MATRIXx™

Xmath™ Interactive System Identification Module, Part 1

April 2004 Edition
Part Number 370758B-01

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices

Australia 1800 300 800, Austria 43 0 662 45 79 90 0, Belgium 32 0 2 757 00 20, Brazil 55 11 3262 3599,
Canada (Calgary) 403 274 9391, Canada (Ottawa) 613 233 5949, Canada (Québec) 450 510 3055,
Canada (Toronto) 905 785 0085, Canada (Vancouver) 514 685 7530, China 86 21 6555 7838,
Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 385 0 9 725 725 11,
France 33 0 1 48 14 24 24, Germany 49 0 89 741 31 30, Greece 30 2 10 42 96 427, India 91 80 51190000,
Israel 972 0 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970, Korea 82 02 3451 3400,
Malaysia 603 9131 0918, Mexico 001 800 010 0793, Netherlands 31 0 348 433 466,
New Zealand 0800 553 322, Norway 47 0 66 90 76 60, Poland 48 22 3390150, Portugal 351 210 311 210,
Russia 7 095 783 68 51, Singapore 65 6226 5886, Slovenia 386 3 425 4200, South Africa 27 0 11 805 8197,
Spain 34 91 640 0085, Sweden 46 0 8 587 895 00, Switzerland 41 56 200 51 51, Taiwan 886 2 2528 7227,
Thailand 662 992 7519, United Kingdom 44 0 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on the documentation, send email to techpubs@ni.com.

Important Information

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Trademarks

Patents

For patents covering National Instruments products, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your CD, or `ni.com/patents`.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

Conventions

The following conventions are used in this manual:

»	The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.
	This icon denotes a note, which alerts you to important information.
	This icon denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.
bold	Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names.
italic	Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.
`monospace`	Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames, and extensions.
`monospace italic`	Italic text in this font denotes text that is a placeholder for a word or value that you must supply.
`monospace bold`	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.

Contents

Chapter 1
Introduction

Chapter 2
Identification Process

Chapter 3
Identification Algorithms

Chapter 4
Tutorial

1 Introduction

This chapter provides an outline of the manual, some use notes, and an overview of the Xmath Interactive System Identification Module. The overview includes a complete listing of the ISID functions, grouped according to their use. The final section of the chapter briefly describes how ISID applies Xmath's interactive graphical user interface (GUI) to the complete identification process.

Using This Manual

The goal of this manual is to explain the general steps of the system identification process, and to show how ISID interactive tools and functions simplify and streamline this process. It defines the mathematics of system identification problems and provides step-by-step examples of problem solving sessions.

Document Organization

This manual includes the following chapters and appendices:

- Chapter 1, *Introduction*, provides an overview of the contents of this manual and pinpoints the sections that will be of most use to both first-time and long-term ISID users. It provides a complete listing of the ISID functions, grouped according to their use. This chapter also describes how ISID applies Xmath's interactive graphical user interface (GUI) to the complete identification process.
- Chapter 2, *Identification Process*, is a general primer on system identification. It describes how Xmath and ISID functions fit into the framework of system identification. It provides general guidelines to help select the most appropriate identification approach for a particular problem.
- Chapter 3, *Identification Algorithms*, covers the concepts and mathematics underlying the identification functions provided with ISID. It discusses the numerical implementation of the algorithms and explains the mathematical significance of the keywords.
- Chapter 4, *Tutorial*, illustrates how to use the interactive GUI tools in a sample identification/validation session. It describes how the ISID

functions can be used to identify models of a simulated two mode mechanical system with two inputs and two outputs.

- Appendix A, *List Data Structures*, summarizes the structure of the list objects used to implement matrix polynomial-based system models and the conversion functions between them.
- Appendix B, *Loading Data with the read( ) Function*, gives a brief overview on loading external data into Xmath with the `read` function.
- Appendix C, *Tool-Specific GUI Features*, describes the ISID interactive tools in detail. It summarizes the general features common to all the tools, then lists the approach-specific features of each tool.
- Appendix D, *Bibliography*, lists a table of bibliographic references. Throughout this document, bibliographic references are cited with bracketed entries. For example, a reference to [VODM1] corresponds to a paper published by Van Overschee and De Moor.
- Appendix E, *Technical Support and Professional Services*, describes support available from National Instruments.

How to Use This Manual

First-time users should read Chapter 2, *Identification Process*, on general features of ISID and the background of the GUI implementation. You may then wish to jump directly to the tutorial in Chapter 4, *Tutorial*, for a hands-on illustration. Because the tutorial focuses on user actions and results using tool-specific GUI features, you may want to look over Appendix C, *Tool-Specific GUI Features*, while working through it.

Users who are interested in details of the identification algorithms and their numerical implementation should read Chapter 3, *Identification Algorithms*, which also is helpful for a better understanding of the recommendations made in Chapter 2, *Identification Process*.

Readers who are not familiar with Parameter Dependent Matrices (PDMs) should consult the *Xmath User Guide* before using ISID functions and tools. Although several ISID functions accept both PDMs and matrices as input parameters, PDMs are preferable because they can include additional information that is useful for simulation, plotting, and signal labeling.

Commonly-Used Nomenclature

This manual uses the following general nomenclature:

- Matrix variables are generally denoted with capital letters; vectors are represented in lowercase.
- $G(s)$ is used to denote a transfer function of a system where s is the Laplace variable. $G(q)$ is used when both continuous and discrete systems are allowed.
- $H(s)$ is used to denote the frequency response, over some range of frequencies of a system where s is the Laplace variable. $H(q)$ is used to indicate that the system can be continuous or discrete.
- A single apostrophe following a matrix variable, for example, x', denotes the transpose of that variable. An asterisk following a matrix variable (for example, A*) indicates the complex conjugate, or Hermitian, transpose of that variable.

Related Publications

For a complete list of MATRIXx publications, refer to Chapter 2, *MATRIXx Publications, Help, and Customer Support*, of the *MATRIXx Getting Started Guide*. The following documents are particularly useful for topics covered in this manual:

- *MATRIXx Getting Started Guide*
- *Xmath User Guide*
- *Xmath Control Design Module*
- *Xmath Interactive Control Design Module*
- *Xmath Interactive System Identification Module, Part 2*
- *Xmath Model Reduction Module*
- *Xmath Optimization Module*
- *Xmath Robust Control Module*
- *Xmath Xμ Module*

MATRIXx Help

Interactive System Identification Module function reference information is available in the *MATRIXx Help*. The *MATRIXx Help* includes all Interactive System Identification functions. Each topic explains a function's inputs, outputs, and keywords in detail. Refer to Chapter 2, *MATRIXx Publications, Help, and Customer Support*, of the *MATRIXx Getting Started Guide* for complete instructions on using the *MATRIXx Help* feature.

Overview

The Xmath Interactive System Identification Module, referred to as ISID, comprises system identification, model reduction, and signal analysis tools for identification of *linear*, *discrete time*, and *multivariable* systems. In addition, model parameters of general nonlinear (SystemBuild) models can be estimated using ISID's `maxlike( )` function.

ISID is designed to encompass the entire identification process, from raw data analysis to validation of identified models. The ISID functions use highly reliable numerical algorithms and are capable of identifying large multivariable models of high-order systems from large amounts of data. They cover a wide range of identification methods. You can select different algorithms depending on the particular type of application and the size of the problem.

Function Categories

The tables in this section give a complete list of functions relevant to the system identification topics addressed in this manual:

- Table 1-1, *Nonparametric Identification Methods*
- Table 1-2, *Identification and Model Reduction*
- Table 1-3, *State Space Model Transformations*
- Table 1-4, *Polynomial Model Transformations*
- Table 1-5, *Validation Functions*
- Table 1-6, *Combining Separate Data Sets*
- Table 1-7, *Input Design*
- Table 1-8, *General Functions*
- Table 1-9, *Preprocessing Functions*

Several Xmath core functions are frequently used with ISID functions, but they are not listed here. Notice that the filtering functions included with the Xmath core are frequently required for preprocessing operations.

Table 1-1. Nonparametric Identification Methods

Function	Description
`etfe( )`	Empirical transfer function estimation
`sdf( )`	Spectral density function estimation

Table 1-2. Identification and Model Reduction

Function	Description
`armax( )`	Prediction error method for ARMAX models
`bj( )`	Prediction error method for Box-Jenkins models
`giv( )`	Generalized instrumental variables
`initmodel( )`	Initial model estimation for use with `pem`
`initx0( )`	Estimator of initial state
`irea( )`	Identification from impulse response data
`ls( )`	Time domain least squares
`maxlike( )`	General maximum likelihood estimation of continuous, discrete, linear, or nonlinear systems
`oe( )`	Prediction error method for output error models
`pem( )`	Prediction error method
`sds( )`	Subspace identification method
`sst( )`	Subspace identification method for output-only data
`tfid( )`	Continuous-time SISO transfer function identification from frequency response data

Table 1-3. State Space Model Transformations

Function	Description
`canform( )`	Converts a state-space system to canonical form
`ctrcf( )`	Converts a state-space system to controllable canonical form
`get_inn( )`	Converts an innovations model in ISID format to a standard innovations model
`obscf( )`	Converts a state-space system to observable canonical form
`put_inn( )`	Converts a standard innovations model to an innovations model in ISID format
`reflect( )`	Reflects poles and zeros from the outside of the unit circle to the inside

Table 1-4. Polynomial Model Transformations

Function	Description
`arma( )`	Creates an ARMA system
`arma2ss( )`	Converts an ARMA system to a state-space system
`bpm( )`	Creates a Backward-Polynomial innovations Model (BPM) system
`bpm2inn( )`	Converts a BPM to a state-space innovations model
`bpmjoin( )`	Combines two ARMA models (stochastic and deterministic part) into a BPM
`bpmsplit( )`	Splits a BPM into its stochastic and deterministic ARMA components
`inn2bpm( )`	Converts a state-space innovations model to a backward-polynomial model
`innjoin( )`	Combines deterministic and stochastic models into a backwards polynomial innovations model

Table 1-4. Polynomial Model Transformations (Continued)

Function	Description
`innsplit( )`	Splits a state space innovations model into its deterministic and stochastic components
`ss2arma( )`	Converts a standard state-space system to an ARMA model

Table 1-5. Validation Functions

Function	Description
`etfe( )`	Empirical transfer function estimation and estimation of the noise spectral density function
`giv2var( )`	Computes equation error norms from the `giv` cross-product matrices for innovations models
`idfreq( )`	Frequency response and noise spectral density function of innovations models
`idimpulse( )`	Impulse response and noise covariance function of innovations models
`idsim( )`	General simulation of innovations models
`inn2pe( )`	Prediction error computation of innovations models
`inn2unc( )`	Computes frequency response confidence intervals for innovations models
`ls2var( )`	Computes prediction error variances from the `ls` square root
`ls2unc( )`	Computes frequency response confidence intervals for least-squares-derived ARX models
`polezero( )`	Pole/zero computation and plot
`sdf( )`	Spectral density function estimation
`val( )`	General model validation

Table 1-6. Combining Separate Data Sets

Function	Description
`givjoin( )`	Combines two instrumental variables cross product matrices into a cross product matrix containing the model information of both
`lsjoin( )`	Combines two least-squares square roots into a square root containing the model information of both

Table 1-7. Input Design

Function	Description
`prbs( )`	Pseudo-random binary sequence generator
`sweep( )`	Sine sweep (chirp) generator

Table 1-8. General Functions

Function	Description
`mtxplt( )`	Plotting tool designed to produce a matrix of plots

Table 1-9. Preprocessing Functions

Function	Description
`poltrend( )`	Polynomial trend removal
`taper( )`	Data tapering

Graphical User Interface

Several ISID functions have a built-in graphical user interface (GUI) designed to let you quickly compare and validate different models and re-examine new results. These GUIs are available for the following functions:

```
etfe( )       irea( )       sst( )

fwls( )       ls( )         val( )

giv( )        sds( )
```

Each of these functions has an optional `{gui}` keyword that you can specify to invoke its GUI. The purpose of these GUIs is twofold:

- With each of the GUI functions, most of the computation time is spent on obtaining an intermediate result—a singular value decomposition, square root, or cross product matrix—from which you can obtain models of different order quickly. The intermediate step is ideally suited to a GUI-supported operation, for you can save a lot of time compared to re-executing the entire computation for each new parameter setting.
- GUIs conveniently obtain and display results efficiently in a standardized format from which you can readily make hard copies.

The graphical interface has been implemented as a set of single-window interfaces (one for each function), each with a standardized pulldown menu for model validation. The data associated with each GUI is stored in a separate Xmath partition. Within any given GUI, you can graphically compare results with those of other methods by exchanging models and/or data through variables in the Xmath `main` partition.

2

Identification Process

This chapter is a general primer on system identification. It describes how Xmath and ISID functions fit into the framework of system identification. It provides general guidelines to help you select the most appropriate identification approach for a particular problem.

System Identification

System identification is by its nature an iterative process. Raw data from a real-world system is acquired, formatted, and processed as necessary, identified through a mathematical algorithm to return a model representation, after which the model is assessed to see how well it describes the observed system behavior.

During the process, several choices have to be made. First, the identification itself involves many variables—sampling interval, model order, the type of mathematical model to be used, and the fit criterion. The user must select all of these, typically experimenting with a range of options. While different criteria can be used to select appropriate values and a priori information about the system can be incorporated to good use, real-world systems very seldom have one true model that completely describes all observed behavior. As a result, the engineer is most likely to have a number of models that describe the behavior to some extent.

When a model has been obtained, it needs to be validated—tested to see how well the behavior of the model corresponds to both the observed data, any prior knowledge of the system, and the purpose for which it will be used. In the event that its behavior is not adequate, the identification process has to be revised or even reconsidered using another approach.

In addition to all these factors influencing the identification result using a particular algorithm, more than one algorithm is sometimes appropriate to a particular data set. In such a case, it is typically desirable to compare the models obtained across algorithms. The choice of algorithm depends on model structure, stochastic assumptions, and numerical properties of the algorithm.

Loading and Preprocessing Data

The first step in performing identification is to import the measured system data and ensure that it is formatted correctly. You can find instructions on how to import data into Xmath in Appendix B, *Loading Data with the read( ) Function*. After the data has been loaded into Xmath and is structured compatibly with the ISID tools, a number of preprocessing techniques can help ensure that the data is as good (as free from external noise and other corruption) as possible. We summarize the most commonly applied techniques, appropriately referring to ISID and Xmath functions.

- **Data inspection**—The human eye is an unsurpassed detector of signal corruptions or errors that occur during the preprocessing. Always plot the data. There is no better way to spot outliers, clipped saturation, or quantization effects. Periodic disturbances are visualized best by plotting the spectral density function of the data using the ISID `sdf( )` function.
- **Subtraction of nominal signal values**—System identification results in models that are a linearized version of the true system around the operating point. Since linearization is done with respect to the signal values relative to the operating point, we have to subtract the operating point signal values from the system identification data. The only exception where subtracting the signal averages is not required is the case where the system is known to be linear; in practice, this is rarely the case.
- **Trend removal**—Due to external influences, it is possible that low frequency and/or periodic signal components, which are irrelevant to the particular modeling problem of interest, are added to the data. For example, variations due to the 24-hour day cycle in power plants, seasonal influences in biological and economical systems, thermal expansion in rolling mills, or 50 and 60 Hz components. The amplitude associated with trends can be quite large and will corrupt the results of signal analysis as well as parametric identification algorithms. Some ways to perform trend removal are as follows:
  - **Polynomial fit**—Fitting a polynomial up to a certain order using a least-squares criterion is a reliable way of removing most trend-like disturbances. Sometimes the fit tends to curl at the extremes of the sampling range, which may make it necessary to discard some of the samples. You can use the ISID function `poltrend( )` to remove trends.
  - **Fourier transform based methods**—In the case where a periodic trend must be removed, we can take the Fourier transform of the data, set the corresponding Fourier coefficients to zero, and

transform the modified series back to the time domain. You can use the Xmath core functions `fft( )` and `ifft( )` to accomplish this. To prevent undesired spikes in the Fourier transform, taper the data before applying such techniques. You can accomplish this with the ISID function `taper( )`.

– **Highpass filtering**—With the filter design functions included with the Xmath core routines, highpass filters can be designed with a sharp cutoff characteristic. Estimating a good initial condition for the filter can be important. You can use the ISID function `initx0( )` for this purpose. To prevent undesired phase shifts, filtering twice in opposite time directions is useful. All of these operations are easily performed using standard Xmath functions.

- **Outliers**—Outliers are easily detected by visual inspection. The functions `sort( )` and `spline( )` are helpful in visualizing and correcting outliers. For instance, with the following sequence,

  ```
  [sortX,indx] = sort(X); plot(indx,sortX)
  ```

  you can easily recognize and retrieve the indices corresponding to outliers of the signal stored in the vector `X`. After that, splining the intermediate values using the `spline( )` function usually does the job.

- **Scaling**—Preferably, the magnitude of the detrended signal values should be in the order of magnitude of 1. The reasons for this are partly numerical and partly due to the way in which some algorithms are influenced by scaling. In particular, SVD-based algorithms such as the ISID functions `irea( )`, `sds( )`, and `sst( )` are quite sensitive to scaling effects in the non-SISO case.

- **Periodic disturbances**—Periodic disturbances are most easily removed using the `fft( )` and `ifft( )` functions, as discussed in the *Trend Removal* bullet.

- **Tapering**—This is a pointwise multiplication of the signal vectors by a function which goes to zero in a smooth way near the extremes of the data interval. The ISID function `taper( )` allows you to specify the fraction of the data to taper as well as the window type. Tapering should be done with most Flatbeds techniques because of the implicit periodicity assumption that holds for all Fourier methods in combination with nonzero initial and final conditions of the data interval. The ISID signal analysis functions `sdf( )` and `etfe( )` incorporate tapering by default. Time domain batch identification functions (`ls( )`, `giv( )`, `sds( )`, `sst( )`, and `irea( )`) perform better without tapering.

- **Signal transformations**—Because the ISID identification algorithms work with linear model structures, it can be important to precompensate for static nonlinearities. Examples are as follows:
  - Multiplication by the inverse of a calibration curve for an optical sensor.
  - The choice between enthalpy or pressure and temperature as model outputs of a boiler system.
- **Postfiltering and desampling**—If the sampling frequency is much larger than the bandwidth of the system or extends way beyond the coherence bandwidth, it is advisable to decrease it by taking every nth sample, thereby constructing a new *desampled* signal. It is imperative to filter the data before the desampling is done; otherwise, the desampled data will be corrupted by aliasing effects.

Signal analysis is an important step to validate the quality of the data at each stage of the preprocessing procedure. In particular, estimates of the *spectral density function* (SDF) and *coherence* are instrumental in obtaining the right desampling rate and in the removal of periodic components. The coherence will also help interpreting the results of parametric identification methods at a later stage in the identification process. How well signal analysis can be applied depends on the data that should be stationary in a stochastic sense. We refer to [PRIES] for a comprehensive treatment of signal analysis in practice.

You can obtain coherence, covariance function, and spectral density estimates using the ISID `sdf( )` function. ISID signal analysis capabilities are illustrated in the *Empirical Transfer Function Estimation* section of Chapter 3, *Identification Algorithms*.

Choosing a Modeling and Identification Scheme

After the preprocessing and signal analysis, you must apply parametric methods to arrive at a final description in the form of a linear system that you can use for prediction and/or control design purposes. There is a wide variety of methods available from the literature. The final choice of algorithm depends on the desired model structure, numerical efficiency, and model quality criterion.

Model Structures

Linear discrete time systems are most commonly represented in *state space* or *polynomial* form. You can define polynomial model structures in the *forward* or *backward shift* operator. From an input/output point of view, the model class that you choose is irrelevant; you can convert polynomial models to state space form and vice versa. The state space representation is often preferred, but polynomial models can be much more efficient in the case of high-order models.

System identification favors using state space models because a lot is known about the parameterization of state space models in so-called *canonical form.* When a state-space system is in canonical form, there is a one-to-one correspondence between the model parameters and the input/output behavior within the model set. Consequently, parameters can be identified uniquely from input/output data.

ISID uses state space systems by default. The result of identification methods that are based on polynomial model representations are returned in their state space equivalent form. These representations include least squares (`ls`), instrumental variables (`giv`), and prediction error identification of Box-Jenkins (`bj`), output error (`oe`), and ARMAX (`armax`) models. In this section we discuss these different ways of model representation and introduce *innovations models*.

- **State-space models**—Discrete-time state-space models are of the form:

$$x_{t+1} = Ax_t + Bu_t$$
$$y_t = Cx_t + Du_t$$

  The corresponding transfer function is given by:

$$G(z) = C(zI - A)^{-1}B + D$$

  For any invertible matrix T, the transfer function of a system described by the matrices (TAT^{-1}, TB, CT^{-1}, D) is identical to that of A,B,C,D. Consequently, if the dimension of the state-space is n, there is an ambiguity of at least n^2 parameters represented by the T matrix. A canonical form is based on a specific choice of T, or equivalently, choice of basis vector of the state space. For an introduction to canonical forms, refer to [KAI] and the references therein.

- **Auto regressive moving average (ARMA) models**—ARMA models are based on polynomials in the backward shift operator. The time domain model equation is as follows:

$$\sum_{k=0}^{n_A} A_k y_{t-k} = \sum_{k=0}^{n_B} B_k u_{t-k}$$

Here, the A_k matrices are square and $A_0 = I$. The transfer function is defined by

$$G(z) = A(z)^{-1} B(z)$$

where:

$$A(z) = \sum_{k=0}^{n_A} A_k z^{-k}$$

$$B(z) = \sum_{k=0}^{n_B} B_k z^{-k}$$

Note ARMA models have a different shift operator than Xmath transfer function models—backward shift instead of forward. The lowest order denominator coefficient of an ARMA model is identity. In transfer function models, the highest order coefficient is identity.

- **State-space equivalents of ARMA models**—Let us assume that an ARMA model ($A(z)$, $B(z)$) is given with transfer function $G(z) = A(z)^{-1}B(z)$ and that the orders of the polynomials are the same: $n = n_A = n_B$.

The following state-space system (A,B,C,D) has the same transfer function $G(z)$:

$$\left(\begin{array}{cc} A & B \\ C & D \end{array}\right) = \left(\begin{array}{cccccc|c} -A_1 & I & 0 & 0 & \cdots & 0 & B_1 - A_1 B_0 \\ -A_2 & 0 & I & 0 & \cdots & 0 & B_2 - A_2 B_0 \\ \vdots & 0 & \ddots & & 0 & \vdots & \vdots \\ -A_{n-2} & 0 & 0 & 0 & I & 0 & B_{n-2} - A_{n-2} B_0 \\ -A_{n-1} & 0 & 0 & 0 & 0 & I & B_{n-1} - A_{n-1} B_0 \\ -A_n & 0 & 0 & 0 & 0 & 0 & B_n - A_n B_0 \\ \hline I & 0 & 0 & 0 & 0 & 0 & B_0 \end{array}\right)$$

The state-space model has $n_s = n \times n_y$ states. Generically (if the coefficient matrices were chosen at random), the state-space representation is minimal.

This demonstrates the storage efficiency of ARMA models—a relatively small number of polynomial coefficients can represent a state-space model that is much larger. This is especially useful for dealing with high-order models that are frequently encountered with least-squares applications.

- **Innovations models**—So far we have considered models with input u_t and output y_t without a separate noise input. In most system identification problems, an additional unobserved input e_t is present. In state-space form, these models are described by Equation 2-1.

$$\begin{aligned} x_{t+1} &= Ax_t + Bu_t + Ke_t \\ y_t &= Cx_t + Du_t + e_t \end{aligned} \qquad (2\text{-}1)$$

Prediction error algorithms compute e_t from y_t and u_t:

$$\begin{aligned} x_{t+1} &= (A - KC)x_t + (B - KD)u_t + Ky_t \\ e_t &= -Cx_t - Du_t + y_t \end{aligned}$$

Asymptotic stability of A – KC is therefore important. When this condition is satisfied, we call the state-space model in Equation 2-1 an *innovations model* or a model in *innovations form*. We call the subsystem with input u_t the *deterministic* part and the subsystem with the noise input e_t the *stochastic* part. Refer to [AndMo] for an in-depth treatment of innovations representations, and the relationship with Kalman Filters.

Note ISID represents innovations models slightly differently—in order to store both stochastic and deterministic parts and also the variance of e_t in a single Xmath system variable, the stored system is of the form (A, [B, KS], C, [D, S]) where S is a left square root matrix of the noise variance: var(e_t) = SS^T. The ISID functions `get_inn( )` and `put_inn( )` transform the ISID form into the standard innovations form and vice versa.

- **Backward polynomial innovations (BPM) models**—Innovations models in polynomial form are expressed as follows:

$$A(z)y_t = F(z)^{-1}B(z)u_t + D(z)^{-1}C(z)e_t$$

As a special case we have the well known ARMAX model structure:

$$A(z)y_t = B(z)u_t + C(z)e_t$$

This model structure has a state-space equivalent of the form:

Another well-known special case is the Box-Jenkins model structure, described by:

$$y_t = F(z)^{-1}B(z)u_t + D(z)^{-1}C(z)e_t$$

Its state-space equivalent is of the form:

$$\left(\begin{array}{ccc} A & B & K \\ C & D & I \end{array}\right) = \left(\begin{array}{cccc|c} -F_1 & I & & & B_1 - F_1B_0 \\ \vdots & & I & & \vdots \\ -F_n & 0 & & & B_n - F_nB_0 \\ \hline & & -D_1 & I & C_1 - D_1 \\ & & \vdots & & \vdots \\ & & & I & \\ & & -D_t & 0 & C_n - D_n \\ \hline I & & I & & B_0 \quad I \end{array}\right)$$

The essential difference between ARMAX and Box-Jenkins is the independent parameterization of the stochastic and deterministic parts with the Box-Jenkins model, whereas those of the ARMAX model have an identical state transition matrix parametrized by the coefficients $A_i (i = 1, \ldots, n)$.

$$\begin{pmatrix} A & B & K \\ C & D & I \end{pmatrix} =$$

$$\left(\begin{array}{cc|cccc|cc} -A_1 & I & 0 & 0 & .. & 0 & B_1 - A_1 B_0 & C_1 - A_1 \\ -A_2 & 0 & I & 0 & .. & 0 & B_2 - A_2 B_0 & C_2 - A_2 \\ : & & & & & : & : & : \\ -A_{n-2} & & & & I & 0 & B_{n-2} - A_{n-2} B_0 & C_{n-2} - A_{n-2} \\ -A_{n-1} & & & & 0 & I & B_{n-1} - A_{n-1} B_0 & C_{n-1} - A_{n-1} \\ -A_n & 0 & & & 0 & 0 & B_n - A_n B0 & C_n - A_n \\ \hline I & 0 & & .. & & 0 & B_0 & I \end{array}\right)$$

Some other well known polynomial model structures are ARX and output error models. ARX models are a special case of ARMAX models with $C(z) \equiv I$. Output error models are a special case of Box-Jenkins models with $C(z) \equiv I$ and $D(z) \equiv I$.

