A Simulink-Driven Dynamic Signal Analyzer

by

Katherine A. Lilienkamp

Submitted to the Department of Mechanical Engineering
in partial fulfillment of the requirements for the degree of

Bachelor of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1999

Author .....
Department of Mechanical Engineering
January 27, 1999

Certified by .....
David L. Trumper
Rockwell Associate Professor of Mechanical Engineering
Thesis Supervisor

Accepted by .....
Ernest G. Cravalho
Chairman, Undergraduate Thesis Committee
Department of Mechanical Engineering

A Simulink-Driven Dynamic Signal Analyzer

by

Katherine A. Lilienkamp

Submitted to the Department of Mechanical Engineering
on January 27, 1999, in partial fulfillment of the requirements for
the degree of Bachelor of Science in Mechanical Engineering

Abstract

Fourier methods can transform the system response to an input sine wave from a discrete data vector in the time domain into the frequency domain components of magnitude and phase at this excitation frequency. The dynamic signal analyzer described here uses such a process to identify the non-parametric transfer function of such a LTI system by sweeping, one-at-a-time, through a range of desired frequencies. The analyzer consists of a Simulink block and a MATLAB script file; together, they access and process data on a dSPACE controller board. The board can, in turn, send and receive signals with an analog system of interest. The sampling rate of the dSPACE board limits the bandwidth of the analyzer to a maximum of about 1 kHz. This document describes the implementation of the dynamic signal analyzer and also serves as a practical guide to its use.

Thesis Supervisor: David L. Trumper
Title: Rockwell Associate Professor of Mechanical Engineering

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Purpose

This document has two principal purposes. First, it describes the theory and operation of a dynamic signal analyzer (or DSA). The DSA will be used as a tool for students in the undergraduate course ‘Mechatronics’ (2.737) at MIT. This report should serve as instruction manual to enable the reader to understand and to operate the dynamic signal analyzer.

In describing the implementation of the DSA, I have a second intention, which I hope you will keep in mind. This dynamic signal analyzer was specifically developed to run on a dSPACE board using basic tools available in the MATLAB/Simulink environment. These tools include MATLAB m-files, the programs COCKPIT and TRACE, and the MATLAB functions mlib and mtrace. All are valuable resources for investigating plant characteristics and designing successful control systems with the dSPACE boards. The specific descriptions which follow can (and should) therefore be used both as a technical manual for understanding the DSA I have implemented and secondly, though no less importantly, as a tutorial guide in becoming familiar with each of the other tools mentioned above.

1.2 What is the Dynamic Signal Analyzer?

Stated briefly, the Dynamic Signal Analyzer identifies the non-parametric transfer function from one system node to another. A sine wave excites the system at a particular desired frequency. Data recorded from the two system locations are then analyzed, compared, and stored. By repeating this process at each discrete frequency of interest, the signal analyzer generates a Bode plot of the system in an automated way.

Figure 1.1 illustrates this process. The signal analyzer outputs a sine wave at each specified frequency to be tested. Once the system has settled into sinusoidal steady state, so that no significant transient response remains, channels 1 and 2 collect data from the desired locations in the system. The gain and phase calculated *from* channel 1 *to* channel 2 at each tested frequency are used to create the transfer function output by the analyzer.

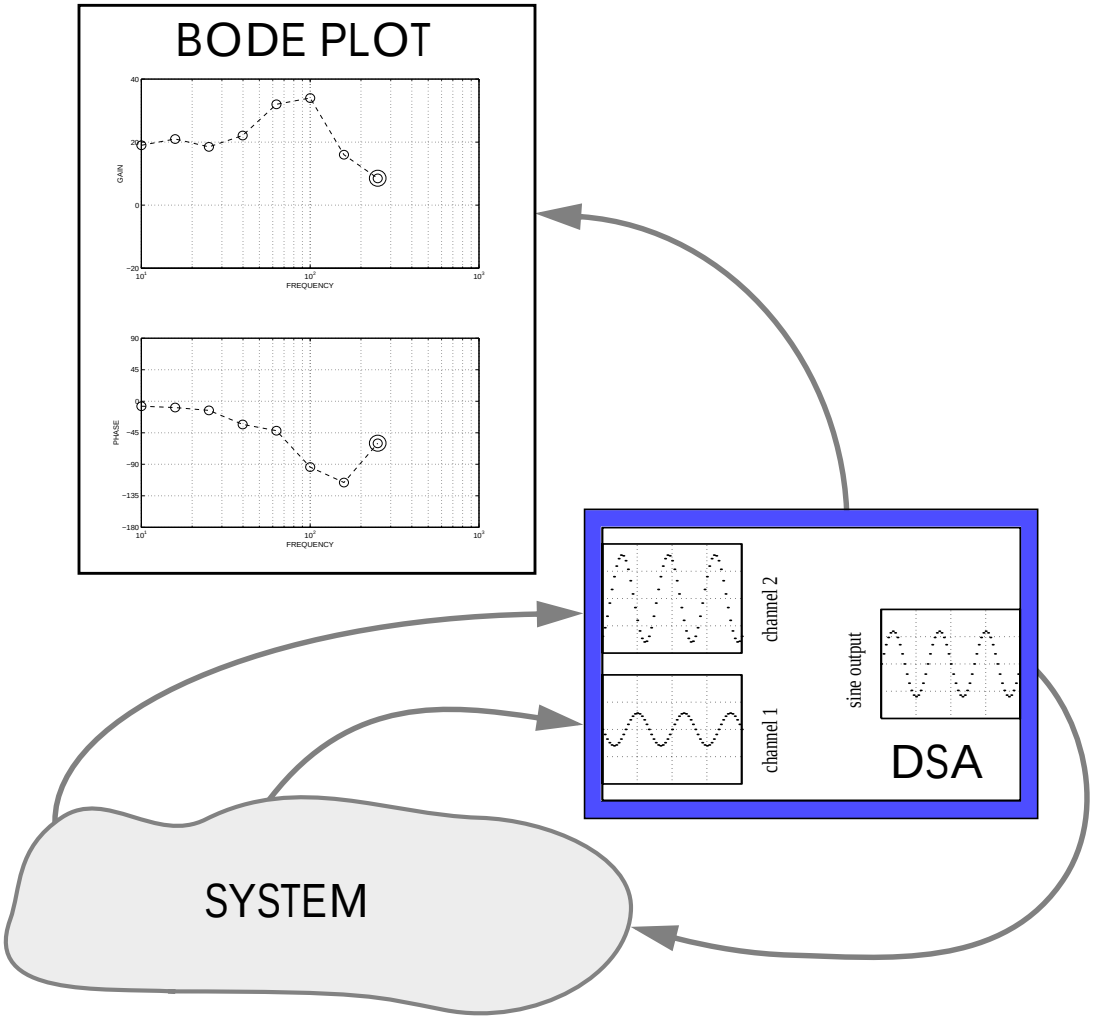


Figure 1.1: DSA and a System to be Analyzed

The DSA operates digitally; it is implemented entirely on the computer. The system to be analyzed may simply be a Simulink model, or, using the D/A and A/D capabillities of the dSPACE board, it may also include components in the ‘real’, analog world.

1.3 Motivation for a Dynamic Signal Analyzer

The dynamic signal analyzer is a tool for characterizing system dynamics in the frequency domain. It will be used by students studying digital controller design of electro-mechanical systems. The output from the DSA is a transfer function which maps both gain and phase as functions of frequency between two nodes of a system. Designers must obtain such empirical data to support and refine models of real-world systems and of combined controller-system dynamics. The signal analyzer simply automates this data collection and processing to provide the information more quickly and accurately than a user could obtain it 'by hand'.

Students should still become familiar with the process of collecting such data 'by hand'. This can be done by (1) using a sine wave output from a function generator to excite a system, (2) recording the requisite gain and phase shift between two system locations from an oscilliscope, once the transient response has subsided, and then (3) repeating this process at a variety of frequencies within a range of interest. The bandwidth of the dynamic signal analyzer is limited by the approximately 10 kHz maximum sampling rate of the dSPACE board, so that, typically, the DSA cannot characterize system dynamics above 1 kHZ. (See chapter 5 for more details.) The user may therefore be required to obtain some data 'by hand' to complete the data set output by the signal analyzer.

1.4 Roadmap

Chapter 2 develops theory to support the 'swept sine' method used by the dynamic signal analyzer. Chapter 3 details the implementation of the DSA, and chapter 4 is a guide to its features and operation. These two chapters also introduce the basic tools mentioned in section 1.1. Several appendices provide more detailed descriptions on basic use of each. These appendices may be referenced directly to obtain concise help getting started with the dSPACE programs COCKPIT and TRACE and in writing script files which use the

MATLAB functions mlib and mtrace to communicate with the dSPACE board in real-time.

Chapter 5 provides comparisons between the simulink-driven DSA and a commercial analyzer manufactured by Hewlitt-Packard and discusses limitations and the expected magnitude of error of the DSA. Finally, chapter 6 suggests some possible modifications to the DSA and gives guidelines for developing novel MATLAB code in the dSPACE environment.

1.5 Acknowledgements

The dynamic analyzer project stemmed from my enrollment in the class 2.73, 'Mechatronics'. I would like to thank Professor Trumper for suggesting this thesis topic and for his support and guidance as my thesis advisor. Steve Ludwick's patient supervision in navigating the mechatronics laboratory at MIT enabled my undertaking of this project, and his support through its completion is also appreciated.

Chapter 2

Theory

2.1 Scope of Theoretical Presentation

Digital controller design is often simplified by transforming data from the time domain into the frequency domain. Fourier's principles of harmonic analysis facilitate this conversion. Fourier transformation converts a function of time, $f(t)$, into a function of frequency, $F(j\omega)$, and the inverse Fourier transform provides the reverse mapping:

$$F(j\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt \tag{2.1}$$

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(j\omega)e^{j\omega t}d\omega \tag{2.2}$$

For digital computations, the counterparts to these continuous integrals form the discrete Fourier transform pair, or DFT:

$$F_n = \sum_{k=0}^{N-1} f_k e^{-j2\pi n(k/N)} \tag{2.3}$$

$$f_k = \frac{1}{N}\sum_{n=0}^{N-1} F_n e^{j2\pi n(k/N)} \tag{2.4}$$

The fast Fourier transform, or FFT, is a clever implementation of this, developed to minimize the number of computations necessary to calculate the DFT. Modern computer technology and FFT algorithms have, in fact, made the field of digital signal processing possible; Fourier methods are the basis of frequency domain analysis.

Most texts on DSP develop the ideas of digital Fourier analysis; several refences are suggested at the end of the chapter which provide more detailed coverage of this topic. This chapter presents aspects of the Fourier series and Fourier transformation relevant to

the implementation of the dynamic signal analyzer. The presentation here is primarily graphical. The objective is to explain the elegant and unique properties of harmonic series which allow direct conversion of sampled data into a frequency domain representation; this is the basic process done by the DSA. The visual format is intended to be intuitive and concise.

2.2 Some Relevant Properties of Fourier Series and Integral

A periodic signal can be represented as a weighted sum of harmonic functions: its Fourier series. The *orthogonality* of harmonics allows us to find these individual weightings ***independently of one another***, using a least-squares fit. Their *completeness* assures that when we find all such individual weightings for a given function, $f(t)$, and then recombine these harmonic components, the sum will be the original function, $f(t)$. We will not have missed any part of the original signal.

Much more can be said about the unique properties of Fourier series. The purpose of this section is to give the reader enough basic intuition to understand how the dynamic signal analyzer is able to extract gain and phase from response data, one frequency at a time.

Any system to be analyzed using the techniques described is assumed to to be linear and time-invariant. Linearity incorporates superposition: that if we add two input signals, the response can be found by adding the two individual outputs, and scaling: that is, in our range of interest, an amplitude change in input signal will result in the same relative change in the output. Time-invariance simply means that shifting the input signal in time shifts the output by the same amount in time. These are basic assumptions.

2.2.1 Summation of Harmonics

Adding a sine and cosine wave of a particular frequency results in a new harmonic wave at the same frequency. An example, represented in the time domain, is shown in Figure 2.1. Figure 2.4 presents the same information in the frequency domain. The latter representa-

tion more clearly illustrates how to obtain the resulting waveform.

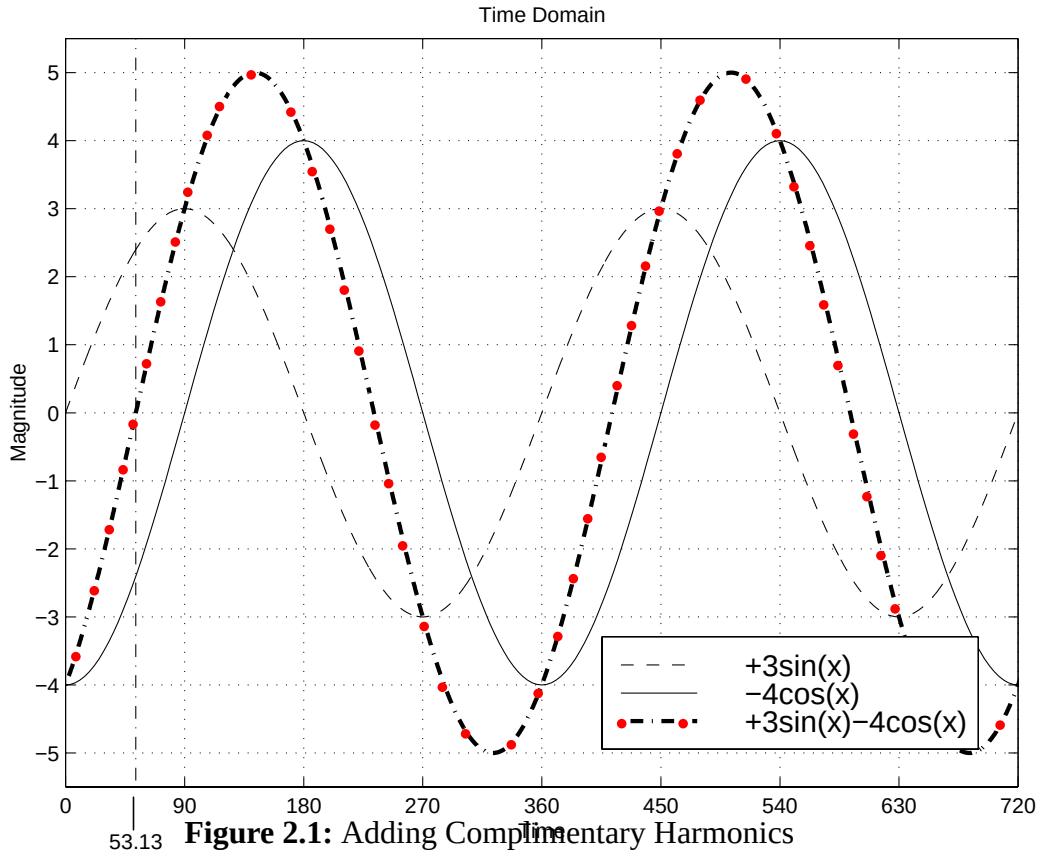


Figure 2.1: Adding Complimentary Harmonics

Figure 2.1 shows how the following function can be built from its individual components of sine and cosine. One representation of the resulting function is:

$$f(t) = a_1 \sin(x) + a_2 \cos(x) \qquad (2.5)$$

with $x = 2\pi\omega t$. In this example, the two component amplitudes are $a_1 = 3$ and $a_2 = -4$. For this example the frequency is $\omega = (1/360)$ [rad/sec], so the x-axis can easily be associated with degrees, as well. Plotting *sine vs cosine* displays their

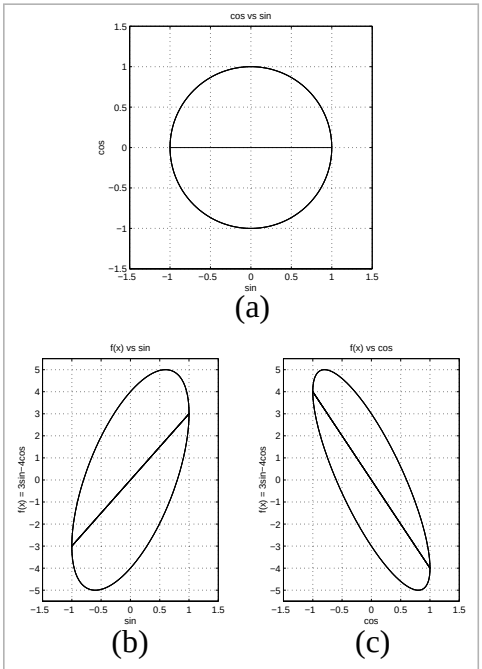


Figure 2.2: Correlated Functions

orthogonality. (See Fig. 2.2 (a).) For each sine coordinate along the x-axis in the plot, either:

1. There are two cosine coordinates. They are equal in magnitude and opposite in sign, adding to zero.

or

2. There is one cosine coordinate, and it is zero.

In either case, the sum of all cosine values at a particular sine value is zero, and cosine is therefore uncorrelated with sine. Swapping the axes, clearly sine is uncorrelated to cosine, as well. In contrast, the plot of $f(t)$ *vs sine* over one period is asymmetric about the x-axis. If we average the values at each point along the x-axis, the result is a line segment. Its slope is equal to the amplitude of the sine wave component of the function, '+3', as shown in Fig. 2.2 (b). Performing the same correlation routine with $f(t)$ *vs cosine* extracts the amount, or amplitude, of the cosine component. Note in Fig. 2.2 (c), that slope is '-4'.

