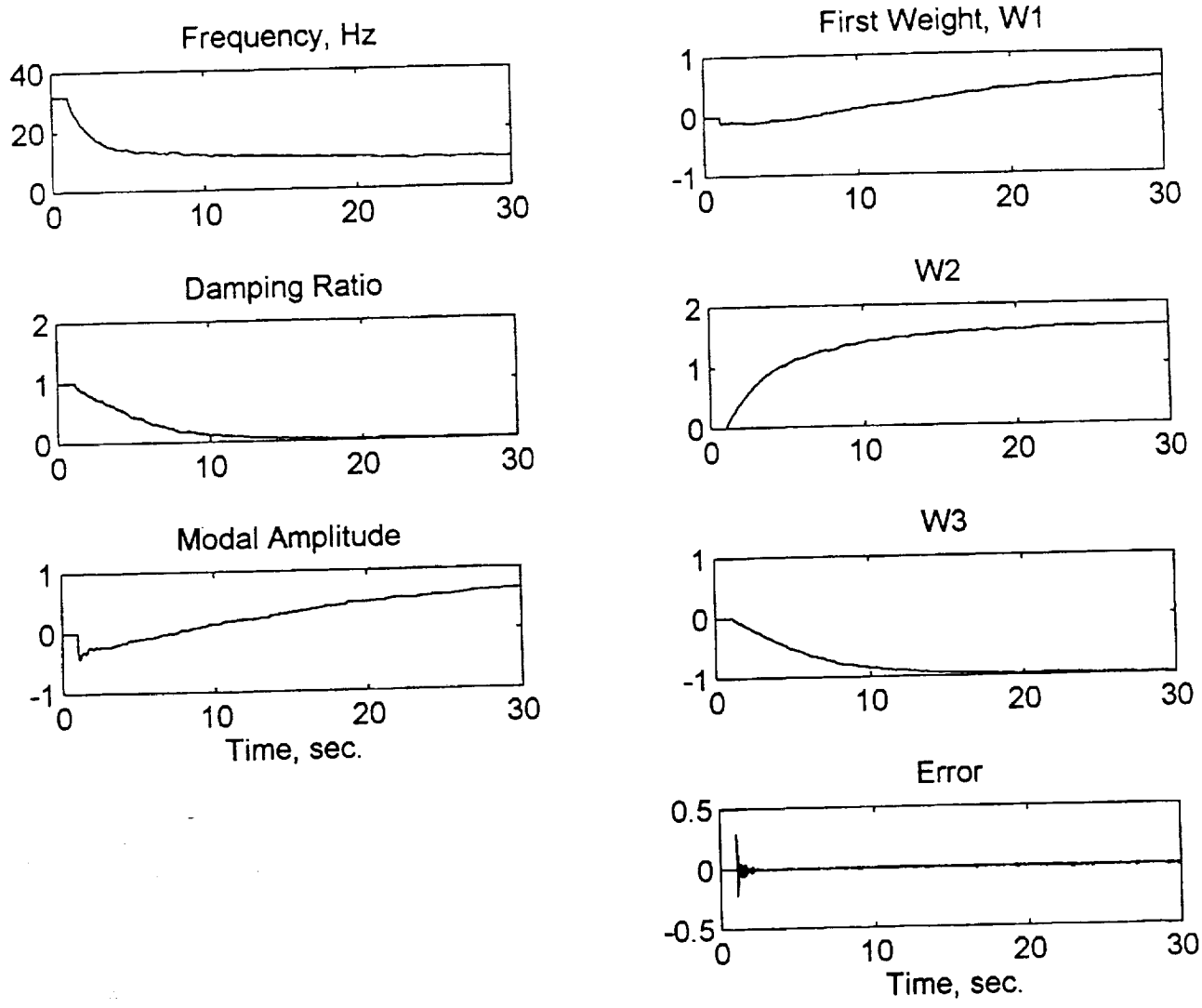

Figure 5.5: Single Mode Real-Time System Identification Results
(Case 1: f_s = 100 Hz, α = 0.01)

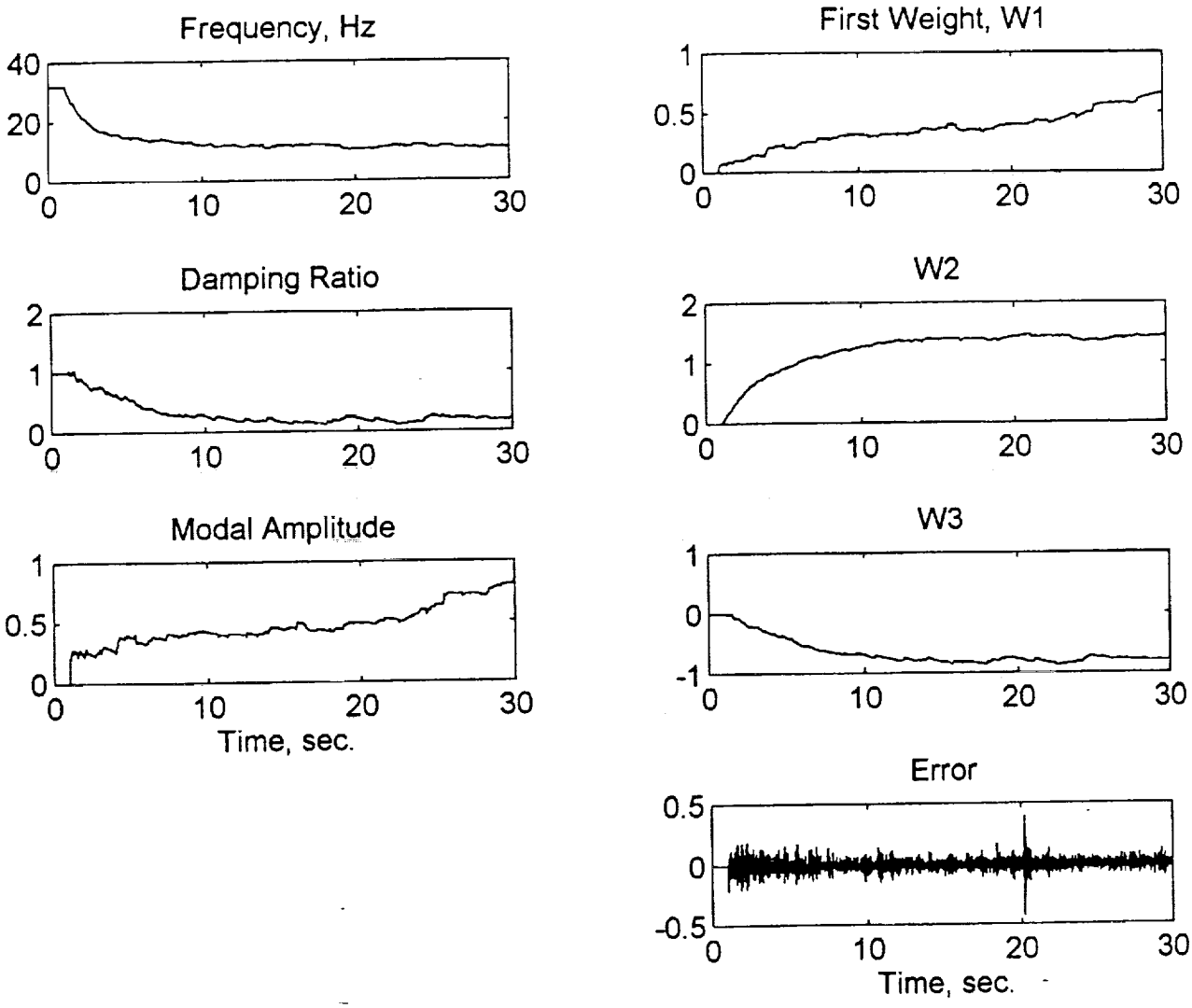

Figure 5.6: Single Mode Real-Time System Identification Results
(Case 2: f_s = 100 Hz, α = 0.01, no filtering)

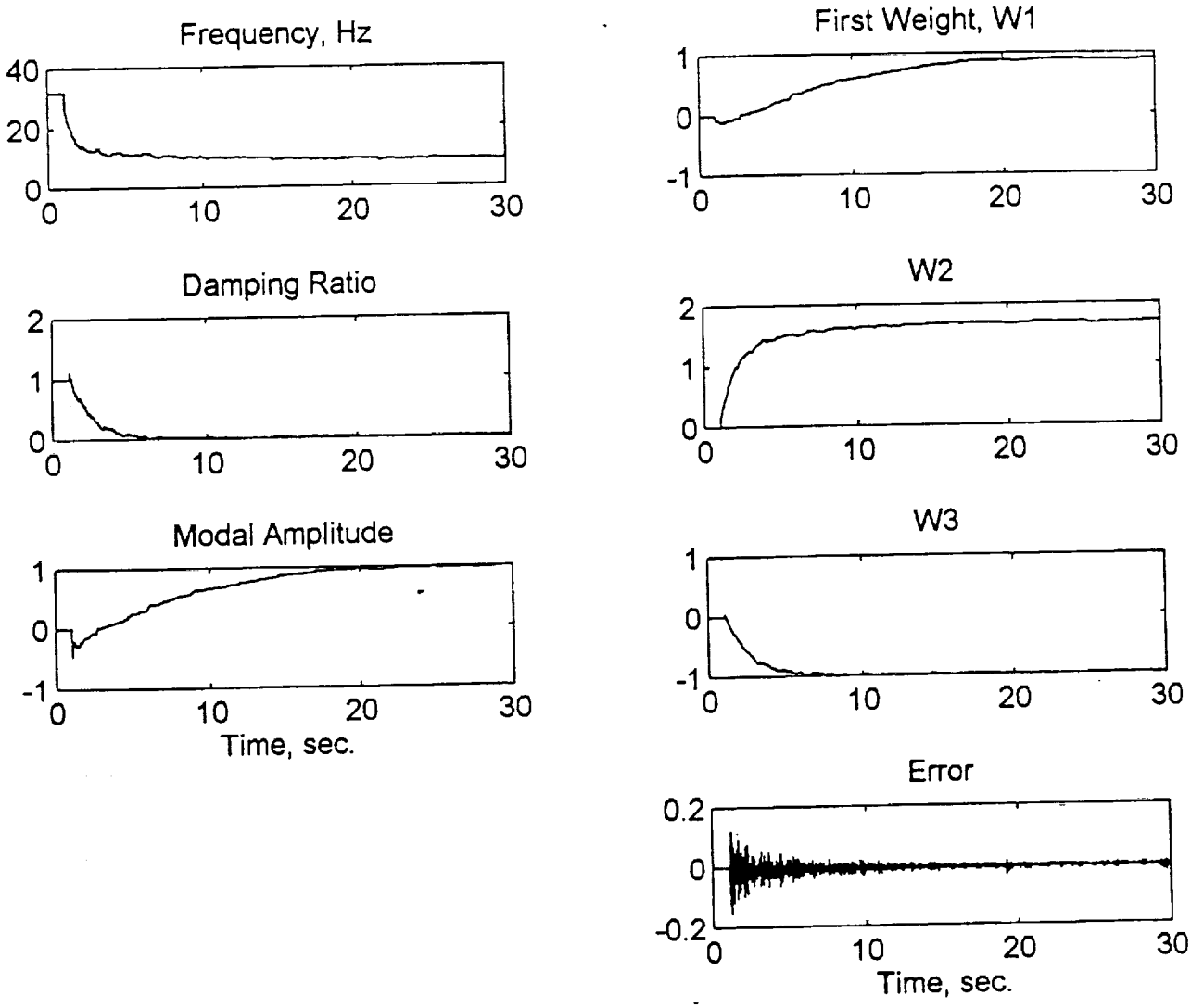

Figure 5.7: Single Mode Real-Time System Identification Results
(Case 3: f_s = 100 Hz, α = 0.03)

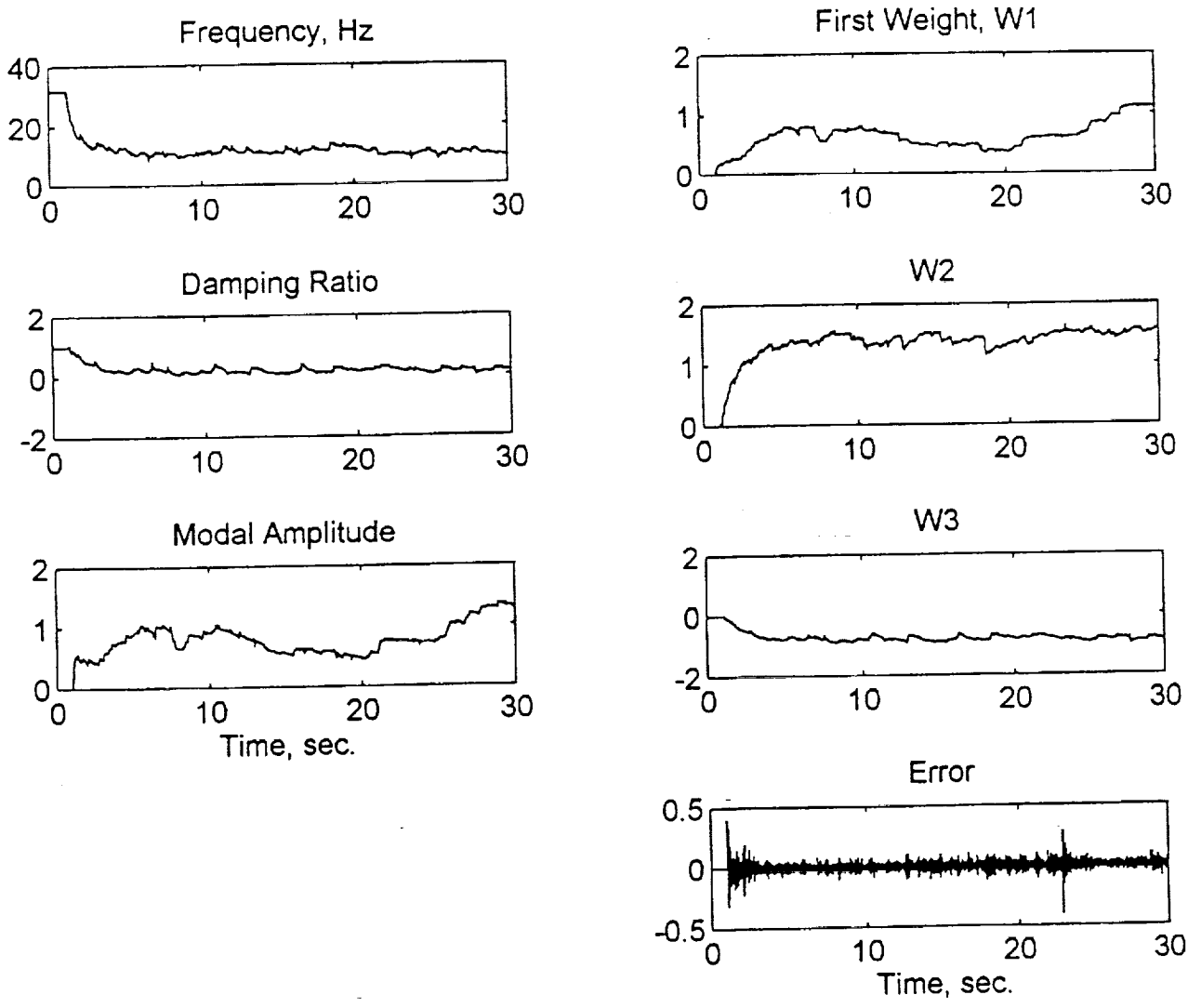

Figure 5.8: Single Mode Real-Time System Identification Results
(Case 4: f_s = 100 Hz, α = 0.03, no filtering)

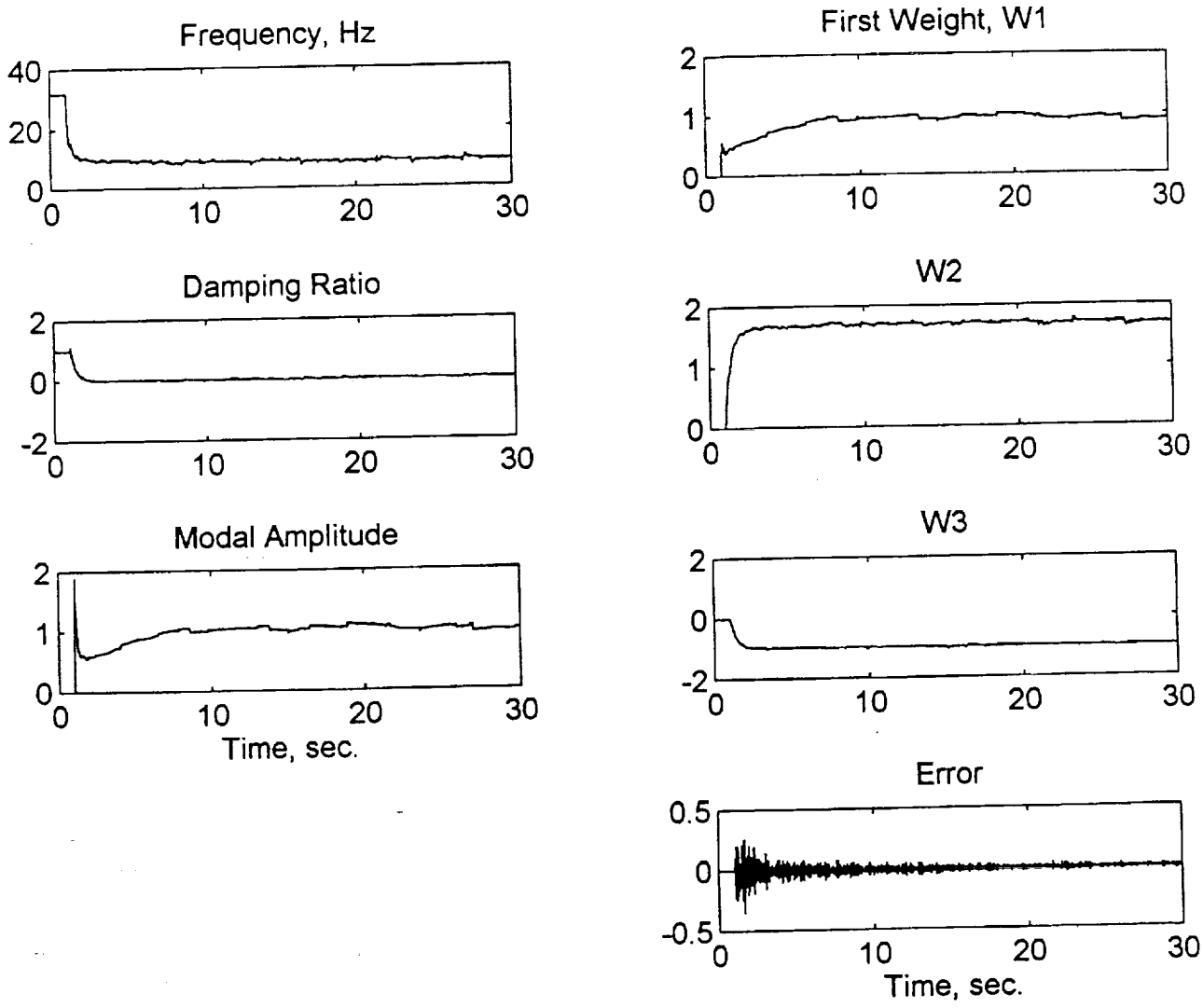

Figure 5.9: Single Mode Real-Time System Identification Results
(Case 5: f_s = 100 Hz, α = 0.1)

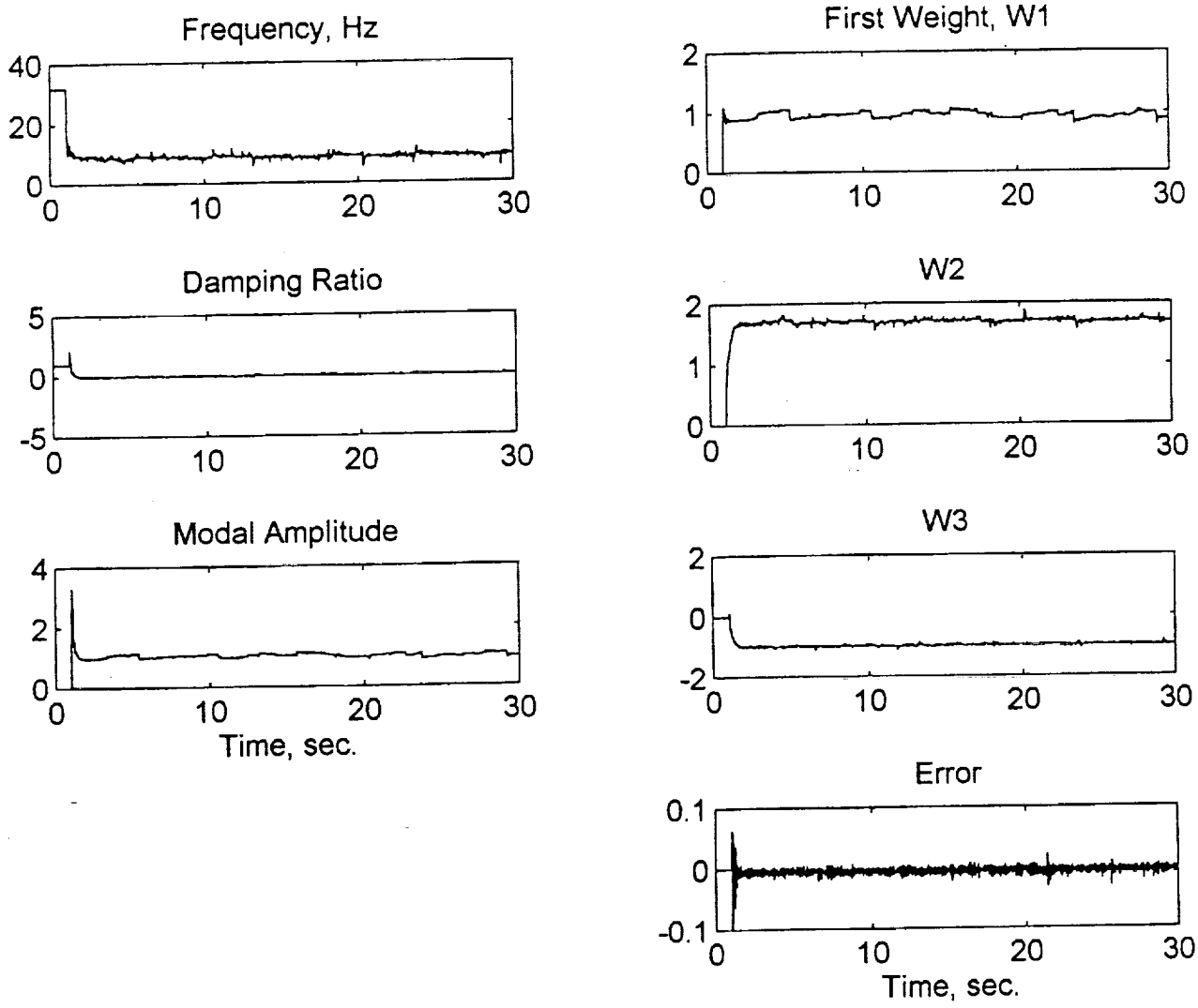

Figure 5.10: Single Mode Real-Time System Identification Results
(Case 6: f_s = 100 Hz, α = 0.2)

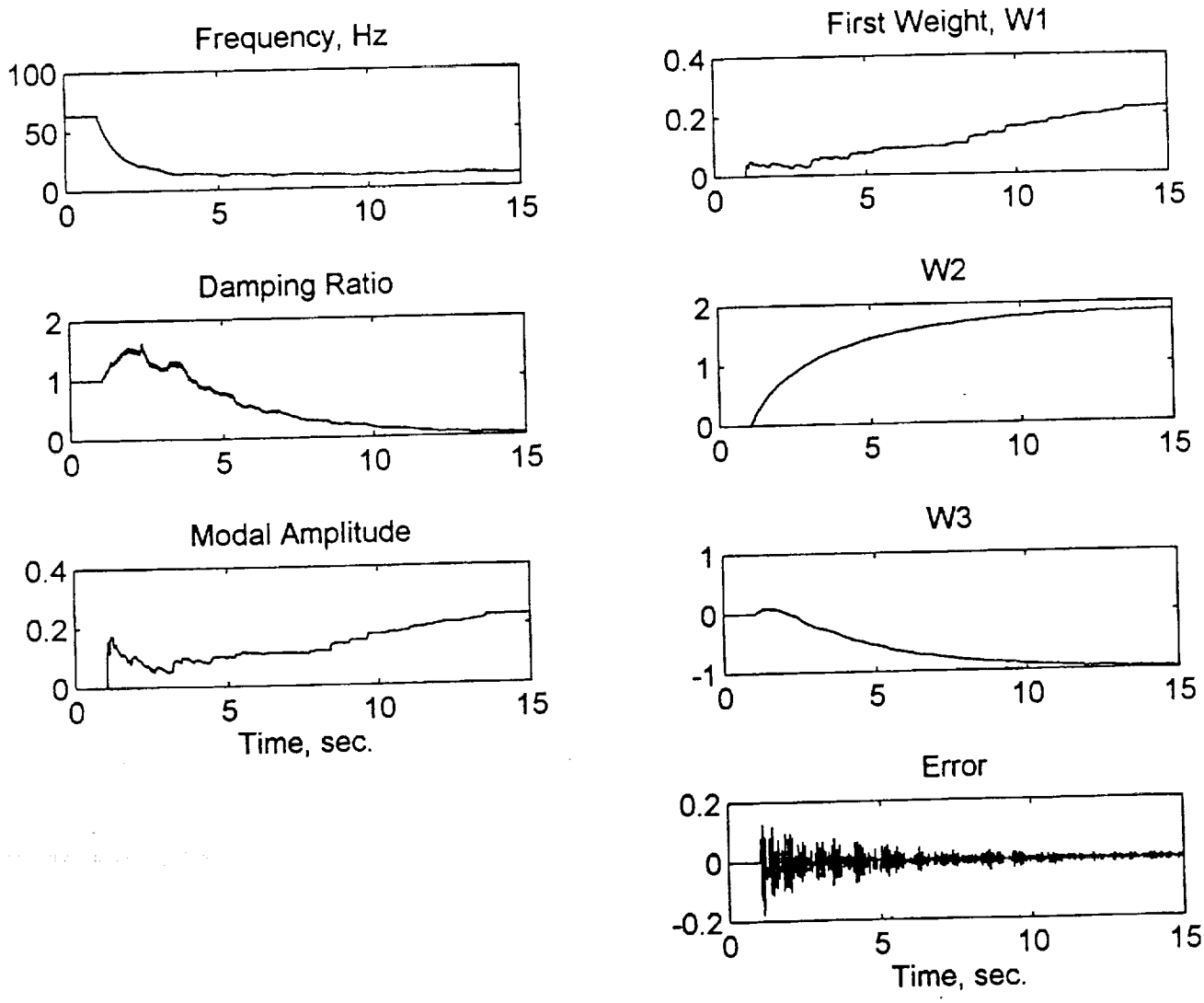

Figure 5.11: Single Mode Real-Time System Identification Results
(Case 7: f_s = 200 Hz, α = 0.01)