It is difficult to select a model structure without prior information. Some guidelines are as follows:

– The prediction errors of ARX models are linear in the parameters. This is a very important reason to use ARX models in practice since quadratic prediction error criteria have a unique global minimum.

– When the poles of the stochastic and deterministic part are known to be identical, you should use an ARMAX structure. Otherwise a Box-Jenkins model is preferable.

– You should only use output error models if the additive noise is white—in other words, if the noise model has no dynamics at all.

- **ISID model structures**—Although ISID uses state space models by default, it provides a set of functions for creation and manipulation of ARMA and backward polynomial models (BPMs) for specialized users and/or applications. These models are stored as list variables. The structures of these lists are described in Appendix A, *List Data Structures*.

  For application of prediction error methods (`pem( )`), ISID has a set of functions to create observable and controllable canonical forms

(`obscf( )` and `ctrcf( )`). The `canform( )` function returns a model in observable or controllable canonical form, depending on the norm of the parameter vector or the number of parameters. (This is not necessarily equal for observable and controllable canonical forms.)

Observable and canonical forms are based on algebraic properties of the system matrices and are therefore not necessarily the best model parameterization. A parameterization based on eigenvector/eigenvalue pairs or any user-defined model structure might be much more useful. It is not well known how the choice of parameterization affects the number of local minima of the (prediction error) criterion function.

Incorporating Prior Knowledge

The capability of incorporating prior knowledge is limited with most black-box identification algorithms. ISID has no specific tools to aid the user in this respect. The following can be accomplished with general ISID tools:

- **Delays**—To implement a delay of d samples, the input data must be shifted forward in time over d samples before the identification takes place. After the model has been identified, you need to add a delay to it. You can do this with standard Xmath functions. For instance, for a SISO system `sys` with time step 1, you can add a delay of 3 samples as follows:

  ```
  del= system(makepoly(1),makepoly([1,0,0,0]),{dt=1})
  sys = del*sys
  ```

- **Static gain**—For batch identification methods like least squares (`ls( )`) and the subspace methods (`sds( )`, `sst( )`) that optionally produce an intermediate square root matrix, a known static gain can be imposed as follows. First, you must obtain a square root of the data set (refer to Equation 3-1). You must create another square root from a data set with input equal to 1 and output equal to the known static gain. For the SISO case, this is easily generalized to MIMO. Then, using `lsjoin( )` with a large weight on the second square root, you combine these square roots into another one that you can then pass to `ls( )`. These steps provide the final model with the desired static gain.

  You also can apply this technique to instrumental variables (`giv( )`), where you use `givjoin( )` to combine the cross product matrices that play the same role as the square root matrix with least squares. Besides static gain, you can emphasize any kind of data in the model this way.

- **Known system matrix elements and parameter structure**—With the prediction error method (`pem( )`, `initmodel( )`), you can pass a model structure described by a template system composed of integer system matrices. The integers define which elements of the system matrices are parameters, which ones are fixed, and which ones are described by identical parameters or, eventually, related by a minus sign.
- **Dependence of deterministic and stochastic parts**—As described earlier, polynomial models are expressed in a form that displays the dependence between the parameters of the deterministic and stochastic parts. For instance, a Box-Jenkins model structure reflects independence of these parameters. A restriction is that the use of specific model structures is restricted to a specific identification method (for instance, ARX to `ls( )`, Box-Jenkins to `pem( )`).
- **Frequency dependent model accuracy**—Least squares in the frequency domain (`fwls( )`) uses an explicit weight function that you can use to emphasize or de-emphasize the fit in certain frequency regions. You can do this for each input independently. The weight function is typically based on expected model accuracy derived from the coherence estimate (`etfe( )`, `sdf( )`).

  With the `etfe( )` function, you can apply a similar technique where you can use a frequency-dependent weight to modify the estimated impulse response (inverse Fourier transform of the empirical transfer function estimate). You can then use the resulting frequency-weighted impulse response provided by `irea( )` to obtain a parametric model.

  Another general way to (de-)emphasize frequency regions using time domain methods is to filter both inputs and outputs with a filter representing the desired weight characteristics. You can relate these characteristics to prior knowledge or the goal for model usage.

The only algorithm where much more specific prior knowledge can be imposed is the `maxlike( )` function. The price for this usage is that this function uses a much less efficient search algorithm.

Identification/Selection of ID Approach

The question of which method to use for a particular data set depends on the data, model (structure) requirements, prior knowledge, and numerical efficiency; it cannot always be answered directly. Each identification method is in some sense suboptimal for most identification problems. Therefore, it might pay to try several methods and pick the model that gives the best validation result.

Table 2-1 provides an overview of the pitfalls and benefits of each individual ISID algorithm. A combination of methods is often recommended; we mention some of them here.

- **Impulse realization of high order least squares models**—(`ls( )`/`irea( )`) The least squares method is widely used for its numerical efficiency, robustness, and uniqueness of solution. However, the stochastic part consists of a poles-only model, which produces limited noise modeling capabilities. Thus, the model order must be taken much higher than what would ideally be required. In this case, impulse model reduction is the best method to obtain the final model. The reduction result can be validated directly by comparison with the impulse response of the least squares model. Experience has shown that this two-step method is one of the most effective ways of obtaining models. Both steps are numerically easily solvable and require essentially just one parameter that must be chosen by the user (model order for `ls( )` and number of singular values for `irea( )`).
- **One-shot identification of time domain data**—(`sds( )`, `giv( )`) To obtain a model of the data in one step, the subspace algorithm and instrumental variables method are the most suitable ones. Using these algorithms, you can obtain models with minimal effort.
- **Identification of frequency domain data**—(`fwls( )`) In the case where you have frequency response data from a spectral analyzer, you can use the frequency domain least squares method directly to obtain a parametric model. You can pass a weight function for each input or create it graphically using the `fwls( )` GUI.
- **Identification of frequency domain data**—(`ifft( )`/`irea( )`) You can use the inverse Fourier transformation (`ifft( )`) to transform empirical frequency response data to the time domain. (To do this, the frequency response needs to be two-sided.) Next, you can apply `irea( )` to obtain a parametric model efficiently.
- **Closed-loop identification with a measured external reference**—(`etfe( )`, `giv( )`) External references are used explicitly by the empirical transfer function estimate (`etfe( )`) and instrumental variables (`giv( )`). To derive a parametric model from the `etfe( )` result, you can use `irea`.
- **Closed-loop identification without a measured external reference**—(`ls( )`, `pem( )`) [AndGev] and [Söd] have proven that under certain conditions on the delay structure of the loop and noise correlation, prediction error methods can be applied in closed loop situations to identify open loop models. This includes the least squares method since that is a prediction error method as well. The background theory of this approach to closed loop identification is based on an

infinite amount of data and idealized model assumptions with some additional restrictions. Therefore, you should interpret the results with care and, if possible, compare them with the result of simulated data of existent models.

- **Efficiency and robustness**—(`etfe( )`/`irea( )`, `ls( )`/`irea( )`, `etfe( )`/`fwls( )`, `sds( )`) In the case of large amounts of data and high-order models, memory-efficiency dictates the use of very efficient methods. You can call `ls( )` and `sds( )` with the `{lattice}` keyword to take advantage of a fast square root algorithm. Empirical transfer function estimation also is a robust and memory- and time-wise efficient method. Next, you can apply impulse realization and/or frequency weighted least squares to convert the nonparametric modeling results to the final model.
- **Modeling of general nonlinear systems in continuous or discrete time**—(`maxlike( )`) You can use the well-known `maxlike` algorithm to identify parameters in virtually any type of model structure. This includes models in SystemBuild, parameterized by variables on the Xmath stack. The time required to complete is generally long due to the fact that numerical derivatives are used.
- **Identification of continuous time systems from frequency domain data**—(`tfid( )`, `makecontinuous( )`) The `tfid( )` function fits a SISO model to a continuous time frequency response. Another approach to obtaining continuous time models is to use `makecontinuous( )`, which you can apply to MIMO models as well. Going from discrete to continuous time models can give problems though, particularly in the case of discrete time model poles on the negative real axis.
- **Fine-tuning the model**—(`pem( )`) When you obtain a good initial model with any of the methods mentioned above, you can obtain a final model using the prediction error method. This method performs a combination of Gauss-Newton and steepest descent methods to find the parameter estimate; it can be initialized with an initial deterministic or innovations model. `pem( )` has a built-in initial model estimator (`initmodel( )`) that you can use in case you make a one-shot call to `pem( )` and pass only model order information.
- **Time series modeling**—(`sst( )`, `ls( )`) You can perform identification of time series without external input using the stochastic subspace algorithm (`sst( )`) or least squares (`ls( )`). `sst( )` has an interactive GUI tool that makes it more convenient to use.

Using Intermediate Results

The command line syntaxes of least squares (`ls( )`), the subspace algorithms (`sds( )`, `sst( )`) and instrumental variables (`giv( )`) can take advantage of an intermediate numeric quantity from which models of different (lower) order can be extracted with a relatively small computational effort. `ls( )`, `sds( )`, and `sst( )` provide a square root matrix and `giv( )` provides a cross product matrix. The functions compute these quantities directly from the data and return them as optional outputs.

Saving these quantities for several data sets allows you to combine the results so that you have identification over several data sets. The functions for this are `lsjoin( )` and `givjoin( )`. This is particularly useful in the case where MIMO systems are excited one input at a time. Another benefit of saving these quantities is that extremely large identification problems can be broken apart into smaller ones.

Frequency weighted least squares (`fwls( )`) also produces a square root of the same format as the `ls( )` square root; therefore it can be combined with time domain square roots in case it is necessary to mix time and frequency domain information.

Model Validation

The final stage in the identification procedure is validation—providing a measure of the quality of the model obtained through the identification. We provide both quantitative and graphical validation tools.

We recommend the use of coherence, spectral density function, and empirical transfer function estimates (`sdf( )`, `etfe( )`). They provide important information about the signal to noise ratios in different frequency regions and help establish the reliability of parametric models later on.

To interpret coherence estimates in the MIMO case, notice that the ISID `sdf( )` function estimates the coherence of input-output pairs in a scalar manner. This implies that the coherence of any given input-output pair is decreased by the effect of all other inputs. Therefore, to judge the coherence of a single output signal of a system with the inputs, you must consider the coherences of all inputs with that particular output.

Frequency domain model error estimate functions are available for the least-squares (`ls2unc( )`) and general system models (`inn2unc( )`).

They are based on a pointwise conversion of the parameter variance estimate to a frequency response variance estimate.

Another important validation tool for least squares is the cross validation provided by `ls2var( )`. It computes prediction error norms from one or two square root matrices. The correct way of using `ls2var( )` is to pass two square roots—one obtained from the identification data set and one from the validation data set. In case a separate validation data set is not available, it is advisable to use the identification data set. In using `ls2var( )`, you should split the identification data set into two parts for obtaining the square roots if the data is stationary in a stochastic sense. For instrumental variables, `giv2var( )` plays a similar role as `ls2var( )` for least squares.

The `val( )` function is a general validation tool that provides a graphical interface for validating a model with or without an additional input-output data set. It allows you to view the data, prediction errors, cross-correlation of the input and prediction error, model responses, and pole-zero locations for the identified model.

Identification Function Feature Summary

For a summary of identification functions and their features, refer to Table 2-1.

Table 2-1. Identification Function Feature Summary

	etfe	fwls	giv	irea	ls	maxlike	pem	sds	sst	tfid
MIMO identification	x	x	x	x	x	x	x	x	x	—
Parametric estimation methods	—	x	x	x	x	x	x	x	x	x
Prediction error methods	—	—	—	—	x	x	—	—	—	—
Incorporates a priori knowledge or initial model estimate	—	—	—	—	—	x	x	—	—	—
Impulse response data	—	—	—	x	—	—	—	—	—	—
Frequency domain data	—	x	—	—	—	—	—	—	—	x
Continuous identification	—	—	—	—	—	x	—	—	—	x
Discrete identification	x	x	x	x	x	x	x	x	x	—
Nonlinear identification	—	—	—	—	—	x	—	—	—	—
Uses square roots for lower order re-identification	—	x	—	—	x	—	—	x	x	—

Table 2-1. Identification Function Feature Summary (Continued)

	etfe	fwls	giv	irea	ls	maxlike	pem	sds	sst	tfid
Includes interactive tool option	x	x	x	x	x	—	—	x	x	—
Large problem efficiency	—	—	—	—	x	—	—	—	—	—
Output-only	—	—	—	—	x	—	—	—	x	—

3

Identification Algorithms

This chapter describes the concepts and mathematics underlying the identification functions provided with ISID. This includes the numerical implementation of the algorithms and the mathematical significance of the keywords.

Least-Squares in the Time Domain

The least-squares approach, implemented in the `ls( )` function, is one of the most important in terms of efficiency, simplicity of operation, and robustness. The following sections describe the basic least-squares algorithm and the use of previously identified data to obtain lower-order models.

Least Squares for ARX Models

Assume that the following data set consisting of system inputs u_t and system outputs y_t is given.

$$\{u_1, y_1, u_2, y_2, \ldots, u_N, y_N\}$$

We want to fit the following ARX model to the data:

$$y_t + A_1 y_{t-1} + \ldots + A_n y_{t-n_A} = B_0 u_t + B_1 u_{t-1} + \ldots + B_n u_{t-n_B} + \varepsilon_t \quad (3\text{-}1)$$

The criterion to be minimized is given by

$$J(\vartheta) = \sum_{t=n+1}^{t=N} \varepsilon_t^T W \varepsilon_t$$

where W is a constant, positive definite, symmetric weighting matrix, and where ϑ is a parameter matrix containing the elements of the A_k's and B_k's. To simplify the problem formulation, we assume temporarily that $n = n_A = n_B$ and write all equations shown in Equation 3-1 for t = n + 1, …, N in matrix form.

$$\begin{pmatrix} u_n^T & y_n^T & u_{n-1}^T & y_{n-1}^T & .. & u_1^T & y_1^T & u_{n+1}^T & y_{n+1}^T \\ u_{n+1}^T & y_{n+1}^T & u_n^T & y_n^T & .. & u_n^T & y_n^T & u_{n+2}^T & y_{n+2}^T \\ : & : & : & : & : & : & : & : & : \\ u_{N-1}^T & y_{N-1}^T & u_{N-2}^T & y_{N-2}^T & .. & u_{N-2}^T & y_{N-2}^T & u_N^T & y_N^T \end{pmatrix} \begin{pmatrix} -B_1^T \\ A_1^T \\ -B_2^T \\ : \\ -B_n^T \\ A_n^T \\ \hline -B_0^T \\ I \end{pmatrix} = \begin{pmatrix} \varepsilon_{n+1}^T \\ \varepsilon_{n+2}^T \\ : \\ \varepsilon_N^T \end{pmatrix} \quad (3\text{-}2)$$

This equation is of the form:

$$(X \;\; Y)\begin{pmatrix} -\vartheta \\ \hline I \end{pmatrix} = E$$

The well-known standard least-squares solution is:

$$\hat{\vartheta} = \left[X^T X\right]^{-1} X^T Y \quad (3\text{-}3)$$

LS Square Root

The solution of the batch least-squares equation for the parameter vector ϑ, shown in Equation 3-3, is easily implemented but can be obtained in a more reliable manner by using a *square root* method as follows. Assume that the QR transform of (X Y) is of the form,

$$\left[X \;\; Y\right] = Q\begin{bmatrix} S_X & S_Y \\ 0 & S_Y \\ 0 & 0 \end{bmatrix} = Q\begin{bmatrix} S \\ 0 \end{bmatrix}$$

with S square and upper triangular. The least-squares solution is then given by

$$\hat{\theta} = {S_X}^{-1} S_Y \quad (3\text{-}4)$$

and the sum of squares error matrix by $S^T{}_E S_E = E^T E$. The term *square root* originates from the fact that S_X is a square root of $X^T X$, that is,

$S_X^T S_X = X^T X$. We will refer to S_X as the *LS square root*. It is stored in a format described in Appendix A, *List Data Structures*.

The arrangement of the data matrix shown in Equation 3-4 makes it relatively easy to extract lower-order models from the upper triangular matrices S_X and S_Y.

Singular Value-Based Solutions

An alternative solution based on the most significant components in X and Y is obtained from a singular value decomposition (SVD). Let the SVD of S_X be given by

$$S_X = U\ S\ V^T$$

where S is of the form:

$$S = \begin{bmatrix} \Sigma & 0 \\ 0 & 0 \end{bmatrix}$$

Σ is a diagonal matrix with positive diagonal elements. U and V are orthogonal matrices that are partitioned accordingly:

$$U = \begin{bmatrix} U_1 & U_2 \end{bmatrix} \qquad V = \begin{bmatrix} V_1 & V_2 \end{bmatrix}$$

Because of the orthogonality property of U, we can reformulate the least-squares problem as the minimization of

$$\left\| S_Y - USV^T\vartheta \right\| = \left\| U^T S_Y - SV^T\vartheta \right\|$$

If there are singular values equal to zero, then there will be several solutions. The one with the smallest norm for $V^T\vartheta$, and therefore also for ϑ, is given by

$$\hat{\vartheta} = \begin{bmatrix} V_1 \Sigma^{-1} U_1^T S_y \\ 0 \end{bmatrix}$$

In most practical applications, the data is always noisy and all singular values of S are larger than zero. In that case we set the singular values below a certain tolerance to zero and continue the computations in the way just shown.

Least Squares with Scalar Denominator

In cases where the elements of a multivariable transfer function are known to have the same poles, it is useful to use an identification algorithm which incorporates that property. In a MIMO model structures, this implies that

$$A_i = a_i I \quad (i = 1, \ldots, n)$$

where the a_i's are real-valued scalars. The fact that the off-diagonal elements are zero does not affect the linearity of the prediction error in the parameters, which implies that it is still possible to solve the parameters based on a standard least-squares method. We do not discuss the details of the required modifications to the algorithm, except to mention that it results in the same computational load as the standard square root least-squares algorithm. In other words, the matrices S_X, S_Y, and S_E for the standard least-squares problem are the same as with the scalar denominator case.

Fast Least Squares with a Lattice Algorithm

You can use the `{lattice}` option to speed up the identification of high-order models from large amounts of data. This option activates an order (n) algorithm that can be significantly faster for large n than the standard algorithm, which is of order (n^2). Asymptotically, the difference in speed is approximately a factor n/2. The storage requirements of the algorithm are of order (n) as well. The lattice algorithm is not a default choice because it is numerically less stable than the standard algorithm. For most data sets, however, the algorithm performs equally well.

Like the standard LS algorithm, the LS-lattice algorithm produces a square root. This *lattice square root* is different; it only contains the upper block row of the standard LS square root. For details, refer to [ALING]. `lsjoin( )` can use the lattice square roots in an identical manner as the standard LS square roots to combine the information of multiple data sets into a single square root. This square root then serves as the input for functions such as `ls( )`, `sds( )`, and `ls2var( )`.

Generalized Instrumental Variables

The generalized instrumental variables (GIV) method resembles the approach employed with LS. The primary difference is that the system to be identified is of the form:

$$\sum_{k=0}^{n_A} A_k y_{t-k} = \sum_{k=0}^{n_B} B_k u_{t-k} + \nu_t$$

where ν_t is a noise term that is not necessarily ideally white, as with the least squares case. Assume that we can define a sequence of *instrumental variables* vectors z_t that is uncorrelated with ν_t. For the open loop case, the input sequence—or a filtered version of it—is often a good choice.

Now define

$$Z = \begin{bmatrix} z^T_{n_1+1_f} & \cdots & z^T_{n_1-l_p} \\ \vdots & & \vdots \\ z^T_{t+l_f} & \cdots & z^T_{t-l_p} \\ \vdots & & \vdots \\ z^T_{n_2+l_f} & \cdots & z^T_{n_2-l_p} \end{bmatrix} \qquad Y = \begin{bmatrix} y^T_{n_1-1} & \cdots & y^T_{n_1-n_A} \\ \vdots & & \vdots \\ y^T_{n_1-n_A} & \cdots & y^T_{t-n_A} \\ \vdots & & \vdots \\ y^T_{n_2-1} & \cdots & y^T_{n_2-n_A} \end{bmatrix}$$

$$U = \begin{bmatrix} u^T_{n_1+1} & \cdots & u^T_{n_i-n_B} \\ \vdots & & \vdots \\ u^T_{t-1} & \cdots & u^T_{t-n_B} \\ \vdots & & \vdots \\ u^T_{n_2-1} & \cdots & u^T_{n_2-n_B} \end{bmatrix} \qquad Y_0 = \begin{bmatrix} y^T_{n_1} \\ \vdots \\ y^T_t \\ \vdots \\ y^T_{n_2} \end{bmatrix} \qquad U_0 = \begin{bmatrix} u^T_{n_1} \\ \vdots \\ u^T_t \\ \vdots \\ u^T_{n_2} \end{bmatrix}$$

where $n_1 = max(n_A, n_B, l_p) + 1$ and $n_2 = N - l_f$. l_f and l_p represent the number of future and past lags accounted for by the instrumental variables. The generalized instrumental variables estimate is then defined by the minimization over ϑ of $Z^T([Y, U_0, U]\vartheta^T - Y_0)$ where $\vartheta = [A_1, \ldots, A_{n_A}, B_0, \ldots, B_{n_B}]$.

For the case where $Z^T[Y, U_0, U]$ is a square matrix, this estimate is the standard instrumental variable (IV) estimate. In cases where this matrix has more rows than columns, ϑ is determined by a least squares solution, which is called the generalized instrumental variables estimate. This estimator can be considerably more robust than the standard IV estimator, in particular when $l_p + l_f$ is four to five times the model order.

Spectral Density Function Estimation

Assume that N samples of an observed signal x_t are available. The discrete Fourier transformation (DFT) $X(\omega_k)$ of x_t is defined as:

$$x_t = \left(\frac{1}{N}\right) \sum_{k=0}^{N-1} X(\omega_k) e^{i\omega_k t}$$

where

$$\omega_k = \frac{2\pi k}{N} \; (k \in Z)$$

This formula reflects the periodicity of period N, which is inherent in the definition of DFT. The inverse DFT formula is given by:

$$X(\omega_k) = \sum_{t=0}^{N-1} x_t e^{-i\omega_t k}$$

In case there are two signals y_t and u_t, the sample cross covariance function is defined by:

$$R_{yu}(m) = \frac{1}{N} \sum_{n=0}^{N-1} y_n u_{n-m}$$

The DFT S_{yu} or R_{yu} is known as the *cross spectral density function* (SDF) of y_t and u_t and is given by:

$$S_{yu}(\omega_k) = \sum_{t=0}^{N-1} R_{yu}(t) e^{-i\omega_t k}$$

In case $y_t = u_t$, the SDF is an estimate of the *auto spectral density function* and represents the power distribution of the signal as a function of frequency. Cross spectral densities are the basis for the computation of the *coherence* C_{yu}. This quantity is a normalized cross spectral density function defined as follows:

$$C_{yu}(\omega_k) = \frac{S_{yu}\omega_k}{\sqrt{S_{yy}(\omega_k)S_{uu}(\omega_k)}}$$

The coherence is a number between 0 and 1 which expresses the extent to which y and u are linearly related. The coherence is most easily interpreted as a frequency-dependent correlation coefficient.

The formulas above represent the basic (raw) definitions of SDF and covariance functions. SDF estimates are based on slightly modified definitions. The modifications are meant to reduce the variance of the estimate and to account for the periodicity assumptions, which do not hold for most practical cases. The most common methods to achieve this follow.

- **Frequency domain averaging**—Averaging the raw SDFs obtained from different windows of the time domain data. In the `sdf( )` and `etfe( )` functions, the window width is an input parameter. It is possible to let the windows overlap using the `{overlap}` keyword. This can give considerable improvement, in particular with nonstationary data.
- **Correlation averaging**—Averaging over the raw covariance functions that were obtained from different time domain windows. Taking the Fourier transform yields the smoothed SDF.
- **Autoregressive estimation**—Using high-order auto regressive (AR) models for parametric spectral estimation. The AR model order plays a similar role to the window width with the standard method.

The effect of nonperiodicity of the data in each window can be reduced by tapering the data in each window. This significantly reduces distortions in the Fourier coefficients. The choices of window type and the equations corresponding to them for an N-point window are:

- **Hamming**—Hamming window, given by the equation

  $0.54 – 0.46 * \cos(2\pi n/(N – 1))$ for $0 \le n \le N – 1$.

- **Hanning**—Hanning window, given by the equation

  $0.50 – 0.50 * \cos(2\pi n/(N – 1))$ for $0 \le n \le N – 1$.

- **Triangular**—Triangular window, given by the expression

$$2 * [0:\text{round}((N-1)/2,\{\text{down}\}), \text{round}((N-2)/2,\{\text{down}\}):0:-1]/(N-1).$$

- **Blackman**—Blackman window, given by the equation

$$0.52 - 0.50 * \cos(2\pi n/(N-1)) + 0.08 * \cos(2\pi n/(N-1)) \text{ for } 0 \le n \le N-1.$$

- **Rectangular**—Corresponds to no windowing, or a window function consisting of all ones.