The orthogonality of sine and cosine is closely related to their lack of correlation. Two vectors are orthogonal if their inner product is zero. Figure 2.3 shows two such vectors, defined generally in N dimensions as:

$$U = (u_1, u_2, \ldots, u_N)$$

$$V = (v_1, v_2, \ldots, v_N)$$

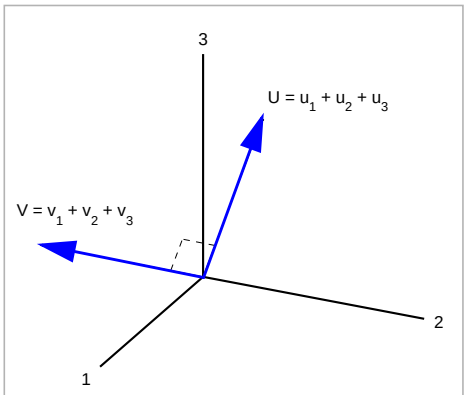


Figure 2.3: Orthogonal Vectors

Connecting the end points will create a right triangle, the side lengths related as:

$$|U - V|^2 = |U|^2 + |V|^2 \tag{2.6}$$

Evaluating the left-hand side:

$$|U - V|^2 = |U|^2 + |V|^2 - 2\{u_1v_1 + u_2v_2 + \ldots + u_Nv_N\} \tag{2.7}$$

The quantity is the curly brackets in Eq. 2.7 *is* the inner, or dot, product:

$$U \cdot V = \{u_1 v_1 + u_2 v_2 + \ldots + u_N v_N\} \tag{2.8}$$

This dot product must equal zero for orthogonal vectors, since Eq. 2.6 holds.

The concept of orthogonal functions is very similar. If the functions are *discrete* vectors of the same length, Eq. 2.8 gives the dot product, which again will be zero if and only if the functions are orthogonal. We can rewrite this for $u(t)$ and $v(t)$ as:

$$\sum_{n=0}^{N-1} u_n v_n = 0 \tag{2.9}$$

Sine and cosine satisfy this requirement, *if these N samples are evenly spaced along an integral number of complete cycles.* This sampling requirement is important to the dynamic signal analyzer and will be discussed in more detail in Section 2.3. In fact, every sine and cosine wave is orthogonal to every other sine and cosine; harmonics at different frequencies are not correlated. This can be represented as:

$$\int_{-\pi}^{\pi} (\cos mx)(\cos nx)dx = 0 \text{ and } \int_{-\pi}^{\pi} (\cos mx)(\sin nx)dx = 0 \tag{2.10}$$

when *m* and *n* are different integers.

More intuitively, if you change the magnitude of any one, harmonic component of a signal, you will not affect the magnitudes of other, different harmonic components of the signal. They act (and add) indepently. Only the *output* is changed. This is the essence of orthogonality.

Given a particular function, there must be some harmonic(s) which can be added to produce this output. No signal can be orthogonal to all harmonics. This is the essence of completeness. Together, orthogonality and completeness allow us to shift conveniently between the time and frequency domains.

Now that we have shown their orthogonality, we can create perpendicular axes with sine and cosine: the coordinates represent the amplitude of each harmonic. Using frequency as a third axis, we can show the components of sine and cosine at each frequency. Fig. 2.4 represents the function from Eq. 2.5. The magnitude of the sine component is +3, and the magnitude of cosine is -4. Translating this to get magnitude and phase is similar to a transposition from Cartesian to polar coordinates, as shown. The magnitude and phase

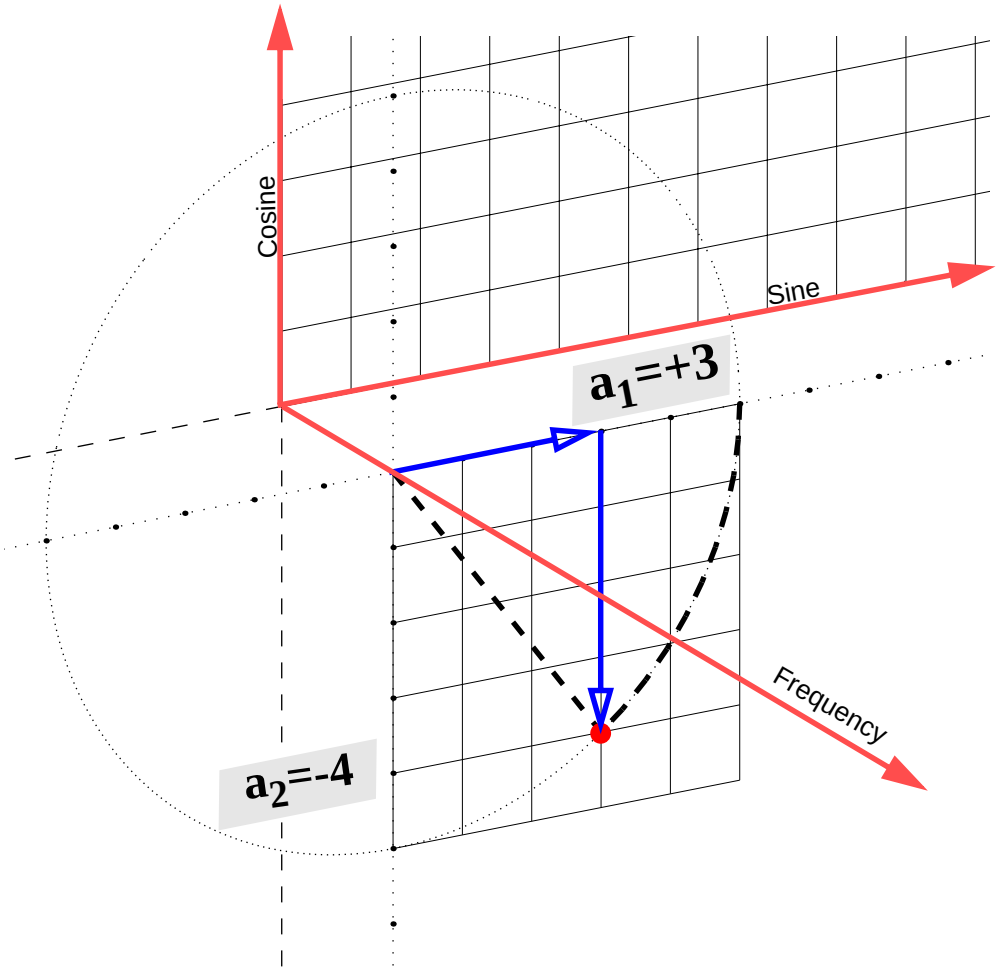


Figure 2.4: Translating a Harmonic to Find Magnitude and Phase

are familiar, geometric quantities. We can rewrite the function in Equation 2.5 as:

$$f(t) = A\sin(x+\phi_1) \quad \text{or} \quad f(t) = A\cos(x-\phi_2) \tag{2.11}$$

where magnitude, $A = \sqrt{a_1^2 + a_2^2}$, and phase as, $\phi_1 = \text{atan}(a_2/a_1)$ or $\phi_2 = \text{atan}(a_1/a_2)$, and where *atan* represents the four quadrant inverse tangent. A Bode plot is a familiar representation of this magnitude and phase information. Note that the complex exponential form below and Equations 2.5 and 2.11 are all completely identical:

$$f(t) = \frac{1}{2}(e^{j(x-\phi_2)} + e^{-j(x-\phi_2)}) \tag{2.12}$$

2.2.2 Harmonic Product

.

Just as the addition of a sine and cosine wave at a particular frequency will produce a new, pure harmonic, their point-wise multiplication will, as well. The top of Fig. 2.5 shows the result of multiplying two components of the function presented in Eq. 2.5. The resulting harmonic is a sine wave:

$$g(t) = a_1 \sin(x) \cdot a_2 \cos(x) = B \sin(2x) \tag{2.13}$$

The frequency of this new function is twice that of the component sine and cosine waves. The amplitude is the product of the component wave amplitudes:

$$B = a_1 \times a_2 \tag{2.14}$$

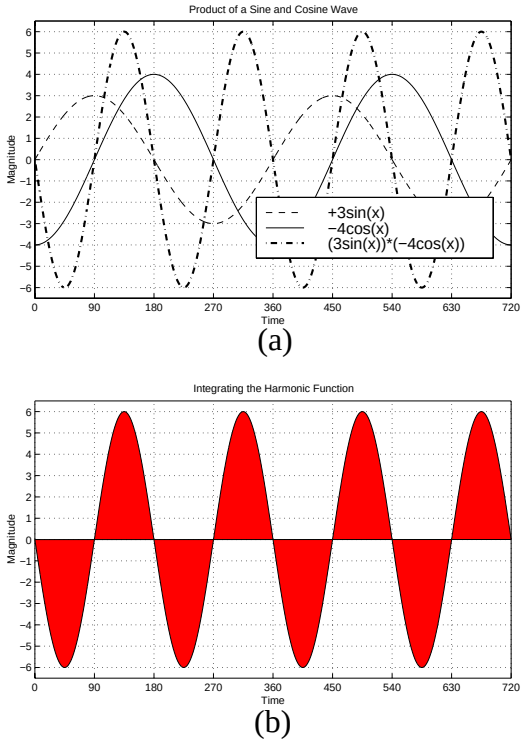
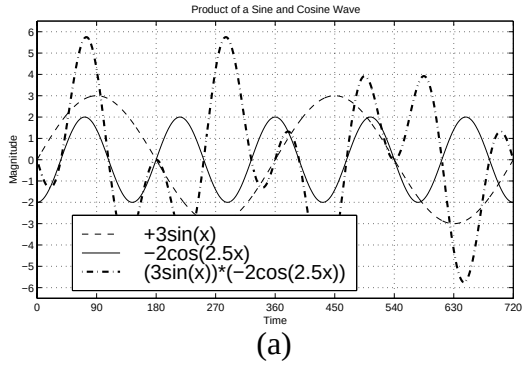


Figure 2.5: Pointwise Multiplication of Same-Frequency Sin and Cos Waves

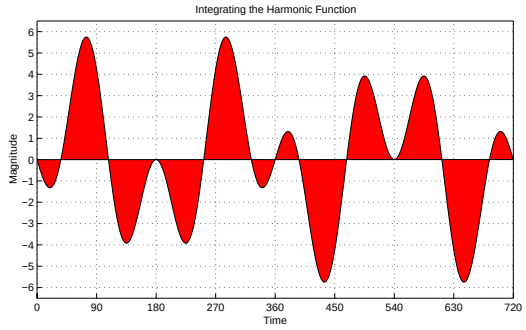
Multiplying two waves of differing frequency produces more ornate-looking results. Recall that any two, different harmonics will be orthogonal, if we use a time period that allows an integral number of each wave. Figure 2.6 shows the function:

$$h(t) = (3\sin(x)) \cdot (-2\cos((5/2)x)) \qquad (2.15)$$

over such a period.



(a)



(b)

Figure 2.6: Multiplying Harmonics of Differing Frequencies

In both examples, the orthogonality of the component waves means the *integral* of each resulting product will be zero. Now we can see more clearly why this is. With same-frequency sine and cosine waves, the resulting product is a sine wave which oscillates about zero. The integral of sine is zero over $2\pi n$. This integral is also zero for the general case of orthogonal waves. For example, in Fig. 2.6 (b), we can visualize spinning the resulting function 180 degrees about its center, (360,0). It is rotationally symmetric, so the integral is again clearly zero. This is essentially a restatement of the fact that the dot product of orthogonal vectors will be zero.

Now, we can consider two, non-orthogonal functions. Figure 2.7 uses the function, $f(t)$, defined in Eq. 2.5. This time, I have shown the product:

$$F_s(t) = \sin(x) \cdot f(t) \qquad (2.16)$$

Fig. 2.7 (b) shows the integral of $F_s(t)$. It is non-zero, because the multiplied vectors are non-orthogonal. The lightly-shaded tips show the regions which cancel one another; these areas are equal in magnitude but opposite in sign. The non-zero quantity remaining is the area of the darker region in this middle image. At the bottom, we see the result of multi-

plying sin(x) times itself: $(\sin(x))^2 = \frac{1}{2}(1 - \cos(2x))$

Because of superposition, we can separate the product in Eq. 2.16 to get:

$$F_s(t) = \sin(x) \cdot (3\sin(x) - 4\cos(x)) \quad (2.17)$$

$$F_s(t) = 3(\sin(x))^2 - 4(\sin(x) \cdot \cos(x))$$

Since sin(x) and cos(x) are orthogonal over this period, their contribution to $\int F_s(t)$ vanishes, and we need only consider $F_s(t) = 3(\sin(x))^2$. The magnitude of the sine component present, in this case +3, will equal the ratio of the integrals of $F_s(t)$ and $(\sin(x))^2$. Section 2.2.1 described how to create a harmonic out of individual, orthogonal components. Here, we have a method to accomplish the reverse - finding the individual, orthogonal components from a given signal. One final illustration, shown below, should make the process clear:

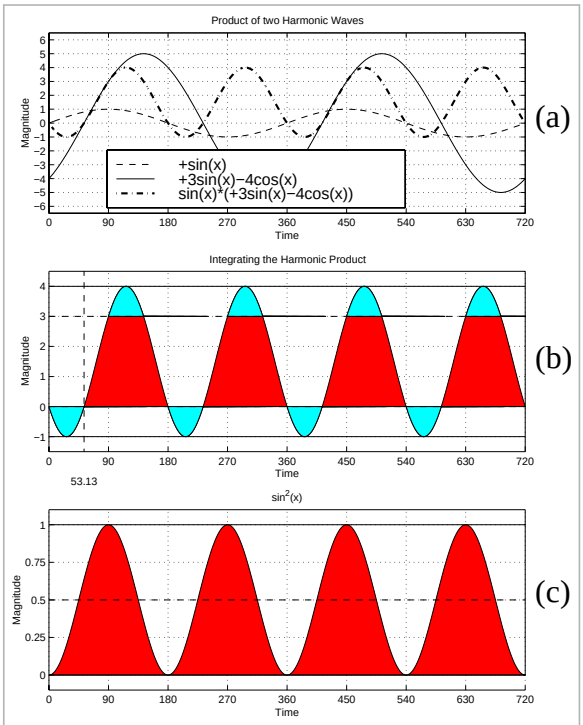
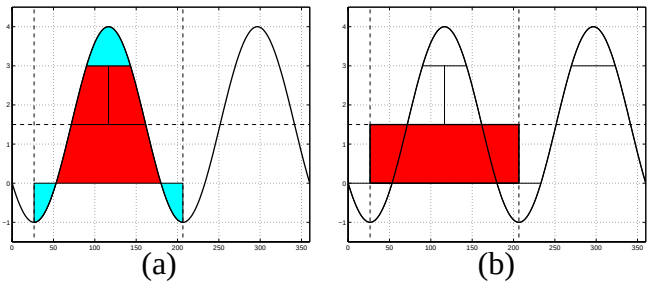


Figure 2.7: Integrating the Product of Non-orthogonal Functions

At top left, integrating sin(x).*f(x) over one wavelength. The light areas are opposite in sign and cancel. Manipulating the dark areas results in the total rectangular region shown at top right. Its area is one wavelength times the mean height of the dot-product wave. Here, that mean height is +1.5.



At bottom, the dot-product of f(x).*cos(x) can be manipulated (as above) to show the integral results in a rectangular area with height of -2. Again, this height is the mean height of the sum of the dot product. Multiplying this mean height by 2 results in the amplitude of the harmonic (sin(x) or cos(x)) used to create this dot product. Above: 2*(1.5) = +3. At right: 2*(-2) = -4. And our function is defined as: f(x) = (+3*sin(x))+(-4*cos(x)). [We have extracted these amplitudes.]

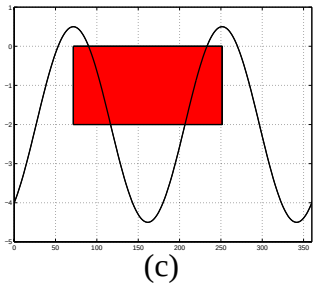


Figure 2.8: Extracting Amplitudes of Component Sine and Cosine

More formally, a given vector, *f(t)*, can be thought of as a single unit of a repeating sequence. Imagine copies of the function tacked end-to-end with one another to form a periodic sequence. This function in the time domain can be represented by a unique function in the frequency domain. That is, *f(t)* is a sum of a set of harmonics, each with some particular amplitude. If we integrate the product of *f(t)* with a sine wave at frequency ω, the contributions from all these other harmonic components in *f(t)* terms vanish [because of orthogonality] *except* the $\sin^2 \omega t$ term. (Figures 2.5 and 2.6 illustrate why the other terms vanish.) Likewise, for the product of *f(t)* and a cosine wave, only the $\cos^2 \omega t$ term remains. Figures 2.7 and 2.8 show how taking an integral of *f(t)* times some harmonic thus allows us to extract the component amplitude of that harmonic. The following equations provide the resulting integrals directly:

$$\int_0^T ((a_1 \sin \omega t) \cdot \sin \omega t) dt = \int_0^T \left(\frac{-a_1}{2} \cos 2\omega t + \frac{a_1}{2} \right) dt = \left[\frac{-a_1}{2} \sin 2\omega t + \frac{a_1 t}{2} \right]_0^T = \frac{a_1 T}{2} \qquad (2.18)$$

$$\int_0^T ((a_1 \cos \omega t) \cdot \cos \omega t) dt = \int_0^T \left(\frac{a_1}{2} \cos 2\omega t + \frac{a_1}{2} \right) dt = \left[\frac{a_1}{2} \sin 2\omega t + \frac{a_1 t}{2} \right]_0^T = \frac{a_1 T}{2} \qquad (2.19)$$

The result is that the magnitude of the sine component is equal to 2 times the mean height of this product-wave. (Divide the result from Eq. 2.18 (or Eq. 2.19) over one period, *T*, to get the mean, and then multiply by two: $2 \times [(a_1 T/2)/T] = a_1$. This returns the amplitude of the harmonic present in our function, *f(t)*.) Recall our original function from Eq. 2.5: $f(t) = 3 \sin x - 4 \cos x$. ($x = 2\pi t/360$ [rad/sec]) In figure 2.8 (a) and (b), the mean value of $f(t) \times \sin x$ is 1.5, so the amplitude of $\sin x$ within f(t) is $a_1 = 2 \times 1.5 = 3$. In Fig. 2.8 (c), the component amplitude of $\cos x$ in f(t) is twice the mean product wave value of negative two: $a_2 = 2 \times (-2) = -4$.