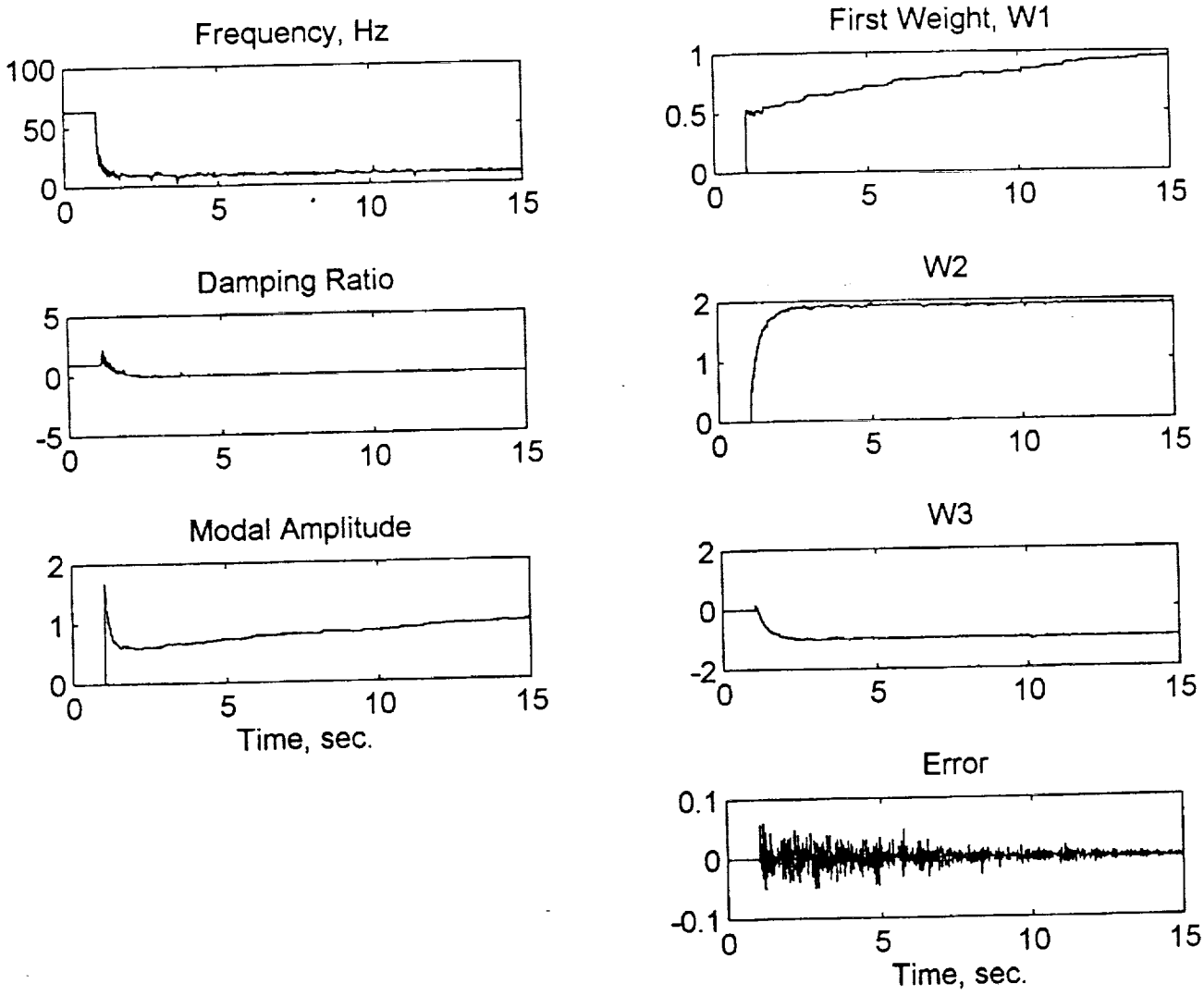

Figure 5.12: Single Mode Real-Time System Identification Results
(Case 8: f_s = 200 Hz, α = 0.1)

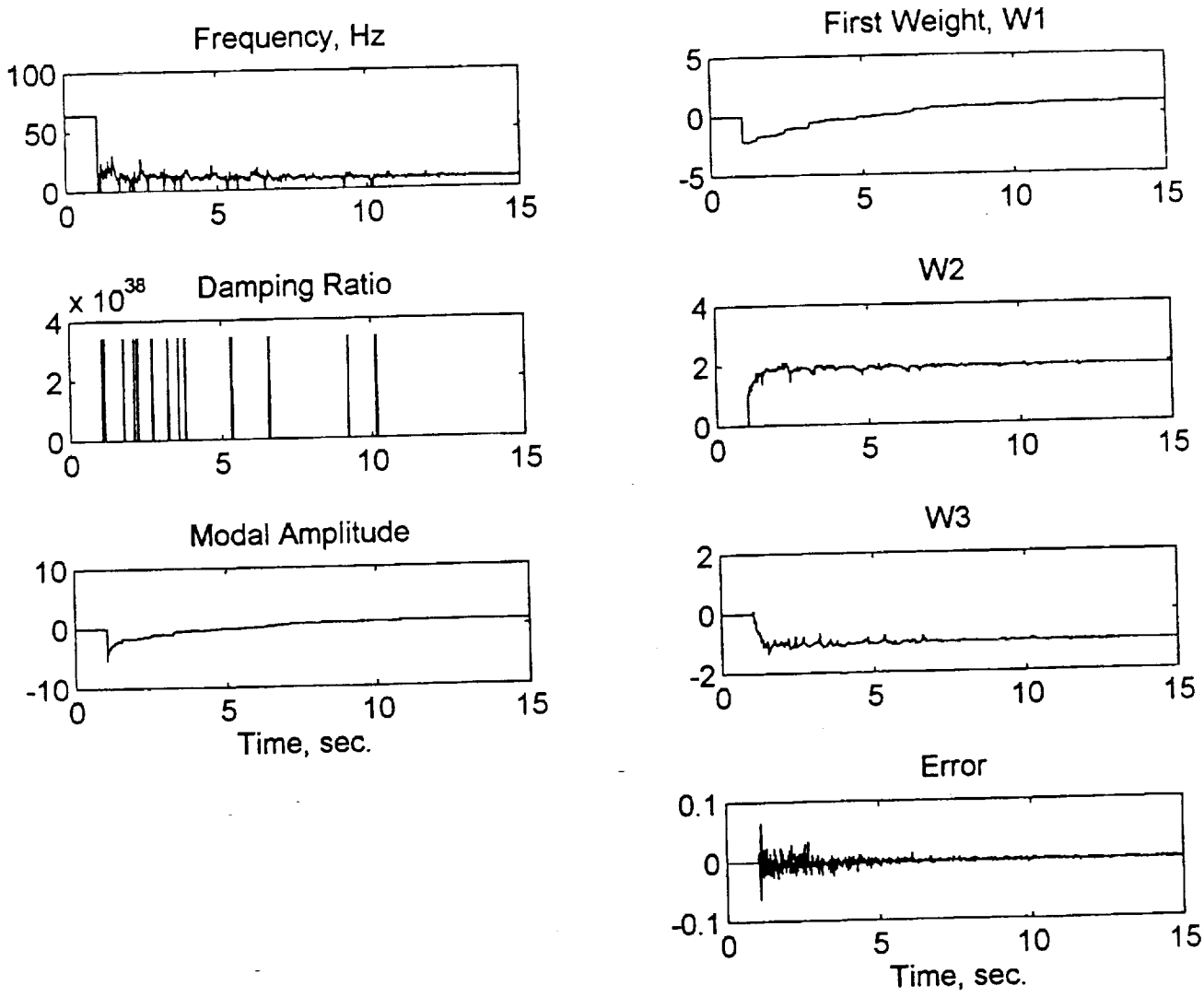

Figure 5.13: Single Mode Real-Time System Identification Results
(Case 9: f_s = 200 Hz, α = 0.2)

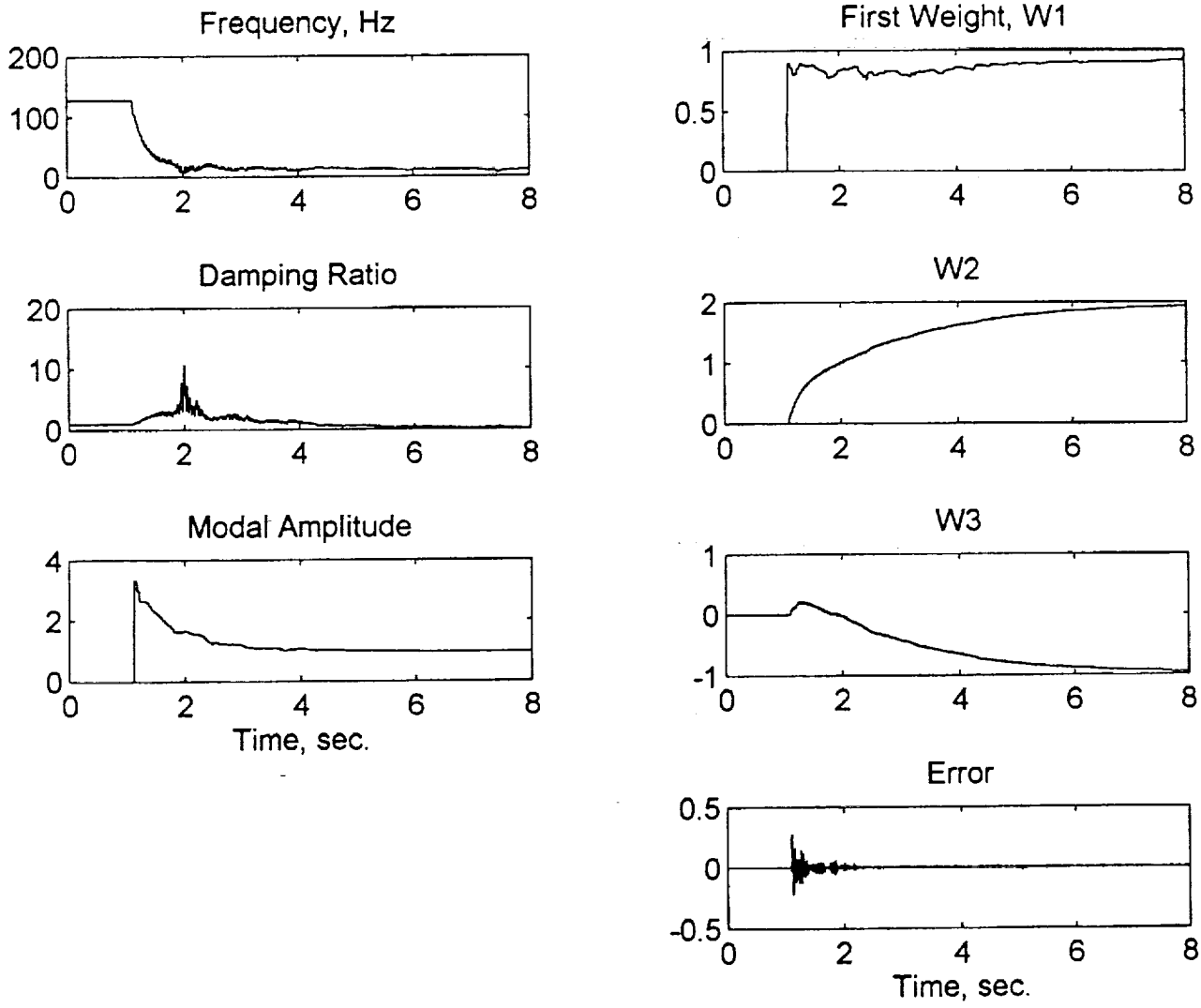

Figure 5.14: Single Mode Real-Time System Identification Results
(Case 10: f_s = 400 Hz, α = 0.017)

5.4.2 Two DOF Case (Two Modes)

The same structure and set up as in the SDOF case, is used here. The excitation was applied at the center of the beam (point 1) and the response was taken either at the tip (point 2) or at the center of the beam to provide the same point results needed to evaluate the mode shape coefficients as discussed in Chapter 2. A fourth order low-pass digital Butterworth filter is used to tackle the aliasing problem similar to the one used in the SDOF case but with a different cutoff frequency in order to isolate the first two modes only (ω_c = 70 Hz for the beam in this setting). Figure 5.15 shows the frequency response magnitude and phase of this filter. Note that the phase for this filter does change significantly in the region below the cutoff frequency. This has not affected the identification process, but can be avoided by using a higher order filter.

The random excitation signal going into the shaker was filtered using a low-pass digital filter built in the **UWNoise.VI** random signal generator at a cutoff frequency of 60 Hz. This causes the shaker to mainly excite the first two modes of the beam. Sample code is shown in the file **C40DDOF.C** in Appendix C, and Table 5.2 contains a summary of test cases in this section. Please note that the cases listed below are the result of many test runs that were done to try and establish some kind of a trend in each case for meaningful comparisons between cases. Each test run, even under the exact same conditions, is unique and can produce different results. Representative results are listed herein.

Table 5.2: Summary of the Two DOF's Real-Time Test Cases

Case No.	Sampling Frequency, Hz	Learning Rate	Input/Output Measurements
1	200	0.01	non-collocated
2	160	0.008	non-collocated
3	200	0.(	collocated
4	200	0.03	non-collocated
5	250	0.03	non-collocated
6	400	0.001	non-collocated
7	500	0.01	non-collocated
8	1000	0.01	non-collocated

Case 1 (Figure 5.16): effective sampling rate: f_s = 200 Hz
learning rate: α = 0.01
non-collocated response measurement (at tip)

First mode frequency is seen slowly converging to the expected value, whereas the damping is slow to do so even after almost 80 seconds. The modal constant seems to be leveling off around 2. The second frequency identified by the adaptive filter is around 74 Hz, which is much higher than the expected second mode frequency (around 50Hz). However, the beam does not have a mode of vibration in the vicinity of 74 Hz.

This was a point of confusion for sometime until a characteristic of the bilinear transformation known as *frequency warping* came to the attention of the author which provided a very logical explanation to this seemingly strange behavior. As it turns out, the frequency identified at a higher value is actually the second mode natural frequency which is around 50Hz, but due to frequency warping it shows up as a higher value. This phenomenon was mentioned briefly in Chapter 3 and it seems appropriate at this point to discuss it and its effect on the results of this project in greater details. The following section provides such a detailed discussion.

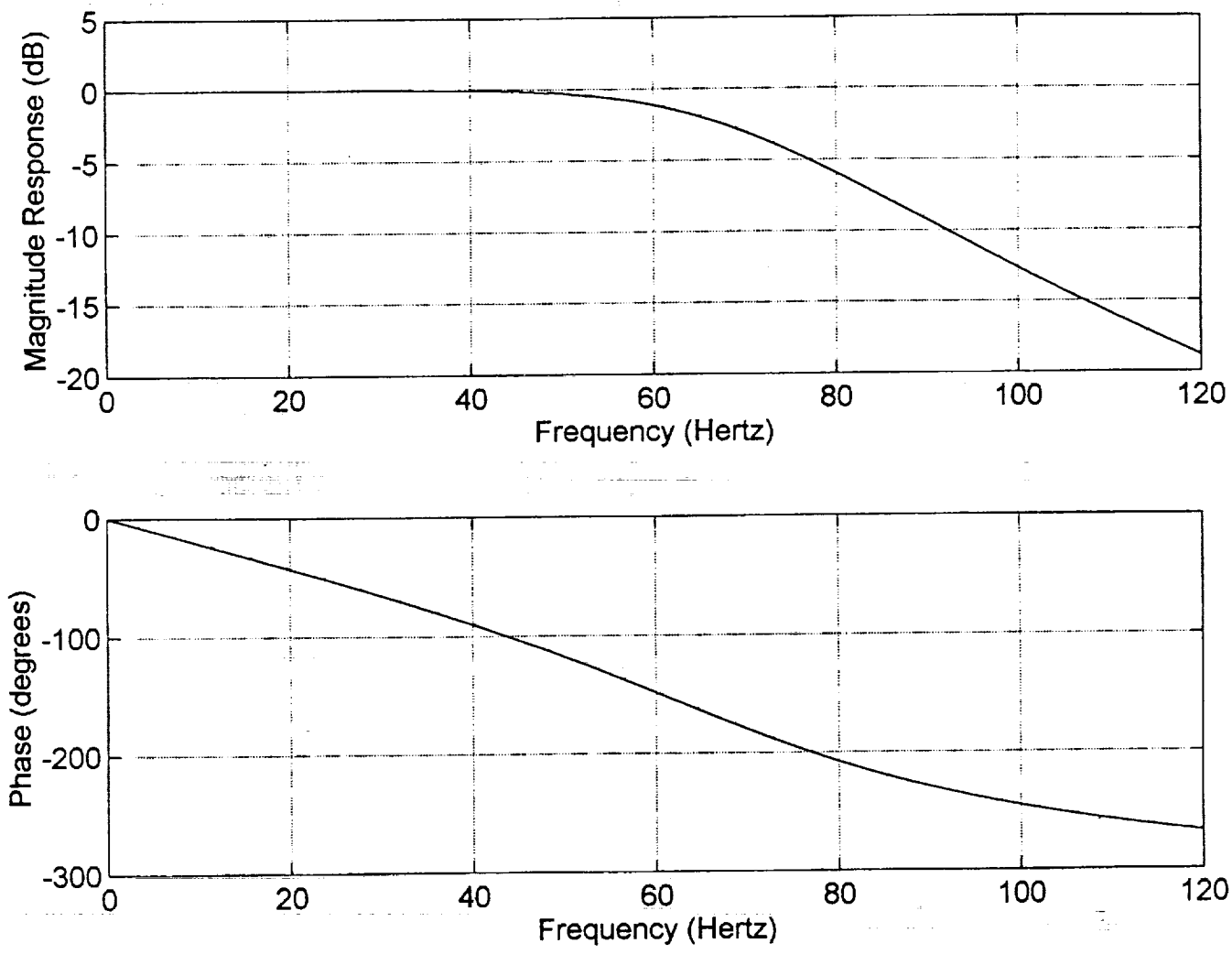

Figure 5.15: Frequency Response of a Fourth-Order, Low-Pass, Digital Butterworth Filter (f_s = 4000 Hz, ω_c = 70 Hz)

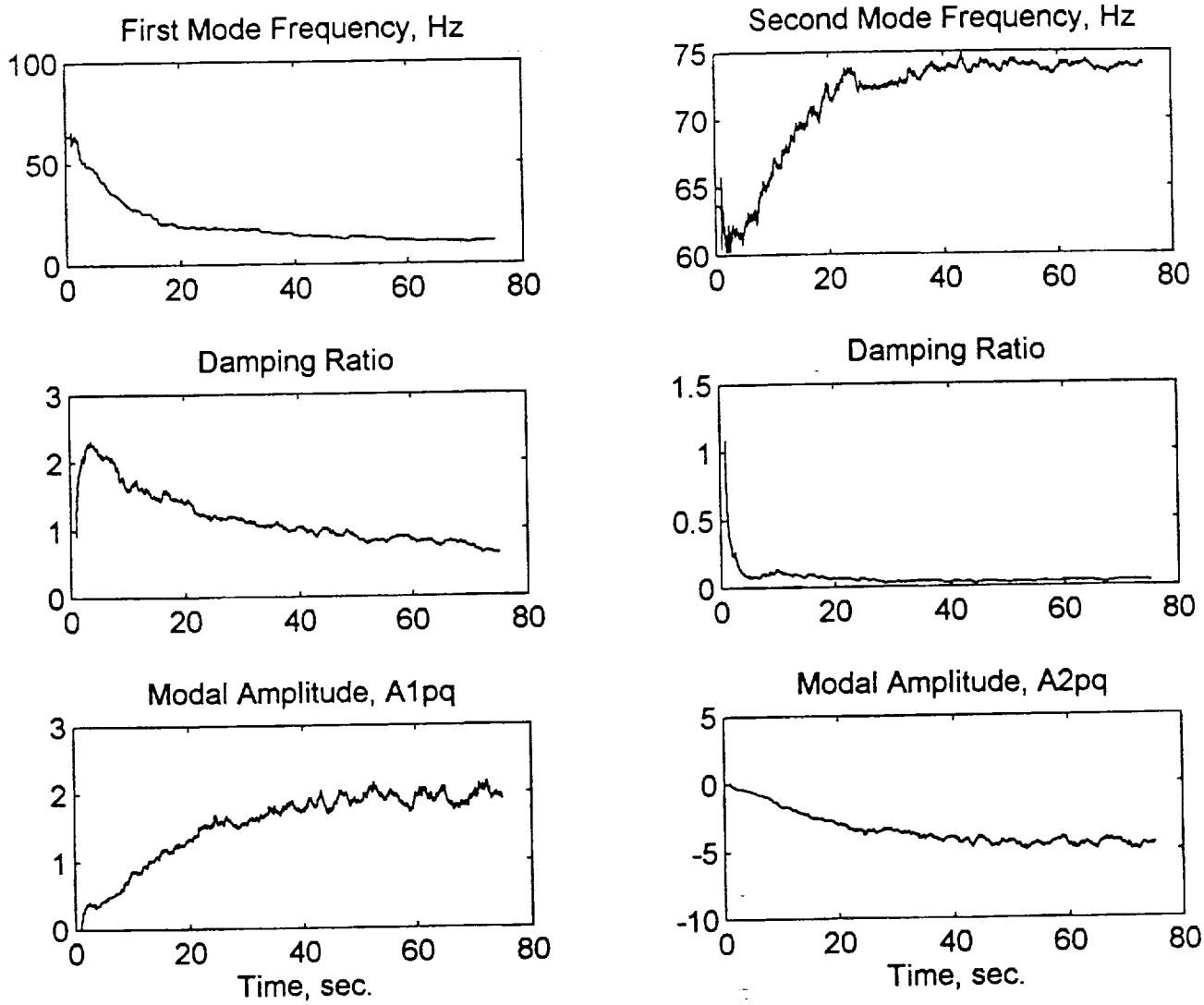

Figure 5.16: Two Mode Real-Time System Identification Results
(Case 1: f_s = 200 Hz, α = 0.01)

Bilinear z-Transformation and Frequency Warping

A popular method of designing discrete time IIR filters is to utilize the already existing highly advanced art of continuous-time IIR filter design and transforming this design into the discrete domain to obtain a discrete IIR filter that meets prescribed specifications [7]. Two of the most widely used methods to achieve this are the impulse invariance method and the bilinear transformation (BT, or Tustin) method. The basic idea of the impulse invariance method is to select an impulse response of the discrete filter to match the impulse response of the continuous filter. The details and problems associated with such a method are beyond the scope of this thesis, but it must be noted that one of the major problems associated with impulse invariance is the aliasing problem. References [7] and [30] have extensive discussions into this topic. The second method that is of interest to us is the bilinear transformation. This technique avoids the problem of aliasing inherent in the impulse invariance method by mapping the entire imaginary ($j\omega$) axis of the s-plane into the unit circle in the z-plane. This non-linear compression of an entire axis that extends from $-\infty$ to $+\infty$ into a unit circle is called *frequency warping*. Figure 5.17 (solid line) shows this mapping using a sampling frequency of 100 Hz. The following relationship gives the discrete frequency as a function of the continuous frequency using the BT:

$$\Omega = 2\arctan\left(\frac{\omega T_s}{2}\right) \tag{5.1}$$

where Ω is the discrete frequency, ω is the analog frequency and T_s is the sampling period ($1/f_s$). Detailed derivation of this formula can be found in Reference [7] which also recommends that the BT be used only if the warping of the frequency axis can be tolerated or compensated for, using the following equation

$$\omega = 2f_s \tan\left(\frac{\Omega}{2}\right) \tag{5.2}$$

Solve for f_s

$$f_s = \frac{\omega}{2\tan\left(\frac{\Omega}{2}\right)} \tag{5.3}$$

which means that in the initial development of the discrete system (Chapter 2), instead of using the actual BT as shown in equation (2.3)

$$s = 2f_s \frac{z-1}{z+1}$$

the following pre-warped formula would have to be used [30]

$$s = \frac{\omega}{\tan\left(\frac{\Omega}{2}\right)} \frac{z-1}{z+1} \tag{5.4}$$

This equation requires previous knowledge of the actual system frequency and the exact discrete frequency that maps it. However, this equation is not used in this project because the warping effect was not discovered and understood early enough to justify a complete redevelopment of the mathematical model of the system and the adaptive filter used.

The BT is used mainly because of its simplicity. It is nothing more than an algebraic transformation between the *s* and *z* variables and can be easily implemented by simple substitution as shown in Chapter 2. According to Reference [7], if the continuous-time system (filter) is band-limited, then the "discrete-time and continuos-time frequency responses are related by a linear scaling of the frequency axes" (p. 408):

$$\Omega = \omega T_s \qquad (5.5)$$

Unfortunately, this relationship requires exactly band-limited continuous systems, otherwise aliasing becomes a problem, which is almost impossible in real-life. However, this linear scaling will be used here to represent the ideal theoretical relationship that relates both frequencies together as shown by the dotted line in Figure 5.17 (where f_s = 100Hz). The figure shows that there exists a region (near the origin) in which the theoretical relationship and the BT relationship between the frequencies is identical. This region can be extended using high sampling rates as shown in Figure 5.18 (f_s = 1000Hz). In other words, by sampling at a high rate, the linear region can be expanded and the effect of frequency warping will be reduced.

Once the concept of frequency warping is understood, the results of the second mode identification can be explained. Substituting the adaptive filter coefficients (b_1, b_2, a_{11}, a_{12} . . . . etc.) into equations 2.8, 2.8 and 2.10 to evaluate the modal parameters of the specific mode is actually performing the inverse of the bilinear transformation to get from the *z*-domain back to the *s*-domain. So, if the mode identified by the adaptive filter has a discrete frequency at a certain value, then by applying the inverse BT to that frequency in the non-linear region causes such a frequency to show up at a higher value in the continuous domain. Figure 5.19 shows the continuous-discrete frequency relationship at a sampling rate of 200Hz (case 1). If a vertical line representing the actual natural frequency of the second mode is drawn at or around the 50Hz point on the horizontal axis (continuous) until it intersects the theoretical plot (dotted), and a horizontal line that goes through this point is drawn to intersect with both the vertical axis and the BT plot (solid), then that horizontal line represents the actual discrete frequency corresponding to the second mode. However, the point of intersection of this discrete frequency and the BT plot corresponds to a continuous frequency that is higher than the actual value. Therefore, the mode appears at a higher frequency. It must be noted that the experimental values do not correspond exactly to this argument, but taking measurement errors into consideration, they are reasonably close. The first mode is usually not affected because it falls in the linear region of the BT plot, and hence the mapping is exact.

Now if the frequency warping explanation holds, then reducing the sampling rate further should magnify the problem. This is exactly what happens in case 2 below as shown in Figures 5.20. The sampling frequency was reduced to 160 Hz, and as expected, the adaptive filter identified the second mode as one with an even higher frequency than that at the 200Hz case. On the other hand, increasing the sampling rate should reduce the effect of frequency warping. This is also true as shown in the cases to follow.