Remarks on the Implementation of SDF

- Only the values in the frequency range $[f_{min}, f_{max}]$ are returned. This range can be specified by keywords. By default, $f_{max} = 1/2$ dt (the Nyquist frequency). Thus, specifying a 128-point SDF (implicitly over the digital frequency range $[0,\pi]$) returns as output a one-sided SDF of 65 points over $[0,\pi]$.
- The average value of the SDF estimates is independent of data length and number of windows. The normalization has been chosen such that the SDF of a N(0,1) distributed time series has an average value of one pointwise for all frequencies. It implies, however, that the vector norm of the SDFs is dependent on these parameters.
- Because of the normalization of the SDF discussed above, the covariance function obtained as the inverse Fourier transform of the `sdf( )` has the right scaling. In the case of white noise, the covariance function has a peak at the first element. To get the peak in the middle, pass the keyword `{covwrap}` to `sdf( )`.
- If the number of data samples n is not an even power of 2, only the first m samples are used within `sdf( )`, where m is the largest power of 2 not exceeding n. For data of a non-power-of-2 length, it is a good idea to taper and pad the data with zeros to a power of 2 before calling `sdf( )`. Refer to the `taper( )` function explanation in the *Tapering* bullet in the *Loading and Preprocessing Data* section of Chapter 2, *Identification Process*.
- Auto spectral density functions produced by `sdf( )` are guaranteed to be positive definite unless the `{bt}` keyword (correlating averaging) is used with a different window than the standard rectangular window. Under the same conditions, coherences computed with the `{coh}` keyword have a range of (0,1).

Empirical Transfer Function Estimation

Let $Y(\omega_k)$ and $U(\omega_k)$ be the discrete Fourier transforms of the system output y_t and input u_t, respectively. Assume that the time domain system equation is given by:

$$y_t = \sum_{k=0}^{N-1} G_k u_{t-k} + \sum_{k=0}^{N-1} H_k e_{t-k}$$

Under the assumption that y_t, u_t, and e_t are periodic with period N, the following frequency domain relationship holds:

$$Y(\omega_k) = G(e^{i\omega_k})U(\omega_k) + H(e^{i\omega_k})E(\omega_k) \tag{3-5}$$

Postmultiplication of Equation 3-5 by $U(\omega_k)^*$ leads to:

$$Y(\omega_k)U(\omega_k)^* = G(e^{i\omega_k})U(\omega_k)U(\omega_k)^* + H(e^{i\omega_k})E(\omega_k)U(\omega_k)^*$$

where $*$ indicates complex conjugate transposed.

If e_t and u_t, and therefore $E(\omega_k)$ and $U(\omega_k)$ are uncorrelated, this results in

$$\varepsilon\{Y(\omega_k)U(\omega_k)^*\} = G(e^{i\omega_k})\varepsilon\{U(\omega_k)U(\omega_k)^*\}$$

which leads to

$$Ge^{i\omega_k} = S_{yu}(\omega_k)S_{uu}(\omega_k)^{-1}$$

where S_{yu} and S_{uu} are the cross and auto spectral density functions defined in the *Spectral Density Function Estimation* section. This is the basic relationship that is used for estimation of the frequency response—the *empirical transfer function estimate* (ETFE).

In case u_t and e_t are correlated by feedback but an additional (reference) signal r_t is available that is uncorrelated with e_t, we would analogously obtain:

$$G(e^{i\omega_k}) = S_{yr}(\omega_k)S_{ur}(\omega_k)^R \tag{3-6}$$

Here, the superscript R denotes the right inverse, which we assume to be well defined. In practice, the SDFs in these equations are replaced with estimates, where similar averaging techniques are used as described in the *Spectral Density Function Estimation* section.

Identification from Impulse Response Data

Because several identification methods result in high-order or nonparametric model structures, model reduction is often required as an intermediate or final step. Methods based on impulse response coefficients, or *Markov parameters*, are particularly useful because they can be applied to parametric as well as nonparametric models. For instance, the impulse response can be obtained from a high-order model or by taking the inverse Fourier transform of an empirical transfer function estimate. We discuss an impulse-response-based realization algorithm (`irea( )`) with such options as the Zeiger/McEwen and Kung/Kailath method. We assume noise-free system equations of the form,

$$x_{t+1} = Ax_t + Bu_t$$
$$y_t = Cx_t + Du_t$$

and define G_k ($k \geq 0$) as the impulse response coefficients. These are defined by matrices within column j of the kth sample of an impulse response on the jth input. Given the system parameters, they are equivalently given by:

$$G_0 = D \text{ and } G_k = CA^{k-1}B,\ k > 0$$

The algorithm is based on a singular-value decomposition of a *Hankel matrix* filled with the Markov parameters $H_{\eta n}(k-1)$, defined by:

$$H_{\eta\nu}(k-1) = \begin{bmatrix} G_k & G_{k+1} & \cdots & G_{k+\nu-1} \\ G_{k+1} & G_{k+2} & \cdots & G_{k+\nu} \\ \vdots & \vdots & \vdots & \vdots \\ G_{k+\eta-1} & G_{k+\eta} & \cdots & G_{k+\nu+\eta-2} \end{bmatrix}$$

This matrix is the product of three terms,

$$H_{\eta n}(k) = V_\eta A^k W_n$$

where,

$$V_\eta = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{\eta-1} \end{bmatrix}$$

and:

$$W_v = [B \; AB \ldots A^{v-1}B]$$

Here, V_η and W_v are the extended observability and controllability matrix.

Let the SVD of $H_{\eta n}(0)$ be defined by:

$$H_{\eta v}(0) = \begin{bmatrix} U_1 & U_2 \end{bmatrix} \begin{bmatrix} U_1 & U_2 \end{bmatrix} \begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix} = U_1 \Sigma V_1^T \tag{3-7}$$

where Σ contains nonzero diagonal elements only.

We will show that the state-space matrices associated with the transformed state $\tilde{x}_t = Tx_t$, where

$$T = \Sigma^{-\frac{1}{2}} U_1^T V_\eta$$

are formulated entirely in terms of Markov parameters and can therefore be obtained from the impulse response data. First, notice that the inverse of T is given by:

$$T^{-1} = W_\eta V_1 \Sigma^{-\frac{1}{2}}$$

The proof is given by straightforward multiplication, using the fact that $V_\eta W_n = H_{\eta n}(0) = U_1 \Sigma V_1^T$. The transformed state matrices are shown in Equation 3-8:

$$\begin{aligned}
\tilde{A} &= TAT^{-1} = \Sigma^{-\frac{1}{2}} U_1^T H_{\eta,\nu}(1) V_1 \Sigma^{-\frac{1}{2}} \\
\tilde{B} &= TB = \Sigma^{-\frac{1}{2}} U_1^T H_{\eta,1}(0) \\
\tilde{C} &= CT^{-1} = H_{1,\nu}(0) V_1 \Sigma^{-\frac{1}{2}} \\
\tilde{D} &= D = Y_0
\end{aligned} \tag{3-8}$$

Here, we have used the equalities $V_\eta A W_\nu = H_{\eta,\nu}(1)$, $V_\eta B = H_{\eta,1}(0)$, $C W_\nu = H_{1,\nu}(0)$.

Equation 3-8 contains Markov parameters only and therefore constitutes a realization algorithm. This is the Zeiger-McEwen *approximate realization* algorithm [Zeig].

Another relationship used to compute the A matrix that is employed in the Kung/Kailath algorithm [Kung] is derived as follows: Define $\overline{U}_1$ and U_1 as the matrices consisting of the top block rows $2,\ldots,\eta$ and $1,\ldots,\eta - 1$ of U_1, respectively; then, by straight substitution it is easily seen that

$$\underline{U}_1 \Sigma^{\frac{1}{2}} \tilde{A} = \overline{U}_1 \Sigma^{\frac{1}{2}}$$

This equation determines $\tilde{A}$ uniquely and can be used as an alternative for Equation 3-8.

If the conditions are non-ideal and the impulse response data is corrupted by noise, Σ in Equation 3-7 will most likely occupy the whole matrix with several small nonzero singular values. In that case, you can set singular values that are smaller than a certain tolerance to zero after which the computation continues as described above. You can easily validate the result by comparing it with the original impulse response.

Remarks

- Depending on the size of the problem, the SVD might take a considerable amount of computation time. After you have obtained the SVD, however, you also can easily obtain approximations of several orders without recomputing the SVD.
- There is no clear interpretation of the effect of the approximation in terms of criteria like H_∞ norms. Thus, we can only conclude that the most significant components of the Hankel matrix are used. In practical applications, however, it turns out that `irea` is a very useful and practical tool.
- Singular value-based model reduction techniques are very sensitive to scaling in the non-SISO case.

Least Squares-Frequency Domain

You also can apply the least squares approach discussed in the *Least-Squares in the Time Domain* section to fit an ARX model to frequency response data obtained experimentally or through other identification routines using the `fwls( )` function.

The method exhibits the same computational simplicity and efficiency as `ls`. Suppose a frequency response G(z) is given and that we want to fit an ARMA frequency response, $\hat{G}(z) = A(z)^{-1}B(z)$, to it, where

$$A(z) = I + \sum_{i=1}^{n} A_i z^{-i}, B(z) = \sum_{i=0}^{n} B_i z^{-i}$$

To simplify the problem, we formulate an error Δ(z) that is linear in the parameters by:

$$\Delta(z) = A(z)G(z) - B(z) = A(z)(G(z) - \hat{G}(z)) \quad (3\text{-}9)$$

These equations can be written as shown in Equation 3-10.

Here, the variables $z_i = e^{j\omega_i}$ are the points on the complex unit circle corresponding to the frequencies ω_i of interest. The matrices X, Y, and E in this case are complex valued matrices so we must add the complex conjugate equations in order to obtain real A_i's and B_i's. The solution is done in the same way as in the time domain, the only difference being that we need a complex QR transform instead of a real one. Because the

structure is exactly the same as in Equation 3-2, we also can obtain lower-order models analogously.

$$\begin{bmatrix} : & : & : & : & : & : & : & : \\ z_i^{-1}I & z_i^{-1}G^T(z_i) & z_i^{-2}G^T(z_i) & .. & z_i^{-n}I & z_i^{-n}G^T(z_i) & I & G^T(z_i) \\ : & : & : & : & : & : & : & : \end{bmatrix} \begin{bmatrix} -B_1^T \\ A_1^T \\ -B_2^T \\ A_2^T \\ : \\ -B_n^T \\ A_n^T \\ -B_0^T \\ I \end{bmatrix} = \begin{bmatrix} : \\ \Delta^T(z_i) \\ : \end{bmatrix} \quad (3\text{-}10)$$

If the fit is not good enough in a certain frequency band, we can emphasize that part of the data by multiplying the corresponding block rows of Equation 3-10 with a weight function that has a large value in that region. Outside the emphasized frequency region, the fit usually displays a smooth roll-off. In the MIMO case, it is possible to weight each individual row; this is equivalent to using a different weight function for each input.

For scalar systems, the weight function

$$W(\omega_i) = \frac{1}{|A(z_i)|}$$

where A(z) is obtained from an earlier obtained unweighted solution, can be a good choice. The reason is that the approximation, which takes place in terms of frequency response, is weighted by A(z) according to Euqation 3-9. The result can be shaped interactively by changing the weight function. You can usually obtain a good fit after a few iterations.

Prediction Error Methods

The least-squares approach implemented in `ls` is an example of the more general Prediction Error Method (PEM). Here the objective is to minimize the criterion $J(\vartheta)$, defined by:

$$J(\vartheta) = \sum_{t=n+1}^{t=N} \varepsilon_t^T W \varepsilon_t \tag{3-11}$$

where ε_t is the *prediction error* or *innovation* determined by the parametric model structure:

$$y_t = \hat{G}(z)u_t + \hat{H}(z)\varepsilon_t \tag{3-12}$$

A statistically optimal result is obtained if we take W as the inverse of the variance Λ of ε_t, or, because we are dealing with an unknown system, an estimate of the variance.

Because the identification problem represented by Equation 3-11 and Equation 3-12 can be highly nonlinear, we are faced with much larger numerical problems than with the least-squares case. The standard implementation is based on the Gauss-Newton method. Here, the term Gauss refers to the statistical assumptions that lead to the quadratic criterion (Equation 3-11), and Newton refers to the optimization scheme that is based on a locally quadratic approximation of the criterion. Alternatively, the steepest descent method is a good alternative if the quadratic approximation does not work.

In any case, this type of optimization requires several iterations before the (local) minimum is obtained. Each iteration involves computation of Equation 3-11 and estimation of the gradient over the whole batch of data. This implies a numerically intensive procedure—the price to pay for the ability to use a more general model structure than ARX.

Estimation Algorithm

A quadratic local approximation of the criterion is:

$$J(\vartheta) = J(\vartheta_0) + \frac{d}{d\vartheta}J|_{\theta_0}(\vartheta - \vartheta_0) + \frac{1}{2}(\vartheta - \vartheta_0)^T \frac{d^2}{d\vartheta^2}J|_{\vartheta_0}(\vartheta - \vartheta_0)$$

The minimum ϑ_1 over ϑ is:

$$\vartheta_1 = \vartheta_0 - \left[\frac{d^2}{d\vartheta^2} J|_{\vartheta_0}\right]^{-1} \left[\frac{d}{d\vartheta} J|_{\theta_0}\right]^T$$

With the definition $\psi_t = \frac{d\varepsilon_t}{d\vartheta}$, this can be approximated by:

$$\vartheta_1 = \vartheta_0 - \left[\sum_t \psi_t^T W \psi_t\right]^{-1} \left[\sum_t \varepsilon_t^T W \psi_t\right]^T$$

The last term is a standard least-squares estimate, which can be implemented with Xmath's backslash operator—a Gauss-Newton update. In case the quadratic approximation is poor, a steepest descent gradient method is a better choice. For this, assume the linear approximation

$$J(\vartheta) = J(\vartheta_0) + \frac{d}{d\vartheta} J|_{\theta_0} (\vartheta - \vartheta_0)$$

If we pick a steepest descent update

$$\vartheta = \vartheta_1 = \vartheta_0 - \mu \left[\frac{d}{d\vartheta} J|_{\theta_0}\right]^T$$

where μ is the *step size*, the new criterion value becomes

$$J(\vartheta_1) = J(\vartheta_0) - \mu \left[\frac{d}{d\vartheta} J|_{\theta_0}\right]^T \left[\frac{d}{d\vartheta} J|_{\theta_0}\right]$$

This is guaranteed to be less than the original value under the assumption of linearity, which, of course, only holds in a small neighborhood. The `pem( )` function uses a combination of Gauss-Newton and steepest descent updates, using a variety of step sizes by iteratively cutting the step size in half for a specified number of times. The next parameter value is defined as the one that gives the best update in the sense of the criterion value, which is recomputed after each update. The whole procedure is then repeated with ϑ_0 replaced by ϑ_1, and so forth. The search procedure stops after one of the following:

- The criterion cannot be improved significantly, that is, beyond a specified improvement threshold.
- A specified maximum number of iterations has been reached.

Specialized Model Structures

For ease of use, the functions `oe( )`, `armax( )`, and `bj( )` have been created for prediction error identification of output error, ARMAX, and Box-Jenkins models, respectively—or, more precisely, their state space equivalents. They make use of `pem( )` internally.

Subspace Identification Methods

The subspace identification algorithms implemented in `sds( )` and `sst( )` identify state-space systems through geometrical concepts such as subspaces, projections, and angles; they exploit numerically robust algorithms, such as the QR and SVD decompositions. An idea central to these algorithms is the concept of a *state* in system identification.

An estimate of the state is first calculated as an intersection (or a projection) between past and future input-output data [LARI2, MOON, VODM1, VODM2]. State estimates identify a state-space model through a least-squares solution of a set of linear equations. This concept is illustrated in Figure 3-1 where the left side shows the subspace approach—first Kalman states and then system matrices; the right side is the classical approach—system first and then an estimate of the states.

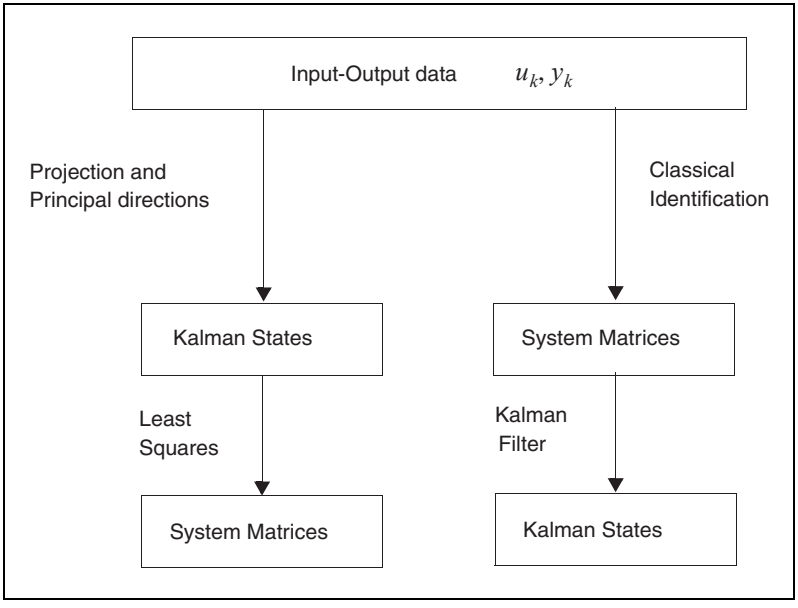

Figure 3-1. Least Squares versus Kalman Filter

Because subspace algorithms are part of a relatively new approach, this section provides a broad description of their characteristics. The differences between subspace identification and existing identification techniques are as follows:

- Choice of model order is based on singular values or angles between data spaces.
- Detailed parameterization data is not required; you need only the model order.
- Numerical search procedures are avoided by using QR and singular value decompositions.

Combined Deterministic-Stochastic Systems

The `sds( )` function for subspace deterministic-stochastic (SDS) systems estimates state-space innovation models as shown below. Assume that the input-output data is generated by the system:

$$x_{k+1} = Ax_k + Bu_k + w_k$$
$$y_k = Cx_k + Du_k + v_k$$

with

$$E\left[\begin{pmatrix} w_k \\ v_k \end{pmatrix}\begin{pmatrix} w_l^T & v_l^T \end{pmatrix}\right] = \begin{bmatrix} Q^s & S^s \\ \left(S^s\right)^T & R^s \end{bmatrix}\delta_{kl} \geq 0$$

where E denotes the expected value operator and δ_{kl}, the Kronecker index delta.

Defining the number of states as ns, the number of inputs as n_u, and the number of outputs as n_y, Q^s is ns × ns, S^s is $n_s \times n_y$, R^s is $n_y \times n_y$. v_k and ω_k correspond to unmeasurable white noise vector sequences with a zero-mean Gaussian distribution. {A,C} is assumed to be observable, while {A, (B $(Q^s)^{1/2}$)} is assumed to be controllable.

The `sds( )` function first determines the matrices A, B, C, D, Q^s, R^s, S^s and then converts them through an algebraic Riccati equation to the state-space innovation model object returned as the function output.

Subspace algorithms typically make extensive use of block Hankel matrices. Input and output block Hankel matrices are defined as:

$$U_{0|i-1} = \begin{bmatrix} u_0 & u_1 & u_2 & \dots & u_{j-1} \\ u_1 & u_2 & u_3 & \dots & u_j \\ \dots & \dots & \dots & \dots & \dots \\ u_{i-1} & u_i & u_{i+1} & \dots & u_{i+j-2} \end{bmatrix}$$

$$Y_{0|i-1} = \begin{bmatrix} y_0 & y_1 & y & \dots & y_{j-1} \\ y_1 & y_2 & y_3 & \dots & y_j \\ \dots & \dots & \dots & \dots & \dots \\ y_{i-1} & y_i & y_{i+1} & \dots & y_{i+j-2} \end{bmatrix}$$

The subscripts of U and Y denote the subscript of the first and last element of the first column. Furthermore, define Γ_i (the extended observability matrix) and H^d_i as:

$$\Gamma_i = \begin{bmatrix} C \\ CA \\ \dots \\ CA^{i-1} \end{bmatrix} \qquad H_i^d = \begin{bmatrix} D & 0 & \dots & 0 \\ CB & D & \dots & 0 \\ \dots & \dots & \dots & \dots \\ CA^{i-2} & CA^{i-3}B & \dots & D \end{bmatrix}$$

For simplicity of notation, let us introduce the following:

$$f = Y_{i|2i-1}, \qquad p = \begin{bmatrix} U_{0|i-1} \\ Y_{0|i-1} \end{bmatrix}, \qquad u = U_{i|2i-1}$$

The definition of projection of the row space of A onto the row space of B can be expressed in matrix form as A/B = A BT (B BT)$^{-1}$ B. Using these definitions, it can be proven that

$$\left[f/u^{\perp}\right]\left[p/u^{\perp}\right]\left[\left(p/u^{\perp}\right)\left(p/u^{\perp}\right)^{T}\right]^{-1} p = \Gamma_i \tilde{X}_i \qquad (3\text{-}13)$$

$$\left[\left(f/u^{\perp}\right)\left(f/u^{\perp}\right)^{T}\right]^{-1/2}\left[f/u^{\perp}\right]\left[p/u^{\perp}\right]^{T}\bullet\left[\left(p/u^{\perp}\right)\left(p/u^{\perp}\right)^{T}\right]^{-1}\left[p/u^{\perp}\right] \quad (3\text{-}14)$$
$$=\left[\left(f/u^{\perp}\right)\left(f/u^{\perp}\right)^{T}\right]^{-1/2}\Gamma_i\tilde{X}_i/u^{\perp}$$

Equation 3-13 is the same as the oblique projection of the row space of f along the row space of u on the row space of p [VODM1, VODM2]. The matrix $\tilde{X}_i$ denotes a sequence of non-steady-state Kalman filter states,

$$\tilde{X}_i = \left[\tilde{x}_i\ \tilde{x}_{i+1}\ \ldots\ \tilde{x}_{i+j-1}\right]$$

and thus contains information about both the deterministic and stochastic part of the state space innovation model. Equation 3-14 can be derived from Equation 3-13.

Determining the Observability Matrix and the Order

The determination of the observability matrix Γ_i and the order n from input-output data can be done on a *scaling-dependent* or *scaling-independent* basis. The term *scaling* refers to scaling the input and/or output data—for example, to correspond with different units or sensors. In `sds( )`, the scaling sensitivity is determined by the keyword `{basis}`, which can be set to `combined` or `unscaled`.

Dependent Scaling

Taking a singular value decomposition of Equation 3-13,

$$\left[f/u^{\perp}\right]\left[p/u^{\perp}\right]^{T}\left[\left(p/u^{\perp}\right)\left(p/u^{\perp}\right)^{T}\right]^{-1}p = \left[U_1^c\ U_2^c\right]\begin{bmatrix} S_1^c & 0 \\ 0 & S_2^c \cong 0 \end{bmatrix}\left[V^c\right]^{T} \quad (3\text{-}15)$$

we find that the number of singular values significantly different from zero (expressed as S_1^c) is equal to the model order. `sds( )` plots these singular values when `{basis="combined"}`. We also find from Equation 3-13 that Γ_i can be set equal to $U_i^c(S_i^c)^{1/2\,a}$.

The reason for this choice of Γ_i is that it determines the state-space basis of the identified model. With this basis, the system is in frequency weighted balanced form (frequency-weighting by the spectrum of the input) [Enns].

Independent Scaling

With the singular value decomposition of Equation 3-14,

$$\left[\left(f/u^{\perp}\right)\left(f/u^{\perp}\right)^{T}\right]^{-1/2}\left[f/u^{\perp}\right]\left[p/u^{\perp}\right]^{T}\bullet\left[\left(p/u^{\perp}\right)\left(p/u^{\perp}\right)^{T}\right]^{-1}\left[p/u^{\perp}\right]$$

$$=\left[U_1^u\ U_2^u\right]\begin{bmatrix} S_1^u & 0 \\ 0 & S_2^u\cong 0\end{bmatrix}\left[V^u\right]^{T}$$

we find once again that the number of singular values significantly different from zero (expressed as S_1^u) is equal to the model order. Moreover, these singular values are the cosines of the principal angles between $p/u^{\perp}$ and $f/u^{\perp}$, so they are all smaller than one. When `{basis="unscaled"}` in `sds( )`, these angles (the arc cosines of the elements of S_1^u) are plotted. The system order is equal to the number of principal angles significantly different from 90°.

We also find from Equation 3-14 that Γ_i can be set equal to

$$\left[\left(f/u^{\perp}\right)\left(f/u^{\perp}\right)^{T}\right]^{1/2}U_1^u\,(S_1^u)^{1/2}$$

It can be shown that scaling the inputs u_k or outputs y_k has no effect on the results. This algorithm is very similar to the algorithm in [LARI2].

All formulas, such as Equation 3-13 and Equation 3-14, are presented in a mathematically convenient way. The implementation within `sds( )`, however, uses numerically reliable methods (particularly, the QR decomposition and SVD) to evaluate the formulas.

Determining the State-Space System

The state-space system can be determined from the input-output data Γ_i and n using either an asymptotically biased or unbiased method. Both biased and unbiased algorithms are implemented in `sds( )`, but the biased algorithm often gives better results than the unbiased algorithm on real data. The keyword `{bias}` in `sds( )` determines which method is used. For a detailed analysis, refer to [VODM1, VODM2].