2.3 The Dynamic Signal Analyzer's Method: Swept Sine Response

The dynamic signal analyzer outputs a discrete sine wave at one particular frequency at a time. Data from the excited system are collected on two channels, A and B. The DSA uses

the procedure outlined in Section 2.2.2 to recover the sine and cosine components of each collected data set. The equations to derive the sine and cosine amplitudes from a discrete data vector, $y(k)$, are:

$$B_C = \frac{2}{N}\sum_{k=0}^{N-1} y(k)\cos(2\pi\omega k(\Delta t)) \quad ; \quad B_s = \frac{2}{N}\sum_{k=0}^{N-1} y(k)\sin(2\pi\omega k(\Delta t)) \tag{2.20}$$

where Δt is the sampling rate, ω is the frequency of the input sine wave, and N is the number of samples.

As mentioned on page 17, the data must be evenly spaced along an integral number of cycles. Otherwise, the orthogonality of the sine and cosine functions is lost, and the integrals will not provide the correct result. (Look at Figures 2.7 and 2.8 again, and it should be clear that integration must occur over a complete number of product-wave cycles.)

The dot products of the sampled vector with sine and cosine (Eq. 2.20) should eliminate most noise. Random noise will be uncorrelated to sine and cosine functions at that frequency (and to anything else, by definition). Other frequency components in the signal will be orthogonal to harmonics at the excitation frequency. They may have a small, non-zero contribution, since the total sampling period may (likely) not extend over an integral number of cycles for some given noise frequency, but this should not be significant.

Once the individual amplitudes of sine and cosine are found, using the equations in 2.20 , the information can be transformed into the frequency domain (see Section 2.2.1):

$$B = \sqrt{(B^2{}_c + B_s^2)} \quad ; \quad \phi = \operatorname{atan}\frac{B_c}{B_s} \tag{2.21}$$

B is the amplitude of the harmonic at this frequency, and ϕ is the phase. Calculating B_1 and ϕ_1 for a signal received by channel one, and B_2, ϕ_2 for channel two, the transfer function from one to two is:

$$|G(e^{2\pi\omega j})| = \frac{B_2}{B_1} \quad ; \quad \angle G(e^{2\pi\omega j}) = \phi_2 - \phi_1 \tag{2.22}$$

A **review** of the method for extracting transfer function data at a particular frequency:

1. Take data at a constant sampling rate. These N data points now form a vector.

2. For each point above, evaluate sin(t/T), where t is the time at which the point was sampled and T is the wavelength period at this particular frequency. This creates a new vector of length N. Do the same for cos(t/T).

3. Since any other harmonics are orthogonal to harmonics at our particular frequency, if we take the sum of the dot product between the data vector and the sine vector, *only the component part of sine at this particular frequency will have non-zero contributions to this integral*. Summing the dot product with cosine will likewise provide non-zero information only for the cosine function at this frequency.

4. We know, therefore, that a non-zero value of the 'dot-product sum' indicates the presence of the particular harmonic we have multiplied with our data vector (from step 1.) *But what exactly is this relationship?* This chapter has illustrated the DSA method to extract the amplitude present from this 'dot-product sum' in four, equivalent ways:

• Figure 2.8 shows rectangular areas of 'mean dot-product' height. Two times this mean height value yields the component sine or cosine amplitude.

• Eq.'s 2.18 and 2.19 derive the result in a continuous integral form.

• The underlined sentence on page 22 describes this verbally.

• Equation 2.20 restates the relationship in the discrete form used by the DSA.

2.4 Section References and Suggested Reading

Complete references for each of the sources below are in the Bibliography on page 67.

For a concise overview of Fourier analysis, 'FFT Fundamentals and Concepts' [10] by Ramirez uses direct, easy-to-understand language. This is a short book, written from the practical perspective of an engineer and intended to convey major concepts. The books by Paul Lynn [7] and by Stearns and Hush [12] have similar emphasis, while they provide more detailed examples of particular equations and concepts, like the Fourier transforms and convolution. I have borrowed the example of orthogonal vectors (on page 16) from Lynn's book.

Section 2.3, "The Dynamic Signal Analyzer's Method: Swept Sine Response" comes directly from "Digital Control of Dynamic Systems," [4] by Franklin, Powell and Workman. Their discussion of non-parametric system identification methods on pages 349-357 is my primary source in implementing the dynamic signal analyzer.

Chapter 3

Implementation in the Simulink/dSPACE Environment

3.1 Introduction

The dynamic signal analyzer is comprised of two separate parts which work in tandem: a Simulink subsystem block and a MATLAB function. The Simulink block interfaces with the dSPACE analog conversion channels and/or with controllers and systems in the Simulink models, allowing analyses of both analog and digital systems. The MATLAB function does the work of reading and writing data (using mlib and mtrace functions), interacting with the user during a run, and processing the sampled signal to approximate the transfer function at each frequency.

3.2 Design Goals

The analyzer should calculate the non-parametric transfer function up to 1 kHz with reasonably accurate results. It must also be useable. Below is a list of requirements and desired features to make the analyzer practical, intuitive and convenient to use.

1. It must not be difficult to learn to run the DSA. The user should be able learn enough by typing the m-function name in a MATLAB window to operate it at a basic level.
2. The Simulink DSA block should be simple and intuitive. It should be reasonably clear how to connect subsystem ports to other simulink blocks in a system.
3. The MATLAB function should exit gracefully if the user wishes to quit before a run is complete. Ideally, a button would allow this option at any time.
4. Pausing the program during a run is also desirable. If properly designed, a feature to pause and restart would allow the user to reset the amplitude of the output sine wave (for instance, by using COCKPIT and TRACE) to an appropriate value when necessary.
5. The program must display sampled data to allow the user to verify it is probably valid. A Bode plot of the transfer function should be displayed and, ideally, updated during the run.
6. The function must output vectors of raw data and of the corresponding frequencies.
7. Values for amplitude and frequency should be read directly from the dSPACE board at each step, so the user may pause and alter the amplitude with COCKPIT.

3.3 Flow Chart of the Design

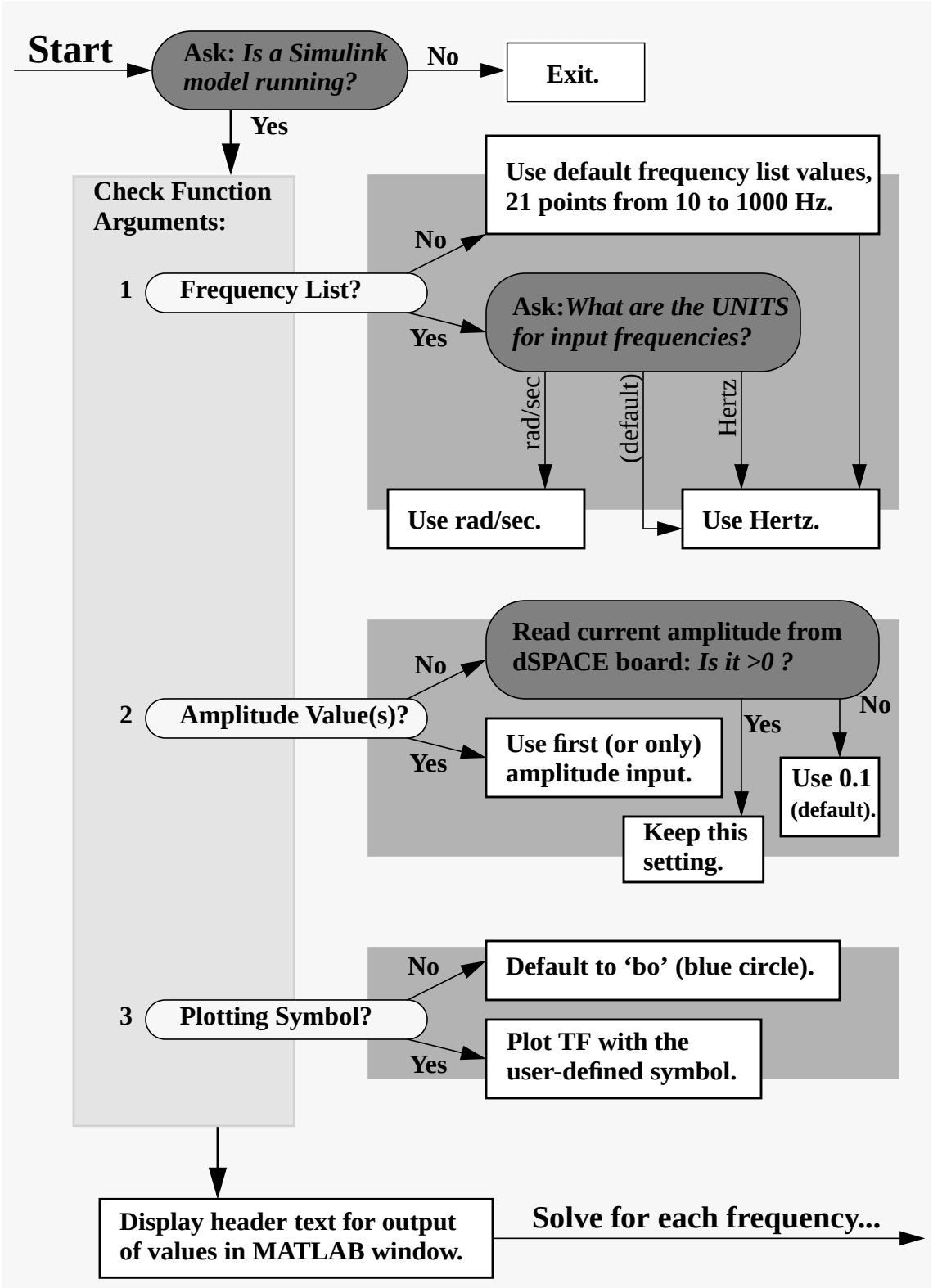


Figure 3.1: Design Flow Chart (left half)

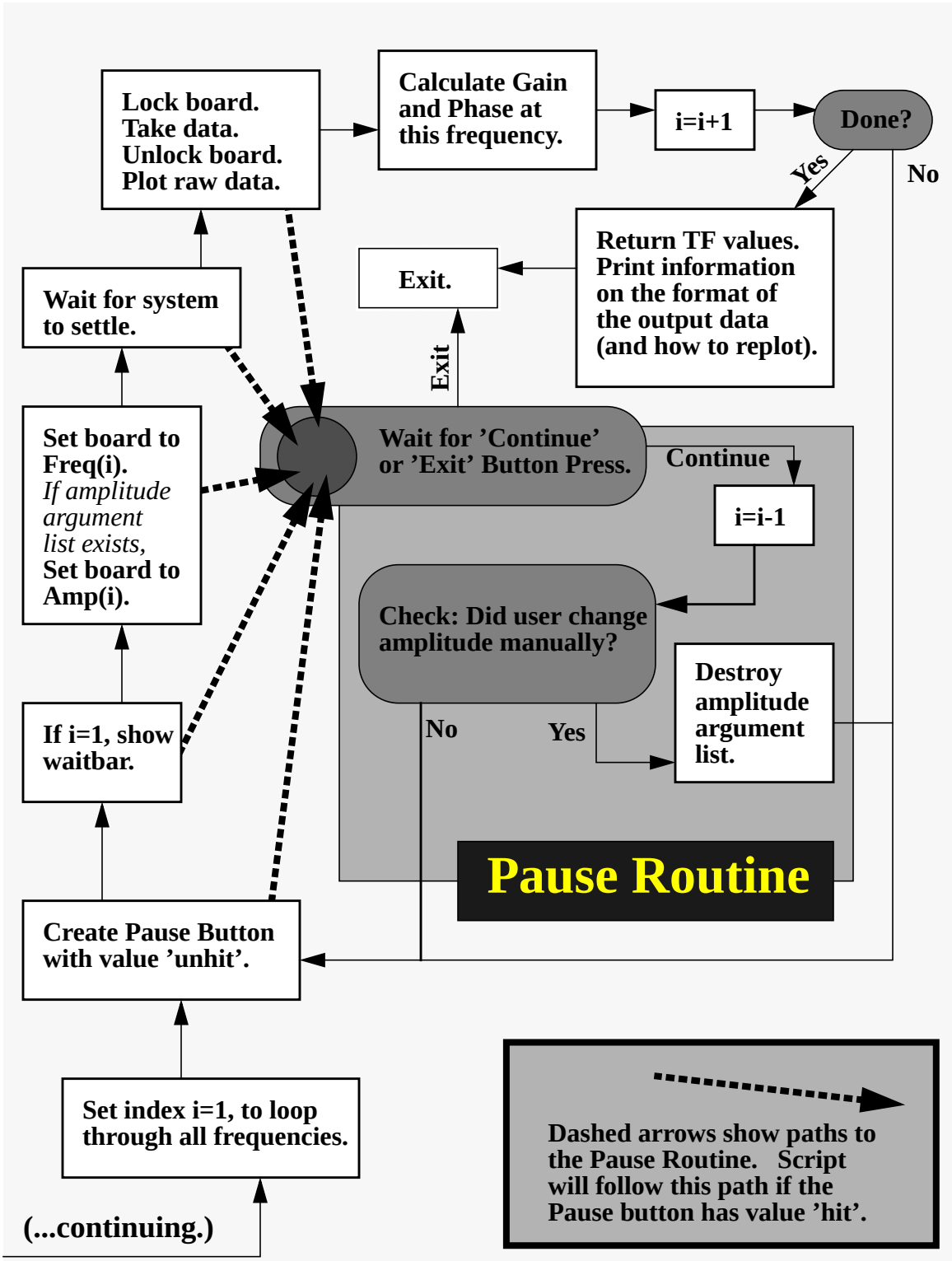


Figure 3.2: Design Flow Chart (right half)

The flow chart across the two, preceding pages illustrates the basic structure of the MATLAB script file. In Figure 3.1, the function parses user input arguments to make sure all necessary information is defined. There are default values for each of the three, possible arguments:

1. Frequency values. If the user does not specify desired values, the range is set from 10 to 1000 Hertz, with 21 points spaced equally on a logarithmic scale. If values are specified, the user is asked is the units are Hertz or rad/sec. Plots of the transfer function during the run will convert rad/sec to Hertz for the x-axis.

2. Amplitude value(s). If no value is given, the current 'Swept Sine.Amplitude' is read from the DSA block. A non-positive value is overwritten by the default of '0.1'. If a vector of values is supplied, each one is associated with the same index frequency value. If this vector is shorter than the frequency list length, the final value will be used for any remaining points to be analyzed.

3. Plotting symbol. The figure plotting the system transfer function is not automatically cleared with each run, so that data from sequential runs can be easily compared, if desired. This argument should be one of the acceptible MATLAB plotting options. The default uses blue circles to plot points.

The program loops through all requested frequency values. A break button signals a pause. At several points in the loop, there is a pause test. Once the program enters pause mode, it will wait until a click on either a 'Continue' or 'Exit' button. Exiting ends the run with a brief message acknowledging the break. If the program continues, data are collected for the previous frequency value, except if the current value was already the first point. The amplitude value is read from the dSPACE board, to compare with the last known value. If the two differ, the user has apparently changed the amplitude manually, for instance by using COCKPIT. Any list of amplitude values for the run is destroyed, if the program detects a manual user change. The amplitude will not change during the rest of the run, unless the user breaks and resets it again manually.

The output provides three columns, with frequency, magnitude and phase for each calculated point. A message with format and unit information is printed to inform the user.

Chapter 4

User's Guide to the Dynamic Signal Analyzer

4.1 Getting Started

The model 'dsa_demo.mdl' incorporates the DSA block in a simple model.

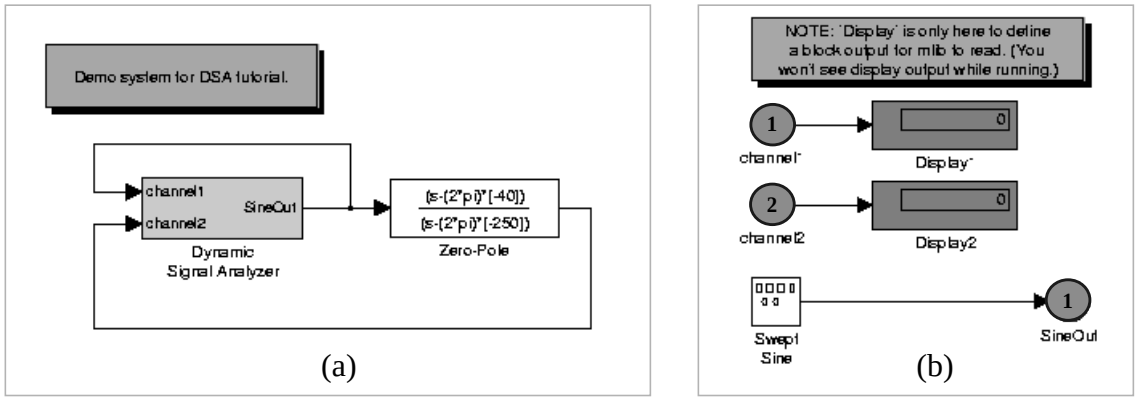


Figure 4.1: Demo Model 'dsa_demo.mdl'

In Figure 4.1 (a), output from the analyzer enters a Zero-Pole block. This system runs entirely on the dSPACE board, with no connections to the outside world. Double-clicking on the 'Dynamic Signal Analyzer' block will open the subsystem (shown Fig. 4.1 (b)). Channel1 measures the input to the Zero-Pole system, and channel2 measures its output.