Furthermore, one might even expect that if this argument was to hold true, then similar behavior should have be seen in simulation. But the simulation results of Chapter 3 did not have this problem, the second mode frequencies always appeared at the correct values even at sampling rates as small as twice the second mode natural frequency. This apparent contradiction can be easily explained by noting that in the MATLAB simulation of Chapter 3 the continuous-time system was converted to a discrete-time system using the BT option. When the adaptive filter coefficients were used to evaluate the modal parameters using the inverse BT, the warping problem automatically corrected itself, and consequently there was no hint that there ever was a problem in the first place. Actually, this is not entirely true, because the FFT plots of the simulated system

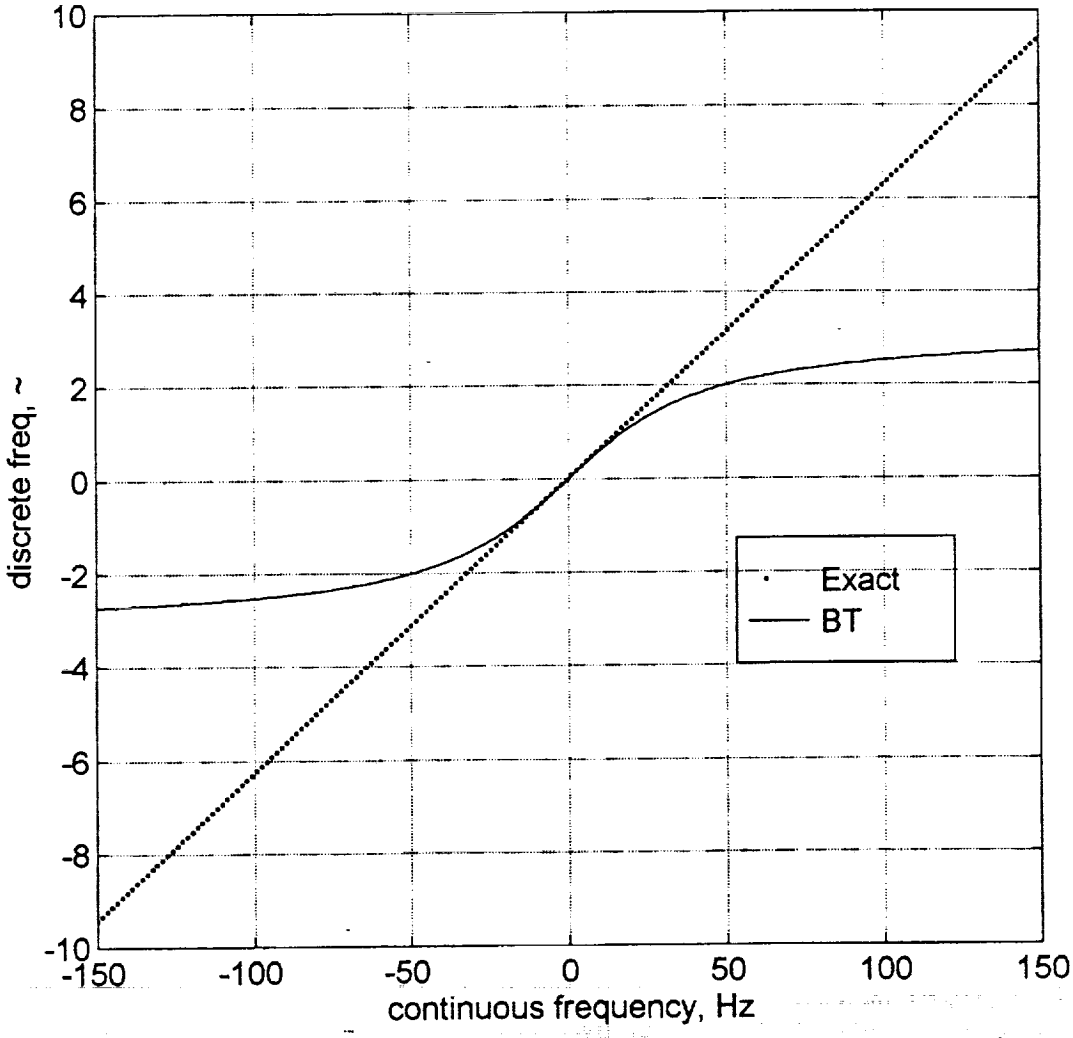

Figure 5.17: Frequency Warping Associated with the Bilinear Transformation (f_s = 100 Hz)

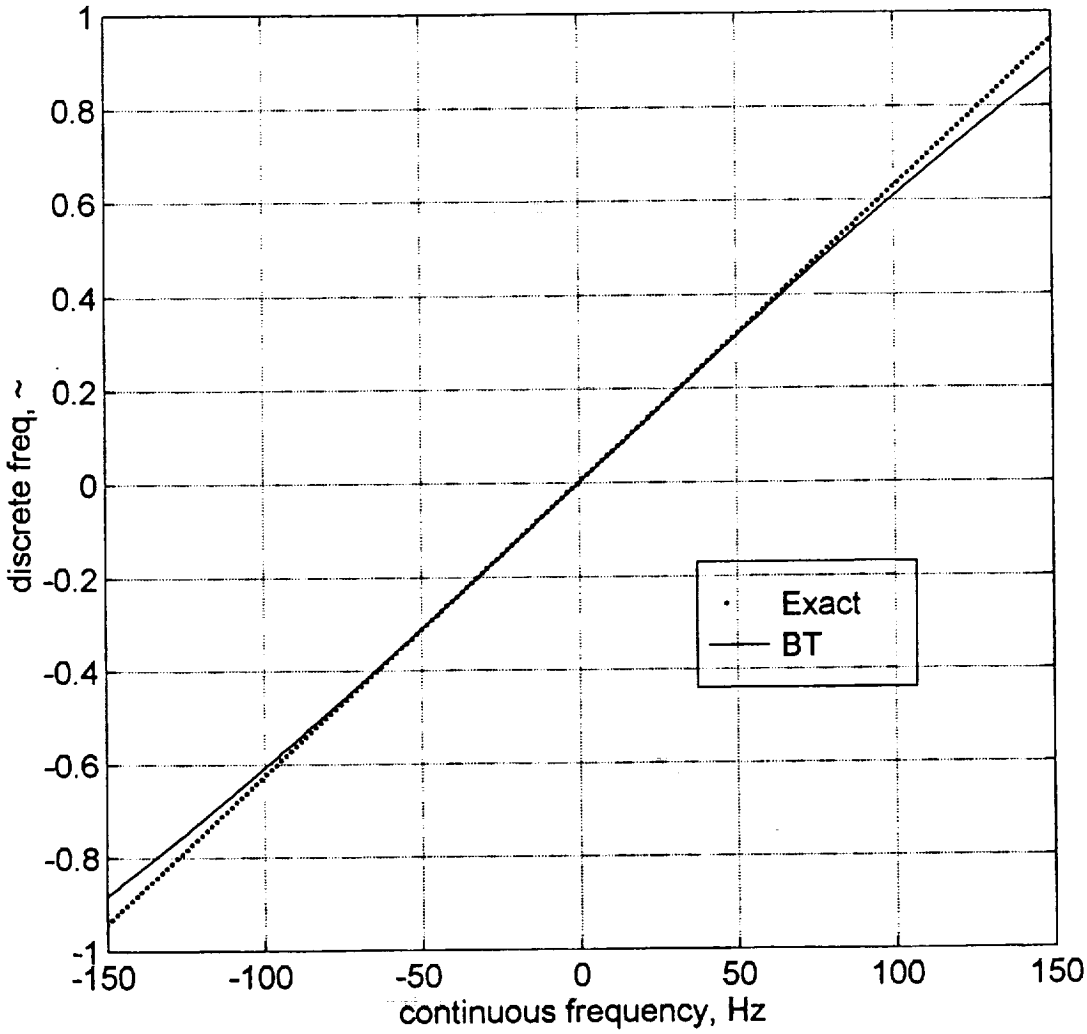

Figure 5.18: Frequency Warping Associated with the Bilinear Transformation (f_s = 1000 Hz)

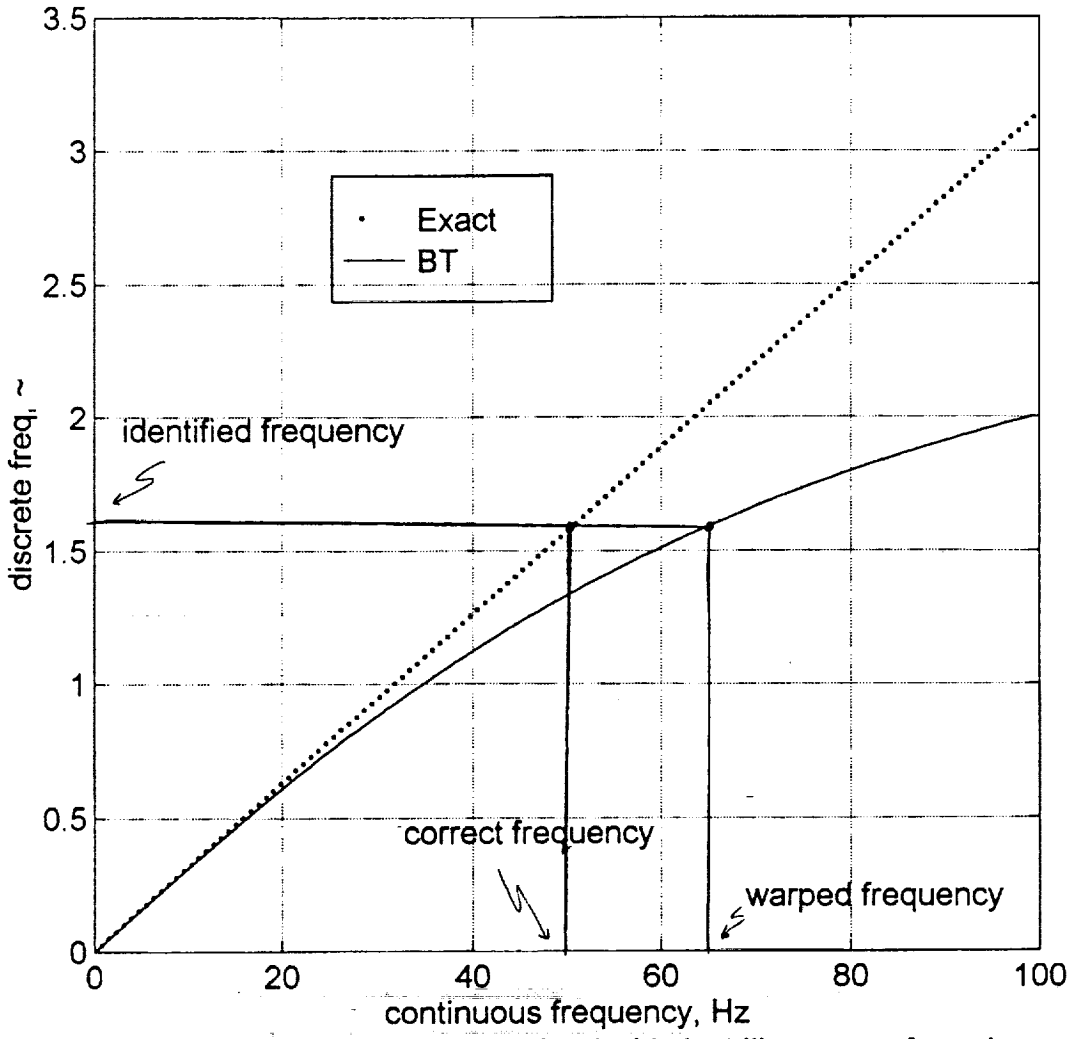

Figure 5.19: Frequency Warping Associated with the Bilinear Transformation (f_s = 200 Hz)

response hinted to this effect. As shown in Figure 3.10, the second mode natural frequency of the spring-mass system shows up in the FFT plot at 44 Hz instead of the actual 52 Hz. This is an indication of the reverse warping effect that is the result of going from the continuous-time domain into the discrete domain via the BT, whereas the effect seen in the real-time testing is the reverse of that, i.e. going from the discrete-time domain into the continuous-time domain. When the two routes are put together, the problem automatically corrects itself. Note that this problem of frequency warping which becomes apparent at low sampling rates probably has nothing to do with the mode-skipping problem discussed at the end of Chapter 3 which usually occurs at high sampling rates.

At this point, it becomes necessary to study the effect of warping on damping and mode shapes. Recall equations (2.8)-(2.10) from chapter 2

$$\omega_i = 2f_s\sqrt{\frac{1+a_{i1}+a_{i2}}{1-a_{i1}+a_{i2}}} \tag{2.8}$$

$$\zeta_i = \frac{2f_s(1-a_{i2})}{\omega_i(1-a_{i1}+a_{i2})} \tag{2.9}$$

$${}_iA_{pq} = \frac{4b_i}{1-a_{i1}+a_{i2}} \tag{2.10}$$

Going back to the original mathematical development of the system in Chapter 2, one can easily see that equations (2.8)-(2.10) are the result of the inverse BT which means that all of them are affected by the warping effect. In addition, the modal parameters of each mode are functions of the filter coefficients and sampling rate that appear in the natural frequency equation which means that they will be affected by frequency warping. Such an effect has to be studied and verified empirically in MATLAB using simulation [31]. The idea is to simulate the continuous-time system response instead of the discrete-time system response using the lsim command instead of dlsim which has been used so far. This method of simulation would duplicate the real-time environment more accurately than the original method. In addition, it could also be used to either buttress the frequency warping argument and show its effect on damping and mode shapes or negate it. A sample test run was done and the results were as follows

Actual modal parameters:	ω_1 = 11.8 Hz	ω_2 = 52 Hz
	ζ_1 = 0.003	ζ_2 = 0.013
	${}_1A_{pq}$ = 0.166	${}_2A_{pq}$ = -0.166
Test results using lsim:	ω_1 = 11.9 Hz	ω_2 = 67.5 Hz
	ζ_1 = 0.003	ζ_2 = 0.0213
	${}_1A_{pq}$ = 0.167	${}_2A_{pq}$ = -0.215

As expected, these results show clearly the effect of warping on all modal parameters of the second mode. All three parameters came out significantly higher than they should be, which means that the effect of frequency warping can also be seen in simulation if the continuous-time response is used to train the adaptive filter instead of the discrete-time response. As discussed earlier, simulating the system response using dlsim with the tustin method masks the frequency warping effect because it is automatically corrected when the modal parameters are calculated using the filter coefficients. This masking of the warping effect is eliminated by using lsim instead of dlsim. As a result the modal parameters show up at higher values than expected, thus supporting the warping effect argument.

Case 2 (Figure 5.20): effective sampling rate: $f_s = 160$ Hz
learning rate: $\alpha = 0.008$
non-collocated response measurement

This case is included to prove that reducing the sampling rate further, causes the second mode to show up as a higher frequency mode due to frequency warping.

Case 3 (Figure 5.21): effective sampling rate: $f_s = 200$ Hz
learning rate: $\alpha = 0.01$
collocated response measurement (at center)

Similar to case 1 in all aspects except for the modal constant results. This case has excitation and response measurements collocated and is done to verify the mode shapes and see if the results given are reasonable. Figure 5.16 shows the first modal amplitude converging to a value near 2.0, and the second converging to almost -5.0. On the other hand, for the same point test, the first modal amplitude is roughly around 0.3 and the second is around 1.3. Summarized, these results are:

$$
\begin{aligned}
{}_1A_{21} &= 2.0 \\
{}_2A_{21} &= -5 \\
{}_1A_{11} &= 0.3 \\
{}_2A_{11} &= 1.3
\end{aligned}
$$

where ${}_iA_{pq}$ is the modal amplitude of the ith mode excited at point q and the response measurement taken at p. Now solving for the mode shape coefficients given by equation 2.2 in chapter 2:

$$ {}_iA_{pq} = {}_i\phi_p \, {}_i\phi_q $$

gives the following mode shapes:

$$ \Phi_1 = \begin{bmatrix} {}_1\phi_1 \\ {}_1\phi_2 \end{bmatrix} = \begin{bmatrix} 0.55 \\ 3.65 \end{bmatrix}, \qquad \Phi_2 = \begin{bmatrix} {}_2\phi_1 \\ {}_2\phi_2 \end{bmatrix} = \begin{bmatrix} 1.14 \\ -4.39 \end{bmatrix} $$

normalize for effective comparison with theoretical values to get

$$ \Phi_1 = \begin{bmatrix} {}_1\phi_1 \\ {}_1\phi_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 6.66 \end{bmatrix}, \qquad \Phi_2 = \begin{bmatrix} {}_2\phi_1 \\ {}_2\phi_2 \end{bmatrix} = \begin{bmatrix} 1 \\ -3.86 \end{bmatrix} $$

but the theoretical mode shapes calculated using formulas taken from Reference [8] are:

$$ \Phi_1 = \begin{bmatrix} {}_1\phi_1 \\ {}_1\phi_2 \end{bmatrix} = \begin{bmatrix} 0.68 \\ 2.00 \end{bmatrix}, \qquad \Phi_2 = \begin{bmatrix} {}_2\phi_1 \\ {}_2\phi_2 \end{bmatrix} = \begin{bmatrix} 1.43 \\ -2.00 \end{bmatrix} $$

normalized:

$$ \Phi_1 = \begin{bmatrix} {}_1\phi_1 \\ {}_1\phi_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2.94 \end{bmatrix}, \qquad \Phi_2 = \begin{bmatrix} {}_2\phi_1 \\ {}_2\phi_2 \end{bmatrix} = \begin{bmatrix} 1 \\ -1.4 \end{bmatrix} $$

As shown, the real-time mode shapes are exaggerated and could very well be in error. For one thing, these values are affected by frequency warping and they were measured from the plots which are not very accurate because they were taken under the assumption that they were the final convergence values which might not be true.

Also, there is always the error introduced by the vibration of the workbench and the steel mounting frame. In addition, as discussed in Chapter 2, this is not a very accurate test because the mode shapes are evaluated at two different test runs instead of one. This is the result of the limited number of channels in the C40 system. Instead of performing the test with three sensors (a force transducer and two accelerometers), the test is done twice with two sensors. However, the results are not completely useless, they do indicate the correct general shape of each mode.

Case 4 (Figure 5.22): effective sampling rate: $f_s = 200$ Hz
learning rate: $\alpha = 0.03$
non-collocated response measurement

Maintaining the sampling rate and increasing the learning rate has improved the speed of convergence noticeably. On the other hand, the smoothness of the curves is much worse. This case illustrates the trade-off between stability and speed.

Case 5 (Figure 5.23): effective sampling rate: $f_s = 250$ Hz
learning rate: $\alpha = 0.03$
non-collocated response measurement

This case is included to illustrate the effect of increasing the sampling rate on the frequency warping problem. Figure 5.23 shows that the second mode frequency converges to a value between 65 and 70Hz, which is less than that identified at 200Hz sampling. This proves that sampling higher, forces the BT to operate in the linear region or close to it, thus reducing the error due to warping. However, any hopes of seeing better results by increasing the sampling rate vanish quickly as the mode-skipping phenomenon which was discussed in section 3.2 comes into play.

Case 6 (Figure 5.24): effective sampling rate: $f_s = 400$ Hz
learning rate: $\alpha = 0.001$
non-collocated response measurement

The first mode as identified here is actually the second mode, and the identification is almost perfect, with the exception of the damping ratio which is slightly high. It appears that the adaptive filter completely ignored the first mode, treated the second mode as the first and tried to identify something that has a frequency much higher than that of the second mode. This problem of totally skipping the first mode was seen in the simulation as well and is found to be common here in almost all cases that had a sampling rate higher than 250-300 Hz. It must be noted that many test runs were done to try and improve the results of this case by both increasing and decreasing the learning rate. It was found that this did not have any significant effect on the final values, they almost always came out similar to the above results.

Case 7 (Figure 5.25): effective sampling rate: $f_s = 500$ Hz
learning rate: $\alpha = 0.01$
non-collocated response measurement

Same problem as seen above, actual second mode appears as first mode, but with an impressive rate of convergence, almost instantaneous identification. In place of the actual second mode, it seems like the adaptive filter is trying to identify something adjacent to the actual second mode.

This supports the argument made at the end of Chapter 3 as a possible cause of the mode-skipping problem. Figure 5.26 shows that the algorithm does actually try to minimize the error.

Since the second natural frequency of the beam (50 Hz) appears at a frequency adjacent to that of the electrical power frequency (60Hz), then it is possible that interference causes this behavior. However, this was not found to be a conclusive explanation because using different lengths of the beam, which shifts the mode location on the frequency spectrum, and even using a totally different structure did not support this argument, the problem still existed as discussed above. The other structure that was used here was a composite ski that was clamped in a cantilever mount. Figure 5.27 shows the ski acceleration response and the FFT of that response. The first mode of vibration is at slightly less than 10 Hz and the second mode appears around 30 Hz. The results obtained using the ski are very similar in nature to those of the beam. The same problems associated with low and high sampling (namely warping and mode-skipping) were seen in the ski tests which for one thing eliminate the possibility of the electrical power frequency being a source of error.

Case 8 (Figure 5.28): effective sampling rate: $f_s = 1000$ Hz
learning rate: $\alpha = 0.01$
different points measurement

Sampling at 1 kHz does not improve the results. As a matter of fact, the apparent second mode is converging to an even higher value than seen before. The error behavior as shown in Figure 5.26 is similar to that of case 7.

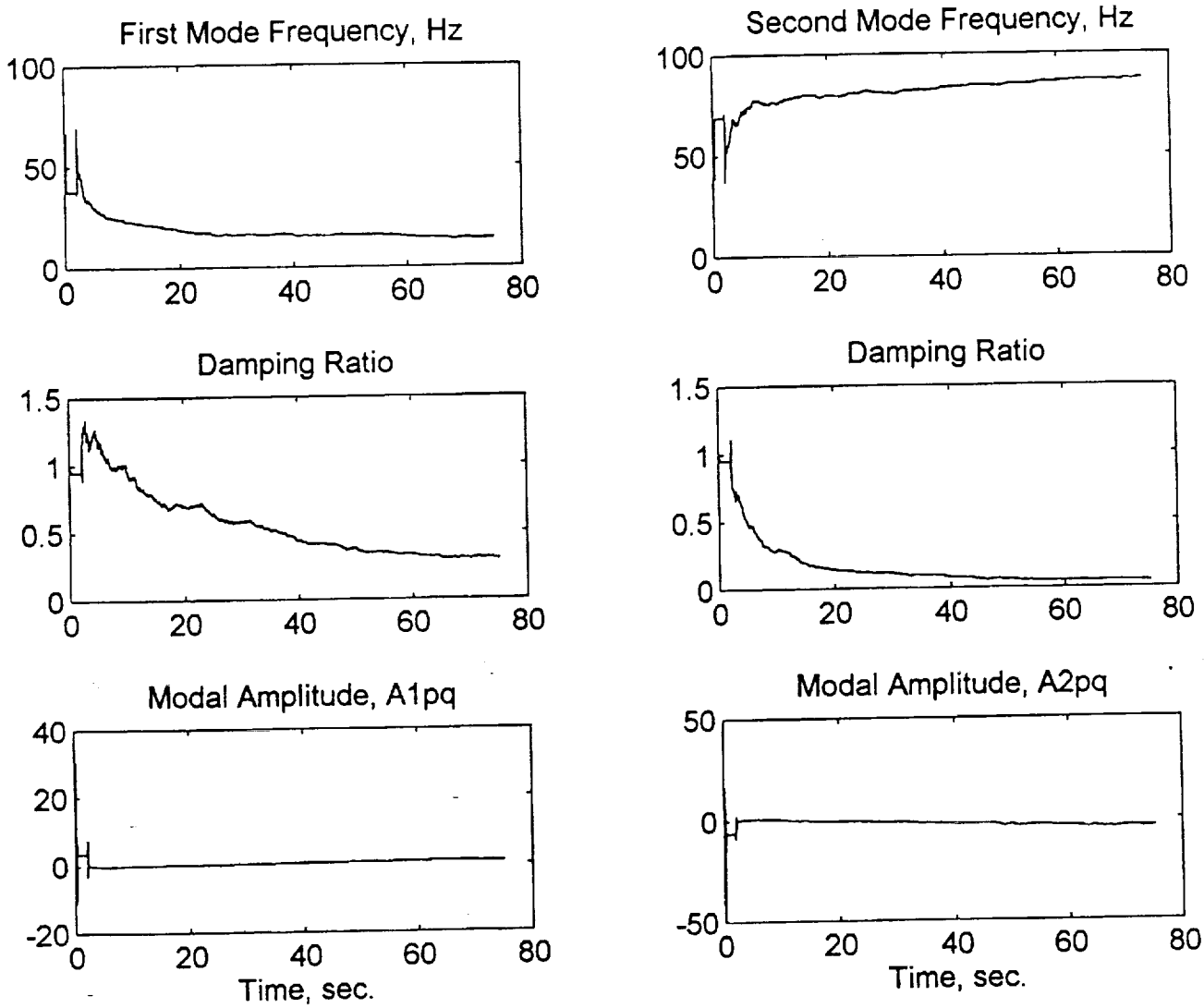

Figure 5.20: Two Mode Real-Time System Identification Results
(Case 2: f_s = 160 Hz, α = 0.008)

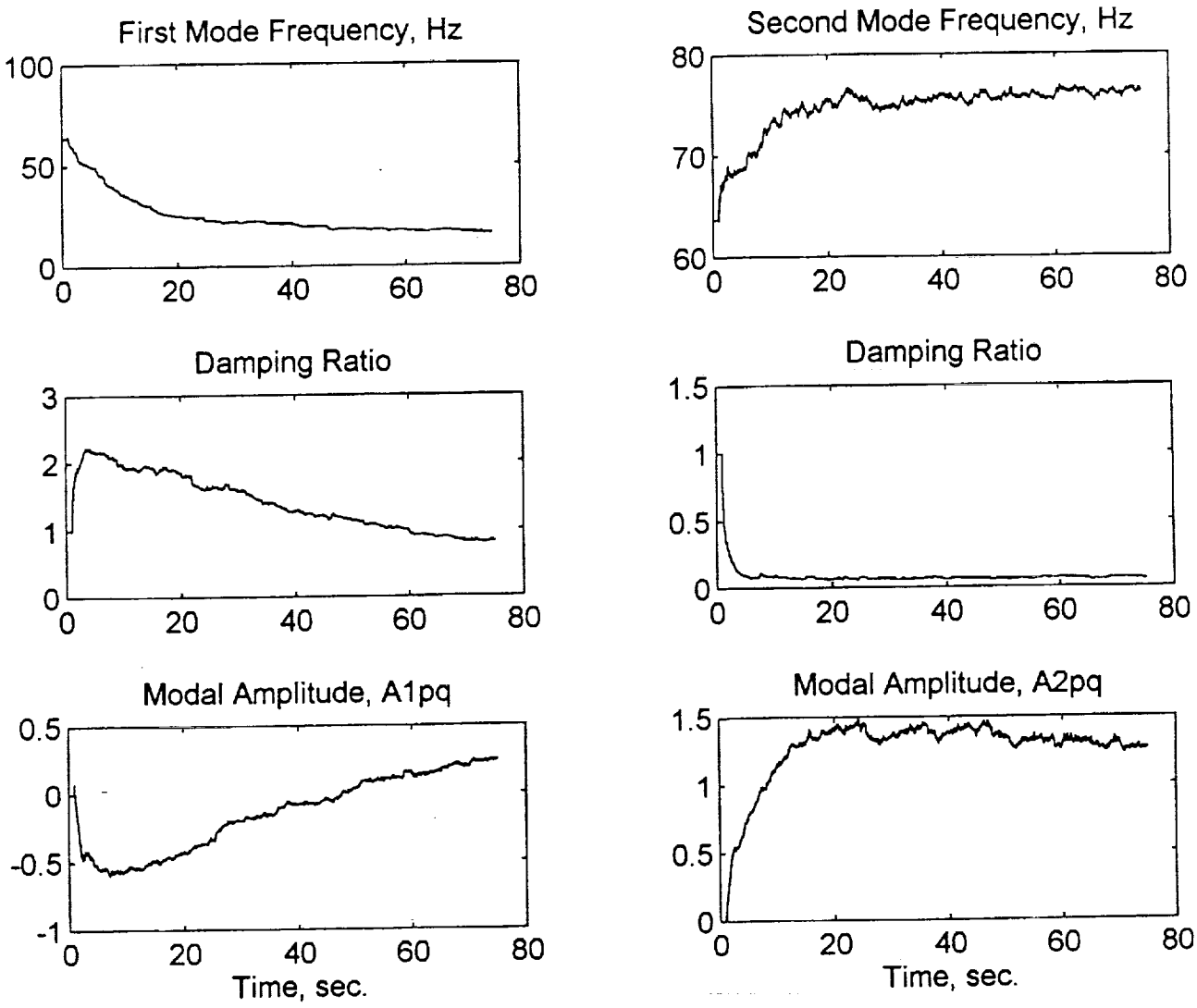

Figure 5.21: Two Mode Real-Time System Identification Results
(Case 3: f_s = 200 Hz, α = 0.01, collocated response measurement)