The unbiased approach is based on the following steps:

1. Determine the projections:

$$Z_i = Y_{i|2i-1} / \begin{bmatrix} U_{0|i-1} \\ U_{i|2i-1} \\ Y_{0|i-1} \end{bmatrix} \qquad Z_{i+1} = Y_{i+1|2i-1} / \begin{bmatrix} U_{0|i} \\ U_{i+1|2i-1} \\ Y_{0|i} \end{bmatrix}$$

2. Determine Γ_i and the order n as described before (sensitive/insensitive scaling). Make $\Gamma_{i-1} = \underline{\Gamma_i}$, where the underbar denotes deleting the last l (number of outputs) rows.
3. Determine the least-squares solution, where ρ_1 and ρ_2 are residuals:

$$\begin{bmatrix} \Gamma_{i-1}^{\dagger} Z_{i+1} \\ Y_{i|i} \end{bmatrix} = \begin{bmatrix} K_{11} K_{12} \\ K_{21} K_{22} \end{bmatrix} \begin{bmatrix} \Gamma_i^{\dagger} Z_i \\ U_{i|2i-1} \end{bmatrix} + \begin{bmatrix} \rho_1 \\ \rho_2 \end{bmatrix}$$

4. The system matrices are determined as follows:

$$A \leftarrow K_{11}$$
$$A \leftarrow K_{21}$$

B, D follow from A, C, K12, and K22 through a set of linear equations:

$$\begin{bmatrix} Q^S & S^s \\ \left(S^s\right)^T & R^s \end{bmatrix} \leftarrow \frac{1}{j} \begin{bmatrix} \rho_1 \rho_1^T & \rho_1 \rho_2^T \\ \rho_2 \rho_1^T & \rho_2 \rho_2^T \end{bmatrix}$$

Biased State-Space System Determination Method

The biased algorithm asymptotically calculates a slightly biased representation of the state-space model. The bias is zero if at least one of the following conditions is satisfied:

- The outputs are not corrupted by noise.
- The inputs u_k are independent, zero mean, white noise sequences.
- The block Hankel matrices are double infinite ($i \rightarrow \infty$).

A nonzero bias depends on the convergence (as a function of i) of the non-steady-state Kalman filter of the system that generated the data. Whenever i is large enough, the bias can be neglected [VODM1, VODM2].

This approach is implemented as follows:

1. Determine the oblique projection O_i as the projection of the row space of $Y_{i\,|\,2i-1}$ along the row space of $U_{i\,|\,2i-1}$ onto the row space of $(U^T_{0|i-1} Y^T_{0|i-1})^T$. This corresponds to Equation 3-13. Also determine the oblique projection O_{i+1} as the projection of the row space of $Y_{i+1|2i-1}$ along the row space of $U_{i+1|2i-1}$ onto the row space of $(U^T_{0|i-1} Y^T_{0|i-1})^{-T}$.
2. Determine Γ_i and the order n as described before (scaling sensitive or insensitive) Put $\Gamma_{i-1} = \underline{\Gamma_i}$, where the underbar denotes deleting as many rows as there are outputs.
3. Determine the states $\tilde{X}_i$ and $\tilde{X}_{i-1}$:

$$\tilde{X}_i = \Gamma_i^{\dagger} O_i \qquad \tilde{X}_{i+1} = \Gamma_{i-1}^{\dagger} O_{i+1}$$

4. Determine the least squares solution:

$$\begin{bmatrix} \tilde{X}_{i+1} \\ \hline Y_{i|i} \end{bmatrix} = \begin{bmatrix} L_{11} L_{12} \\ \hline L_{21} L_{22} \end{bmatrix} \begin{bmatrix} \tilde{X}_i \\ \hline U_{i|i} \end{bmatrix} + \begin{bmatrix} \rho_1 \\ \hline \rho_2 \end{bmatrix}$$

5. The system matrices are determined as follows:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \leftarrow \begin{bmatrix} L_{11} L_{12} \\ \hline L_{21} L_{22} \end{bmatrix}$$

$$\begin{bmatrix} Q^S & S^s \\ \left(S^s\right)^T & R^s \end{bmatrix} \leftarrow \frac{1}{j} \begin{bmatrix} \rho_1 \rho_1^T & \rho_1 \rho_2^T \\ \rho_2 \rho_1^T & \rho_2 \rho_2^T \end{bmatrix}$$

Subspace Identification of Stochastic Systems

The `sst( )` function for subspace stochastic (SST) identification estimates stochastic state-space models from output data. With white noise applied to the input of the identified system, the output generated has approximately the same second-order statistics as the original output data used in the identification. While, in principle, this would be possible by passing a zero input vector to `sds( )`, the stochastic identification problem

is implemented as a separate function for computational and practical reasons. This section provides a basic overview of the `sst( )` algorithm. For further details, refer to the references listed in the Appendix D, *Bibliography*.

Assume the input-output data is generated by the system:

$$x_{k+1} = Ax_k + w_k$$
$$y_k = Cx_k + v_k$$

with

$$E\left[\begin{pmatrix} w_k \\ v_k \end{pmatrix}(w_l^T \quad v_l^T)\right] = \begin{bmatrix} Q^s & S^s \\ (S^s)^T & R^s \end{bmatrix}\delta_{kl} \geq 0$$

and $A, Q^s \in R^{n \times n}$, $C \in R^{l \times n}$, $S^s \in R^{n \times 1}$, $R^s \in R^{l \times l}$.

The output vectors $y_k \in R^{l \times l}$ are measured. $v_k \in R^{l \times l}$ and $w_k \in R^{m \times 1}$, on the other hand, are unmeasurable, Gaussian distributed, zero mean, white noise vector sequences. {A,C} is assumed to be observable, while $\{A,(Q^s)^{1/2}\}$ is assumed to be controllable.

Define Δ_i as,

$$\Delta_i = (A^{i-1} \ldots AG \; G)$$

with

$$P = APT^T + Q^s$$
$$G = APC^T + S^s$$
$$\Lambda_0 = CPC^T + R^s$$

However, with

$$f = Y_{i|2i-1}, \; p = T_{0|i-1}$$

we find

$$\begin{bmatrix} f \\ p \end{bmatrix} = [ff^t]^{-1/2}[pp^t]^{-1}$$
$$p = \Gamma_i \tilde{X}_i \tag{3-16}$$

$$[ff^t]^{-1/2}[fp^t][pp^t]^{-1}p = [ff^t]^{-1/2}\Gamma_i\tilde{X}_i \tag{3-17}$$

[VODM1, VODM2].

Determining the Observability Matrix and the Order

Determining the observability matrix Γ_i and the model order n is the same for `sst( )` and `sds( )`. The scaling-sensitive case from Equation 3-16 corresponds to the unweighted principal component algorithm in [ARUN]. The scaling-insensitive case, Equation 3-17 corresponds to the canonical correlation approach of [VODM1, VODM2, and ARUN].

Determining the State-Space System for SST

To ensure positive realness of the identified covariance sequence, the system matrices are determined in a slightly asymptotically biased way. [VODM1, VODM2] describe a method to avoid this bias, but it does not necessarily lead to a positive real covariance sequence, which would imply that the innovation model cannot be calculated. Once again, as for `sds( )`, the bias is a function of the convergence (as a function of i) of the non-steady-state Kalman filter [VODM1, VODM2]. When i is large enough, the bias can be neglected. For most practical purposes, $i \geq 10$ is sufficient.

1. Determine the projections:

$$Z_i = \frac{Y_{i|2i-1}}{Y_{0|i-1}} \qquad Z_{i+1} = \frac{Y_{i+1|2i-1}}{Y_{0|i}}$$

2. Determine Γ_i and the order n as described in the *Determining the Observability Matrix and the Order* section (scaling sensitive or insensitive). Put $\Gamma_{i-1} = \underline{\Gamma_i}$ (where the underbar denotes deleting the last l (number of outputs) rows).

3. Determine the states $\tilde{X}_i$ and $\tilde{X}_{i+1}$ as follows:

$$\tilde{X}_i = \Gamma_i^{\dagger} O_i$$
$$\tilde{X}_{i+1} = \tilde{X}_{i+1}$$

4. Determine the least squares solution:

$$\begin{bmatrix} L_{21} \\ Y_{i|i} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} \tilde{X}_i + \begin{bmatrix} \rho_i \\ \rho_2 \end{bmatrix}$$

5. The system matrices are determined as follows:

$$\begin{bmatrix} A \\ C \end{bmatrix} \leftarrow \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} \qquad \begin{bmatrix} Q^s & S^s \\ \left(S^s\right)^t & R^s \end{bmatrix} \leftarrow R^s \begin{bmatrix} \rho_1 \rho_1^T & \rho_1 \rho_1^T \\ \rho_2 \rho_1^T & \rho_2 \rho_2^T \end{bmatrix}$$

Maximum Likelihood Method

The `maxlike( )` function uses a Karmarkar-type algorithm to vary the parameters until the output from the parameterized system model minimizes a least-squares cost function of the difference between the measured and computed outputs. Denoting the estimated output based on the parameter vector p by $\hat{y}(p)$, this cost function is:

$$\Phi(p) = \left\| \hat{y}(p) - y \right\|_2^2 \tag{3-18}$$

Φ(p) is re-evaluated using a MathScript function that returns the output y for successive perturbations of p. The gradient of the cost is then computed with respect to the changes in each of the parameters. This makes it possible to compute the Jacobian of the cost with respect to the parameters (approximating the Hessian, or second derivative matrix) and indicates which parameters are most important. The parameters are then updated, and the process is repeated until the maximum number of iterations specified has been performed, or a minimum cost value is obtained (`maxlike( )` reports convergence).

The problem given in Equation 3-18 is approached using a bounded “trust” region method to solve the quadratic suboptimization problem stated in Equation 3-20, thus obtaining a descent direction d. Using a linear

approximation for the function $\hat{y}(p_0)$ at the current parameter value p_0, we obtain:

$$\hat{y}(p) = \hat{y}(p_0) + J_0\{\hat{y}(p) - \hat{y}(p_0)\} \tag{3-19}$$

where J_0 is the Jacobian of $\hat{y}(p_0)$. The Jacobian is calculated by perturbing the parameter p by $\delta = \max(0.001, 0.001 * |p|)$, and then calculating:

$$J_0 = \frac{\{\hat{y}(p_0 + \delta) - \hat{y}(p_0)\}}{\delta}$$

To find a search direction d that points toward the measured output y from a given y_0, we can expand the expression as follows:

$$\Phi(d) = \left\|\hat{y}(p_0 + d) - y\right\|_2^2 = \left\|y_0 + Jd - y\right\|_2^2 \tag{3-20}$$

$$= \left\|y_0 - y\right\|_2^2 + 2y_0^T Jd - d^T J^T y + d^T Jd$$

and find a d that minimizes Φ. If we set,

$$\frac{d\Phi}{dd} = 2J^T y_0 - 2J^T y + 2J^T Jd = 0 \tag{3-21}$$

then it is evident that any d that minimizes Equation 3-20 must satisfy:

$$\left[J^T J\right]d = J^T\left[y_0 - y\right] \tag{3-22}$$

Unfortunately, Equation 3-22 does not have a solution d if $J^T J$ is singular. To avoid this problem, the following problem is solved instead:

$$\left[J^T + \mu I\right]d = J^T[y - y_0] \tag{3-23}$$

where I is an appropriately-sized identity matrix. Equation 3-23 always has a well-defined solution because $[J^T J = \mu I]$ is a positive-definite matrix given that μ is a positive scalar. It can be shown that the solution to Equation 3-23 is a descent direction for Φ [GMW].

Once a search direction d is calculated, one needs to decide how far to go in that direction. A line search based on the bisection method is used to calculate the step size. The use of a line search, combined with the

assurance that d is a descent direction, guarantees that the root sum of the squares of the output error (RSS)

$$RSS(p_k) = \left\| \hat{y}(p_k) - y \right\|$$

monotonically decreases as k increases. In turn, this guarantees that the algorithm will converge to a minimum, although this might only be a locally minimal solution.

Extensive scaling is used in the implementation to ensure that the parameters do not become extremely large and that the algorithm is overall numerically well-conditioned. Furthermore, the range of μ used in the implementation ($\mu = 0.01$) restricts the absolute changes in each parameter so that they do not exceed $k\left\| y - \hat{y}(p) \right\|$ at each iteration of k. This can be seen from Equation 3-23, defining $\sigma_n(A)$ as the smallest singular value of the matrix A:

$$\begin{aligned} \left\| d \right\| &= \left\| (J^T J + \mu I)^{-1} J (y - \hat{y}) \right\| \le \left\| J^T J + \mu I \right\| \left\| J^T \right\| \left\| y - \hat{y} \right\| \\ &\le \left(\frac{1}{\sigma_n \left[J^T J + \mu I \right]} \right) \left(\frac{1}{\sigma_n \left[J^T \right]} \right) \left\| y - \hat{y} \right\| \le \frac{100}{\sigma_n(j)} \left\| y - \hat{y} \right\| \end{aligned} \tag{3-24}$$

This limitation on the changes in the parameters keeps the optimization region limited.

If at all possible, it is good practice to get an idea of what the cost function RSS(p) looks like around the initial guess p_0. A convex cost function is much easier to minimize than a jagged cost function with many valleys, where `maxlike( )` is likely to yield a local minimum. Another way to avoid mistaking a local minimum for the global minimum is to try many different initial guesses and see if they all converge to the same solution.

A very common problem that even experienced users often encounter while solving an identification problem is the presence of redundant parameters in the model. This means that the problem's solution is not unique. One way to identify the occurrence of this kind of problem is to calculate the rank of the $J^T J$ matrix `maxlike( )` returns. If the matrix is singular, you should try to eliminate one or more unnecessary parameters.

4

Tutorial

This tutorial provides a hands-on introduction to the ISID command-line and interactive tools. Each section in this chapter deals with a specific identification approach and illustrates how to use the associated command-line and interactive tools.

Preparing to Use This Tutorial

Before beginning the tutorial, if you are a new user you should read the *Tutorial Data* and *Graphical User Interface* sections, which describe the general structure and use of ISID tools. Discussion of the interactive tools within the tutorial focuses on their algorithm-specific features. If you are not familiar with parameter dependent matrices (PDMs), you should first read the section on PDMs in *Xmath User Guide*. Although several ISID functions accept both PDMs and matrices as input parameters, PDMs are preferred because they contain additional information that is useful for simulation, plotting, and signal labeling. While we recommend that you peruse Chapter 3, *Identification Algorithms*, and the *Xmath Help* for more algorithm and syntax details, you do not need additional background on the algorithms to begin using the tutorial.

Because different identification methods are compared using the same data, this tutorial and the following chapter have the character of a mini introductory course in system identification.

Some general conventions are as follows:

- We do not present the full syntax of each function because it is available in the *Xmath Help*. The examples use the most common ways of calling these functions. When a particular operation can be accomplished either from the command line or through an interactive tool, an example is generally provided for both approaches. We focus in somewhat greater detail on the interactive examples, however, because the *Xmath Help* for each function provides additional command-line examples.
- Both in this chapter and elsewhere in the manual, we use acronyms for different algorithms. In several cases, these acronyms are used to represent both a mathematical approach and the function used to

implement it. The casing and font used are intended to clarify the reference. Acronyms are printed in upper case (LS, for least squares) and all function names are in `monospace` (for example, `ls( )`, the function based on the LS algorithm). All examples in this tutorial appear in `monospace` as well to indicate that they can be entered at the Xmath command line.

- The examples are oriented towards open-loop situations. The closed-loop case is much more complicated and requires various assumptions and conditions to be checked [AndGev], [Söd]. Empirical transfer function estimation, instrumental variables, and prediction error methods give asymptotically correct results if these conditions are met. Subspace-based solutions, however, do not.

Tutorial Data

You can run each section of the tutorial independently by first loading the tutorial data and then following the example text.

To obtain the data required, go to the command area and execute the demo file `tut_datgen.ms`:

```
execute file = "$XMATH/demos/tut_datgen.ms"
```

This takes a few minutes.

You should immediately save the data to a location you will remember. The following example suggests a file named `tut_data` in the current directory.

```
save file = "tut_data"
```

To load the data, go to the Xmath Commands window command area and issue the load command:

```
load file = "tut_data.xmd"
```

This command assumes that the current directory is the directory in which you saved the data.

The model used to generate most of the data is a discretized version of a two-mode mechanical system. An additional model is provided for the stochastic subspace identification.

The variables generated are as follows:

- `sys_true`—Discrete-time Xmath system representing the true system. `sys_true` has two measured outputs and two measured inputs. In addition to this, white measurement noise is added to each output.
- `g_true`—PDM representing the frequency response of the true (discretized) system `sys_true`, computed over the 256-point frequency range `f`.
- `f`—Frequency range vector for `g_true` in Hz.
- `dt`—A scalar value indicating discretization interval in seconds (equal to `period(sys_true)`).
- `y_prbs1`, `u_prbs1`, `y_prbs2`, `u_prbs2`, `y_ss1`, `u_ss1`, `y_ss2`, `u_ss2`—Four pairs of output/input data used in the tutorial (wide band PRBS input, low band PRBS input, sine sweeps on inputs 1 and 2). All are PDMs with two channels corresponding to the two measured outputs and the two measured inputs, respectively.
- `syssto_true`—A state-space innovations model (list object). `syssto_true` has two inputs and two outputs.
- `sdfsto_true`—The frequency response (spectral density function) of `syssto_true`.
- `covsto_true`—The 20-second impulse response (covariance) of `syssto_true`.
- `y_sto`—The output response of `syssto_true` to Gaussian (zero mean and unity variance) noise input.

Graphical User Interface

The Xmath programmable GUI is an interface layer that allows you to create customized interfaces for solving design problems or performing analyses. GUI-based interactive interfaces are available to the major identification functions: `ls( )`, `fwls( )`, `etfe( )`, `giv( )`, `irea( )`, `sds( )`, `sst( )`, and `val( )`. Each function has an optional `{gui}` keyword that you can specify to invoke an interactive GUI-based tool for that particular identification scheme. This section describes the general principles underlying the GUI as they apply to the system identification problem. For information on the standard interface used for all the ISID interactive tools, refer to the *General Features of ISID Interactive Tools* section.

Structure and Concept of the GUI

The *MATRIXx Help* provides detailed information on the structure of the GUI and its interface to Xmath, as well as a guide to programming GUI-based tools in Xmath.

When you instantiate a GUI-based identification tool, a special hidden partition with a name prefaced by an underscore is created and used to store all variables related to the identification. This means that even if you exit a GUI tool, you have the option of starting it up again later and continuing exactly where you left off because the data in the partition is not altered by non-GUI-based operations.

A brief description of the widgets used in this module is in order. Most users are familiar with *pull-down menus*, *push buttons*, and *toggle buttons*. *Sliders* resemble their real-world counterparts; use them for selecting a specific value from a preset range of values. The *VarEdit* widget is used widely in the ISID GUI tools. It resembles a push button with a label indicating a value; when selected with the mouse, it expands into a text entry area where you can type a new desired value.

Note You must press <Return> or <Enter> to update the widget and the associated Xmath variable; typing alone is not sufficient.

General Features of ISID Interactive Tools

All the interactive tools associated with identification routines have a similar interface. When you first invoke an interactive tool, a hidden partition named `_routineName_gui` is created to store variables pertaining to the tool (for example, `_ls_gui`, `_sds_gui`). The interactive tool for `ls` is shown in Figure 4-1.

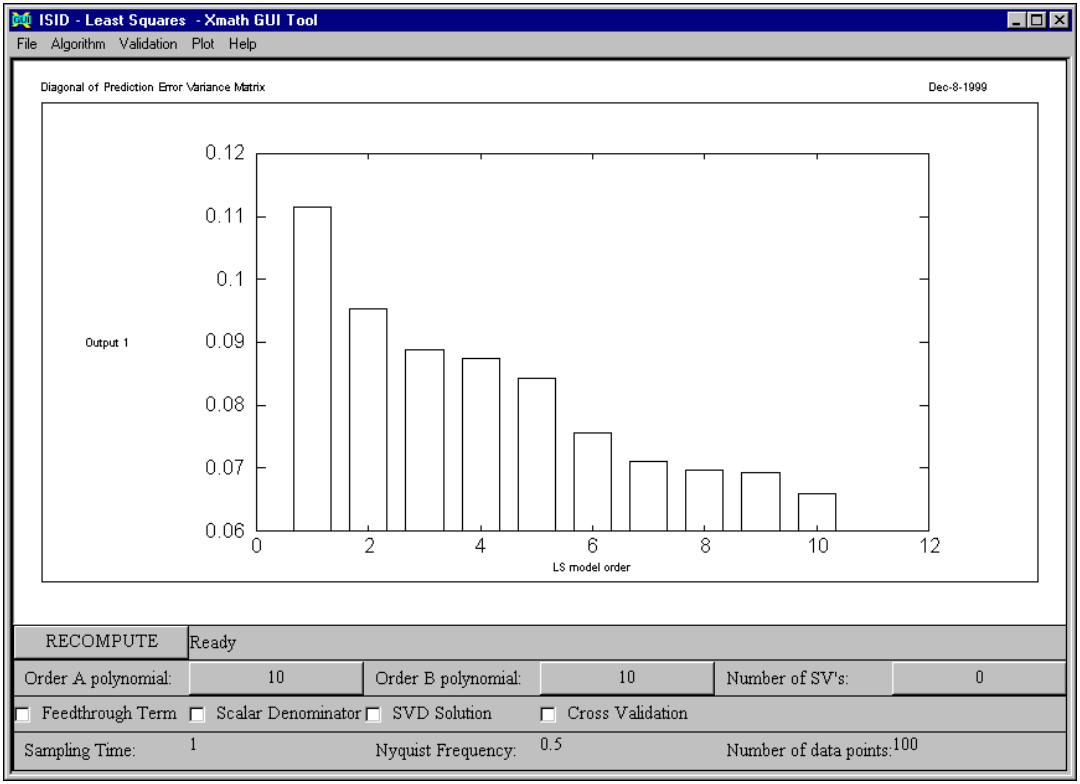

Figure 4-1. Interactive Tools for Is

The interactive tools consist of a menu bar near the top of the window, a plotting area, and additional modeling and validation selections below the plotting area.

Menus

Five menus (**File**, **Algorithm**, **Validation**, **Plot**, and **Help**) appear on the menu bar at the top of the window. The options under the **File**, **Validation**, and **Plot** menus are the same for all interactive tools, although some options may be grayed out if they are not relevant to the particular tool. The **Algorithm** options and the general help provided by the **Help** menu are specific to the identification function being used.

For a number of menu options, a shortcut key binding is provided. Key bindings are shown on the menus as ^key, where ^ represents the <Ctrl> key.

Note Shortcut key bindings are available on UNIX systems only.

File Menu

The **File** menu contains the following options:

- **Compare with Data**—Brings up a dialog for you to enter a variable containing data (PDM or matrix in Xmath) compatible with the data in the current plot. Clicking **OK** on the dialog displays both the original data and the comparison data; a legend indicating the line-data correspondence appears in the upper-left corner of the display. You can enter data in the current partition by variable name only; enter data in other partitions in the form *partitionName.variableName*.
- **Compare with Model**—Brings up a dialog for you to enter a variable representing an Xmath or ISID system variable with the same number of inputs and outputs. Clicking **OK** in the dialog causes the current results to be recomputed for the comparison model. Both the original and comparison-model-based results are then displayed with a legend. You can enter models in the current partition by variable name; enter data in other partitions in the form *partitionName.variableName*.
- **Save Plot»Xmath**—Brings up a dialog in which you enter a variable name. The data currently being displayed in the tool is then saved to the current partition under the name you choose. If you want to save the plotted data in another partition, specify the name under which to save it in the form *partitionName.variableName*.
- **Save Model»Xmath**—Brings up a dialog in which you enter a variable name. The most recently updated model within the interactive session is then saved to the current partition under the name you choose. If you want to save the model in another partition, specify the name under which to save it in the form *partitionName.variableName*.
- **Exit**—Closes and exits the interactive tool. Selecting **Close** from the window manager menu also exits the tool.

Validation Menu

The **Validation** menu contains the following options:

- **ViewInput-Output Data**—Plots the input and output data originally provided to the identification routine. This can be particularly useful for identifying spurious data points or outliers.
- **Prediction**—Plots, on the same axes, the predicted output from the input data and the actual measured output. A cascading submenu allows you to select whether you want to view the prediction output based on an input-output model or on an innovations model.
- **Prediction Errors**—Plots the difference between the predicted output from the input data and the actual measured output. A cascading submenu allows you to select whether to view the prediction output based on an input-output model or on an innovations model.
- **Covariance Pred. Err.**—Plots the prediction error covariance as an $(n_y \times n_y)$ matrix of plots. A cascading submenu allows you to select whether to view the prediction output based on an input-output model or on an innovations model.
- **Crosscorr. Input <–> Pred. Err.**—Plots the cross correlation of the input and the (output) prediction error as an $(n_y \times n_u)$ matrix of plots. A cascading submenu allows you to select whether to view the prediction output based on an input-output model or on an innovations model.
- **Frequency Response**—This option submenus allow you to select whether to compute the input-output frequency response of the identified model or the spectral density function of the output noise model. In either case, you can then view the magnitude, phase, or singular values of the response depending on your selection from the submenu.
- **Impulse Response**—Plots either the system impulse response as an $(n_y \times n_u)$ plot matrix or the covariance of the output noise model, depending on your selection from the submenu.
- **Poles and Transmission Zeros**—Provides a plot of pole-zero locations in the imaginary plane for either the input-output (deterministic) model or the noise (stochastic) model, depending on your selection from the submenu. The location of the unit circle is shown as a solid line with *'s denoting pole locations and o's denoting zero locations.

Caution If the function hits the memory limit, it will abort and re-identification of the data will be necessary.

- **Error Bounds**—Computes the one-sigma error bounds on the frequency response of the identified system and plots them along with the actual frequency response. This option may require a relatively large amount of memory and computation time. Therefore, it is not always possible to use it with large models.