4.1.1 Zero-Pole System

The demo system we will measure has one pole, with a breakpoint at 250 Hz, and one zero, with a breakpoint at 40 Hz. To convert to radians/sec for the zero-pole block, each value is multiplied by 2π. If you are creating this model from scratch, the values would be:

```
zero=(2*pi)*[-40]; pole=(2*pi)*[-250];
gain=[1].
```

MATLAB can provide a quick profile on the expected system Bode plot, as shown in Figure 4.2. In MATLAB:

```
>> sys1=zpk((2*pi)*[-40],(2*pi)*[250],[1]);
```

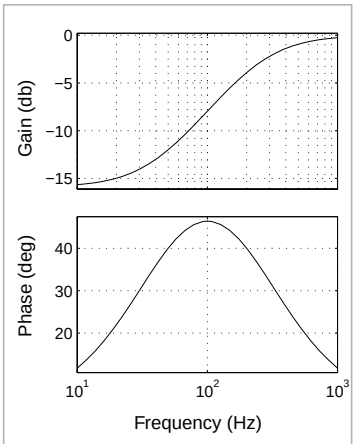


Figure 4.2: Expected System Bode Plot

```
>> bode(sys1)
```

4.1.2 Parameter Settings and Build

1. There should be a DSA directory containing two files you will need. **Copy these files** to a directory in your own locker:

- dsa_demo.mdl
- dsa_tf.m

2. Select the **Parameters:Settings** menu at the top of the model window. The demo model should already have the following settings:

- The **stop time** should be a huge number, like **1e10**, so we won't worry about the model ending during our experimentations.
- Use a **fixed step size** of **0.5e-4** (or **1.0e-4**)
- Solve using **Euler ode1**.

3. **Save** the Simulink model and select **Tools:RTW Build** to begin running it on the dSPACE board. (You may sometimes need to resave and begin a second build, due to the quirkiness of the file server in the lab.)

4. **Run the MATLAB function dsa_tf()** to calculate the system frequency response. You should be able run the program simply by entering

```
>> dsa_tf
```

at the MATLAB command prompt. You will be prompted to hit enter again to continue:

```
Note: Make sure your (DSA) model is built and running.
      This program will erase any images or plots in figure 1 and figure 2.

      If you would like to STOP this program now, enter 'q' to quit,
      otherwise, just hit enter to continue :
```

The analyzer will use a default sine wave amplitude of 0.1 range from 10 Hz to 1kHz. The next section provides a quick guide to using the function.

4.2 Running the MATLAB Function dsa_tf()

Here is an example of a command to run the analyzer MATLAB function:

```
>> my_tf = dsa_tf(10.^[1:.1:3],0.1,'bo')
```

The function takes up to 3 arguments, separated by commas. The first two are important:

1. **A vector of frequencies**. Frequencies below 10 Hz may take a fairly long time, since the program will wait for the system to settle, and you should not use values over 1kHz. In the underlined command above, the frequencies have logarithmic spacing from 10^1 to 10^3. [All of the argument values shown in the underlined example above are also the default values for the function.]

The program will ask you whether these values are in Hertz or radians/sec. Run-time transfer function plots will be done using Hertz for the x-axis, regardless. Each radian/sec value will be converted to Hertz for the plot.

2. **A vector or scalar for amplitude(s) for the sine wave generator.** If you use a list, the nth amplitude is associated with the nth value from your frequency list. If you just give a

single number, that amplitude will be used at all frequencies.

3. **A symbol to use for plotting the transfer function.** If you do two, consecutive runs, specifying a different symbol for each will make the plot more clear. Type

```
>> help plot
```

at the MATLAB prompt for a description of available plotting symbols. To erase a previos plot before a given run, clear the figure by typing:

```
>> figure(1)
>> clf
```

4.3 Pause and Other Features

You should now have dsa_tf running. The program will wait some time for the system to settle; you should a screen similar to the one in Figure 4.3, with a waitbar indicating progress overlaid.

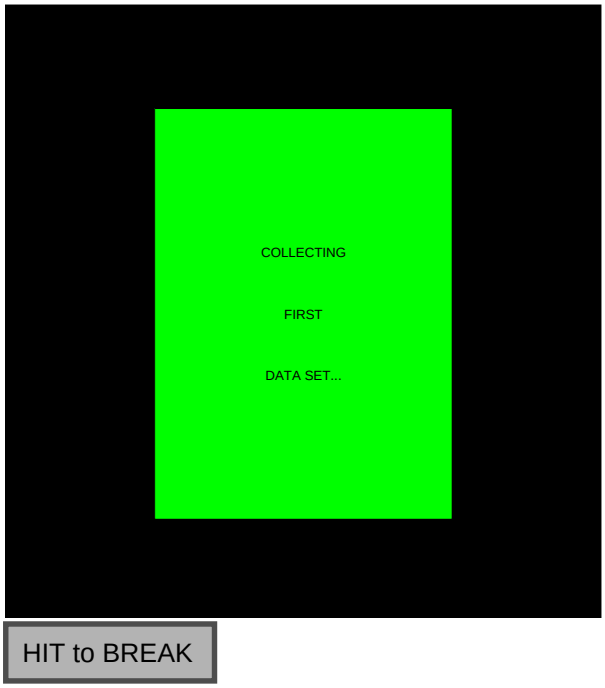


Figure 4.3: Initial Display

After a minute or so, the first set of data should be displayed in the left-hand MATLAB figure ('figure(2)' on your computer screen) and the first transfer function values should be plotted at the right ('figure(1)').

Let the analyzer collect the first few points. MATLAB should display two figures which look similar to those in Figure 4.4. While the analyzer is running, you should watch the figure at the left carefully, to make sure data from both **channel1** and **channel2** look reasonable. More specifically, you should see basically sinusoidal output, and you must be sure the peaks of the waves are not being clipped by saturation. The D/A channels will only allow values in the range of -1 to +1, for instance. If the signal looks problematic, you should hit the **Break** button at the bottom of the lefthand figure to pause the run.

Your data should look fine, but hit the Break button now, anyway, to experiment with stopping and restarting.

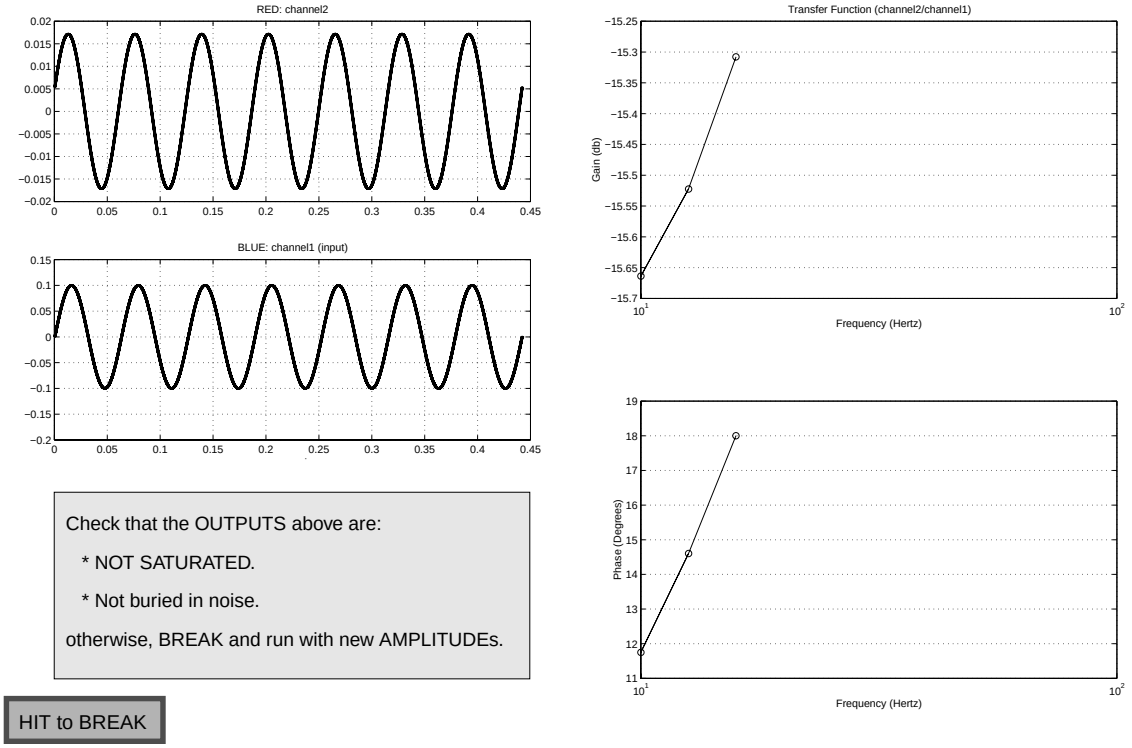


Figure 4.4: The Dynamic Signal Analyzer in Progress

When you hit the Break button, the bottom of the screen should have changed to display a new message, indicated the program has halted and reminding you that the sine wave amplitude can be reset while the program is in pause mode. Appendix C describes how to use COCKPIT to modify variables like the sine wave amplitude and to create displays of both system outputs and variables. Specifically, if the sinusoidal output from either channel looks saturated, you can use COCKPIT to reduce the sine wave amplitude while the program is in pause mode. There is also a guide to using TRACE, which operates like an oscilloscope for Simulink/dSPACE system outputs.

The analyzer will not continue until you select one of two buttons at the bottom of the lefthand figure: **Exit** or **Continue**. Choose one when you are ready

For the example MATLAB run below, the program was interrupted after the third data point (at 15.8 Hz), and restarted. Shortly after this, it was paused again and then exited. The transfer function for the first three points is output after the exit. Each row in the 3x3 matrix at the bottom of the page refers to a particular frequency (in Hertz). The second and third columns indicate gain and phase (see next section).

```
» dsa_tf7

Note: Make sure your (DSA) model is built and running.
      This program will erase any images or plots in figures 1 and 2.

      If you would like to STOP this program now, enter 'q' to quit,
      otherwise, just hit enter to continue :

Running 21 points.
  [Range = 10.00 [Hz] (min) to 1000.00 [Hz] (max)]
Current model amplitude is 0.100

---------------------------------------- :: ----------------------
_________Frequency_________   _SineAmp__  ::  __GAIN___      _PHASE_
   Hertz        [rad/sec]                ::     db          Degrees
---------------------------------------- :: ----------------------
  10.0000     [   62.8319]     0.100000   ::   -15.664         11.75
  12.5893     [   79.1006]     0.100000   ::   -15.522         14.60
  15.8489     [   99.5818]     0.100000   ::   -15.308         18.00
  19.9526     [  125.3660]     0.100000   ::
BREAK: going back to previous frequency.
 * Use COCKPIT to reset Amplitude.
 * USE TRACE to view resulting sine waves.
Restart DSA by hitting RESTART button in figure 2.
   15.8489     [   99.5818]     0.199000   ::
BREAK: going back to previous frequency.
 * Use COCKPIT to reset Amplitude.
 * USE TRACE to view resulting sine waves.
Restart DSA by hitting RESTART button in figure 2.

   ************************************************
   *** Program exited by user during run.... bye. ***
   ************************************************

ans =

   10.0000    0.1647    11.75
   12.5893    0.1674    14.60
   15.8489    0.1716    18.00
```

4.4 Data Output Format and Replotting

The function dsa_tf() returns a matrix of three columns.

1. Column one is a list of the **frequencies** analyzed, in the units selected by the user.

2. The second column gives the **gain** from channel1 to channel2 at this frequency, as an **absolute quantity**. (For instance, if the amplitude at channel2 is 10 times that of channel1, the gain returned will be 10.0, *not 20 db.*)

3. The third column gives the phase from channel1 to channel2 at this frequency, in units of **degrees**.

You can define a new variable to store the returned data by invoking the function with a MATLAB command such as:

```
>> my_tf = dsa_tf(my_freqs, my_amp)
```

To replot the transfer function data now stored in the new matrix 'my_tf', you can use the following plotting commands:

```
>> figure(3); clf
>> subplot(211)
>> semilogx(my_tf(:,1), 20*log10(my_tf(:,2)), 'r--')
>> ylabel('Gain (db)')
>> subplot(212)
>> semilogx(my_tf(:,1), my_tf(:,3), 'r--')
>> ylabel('Phase (degrees)')
```

Chapter 5

Results: Comparisons and Estimated Error

5.1 Comparison with a Commercial Dynamic Signal Analyzer.

Below are data for the low-pass RC circuit in Figure 5.2 with a breakpoint near 100 Hz.

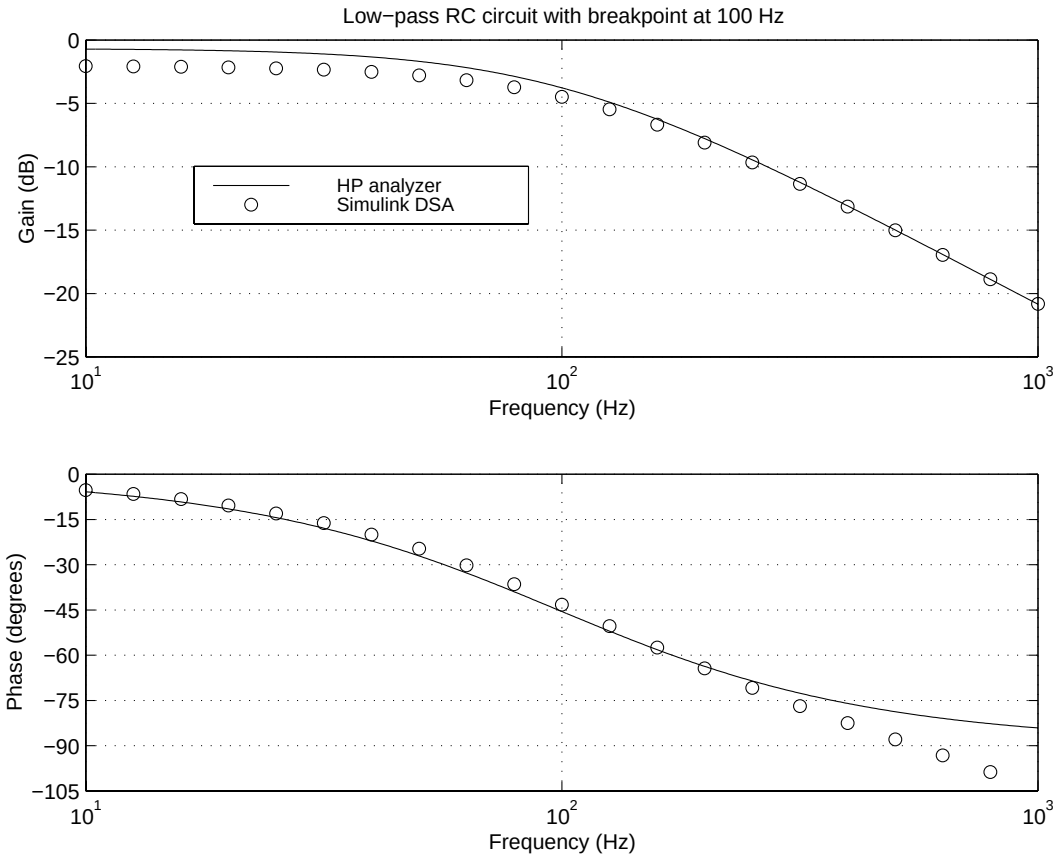


Figure 5.1: Low-Pass RC Cicuit Response

I tested the circuit both with the Simulink analyzer and with a commercial dynamic signal analyzer manufactured by Hewlitt-Packard. The solid lines in Figure 5.1 show the transfer function obtained by the HP machine, which is very close to the expected output. The Simulink data are shown as points.

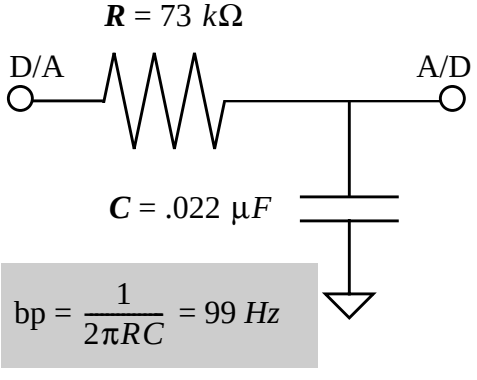


Figure 5.2: Low-pass RC Circuit

There are two primary discrepancies between the expected and measured transfer function. First, the gain values at low frequencies measured by the Simulink analyzer are approximately 2db lower than predicted. This occurs because the RC circuit resistance is high, compared to the resistance of a dSPACE A/D channel to ground, creating a voltage divider.