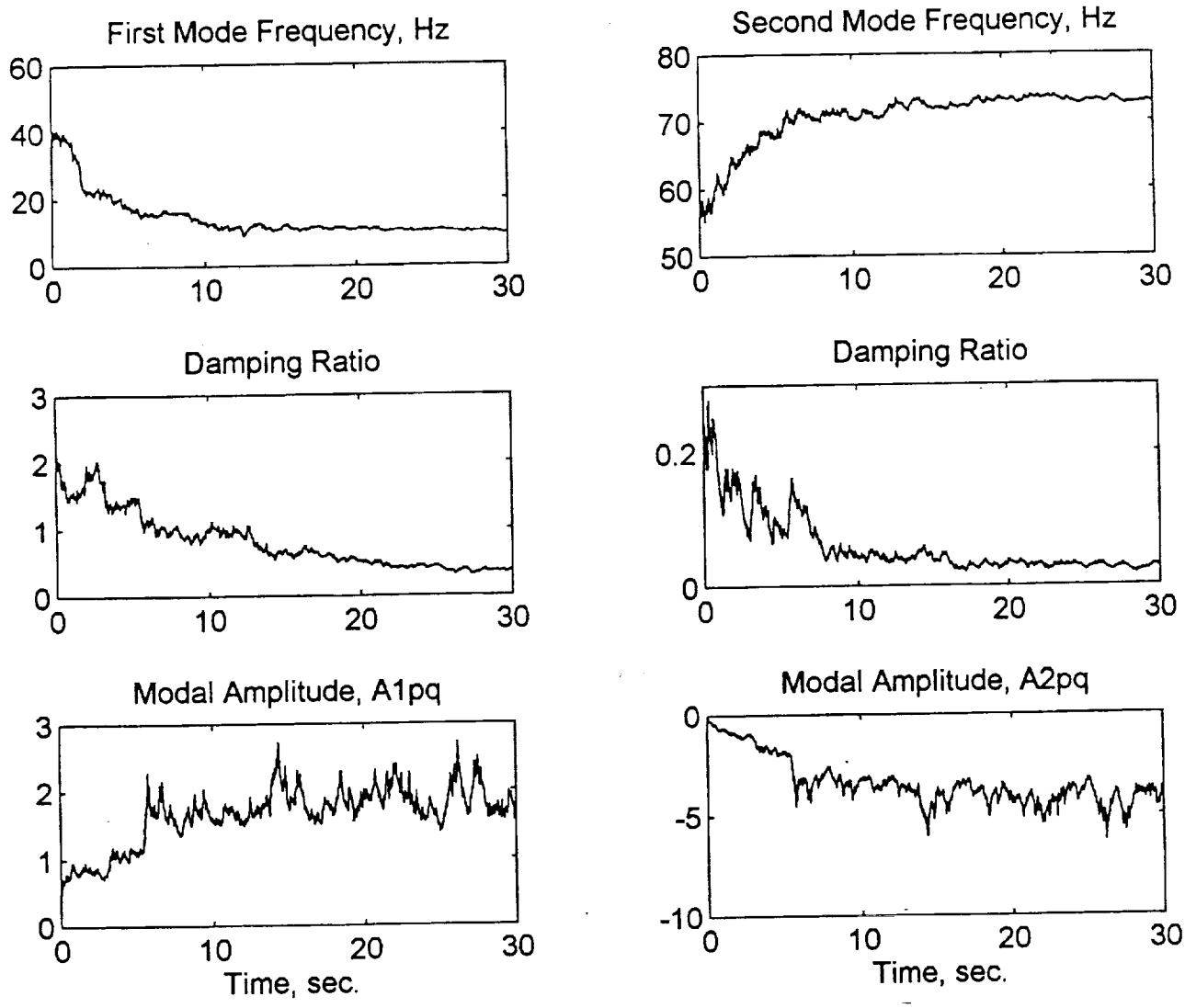

Figure 5.22: Two Mode Real-Time System Identification Results
(Case 4: f_s = 200 Hz, α = 0.03)

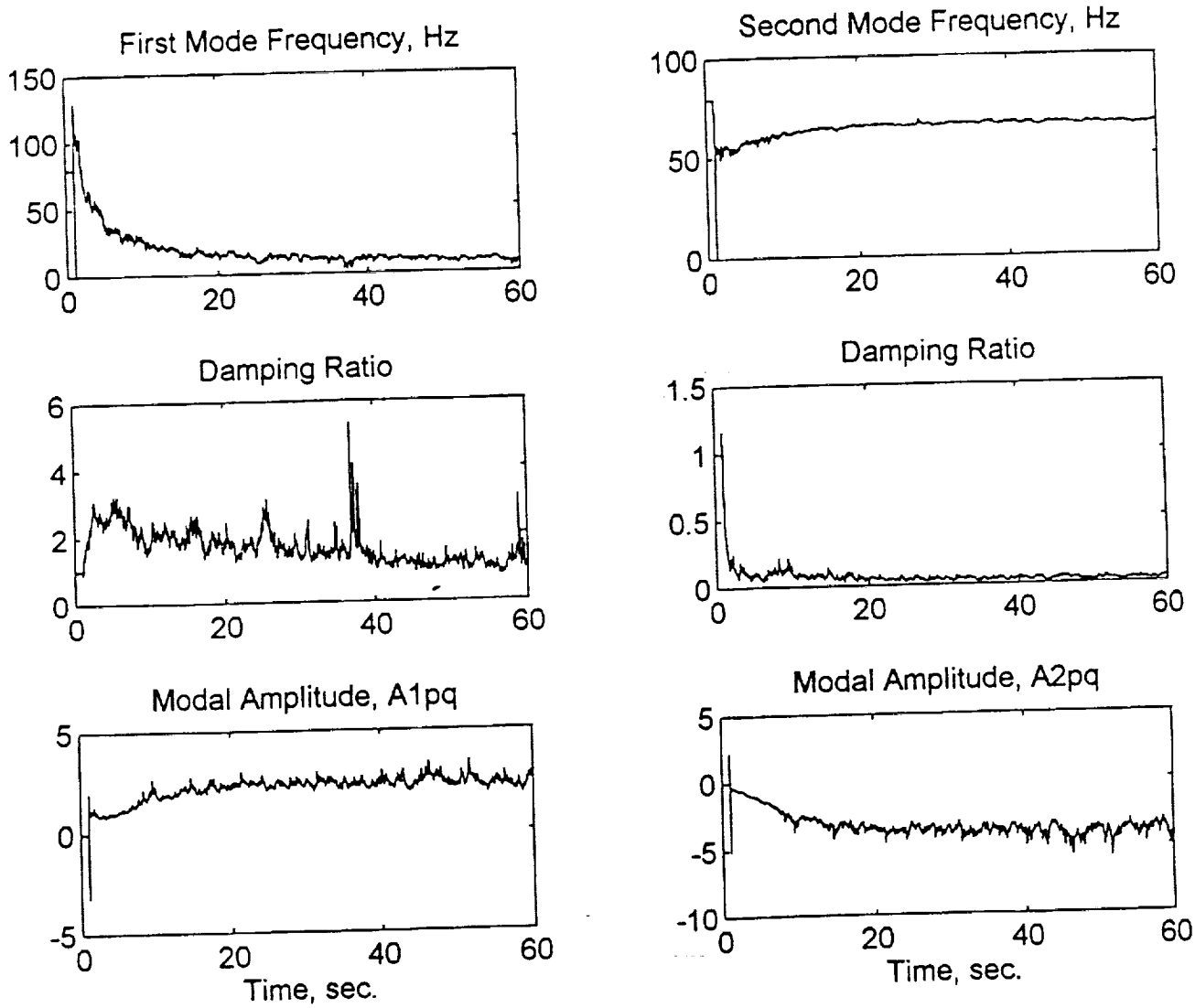

Figure 5.23: Two Mode Real-Time System Identification Results
(Case 5: f_s = 250 Hz, α = 0.03)

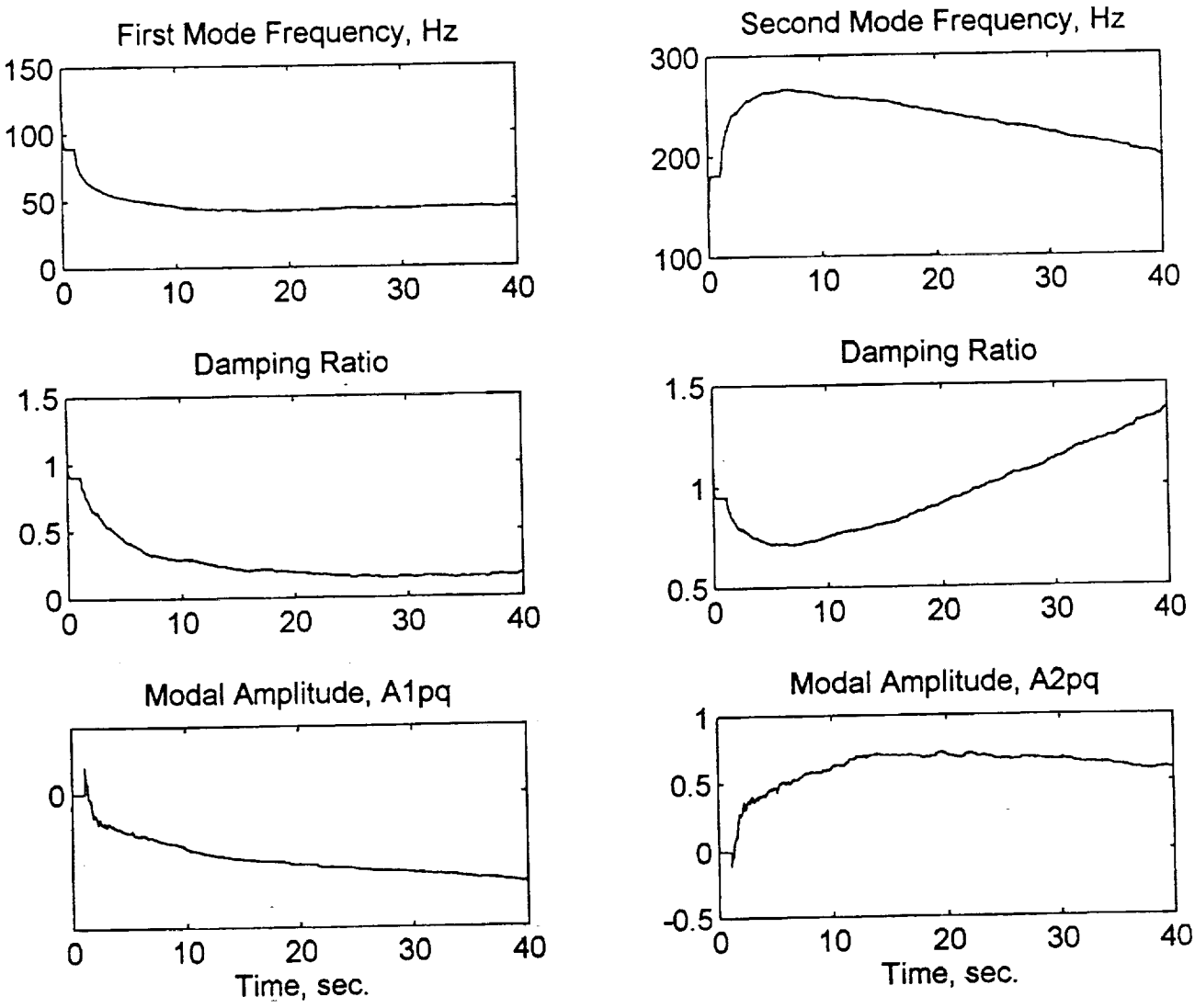

Figure 5.24: Two Mode Real-Time System Identification Results
(Case 6: f_s = 400 Hz, α = 0.001)

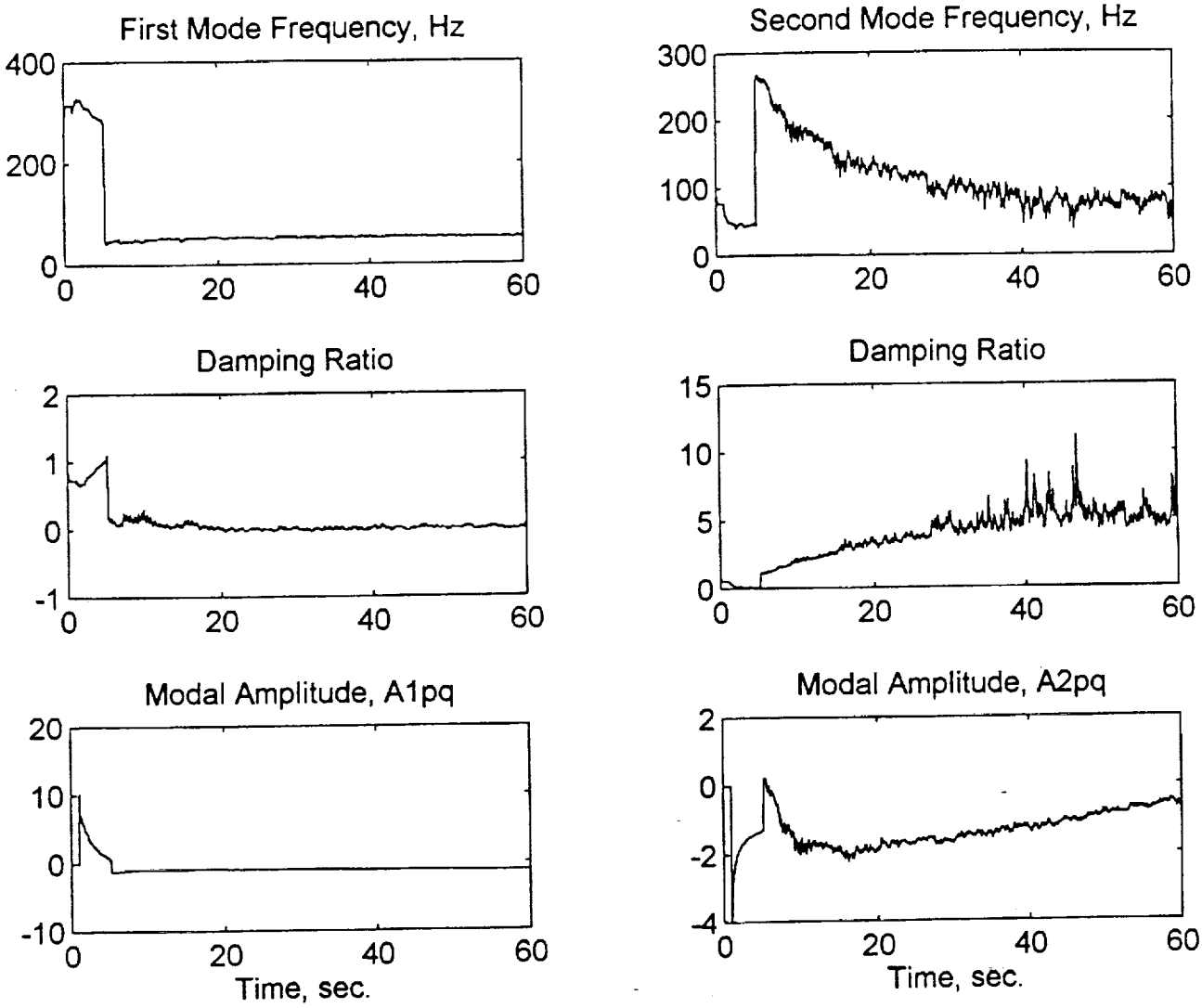

Figure 5.25: Two Mode Real-Time System Identification Results
(Case 7: f_s = 500 Hz, α = 0.01)

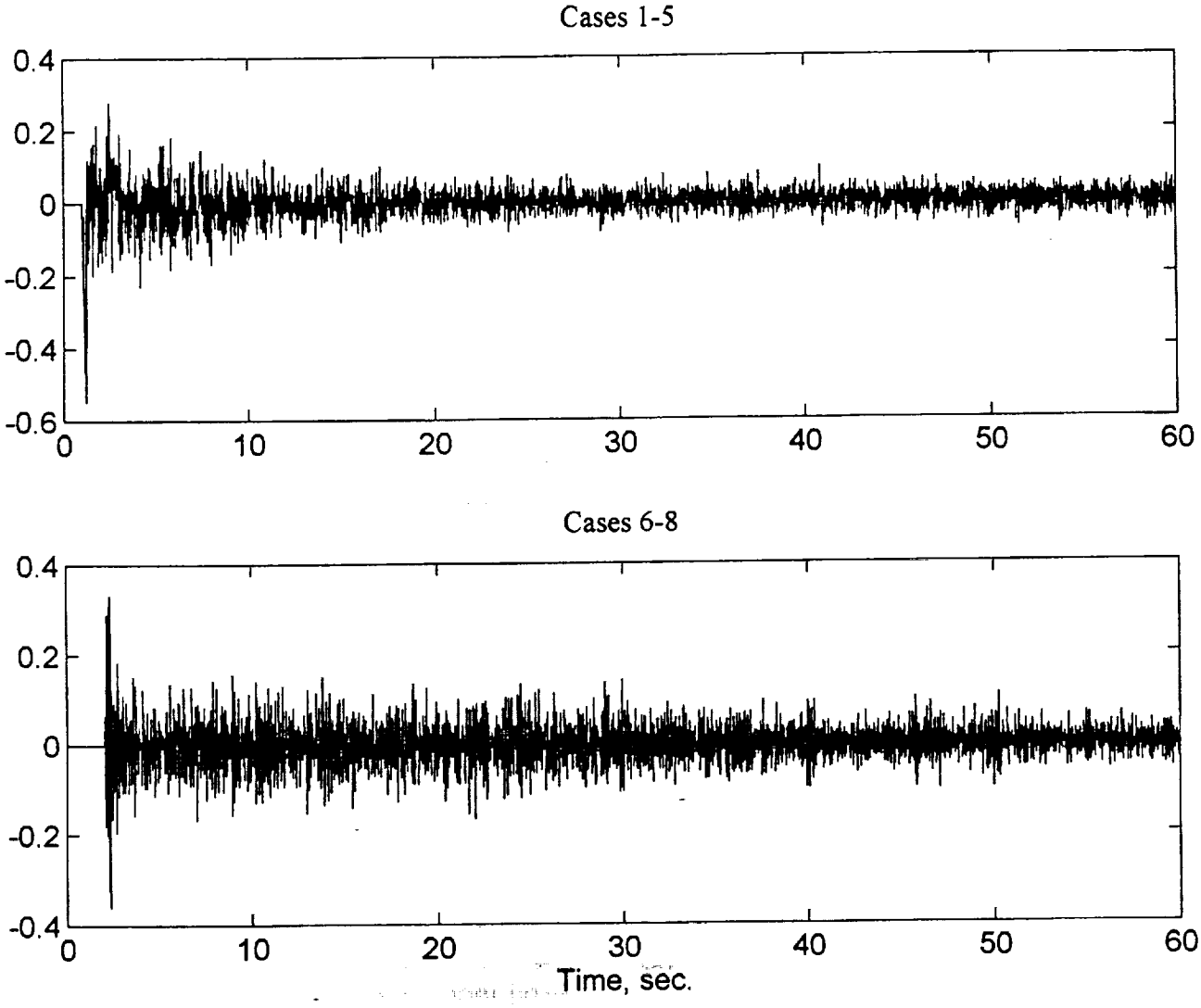

Figure 5.26: Representative Error Time Histories for the Two Mode Real-Time Test Cases

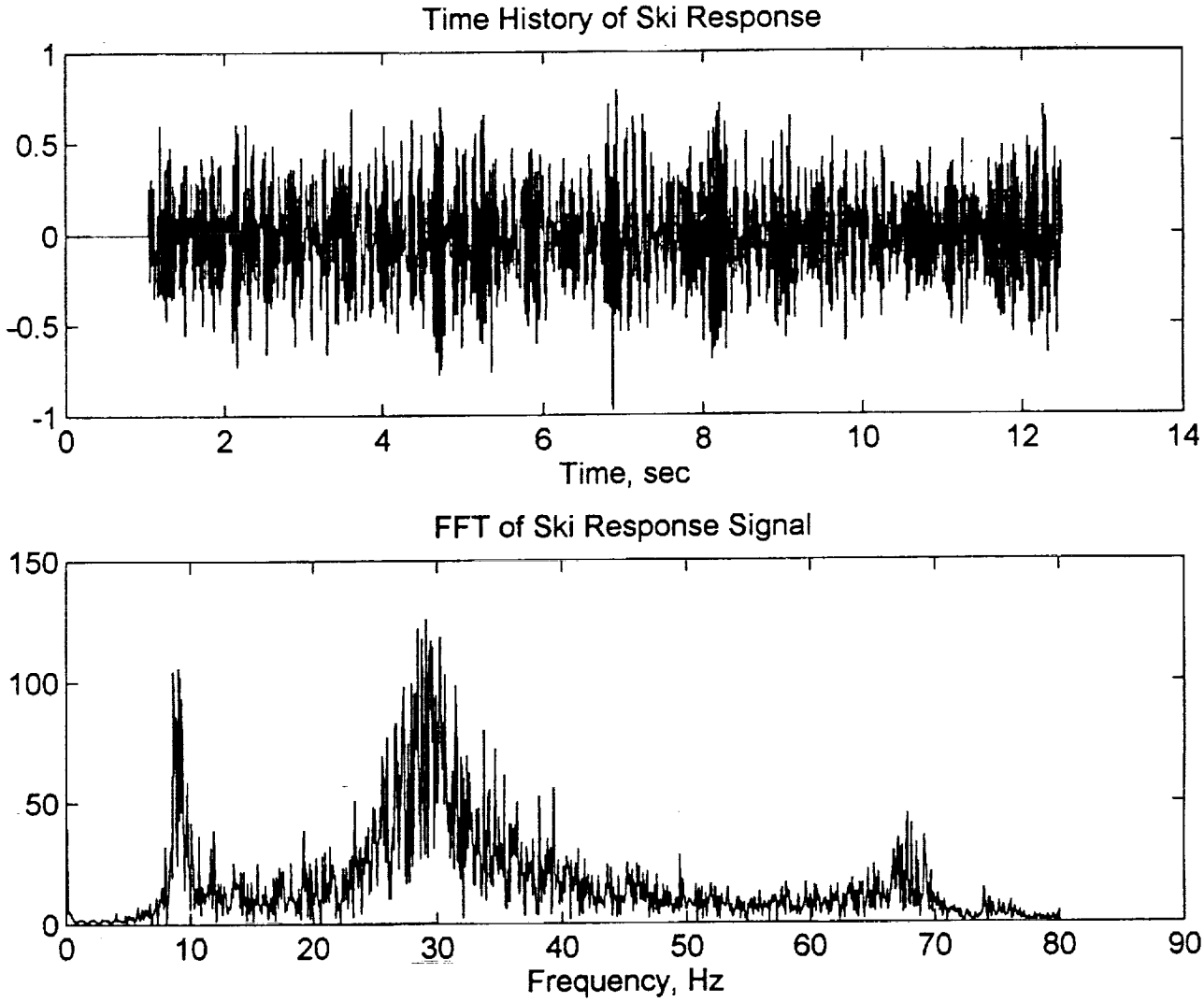

Figure 5.27: Ski Response and FFT Plot Showing the First Two Modes of Vibration, (f_s = 250 Hz)

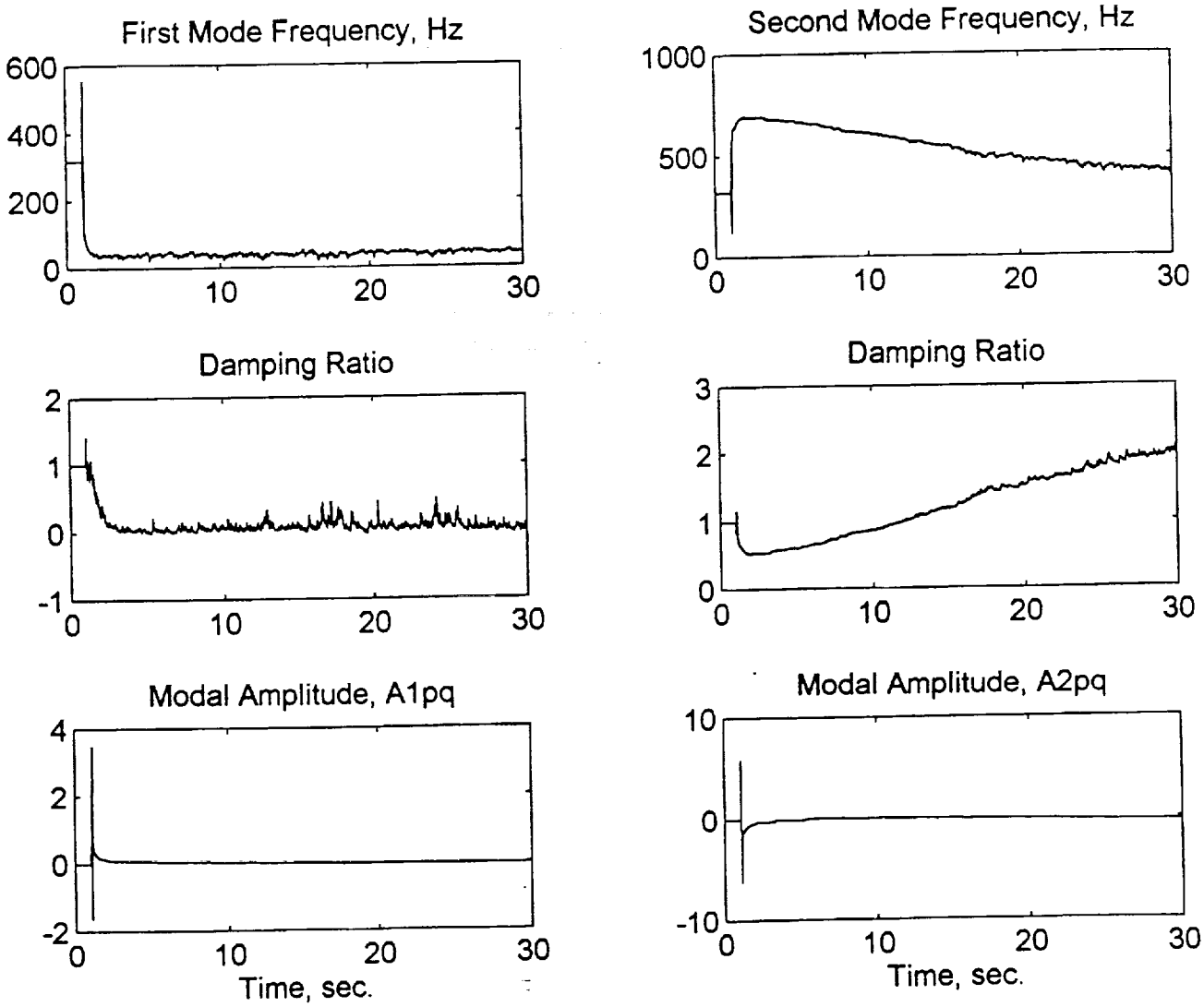

Figure 5.28: Two Mode Real-Time System Identification Results
(Case 8: f_s = 1000 Hz, α = 0.01)

5.4.2.1 Single DOF Algorithm Applied to Two DOF System with Filtering

This section is included to see how the SDOF algorithm applied twice to the two DOF system with the aid of band-pass filtering will perform. The idea is to try and isolate each mode and apply the SDOF algorithm to it separately in real-time. This idea is similar to work done by Lim, Cabell and Silcox [1]. As shown in the sample code **C40DDOFF.C** in Appendix D, a 4th order low-pass digital Butterworth filter is used to isolate the first mode of the beam (ω_c = 20Hz), and another 4th order band-pass digital Butterworth filter is used to isolate the second mode (ω_L = 30Hz, ω_H = 70Hz). Figure 5.29 shows the FFT of the beam response before and after applying each filter separately. Each mode is effectively isolated. The filters are also applied to the excitation signal. It is realized that this technique is outside the main focus of the thesis which is the simultaneous multi-mode identification. Therefore, only three cases are discussed below to summarize the results of this technique which could help in drawing a better picture of the behavior of the adaptive filter.