Plot Menu

The **Plot** menu allows you to customize the appearance of the plot display in the interactive tool, as well as obtain a hardcopy. It contains the following options:

- **X-axis**—A submenu allows you to switch the axis scaling between linear and logarithmic.
- **Y-axis**—A submenu allows you to switch the axis scaling between linear and logarithmic.
- **Bar Graphs**—A **Yes/No** submenu option allows you to switch from standard line plots to bar graphs.
- **Date**—A **Yes/No** submenu option allows you to display or suppress the date stamp in the upper right corner of the display.
- **Texts/Hardcopy**—Brings up a small window containing VarEdit widgets. The first VarEdit, **Hardcopy to file**, takes the name of a file in which you want the PostScript hardcopy of the current display to be saved. When you enter the filename and press the <Return> or <Enter> key, the file is automatically generated and stored in the current Xmath directory. The next two VarEdits allow you to change the text appearing at the upper left and bottom of the plot. The last VarEdit, **GUI Plot Options**, allows you to pass a valid `uiPlot( )` option (for example, grid) to the plotted display in the tool. For more details on the valid `uiPlot( )` options, consult the *uiPlot* topic in the *Xmath Help*.

Help Menu

The **Help** menu on the right end of the menu bar brings up a page of Help text describing all the facilities available from the GUI tool.

Modeling and Validation Selections

A **RECOMPUTE** button and a status (**Ready**) appear beneath the plot area, directly above the set of identification options particular to the routine. When you modify one of these options through the menus, buttons, or VarEdits provided, you should click **RECOMPUTE** to begin the updated identification. At this point the status changes from **Ready** to a

status message indicating the type of recomputation taking place. When the computations are finished, the plot area is updated and status returns to **Ready**. A menu option to perform a recomputation also appears in the **Algorithm** menu.

Beneath the **RECOMPUTE** button and the status message is a set of options, toggle buttons, and VarEdit widgets that allow you to change the identification-specific parameters or options. These are discussed in conjunction with the particular identification functions.

The bottom line always shows the following parameters: sampling interval in seconds, the Nyquist frequency in Hertz, and the number of data points used for the identification.

You can close all windows using the default window manager menu, although all the interactive tools also have **Exit** options under the **File** menu. When you close an interactive tool, the partition that was created with it is not destroyed. Therefore, you can restart an interactive session by calling the identification function with empty parentheses and no input arguments; the data last used with the interactive tool is reloaded. If you start an interactive tool by calling the associated identification with new inputs, the information displayed in the tool reflects the model obtained from the new inputs. If an instance of the interactive tool for a function already exists, subsequent calls to that function with the `{gui}` keyword cause the first tool to close before the newest one is displayed on the screen.

The output of a function call including the `{gui}` keyword (bringing up an interactive tool) is fixed; the value of the output variable is not changed with later interactive modifications. Successive data and models obtained through an interactive tool are stored only in the *_routineName*_gui partition and can be overwritten with the next interactive change you make in the tool. To save the current model or data displayed in the tool, you must save it explicitly by selecting **File»Save Plot»Xmath** or **File»Save Model»Xmath**.

Graphics Utilities for GUI Tools

The graphics in the plot area are not based on the default Xmath graphics available from the command line, but are a GUI facility (as described in the *Xmath Help* under `uiPlot( )`). They are not interactive but can be modified substantially through the **Plot** menu options. Default features common to these and other `uiPlot( )`-based plots are the *magnifying glass* that appears over a portion of the plot when the middle mouse button is held down and the *data value indicator box* that appears when you

position the cursor on the plotted line and hold down the right mouse button. As you hold the right mouse button down and move along a line, the information about the data corresponding to that line is updated point by point. This is termed *data viewing*.

A *lasso plot* facility is available with all plots created with ISID functions. This allows you to zoom in on a particular section of the plot that you would like to see in more detail. To do this, press the <Ctrl> key and, with the left mouse button, click-and-drag a rectangular section within a given plot. (Light lines appear showing the region currently being selected.) When you release the button, the plot is regenerated showing a zoom plot of the selected area. Clicking the zoomed plot restores the normal view.

Another special plot facility is `mtxplt( )`. This function is a matrix plot utility particularly suited to plotting multivariable system data in an attractive format; all the ISID GUI tools call `mtxplt( )` internally. A typical use of `mtxplt( )` is to display the frequency response at each system output due to a particular signal at each input. In such a case, the matrix of plots has as many rows as system outputs and as many columns as system inputs; consequently, the plot in the (1,2) location corresponds to the response at output 1 of the system to input 2. If you hold the control key down and single-click on a subplot it expands for closer examination. Hold down the <Ctrl> key and click the enlarged subplot to re-display the full matrix of plots.

You also can call `mtxplt( )` from the command line using its numerous keywords to specify the plot appearance you want. In this case a separate `mtxplt( )` **Plot** window appears. `mtxplt( )` is implemented as a MathScript function so that while users may enter expressions as input arguments, it returns a `NULL` value. For this reason, If you call `mtxplt( )` without assigning its output to a variable, the message **`No match`** appears in the status area and **`ans is NULL`** is written to the output log.

Least-Squares in the Time Domain

The least-squares approach is implemented in the `ls( )` function. You can call `ls( )` completely from the command line, or you can use the associated interactive tool to change or validate the model. Other commands illustrated in this section are `ls2var( )`, `ls2unc( )`, and `lsjoin( )`.

If you have not already done so, load the data you created in the *Tutorial Data* section.

We're interested in the system response to a pseudo-random binary input signal (PRBS). We begin by invoking `ls( )` in this example with a maximum model order of 20. This means that all models of lower order (less than 20) can be obtained from the LS square root without having to re-identify the data. Define the data matrices, and call `ls( )` with a minimum number of arguments:

```
nmax = 20;
[sys20, sr] = ls(y_prbs2,u_prbs2,nmax)
```

Using the default noninteractive syntax for `ls( )`, this call returns **`sys20`**, a 20th-order Xmath system. The second output, **`sr`**, is the LS square root for model orders up to `nmax`. It also is represented as a list whose structure is discussed in Appendix A, *List Data Structures*.

As the model input and output data are processed, brief messages indicating how many data samples have been processed appear in the log area of the Xmath Commands window. This provides a status check when you are processing large amounts of data.

Using the same data, we now want to obtain the 10th-order least squares model:

```
n = 10;
[sys10] = ls(sr,n)
```

The desired model order (`n`) must not exceed `nmax`, the maximum order with which `sr` was created. Identifying a lower-order model from a previously-computed square root takes much less computation time than directly identifying models from input and output data.

Interactive LS Tool

We continue with the `ls( )` call, this time invoking the `{gui}` keyword to bring up the interactive tool. We also can use the `{yval}` and `{uval}` keywords to specify validation data sets to be used in conjunction with the tool. This validation set feature is unique to the `ls( )` function; it was implemented mainly because of the ease of computation using LS square roots. We use the wide-band PRBS data for validation:

```
sys20 = ls(y_prbs2,u_prbs2, 20,
     {gui,yval=y_prbs1,uval=u_prbs1});
```

The `ls( )` interactive tool comes up displaying the diagonal terms of the prediction error variance matrix for all orders up to the maximum (refer to Figure 4-2).

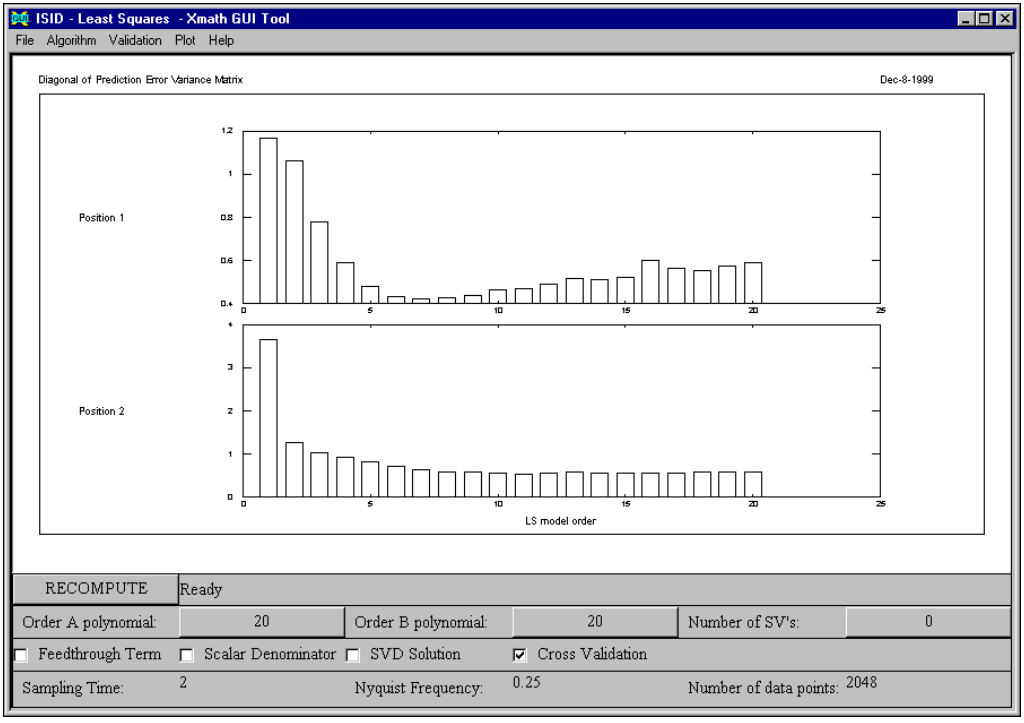

Figure 4-2. ls Error Variances

As described in the *Menus* section, only the options under the Algorithm menu are unique to the `ls( )` interactive tool; options on other menus are common to all tools.

In Figure 4-2, the error norms show a gradual decline as a function of ARX order. It is interesting to see what happens in terms of frequency response for increasing model order. To look at the frequency response for orders 4, 8, and 12, enter `[4,8,12]` for `Order A polynomial`; remember to press <Return> or <Enter>. (Xmath places this value in **Order B polynomial** also.) Then select **Validation»Frequency Response» Input - Output model (Magnitude)**. The new model is identified, and the response is plotted for each order as shown in Figure 4-3.

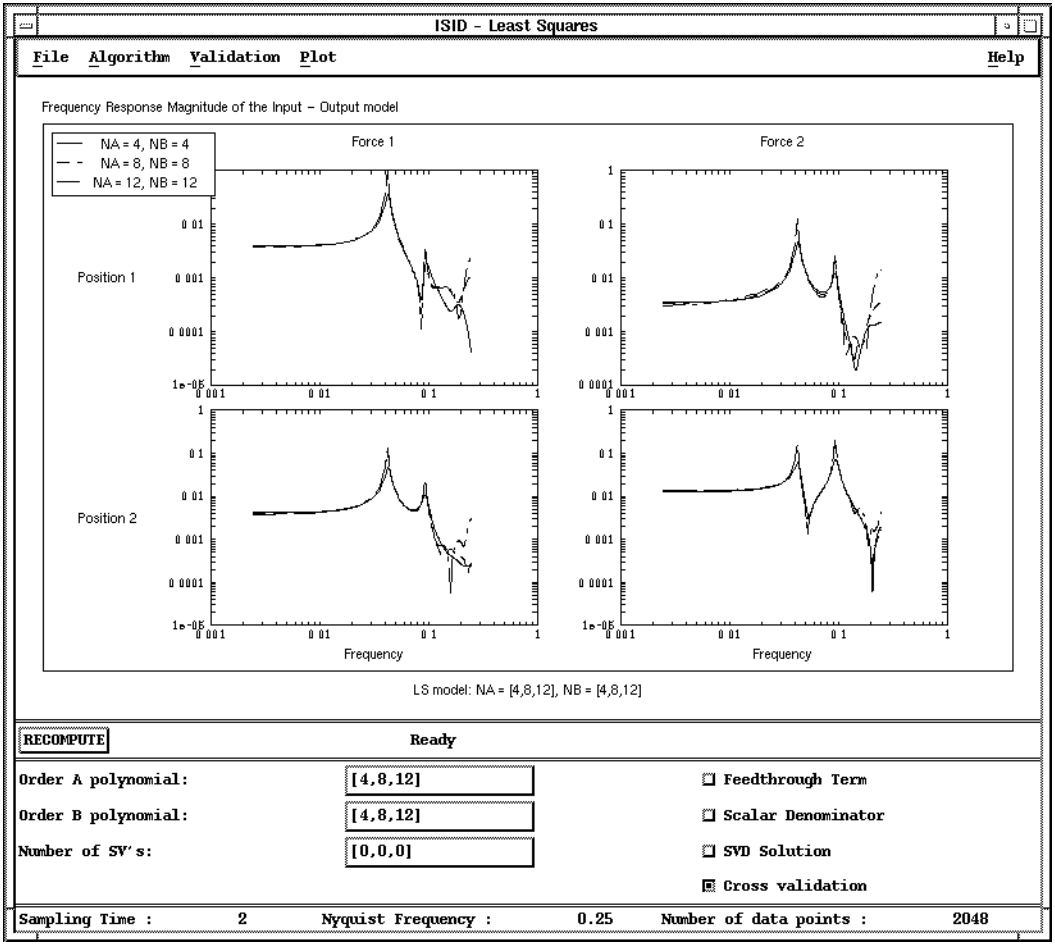

Figure 4-3. 4th, 8th, and 12th Order Model Frequency Responses

Judging from the difference in response for frequencies higher than 0.1 Hz, we might suspect that the fourth-order model does not have a sufficiently high order to model the noise. We can investigate that by comparing the spectral density function of the prediction errors of ARX models of orders 4 and 20. To do this, reset **Order A** (and **B**) **polynomial** to `4`, and then select **Algorithm»SDF prediction errors»Magnitude** to generate Figure 4-4.

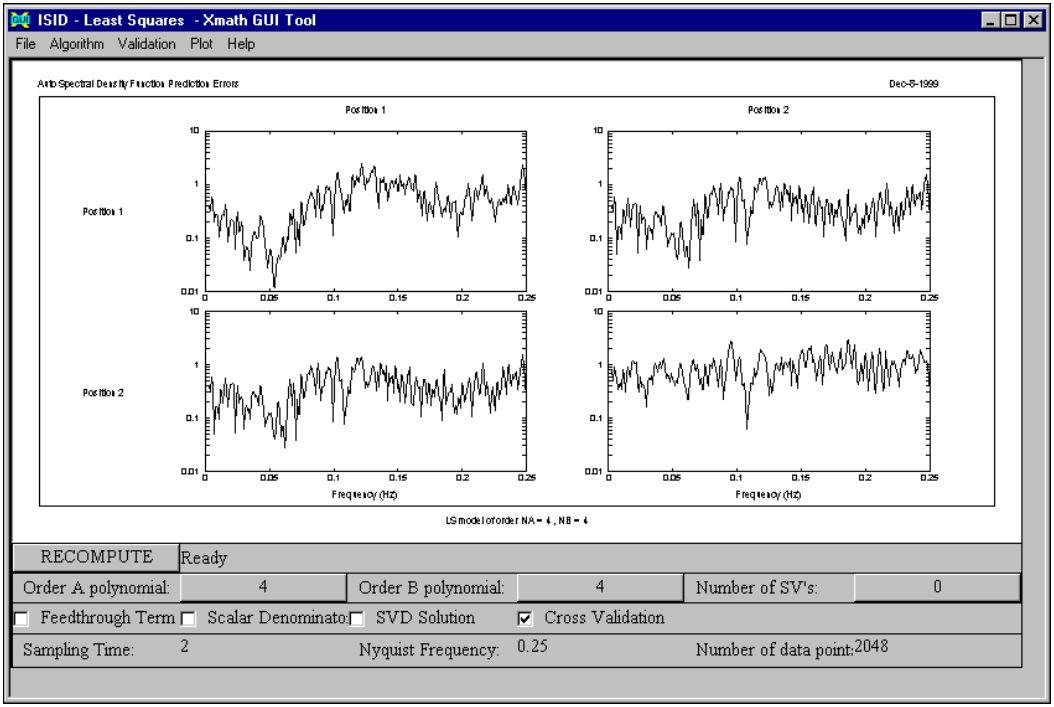

Figure 4-4. SDF Prediction Errors for Fourth-Order Model

Replace the `4` with `20`, and click **RECOMPUTE** to obtain the spectral density function of the prediction errors for the 20th-order model. (Vector input for the model orders is not supported with spectral density function plots and error bound plots for ease in reading the resulting plots.) We observe that the prediction errors of the 20th order have a much flatter SDF and are therefore whiter (plot not shown). Since the frequency responses do not change much beyond order eight, let us select the eighth-order model as our final candidate (entering `8` as the **Order A** (and **B**) **polynomial** values) and look at its frequency response uncertainty by selecting **Validation»Error Bounds**. This uncertainty is based on conversion of the parameter variance to a pointwise frequency response variance and can therefore be interpreted as a one-sigma bound.

The resulting plot, shown in Figure 4-5, displays three lines: the middle line is the 8th-order frequency response, while the others have the standard deviation added to and subtracted from it. The model error estimate is large for the higher frequencies, which agrees perfectly with the low coherences that you can see in the *Empirical Transfer Function Estimation* section.

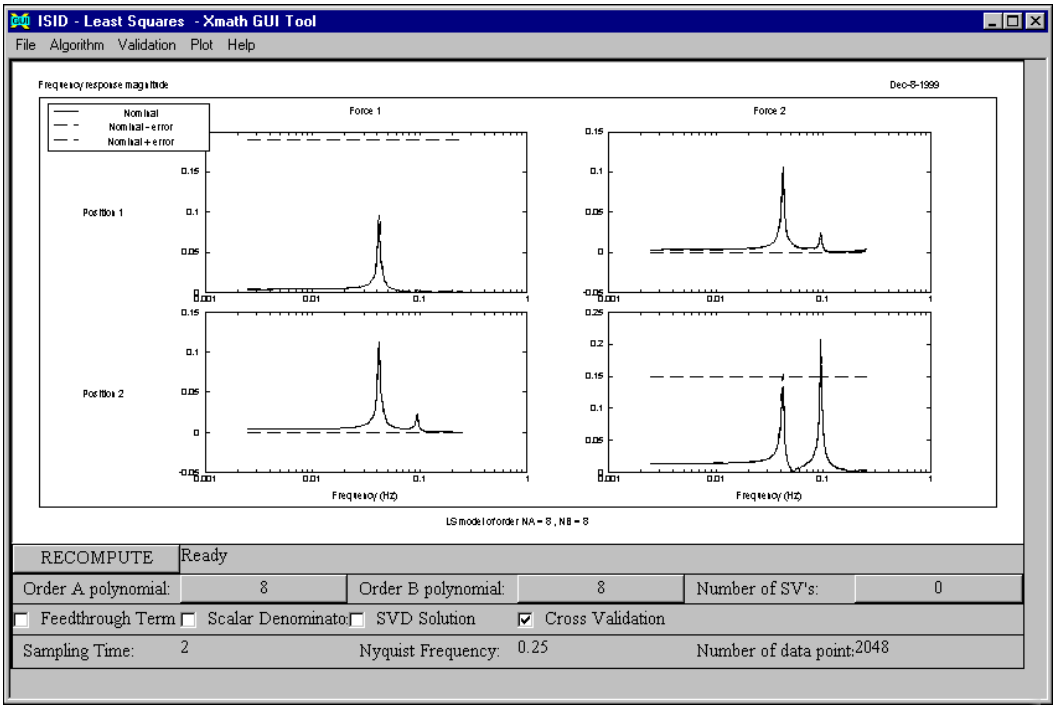

Figure 4-5. Eighth-Order Model Error Estimate

`_ls_gui.sys` is updated whenever you change or recompute the system model. When you exit the interactive tool by selecting **File»Exit**, `_ls_gui.sys` corresponds to the last model that was identified; in this case, it is an eighth-order model. (The original 20th-order system model returned by the call to `ls( )`, `sys20`, remains in your current working partition and is not affected by subsequent interactive modifications.) This, in turn, corresponds to a 16th-order state-space model. The model order can be reduced significantly by `fwls( )` or `irea( )` as we will see. We can store the eighth-order model for future use as a regular Xmath system; do this using the **File»Save Model»Xmath**, and save the model as `sys8`.

Because we have the true system model, we can now compare its frequency response to that of the eighth-order backward polynomial model. First, select **Validation»Frequency Response»Input - Output Model (Magnitude)**. Then select **File»Compare With Model** and specify `main.sys_true` as the model with which to compare. This action displays the magnitude frequency response of both systems with a legend. The comparison with the true model is shown in Figure 4-6. Notice how good

the fit is up to 0.1 Hz and that the mismatch beyond this frequency is consistent with the error band that we have seen before.

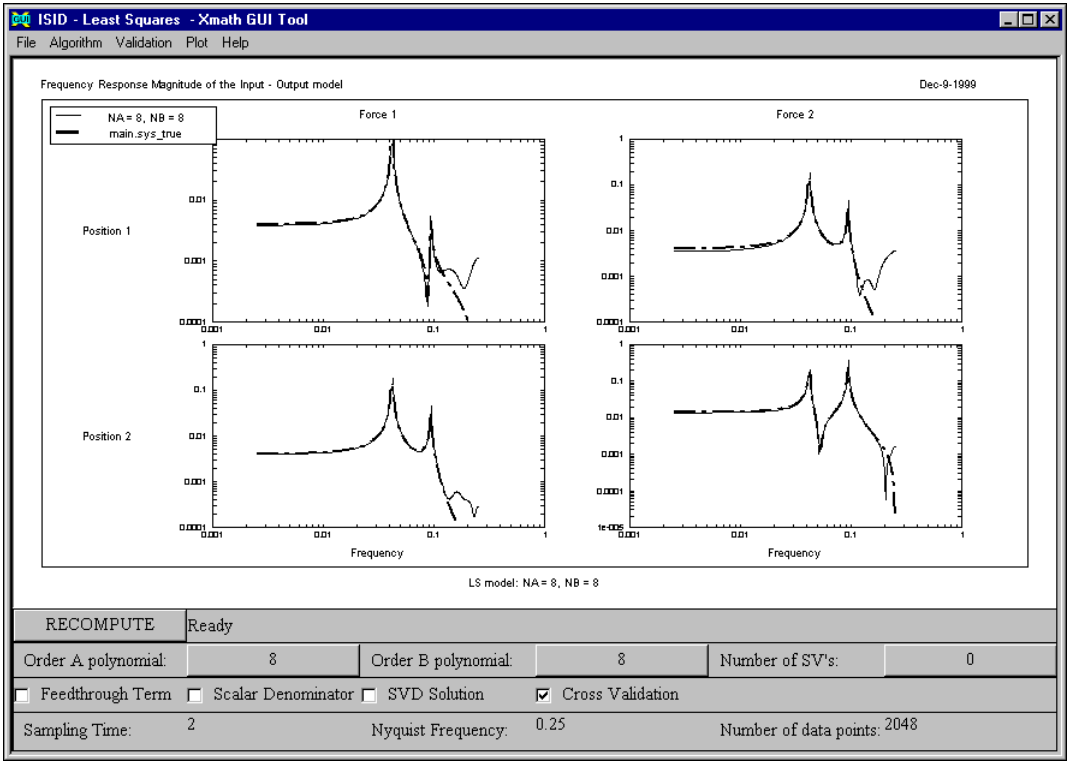

Figure 4-6. Comparison of Eighth-Order LS Versus True Models

Filtering

Often data filtering is useful to emphasize the data contents in a certain frequency band. This results in a better model quality inside that band. This section illustrates how to filter the data for our example, where we know that the bandwidth of the input sequence is 0.125 Hz.

Here we use the filter design functions included in the Xmath core to create an eighth-order lowpass Butterworth filter with no more than 0.1 dB ripple in the passband and a cutoff frequency of 0.125 Hz. For a complete listing of the available filter design functions, refer to the *Xmath Help*.

```
blpfilt = buttconstr({fixOrder=8,
   Fpass = 0.125,dt = dt, lowPass, dBpass=.1});
y_filt = filter(y_prbs2,blpfilt);
u_filt = filter(u_prbs2,blpfilt);
```

We can then perform the identification on the filtered data and compute its frequency response:

```
sys_filt = ls(y_filt,u_filt,4);
g_filt = freq(sys_filt,f);
```

Reusing the information from the 20th-order square root object `sr`, which was computed using unfiltered data, we can get the fourth-order response:

```
sys_unfilt = ls(sr,4);
g_unfilt = freq(sys_unfilt,f);
```

We compute the responses using `mtxplt( )`. The first instruction assigns a string to a variable name; the desired label is too long to fit in this document so we append two strings:

```
strng="4th-order ls filtered estimate (solid), "+...
   "true model (dashed), unfiltered estimate
(dot-dashed)"?

mtxplt([abs(g_filt), abs(g_true(1:2,
1:2)),abs(g_unfilt)],
   {y_log,columns=2,y_lab=["Output 1","Output 2"],
   x_lab=["Input 1","Input 2"],axtxt="Frequency (Hz)",
   bottxt=strng})?
```

This `mtxplt( )` function produces Figure 4-7, which illustrates that filtering can substantially reduce the model order required for a good fit.