The more significant errors in the Simulink output occur in the calculations of phase at higher frequencies. Using a step size of 1e-4 seconds, the phase measurements of the Simulink analyzer lag the predicted value by about 18 degrees. This is a predictable result of the discrete sine wave output from the DSA.

Both discrepancies are described in more detail below. They are both real effects, due to the design of the dSPACE board and the sampling rate, which will affect models. The DSA is measuring the actual system response accurately. Users should note that systems implemented within the Simulink/dSPACE environment will differ from idealized, continuous system models, as described below, and that this may sometimes be a notable design concern.

5.2 Lower Than Expected Gain

The A/D channels on the dSPACE board have an effective resistance of about

$$R_{AD} = 300k\Omega$$

to ground. The fraction of the total voltage drop measured by the dSPACE A/D channel is:

$$V_{AD} = \left(\frac{R_{AD}}{R_{AD} + R_{RC}}\right)V_{total}$$

For R_{RC} less than a few $k\Omega$, the voltage measured by the AD channel will be within a percent or two of the predicted value. The RC circuit resistance used here was 73 $k\Omega$, however, creating a significant voltage divider. The predicted AD measurement would be $\frac{300}{300 + 73}$, or about 80% of the ideal. This corresponds to $20\log\left(\frac{300}{373}\right) = -1.9db$. The gain seen was about 2 db below the ideal value, so this seems to explain the discrepancy.

5.3 Phase Lag

Figure 5.3 shows the expected lag in output caused by using a discrete input signal. The plotted data come from a Simulink model of the same low-pass filter analyzed to generate Figure 5.1. When the input signal (Fig. 5.3 (a)) is continuous, the RC filter output (Fig. 5.3 (b)) lags the input by nearly 90 degrees of phase. With the discrete inputs, the additional output lag is one half the zero-order hold sampling rate. With a sampling period of 1.0e-4 seconds (10 points per 1kHz signal wave), this amounts to a delay of 0.5e-4 seconds, or:

$$\text{Additional Phase Lag} = 360° \left(\frac{0.5 \times 10^{-4}}{1.0 \times 10^{-3}} \right) = 18°$$

With a sampling period half as long (0.5e-4 seconds), this additional "error" lag is half as great. In Figure 5.3 (b), you can see the relative delays for each.

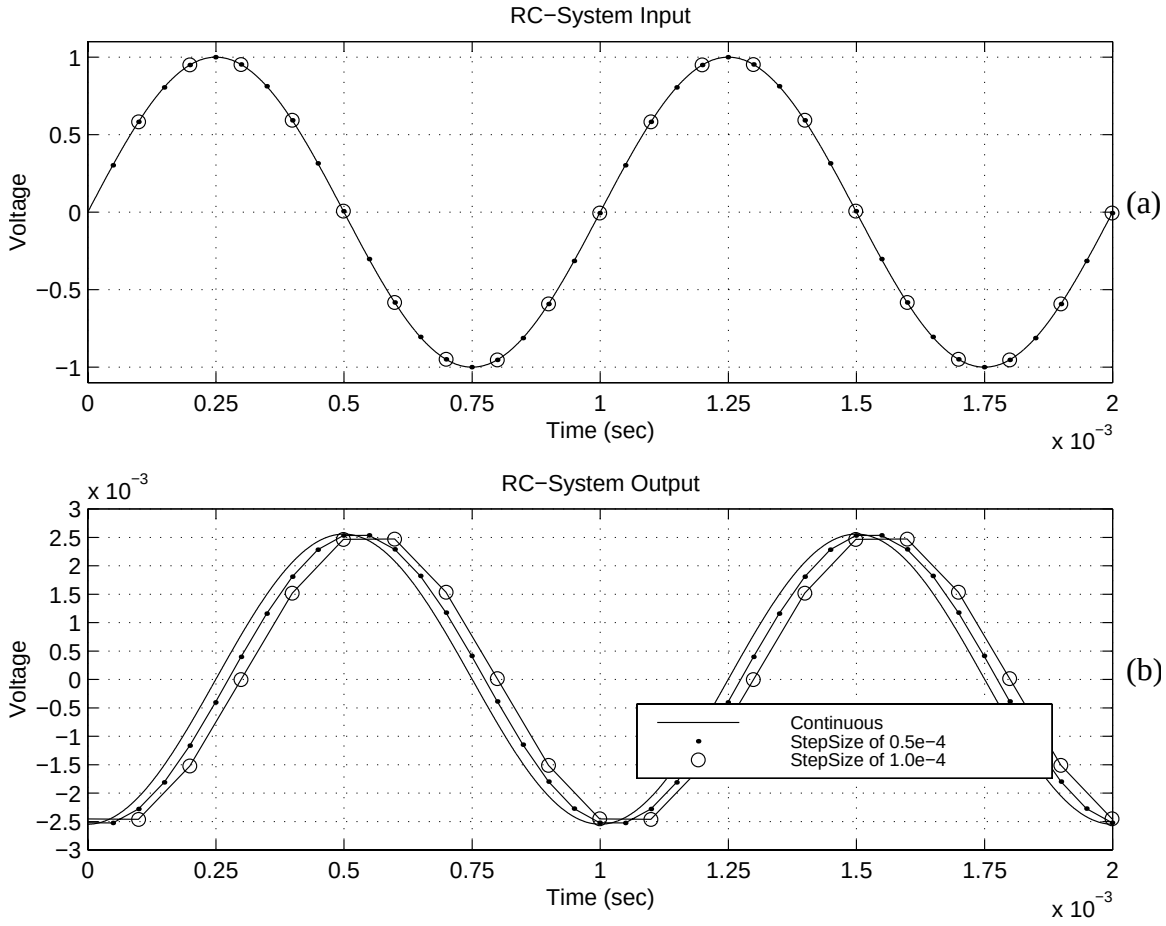


Figure 5.3: Discrete Sampling Lag with 1kHz Sine Wave

The general result is a "one-half sample delay" in output[1] This means the sampling-induced "error" is linearly related to the sampling period: if you can speed your sampling rate by a factor of two, you will cut the lag error by half.

The sampling rate capability of the dSPACE board depends on the task load of the entire model. For the DS1102, a rate of 20kHz is possible in a simple model. For the dynamic signal analyzer to provide reasonable results, I would suggest no fewer that ten points per wavelength. For a bandwidth of 1kHz, this means using a stepsize setting of no more than 1e-4 in the Simulink model.

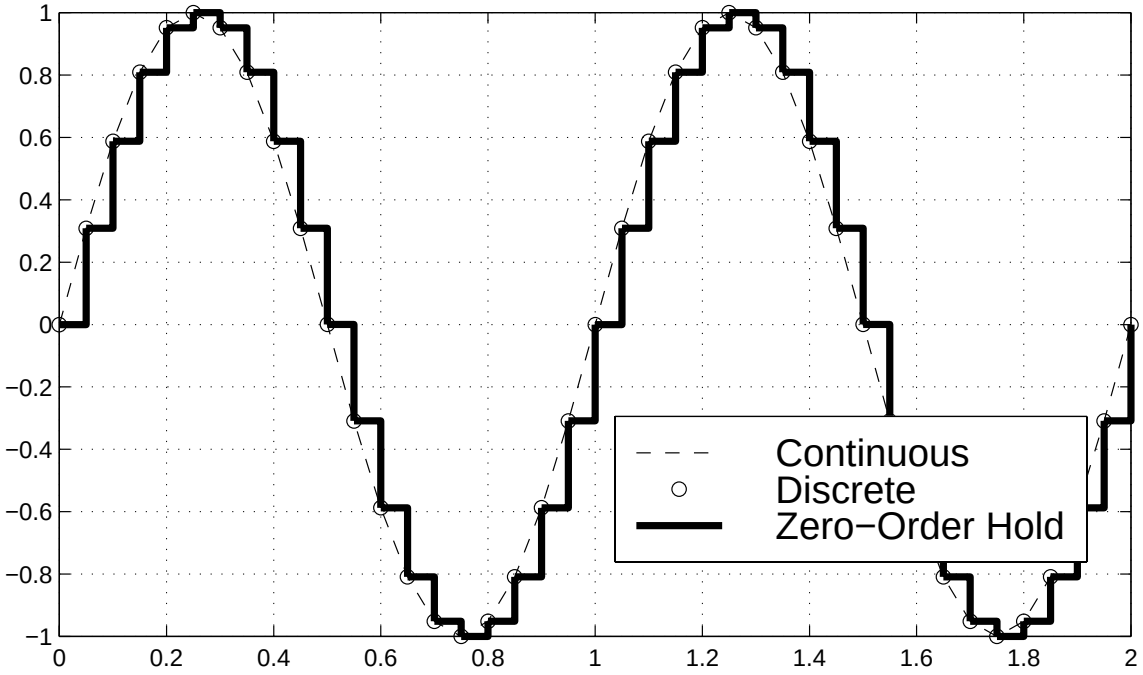


Figure 5.4: Zero-Order Hold Sampling

One way to quickly visualize the half-sample delay is to reconstruct part of the Fourier series needed to create the continuous equivalent of this staircase signal created by sampling. With a zero-order hold, the D/A channel continues to output the same value during the interval between updates. Figure 5.4 shows a zero-order hold signal. If we wish to rep-

1. See Oppenheim [9], pages 538-540. Refer to the rest of chapter 8 in "Signals and Systems" for discriptions of other sampling issues, as well.

resent this jagged path in the *frequency domain*, there is clearly a large component at the frequency of the pre-sampled, continuous signal (represented by the dashed line).

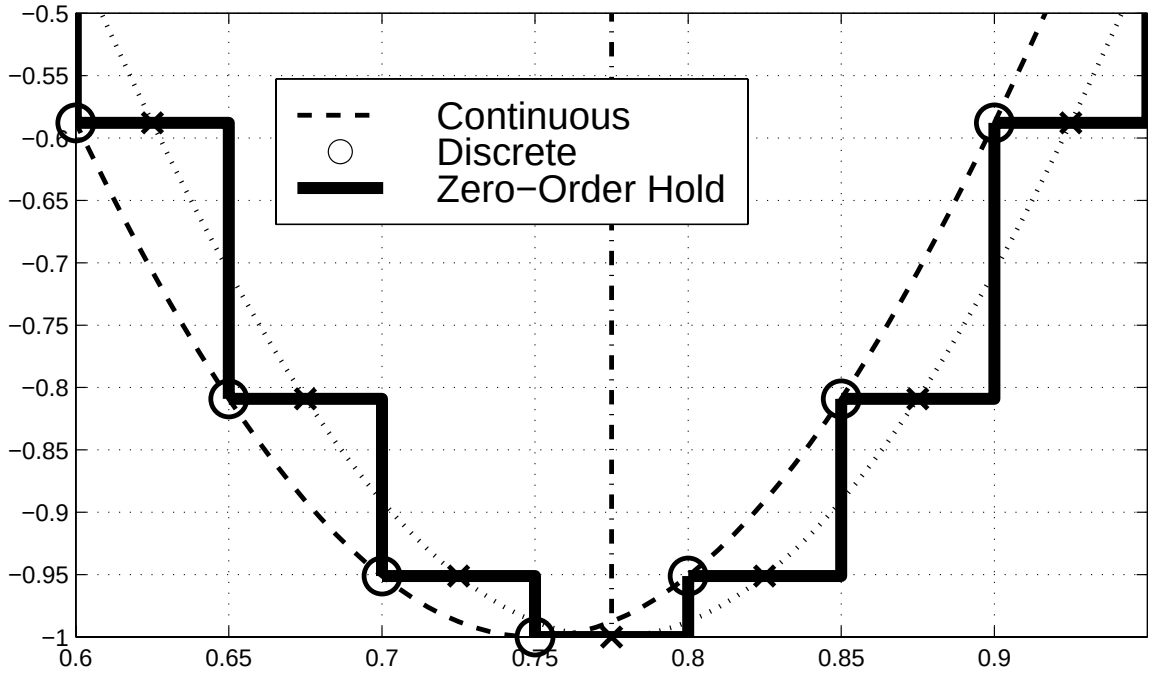


Figure 5.5: Half Sample Phase Shift

Figure 5.5 shows an enlarged section of the previous figure. The sampling interval here creates a stepping pattern that is symmetric about the vertical line at $x = 0.775$, as shown above. Because of the symmetry, it is clear the phase of the predominant harmonic component (i.e. the original, continuous frequency) is shifted by exactly half a sample width. For a particular sampling rate, however, the phase shift is still the same for any set of evenly spaced samples. In Figure 5.6, the steps do not have a vertical axis of symmetry, but a Fourier transform will generate exactly the same phase shift for that main frequency component. (It is simply easier to notice this initially in the symmetric case.)

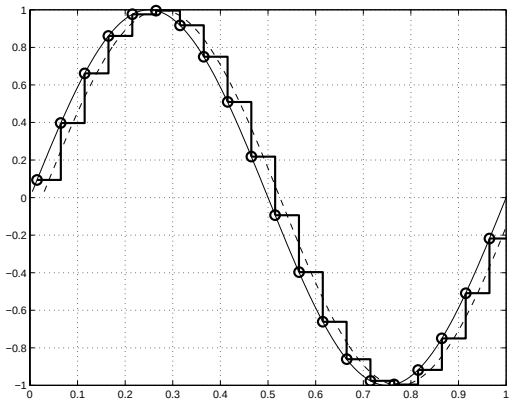


Figure 5.6: Asymmetric Zero-Order Hold

Chapter 6

Suggestions for Future Simulink/dSPACE Tools

6.1 Sampling Rate

As mentioned on page 17, the sampling rate should ideally be set to space the data equally and evenly along some number of complete cycles. To accomplish this would require resetting the sampling rate for each frequency. This probably requires resetting the model step size after the model is built and running. The DSA m-file function does not do this. Future refinements could include investigating whether mtrace and mlib functions *can* change the simulation step size in real-time and, if so, then implementing this in the m-file code.

6.2 Error Correction for High-Frequency Phase Calculations

As discussed in Section 5.1, the calculated phase will lag the actual system phase because the sine wave exciting the system is not continuous. It is certainly possible to predict the magnitude of this lag, thus correcting the phase calculation to some degree. Investigating the how to do this, calculating the extent of the expected improvement, and implementing the correction routine as part of the current dsa M-file could be beneficial.

6.3 Outside Sine Wave Source

The bandwidth of the DSA is limited by the model step size, which is typically about 1e-4 seconds for RTI models. For frequencies approaching 1 kHz, the discrete sine wave output by the analyzer hampers the accuracy of the analyzer (as mentioned above). To partially automate data acquisition above the current bandwidth, an outside source could generate the input sine wave. A discrete Fourier transform would distinguish the frequency of the outside source. This system would require more hands-on attention from

the user, particularly to insure the input values are reasonable.

The sampling rate would still be limited by the step size of the dSPACE board, and will in turn limit the maximum bandwidth. (Sampling theory requires more than two samples per cycle.) With some careful planning, such a Simulink tool might still provide a reasonably automated data acquisition system for higher frequencies.

Appendix A

The Simulink DSA Subsystem Block

This appendix describes some typical DSA configurations.

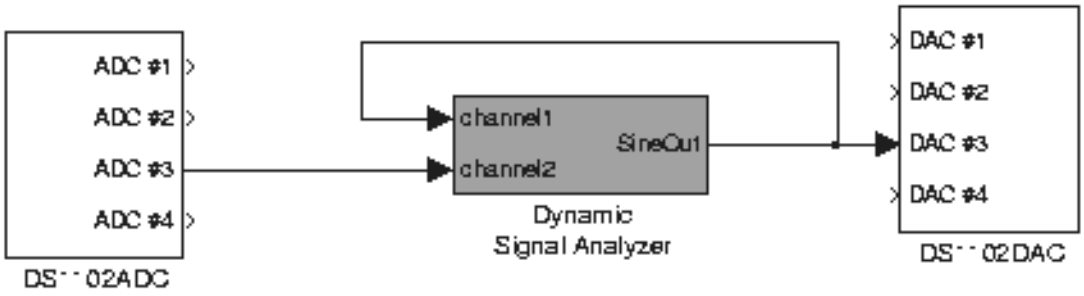


Figure A.1: Dynamic Signal Analyzer Measuring System Response

Measuring System Response

Figure A.1 shows the DSA subsystem in its simplest arrangement. Here, the analyzer outputs a discrete sine wave to a D/A channel and also uses this signal as input to 'channel1' of the DSA. Channel2 receives system response through an A/D channel. This configuration measures system response.

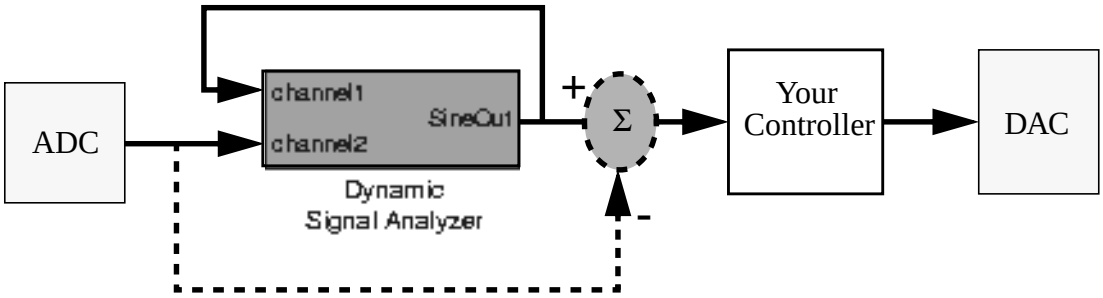


Figure A.2: Linking Additional Components with the DSA Block

Loop Transmission

To measure the loop transmission, insert the analyzer in a desired signal-input location. As in the previous example, feed SineOut into channel1 of the DSA. In general, there will be additional elements - for instance, some digital controller. The solid lines in Figure A.2 connect a DSA block to measure loop transmission.