Case 1 (Figure 5.30):
effective sampling rate: f_s = 400 Hz
first learning rate: α_1 = 0.003
second learning rate: α_2 = 0.015
non-collocated response measurement

Both modes are correctly identified. However, the first mode results are not very good. The damping drifts around for a while before converging and the modal constant is slowly climbing up to some unknown value. The second mode on the other hand looks very good. The frequency converges to a value slightly higher than the expected 50Hz (effect of warping). The damping converges to an average value of 2.3% which is also acceptable and the modal constant seems to be moving towards some negative value. This case and following cases show that the SDOF can very efficiently identify the second mode with absolutely no trouble at all. However, the problem with this method is that one sampling rate is used for both modes which means that using a low sampling rate in order to have a good identification of the first mode will cause the second mode to be undersampled and give inaccurate results. Conversely, if a high sampling rate is used to cater to the second mode, the first mode identification process starts having problems and becomes very sensitive to the learning rate as discussed is section 5.4.1.

Case 2 (Figure 5.31):
effective sampling rate: f_s = 400 Hz
first learning rate: α_1 = 0.004
second learning rate: α_2 = 0.015
non-collocated response measurement

Slightly increasing the first mode learning rate produces a slight improvement in the first modal constant.

Case 3 (Figure 5.32):
effective sampling rate: f_s = 400 Hz
first learning rate: α_1 = 0.01
second learning rate: α_2 = 0.015
non-collocated response measurement

Increasing the learning further, causes the floating point overflow in the first mode damping as a result of the frequency becoming complex. The first modal constant is trying to approach the expected value of 1.0.

In conclusion, the filtered two DOF method discussed here does actually work and can be effectively implemented in real-time as shown, however choosing appropriate sampling and learning rates to cater to both modes can be a tricky problem. A possible solution is to use two different interrupt service routines (ISR) that can be executed at different sampling rates, but that is definitely outside the main focus of this thesis. In addition, this method does not work very well for closely-spaced modes, because band-pass filtering does not become effective.

Remarks

Although the discussion of the mode-skipping phenomenon associated with simultaneous two mode identification at high sampling rates is mentioned first in Chapter 3, it was not noticed until the real-time testing began and the results were studied. When the real-time testing was first done at low sampling rates, the warping problem emerged. The conclusion was that increasing the sampling rate should solve the warping problem, and that was when the second phenomenon was noticed. For a while it was thought that this was the result of problems associated with the real-time testing. One of the main concerns was that the ISR was too long which could have been causing the C40 to skip samples thus giving a false sampling rate. This is discussed next.

Sampling Rate and ISR

The ISR execution time discussed in Chapter 4 was considered as a possible potential source of error. The sampling frequency used in the testing for this project is 4kHz. This was chosen because it was the lowest allowed by the internal circuitry of the C40 data acquisition system, yet it was not too low as to limit the use of a digital filter to overcome the aliasing problem. This sampling rate means that the time the ISR takes to execute entirely must not exceed 250μs, which translated in clock cycles means 12500 cycles. When checked in the debugger, the **?clk** command discussed in Chapter 4 indicated that the ISR of the two DOF identification code takes approximately 3683 clock cycles to execute. But this can not be taken at face value because in the ISR there is a math function call, the square root command (sqrt) needed to evaluate the natural frequency. In addition, there is a for statement needed for the Newton root finding algorithm. The sqrt command causes the program to branch off to the math library and evaluate the function. This can take more clock cycles than actually indicated; in addition, the stacking can get messed up which would also cause delays in the ISR. The repetitive nature of the for statement can cause it to take more clock cycles than predicted by the debugger. These problems are very hard to track down and verify; so as a rule of thumb, function calls and loops should not be placed in the ISR unless there is no other way. This was actually emphasized by the technical support staff at Spectrum Inc. and a DSP expert at NASA, Langely [32]. However, in the code used here, there are only two function calls made and the for statement necessary for root finding does not exceed 10 iterations, so it is assumed that even with these uncertainties, there is enough time for the ISR to execute completely. Nevertheless, this is not enough to draw a definite conclusion regarding this possibility. So data was collected using the C40 with limited code in the ISR and was imported into MATLAB where the adaptive filter was applied to the data and the results were consistent with those above. Therefore it was concluded that the ISR execution time is not a problem. At this point, the simulation code was tested to see if the same behavior exists at high sampling rates, and sure enough it was. This is when the mode-skipping issue was looked into. In conclusion, the simultaneous two DOF identification process has been found to work but with difficulties, namely those of warping and mode-skipping. The warping problem is well understood, but the mode-skipping problem is yet to be analyzed.

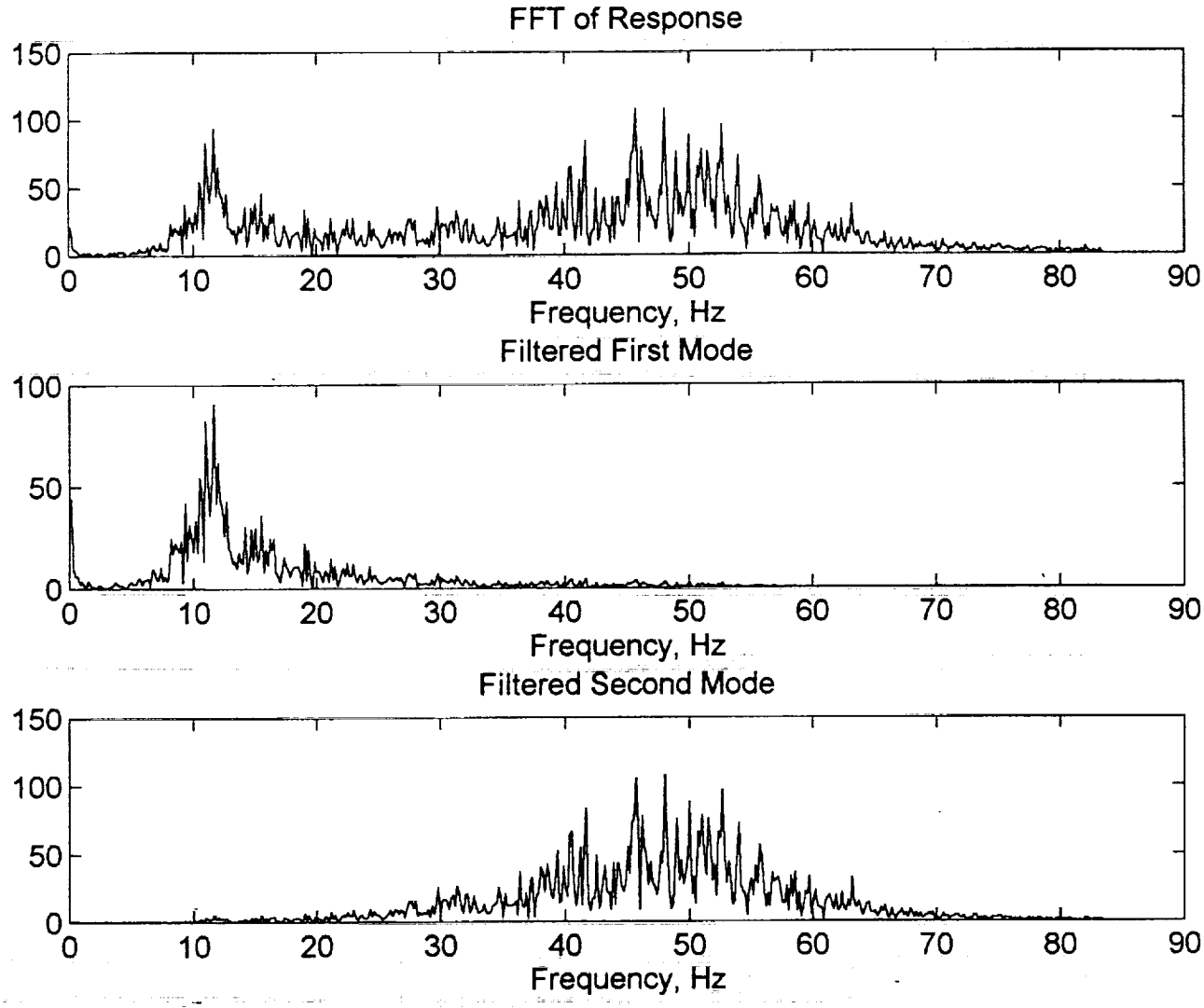

Figure 5.29: FFT of Beam Response Showing the Effectiveness of Band-Pass Filtering in Isolating the First and Second Modes (f_s = 500)

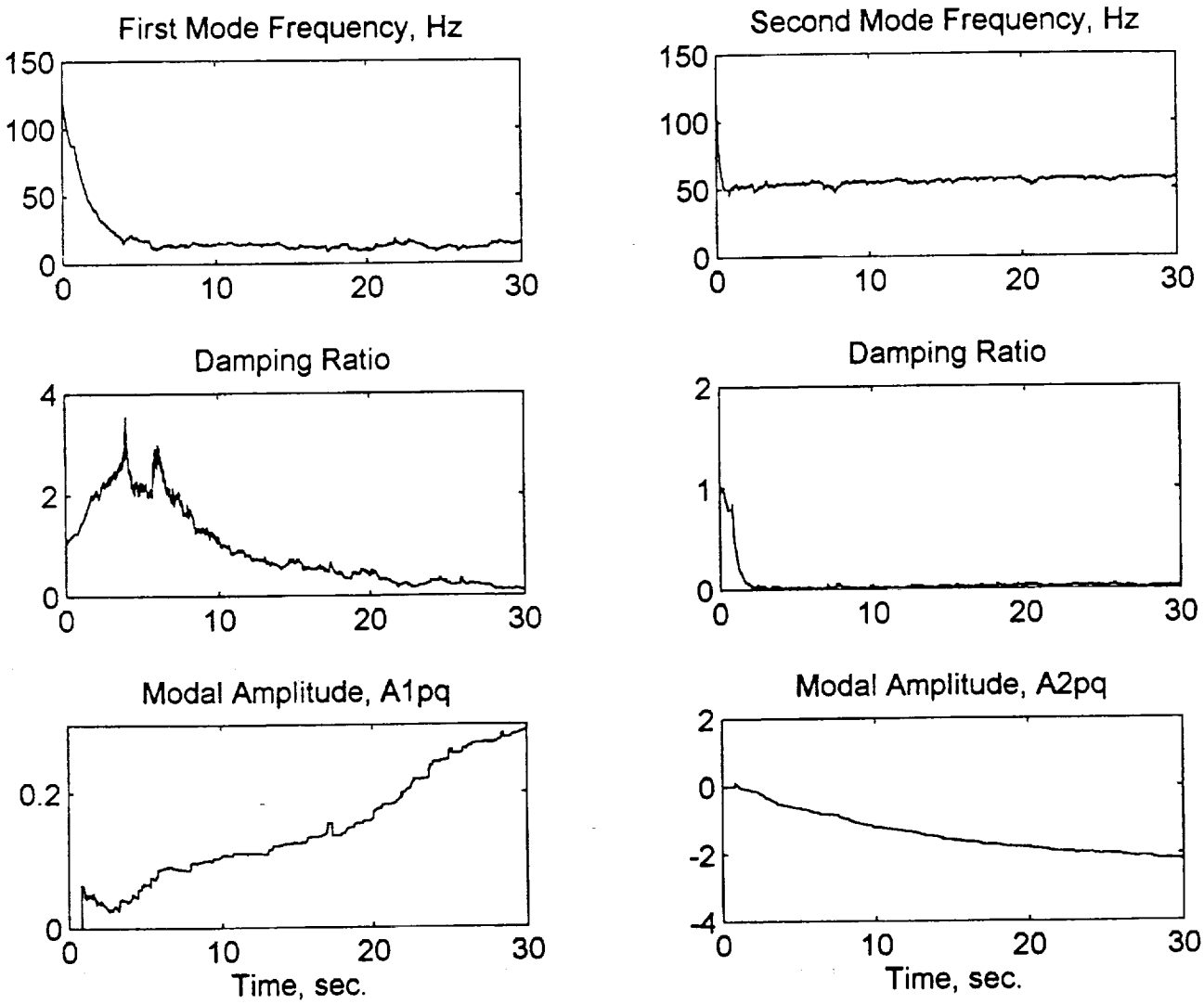

Figure 5.30: Two Mode Real-Time System Identification Results
(With Band-Pass Filtering)
(Case 1: f_s = 400 Hz, α_1 = 0.003, α_2 = 0.015)

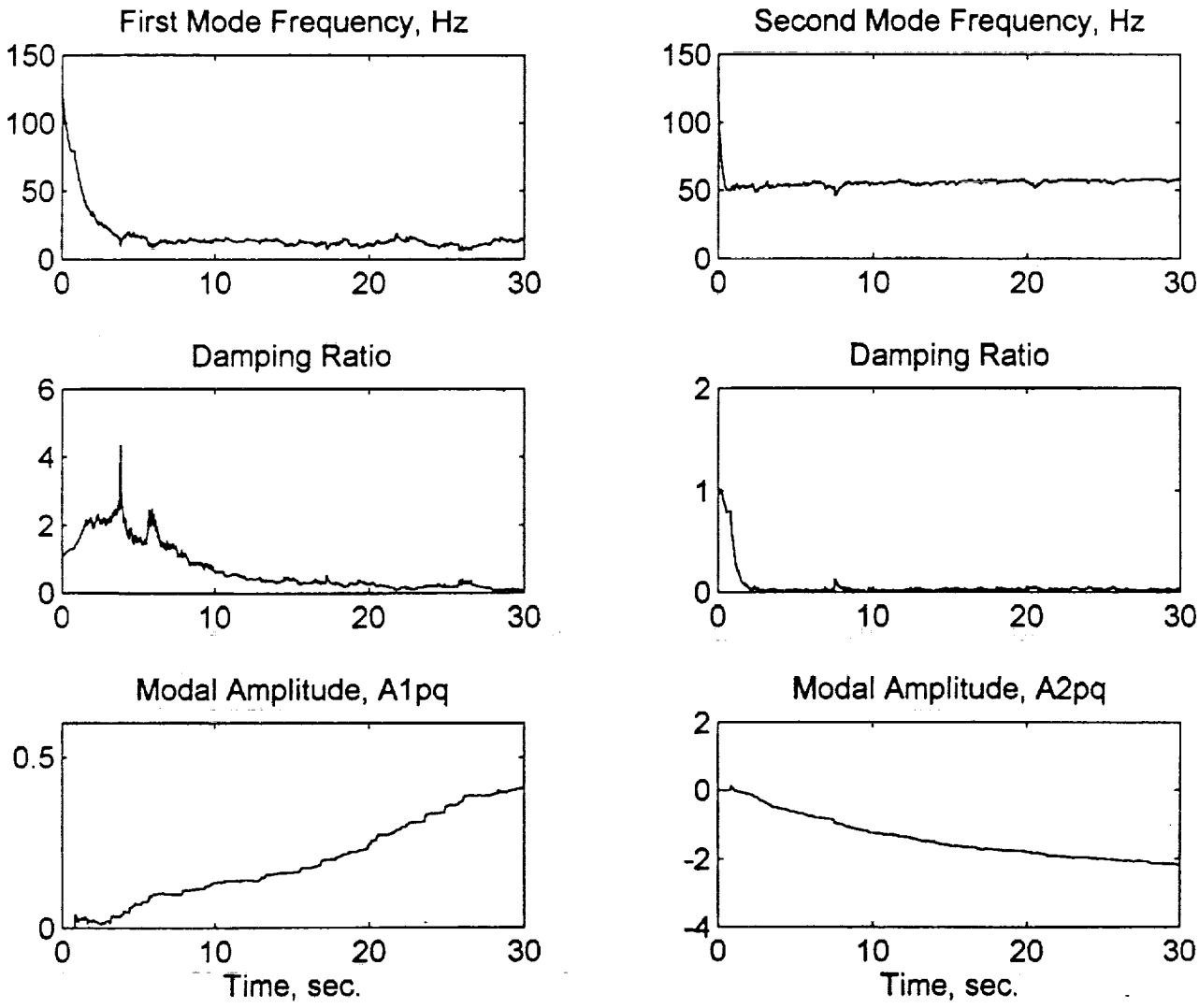

Figure 5.31: Two Mode Real-Time System Identification Results
(With Band-Pass Filtering)
(Case 2: f_s = 400 Hz, α_1 = 0.004, α_2 = 0.015)

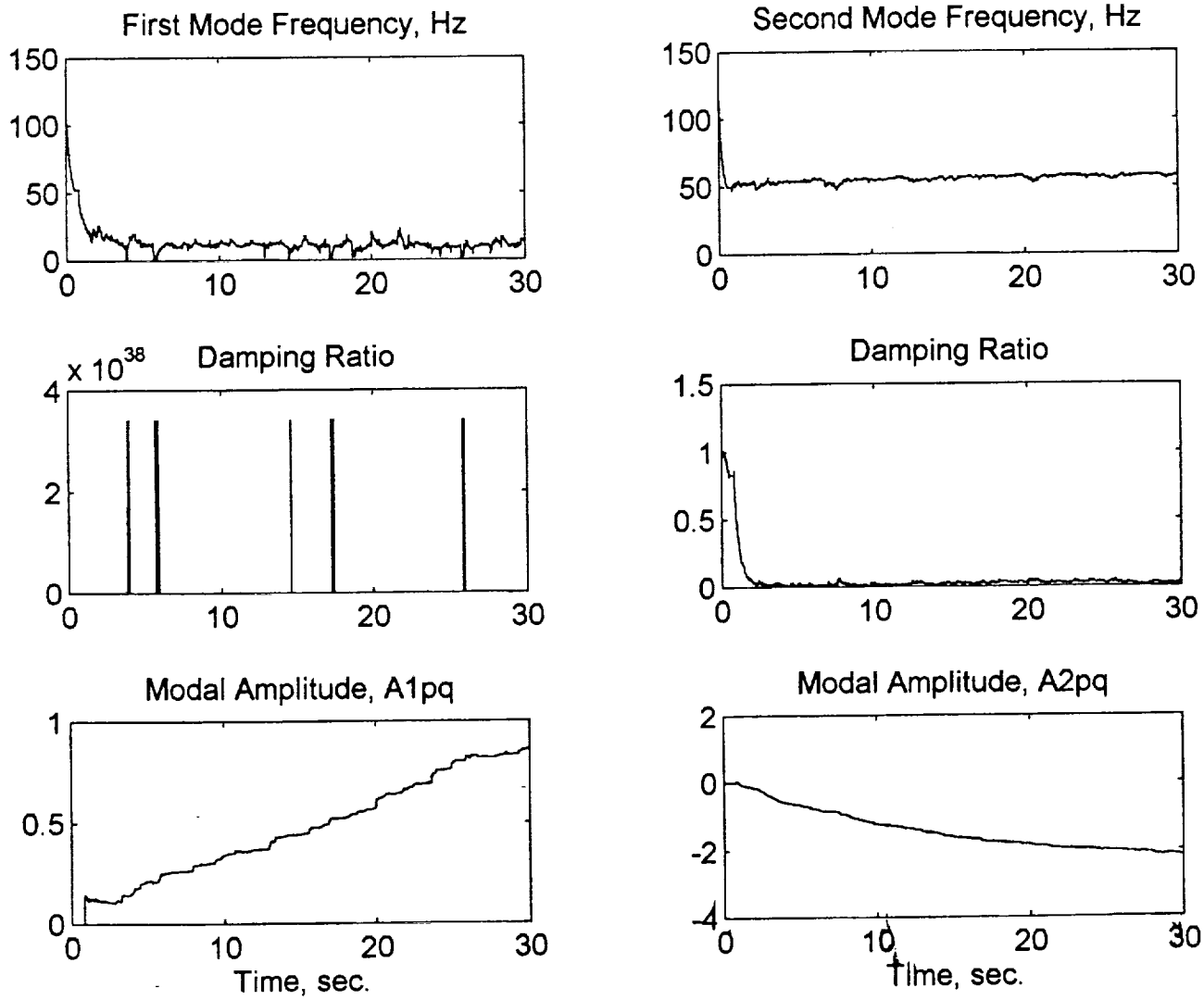

Figure 5.32: Two Mode Real-Time System Identification Results
(With Band-Pass Filtering)
(Case 3: f_s = 400 Hz, α_1 = 0.01, α_2 = 0.015)

5.5 Correction for Frequency Warping

In the previous section, the phenomenon described as frequency warping was presented. Due to the approximate nature of the bilinear transformation employed in the process of converting the identified filter coefficients into modal parameters, the frequency estimates are biased and the error tends to grow as the frequency gets closer to the sampling frequency. An alternative modal parameter extraction method is developed to mitigate the problem. The method along with a numerical example is presented herein.

In this new approach, rather than using the inverse bilinear transformation to identify the modal parameters, the definition, $z = e^{sT_s}$, is directly used to relate the filter coefficients in the z-domain to the modal parameters in the s-domain. As the first step of the approach, the poles are computed from the identified filter coefficients, i.e., the denominator of Eq. (2.12). Assuming the system is underdamped, the complex conjugate poles are obtained as

$$z = -\frac{a_1}{2} \pm j\sqrt{a_2 - \frac{a_1^2}{4}} \tag{5.6}$$

The poles in the z-domain are converted into the poles in the s-domain using

$$s = f_s \ln(z) \tag{5.7}$$

Then the undamped natural frequency and damping ratio are defined as

$$\begin{aligned} \omega_n &= \text{abs}(s) \\ \zeta &= \text{abs}(\text{real}(s))/\omega_n \end{aligned} \tag{5.8}$$

The corresponding mode shape amplitude is still defined using Eq. (2.10).

Numerical Example

An example is given here to demonstrate the performance of the new modal parameter identification process. Consider the single DOF example given in Section 3.1. In Section 3.1, the identified modal parameters appears to be exact and are free from frequency warping problems. This apparent accuracy stems from the fact that the bilinear transformation is used to discretize the continuous system and to conduct a response simulation. In other words, the bilinear transformation was used consistently for the conversions process between continuous and discrete systems. Because of this, the frequency warping was never noticed. However, in an experimental setting, the plant dynamics remains as a continuous system while input and output signals are sampled at a given frequency. To represent correctly the experimental setting, the response calculation should be conducted using a continuous simulation not a discrete simulation using a bilinear transformation. Figure 5.33 shows the results obtained using a continuous simulation response data with the old modal parameter extraction approach, i.e., Eqs. (2.8) and (2.9). As expected, the results produces a bias in the frequency and damping ratio estimates. The identified frequency and damping ratio are 12.2Hz and 0.012, respectively, compared to the correct values of 11Hz and 0.01. The results using the new modal parameter identification approach are shown in Fig. 5.34. The frequency and damping ratio are identified correctly.

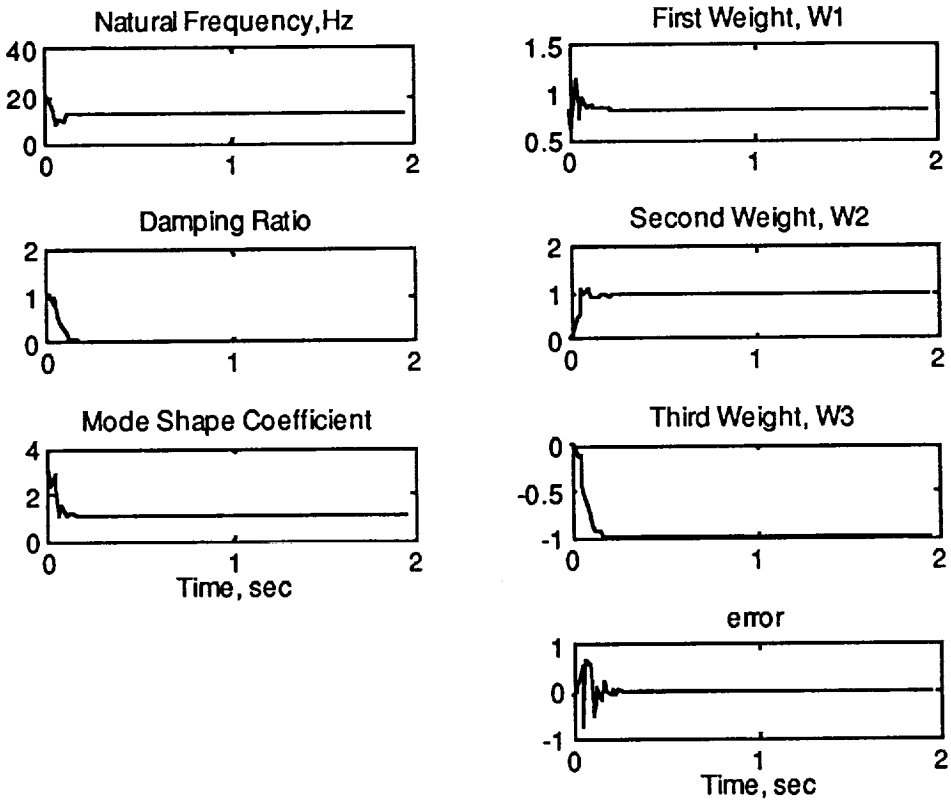

Fig. 5.33 Results of the old modal parameter identification approach

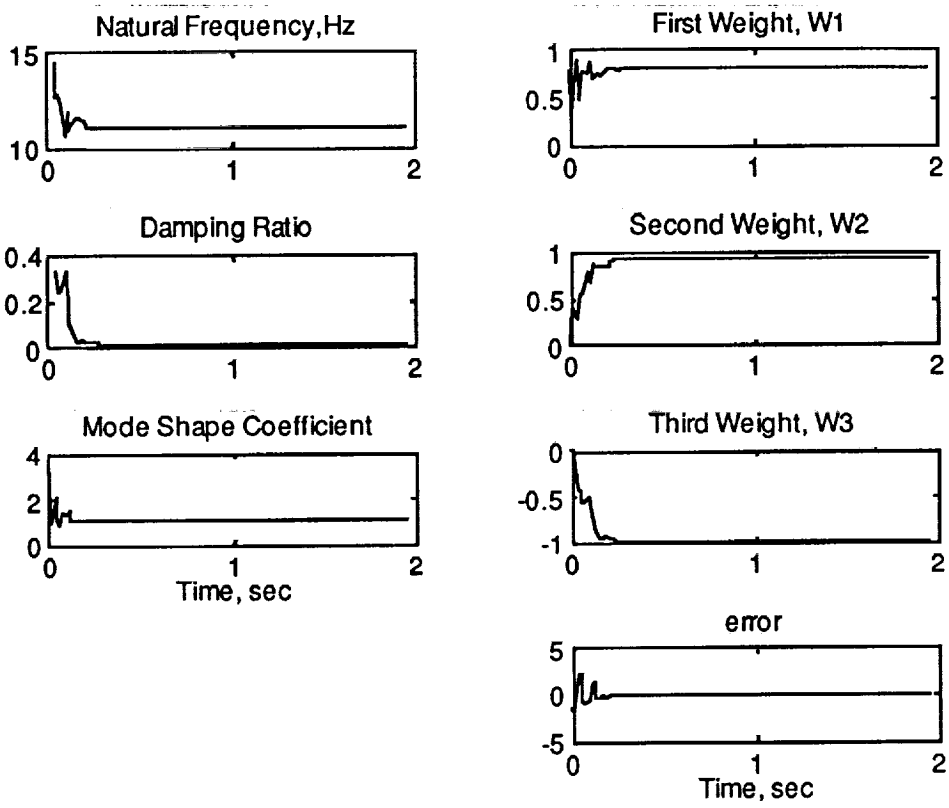

Fig. 5.34 Results of the new modal parameter identification approach

6 Summary and Recommendations

This study has shown that adaptive filters can be used effectively to identify the modal parameters of a vibrating structure. Summary on the findings and recommendations for further research are described herein.