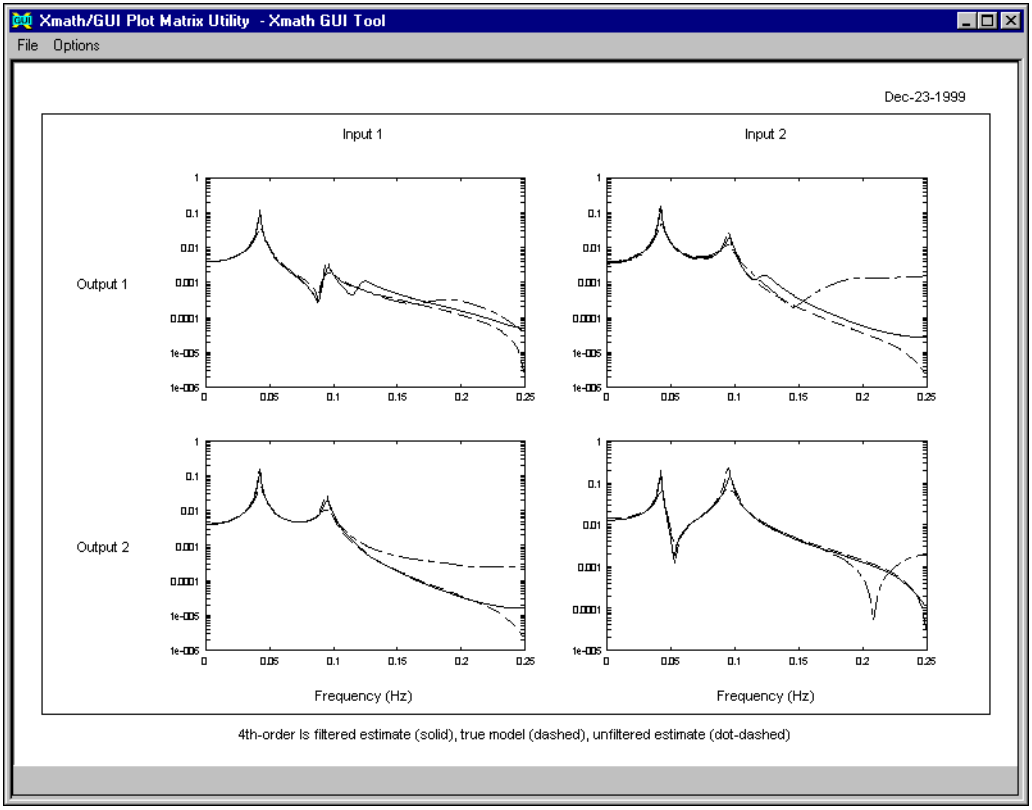

Figure 4-7. Comparison of Fourth Order LS (Filtered Data) Versus True Models

Square Root Based Cross Validation

The main problem with obtaining an LS model is choosing the order. When the order is too low, not all relevant dynamics can be modeled. When the order is too high, we are fitting parameters to noise, which causes the results to deteriorate significantly. Because `ls` is a prediction error method, we have to base the choice of order on the prediction error characteristics for different model orders. This can be done by looking at their norm in the first place. The function `ls2var( )` returns the norm of the prediction error variances (one for each output) from the square root object.

The syntax is `var = ls2var(sr)`.

This is the simplest form of using `ls2var,( )` where the prediction error norms are computed for the same data set that was used to obtain `sr` and the parameter estimates. We have seen the results already in Figure 4-2.

We also can get a graphical presentation of the value of the norm through the GUI.

If you exited the GUI at the end of the last section, you can restart it by typing:

```
ls()
```

Xmath automatically reloads the most-recently performed identification whose results are still in the `_ls_gui` partition. Selecting **Algorithm» Error norms** returns a plot of the value of the norms as a function of the model order up to the maximum model order listed in the square root object.

Whether you call `ls2var( )` directly or through the GUI interface, wherever possible model validation based on prediction errors should be done on a separate (statistically independent) validation data set. To see why this is necessary, consider the following.

As the model order increases, the prediction errors become smaller by definition of the least-squares estimate. They eventually become zero when we continue adding parameters. However, the parameters are fitted to noise and the model performs badly on any other data set.

To illustrate this phenomenon, consider the following example of `ls2var( )`, which is now used to produce the prediction error norms on a validation data set based on models from the identification data set. The computation takes place based on square roots only. We identify a model over the first 1,024 data points and validate them over the next 1,024, up to model order 20.

```
nmax = 20;[,sri]=ls(y_prbs2(1:1024),u_prbs2(1:1024),
nmax);
[,srv]=ls(y_prbs2(1025:2048),
u_prbs2(1025:2048),nmax);
var = ls2var(sri,{srval = srv})?
```

Looking at the two columns of `var` (one corresponding to each output) to see at what order the norms converge, the results are quite revealing. For the first output, the seventh-order model is optimal; for the second output, the ninth-order model is the best. An inspection of the frequency responses of these models indicates that these models are indeed the best ones. This type of cross validation is recommended because it can be done very efficiently and the model order is the most important `ls( )` parameter.

Alternatively, this step can be accomplished through the interactive tool using the validation data directly or the square root object for the validation data set (`srv`).

- If you already have the validation square root `srv`, type:

```
[sysi,sri] = ls(y_prbs2(1:1024),
u_prbs2(1:1024),20,{srval = srv, gui})
```

- If you do not have the validation square root, you can specify the input and output validation data explicitly by calling `ls( )` with the `{yval}` and `{uval}` keywords. You can only use these keywords in conjunction with the `{gui}` keyword and the interactive tool. This syntax is:

```
[sysi,sri] = ls(y_prbs2(1:1024),
u_prbs2(1:1024),20,{yval=y_prbs2(1025:2048),
uval=u_prbs2(1025:2048),gui})
```

The `ls` GUI pops up, displaying the error norms first.

Model Uncertainty Estimates

The model uncertainty displayed by `ls( )` can be obtained using the `ls2unc( )` function, which returns both the model frequency response and the one-sigma frequency response confidence intervals as a function of frequency. Here we call it with the square root object `sr` and an order of 8, as well as specifying a frequency range vector in Hz.

```
f_unc = 0.0025:(0.25-0.0025)/199:0.25
[g,deltag] = ls2unc(8,sr,f_unc)
```

The relative frequency response model error is then defined by `deltag/abs(g)`.

As described in the *Interactive LS Tool* section, we can use the **Validation»Error Bounds** option within the interactive tool to obtain a graphical display of the model frequency response with the standard deviation added to and subtracted from it.

Whether we call `ls2unc( )` from the command line or through the interactive tool, the results indicate that the relative model errors are large only for those frequencies corresponding to a small system response. We can therefore conclude that the model quality is good.

Combining Data Sets with lsjoin

The square root objects are particularly useful in cases where several data sets are available and you want to estimate one model based on all data sets. As an example, we can use the data sets `y_ss1`, `u_ss1` and `y_ss2`, `u_ss2`. These data sets were generated with a sine sweep on inputs 1 and 2, respectively. The sine sweep covers the frequency band from 0 to 0.125 Hz. Each of these data sets contains enough information to identify a model with one input only. Of course, we could identify two models and combine the resulting models in state-space form using the + system operator, but that would lead to a much higher dimensional state-space model than required. A better way is to compute the LS square roots of both data sets and combine them through a command-line call to the `lsjoin( )` function. In this particular call to `ls( )`, we omit the first `ls( )` output (a system model), since all we want is the second output (the square root object).

```
nmax = 8;
f_join = 0.0025:(0.25-0.0025)/199:0.25;
[,sr1] = ls(y_ss1,u_ss1,nmax);
[,sr2] = ls(y_ss2,u_ss2,nmax);
[sr12] = lsjoin(sr1,sr2)
```

To obtain the frequency response `g12` directly from the square root we need to re-create a system model using `ls( )` and then pass it to `idfreq( )`:

```
[g12,sdf_n] = idfreq(ls(sr12,nmax),f_join)
```

Within the LS interactive tool, we can then plot the frequency response of this eighth-order model together with the frequency response of the eighth-order LS model obtained earlier. Select **Validation»Frequency Response»Input - Output model (Magnitude)**. Select **File»Compare With Data**, and enter `g12` in the dialog. The results are characteristic for this type of input signal. The frequency response of the sine sweep model matches that of the pseudo-random data-based (PRBS) model well up to the bandwidth of the input and becomes much larger beyond that. This is apparently caused by the fact that the sine sweep contains no input power at all beyond 0.125 Hz, whereas the PRBS input still has some power in that region.

A frequently observed phenomenon is that high-order least-squares estimates overestimate the frequency response magnitude in those regions where the signal-to-noise ratio is poor. This holds for the PRBS model to a lesser extent as well, as we have seen from the comparison between the true system and the eighth-order LS model in Figure 4-6. The best way to obtain the final model is to perform model reduction based on those frequencies

where the signal-to-noise ratio is good. The *Signal Analysis* section provides further discussion on how we can obtain an impression of the signal-to-noise ratios. Frequency domain model reduction is covered in the *Least Squares in the Frequency Domain* section, which details the `fwls( )` function. Alternatively, you can use `irea( )` for impulse realization (refer to the *Impulse Realization* section). This function does not explicitly deal with frequency domain information but often gives good results.

SVD-Based Solutions

You can obtain a different kind of least-squares solution through the singular value decomposition (refer to the *Singular Value-Based Solutions* section of Chapter 3, *Identification Algorithms*). This option is implemented in `ls( )` through the `{nsvd=n}` keyword. With this syntax, `nsvd` should be set equal to the number n of singular values you plan to keep. The n largest singular values are retained and used in the solution while the rest are set to zero. The `ls` interactive tool provides an effective graphic to display the singular values and can be useful in determining how many you should retain.

To illustrate using the PRBS data:

```
[syssv] = ls(y_prbs2,u_prbs2,8,{nsvd = 10, gui})
```

When the interactive tool appears, the SVD solution toggle button is enabled. Select **Algorithm»SV Selection** to bring up a bar plot showing the singular values. You can then change the number of singular values to be retained, modify the **Number of SV's**, and then click **RECOMPUTE**.

Least Squares with Scalar Denominator

Using the `{scden}` keyword, you can specify that a system model with a scalar denominator be used; for more details, refer to the *Least Squares with Scalar Denominator* section of Chapter 3, *Identification Algorithms*. To illustrate briefly using a fourth-order model:

```
syssd = ls(y_prbs2,u_prbs2,4,{scden})
```

Examining the polynomial of the `syssd` list with

```
syssd(5)
```

illustrates that the denominator matrix polynomials are structured as scalar multiples of identity matrices.

To recompute a model currently displayed through the `ls` interactive tool, enable the **Scalar Denominator** checkbox, and click **RECOMPUTE**.

Lattice-Based Least Squares

Use of the `{lattice}` keyword in `ls( )` indicates that a special lattice-based algorithm should be used rather than the approach described in the *Least Squares for ARX Models* section of Chapter 3, *Identification Algorithms*. In spite of the efficiency of the default `ls( )` algorithm, this alternate method is valuable because we are often forced to identify high-order models due to the limited noise modeling capacity of the default `ls( )` algorithm. If we define n as the ARX model order, then the computational work of `ls( )` is proportional to n^2. For large problems, the lattice algorithm offers an order n alternative. It requires significantly less computational work than does the default `ls( )` algorithm for high model orders and may be the only practical algorithm for large problems that exceed your memory limitations. It computes only the top block row of the square root object. From this matrix, the square root can be recomputed along with the model parameters.

All other `ls`-associated options and keywords discussed in this section, including the interactive syntax, can use the `{lattice}` keyword. Square root objects generated with both `ls` algorithms are compatible and can be combined using `lsjoin( )`.

Although the lattice functions have all been implemented in square root form, there might be numerical problems associated with downdate operations. In that case, warnings are issued, and the results may be unreliable. However, in most cases with full-rank additional output noise, this does not happen and the lattice-based models are just as good as those obtained by the default algorithm.

To illustrate:

```
[sys8,sr] = ls(y_prbs2,u_prbs2,8)
[sys8lat,srlat] = ls(y_prbs2,u_prbs2,8,{lattice})
```

Select **File»Compare With Model** option to load in `sys8lat` while viewing the frequency or impulse response in the interactive `ls( )` tool. The response of the system models is indistinguishable.

Subspace Identification of Deterministic-Stochastic Systems

The subspace identification function `sds( )` returns a state-space model from input and output data. Refer to the *Subspace Identification Methods* section of Chapter 3, *Identification Algorithms*, for a general discussion of subspace methods; `sds( )` is described in more specific detail in the *Combined Deterministic-Stochastic Systems* section of Chapter 3, *Identification Algorithms*.

If you have not already done so, load the data you created in the *Tutorial Data* section. Call `sds( )` with the `{gui}` keyword to invoke the associated interactive tool:

```
[sys_sds,sr_sds] = sds(y_prbs2,u_prbs2,{gui})
```

Alternatively, if you call `sds( )` without the `{gui}` keyword and specify no model order, a bar plot of singular values is generated in the Xmath Graphics window, shown in Figure 4-8, and a popup queries you for the order of the model to be identified.

The `sds( )` interactive tool displaying a bar plot of the system singular values similar to Figure 4-8 appears. You can input **Model Order** to regenerate the bar plot of singular values or principal angles shown when the interactive tool is first instantiated. The **Model Order** is initialized to 0. Examining Figure 4-8, we see that there are four dominant singular values and thus enter `4` as the **Model Order**.

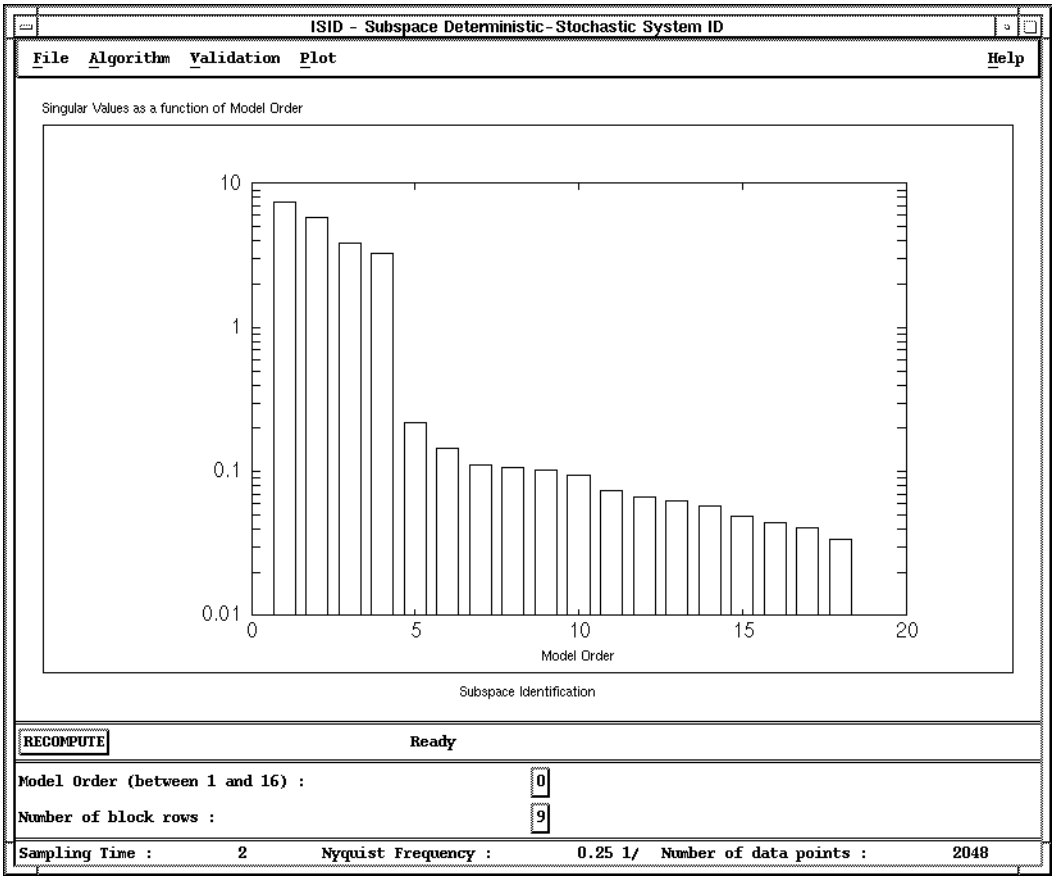

Figure 4-8. Subspace System Singular Values

By selecting **Validation»Frequency Response»Input - Output model (Magnitude)**, you generate the frequency response magnitude of the model. To compare the identified model to the true model, select **File»Compare with Data** and specify `g_true` in the dialog. The results are shown in Figure 4-9. Observe that the fit is very good for low frequencies.

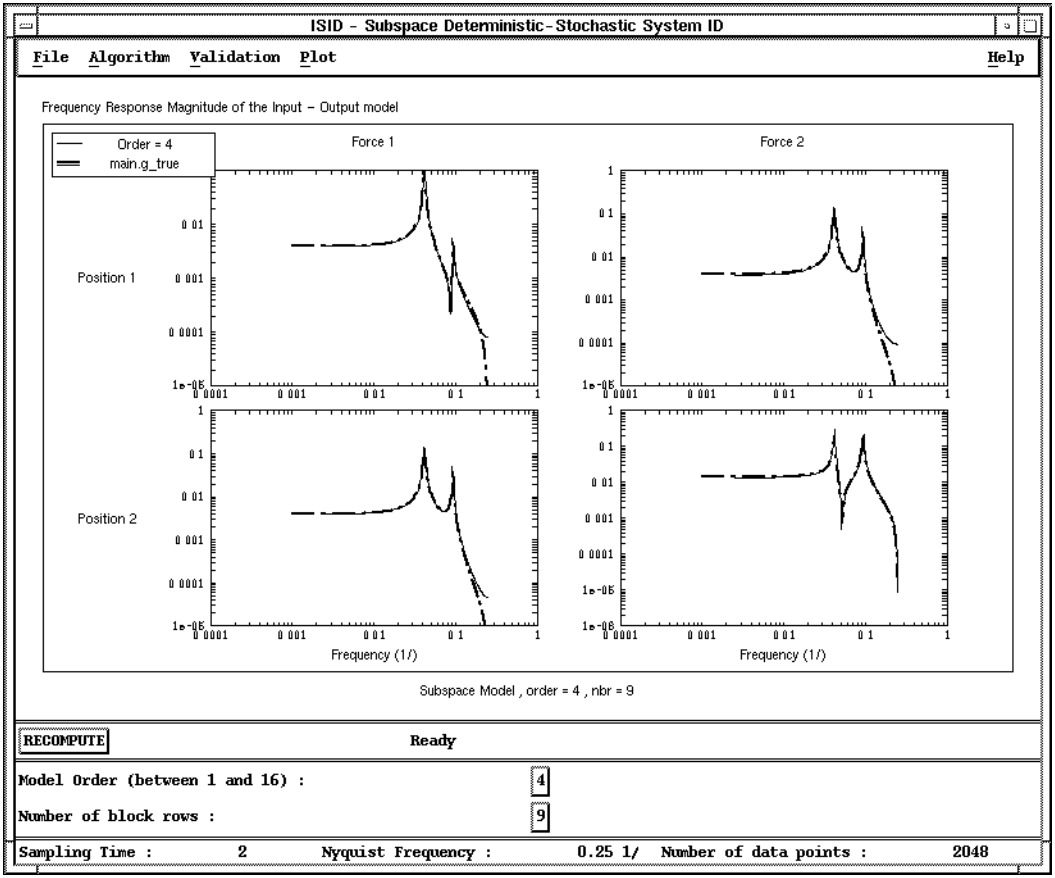

Figure 4-9. Comparison of Fourth-Order Model Frequency Response and True System Frequency Response

Like `ls( )`, `sds( )` optionally generates a square root (`sr_sds`). In fact, the square root object from `sds( )` is identical to that created by `ls( )` (with $n_{Ar} = n_B = 2 \times nbr_{sds} - 1$, where n_A and n_B represent the order of the A and B polynomials and nbr, the number of block Hankel rows). This means that square roots from `ls` can be passed directly to `sds` and vice-versa. You can easily generate all models of order less than $n_y \times (nbr - 1)$, where n_y is the number of system outputs.

To identify a second-order model on the same data, change the **Model Order** to `2` and click **RECOMPUTE**. This identification proceeds more quickly than the initial one due to using previously obtained intermediate results internally. Compare the model with the true model by selecting

File»Compare With Model. When the dialog appears, supply the name `sys_true`.

The results are shown in Figure 4-10. We see that the low frequency mode is modeled, while the higher frequency mode is neglected. When the chosen order is smaller than the real order (undermodeling), `sds( )` closely fits ranges corresponding to ranges where the input energy is the largest, as in classical identification.

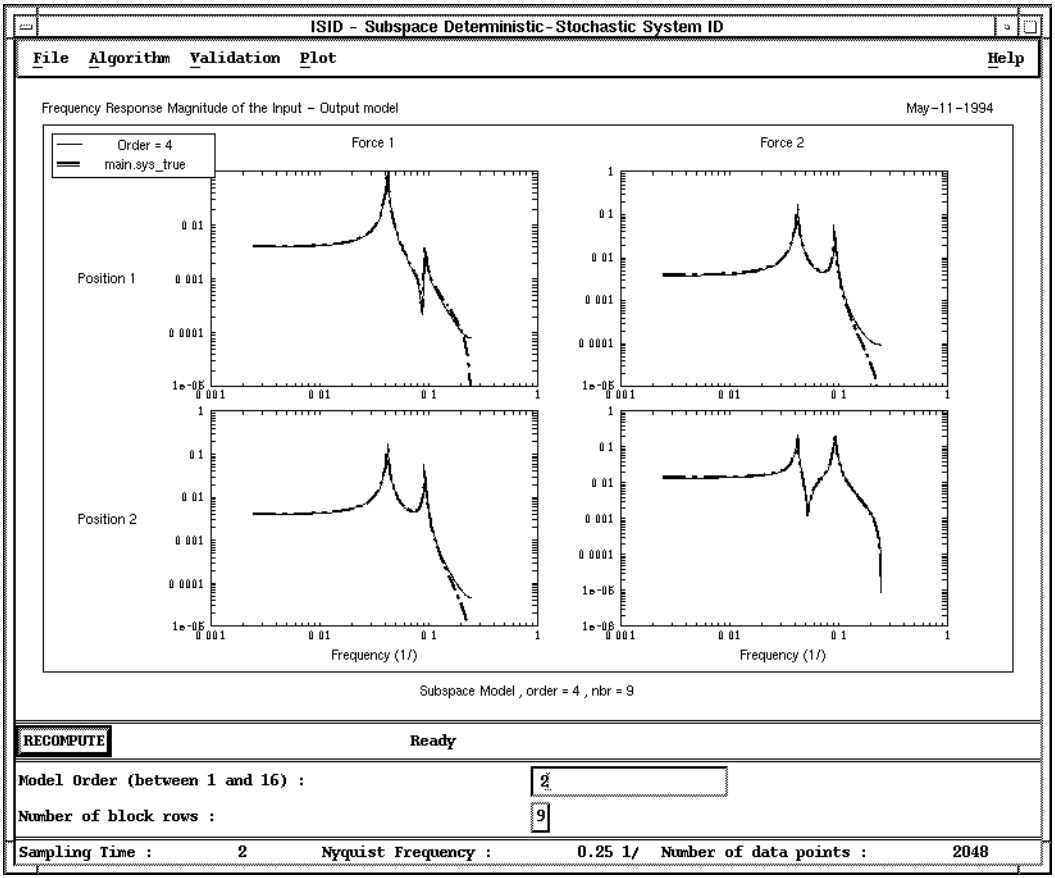

Figure 4-10. Second-Order Model Versus True Model

Now look at some other features:

```
void=sds(y_prbs2,u_prbs2,
{nbr=10,basis="unscaled",gui,lattice})
```

The keyword `{nbr}` indicates the number of block rows in the block Hankel matrices. Refer to the more detailed information in the *Subspace Identification of Stochastic Systems* section of Chapter 3, *Identification Algorithms*, or refer to the *sds* topic in the *Xmath Help*. This number is typically between 10 and 20. The number chosen for `nbr` should not be too large because the computational time is proportional to `nbr`. Generically speaking, the maximal order that can be selected is equal to $(\texttt{nbr} - 1) \times n_y$; For good results, however, the specified order should not exceed half this number, although `sds( )` allows you to go higher.

This time the interactive tool comes up displaying a bar plot of principal angles, shown in Figure 4-11, due to the use of the keyword `{unscaled}`. It is clear that the system order is 4 because there are four principal angles with values significantly different from 90°. Enter this value for the order in the **Model Order** to calculate a fourth-order (stochastic) model.

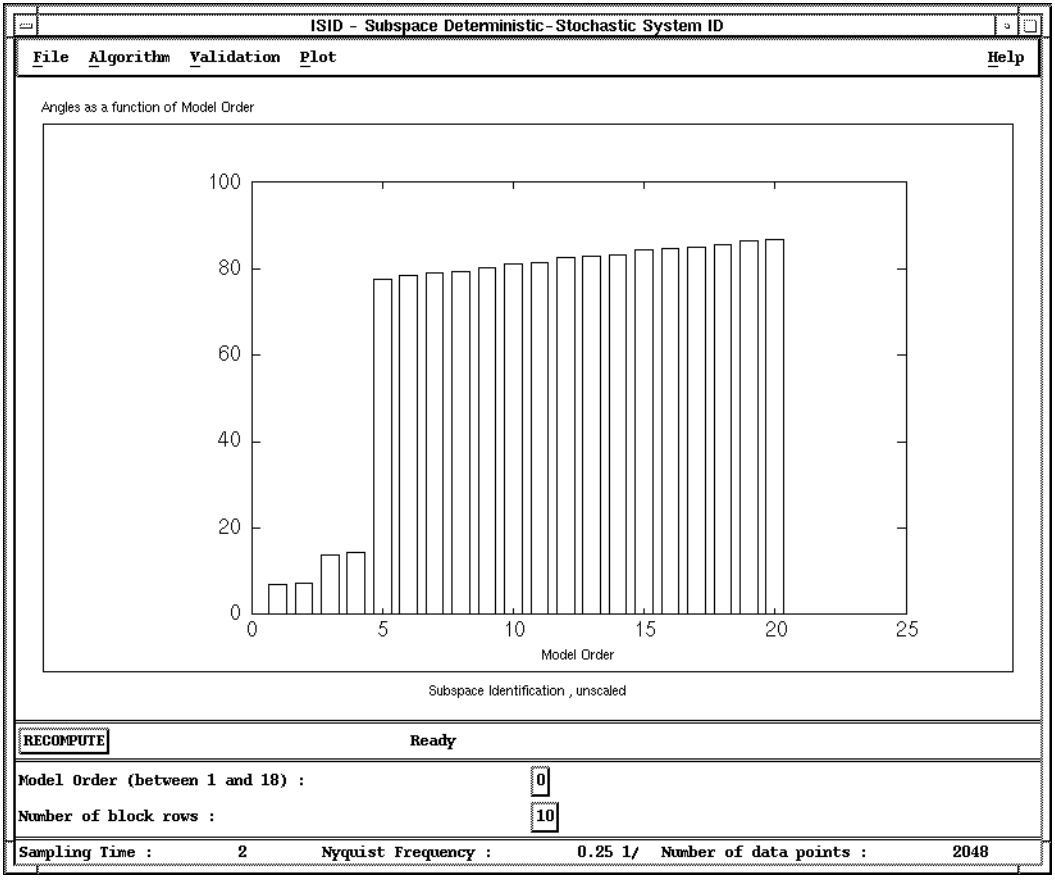

Figure 4-11. Principal Angles as Functions of Model Order

Another approach to validating this system involves selecting the **Validation»Covariance Prediction Error»Innovations Model** to examine the quality of the stochastic model. As shown in Figure 4-12, this produces a plot of the prediction error and a 95% confidence level. In general, the innovations model predictions errors should be white unless the data was generated by an output error system model. Clearly, the covariances are within the 95% confidence bounds.