Closed-Loop Response

You can measure closed-loop response by branching from the channel2 input to feed the signal into a summing block. The additional elements needed are drawn in dashed lines in Figure A.2.

Details About the Subsystem Block

The examples in this section show the SineOut signal being fed into channel1. This configuration is typical but not necessary. Channel1 and channel2 are used to obtain the transfer function between two, desired locations in a system. The sine wave input can go to a completely different location within the system.

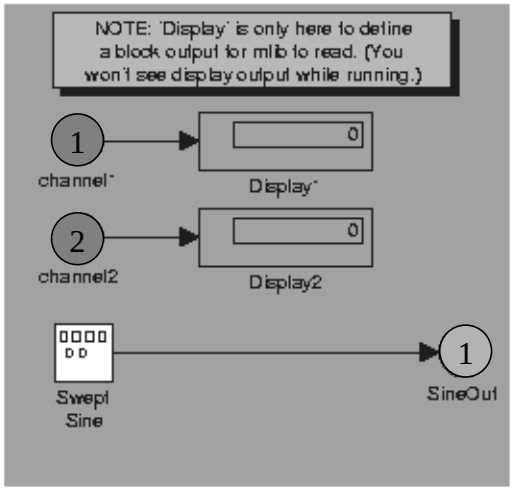


Figure A.3: Inside the DSA Subsystem Block

The DSA subsystem block consists of a swept sine source, sent to the output port, 'SineOut', and two input channels. The input channels connect to Simulink display sinks, as shown in Figure A.3. The display does not function when the dSPACE code runs on the board. When I left channel1 and channel2 unconnected to any Simulink blocks, however, their variable names did not seem to appear in the generated trace file. The displays exist to force the RTI to identify channel inputs, so they may be identified and sampled using mtrace functions.

Appendix B

MATLAB Source Code to Run the DSA

The m-file below runs the dynamic signal analyze block. The Simulink block and the MATLAB function are designed to run together. The DSA can not run without this (or a similar) MATLAB function. The Simulink DSA subsystem block defines particular variable names which the code below references directly. Either may be edited (and improved), but care should be exercised in doing so.

Figure B.1: MATLAB M-file Code: dsa_tf.m
(below)

```
function [mytf] = dsa_tf(w_list,amp_list,sval)
% function [mytf] = dsa_tf1(w_list,amp_list,sval)
% w_list : optional vector of INPUT FREQUENCY values at which to find TF
% amp_list : optional vector of INPUT AMPLITUDE values at which SINE
%          output will sweep.  (If amp_list is a scalar, the same
%          amplitude will be used at ALL frequencies...)
% sval : optional string value for the plot (e.g. 'r*' would plot
%          red *'s at points on figure(1)
%---------------------------------------------------------------------
% OUTPUTS: The output is an (N x 3) matrix. 'N' is the length of w_list.
%    Column 1:  Returned FREQUENCY list (w_list values)
%    Column 2:  GAIN, as the ratio of channel2/channel1 (not in db)
%    Column 3:  PHASE, in degrees
%---------------------------------------------------------------------
% NOTES:
%  (1) You must be running a SIMULINK model on the ds1102 board
%      for the MATLAB function dsa_tf() to work, and the simulink
%      model must include a special block called 'Dynamic Signal Analyzer'
%  (2) This program will overwrite figures 1 and 2 (in matlab)!!
%      Before running dsa_tf(), make sure you do not have images/plots
%      in either figure which should not be destroyed.

%---------------------------------------------------------------------
% Section 0: Define number of samples and default siggen amplitude.
N=10000;      % N: number of points to sample.
w_rad=0;      % indicates frequencies are NOT in rad/sec by default (Hz default)
dsa_A=.1;     % default AMPLITUDE of Swept Sine from DSA
exitval=0;    % program will exit if this is non-zero.
fprintf(1,'\nNote: Make sure your (DSA) model is built and running.\n');
```

```
fprintf(1,'       This program will erase any images or plots in figures 1 and
2.\n\n');
fprintf(1,'        If you would like to STOP this program now, enter ''q'' to
quit,\n');
isok=input('      otherwise, just hit enter to continue : ','s');
if strcmpi(isok,'q')
   fprintf(1,'\nOK, bye.\n')
else
   %-----------------------------------------------------------------------
   % Section 1: Get dSPACE board parameter addresses with mlib().
   mlib('SelectBoard','ds1102');
   mtrc31('SelectBoard','ds1102');
       amp_addr=mlib('GetSimAddr','P[Model  Root/Dynamic  Signal  Analyzer/Swept
Sine.Amplitude]');
      freq_addr=mlib('GetSimAddr','P[Model  Root/Dynamic  Signal  Analyzer/Swept
Sine.Frequency]');
   phi_last=0; B=1;
   %-----------------------------------------------------------------------
   % Section 2: Make sure all frequencies and amplitudes are set for this run:
   if ~exist('sval') % default symbol for Bode plot
      sval='bo';
   end
   if ~exist('w_list')
      w0=1:(1/10):3.0;
      w_use=10.^w0;
       fprintf('\nRunning %d points.\n  [Range = %.2f [Hz] (min) to %.2f [Hz]
(max)] \n',length(w_use),min(w_use),max(w_use));
   else
      is_rad=input('\nThe list of frequencies you entered was in:\n HERTZ (h) or
RAD/SEC (r) [default to Hertz]? ','s');
      if strcmpi(is_rad,'r')
         fprintf('\n--> using RAD/SEC\n');
         w_use=(1/(2*pi))*w_list;
         w_rad=1;
      elseif strcmpi(is_rad,'h')
         fprintf('\n--> using HERTZ\n');
         w_use=w_list;
      else
          fprintf('\n...hmmm, I don''t understand, so I''m going to default and
use HERTZ.\n');
         w_use=w_list;
      end
   end
   if ~exist('amp_list')
      A=mlib('Readf',amp_addr);
      if A<=0
         A=dsa_A;           % initial SINE WAVE amplitude
         fprintf('Using a DEFAULT amplitude of %5.3f\n',A);
         mlib('WriteF',amp_addr,A);
      else
         fprintf('Current model amplitude is %5.3f\n',A);
      end
      %amp_list=A+(0*w_use);
   else
```

```
      if length(amp_list)<length(w_use)
         if length(amp_list)==1
                  fprintf('Using %.4f as SINE WAVE amplitude as ALL frequen-
cies...\n',amp_list);
         else
              fprintf('WARNING: User amplitude and frequency vectors of unequal
length\n');
         end
         %amp_list=amp_list(1)+(0*w_use);
         A=amp_list(1);
      end
   end
   %-------------------------------------------------------------------------
   % Section 3: Begin outputting the frame for a TABLE to the matlab screen.
   fprintf('\n-------------------------------------- :: --------------------
--\n');
       fprintf('_________Frequency_________       _SineAmp__     ::     __GAIN___
_PHASE_\n');
     fprintf('     Hertz            [rad/sec]                         ::       db
Degrees\n');
   fprintf('-------------------------------------- :: ----------------------
\n');
   %-------------------------------------------------------------------------
   % Section 4: Run through all requested FREQUENCY values; find gain and phase
at each.
   f1=figure(1); set(f1,'Position',[465,220,450,520]);
   f2=figure(2); clf; set(f2,'Position',[10,220,450,520]);
   patch([-.5 1.5 1.5 -.5 -.5],[-.5 -.5 1.5 1.5 -.5],[0 0 0])
   patch([0 0 1 1 0],[0 1 1 0 0],[0 1 0])
   axis([-.5 1.5 -.5 1.5])
   t1=text(.5,.65,'COLLECTING'); set(t1,'HorizontalAlignment','center');
   t1=text(.5,.5,'FIRST'); set(t1,'HorizontalAlignment','center');
   t1=text(.5,.35,'DATA SET...'); set(t1,'HorizontalAlignment','center');
   axis off
   u1=uicontrol(2,'Position',[10 10 150 30],...
               'String','HIT to BREAK',...
               'Callback','stopval=1;',...
               'Enable','on','Value',5);

   i=1;
   while (i<=length(w_use)) & (exitval==0)
      w=w_use(i);      % frequency for this data set
      if exist('amp_list')
         if length(amp_list)>=i
            A=amp_list(i);   % NOTE: amplitude must be 'reasonable'...
            mlib('WriteF',amp_addr,A);
            % USER must visually check that OUTPUT is not saturated!!!
         end
      end
      A=mlib('ReadF',amp_addr);
      fprintf('%9.4f     [%10.4f]    %9.6f   ::  ',w,2*pi*w,A);
      drawnow;
      mlib('WriteF',freq_addr,w); % mlib('WriteF',amp_addr,A);
      tmax=(150/w)+2;
```

```
      if i==1
         hwait=waitbar(0,'Please wait...');
      end
      tic                                % give system some time to settle...
      while (toc<(150/w)) & (get(u1,'Value')>1)
         drawnow
         if exist('hwait')
            waitbar(toc/tmax)
         end
      end
      %----------------------------------------------------------------------
      % Section 4-1: Here is the actual procedure to get gain and phase at
      % a particular frequency.  This would be more elegantly implemented
      % as a separate function.  Instead, it is included within this loop
      % so that the dynamic signal analyzer can be run from a SINGLE FILE.
      % (Otherwise, the user has to worry about having all files needed.)
      %k=[0:(N-1)]';
      A=mlib('Readf',amp_addr); w=mlib('Readf',freq_addr); T=1/w;
      %mtrc31('SelectBoard','ds1102');
           y_addr=mtrc31('GetAddr','rti  B[Model  Root/Dynamic  Signal  Analyzer/
channel1]',...
         'rti B[Model Root/Dynamic Signal Analyzer/channel2]',...
         'rti B[Model Root/Dynamic Signal Analyzer/Swept Sine]');
      mtrc31('TraceVars',y_addr);
      samp_per=mlib('GetSimAddr','Task Info/Timer Task 1/sampleTime');
      dt=mlib('ReadF',samp_per);
      ncyc=floor(w*dt*N);          % Use an INTEGRAL number of sine waves!!
      Nlast=round(ncyc/(w*dt));
     if ncyc>10                  % Display no more than this number of sine waves
         Nplot=round(10/(w*dt));   % for the user to observe during the run...
      else
         Nplot=Nlast;
      end
      tic
      while (toc<2) & (get(u1,'Value')>1) % settle time pause...
         drawnow
         if exist('hwait')
            waitbar((tmax-2+toc)/tmax)
         end
      end
      if exist('hwait')
         close(hwait)
         clear hwait
      end
      if (get(u1,'Value')>1)
         % !@!!! set frame in desired way!!!
         mtrc31('SetFrame',[],1,0,N);
         %mtrc31('SetFrame',dt,1,0,dt*(N));
         mtrc31('SetTrigger',y_addr(3,:),0,1);
         mtrc31('LockProgram');
         mtrc31('StartCapture');
         while mtrc31('CaptureState')~=0
            drawnow;
         end
```

```
      my_data=mtrc31('FetchData');
      % Take an INTEGRAL number of sine waves, total:
      k=[0:(Nlast-1)]';
      ncyc=floor(w*dt*Nlast);
      y_out=my_data(2,1:Nlast);
      y_in=my_data(1,1:Nlast);
      t_out=dt*[1:Nlast];
      mtrc31('UnlockProgram');
   end
   set(u1,'Enable','off');  % do not allow a break until new data presented
   if (get(u1,'Value')<1)
      i=max(1,(i-1));
      w=w_use(i);      % frequency for this data set
      if exist('amp_list')
         if length(amp_list)>=i
            A=amp_list(i);   % NOTE: amplitude must be 'reasonable'...
            mlib('WriteF',amp_addr,A);
         end
      end
      % USER must visually check that OUTPUT is not saturated!!!
      fprintf(1,'\nBREAK: going back to previous frequency.\n');
      fprintf(1,' * Use COCKPIT to reset Amplitude.\n');
      fprintf(1,' * USE TRACE to view resulting sine waves.\n');
      fprintf(1,'Restart DSA by hitting RESTART button in figure 2.\n');
      %fprintf('%9.4f     [%10.4f]    %9.6f   ::  ',w,2*pi*w,A);
      drawnow;
      % Go back and RESET freq and amp to last values, in case
      % user decides to view and change, using TRACE and COCKPIT
      mlib('WriteF',freq_addr,w); %mlib('WriteF',amp_addr,A);
      startval=0; exitval=0;
      subplot(3,1,3); cla;
      patch([0 30 30 0 0],[2 2 7 7 2],[0 1 0]);
      axis off; axis([0 30 2 7]);
      text(.5,6,'You can use COCKPIT and TRACE to reset');
      my_str=num2str(w_use(i),'%.1f');
      text(.5,5,['Amplitude and then CONTINUE at ' my_str ' Hz...']);
      text(.5,4,'...or you can EXIT to completely rerun DSA.');
      text(.5,3,'[choose EXIT or CONTINUE]');
      set(u1,'String','HIT to EXIT','Callback','exitval=1;');
      u2=uicontrol('Position',[200 10 150 30],'Value',5,...
         'String','CONTINUE','Callback','startval=1;');
      set(u1,'Enable','on','Value',5);
      while (get(u1,'Value')~=0) & (get(u2,'Value')~=0)
         drawnow;  % wait for user to hit a button...
      end
      if get(u1,'Value')==0
         exitval=1;     % exit the dsa
      else
         startval=1;    % restart the dsa
         A=mlib('ReadF',amp_addr);
         if exist('amp_list')
            if length(amp_list)>=i
                 if A~=amp_list(i)   % USER HAS RESET AMPLITUDE! ignor future
presets
```

```
                  clear amp_list;  % amp_list no longer exists for this run.
               end
            end
         end
         figure(2); clf;
         patch([-.5 1.5 1.5 -.5 -.5],[-.5 -.5 1.5 1.5 -.5],[0 0 0])
         patch([0 0 1 1 0],[0 1 1 0 0],[0 1 0])
         axis([-.5 1.5 -.5 1.5])
        t1=text(.5,.65,'COLLECTING'); set(t1,'HorizontalAlignment','center');
           t1=text(.5,.5,[my_str ' Hz']); set(t1,'HorizontalAlignment','cen-
ter');
           t1=text(.5,.35,'DATA SET...'); set(t1,'HorizontalAlignment','cen-
ter');
         axis off
         u1=uicontrol(2,'Position',[10 10 150 30],...
            'String','HIT to BREAK',...
            'Callback','stopval=1;',...
            'Enable','on','Value',5);


      end
   else
      % user did NOT request a break, so analyze this data set:
      if exist('y_out')
              if (max(y_out)<0.95) & (min(y_out)>-.95) & (max(y_in)<.95) &
(min(y_out)>-.95)
          warncolor=[1 1 0];                    % Amplitude not 'saturated'.
OK to use this data.
         else
            warncolor=[1 0 0];
         end
         %         y_sin=(dt*(w*2*pi)*(k))';
         y_sin=((w*2*pi)*t_out);       % Ideal SINE at this freq
         Bc=(2/(length(y_out)))*sum((y_out).*cos(y_sin));
         Bs=(2/(length(y_out)))*sum((y_out).*sin(y_sin));
         B_=sqrt(Bc^2+Bs^2);           % Amplitude of OUTPUT at this freq
         Ac=(2/(length(y_out)))*sum((y_in).*cos(y_sin));
         As=(2/(length(y_out)))*sum((y_in).*sin(y_sin));
        A_=sqrt(Ac^2+As^2);          % Amplitude of INPUT at this freq (check)
         Gain=B_/A_;
         phi_out=atan2(Bc,Bs);
         phi_in=atan2(Ac,As);
         phi=phi_out-phi_in;      % Difference in phase (input->output) (rad)
         figure(2); clf
         subplot(311)
         plot(t_out(1:Nplot),y_out(1:Nplot),'r.');
         hold on; grid on; title('RED: channel2'); %xlabel('seconds')
         subplot(312)
         plot(t_out(1:Nplot),y_in(1:Nplot),'b.');
         hold on; grid on; title('BLUE: channel1 (input)'); xlabel('seconds')
         subplot(313)
         patch([0 30 30 0 0],[2 2 7 7 2],warncolor);
         axis off; axis([0 30 2 7]);
         text(.5,6,'Check that the OUTPUTS above are :');
```

```
            text(1.5,5,'* NOT SATURATED.');
            text(1.5,4,'* Not buried in noise.');
            text(.5,3,'otherwise, BREAK and run with new AMPLITUDEs.');
            u1=uicontrol(2,'Position',[10 10 150 30],...
               'String','HIT to BREAK',...
               'Callback','stopval=1;',...
               'Enable','on', 'Value',5);
            stopval=0;   % insure 'stopval' is reset to show 'OK' state
            %else
            %   A=A*3/4;   % REDUCE INPUT AMPLITUDE AND RETAKE DATA!
               %    fprintf('Reducing input amplitude from %5.3f volts to %5.3f
volts...\n',(4/3)*A,A);
            %   mlib('SelectBoard','ds1102');
            %   mlib('WriteF',amp_addr,A);
            %end
            %end
         end
         if phi>phi_last
            while (phi-phi_last)>(pi)
               phi=phi-(2*pi);
            end
         else
            while (phi-phi_last)<(-pi)
               phi=phi+(2*pi);
            end
         end

         mytf(i,:)=[w Gain (180/pi)*phi];
         fprintf(1,'%8.3f     %9.2f\n',20*log10(Gain),(180/pi)*phi);
         figure(1)
         subplot(2,1,1); semilogx(mytf(i,1),20*log10(mytf(i,2)),sval); hold on;
         semilogx(mytf(1:i,1),20*log10(mytf(1:i,2)),'-'); axis auto; grid on;
         xlabel('Frequency (Hertz)');
         ylabel('Gain (db)'); title('Transfer Function (channel2/channel1)');
         subplot(2,1,2); semilogx(mytf(i,1),(i,3),sval);hold on;
         semilogx(mytf(1:i,1),mytf(1:i,3),'-'); axis auto; grid on;
         xlabel('Frequency (Hertz)');
         ylabel('Phase (Degrees)');
         %if ((180/pi)*(max(mytf(1:i,3))-min(mytf(1:i,3))))>80
                                    %            set(gca,'YTick',[45*floor((180/
pi)*min(mytf(1:i,3))):45:45*ceil((180/pi)*max(mytf(1:i,3)))]);
                %elseif  (((180/pi)*(max(mytf(1:i,3))-min(mytf(1:i,3))))>40)  &
(get(gca,'YTickMode')=='manual')
                                    %            set(gca,'YTick',[15*floor((180/
pi)*min(mytf(1:i,3))):15:15*ceil((180/pi)*max(mytf(1:i,3)))]);
         %end
         drawnow
         phi_last=phi;
         figure(2)  % PUT FIGURE 2 ON TOP TO FORCE USER TO LOOK AT SINE WAVE DATA
         i=i+1;
      end
      %if startval==0
      %   i=max(1,(i-1));
      %end
```

```
   end
   if exitval==0
      if w_rad==1
         fprintf(1,'\nThe TF created has 3 columns:  Frequency (rad/sec), Magni-
tude (absolute), Phase (degrees)\n')
         mytf(:,1)=(2*pi)*mytf(:,1);
      else
          fprintf(1,'\nThe TF created has 3 columns:  Frequency (Hz), Magnitude
(absolute), Phase (degrees)\n')
      end
                 fprintf(1,'To   replot   GAIN   in   Bode   format:      semi-
logx(tf(:,1),20*log10(tf(:,2))\n');
     fprintf(1,'To replot PHASE in Bode format: semilogx(tf(:,1),tf(:,3)\n');
   else
      if w_rad==1
         mytf(:,1)=(2*pi)*mytf(:,1);
      end
      figure(2)
      subplot(313)
      cla
      patch([0 30 30 0 0],[2 2 7 7 2],[0 0 .7]);
      axis off; axis([0 30 2 7]);
      t1=text(1,4.5,'OK! Program has stopped, bye.');
      set(u1,'Enable','off'); set(u2,'Enable','off');
      set(t1,'Color',[1 1 .3]);
      set(t1,'FontSize',18);
      set(t1,'FontName','Times');
      fprintf(1,'\n\n   ************************************************\n');
      fprintf(1,    '   *** Program exited by user during run.... bye. ***\n');
     fprintf(1,    '   ************************************************\n\n');
   end  % ends user query if 'ok' to continue
   mlib('WriteF',freq_addr,w_use(1));
   mlib('WriteF',amp_addr,0);

end
return
function y = setstop()
y=1;
return
```