6.1 Summary

This section details most significant findings obtained from the research conducted.

◊ The simultaneous multiple mode identification approach developed in this research has an advantage over the multiple mode identification process using the single mode identification algorithm with the aid of band-pass filtering. The simultaneous identification method is more effective in identifying closely-spaced modes which frequently appear in aerospace structures.

◊ The real-time identification capability of the algorithms will help model accurately the varying dynamics of aerospace structures and vehicles. This in turn can be used to develop adaptive controllers for vibration suppression, active noise control, and system health monitoring.

◊ The use of the bilinear transformation (BT) to relate the adaptive filter coefficients to the dynamic system parameters is fairly simple for the single and two mode cases, but for more than two modes the mathematics becomes very complex. In addition, the BT is an approximation that has a built-in frequency warping problem which becomes severe as the separation between the signal and sampling frequencies grows larger. An alternative approach of relating identified filter coefficients to the modal parameters was developed to rectify the problem.

◊ Simultaneous identification of modal parameters of more than two modes is theoretically feasible but it was not able to be implemented due to mathematical complexity.

◊ The mode-skipping phenomenon associated with the simultaneous two mode identification at high sampling rates could be the result of poor signal quality and the algorithm's inability to distinguish between actual modes of vibration and noise.

◊ The learning rate which can also be thought of as a step-size, has been found to have a significant effect on the speed of convergence and stability of the learning process. A wide range of learning rates exists in each test which produce good results. However, there is usually a trade-off between speed and stability.

6.2 Recommendations

Recommendations for further research are listed herein.

◊ The real-time identification of more than two modes should be investigated further. Especially, an alternative approach of identifying multiple modes simultaneously, by developing different adaptive filter structures, should be studied further. Although various approaches are looked at, no clear-cut winner has been identified.

◊ A state estimator construction process should be investigated using the modal parameters identified via the on-line identification algorithms.

◊ Active control simulation and experiments should be conducted to evaluate the performance of the state estimator.

References

[1] Lim, T. W., Cabell, R. H. and Silcox, R. J. *On-Line Identification of Modal Parameters Using Artificial Neural Networks*, Journal of Vibration and Acoustics, Vol. 118, pp. 649 656, October 1996.

[2] Lim, T. W., Bosse, A. and Fisher, S. *Adaptive Filters for Real-Time System Identification and Control*, Journal of Guidance, Control and Dynamics, Vol. 20, No. 1, Jan-Feb 1997.

[3] Lim, T. W., Alhassani, A. A., *On-Line Simultaneous Identification of Multiple Modes for Flexible Structures Using Adaptive Filtering Techniques*, AIAA GNC Conference in New Orleans, LA, Aug. 1997.

[4] Widrow, B. and Walach E. *Adaptive Inverse Control*, Prentice Hall, NJ, 1996.

[5] Treichler, J. R., Johnson Jr., C. R. and Larimore, M. G. *Theory and Design of Adaptive Filters*, John Wiley & Sons, New York, 1987.

[6] Ewing, M. , *Constructing Modal Properties from a measured FRF,* Class Handout (AE680), Department of Aerospace Eng., University of Kansas, Lawrence, KS, Spring 1996.

[7] Oppenheim, A. V. and Schafer, R. W. *Discrete-Time Signal Processing*, Prentice Hall, NJ, 1989.

[8] Inman, D. J. *Engineering Vibration*, Prentice Hall, NJ, 1994.

[9] Kreyszig, E. *Advanced Engineering Mathematics*, John Wiley & Sons, New York, 1988.

[10] Anonymous, *MDC40S TIM-40 Module User Manual*, Document No. 500- 00094, ver. 1.03, Spectrum Signal Processing Inc., B.C., August 1994.

[11] Anonymous, *QUAD C40 Processor Board Technical Reference Manual*, Document No. 500-00120, ver. 1.03, Spectrum Signal, Inc., B.C., Sept. 1994.

[12] Anonymous, *TIM-40 Module and Carrier Board PC Support Manual*, Document No. 500-00166, ver. 1.10, Spectrum Signal Inc., B.C., March 1995.

[13] Anonymous, *Crystal Analog Daughter Module User Manual*, Document No. 500-00055, ver. 1.04, Spectrum Signal Processing Inc., B.C., August 1994.

[14] Anonymous, *PC Daughter Module Carrier Board User Manual*, Document No. 500-00086, ver. 1.01, Spectrum Signal Processing Inc., B.C., Dec. 1994.

[15] Anonymous, *C40 Network API Support for PC Systems, a Quickstart Guide*, Document No. 500-00275, ver. 1.00, Spectrum Signal, B.C., March 1995.

[16] Anonymous, *C40 Network API Support for PC Systems, User Guide*, Document No. 500-00236, ver. 1.10, Spectrum Signal, B.C., March 1995.

[17] Anonymous, *TMS320C4x User's Guide*, Document No. SPRU063A, Texas Instruments, 1993.

[18] Anonymous, *TMS320 Floating-Point DSP Optimizing C Compiler User's Guide*, Document No. SPRU034F, Texas Instruments, 1995.

[19] Anonymous, *TMS320C4x C Source Debugger User's Guide*, Document No. SPRU054F, Texas Instruments, 1992.

[20] Anonymous, *TMS320C4x Parallel Runtime Support Library User's Guide*, Document No. SPRU084A, Texas Instruments, 1994.

[21] Anonymous, *TMS320 Floating-Point DSP Assembly Language Tools User's Guide*, Document No. SPRU035B, Texas Instruments, 1995.

[22] Anonymous, *Microsoft Visual C++ Tools-Command-Line Utilities User's Guide*, Vol. 2, Document No. DB64018-0395, Microsoft Corporation, 1995.

[23] Waner, R. C. *Ground Vibration Tests and Finite Element Model Updating for a Seven-Bay Space Truss*, MS Thesis, AE Dept., University of Kansas, 1996.

[24] Anonymous, *LING STAR 1.0 Power Amplifier Operating Manual*, Document No. 706789B, LING Electronics, 1994.

[25] Anonymous, *LING LMT-50 Modal Shaker Operating Manual*, Document No. 706767A, LING Electronics, 1993.

[26] Anonymous, *PCB Model 208 Series Force Transducer Operator's Manual*, PCB Piezotronics, NY, 6/1993.

[27] Anonymous, *PCB Series 353 Quartz Shear Mode ICP Accelerometer Operator's Manual*, PCB Piezotronics, NY, 6/1993.

[28] Anonymous, *PCB Model 483B18 Line Power Voltage Amplifier Operating Instructions*, PCB Piezotronics, NY, 6/1993.

[29] Prescott, G., Personal Communication, Dec. 1996.

[30] Ziemer, R. E., Tranter, W. H. and Fannin, D. R. *Signals and Systems Continuous and Discrete*, Macmillan Publishing Co., Inc., 1983.

[31] Lim, T. W., Personal Communication, Jan. 1997.

[32] Brown, D., Personal Communication, Spring-Fall 1996.

[33] Parks, T. W. and Burrus, C. S., *Digital Filter Design*, John Wiley & Sons, Inc., 1987.

[34] Ogata, K. *Discrete-Time Control Systems*, Prentice Hall, N.J., 1995.

[36] Bendat, J. S. and Piersol, A. G. *Random Data Analysis and Measurement Procedures*, John Wiley and Sons, 1986

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%     PROGRAM: SDOFSIM.M
%
%     DESCRIPTION:    A MATLAB® M-file (simulation program) that tests the
%     single mode identification algorithm.  The system used is a randomly excited
%     SDOF damped spring mass system.
%
%     Hadi Alhassani, KUAE
%     12/26/1996
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all

          %%%%% DYNAMIC SYSTEM SIMULATION %%%%%

np = 1024;     % number of iterations
fs = 256;      % sampling freq. in Hz
Ts = 1/fs;
w = 42;        % system natural frequency in Hz
z = 0.01;      % system damping ratio
Apq = 1;       % modal constant
t=[0:1:np-1]';
Exc=randn(size(t));    % random excitation

Num=[Apq,0,0];                  % continuous system transfer function
Den=[1 (2*z*w*2*pi) (w*2*pi)^2];
[Numd,Dend]=c2dm(Num,Den,Ts,'tustin');   % pulse transfer function
[Resp,xx]=dlsim(Numd,Dend,Exc);   % system response simulation

          %%%%% SYSTEM IDENTIFICATION %%%%%
                 % PART I: OFF-LINE ID    %

P1off=Exc-(2*[0;Exc(1:np-1)])+[0;0;Exc(1:np-2)];  % linear combiner inputs
P2off=[0;Resp(1:np-1)];
P3off=[0;0;Resp(1:np-2)];
Ppinv = pinv([P1off,P2off,P3off]);   % pseudoinverse of LC input vector
W_off=Ppinv*Resp                % off-line weights
b=W_off(1);                     % off-line adaptive filter coeffs.
a1=-W_off(2);
a2=-W_off(3);
w_off=2*fs*sqrt((1+a1+a2)/(1-a1+a2))        % off-line mod. param.
Z_off=2*fs*(1-a2)/(w_off*(1-a1+a2))
Apq_off=4*b/(1-a1+a2)
```

```
                    % PART II: ON-LINE ID     %

        % initailization
yk=0;  % present value of response
yk1=0; % previous value of response (1 tap delay)
yk2=0; % value before previous (2 tap delays)
uk=0;  % present value of excitation
uk1=0; % 1 tap delay
uk2=0; % 2 tap delays
W1=0; % initialize weights to zero
W2=0;
W3=0;
alpha = input(' Enter learning rate:  ');      % user defined learning rate

for k=1:np      % start system identification process

  yk2 = yk1;
  yk1 = yk;
  yk = Resp(k);          % system response
  uk2 = uk1;
  uk1 = uk;
  uk = Exc(k);           % system excitation
  P1 = uk-(2*uk1)+uk2;       % linear combiner (adaptive IIR filter) inputs
  P2 = yk1;
  P3 = yk2;
  val = P1*P1+P2*P2+P3*P3;          % Normalizer
  yout=P1*W1+P2*W2+P3*W3;     % LC output
  e(k) = yk - yout;                 % Error
  W1=W1+e(k)*alpha*(P1/val);        % Widrow-Hoff delta rule
  W2=W2+e(k)*alpha*(P2/val);
  W3=W3+e(k)*alpha*(P3/val);
  W123(:,k)=[W1;W2;W3];  % store weights for plotting
  b=W1;                  % evaluate adaptive filter coefficients using weights
  a1=-W2;
  a2=-W3;
  w_on(k)=2*fs*sqrt((1+a1+a2)/(1-a1+a2));          % evaluate modal parameters
  z_on(k)=2*fs*(1-a2)/(w_on(k)*(1-a1+a2));
  Apq_on(k)=4*b/(1-a1+a2);

end
```

%%%%% Plotting the Results %%%%%

```
Hz=[1/(np*Ts):1/(np*Ts):1/Ts]';          % set freq. vector for fft
fftResp=fft(Resp);                       % find fft of response
figure(1),subplot(411),plot(W123(1,:))   % plot weights and error
title('First Weight, W1')
subplot(412),plot(W123(2,:)),title('Second Weight, W2')
subplot(413),plot(W123(3,:)),title('Third Weight, W3')
subplot(414),plot(e), title('error')
xlabel('Number of Iterations')
figure(2), subplot(311), plot(Exc)       % plot excitation and response and fft
title('Random Excitation')
subplot(312), plot(Resp)
title('System Acceleration Response')
xlabel('Number of Iterations')
subplot(313),plot(Hz(1:np/2),abs(fftResp(1:np/2)))
title('FFT of System Response')
xlabel('Frequency, Hz')
figure(3),subplot(311),plot(w_on/pi/2)   % plot modal parameters
title('Natural Frequency, Hz')
subplot(312),plot(z_on),title('Damping Ratio')
subplot(313),plot(Apq_on)
title('Modal Constant')
xlabel('Number of Iterations'))
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       PROGRAM: DDOFSIM.M
%
%       DESCRIPTION:     MATLAB M-file used to test the two degree of freedom
%       system identification algorithm.  The system used is a randomly excited DDOF
%       proportionally damped spring mass system in series.
%
%       Hadi Alhassani, KUAE
%       12/26/1996
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all

                %%%%% DYNAMIC SYSTEM SIMULATION %%%%%

np = 1024;                  % number of iterations
t = [0:1:np-1]';            % time vector
m1 = 3;                     % first mass
m2 = 3;                     % second mass
k1 = 50000;                 % first spring constant
k2 = 50000;                 % second spring constant
fs=128;                     % sampling frequency
Ts=1/fs;
M=[m1 0;0 m2];              % mass matirx
K=[k1+k2  -k2;-k2  k2];     % stiffness matrix
C=0.0001*K;                 % damping matrix
Mmh = inv(sqrt(M));         % inverse the squre root of M
Ktilda = Mmh*K*Mmh;
Ctilda = Mmh*C*Mmh;
[P,wsqold] = eig(Ktilda);   % find eigenvalues and vecors
P = [P(:,2),P(:,1)];        % rearrange eigenvector
wsqnew = P'*Ktilda*P;       % eigenvalues check
tzw = P'*Ctilda*P;          % 2*zeta*omega
Ftemp=eye(2);               % identity matrix
modeshape1 = Mmh*(P(:,1)) % find mode shapes
modeshape2 = Mmh*(P(:,2))


Ac=[zeros(2,2) eye(2);-wsqnew  -tzw];        % system in continuous state-space form
Bc=[zeros(2,2);P'*Mmh*Ftemp];
Cc=[Mmh*P*(-wsqnew)  Mmh*P*(-tzw)];
Dc=[Mmh*P*P'*Mmh*Ftemp];
[w,z] = damp(Ac)                             % system natural freq. and damping ratio
w=[w(1),w(3)];
```

```
fprintf('\n Natural Frequencies:\n')
fprintf('      %f Hz\n',(w/pi/2))

noexc = zeros(size(t));                    % no excitation
Exc = randn(size(t));                      % random excitation
EP = input('  Enter Excitation Point. (1 or 2)  ');      % user defined point of excitation
RP = input('  Enter Response Point. (1 or 2)  ');       % user defined point of response
if EP == 1                                 % detemine where excitaion is applied
  EXC = [Exc noexc];                       % Excitation is at 1
elseif EP == 2
  EXC = [noexc Exc];                       % Excitation is at 2
end

[Ad,Bd,Cd,Dd]=c2dm(Ac,Bc,Cc,Dc,Ts,'tustin');      % system pulse T.F.
[Acc,XX]=dlsim(Ad,Bd,Cd,Dd,EXC);                  % system response simulation
Resp = Acc(:,RP);                                 % response measured at point RP
Hz=[1/(np*Ts):1/(np*Ts):1/Ts]';                   % set freq. vector for fft
fftResp=fft(Resp);                                % find fft of response
figure,subplot(311), plot(Exc)
title('Random Excitation')
subplot(312), plot(Resp)
title('System Acceleration Response')
xlabel(' Number of Iterations')
subplot(313), plot(Hz(1:np/2),abs(fftResp(1:np/2)))
title('FFT of System Response')
xlabel('Frequency, Hz')

cont = input(' Continue . . . . . ? (y/n)   ','s');
if cont == 'y'
```

%%%%% SYSTEM IDENTIFICATION %%%%%

% PART I: OFF-LINE ID %

```
Vk2 = Exc(1:np)+(-2*[0;Exc(1:np-1)])+[0;0;Exc(1:np-2)];
P1off = [0;Resp(1:np-1)];                  % linear combiner (adaptive IIR filter) inputs
P2off = [0;0;Resp(1:np-2)];
P3off = [0;0;0;Resp(1:np-3)];
P4off = [0;0;0;0;Resp(1:np-4)];
P5off = Vk2;
P6off = [0;Vk2(1:np-1)];
P7off = [0;0;Vk2(1:np-2)];
Ppinv = pinv([P1off,P2off,P3off,P4off,P5off,P6off,P7off]);
Woff = Ppinv*Resp                          % off-line weights
```

```
% Evaluate the modal parameters using Newton root approximation algorithm
fxo = 0;        % initialize function value
fxop = 0;       % initialize function derivative
Xn = 0;         % initialize next value of x
Xo = 1;         % starting value

for i=1:20
 fxo =  (Xo^6)+...              % weights function
        (Woff(2)*Xo^5)+...
        ((Woff(1)*Woff(3)+Woff(4))*Xo^4)+...
        ((2*Woff(2)*Woff(4)-(Woff(3)^2)+Woff(4)*Woff(1)^2)*Xo^3)+...
        ((-Woff(1)*Woff(3)*Woff(4)-Woff(4)^2)*Xo^2)+...
        (Woff(2)*Woff(4)^2)*Xo+(-Woff(4)^3);
 fxop = (6*Xo^5)+...            % function derivative
        (Woff(2)*5*Xo^4)+...
        ((Woff(1)*Woff(3)+Woff(4))*4*Xo^3)+...
        ((2*Woff(2)*Woff(4)-(Woff(3)^2)+Woff(4)*Woff(1)^2)*3*Xo^2)+...
        (-Woff(1)*Woff(3)*Woff(4)-Woff(4)^2)*2*Xo+(Woff(2)*Woff(4)^2);
 Xn = Xo - fxo/fxop;  % Newton algorithm
 Xo = Xn ;
end

a12 = Xn;               % Evaluate filter coefficients using linear combiner weights
a22 = -Woff(4)/a12;
a21 = (a22*Woff(1)-Woff(3))/(a12-a22);
a11 = -Woff(1)-a21;
b2 = (a22*Woff(5)-Woff(7))/(a22-a12);
b1 = Woff(5)-b2;
w1_off=2*fs*sqrt((1+a11+a12)/(1-a11+a12));      % off-line modal parameters
w2_off=2*fs*sqrt((1+a21+a22)/(1-a21+a22));
z1_off=2*fs*(1-a12)/(w1_off*(1-a11+a12));
z2_off=2*fs*(1-a22)/(w2_off*(1-a21+a22));
A1pq_off=4*b1/(1-a11+a12);
A2pq_off=4*b2/(1-a21+a22);

                % PART II: ON-LINE ID %

W1 = 0;         % initialize weights
W2 = 0;
W3 = 0;
W4 = 0;
W5 = 0;
W6 = 0;
```

```
W7 = 0;
                % initialize excitation and response values
yk=0;           % present response value
yk1=0;          % 1 tap delay
yk2=0;          % 2 tap delays
yk3=0;          % 3 tap delays
yk4=0;          % 4 tap delays
uk=0;           % present value of excitation
uk1=0;
uk2=0;
vk2=0;          % where V2=F0-(2*F1)+F2
vk3=0;          % 1 tap delay
vk4=0;          % 2 tap delays
alpha=0.95;     % learning rate

for k = 5:np    % start system identification process

  yk4 = yk3;
  yk3 = yk2;
  yk2 = yk1;
  yk1 = yk;
  yk = Resp(k);         % system response
  uk2 = uk1;
  uk1 = uk;
  uk = Exc(k);          % system excitation
  vk4 = vk3;
  vk3 = vk2;
  vk2 = uk-(2*uk1)+uk2;
  P1 = yk1;             % linear combiner inputs
  P2 = yk2;
  P3 = yk3;
  P4 = yk4;
  P5 = vk2;
  P6 = vk3;
  P7 = vk4;
  val=P1*P1+P2*P2+P3*P3+P4*P4+P5*P5+P6*P6+P7*P7;        % Normalizer
  yout = P1*W1+P2*W2+P3*W3+P4*W4+P5*W5+P6*W6+P7*W7; % LC output
  e(k) = yk - yout;                     % Error
  W1 = W1+e(k)*alpha*(P1/val);      % Widrow-Hoff Delta Rule
  W2 = W2+e(k)*alpha*(P2/val);
  W3 = W3+e(k)*alpha*(P3/val);
  W4 = W4+e(k)*alpha*(P4/val);
  W5 = W5+e(k)*alpha*(P5/val);
```

```
  W6 = W6+e(k)*alpha*(P6/val);
  W7 = W7+e(k)*alpha*(P7/val);
  WGHT_ON(:,k)=[W1;W2;W3;W4;W5;W6;W7]; % store weights

%       Newton algorithm to solve for adaptive filter coefficients
  fxo = 0;       % initialize function value at root
  fxop = 0;      % initialize function derivative
  Xn = 0;        % initialize next value of x
  Xo = 1;        % starting value

  for i=1:20
   fxo = (Xo^6)+...              % weights function
         (W2*Xo^5)+...
         ((W1*W3+W4)*Xo^4)+...
         ((2*W2*W4-(W3^2)+W4*W1^2)*Xo^3)+...
         ((-W1*W3*W4-W4^2)*Xo^2)+...
         (W2*W4^2)*Xo+(-W4^3);
   fxop =(6*Xo^5)+...                  % function derivative
         (W2*5*Xo^4)+...
         ((W1*W3+W4)*4*Xo^3)+...
         ((2*W2*W4-(W3^2)+W4*W1^2)*3*Xo^2)+...
         (-W1*W3*W4-W4^2)*2*Xo+(W2*W4^2);
   Xn = Xo - fxo/fxop;          % Newton algorithm
   Xo = Xn ;
  end

  a12 = Xn;              % on-line adaptive filter coefficients
  a22 = -W4/a12;
  a21 = (a22*W1-W3)/(a12-a22);
  a11 = -W1-a21;
  b2 = (a22*W5-W7)/(a22-a12);
  b1 = W5-b2;
  w1_on(k)=2*fs*sqrt((1+a11+a12)/(1-a11+a12));  % on-line modal parameters
  w2_on(k)=2*fs*sqrt((1+a21+a22)/(1-a21+a22));
  z1_on(k)=2*fs*(1-a12)/(w1_on(k)*(1-a11+a12));
  z2_on(k)=2*fs*(1-a22)/(w2_on(k)*(1-a21+a22));
  A1pq_on(k)=4*b1/(1-a11+a12);
  A2pq_on(k)=4*b2/(1-a21+a22);

 end
```

%%%%% Plotting the Results %%%%%

```
figure,subplot(411),plot(WGHT_ON(1,:))
title('First Weight, W1')
subplot(412),plot(WGHT_ON(2,:)),title('Second Weight, W2')
subplot(413),plot(WGHT_ON(3,:)),title('Third Weight, W3')
subplot(414),plot(WGHT_ON(4,:)),title('Fourth Weight, W4')
xlabel('Number of Iterations')
figure,subplot(411),plot(WGHT_ON(5,:))
title('Fifth Weight, W5')
subplot(412),plot(WGHT_ON(6,:)),title('Sixth Weight, W6')
subplot(413),plot(WGHT_ON(7,:)),title('Seventh Weight, W7')
subplot(414),plot(e), title('Error')
xlabel('Number of iterations')
figure,subplot(311),plot(w1_on)
title('First Mode Natural Frequency (rad/sec)')
subplot(312),plot(z1_on),title('First Mode Damping Ratio')
subplot(313),plot(A1pq_on),title('Modal Constant, A1pq')
xlabel('Number of Iterations')
figure,subplot(311),plot(w2_on)
title('Second Mode Natural Frequency (rad/sec)')
subplot(312),plot(z2_on),title('Second Mode Damping Ratio')
subplot(313),plot(A2pq_on),title('Modal Constant, A2pq')
xlabel('Number of Iterations')

end
```

```
/*********************************************************************
```
PROGRAM: C40SDOF.C (C40 Source Code)

DESCRIPTION: C40 code that is downloaded onto the C40 system using the PCSDOF.C host program via the application NETAPI library. The code in the ISR reads two signals, an excitation and a response of a low frequency flexible structure and trains an adaptive IIR filter to identify the fundamental mode of vibration in real-time.

>>> SAMPLING FREQUENCY = 4 kHz <<<

Hadi Alhassani, KUAE
12/26/1996

```
*********************************************************************/
/*********************************************************************
                Header Files
*********************************************************************/
#include "c:\dsptools\math.h"          /* math library */
#include "c:\dsptools\intpt40.h"       /* Interrupt support (PRSL) */
#include "c:\dsptools\compt40.h"       /* Comm port support (PRSL) */
#include "carrier.h"              /* Defines pointers to DSPLINK registers for DM */
/*********************************************************************
                Define Constants
*********************************************************************/
#define  BUF_SIZE    3000          /* Define array size */
#define  LIA_CHAN_NO  1            /* Comm port channel number */
/*********************************************************************
                Global Variables
*********************************************************************/
long int counter = 0, index = 0;
long y, u;
float yk=0,yk1=0,yk2=0,uk=0,uk1=0,uk2=0;
float TempW[BUF_SIZE], TempZ[BUF_SIZE], TempM[BUF_SIZE];
float WW1[BUF_SIZE], WW2[BUF_SIZE], WW3[BUF_SIZE];
float P1=0, P2=0, P3=0, alpha=0.01 , fs=333.33;
float TY, TU, FY=0, FU=0, ya1=0, ya2=0, ya3=0, ya4=0;
float yf1=0, yf2=0, yf3=0, yf4=0;
float xa=0, xa1=0, xa2=0, xa3=0, xa4=0;
float xf=0, xf1=0, xf2=0, xf3=0, xf4=0;
float W1=0, W2=0, W3=0, yout=0, val=0, temp=0;
float e=0, omega=0, zeta=0, ms=0, b=0, a1=0, a2=0;
/*********************************************************************/
void c_int04(void);     /* Function prototype for the IIOF1 ISR */
```