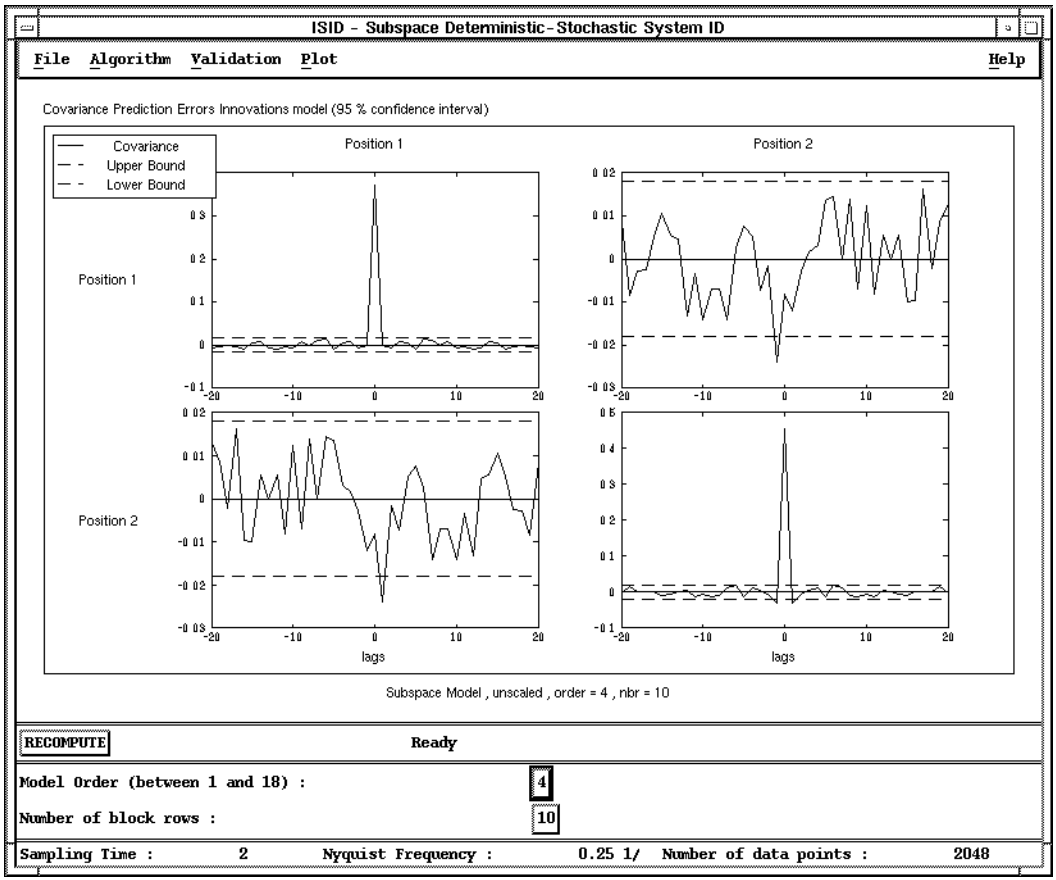

Figure 4-12. Covariance Prediction Errors for Fourth-Order Innovations Model

To examine a similar criterion for the prediction error covariance of the deterministic model, select **Validation»Covariance Pred. Err.» Input-Output model** and thus generate Figure 4-13. Notice that the residuals have a coloring roughly equal to that introduced by the stochastic subsystem. It can be concluded that it is not too important that the 95% intervals are violated by the deterministic prediction error covariances. These covariances provide a useful check of the importance (in a sense of coloring) of the stochastic model.

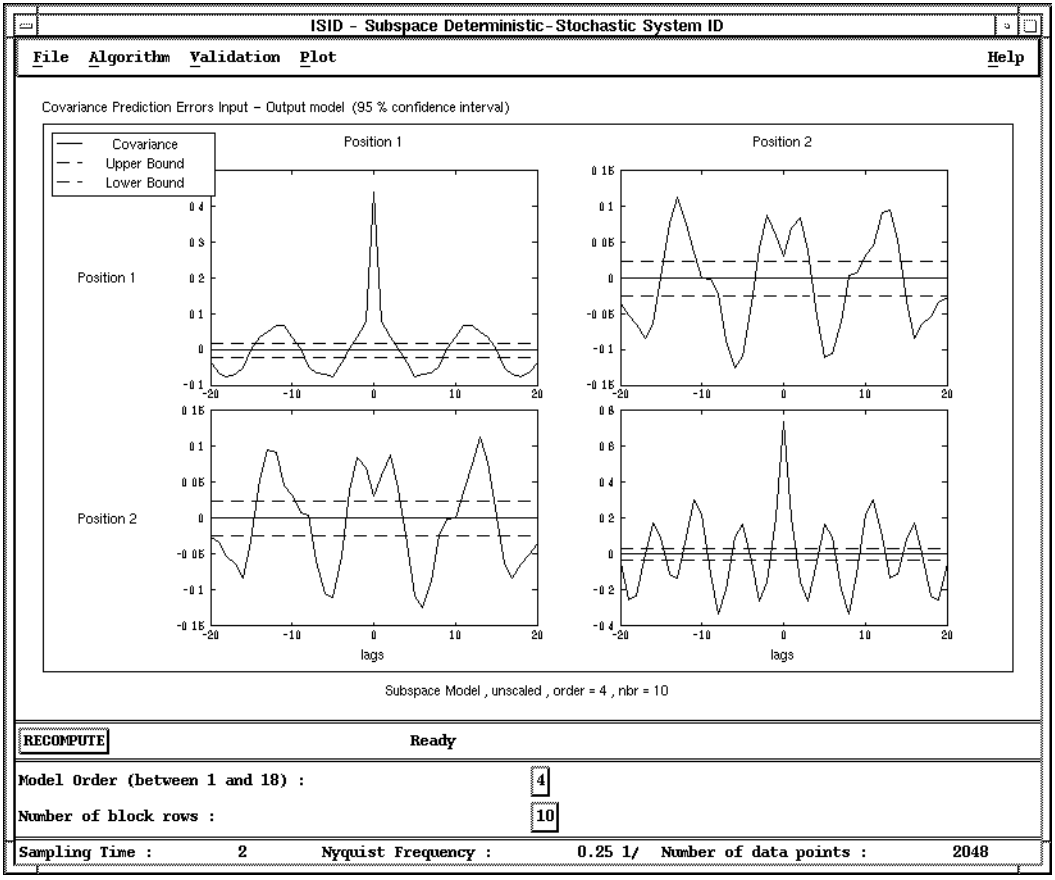

Figure 4-13. Prediction Error Covariance for the Fourth-Order Input-Output Model

The **Validation»Crosscorr. Input <-> Pred. Err.** is a useful technique to detect undermodeling. Selecting the **Input-Output Model** submenu option generates Figure 4-14.

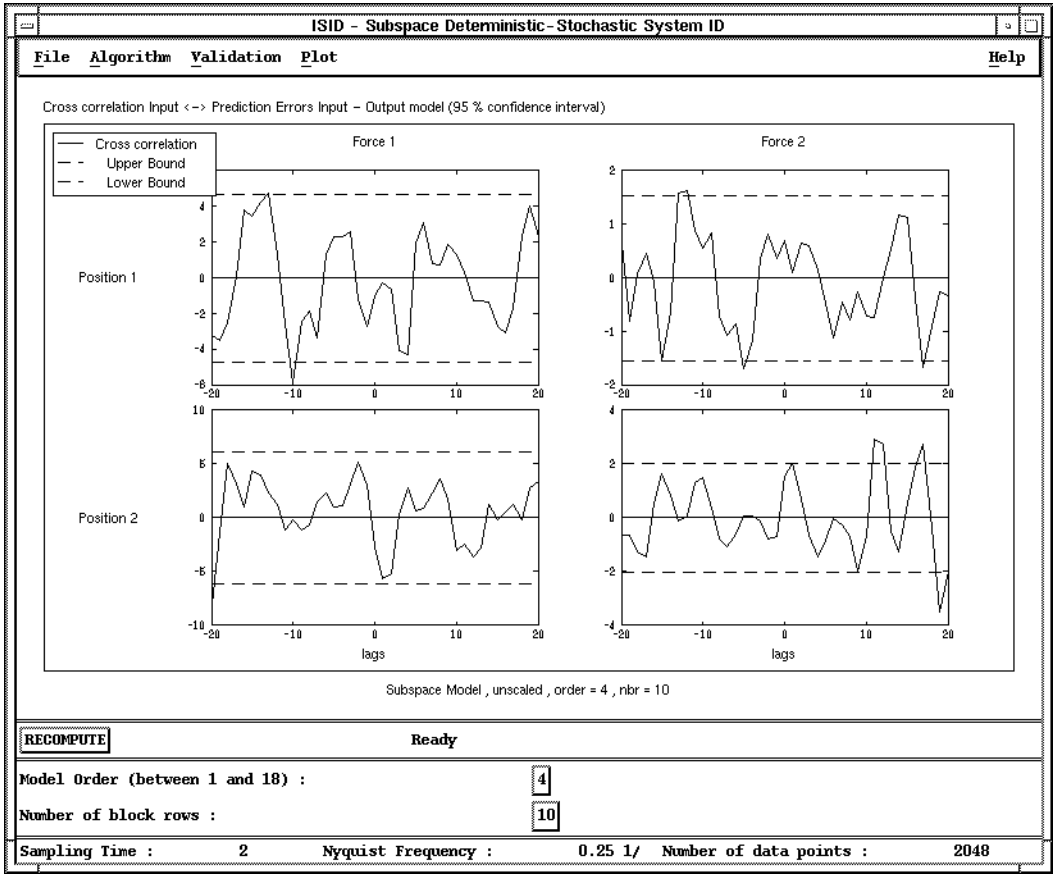

Figure 4-14. Cross-Correlation of Input and Prediction Errors for the Fourth-Order Model

Normally, the residuals from the input-output (deterministic) simulation should be independent of past inputs. If there is still a significant correlation for positive lags outside the 95% confidence bounds, then this would indicate undermodeling. It basically indicates that there is still energy in the residuals that could be explained by using the inputs. Even though there is still some significant cross correlation, it is a lot worse for n = 2, as shown in Figure 4-15. There is no improvement for n = 6 (plot not shown here), providing further indication that n = 4 seems to be the right order.

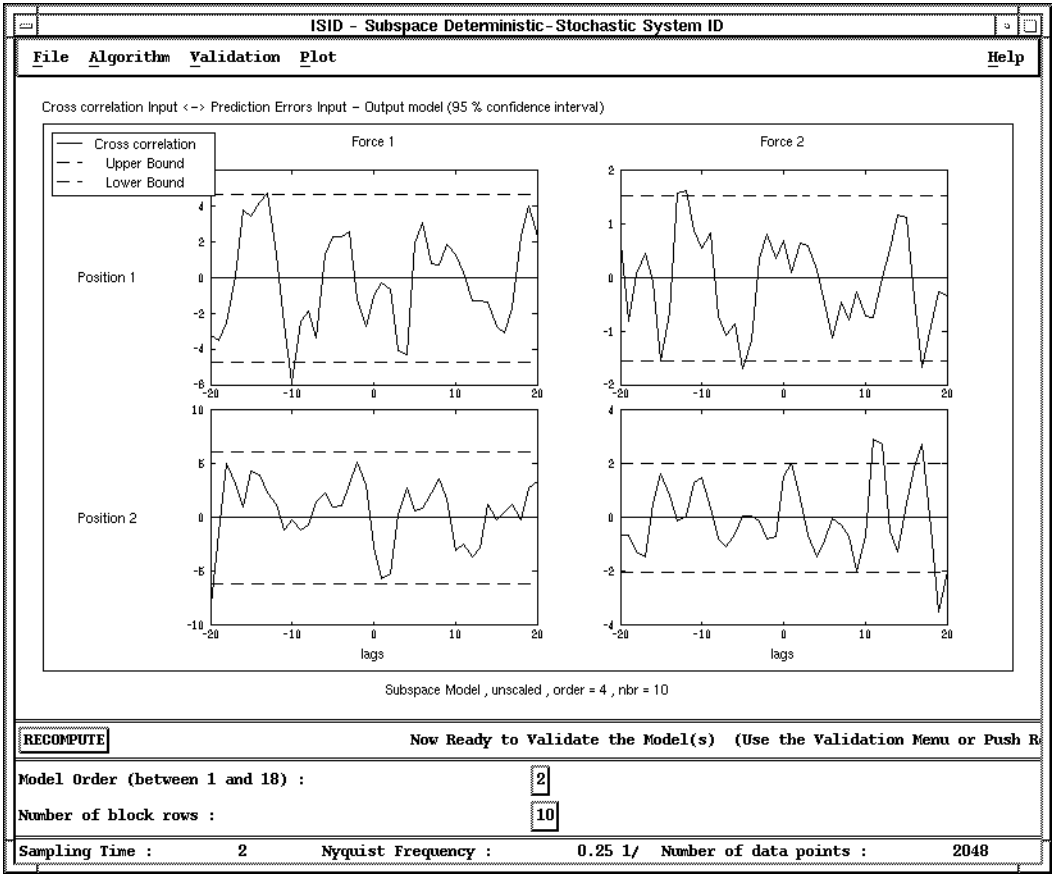

Figure 4-15. Cross-Correlation of Input and Prediction Errors for Second-Order Model

Selecting **Validation»Poles and Transmission Zeros»Input-Output Model** generates the deterministic system's pole-zero plot in the complex plane, as shown in Figure 4-16. The two lightly-damped modes are clearly visible. The plot only shows poles and zeros falling between –2 and 2. This pole-zero feature also can be used to detect overmodeling.

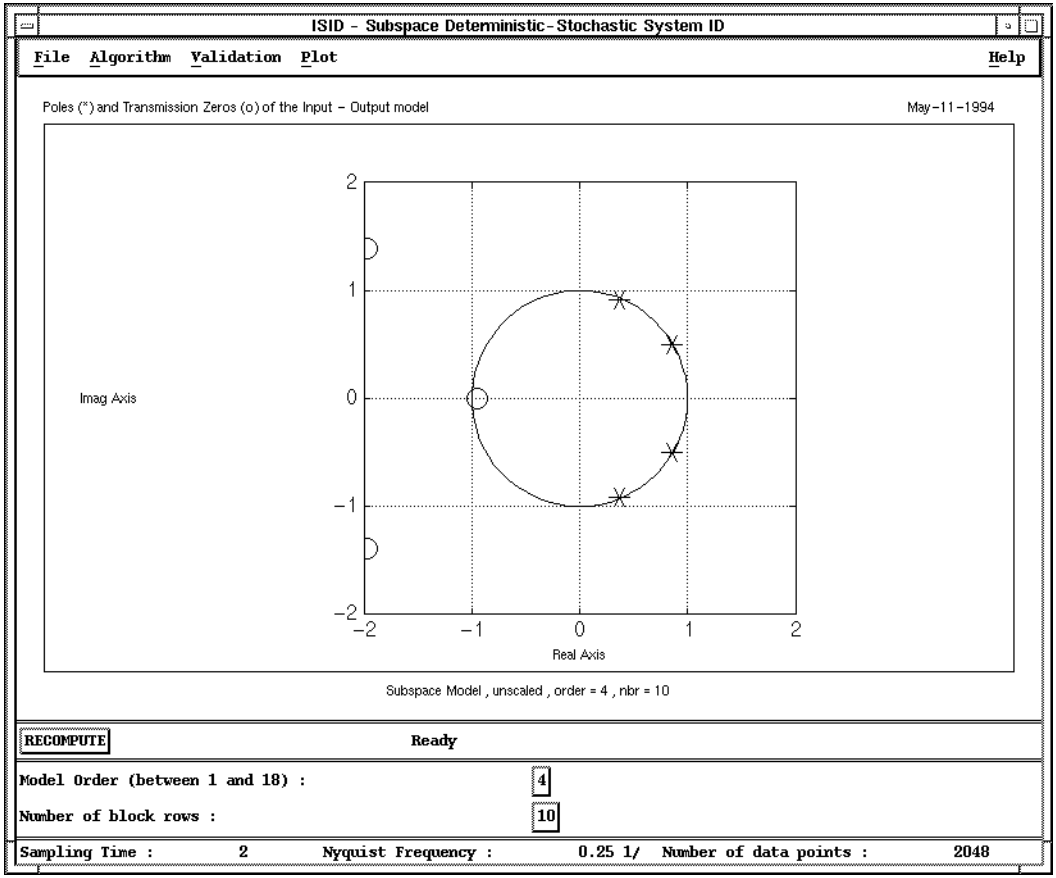

Figure 4-16. Pole-Zero Plot for Fourth-Order Model

Most of the time this is the case when a pole and a zero almost cancel. As an example, enter `n = 8`, and look at the resulting pole-zero plot, as shown in Figure 4-17, and notice the near pole-zero cancellations. There are indications of overmodeling (four poles and zeros that cancel).

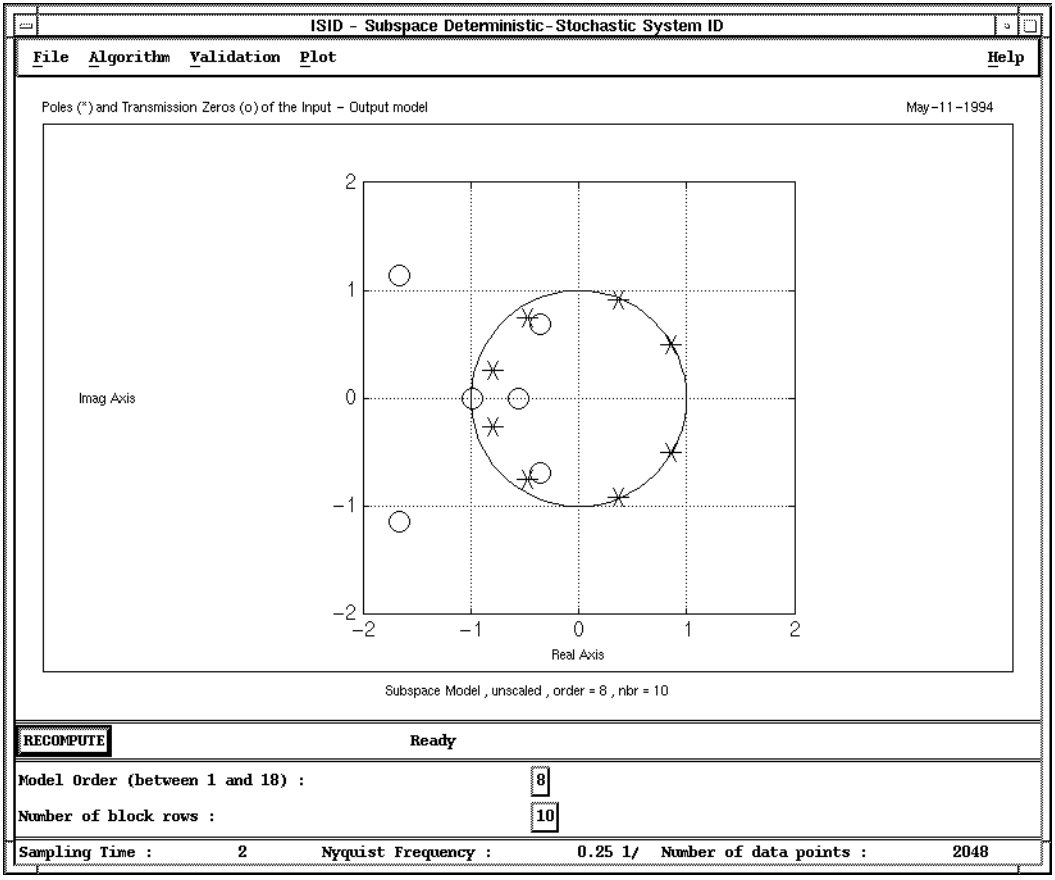

Figure 4-17. Pole-Zero Plot for the Eighth-Order (Overmodeled) Model

Finally, selecting the **Algorithm»n** popup, you can change `basis` from `unscaled` to `combined`, and `bias` from `no bias` (0) to `bias` (1). In practical situations, it is often useful to try out different combinations of these two keywords. In this simple example, the difference is slight, but in practical situations, the differences between the different combinations are often on the order of percentages. Table 4-1 compares the errors (in percentage) for the four different combinations. Your results could differ slightly due to different noise realizations.

Table 4-1. Percentage Errors

Bias	Basis	Input-Output		Innovations	
		Output 1	Output 2	Force 1	Force 2
No	Combined	31.99%	26.80%	29.29%	20.97%
No	Unscaled	31.94%	26.81%	29.30%	20.99%
Yes	Combined	31.93%	26.72%	29.29%	21.04%
Yes	Unscaled	31.84%	26.64%	29.29%	21.02%

Subspace Identification of Stochastic Systems

The `sst( )` function for stochastic-system identification is the only function specifically designed to identify a system model from output data only. `ls( )` can be called with either input and output data, or output data and a null entry for the input data; however, the interactive tool option is not supported for `ls( )` in the latter case.

If you have not already done so, load the data you created in the *Tutorial Data* section.

For this problem, we will use a "true" stochastic system model, `syssto_true`, in state-space innovations form, as well as the corresponding SDF and covariance functions `sdfsto_true( )` and `covsto_true( )`. The noise-added output measurement is `y_sto`. No other *a priori* information is required. The default basis for `sst` is unscaled because it is more useful to examine the principal angles for stochastic systems.

```
[sts_sst,sr_sst] = sst(y_sto,{gui})
```

Alternatively, if `sst( )` is called without the `{gui}` keyword and no model order is specified, a bar plot of the principal angles is generated in the Xmath Graphics window and a popup queries you for the order of the model you need to identify.

The `sst( )` interactive tool appears displaying a bar plot of the principal angles as a function of the innovations model order.

The Model Order is initialized to 0. Examining Figure 4-18, notice that there are three small angles and thus enter `3` as the **Model Order**.

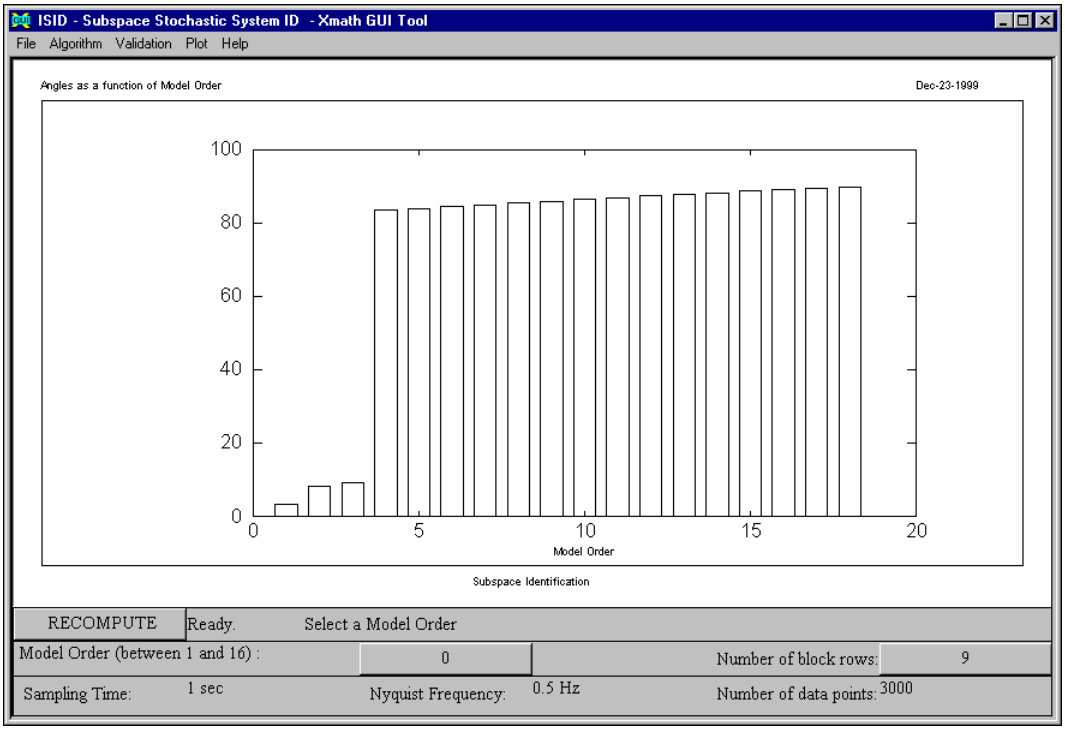

Figure 4-18. Principal Angles as a Function of Model Order

To compare the spectral density function (SDF) of the third-order model with that of the true stochastic model, select **Validation»Frequency Response»SDF Noise Model (Magnitude)**, and then select **File» Compare With Data** and enter `main.sdfsto_true`. This generates a good fit, as shown in Figure 4-19.