Appendix C

TRACE and COCKPIT

TRACE and COCKPIT are programs which access data on the dSPACE board. TRACE plots sampled streams of data versus time. COCKPIT can display and/or change values of system parameters and outputs. Both programs operate while dSPACE code is running, making them useful for analyzing and adjusting models. This section provides a brief tutorial on each. These are intended as a guides for getting started; users should refer to the dSPACE manuals for TRACE and COCKPIT for more detailed information.

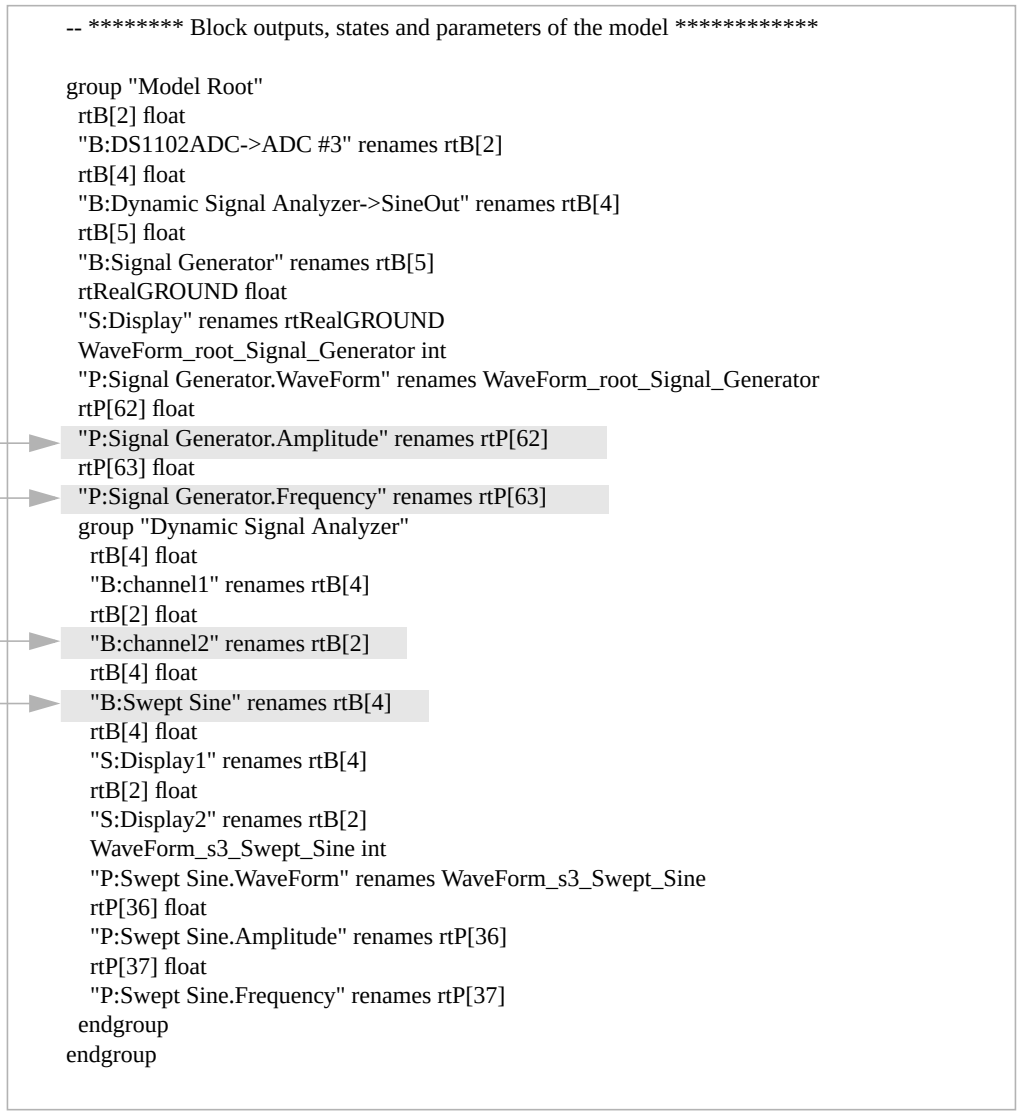


Figure C.1: Part of a .trc File

C.1 The Trace File

The dSPACE RTI will automatically create a trace file during a Simulink build. This is essentially a map, identifying variable names from the Simulink model with their counterparts in the newly-created C code. Figure C.1 shows a section of the trace file **rc_test1.trc**, created for the a simulink model named **rc_test1.mdl**. Both COCKPIT and TRACE load parameters from a '.trc' file. For either application, each time a model is rebuilt, the new trace file must be reloaded.

The trace file information can also be used to identify variable names for mlib and mtrace functions; **trcview** can provide the correct syntax. (See Appendix D.)

C.2 Using TRACE

The guide below uses the Simulink demo model, **dsa_demo.mdl**. Refer to the User's Guide (Chapter 4) for a model description. The Dynamic Signal Analyzer subsystem block measures the response of a single lead, zero-pole system. The same basic steps can be used to view data from other Simulink models.

1. Build a Simulink model. Preset parameter settings for dsa_demo.mdl should be OK.

2. Begin TRACE. Use the desktop icon, if available, or start the program 'Trace' using the Windows Finder (within the Start button options at the lower left screen corner)**.**

3. Load the '.trc' file. Use the mouse to select **'File:Load Trace File...'** Select the appropriate directory path at the prompt **'Look in...'**. The graphical interface automatically filters file names to present only trace files; select the one corresponding to your current model. It will have the same base name as the model, for instance 'dsa_demo.trc' for the example used here.

4. Select desired plotting variables. Trace will present a tree structure representing the

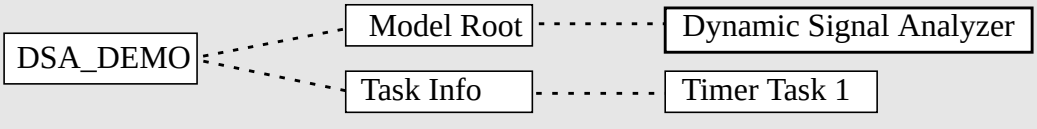


Figure C.2: System Tree Showing Trace File Groups

model substructures. Scroll to the right side of the tree and click on the group 'Dynamic Signal Analyzer'. A list of parameter names and block outputs will appear to the right of the tree. Select the first two to view **channel1** and **channel2**, the signals the DSA subsystem block compares to calculate gain and phase at each frequency. Also choose the **Swept Sine** output - since it is fed into **channel1**, these two signals should look identical. TRACE will open a second window with subplot spaces for each signal selected.

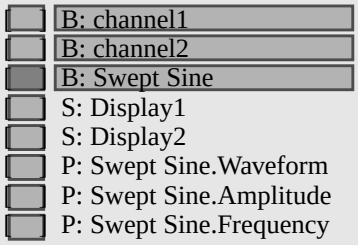


Figure C.3: Trace Output List

5. Highlight the box to the left of Swept Sine. This selects the sine wave generator as the trigger for TRACE data collection. Using this source signal insures there will be a zero crossing on which to trigger. In general, the channel signals can have significant offsets, preventing a zero crossing. For the dsa_demo model, any of the three, selected signals will work as triggers, but the **Swept Sine** is the safe choice for typical DSA models.

6. Select an appropriate length of time for each TRACE display of data. The Length input box is located just below the model tree shown in Figure C.2. For a 10 Hertz input signal, a length of 0.1 seconds will fit one wavelength.

7. Hit the START button at the upper left of the screen. The second window should now display your requested signal outputs. Click AUTO in the plot window to enable continual retriggering. Toggle **START/STOP** to freeze and then capture new data.

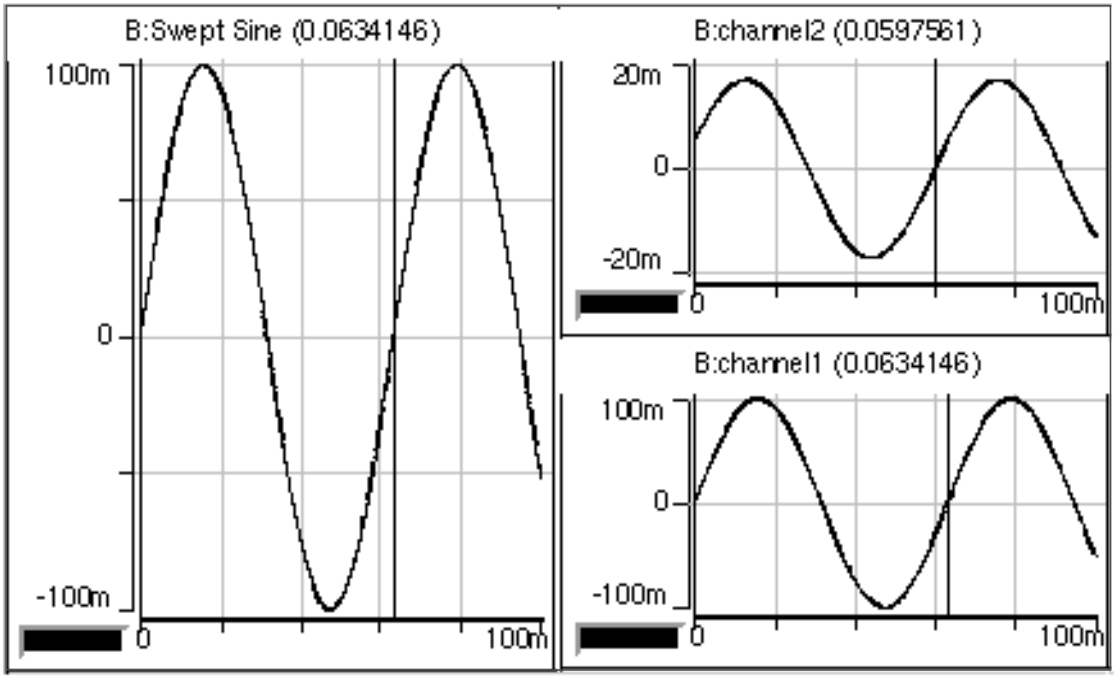


Figure C.4: Sample TRACE Output

8. Set desired Options. Within the Options menu, you can turn on and off grid lines, or select cross hairs to measure 'delta' distances on a given plot. Explore on your own and refer to the TRACE manual to learn more detailed information.

9. Select File:Print to make a hard copy of the traces. Figure C.4 shows a typical printer output. Here, the dark, vertical lines are cross hairs. They were placed manually, to measure the x-axis distance (in time) from the origin (0,0) to the next rising zero crossing. This is displayed in parentheses at the top of each subplot. We can see the frequency here is roughly (1/(0.063)) Hertz: about 15.9 Hz; and that channel2 leads channel1 somewhat. TRACE can provide more detailed and precise information, too.

10. Use File:Save to store data for later processing, if you wish to analyze captured data more directly with MATLAB. You can also save your TRACE set-up.

C.3 Using COCKPIT

COCKPIT allows you to display outputs and modify variables while a Simulink model is running. You should find some of the program interface similar to TRACE.

We will continue here with the same model, **dsa_demo.mdl**, used to introduce TRACE on page 54. The steps below describe how to create displays and controls for the system. As with TRACE, you will need to have this model running.

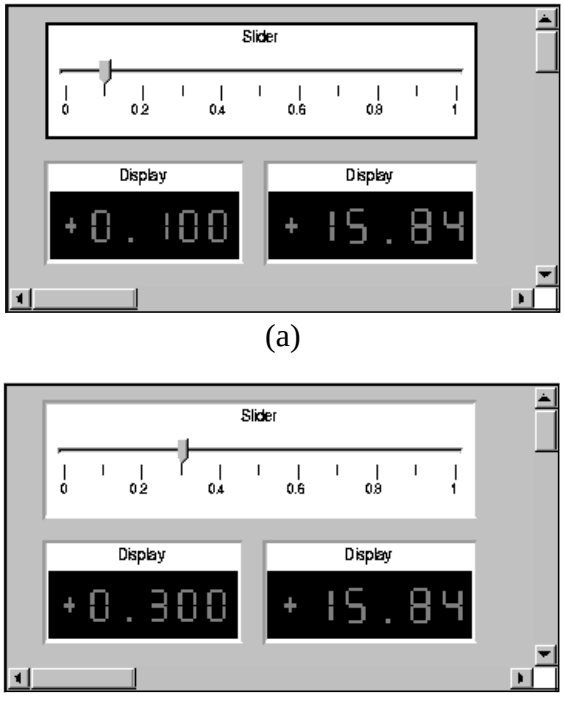


Figure C.5: COCKPIT Display for Amplitude Change

1. Build the Simulink model.

2. Begin COCKPIT using the desktop icon (or Windows Finder program).

3. File: Load Trace File... and select the appropriate '.trc' file

4. Go to the Control menu at the top of the COCKPIT window and select:
Input/Output Control -> Slider

5. Click-and-Drag the mouse within the COCKPIT window to create a 'Slider', as shown in Figure C.5. When you do this, make sure you *hold down* the left mouse button as you stretch the rubber band box outlining the slider shape. You can create multiple controls and displays in the window using this basic procedure.

6. Double-click on the new Slider control. COCKPIT will display a system tree similar to the one shown in Figure C.2.

7. Select a system parameter from the tree. For this example, we will select

P: Swept Sine.Amplitude

You can change the scale limits for the Slider to range from 0 to +1.0. The fields allowing this should be in the upper right of the window which displayed the system tree.

8. Go to the Control menu again, and now select:

Output Controls -> Display

Once again, click-and-drag to create the Display, as you did in step 5. Repeat steps 6 and 7 to select 'P: Swept Sine. Amplitude' again. The Slider will allow you to change the sine wave amplitude, and the Display will show its (changing) value. In Figure C.5, there is also a Display which outputs the parameter:

P: Swept Sine.Frequency

You can create this Display, as well, or experiment with other devices in the Control menu. *Note that you can* ***set the format*** of the Display using syntax similar to that for C-language numbers. [For instance %6.3f would display 6-digit floating point numbers with 3 digits to the right of the decimal place.] The field for this input will appear near the same location you found the scale limits in step 7.

9. Press the START button to make your Control and Display(s) active.

10. Hold down the left mouse button on the Slider knob and move it to change the amplitude of the sine wave. You should see the value change in the amplitude Display. You can rename each Control or Display, resize them and move them. Reset the amplitude to some new value, for instance '0.3'.