```
/*************************************************************************
                        Main Program
*************************************************************************/
void main(void)
{
volatile int dummy;

for(index=0; index<=BUF_SIZE-1; index++){
        TempW[index]=0;
        TempZ[index]=0;
        TempM[index]=0;
        WW1[index]=0;
        WW2[index]=0;
        WW3[index]=0;
        }
index=0;

asm(" PUSH AR0");           /* Set Up C40 Interruptts */
asm(" PUSH DP");            /* assembly language instructions */
asm(" LDI 030H,AR0");       /* needed to perform an interrupt */
asm(" LSH 16,AR0");         /* acknowledge instruction to allow */
asm(" IACK *AR0");          /* external interrupts to the C40. */
asm(" POP DP");
asm(" POP AR0");

INT_DISABLE();              /* Global disable of interrupts */
set_ivtp(DEFAULT);          /* Set IVTP on 512 word boundary,*/
                            /* see vector in linker file */
install_int_vector((void *)c_int04, 0x04);     /* Set the IIOF1 int vector */
load_iif(0x00B0);      /* Enable the IIF01 pin to be level trigger interrupt  */

/* Set up DSPLINK registers for the DMCB and DM. */
dummy = *DM1_RESET;          /* Do a read to Reset the Site A DM */
*DM1_ROUTE = 0x0000;   /* Set LKCLK1 to choose MCLK1 as system clock */
*DM1_INT_MASK = 0x00010000;    /* Interrupt when Input Data Regs are full */
*DM1_AMELIA_CTRL = 0xB30000;   /* CM6=0, board now in reset */
*DM1_AMELIA_CTRL = 0xF30000;   /* CM6=1, calibration cycle started */
                        /* CM0=1 CM1=1 and CM4=1; use MCLK1 as system clock*/
*DM1_USER_CONTROL = 0xA8E00000; /* Selcet prescale factor, clock source */
                        /* PT11=PT10=1,CS11=CS10=0,MCD1=1 and MCD2=1 */
                  /* SAMPLING RATE IS NOW SET TO 4kHz */
*DM1_CONFIGURATION = 0xB3900000;/* Write the KEY for the Crystal ADM  */
```

```
INT_ENABLE(); /* Globally enable interrupts */

/* create a while loop that activates an INT_DISABLE function to halt the C40
operations as soon as the arrays are full, then send the data to the Host PC */

while(index <= BUF_SIZE){
  if(index == BUF_SIZE){
        INT_DISABLE();
        send_msg(LIA_CHAN_NO,TempW,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
        send_msg(LIA_CHAN_NO,TempZ,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
        send_msg(LIA_CHAN_NO,TempM,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
        send_msg(LIA_CHAN_NO,WW1,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
        send_msg(LIA_CHAN_NO,WW2,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
        send_msg(LIA_CHAN_NO,WW3,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
}  /* end of if statement */
}  /* end of while loop */
} /* end of main program  */

/*************************************************************************
            Interrupt Service Routine
This subroutine contains the data acquistion and system identification code.  It is
executed at the sampling rate.
*************************************************************************/
void c_int04(void)
{
volatile long clear;
clear = *DM1_INT_STATUS;   /*Read Interrupt Status Regis to Clear Interrupts */
y = *DM1_CH0_IN_DATA;      /* Read Channel 0 Input Data Register */
u = *DM1_CH1_IN_DATA;      /* Read Channel 1 Input Data Register */

y = (y >> 16);             /* Shift data by 16 bits to map DSPLINK */
u = (u >> 16);
TY = -1*((float)y)/16384;          /* True value in voults */
TU = -1*((float)u)/16384;

/* Implement a 4th order low-pass Butterworth filter on both inputs.  ( wc = 20 Hz) */
/ * Response Signal (Channel 0) */
```

```
xa4=xa3;
xa3=xa2;
xa2=xa1;
xa1=xa;
xa=TY;
ya4=ya3;
ya3=ya2;
ya2=ya1;
ya1=FY;
FY=3.917907865*ya1-5.757076379*ya2+3.760349508*ya3-
0.921181929*ya4+1.0e-08*(5.845142437*xa+23.380569747*xa1+
35.070854487*xa2+23.380569747*xa3+5.845142437*xa4);

/* Excitationi Signal (Channel 1) */
xf4=xf3;
xf3=xf2;
xf2=xf1;
xf1=xf;
xf=TU;
yf4=yf3;
yf3=yf2;
yf2=yf1;
yf1=FU;
FU=3.917907865*yf1-5.757076379*yf2+3.760349508*yf3-
0.921181929*yf4+1.0e-08*(5.845142437*xf+23.380569747*xf1+
35.070854487*xf2+23.380569747*xf3+5.845142437*xf4);

counter += 1;
if(counter == 12){              /* time decimation to set effective fs */

      /****** REAL-TIME SYSTEM IDENTIFICATIOIN CODE ******/

       yk2=yk1;
       yk1=yk
       yk=FY;        /* filtered system response */
       uk2=uk1;
       uk1=uk;
       uk=FU;        /*filtered system excitation */
       P1 = uk-(2*uk1)+uk2;        /* linear combiner (adaptive IIR filter) inputs */
       P2 = yk1;
       P3 = yk2;
       val = P1*P1+P2*P2+P3*P3;  /* Normalizer */
       yout = P1*W1+P2*W2+P3*W3;      /* linear combiner output */
```

```
        e = yk-yout;                            /* Error */
        W1 = W1+e*alpha*(P1/val);               /* Widrow-Hoff Delta Rule */
        W2 = W2+e*alpha*(P2/val);
        W3 = W3+e*alpha*(P3/val);
        b = W1;         /* evaluate the filter coefficients using the weights of the LC */
        a1 = -W2;
        a2 = -W3;
        temp = (1+a1+a2)/(1-a1+a2);
        omega = 2*fs*sqrt(temp);        /* evaluate the modal parameters */
        zeta = 2*fs*(1-a2)/(omega*(1-a1+a2));
        ms = 4*b/(1-a1+a2);
        TempW[index]=omega;             /* store modal parameter results in arrays */
        TempZ[index]=zeta;
        TempM[index]=ms;
        WW1[index]=W1;                  /* store weights in arrays */
        WW2[index]=W2;
        WW3[index]=W3;
        index += 1;
        counter = 0;
} /* end if statement that sets effective fs */
} /* end Interrupt Service Routine */
```

```
/*********************************************************************
PROGRAM: C40SDOF.CMD

DESCRIPTION:    A linker command file that contains the linker options, standard
memory configuration and sections allocation to be used with the C40SDOF.C code
shown in this appendix.  This file is called by the batch file that runs the compiler.

Hadi Alhassani, KUAE
12/26/1996
*********************************************************************/
/* 1)  Linker Options */

-x                      /* Reread libraries if unresolved symbols have not been found.*/
-c                      /* ROM  autoinitialization.          */
-o C40SDOF.OUT          /* Linker option to name the output file.  */
-m C40SDOF.MAP          /* Linker option to generate a map file.   */
C40SDOF.OBJ             /* Input file specification. */
-i c:\dsptools          /* Run Time Support Library directory      */
-l rts40.lib            /* Link small memory C40 PRTS Library.*/
-l prts40.lib           /* Link C40 PRTS.*/
-e c_int00              /* Define the entry point.*/

/* 2)  Standard Memory Configuration */
MEMORY
{
        IRAM0:  origin = 002FF800h     length = 0400h    /* Internal 0, 1k   */
        IRAM1:  origin = 002FFC00h     length = 0400h    /* Internal 1, 1k   */
        ERAM0:  origin = 00300000h     length = 10000h   /* External 0, 64k  */
        /* ERAM1:  origin = 00308000h  length = 8000h    /* External 1, 32k */*/
        PEROM:  origin = 40000000h     length = 8000h    /* EPROM, 32k  */
        ERAM2:  origin = 80000000h     length = 8000h    /* External 2, 32k  */
}

/* 3) Allocate Sections into memory */
SECTIONS
{
        .text                   : { }  > ERAM2
        .data                   : { }  > ERAM0
        .vector align=512       : { }  > IRAM1
        .cinit                  : { }  > IRAM1
        .bss                    : { }  > ERAM0
        .stack                  : { }  > IRAM0
}
```

```
/*********************************************************************
```
PROGRAM: PCSDOF.C (Host PC code)

DESCRIPTION: This is the PC Host code that downloads the C40SDOF.OUT executable file onto the C40 system using the application NETAPI library. Dynamic memory allocation is used to store six large arrays that are sent back by the C40 system when the system identification process is complete. These 6 arrays contain the modal parameters of the structure and the weights. They are written to data files that can be read be MATLAB and then displayed.

```
Hadi Alhassani
12/26/1996
*********************************************************************/
/*********************************************************************
            Header Files and Definitions
*********************************************************************/
#include <stdio.h>          /* Standard I/O library      */
#include <conio.h>          /* Standard Console and Port IO library */
#include "c4xapp.h"         /* NETAPI Applications library */
#include "chkerror.c"       /* Subroutine for NETAPI error handler */

#define  C40_FILE   "C40SDOF.OUT"    /* Executable file to be downloaded */
#define  BUF_SIZE   3000             /* Size of data arrays */
void checkReturnCode(UINT returnCode); /* Error code function prototype */
/*********************************************************************
            Main Program
*********************************************************************/
void main (void)
{
 int i;
 ULONG NumToRec1;
 UINT buf_length = BUF_SIZE;
 UINT ret;
 PROC_ID *handle;
 FILE *cfPtr1, *cfPtr2;
 float *w;
 float *z;
 float *a;
 float *W1;
 float *W2;
 float *W3;

 w = calloc(BUF_SIZE,sizeof(float));      /* Dynamic memory allocation */
```

```
 z  = calloc(BUF_SIZE,sizeof(float));
 a  = calloc(BUF_SIZE,sizeof(float));
 W1  = calloc(BUF_SIZE,sizeof(float));
 W2  = calloc(BUF_SIZE,sizeof(float));
 W3  = calloc(BUF_SIZE,sizeof(float));

if(w==NULL||z==NULL||a==NULL||W1==NULL||W2==NULL||W3==NULL)
        printf("Sorry, dynamic memory could not be allocated. ");
else{

 system("cls");          /* clear screen */
 /***** Reboot the C40 system *****/
 printf("\nRebooting the C40 network ....\n");
 ret = Global_Network_Reboot();
 checkReturnCode(ret);

 /***** Open Processor on Site A*****/
 printf("Opening processors .....\n");
 ret=Open_Processor_ID(&handle,"CPU_A",NULL);
 checkReturnCode(ret);

 /***** Download executable C40 code onto processor  *****/
 printf("\nLoading program %s to C40 in Site A.....\n", C40_FILE);
 ret=Load_And_Run_File_LIA(handle,C40_FILE);
 checkReturnCode(ret);

 /* Wait for data to arrive and receive when ready */

 printf("Training  . . . . . . . . ");

 ret=Read_LIA_Words_32(handle,1,&NumToRec1);
 ret=Read_LIA_Floats_32(handle,buf_length,w);

 ret=Read_LIA_Words_32(handle,1,&NumToRec1);
 ret=Read_LIA_Floats_32(handle,buf_length,z);

 ret=Read_LIA_Words_32(handle,1,&NumToRec1);
 ret=Read_LIA_Floats_32(handle,buf_length,a);

 ret=Read_LIA_Words_32(handle,1,&NumToRec1);
 ret=Read_LIA_Floats_32(handle,buf_length,W1);

 ret=Read_LIA_Words_32(handle,1,&NumToRec1);
```

```
ret=Read_LIA_Floats_32(handle,buf_length,W2);

ret=Read_LIA_Words_32(handle,1,&NumToRec1);
ret=Read_LIA_Floats_32(handle,buf_length,W3);

printf("\n Data has been successfully received.\n");

/* Write results to data files */
 if ((cfPtr1 = fopen("SDModPar.dat","w")) == NULL)
        printf("File could not be opened.\n");
else{
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr1,"%.5f\n",w[i]);
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr1,"%.5f\n",z[i]);
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr1,"%.5f\n",a[i]);
        fclose (cfPtr1);
   }
if ((cfPtr2 = fopen("SDWeight.dat","w")) == NULL)
        printf("File could not be opened.\n");
else{
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr2,"%.5f\n",W1[i]);
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr2,"%.5f\n",W2[i]);
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr2,"%.5f\n",W3[i]);
        fclose (cfPtr2);
   }
 /* Free memory */
 ret=Close_Processor_ID(handle);
 checkReturnCode(ret);
 Clear_All_Lib_Memory();
 checkReturnCode(ret);
 free(w);
 free(z);
 free(a);
 free(W1);
 free(W2);
 free(W3);
}
} /* end of main program */
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PROGRAM:      C40SDOF.M
%
% DESCRIPTION:  This MATALB M-file reads the six data arrays created by the
% PCSDOF.C program which contain the training wieghts and modal parameters of
% vibration of the structure.
%
% Hadi Alhassani, KUAE
% 12/26/1996
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all
%     Read Data
cd c:\..\workfile
%
fid1=fopen('SDModPar.dat');          % open modal parameters data file
[MP,NP1]=fscanf(fid1,'%f');          % read modal parameters
fclose(fid1);
%
fid2=fopen('SDWeight.dat');          % open training weights data file
[WW,NP2]=fscanf(fid2,'%f');          % read training weights
fclose(fid2);
cd c:\matlab
%     Rearrange Data
np1 = NP1/3;
np2 = NP2/3;
freq = MP(1:np1);
damp = MP(np1+1:2*np1);
mode = MP(2*np1+1:3*np1);
W1 = WW(1:np2);
W2 = WW(np2+1:2*np2);
W3 = WW(2*np2+1:3*np2);
%     Plot Results
figure,subplot(311), plot(freq), title('First Mode Frequency')
subplot(312), plot(damp), title('Damping Ratio')
subplot(313), plot(mode), title('Modal Constant')
figure,subplot(311),plot(W1), title('First Weight')
subplot(312), plot(W2), title('Second Weight')
subplot(313), plot(W3), title('Third Weight')
```

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

FILE: CARRIER.H

DESCRIPTION: A header file that defines pointers to the DSPLINK interface registers for the Crystal Daughter Module Carrier Board which is used as a slave board to the QPC/C40B board. Note that the PC/DMCB slave board is mapped into Space 4 in the global memory map of the Tim-40 module in site A on the QPC/C40 B board. Although Space 4 allows operation of the slave board at the slowest speed, it is always guaranteed to work with any LSI DSPLINK slave board [Reference 9].

Courtesy of Spectrum Signal Processing Inc.

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

#define DM1_CH0_IN_DATA          ((unsigned long*) 0xB0000300)
#define DM1_CH0_OUT_DATA         ((unsigned long*) 0xB0000300)
#define DM1_TIMER1               ((unsigned long*) 0xB0000301)
#define DM1_RESET                ((unsigned long*) 0xB0000301)
#define DM1_CH2_IN_DATA          ((unsigned long*) 0xB0000302)
#define DM1_CH2_OUT_DATA         ((unsigned long*) 0xB0000302)
#define DM1_INT_MASK             ((unsigned long*) 0xB0000303)
#define DM1_INT_STATUS           ((unsigned long*) 0xB0000303)
#define DM1_CH1_IN_DATA          ((unsigned long*) 0xB0000304)
#define DM1_CH1_OUT_DATA         ((unsigned long*) 0xB0000304)
#define DM1_AMELIA_CTRL          ((unsigned long*) 0xB0000305)
#define DM1_AMELIA_STATUS        ((unsigned long*) 0xB0000305)
#define DM1_CH3_IN_DATA          ((unsigned long*) 0xB0000306)
#define DM1_CH3_OUT_DATA         ((unsigned long*) 0xB0000306)
#define DM1_ROUTE                ((unsigned long*) 0xB0000307)
#define DM1_USER_CONTROL         ((unsigned long*) 0xB0000307)
#define DM1_CONFIGURATION        ((unsigned long*) 0xB0000307)
```

```
**********************************************************************
```

FILE: NETAPI.CFG

DESCRIPTION: System configuration file for the C40 Network that contains a listing of the devices on the carrier board, the modules and processors present, the processor unique name as well as base address of the PC I/O blocks, default board register values, memory map details and a link map. The information contained in this file is used to initialize various addresses and registers in the system.

Courtesy of Spectrum Signal Processing Inc.

```
**********************************************************************

Host {
 Hostname: LSI_HOST

 Board {
  Board_Type: QPC/C40B
  Host_Connection: 0300h        ; Block 0 Base Address
  Register: 010h, 00000h       ; Control Register
  Host_Connection: 0320h        ; Block 1 Base Address (JTAG)
  Host_Connection: 0400h        ; LIA Base Address
  Module {
   Module_Type: MDC40S1
   Site: A
   Processor CPU_A {

    Processor_Type: C40
    Clock_Speed: 50

    Memory_Map {
     Page 0002FF800h 0002FFBFFh No_RT_Access ; INT0
     Page 0002FFC00h 0002FFFFFh No_RT_Access ; INT1
     Page 000300000h 000307FFFh No_RT_Access ; BANK0
     Page 000308000h 00030FFFFh No_RT_Access ; BANK1
     Page 040000000h 040007FFFh No_RT_Access ; PEROM
     Page 070000000h 070007FFFh No_RT_Access ; IDROM
     Page 080000000h 080007FFFh No_RT_Access ; BANK2
    }
   }
  }
 }
}
; Link map to be placed here if required.
```

```
/*************************************************************************
```
PROGRAM: C40DDOF.C (C40 Source Code)

DESCRIPTION: This program is downloaded onto the C40 system using the PCDDOF.C host program via the application library. The code in the ISR reads two signals, an excitation and a response of a low frequency flexible structure and trains an adaptive IIR filter to identify the first two modes of vibration in real-time.

>>> SAMPLING FREQUENCY = 4 kHz <<<

Hadi Alhassani, KUAE
12/26/1996

```
*************************************************************************/
/*************************************************************************
            Header Files
*************************************************************************/
#include "c:\dsptools\math.h"    /* math library */
#include "c:\dsptools\intpt40.h"  /* interrupt support (PRSL) */
#include "c:\dsptools\compt40.h"  /* communication port support (PRSL) */
#include "carrier.h"              /* defines pointers to DSPLINK registers for DM */
/*************************************************************************
        Define Constants
*************************************************************************/
#define  BUF_SIZE    3000         /* define array size */
#define  LIA_CHAN_NO  1           /* comm port channel number */
/*************************************************************************
        Global Variables
*************************************************************************/
long int counter = 0, count2 = 0, index, i=0;
long y, u;
float yk=0,yk1=0,yk2=0,yk3=0,yk4=0,uk=0,uk1=0,uk2=0;
float vk2=0, vk3=0, vk4=0;
float freq1[BUF_SIZE], mode1[BUF_SIZE], damp1[BUF_SIZE];
float freq2[BUF_SIZE], mode2[BUF_SIZE], damp2[BUF_SIZE];
float TY, TU, FY=0, FU=0, P1, P2, P3, P4, P5, P6, P7;
float W1=0, W2=0, W3=0, W4=0, W5=0, W6=0, W7=0;
float ya1=0, ya2=0, ya3=0, ya4=0;
float yf1=0, yf2=0, yf3=0, yf4=0;
float xa=0, xa1=0, xa2=0, xa3=0, xa4=0;
float xf=0, xf1=0, xf2=0, xf3=0, xf4=0;
float e=0, val=0, yout=0, a11, a12, a21, a22, b1, b2;
float fxo=0, fxop=0, Xn=0, Xo=0.5, alpha=0.1, fs=200;
/*************************************************************************/
```

```
void c_int04(void);              /* Function prototype for the IIOF1 ISR */
/*************************************************************************/
                Main Program
/*************************************************************************/
void main(void)
{
volatile int dummy;

/** Initialize arrays that hold modal parameters **/
for(index=0; index<=BUF_SIZE-1; index++){
        freq1[index]=0;
        freq2[index]=0;
        damp1[index]=0;
        damp2[index]=0;
        mode1[index]=0;
        mode2[index]=0;
        }
index=0;

asm(" PUSH AR0");               /* Set Up C40 Interruptts */
asm(" PUSH DP");                /* assembly language instructions */
asm(" LDI 030H,AR0");           /* needed to perform an interrupt */
asm(" LSH 16,AR0");             /* acknowledge instruction to allow */
asm(" IACK *AR0");              /* external interrupts to the C40. */
asm(" POP DP");
asm(" POP AR0");

INT_DISABLE();                  /* Global disable of interrupts */
set_ivtp(DEFAULT);              /* Set IVTP on 512 word boundary,*/
                                /* see vector in linker file */
install_int_vector((void *)c_int04, 0x04);     /* Set the IIOF1 int vector */
load_iif(0x00B0);       /* Enable the IIF01 pin to be level trigger interrupt  */

/* Set up DSPLINK registers for the DMCB and DM. */
dummy = *DM1_RESET;             /* Do a read to Reset the Site A DM */
*DM1_ROUTE = 0x0000;    /* Set LKCLK1 to choose MCLK1 as system clock */
*DM1_INT_MASK = 0x00010000;     /* Interrupt when Input Data Regs are full */
*DM1_AMELIA_CTRL = 0xB30000;    /* CM6=0, board now in reset */
*DM1_AMELIA_CTRL = 0xF30000;    /* CM6=1, calibration cycle started */
                        /* CM0=1 CM1=1 and CM4=1; use MCLK1 as system clock*/
*DM1_USER_CONTROL = 0xA8E00000; /* Selcet prescale factor, clock source */
                        /* PT11=PT10=1,CS11=CS10=0,MCD1=1 and MCD2=1 */
                /* SAMPLING RATE IS NOW SET TO 4kHz */
```

```
*DM1_CONFIGURATION = 0xB3900000;/* Write the KEY for the Crystal ADM  */

INT_ENABLE();  /* Globally enable interrupts */

/* create a while loop that activates an INT_DISABLE function to halt the C40
operations as soon as the arrays are full, then send the data to the Host PC */

while(index <= BUF_SIZE){
  if(index == BUF_SIZE){
        INT_DISABLE();

        send_msg(LIA_CHAN_NO,freq1,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
        send_msg(LIA_CHAN_NO,damp1,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
        send_msg(LIA_CHAN_NO,mode1,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
        send_msg(LIA_CHAN_NO,freq2,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
        send_msg(LIA_CHAN_NO,damp2,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
        send_msg(LIA_CHAN_NO,mode2,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));

  } /* end of if statement */
 } /* end of while loop */
} /* end of main program */


/*************************************************************************
                Interrupt Service Routine
This subroutine contains the data acquistion and system identification code.  It is
executed at the sampling rate.
*************************************************************************/
void c_int04(void)
{
volatile long clear;
clear = *DM1_INT_STATUS; /* Read Interrupt Status Register to Clear Interrupts */
y  = *DM1_CH0_IN_DATA;  /* Read Channel 0 Input Data Register */
u  = *DM1_CH1_IN_DATA;  /* Read Channel 1 Input Data Register */
y  = (y >> 16);                 /* Shift data by 16 bits to map DSPLINK */
u  = (u >> 16);
TY = -1*((float)y)/16384;      /* True Value in volts */
```

```
TU = -1*((float)u)/16384;

/* Implement a fourth order digital low-pass Butterworth filter
  on both inputs.  ( wc =  75 Hz) */

/* Response Signal (channel 0) */
xa4 = xa3;
xa3 = xa2;
xa2 = xa1;
xa1 = xa;
xa  = TY;
ya4 = ya3;
ya3 = ya2;
ya2 = ya1;
ya1 = FY;
FY  = 3.692234261*ya1- 5.123180777*ya2 + 3.165651468*ya3 -
0.734870850*ya4 + 1.0e-04*( 0.103686192*xa + 0.414744770*xa1 +
0.622117156*xa2 + 0.414744770 *xa3 + 0.1036861926 *xa4);

/* Excitation Signal (channel 1) */
xf4 = xf3;
xf3 = xf2;
xf2 = xf1;
xf1 = xf;
xf  = TU;
yf4 = yf3;
yf3 = yf2;
yf2 = yf1;
yf1 = FU;
FU  = 3.692234261*yf1- 5.123180777*yf2 + 3.165651468*yf3 -
0.734870850*yf4 + 1.0e-04*( 0.103686192*xf + 0.414744770*xf1 +
0.622117156*xf2 + 0.414744770 *xf3 + 0.1036861926 *xf4);

counter += 1;
if(counter == 20){              /* time decimation to establish desired fs */

      /****** REAL-TIME SYSTEM IDENTIFICATIOIN CODE ******/

       yk4 = yk3;
       yk3 = yk2;
       yk2 = yk1;
       yk1 = yk;
       yk  = FY;       /* filtered system response */
```

```
        uk2 = uk1;
        uk1 = uk;
        uk  = FU;       /* filtered system excitation */
        vk4 = vk3;
        vk3 = vk2;
        vk2 = uk-(2*uk1)+uk2;
        P1  = yk1;
        P2  = yk2;
        P3  = yk3;  /* linear combiner (adaptive IIR filter) inputs */
        P4  = yk4;
        P5  = vk2;
        P6  = vk3;
        P7  = vk4;
        val  = P1*P1 + P2*P2 + P3*P3 +              /* Normalizer */
             P4*P4 + P5*P5 + P6*P6 + P7*P7;
        yout = P1*W1 + P2*W2 + P3*W3 + P4*W4 +
             P5*W5+P6*W6+P7*W7;                     /*  Adaptive Filter output */
        e = yk-yout;;                   /* Error */
        W1 = W1+e*alpha*(P1/val);
        W2 = W2+e*alpha*(P2/val);
        W3 = W3+e*alpha*(P3/val);  /* Widrow-Hoff Delta Rule */
        W4 = W4+e*alpha*(P4/val);
        W5 = W5+e*alpha*(P5/val);
        W6 = W6+e*alpha*(P6/val);
        W7 = W7+e*alpha*(P7/val);

/* use a Newton root finding algorithm to solve a sixth
   order equation for filter coefficients */
        for(i=0; i<=10; i++){
         fxo =  (Xo*Xo*Xo*Xo*Xo*Xo)+(W2*Xo*Xo*Xo*Xo*Xo)+
                ((W1*W3+W4)*Xo*Xo*Xo*Xo)+
                ((2*W2*W4-(W3*W3)+W4*W1*W1)*Xo*Xo*Xo)+
                ((-W1*W3*W4-W4*W4)*Xo*Xo)+(W2*W4*W4)*Xo+
                (W4*W4*W4);
         fxop = (6*Xo*Xo*Xo*Xo*Xo)+(W2*5*Xo*Xo*Xo*Xo)+
                ((W1*W3+W4)*4*Xo*Xo*Xo)+
                ((2*W2*W4-(W3*W3)+W4*W1*W1)*3*Xo*Xo)+
                (-W1*W3*W4-W4*W4)*2*Xo+(W2*W4*W4);
         Xn = Xo - fxo/fxop;  /* Newton Formula */
         Xo=Xn ;
        } /*end of root-approximation using Newton-Rapson */

        a12 = Xn;       /* adapt. IIR filt. coefficients */
```

```
        a22 = -W4/a12;
        a21 = (a22*W1-W3)/(a12-a22);
        a11 = -W1-a21;
        b2 = (a22*W5-W7)/(a22-a12);
        b1 = W5-b2;

        count2 += 1;
        if(count2 == 1){ /* save only every count2 result */

/* Finally, solve for modal parameters using filt. coefficients */
        freq1[index]=2*fs*sqrt((1+a11+a12)/(1-a11+a12));
        freq2[index]=2*fs*sqrt((1+a21+a22)/(1-a21+a22));
        damp1[index]=2*fs*(1-a12)/(freq1[index]*(1-a11+a12));
        damp2[index]=2*fs*(1-a22)/(freq2[index]*(1-a21+a22));
        mode1[index]=4*b1/(1-a11+a12);
        mode2[index]=4*b2/(1-a21+a22);
        count2 = 0;
        index += 1;
        }
        counter = 0;

} /* end of if statement that sets effective fs */
} /* end of Interrupt Service Routine */
```

```
/*************************************************************************
PROGRAM: C40DDOF.CMD

DESCRIPTION:    A linker command file that contains the linker options, standard
memory configuration and sections allocation to be used with the C40DDOF.C code
shown in this appendix.  This file is called by the batch file that runs the compiler.

Hadi Alhassani, KUAE
12/26/1996
*************************************************************************/
/* 1)  Linker Options */

-x                  /* Reread libraries if unresolved symbols have not been found.*/
-c                  /* ROM  autoinitialization.          */
-o C40DDOF.OUT      /* Linker option to name the output file.  */
-m C40DDOF.MAP      /* Linker option to generate a map file.   */
C40DDOF.OBJ         /* Input file specification.*/
-i c:\dsptools      /* PRSL directory      */
-l rts40.lib        /* Link small memory C40 PRTS Library.*/
-l prts40.lib       /* Link PRSL. */
-e c_int00          /* Define the entry point */

/* 2)  Standard Memory Configuration */
MEMORY
{
        IRAM0:  origin = 002FF800h     length = 0400h     /* Internal 0, 1k   */
        IRAM1:  origin = 002FFC00h     length = 0400h     /* Internal 1, 1k   */
        ERAM0:  origin = 00300000h     length = 10000h    /* External 0, 64k  */
        /* ERAM1:  origin = 00308000h  length = 8000h   /* External 1, 32k */*/
        PEROM:  origin = 40000000h     length = 8000h     /* EPROM, 32k  */
        ERAM2:  origin = 80000000h     length = 8000h     /* External 2, 32k  */
}

/* 3)  Allocate Sections into memory */
SECTIONS
{
        .text                   : { }  > ERAM2
        .data                   : { }  > ERAM0
        .vector align=512       : { }  > IRAM1
        .cinit                  : { }  > IRAM1
        .bss                    : { }  > ERAM0
        .stack                  : { }  > IRAM0
}
```

```
/*********************************************************************
```
PROGRAM: PCDDOF.C (Host PC code)

DESCRIPTION: This is the PC Host code that downloads the C40DDOF.OUT executable file onto the C40 system using the application NETAPI library. Dynamic memory allocation is used to store six large arrays that are sent back by the C40 system when the system identification process is complete. These 6 arrays contain the modal parameters of the first and second modes of the structure. They are written to data files that can be read be MATLAB and then displayed.