11. Select Animation: Stop where the START button used to be to make the COCKPIT devices inactive.

12. File:Print to get a hard copy of the COCKPIT window (if you wish).

13. File: Save to save this COCKPIT set-up. You can then reopen the same configuration without having to recreate all of the devices.

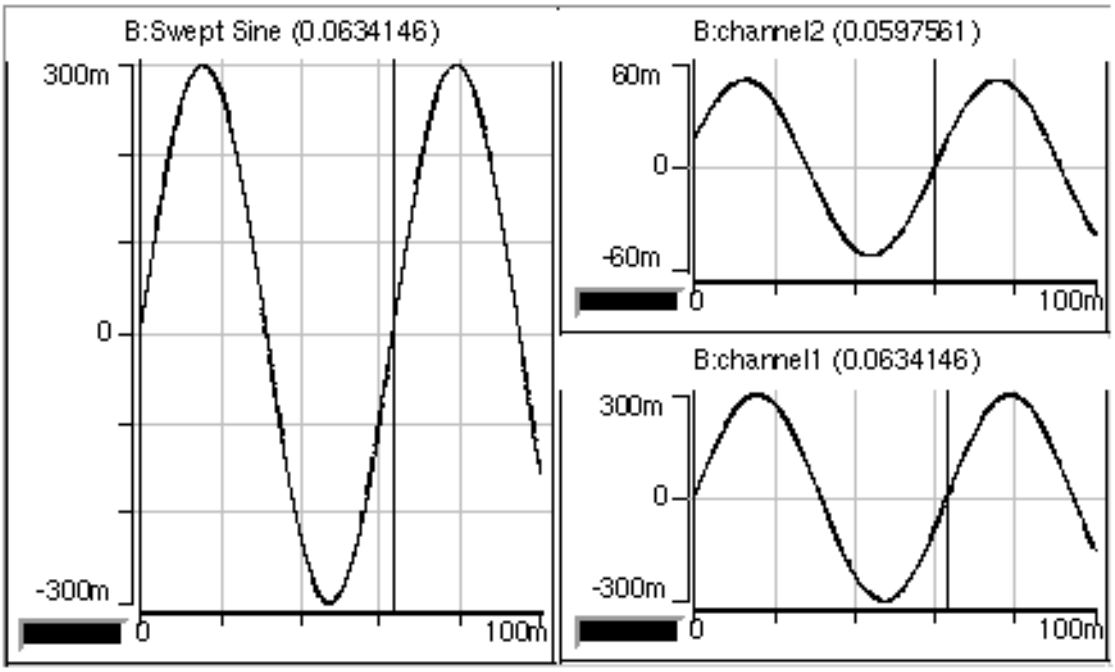


Figure C.6: TRACE Output After Amplitude Change

Now that you have changed the amplitude of the sine wave output from the dynamic signal analyzer, you can return to TRACE and view the outputs of the Simulink model again. Figure C.6 shows TRACE output for a 'Swept Sine.Amplitude' set to '0.3'. You can compare this with Figure C.4. The frequency and phase of each output are the same; the amplitudes are now three times their previous values.

You can refer to the manuals for both COCKPIT and TRACE for more information on their use.

Appendix D

Mlib, Mtrace, and Trcview

Real-time data acquisition and on-the-fly parameter modification can greatly facilitate DSP design. COCKPIT and TRACE do just this. Mlib and mtrace are MATLAB functions which can be incorporated in script files, to access the same information more directly. Mlib reads from and writes to variable addresses on the dSPACE board asynchronously, while mtrace can downsample data for real-time analysis. M-files are relatively quick and easy to write, compared with equivalent C-code, making these two functions convenient and complementary tools for design in the Simulink/dSPACE environment.

Below are brief tutorials for mlib and mtrace. Also included is a description of the MATLAB tool trcview, which indentifies the proper mlib/mtrace syntax for variable names.

D.1 Mlib Tutorial

dSPACE Demos

We'll start with the example programs described in the mlib user's guide [8], pp.35-40. The code for this is already on the lab computers. I would suggest copying the entire directory of examples into your own (z:) directory, so you can play with it:

1. Go to c:\Dsp_cit\Demo1102
2. Select the 'mlib' folder.
3. Copy it (ctr-c), and **Paste it (ctr-v) into your personal directory**
4. Start MATLAB and **'cd' to your new, personal 'mlib' directory**
5. At the MATLAB prompt, type:'**rtilib**' and then **open the simulink model 'smd.mdl'**

This a basic, Spring-Mass-Damper system. We don't need any connections (I/O via nerdkit, etc) to build and run the model on the dSPACE board. Let's run the model:

6. In the Simulink model (smd), **select 'Tools:RTW Build'** (or ctr-B)

C-code is now generated and downloaded onto the DS1102.

Note that the files generated during a build with extensions '.trc' and '.map' have information about the variables. Specifically, 'smd.trc' now provides a roadmap which mlib (or applications like TRACE and COCKPIT) will use to map the model names for variables to their c-code equivalents. You can open 'smd.trc' to look at the general format. The 'Model Root' variables will remind you of TRACE and COCKPIT. The naming convection used labels Block Outputs ('B:'), States ('S:') and Parameters ('P:') [with their respective names from the Simulink model]. You can find the appropriate mlib and mtrace syntax for Simulink variable names using the program trcview; load the associated '.trc' file and click on the groups and variables for which you need name.

The spring-mass-damper model should now be running on the DS1102.

7. Start up TRACE. Select **'File: Load Trace File'** ('smd.trc' should now exist, from the Build).

8. In the 'TRACE Control Panel': **Click on 'Model Root'** in the main window and **select 'B:Spring-Mass-Damper System'** and **'B:Signal Generator'** to view system response and input, respectively.

9. In the 'Trace Plots' template: **Click 'Auto'** and **'Start'.**

You should see a square wave from the generator and the 2nd order response of the system. I would suggest fixing the y-axis (since we're about to play with the system characteristics). Select 'Options: Scaling of Axes...' (Shift-S). Select 'Scaling Mode Y-Axis: Fixed (manual)', and set the new limits to max=0.2 and min=-0.2. Repeat for the other trace plot (signal generator vs system plot).

Let's look at some of the m-files in your new mlib directory. (As mentioned, all the examples are documented in the mlib guide.)

siggen()

10. At the MATLAB prompt, just type:

>> **siggen**

Without arguments, this returns the current 'Signal Generator' settings:

Usage: siggen(peak,period)

Signal generator parameters: Peak=0.1, Period=0.02 sec

11. Now, change the amplitude to 0.05 and period to 0.01. **Enter:**

>> **siggen(.05,.01)**

You can see the result in TRACE.

12. Open the m-file 'siggen.m' to look at the script file. Here we see 4 very basic mlib functions. If you eventually use mlib in your own m-file scripts, you will probably include lines similar to the ones presented below. For more information on these and all other mlib functions, refer to the mlib user's guide[8]. (Section 4, pp. 7-34):

1. 'SelectBoard' : You'll always select the 'ds1102'. By the way, you can check to see what board(s) we have installed on the lab computers by typing your first direct 'mlib' call at the MATLAB prompt:

>> **mlib('GetBoard',0)**

MATLAB will return 'ds1102' as the dSPACE board with index '0'. Try "mlib('GetBoard',1)", and MATLAB will tell you there's no board installed with this index. (We just have the DS1102.) 'SelectBoard' may not seem critical for us, (there being only one board!) but it is necessary some board is defined. When you write your own scripts using mlib, you should include this line near the top, which will always read:

```
mlib('SelectBoard','DS1102');
```

2. 'GetSimAddr' : Find the address of a Simulink variable. (Compare with 'GetVarAddr', used for global variables.) The syntax for naming Simulink model variables is similar to the '.trc' and TRACE formats for displaying them. Here are examples from 'siggen.m':

```
peak_addr = mlib('GetSimAddr','P[Model Root/Signal Generator.Amplitude]');

freq_addr = mlib('GetSimAddr','P[Model Root/Signal Generator.Frequency]');
```

'P' identifies a 'Parameter'. Our smd signal generator is called 'Signal Generator'. (Note that *white space* is important in these names.) Look at 'smd.trc' again and you can find "P:Signal Generator.Amplitude" renames rtP[4] listed under group "Model Root" You can see the basic naming convention from the mlib() calls shown. (See pages 14-16)

3. ’ReadF’ : Get the value of the floating point variable at this address. For instance,

```
peak = mlib(’ReadF’,peak_addr);
freq = mlib(’ReadF’,freq_addr);
```

4. ’WriteF’ : Change the value of the floating point variable at this address:

```
mlib(’WriteF’,freq_addr,1/period);
mlib(’WriteF’,peak_addr,peak);
```

You can read and write integers and doubles, too. (e.g. ’ReadI’ to read an integer. See mlib manual for more details, of course, but essentially you are using mlib to read and write variable from the ds1102 while the model is running.)

Note: When you use mlib to access variables on the DSP board, the board actually ’halts’ very briefly while the PC reads or writes data from it. (’Very briefly’ means only 300 ns per (32 bit) word!) See the mlib manual for more details, if you are interested (section 5.2 ’MLIB Programming Details’, on p.41 of the mlib user’s guide).

ch_damp()

Have your TRACE window active and handy (so you can watch the system running), and then:

13. Type at the MATLAB prompt:

```
>> ch_damp
```

You can just hit ’return’ two more times after entering this, in response to the ’?’ prompts.

You should see the TRACE output changing and ch_damp() modifies the damping ratio. The MATLAB window should be updating you at each step. Set the ’siggen’ values as you prefer and play around more with ’ch_damp’, if you like. You can open up the M-file ’ch_damp.m’ to see how it works, too (as we did with siggen.m).

Other example files:

There are other M-files in your new mlib directory which you might examine to get more ideas on how to use mlib() for your own script files.

14. Check out 'plotuy.m', for instance.

The time units for plotuy are not evenly spaced. The mlib function just grabs a datum asynchronously and goes on. *To get the data at the rate of sampling* (which can be set in the model under 'Simulation:Parameters' before a Build of the model), *you can capture a set of variable values using mtrc31, called '**mtrace**'*. (See the mtrace user's guide for specifics. The actual function you will use within MATLAB is called 'mtrc31'. 'Mtrace' and 'mtrc31' are used interchangeably in this document; they are the same.)

D.2 Mtrace Tutorial

Mtrace allows you to capture data of system outputs at fixed time steps. You can specify which variables to collect and set 'triggering' options for the step size and number of data points. The resulting vector(s) of data can be plotted and manipulated in the MATLAB environment. This data can also be collected using the program TRACE, by saving captured data. For repetitive operations, mtrc31() just allows you to do this in a more 'hands on' way.

What follows is a script file you can type in yourself. By now in your MIT experience, you are probably well-acquainted with MATLAB script files. If not, begin now! They are extremely useful. (Ask around the lab for help.)

Note that when using the mtrc31 function 'GetAddr' (p.13 in the mtrace user's guide), you can use rti to specify a dSPACE RTI variable from a SIMULINK block diagram (in addition to more typical types, link int, etc).

15. Open a new m-file from the MATLAB File:Open menu and **type in the lines on the next page.** (You can ignor any lines beginning with '%', which are comments and will be ignored by MATLAB). You should still have the model smd.mdl running (which you built in step **6**).

mtrc31() example code:

```
%   You can use many commands here as templates to your own files.
%   mtrc31 example script: For use while running smd.mdl
%   ** Collects data during a Simulink run and then plots it.

%   Always SELECT BOARD. then, Get address of smd system

mtrc31('SelectBoard','DS1102');
kt1=mtrc31('GetAddr','rti B[Model Root/Spring-Mass-Damper System]');
mtrc31('TraceVars',kt1)

%   Define triggering: 410 samples, 10 samples pre-triggering...

mtrc31('SetFrame',[],1,-10,410);
mtrc31('SetTrigger',kt1(1,:),0,1);

%   lock program during capture...

mtrc31('LockProgram');

%   start capture and wait until it is complete (w/ drawnow loop)

mtrc31('StartCapture')
while mtrc31('CaptureState')~=0
   drawnow;
end

%   transfer data when capture is complete and then release lock

out_data = mtrc31('FetchData');
mtrc31('UnlockProgram');

%   plot results
%   Since we should have sampling at 1e-4 (as set in the
%   'Simulation:Parameters' menu of the SIMULINK model,
%   before we ever did our Build), we can use this scaling
%   for the data collected [for the units of Time. One point
%   was recorded per sample period]

clf;
plot((1e-4)*[1:410],out_data'); hold on; grid on;
plot((1e-4)*[1:410],out_data','.');

% Note: You should be able to convince yourself the period is OK,
% by looking at the plot.  How often does the signal repeat,
% on the x-scaling here?
```

D.3 Using trcview

Both mlib and mtrace reference variables used in the RTI-generated C-code. The trace file identifies these parameter, but the syntax required for mlib and mtrace is somewhat different from that found in the trace file. Trcview is a convenient tool for finding these appropriate names.The steps below show how to use trcview to find correct syntax.

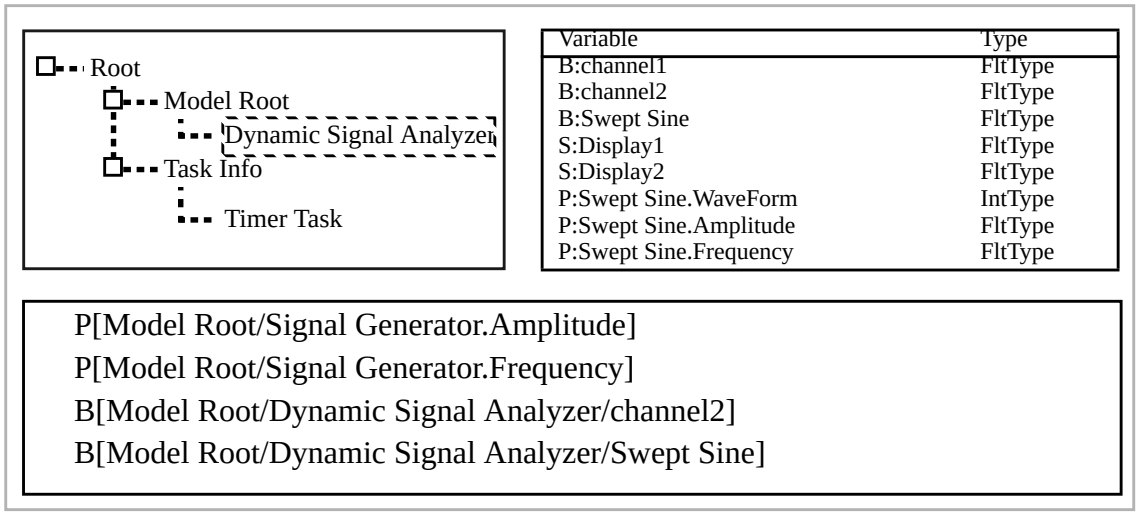


Figure D.1: The Trcview Layout

1. At the MATLAB prompt, enter: **trcview**

2. Select File:Load and pick from available trace files (with the '.trc' extension).

3. Figure D.1 shows the layout of the trcview tool. **In the upper, left window, clicking on a box will open (or close) branches stemming from this location.** (For example, the box to the left of 'Model Root' will reveal or hide the subgroup 'Dynamic Signal Analyzer'.)

4. In the same window, **select the group (or subgroup) of interest** to display variables.

5. These variable names will appear in the upper, right window. **Click on a variable**, and its mlib-appropriate name will appear in the lower window.

The variable names shown in the lower window of Figure D.1 correspond to the trace names in the shaded regions of Figure C.1 With some experience, it is typically possible to derive the right syntax for mlib and mtrace directly from the '.trc' file.

References

[1] Cooley, J.W.; Lewis, P.A.W.; and Welch, P.D., *The Fast Fourier Transform and its Application* (Yorktown Heights: IBM Watson Research Center, 1967).

[2] Dabney, James B. and Thomas, L. Harman, *Mastering Simulink* (Upper Saddle River, N.J.: Prentice-Hall, 1998).

[3] Ersoy, Okan, *Fourier-Related Transforms, Fast Algorithms and Applications* (Upper Saddle River, N.J.: Prentice Hall, 1997).

[4] Franklin, Gene F., Powell, J. David and Workman, Michael L., *Digital Control of Dynamic Systems* (Reading: Addison-Wesley 1990).

[5] *The Fundamentals of Signal Analysis* [Application Note 243] (Hewlitt-Packard Co., 1994).

[6] Horowitz, Paul and Hill, Winfield, *The Art of Electronics, 2nd Edition* (New York: Cambridge University Press, 1989)

[7] Lynn, Paul A. and Fuerst, Wolfgang, *Digital Signal Processing* (New York: John Wiley & Sons, 1989).

[8] *MLIB MATLAB-DSP Interface Library: User's Guide* [Version 3.0] (Paderborn, Germany: dSPACE).

[9] Oppenheim, Alan V., Willsky, Alan S. and Young, Ian T., *Signals and Systems* (Englewood Cliffs, N.J.: Prentice-Hall, 1983).

[10] Ramirez, Robert W., *The FFT: Fundamentals and Concepts* (Englewood Cliffs, N.J.: Prentice-Hall, 1985).

[11] *Real-Time TRACE Module for MATLAB* [Version 2.0] (Paderborn, Germany: dSPACE).

[12] Stearns, Samuel D. and Hush, Don R., *Digital Signal Analysis* (Englewood Cliffs, N.J.: Prentice Hall, 1990).

[13] Strang, Gilbert, *Introduction to Applied Mathematics* (Wellesley: Wellesley-Cambridge Press, 1986).