Hadi Alhassani
12/26/1996

```
*********************************************************************/
/*********************************************************************
                Header Files and Definitions
*********************************************************************/
#include <stdio.h>      /* Standard I/O library for C code*/
#include <conio.h>      /* Standard Console and Port I/O library */
#include "c4xapp.h"     /* NETAPI Applications library */
#include "chkerror.c"   /* Subroutine for NETAPI error handler */

#define  C40_FILE   "C40DDOF.OUT"     /* The executable to be downloaded */
#define  BUF_SIZE   3000              /* size of data arrays */
void checkReturnCode(UINT returnCode); /* error code function prototype */
/*********************************************************************
                Main Program
*********************************************************************/
void main (void)
{
 int i;
 ULONG NumToRec1;
 UINT buf_length = BUF_SIZE;
 UINT ret;
 PROC_ID *handle;
 FILE *cfPtr1, *cfPtr2;
 float *w1;
 float *z1;
 float *a1;
 float *w2;
 float *z2;
 float *a2;

 w1 = calloc(BUF_SIZE,sizeof(float)); /* Dynamic memory allocation */
```

```
  z1 = calloc(BUF_SIZE,sizeof(float));
  a1 = calloc(BUF_SIZE,sizeof(float));
  w2 = calloc(BUF_SIZE,sizeof(float));
  z2 = calloc(BUF_SIZE,sizeof(float));
  a2 = calloc(BUF_SIZE,sizeof(float));

if(w1==NULL||z1==NULL||a1==NULL||w2==NULL||z2==NULL||a2==NULL)
        printf("Sorry, dynamic memory could not be allocated. ");
else{

  system("cls");          /* clear screen */

  /***** Reboot the C40 system *****/
  printf("\nRebooting the C40 system ....\n");
  ret = Global_Network_Reboot();
  checkReturnCode(ret);

  /***** Open Processor on Site A*****/
  printf("Opening processors .....\n");
  ret=Open_Processor_ID(&handle,"CPU_A",NULL);
  checkReturnCode(ret);

  /***** Download C40 program to processor *****/
  printf("\nLoading program %s to C40 in Site A.....\n", C40_FILE);
  ret=Load_And_Run_File_LIA(handle,C40_FILE);
  checkReturnCode(ret);

  /* Wait for data to arrive and receive when ready */
  printf("Training . . . . . ");

  ret=Read_LIA_Words_32(handle,1,&NumToRec1);
  ret=Read_LIA_Floats_32(handle,buf_length,w1);

  ret=Read_LIA_Words_32(handle,1,&NumToRec1);
  ret=Read_LIA_Floats_32(handle,buf_length,z1);

  ret=Read_LIA_Words_32(handle,1,&NumToRec1);
  ret=Read_LIA_Floats_32(handle,buf_length,a1);

  ret=Read_LIA_Words_32(handle,1,&NumToRec1);
  ret=Read_LIA_Floats_32(handle,buf_length,w2);

  ret=Read_LIA_Words_32(handle,1,&NumToRec1);
```

```
ret=Read_LIA_Floats_32(handle,buf_length,z2);

ret=Read_LIA_Words_32(handle,1,&NumToRec1);
ret=Read_LIA_Floats_32(handle,buf_length,a2);

printf("\nData has been successfully received.\n");

/* Write results to data files */
 if ((cfPtr1 = fopen("DDMP1.dat","w")) == NULL)
        printf("File could not be opened.\n");
else{
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr1,"%.5f\n",w1[i]);
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr1,"%.5f\n",z1[i]);
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr1,"%.5f\n",a1[i]);
        fclose (cfPtr1);
   }
if ((cfPtr2 = fopen("DDMP2.dat","w")) == NULL)
        printf("File could not be opened.\n");
else{
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr2,"%.5f\n",w2[i]);
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr2,"%.5f\n",z2[i]);
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr2,"%.5f\n",a2[i]);
        fclose (cfPtr2);
   }
 /* Free memory */
 ret=Close_Processor_ID(handle);
 checkReturnCode(ret);
 Clear_All_Lib_Memory();
 checkReturnCode(ret);
 free(w1);
 free(z1);
 free(a1);
 free(w2);
 free(z2);
 free(a2);
}
} /* end of main program */
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PROGRAM:       C40DDOF.M
%
% DESCRIPTION:  This MATALB M-file reads the six data arrays created by the
% PCDDOF.C program which contain the modal parameters of the first two modes of
% vibration of the structure.
%
% Hadi Alhassani, KUAE
% 12/26/1996
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all
%       Read Data
cd c:\..\workfile
fid1=fopen('DDMP1.dat');                % read mode 1
[MP1,NP1]=fscanf(fid1,'%f');
fclose(fid1);
fid2=fopen('DDMP2.dat');                % read mode 2
[MP2,NP2]=fscanf(fid2,'%f');
fclose(fid2);
cd c:\matlab
%       Rearrange Data
np1 = NP1/3;
np2 = NP2/3;
freq1=MP1(1:np1);
damp1=MP1(np1+1:2*np1);
mode1=MP1(2*np1+1:3*np1);
freq2=MP2(1:np2);
damp2=MP2(np2+1:2*np2);
mode2=MP2(2*np2+1:3*np2);
%       Plot Results
figure,subplot(311),plot(freq1(10:np1)),title('First Mode Frequency')
subplot(312),plot(damp1(10:np1)),title('Damping')
subplot(313),plot(mode1(10:np1)),title('Modal Constant')
figure,subplot(311),plot(freq2(10:np2)),title('Second Mode Frequency')
subplot(312),plot(damp2(10:np2)),title('Damping')
subplot(313),plot(mode2(10:np2)),title('Modal Constant')
```

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

PROGRAM: C40DDOFF.C (C40 Source Code)

DESCRIPTION: This program is downloaded onto the C40 system using the PCDDOFF.C host program via the application library. The code in the ISR reads two signals, an excitation and a response of a low frequency flexible structure, apply a low-pass butterworth filter to isolate the first mode and a band-pass filter to isolate the second mode and use a SDOF adaptive IIR filter to identify the first two modes of vibration in real-time.

>>> SAMPLING FREQUENCY = 4 kHz <<<

Hadi Alhassani, KUAE
12/26/1996

```
*********************************************************************/
/*********************************************************************
            Header Files
*********************************************************************/
#include "c:\dsptools\math.h"          /* math library */
#include "c:\dsptools\intpt40.h"       /* interrupt support (PRSL) */
#include "c:\dsptools\compt40.h"       /* communication port support (PRSL) */
#include "carrier.h"            /* defines pointers to DSPLINK registers for DM */
/*********************************************************************
      Define Constants
*********************************************************************/
#define  BUF_SIZE    3000          /* define array size */
#define  LIA_CHAN_NO  1            /* comm port channel number */
/*********************************************************************
      Global Variables
*********************************************************************/
long int counter = 0, index = 0, count2 = 0;
long y, u;
float yk=0,yk1=0,yk2=0,uk=0,uk1=0,uk2=0;
float ykb=0,yk1b=0,yk2b=0,ukb=0,uk1b=0,uk2b=0;
float Freq1[BUF_SIZE], Damp1[BUF_SIZE], Mode1[BUF_SIZE];
float Freq2[BUF_SIZE], Damp2[BUF_SIZE], Mode2[BUF_SIZE];
float P1=0, P2=0, P3=0, alpha1=0.009 , fs=250;
float P1b=0, P2b=0, P3b=0, alpha2=0.015;
float TY, TU, FYL=0, FUL=0, FYB=0, FUB=0;
float ya1=0, ya2=0, ya3=0, ya4=0;
float yf1=0, yf2=0, yf3=0, yf4=0;
float ya1b=0, ya2b=0, ya3b=0, ya4b=0;
float yf1b=0, yf2b=0, yf3b=0, yf4b=0;
```

```
float xa=0, xa1=0, xa2=0, xa3=0, xa4=0;
float xf=0, xf1=0, xf2=0, xf3=0, xf4=0;
float W1=0, W2=0, W3=0, yout=0, val=0, temp1=0;
float e1=0, omega1=0, zeta1=0, ms1=0, b1=0, a11=0, a12=0;
float W1b=0, W2b=0, W3b=0, youtb=0, valb=0, temp2=0;
float e2=0, omega2=0, zeta2=0, ms2=0, b2=0, a21=0, a22=0;
/*************************************************************************/
void c_int04(void);              /* Function prototype for the IIOF1 ISR */
/*************************************************************************
                    Main Program
*************************************************************************/
void main(void)
{
volatile int dummy;

for(index=0; index<=BUF_SIZE-1; index++){
        Freq1[index]=0;
        Damp1[index]=0;
        Mode1[index]=0;
        Freq2[index]=0;
        Damp2[index]=0;
        Mode2[index]=0;
        }
index=0;

asm(" PUSH AR0");           /* Set Up C40 Interruptts */
asm(" PUSH DP");            /* assembly language instructions */
asm(" LDI 030H,AR0");       /* needed to perform an interrupt */
asm(" LSH 16,AR0");         /* acknowledge instruction to allow */
asm(" IACK *AR0");          /* external interrupts to the C40. */
asm(" POP DP");
asm(" POP AR0");

INT_DISABLE();              /* Global disable of interrupts */
set_ivtp(DEFAULT);          /* Set IVTP on 512 word boundary,*/
                            /* see vector in linker file */
install_int_vector((void *)c_int04, 0x04);     /* Set the IIOF1 int vector */
load_iif(0x00B0);       /* Enable the IIF01 pin to be level trigger interrupt  */

/* Set up DSPLINK registers for the DMCB and DM. */
dummy = *DM1_RESET;             /* Do a read to Reset the Site A DM */
*DM1_ROUTE = 0x0000;   /* Set LKCLK1 to choose MCLK1 as system clock */
*DM1_INT_MASK = 0x00010000;     /* Interrupt when Input Data Regs are full */
```

```
*DM1_AMELIA_CTRL = 0xB30000;   /* CM6=0, board now in reset */
*DM1_AMELIA_CTRL = 0xF30000;   /* CM6=1, calibration cycle started */
                    /* CM0=1 CM1=1 and CM4=1; use MCLK1 as system clock*/
*DM1_USER_CONTROL = 0xA8E00000; /* Selcet prescale factor, clock source */
                    /* PT11=PT10=1,CS11=CS10=0,MCD1=1 and MCD2=1 */
                /* SAMPLING RATE IS NOW SET TO 4kHz */
*DM1_CONFIGURATION = 0xB3900000;/* Write the KEY for the Crystal ADM  */

INT_ENABLE(); /* Globally enable interrupts */

/* create a while loop that activates an INT_DISABLE function to halt the C40
operations as soon as the arrays are full, then send the data to the Host PC */

while(index <= BUF_SIZE){
  if(index == BUF_SIZE){
        INT_DISABLE();
        send_msg(LIA_CHAN_NO,Freq1,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
        send_msg(LIA_CHAN_NO,Damp1,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
        send_msg(LIA_CHAN_NO,Mode1,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
        send_msg(LIA_CHAN_NO,Freq2,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
        send_msg(LIA_CHAN_NO,Damp2,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
        send_msg(LIA_CHAN_NO,Mode2,BUF_SIZE,1);
        while(chk_dma(LIA_CHAN_NO));
}  /* end of if statement */
}  /* end of while loop */
}  /* end of main program  */

/*************************************************************************
                Interrupt Service Routine
This subroutine contains the data acquistion and system identification code.  It is
executed at the sampling rate.
*************************************************************************/
void c_int04(void)
{
volatile long clear;
clear = *DM1_INT_STATUS;        /* Read Interrupt Status Register */
                                /* to Clear Interrupts */
y = *DM1_CH0_IN_DATA;           /* Read Channel 0 Input Data Register */
```

```
u = *DM1_CH1_IN_DATA;          /* Read Channel 1 Input Data Register */
y = (y >> 16);                 /* Shift data by 16 bits to map DSPLINK */
u = (u >> 16);
TY = -1*((float)y)/16384;      /* True value in volts */
TU = -1*((float)u)/16384;

/* Implement a 4th order digital low-pass Butterworth filter to isolate the first mode of
vibration ( wc = 20Hz) */

/* Response Signal (Channel 0) 8/
xa4=xa3;
xa3=xa2;
xa2=xa1;
xa1=xa;
xa=TY;
ya4=ya3;
ya3=ya2;
ya2=ya1;
ya1=FYL;
FYL=3.917907865*ya1-5.757076379*ya2+3.760349508*ya3
-0.921181929*ya4+1.0e-08*(5.845142437*xa+23.380569747*xa1
+35.070854487*xa2+23.380569747*xa3+5.845142437*xa4);

/* Excitation Signal (Channel 1) */
xf4=xf3;
xf3=xf2;
xf2=xf1;
xf1=xf;
xf=TU;
yf4=yf3;
yf3=yf2;
yf2=yf1;
yf1=FUL;
FUL=3.917907865*yf1-5.757076379*yf2+3.760349508*yf3
-0.921181929*yf4+1.0e-08*(5.845142437*xf+23.380569747*xf1
+35.070854487*xf2+23.380569747*xf3+5.845142437*xf4);


/* Now, implement 4th order digital band-pass butterworth filter to isolate the second
mode of vibration (wl = 30, wh = 70Hz) **********/

/* Response Signal (Channel 0) */
ya4b=ya3b;
```

```
ya3b=ya2b;
ya2b=ya1b;
ya1b=FYB;
FYB=3.901065092*ya1b-5.717572207*ya2b+3.731457273*ya3b-
0.914975835*ya4b+9.44691844e-04*xa+1.33226763e-015*xa1
-1.88938369e-03*xa2+3.55271368e-015*xa3+9.44691844e-04*xa4;

/* Excitation Signal (Channel 1) */
yf4b=yf3b;
yf3b=yf2b;
yf2b=yf1b;
yf1b=FUB;
FUB=3.901065092*yf1b-5.717572207*yf2b+3.731457273*yf3b-
0.914975835*yf4b+9.44691844e-04*xf+1.33226763e-015*xf1
-1.88938369e-03*xf2+3.55271368e-015*xf3+9.44691844e-04*xf4;

counter += 1;
if(counter == 16){      /* time decimation to set effective fs */

    /****** REAL-TIME SYSTEM IDENTIFICATIOIN CODE ******/

    /** FIRST MODE */
    yk2=yk1;
    yk1=yk;
    yk=FYL;       /* filtered system response, first mode only */
    uk2=uk1;
    uk1=uk;
    uk=FUL;       /* filtered system excitation, first mode only */
    P1 = uk-(2*uk1)+uk2;          /* linear combiner (adaptive IIR filter) inputs */
    P2 = yk1;
    P3 = yk2;
    val = P1*P1+P2*P2+P3*P3;          /* Normalizer */
    yout = P1*W1+P2*W2+P3*W3;         /* linear combiner output */
    e1 = yk-yout;                     /* Error */
    W1 = W1+e1*alpha1*(P1/val);       /* Update the weights */
    W2 = W2+e1*alpha1*(P2/val);
    W3 = W3+e1*alpha1*(P3/val);
    b1 = W1;    /* Evaluate first adaptive filter coefficients using LC weights */
    a11 = -W2;
    a12 = -W3;
    temp1 = (1+a11+a12)/(1-a11+a12);
    omega1 = 2*fs*sqrt(temp1);  /* modal parameters of first mode */
    zeta1 = 2*fs*(1-a12)/(omega1*(1-a11+a12));
```

```
        ms1 = 4*b1/(1-a11+a12);

        /* SECOND MODE */
        yk2b=yk1b;
        yk1b=ykb;
        ykb=FYB;      /* Filtered system response, second mode only */
        uk2b=uk1b;
        uk1b=ukb;
        ukb=FUB;      /* Filtered system excitation, second mode only */
        P1b = ukb-(2*uk1b)+uk2b;    /* Linear combiner (adaptive IIR filter) inputs */
        P2b = yk1b;
        P3b = yk2b;
        valb = P1b*P1b+P2b*P2b+P3b*P3b;          /* Normalizer */
        youtb = P1b*W1b+P2b*W2b+P3b*W3b;    /* linear combiner output */
        e2 = ykb-youtb;                          /* Error */
        W1b = W1b+e2*alpha2*(P1b/valb);          /* Update the weights */
        W2b = W2b+e2*alpha2*(P2b/valb);
        W3b = W3b+e2*alpha2*(P3b/valb);
        b2 = W1b;   /* Evaluate second adaptive filter coefficients using LC weights */
        a21 = -W2b;
        a22 = -W3b;
        temp2 = (1+a21+a22)/(1-a21+a22);
        omega2 = 2*fs*sqrt(temp2);          /* modal parameters of second mode */
        zeta2 = 2*fs*(1-a22)/(omega2*(1-a21+a22));
        ms2 = 4*b2/(1-a21+a22);

        count2 += 1;
        if(count2 == 2){       /* keep every count2 result */
        Freq1[index]=omega1;        /* store modal parameters in arrays */
        Damp1[index]=zeta1;
        Mode1[index]=ms1;
        Freq2[index]=omega2;
        Damp2[index]=zeta2;
        Mode2[index]=ms2;
        index += 1;
        count2 = 0;
        }
        counter = 0;
} /* end if statement that sets effective fs */
} /* end Interrupt Service Routine */
```

```
/*********************************************************************
PROGRAM: C40DDOFF.CMD

DESCRIPTION:    A linker command file that contains the linker options, standard
memory configuration and sections allocation to be used with the C40DDOFF.C code
shown in this appendix.  This file is called by the batch file that runs the compiler.

Hadi Alhassani, KUAE
12/26/1996
*********************************************************************/
/* 1)  Linker Options */

-x                  /* Reread libraries if unresolved symbols have not been found.*/
-c                  /* ROM  autoinitialization.          */
-o C40DDOFF.OUT     /* Linker option to name the output file.  */
-m C40DDOFF.MAP     /* Linker option to generate a map file.   */
C40DDOFF.OBJ        /* Input file specification.*/
-i c:\dsptools      /* PRSL directory      */
-l rts40.lib        /* Link small memory C40 PRTS Library.*/
-l prts40.lib       /* Link PRSL. */
-e c_int00          /* Define the entry point */

/* 2)  Standard Memory Configuration */
MEMORY
{
        IRAM0:  origin = 002FF800h     length = 0400h     /* Internal 0, 1k   */
        IRAM1:  origin = 002FFC00h     length = 0400h     /* Internal 1, 1k   */
        ERAM0:  origin = 00300000h     length = 10000h    /* External 0, 64k  */
        /* ERAM1:  origin = 00308000h  length = 8000h     /* External 1, 32k */*/
        PEROM:  origin = 40000000h     length = 8000h     /* EPROM, 32k  */
        ERAM2:  origin = 80000000h     length = 8000h     /* External 2, 32k  */
}

/* 3)  Allocate Sections into memory */
SECTIONS
{
        .text                   : { }  > ERAM2
        .data                   : { }  > ERAM0
        .vector align=512       : { }  > IRAM1
        .cinit                  : { }  > IRAM1
        .bss                    : { }  > ERAM0
        .stack                  : { }  > IRAM0
}
```

```
/*********************************************************************
```
PROGRAM: PCDDOFF.C (Host PC code)

DESCRIPTION: This is the PC Host code that downloads the C40DDOFF.OUT executable file onto the C40 system using the application NETAPI library. Dynamic memory allocation is used to store six large arrays that are sent back by the C40 system when the system identification process is complete. These 6 arrays contain the modal parameters of the first and second modes of the structure. They are written to data files that can be read be MATLAB and then displayed.

Hadi Alhassani, KUAE
12/26/1996

```
*********************************************************************/
/*********************************************************************
                Header Files and Definitions
*********************************************************************/
#include <stdio.h>      /* Standard I/O library for C code*/
#include <conio.h>      /* Standard Console and Port I/O library */
#include "c4xapp.h"     /* NETAPI Applications library */
#include "chkerror.c"   /* Subroutine for NETAPI error handler */

#define  C40_FILE   "C40DDOFF.OUT"   /* Executable file to be downloaded */
#define  BUF_SIZE   3000             /* size of data arrays */
void checkReturnCode(UINT returnCode);  /* error code function prototype */
/*********************************************************************
                Main Program
*********************************************************************/
void main (void)
{
 int i;
 ULONG NumToRec1;
 UINT buf_length = BUF_SIZE;
 UINT ret;
 PROC_ID *handle;
 FILE *cfPtr1, *cfPtr2;
 float *w1;
 float *z1;
 float *a1;
 float *w2;
 float *z2;
 float *a2;

 w1 = calloc(BUF_SIZE,sizeof(float)); /* Dynamic memory allocation */
```

```
 z1 = calloc(BUF_SIZE,sizeof(float));
 a1 = calloc(BUF_SIZE,sizeof(float));
 w2 = calloc(BUF_SIZE,sizeof(float));
 z2 = calloc(BUF_SIZE,sizeof(float));
 a2 = calloc(BUF_SIZE,sizeof(float));

if(w1==NULL||z1==NULL||a1==NULL||w2==NULL||z2==NULL||a2==NULL)
        printf("Sorry, dynamic memory could not be allocated. ");
else{

 system("cls");          /* clear screen */

 /***** Reboot the C40 system *****/
 printf("\nRebooting the C40 system ....\n");
 ret = Global_Network_Reboot();
 checkReturnCode(ret);

 /***** Open Processor on Site A*****/
 printf("Opening processors .....\n");
 ret=Open_Processor_ID(&handle,"CPU_A",NULL);
 checkReturnCode(ret);

 /***** Download C40 program to processor *****/
 printf("\nLoading program %s to C40 in Site A.....\n", C40_FILE);
 ret=Load_And_Run_File_LIA(handle,C40_FILE);
 checkReturnCode(ret);

 /* Wait for data to arrive and receive when ready */
 printf("Training . . . . . ");

 ret=Read_LIA_Words_32(handle,1,&NumToRec1);
 ret=Read_LIA_Floats_32(handle,buf_length,w1);

 ret=Read_LIA_Words_32(handle,1,&NumToRec1);
 ret=Read_LIA_Floats_32(handle,buf_length,z1);

 ret=Read_LIA_Words_32(handle,1,&NumToRec1);
 ret=Read_LIA_Floats_32(handle,buf_length,a1);

 ret=Read_LIA_Words_32(handle,1,&NumToRec1);
 ret=Read_LIA_Floats_32(handle,buf_length,w2);

 ret=Read_LIA_Words_32(handle,1,&NumToRec1);
```

```
ret=Read_LIA_Floats_32(handle,buf_length,z2);

ret=Read_LIA_Words_32(handle,1,&NumToRec1);
ret=Read_LIA_Floats_32(handle,buf_length,a2);

printf("\nData has been successfully received.\n");

/* Write results to data files */
 if ((cfPtr1 = fopen("ModPar1.dat","w")) == NULL)
        printf("File could not be opened.\n");
else{
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr1,"%.5f\n",w1[i]);
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr1,"%.5f\n",z1[i]);
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr1,"%.5f\n",a1[i]);
        fclose (cfPtr1);
   }
if ((cfPtr2 = fopen("ModPar2.dat","w")) == NULL)
        printf("File could not be opened.\n");
else{
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr2,"%.5f\n",w2[i]);
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr2,"%.5f\n",z2[i]);
        for (i=1; i<BUF_SIZE; i++)
        fprintf(cfPtr2,"%.5f\n",a2[i]);
        fclose (cfPtr2);
   }
 /* Free memory */
 ret=Close_Processor_ID(handle);
 checkReturnCode(ret);
 Clear_All_Lib_Memory();
 checkReturnCode(ret);
 free(w1);
 free(z1);
 free(a1);
 free(w2);
 free(z2);
 free(a2);
}
} /* end of main program */
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PROGRAM:       C40DDOFF.M
%
% DESCRIPTION:  This MATALB M-file reads the six data arrays created by the
% PCDDOFF.C program which contain the modal parameters of the first two modes
%of vibration of the structure.
%
% Hadi Alhassani, KUAE
% 12/26/1996
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all
%       Read Data
cd c:\..\workfile
fid1=fopen('DDMP1.dat');            % open file
[MP1,NP1]=fscanf(fid1,'%f');        % read first mode data
fclose(fid1);
fid2=fopen('DDMP2.dat');            % open second file
[MP2,NP2]=fscanf(fid2,'%f');        % read second mode data
fclose(fid2);
cd c:\matlab
%       Rearrange Data
np1 = NP1/3;
np2 = NP2/3;
freq1=MP1(1:np1);
damp1=MP1(np1+1:2*np1);
mode1=MP1(2*np1+1:3*np1);
freq2=MP2(1:np2);
damp2=MP2(np2+1:2*np2);
mode2=MP2(2*np2+1:3*np2);
%       Plot Results
figure,subplot(311),plot(freq1(10:np1)),title('First Mode Frequency')
subplot(312),plot(damp1(10:np1)),title('Damping')
subplot(313),plot(mode1(10:np1)),title('Modal Constant')
figure,subplot(311),plot(freq2(10:np2)),title('Second Mode Frequency')
subplot(312),plot(damp2(10:np2)),title('Damping')
subplot(313),plot(mode2(10:np2)),title('Modal Constant')
```