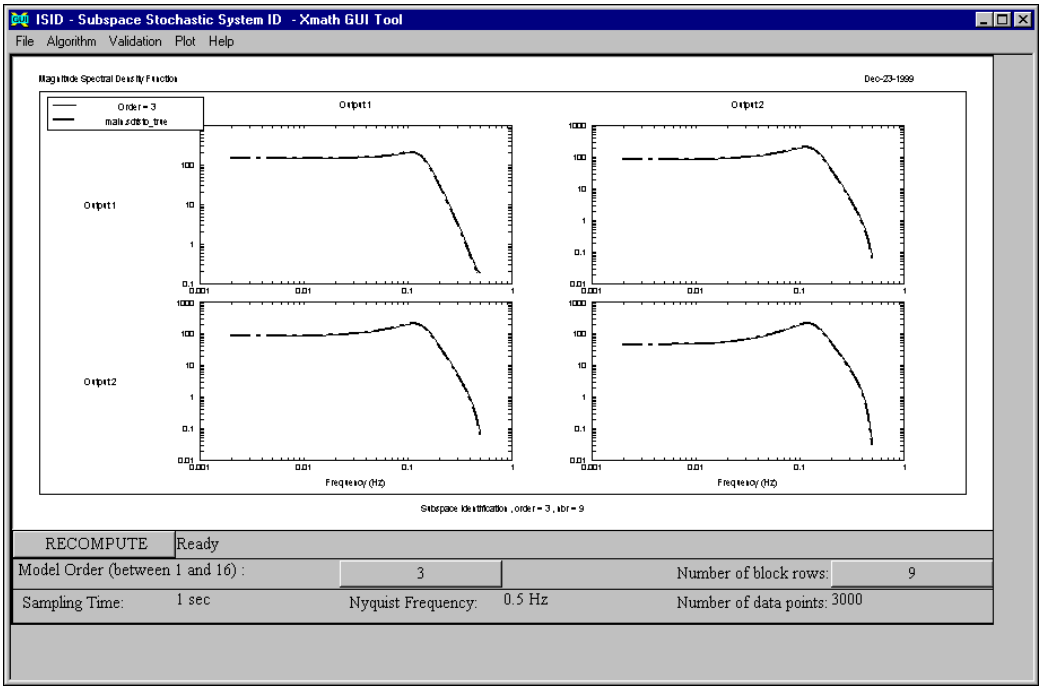

Figure 4-19. Comparison of Third-Order Model Response and True Model Response

Examine the covariance of the noise model based on its impulse response. Select **Validation»Impulse Response»Covariance Sequence Noise Model**, and then **File»Compare With Data** to compare it with the "true" impulse-based covariance, `main.covsto_true`. Figure 4-20 provides further evidence that the model accurately captures the statistics of the measured output.

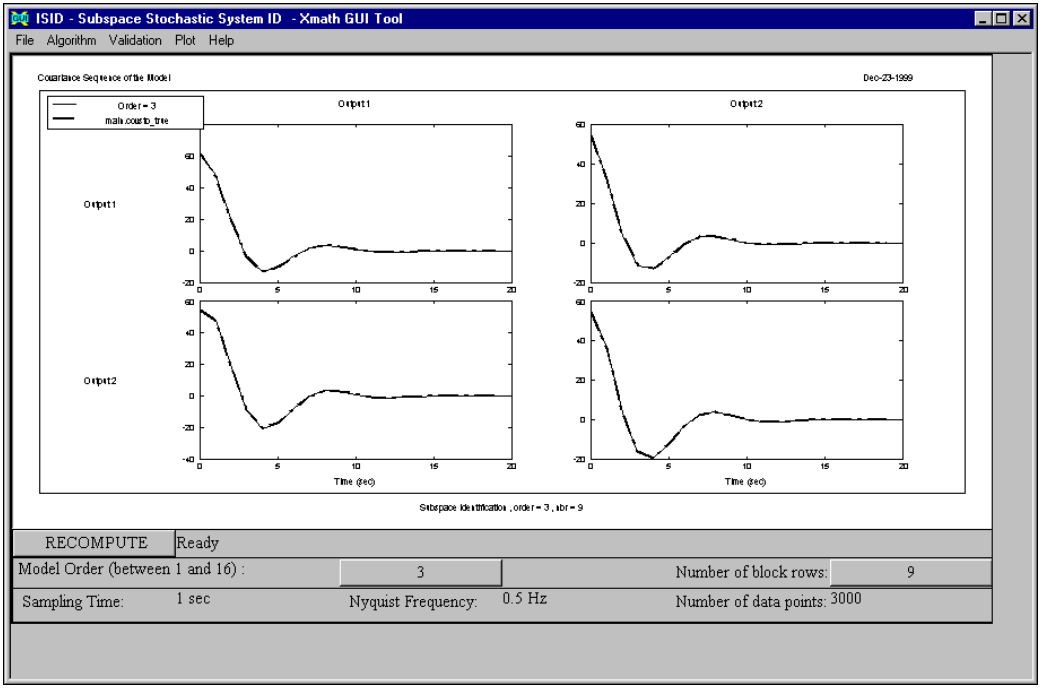

Figure 4-20. Comparison of True and Model Covariance Sequences

The covariance of the prediction errors of the model also fall within the 95% confidence interval. To see this, select **Validation»Covariance Pred. Err.»Innovations model**. To save the current model, select **File»Save Model»Xmath** and enter a variable name.

A square root object is computed internally and stored in the hidden partition `_sst_gui` when the interactive tool is enabled. It can, of course, also be returned as the second output in a call to `sst( )` whether or not the interactive tool is used. As with the other functions returning a square root object (`ls( )` and `sds( )`), `sst( )` identifies lower-order models more quickly if it is called with a square root object input. It is important to notice, however, that because `sst` does not use input data, the `sst` square root is not interchangeable with roots from `ls( )` and `sds( )`.

To try re-identifying the system for a lower-order model, enter `2` as the **Model Order**. Notice that the computation speed is faster than for the original third-order system identification. Validating the resulting system, however, shows that the system is undermodeled—for example, the prediction error covariance exceeds the confidence bounds—you might want to verify this for yourself.

Prediction Error Method

The `pem( )` function is the main command for the identification of systems in state-space form. Because `pem( )` is based on a parameterized model structure using a search algorithm, it is essential to specify:

- An adequate model structure
- A good initial estimate

In case only model-order information is passed, `pem( )` calls `initmodel( )` internally to create a model structure and to come up with an initial parameter estimate.

For ease of use, wrappers `oe( )`, `bj( )`, and `armax( )` have been provided for prediction error estimation of output error, Box-Jenkins, and ARMAX models, respectively; they return the state space equivalent of these polynomial model structures.

Model Structures

`pem( )` can deal with a variety of ways of passing model structures and initial models. We summarize how `pem( )` is initialized for each type of model structure passed through the keyword `{struc}`, which can contain the different types of data discussed below:

- **Model order n**—If specified, the model order is a single integer or a 1×2 vector of integers.

  A single integer implies a common state space of deterministic and stochastic part of dimension `struc`. If `struc` is a 1×2 vector of integers, the deterministic and stochastic parts are separate and have dimensions `struc(1)` and `struc(2)`, respectively. These cases can be compared with the well-known ARMAX and Box-Jenkins polynomial model structures. When used in this way, `initmodel( )` is called internally to produce a model structure and initial parameter estimate. This is the easiest way of using `pem( )` and is most suitable for regular use.

If an initial input/output model is passed to `pem( )` through `initm( )`, the last element of struc is used to define the model order of the stochastic part. If `struc` is not passed, the order of the stochastic part is taken to be equal to that of the deterministic part. This option should be used when you want to pass a model identification result obtained earlier as an initial estimate for `pem( )`. Preferably, initial models should be in innovations form (refer to the next bullet). Innovations models are produced by `ls( )` and `sds( )` when the `{inn}` keyword is passed. Other algorithms such as `giv( )`, `irea( )` and `fwls( )` return models of the deterministic part only.

- **Model structure in state space form**—Model structures are specified in the form of a template system containing system matrices. This template must be dimensioned identically to the initial model that is passed through the keyword `{initm}`. Each nonzero number of `struc` indicates that the corresponding element of `initm` is a parameter that needs to be estimated. Special indications are as follows:
  - When a number occurs multiple times, the corresponding elements are parameterized by the same parameter.
  - A negative sign indicates that the element corresponds to minus that parameter value.
  - The zero elements of `struc` indicate that the corresponding elements of `initm` define unparameterized elements, but not necessarily of value zero.

  We show an example of such a model structure the way it is used internally with the `oe( )` function.

  The `initm( )` function must be defined as an innovations model. Therefore, it must have a second set of inputs corresponding to the stochastic part. These inputs are labeled `Noise 1`, `Noise 2`, and so forth. Another constraint is that the predictor, corresponding to this innovations model, must be stable—in other words, the zeros of the stochastic part must be inside the complex unit circle. The function `reflect( )` can help to achieve this.
- **struc not passed**—In this case, an initial model (`initm`) must be passed. `pem( )` calls `initmode( )` to transform it to canonical form. In case the initial model is an input/output model, `pem( )` adds the stochastic part in canonical form.

For the use of `oe( )`, `bj( )`, and `armax( )`, notice that the state space model order is n_y times the polynomial order, where n_y is the number of outputs. Thus, the following calls:

```
sys = oe(y, u, 3)
sys = bj(y, u, 3)
sys = armax(y, u, 3)
```

are equivalent to the `pem( )` calls:

```
sys = pem(y, u, ny*[3,0])
sys = pem(y, u, ny*[3,3])
sys = pem(y, u, ny*3)
```

Example

Typing:

```
[oe2, mstruc, all_oe2] = pem(y_prbs2, u_prbs2, 2,
{niter=5})
```

activates the prediction error method search algorithm.

The following messages (truncated below) are displayed in the log area:

```
Sample no. 150
    :       :
    :       :
Sample no. 1950
Computing the prediction error variance …
Iteration 1
Weighted trace prediction errors = 1
Computing the gradient …
Parameter no. 1
    :       :
    :       :
Parameter no. 16
Computing the Gauss-Newton gradient …
Stepsize = 0.12263    , criterion value = 1.1005
Stepsize = 0.061315   , criterion value = 0.60981
Stepsize = 0.030658   , criterion value = 0.78126
Stepsize = 0.015329   , criterion value = 0.88831
Stepsize = 0.0076644  , criterion value = 0.94387
Computing the steepest descent gradient …
Stepsize = 0.1        , criterion value = 1.0583e+05
Stepsize = 0.05       , criterion value = 23562
Stepsize = 0.025      , criterion value = 5618.4
Stepsize = 0.0125     , criterion value = 1413.2
```

```
Stepsize = 0.00625    , criterion value = 353.75
Stepsize = 0.003125   , criterion value = 87.258
Stepsize = 0.0015625  , criterion value = 21.339
Stepsize = 0.00078125 , criterion value = 5.4395
Stepsize = 0.00039063 , criterion value = 1.7832
Stepsize = 0.00019531 , criterion value = 1.0319
Stepsize = 9.7656e-05 , criterion value = 0.926
Stepsize = 4.8828e-05 , criterion value = 0.9405
Stepsize = 2.4414e-05 , criterion value = 0.96462
Stepsize = 1.2207e-05 , criterion value = 0.98091
Stepsize = 6.1035e-06 , criterion value = 0.9901
Stepsize = 3.0518e-06 , criterion value = 0.99496
Stepsize = 1.5259e-06 , criterion value = 0.99746
Stepsize = 7.6294e-07 , criterion value = 0.99872
Stepsize = 3.8147e-07 , criterion value = 0.99936
Stepsize = 1.9073e-07 , criterion value = 0.99968
Computing the prediction error variance … Iteration 2
Weighted trace prediction errors = 0.609814
Computing the gradient …
Parameter no. 1
    :       :
    :       :
Parameter no. 16
Computing the Gauss-Newton gradient …
Stepsize = 0.030456  , criterion value = 0.53561
Stepsize = 0.015228  , criterion value = 0.55138
Stepsize = 0.0076141 , criterion value = 0.57646
Stepsize = 0.003807  , criterion value = 0.59223
Stepsize = 0.0019035 , criterion value = 0.60081
Computing the steepest descent gradient …
Stepsize = 0.1       , unstable predictor …
Stepsize = 0.05      , criterion value = 2.19e+05
Stepsize = 0.025     , criterion value = 13780
    :       :
    :       :
Stepsize = 1.9073e-07 , criterion value = 0.60963
Computing the prediction error variance …
Iteration 3
Weighted trace prediction errors = 0.535608
Computing the gradient …
Parameter no. 1
    :       :
    :       :
Parameter no. 16
```

```
Computing the Gauss-Newton gradient …
Stepsize = 0.0019311  , criterion value = 0.53406
Stepsize = 0.00096553 , criterion value = 0.53445
Stepsize = 0.00048277 , criterion value = 0.53493
Stepsize = 0.00024138 , criterion value = 0.53524
Stepsize = 0.00012069 , criterion value = 0.53542
Computing the steepest descent gradient …
Stepsize = 0.1        , criterion value = 62482
    :       :
    :       :
Stepsize = 1.9073e-07  , criterion value = 0.53558
Stop - criterion cannot be improved any further
```

The first lines referring to sample numbers are produced by the least-squares initial model estimate. The lines referring to parameter indices indicate the construction of the derivative of the prediction error to each individual parameter element.

The step sizes refer to the norm of the parameter update difference relative to the norm of the current parameter vector. For the steepest descent method, this always starts at a value of 0.1 for the first update.

The criterion represents the norm of the prediction errors relative to that of the initial model. This number is computed as a weighted quadratic sum, where the weight matrix is either specified by the user or has been computed as the inverse of the prediction error variance of the initial model.

A comparison of the model with the true system is obtained as follows:

```
g = freq(oe2,f);
mtxplt(abs([g,g_true]), 2, 2, {x_log, y_log,
    ultxt="Frequency response magnitude",
    bottxt="Second order output error model",
    legend=["Estimate";"True system"]});
```

and is displayed in Figure 4-21.

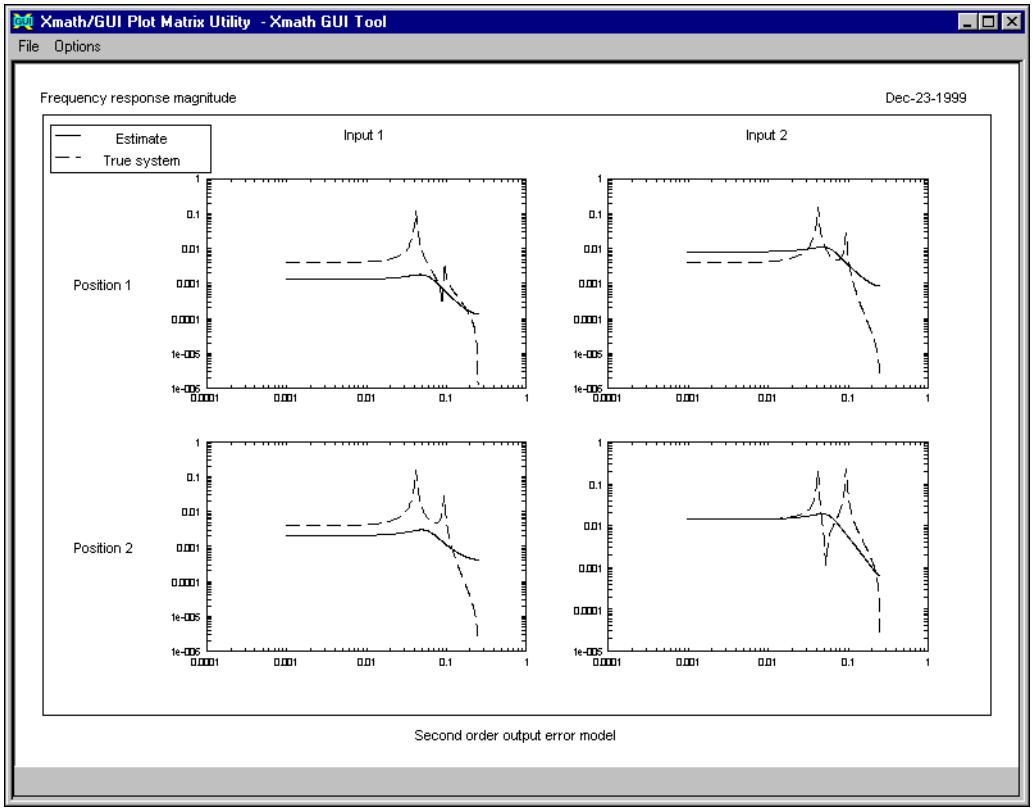

Figure 4-21. Estimate versus True Order System

The model structure that was produced by `initmodel( )` and returned as `mstruc( )` is as follows:

```
mstruc (a state space system) =

A
0     0     1     5
0     0     2     6
0     0     3     7
0     0     4     8

B
0     0     0     0
0     0     0     0
0     0     0     0
0     0     0     0
```

```
C
 9    11    13    15
10    12    14    16

D
0    0    0    0
0    0    0    0

X0
0
0
0
0

Input Names
-----------
Input 1
Input 2
Noise 1
Noise 2

Output Names
------------
Output 1
Output 2

System is discrete, sampling at 2 seconds.

The model oe2 is:
oe2 (a state space system) =

A
0    0    -0.987856      -0.0601275
0    0     0.00478834    -0.962385
1    0     1.58235        1.14707
0    1     0.0985099      0.869246

B
1    0
0    1
0    0
0    0

C
0.000522305    -0.000112008    0.00143108    0.0...
```

```
2.61235e-05     0.00592772     0.000690598     0.0...

D
0     0
0     0

X0
0
0
0
0

Input Names
-----------
Input 1
Input 2

Output Names
------------
Output 1
Output 2

System is discrete, sampling at 2 seconds.
```

These systems reflect the controllable canonical structure that was used by `pem( )`.

Maximum Likelihood Method

The `maxlike( )` function is unique among ISID functions in that it can be used to identify models that are both

- Nonlinear and linear
- Discrete and continuous-time

The `maxlike( )` function is a parametric method but is not restricted to finding linear system models of the types discussed in Chapter 3, *Identification Algorithms*; you can provide any set of parameters that describe a system. A restriction of `maxlike( )` is that the noise is assumed to be added directly to the model output, as with output error model structures.

Its flexibility stems from the fact that the you provide a MathScript function that computes the system output given the system input and parameter

values. This function may have an arbitrary name, but the default name is `model.msf`.

In addition to returning the parameter history, final parameter values, and the output estimates, `maxlike( )` also returns the Jacobian and the root sum square of the output error terms. You can find a more detailed discussion of the MAXLIKE algorithm in Equation 3-8.

To illustrate maximum likelihood identification of a second-order ARMA model using the PRBS data:

```
y = y_prbs2;
u = u_prbs2;
```

Calculate least squares for initial parameter estimate:

```
sys_arma = ls(y, u, 2, {armaform});
tha = sys_arma(5);
thb = sys_arma(6);
p0 = [tha(:, 3:6), thb(:, 3:6)];
p0 = p0(:)';
```

Call `maxlike( )`:

```
p = ones(p0);
[rss,p] = maxlike(u, y, p, {iter=10, delta=0.005});
```

The file `model.msf` has the following contents:

```
Function y = model(p, u);

p = p.*main.p0
tha = zeros(2,4);
tha(:) = p( 1:8 )';
tha = [  eye(2,2), tha]
thb = zeros(2,4);
thb(:) = p( 9:16)';
thb = [zeros(2,2), thb]
sys_arma = arma(thb, tha, 2, 2, {dt=2})
sys = arma2ss(sys_arma)
if max(abs(poles(sys))) > 1 then
  y = 100*main.y
else
  y = sys*u
endif
endfunction
```

In this case, the parameterization has been done relative to the initial least squares parameters; it turns out that the A and B model parameters are orders of magnitude different. Rescaling might improve the `maxlike( )` results in such cases; in general, however, rescaling is not required. Notice also that some safety against instability has been incorporated in the model.

To compare the quality of the model obtained with the final parameter values, we can reformat them into an ARMA system model. Construct the model, and plot the frequency response:

```
p = p.*p0;
tha_ml = zeros(2,4);
tha_ml(:) = p( 1:8 )';
tha_ml = [eye(2,2), tha_ml];
thb_ml = zeros(2,4);
thb_ml(:) = p(9:16)';
thb_ml = [zeros(2,2), thb_ml];
sys_arma = arma(thb_ml, tha_ml, 2, 2, {dt=2});
sys_ml = arma2ss(sys_arma);
g_ml = freq(sys_ml, f);
mtxplt(abs([g_ml, g_true]),
    {columns=2, x_log, y_log})
```

This also illustrates the use of `arma( )` and `arma2ss( )` for conversion of the model to state-space form. The results are shown in Figure 4-22. The example is not very illustrative of the much more powerful capabilities that `maxlike( )` has to identify parameters of SystemBuild models. `pem( )` would be a more appropriate way of identifying the example data; however, the technique required is analogous.

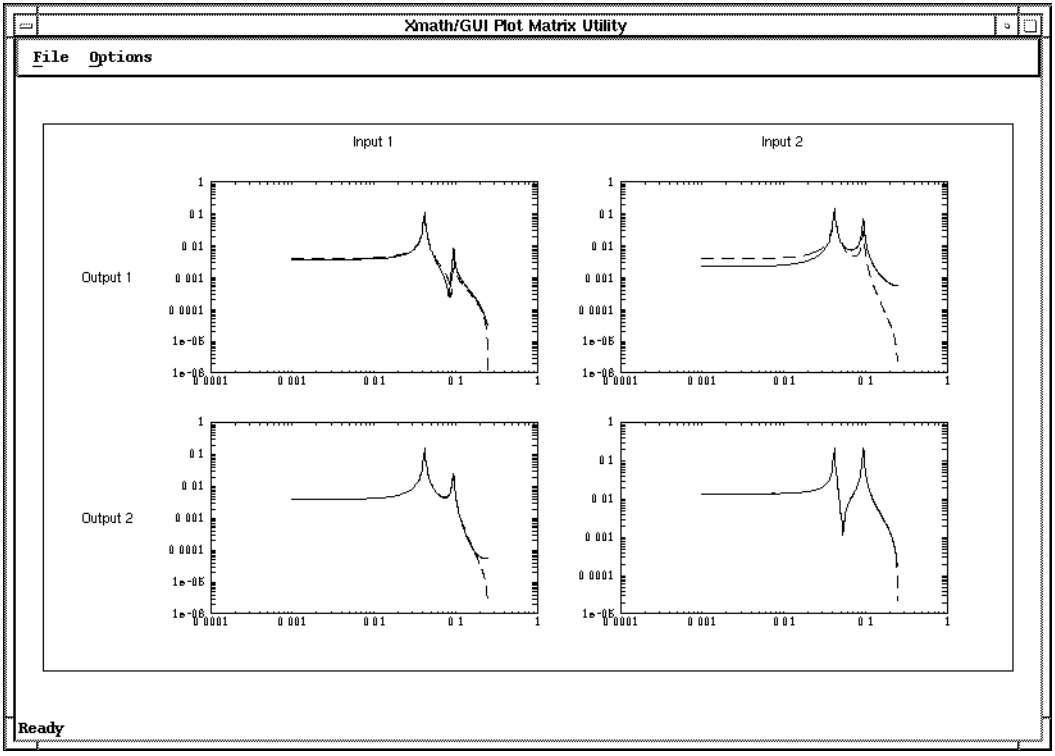

Figure 4-22. Frequency Response for Maxlike Model

Generalized Instrumental Variables

The `giv( )` function implements the generalized instrumental variables approach discussed in the *Generalized Instrumental Variables* section of Chapter 3, *Identification Algorithms*. The optional keyword `{gui}` invokes an interactive tool similar to that used with `ls`, creating a partition named `_giv_gui` to store the data changed interactively through the tool.

If you have not already done so, load the data you created in the *Tutorial Data* section.

To illustrate `giv( )`, try:

```
nmax = 10; lags = 15
sys_giv=giv(y_prbs2,u_prbs2,u_prbs2,nmax,lags,{gui})
```

The interactive tool appears showing the large diagonal terms of the equation error norms, as shown in Figure 4-23. The prediction error variance diagonal terms is a function of model order.

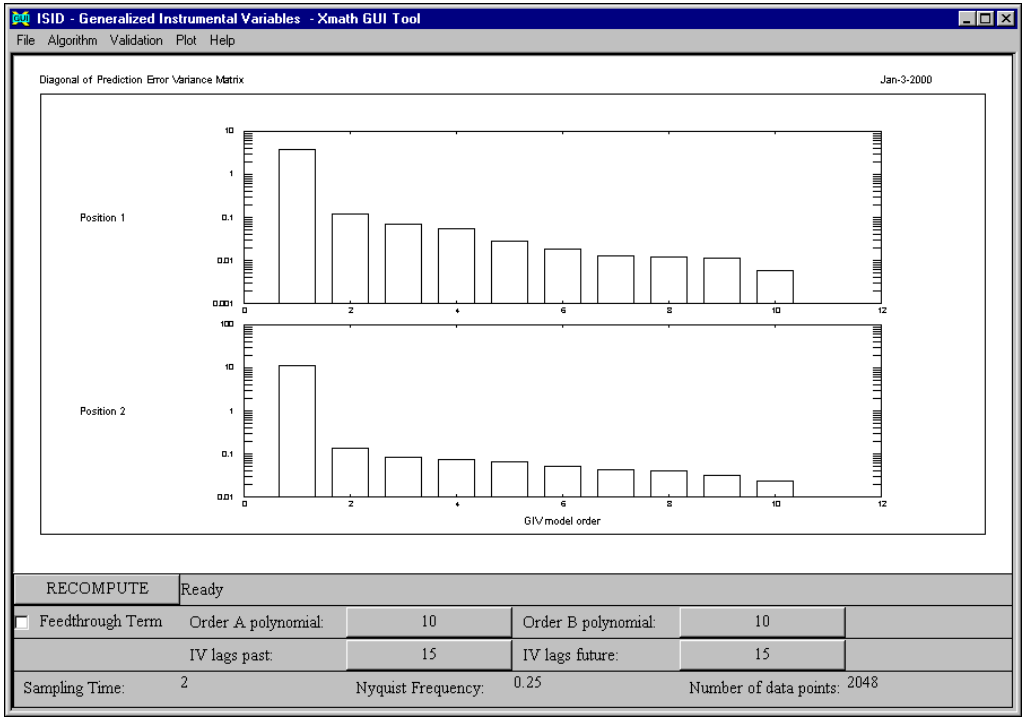

Figure 4-23. Prediction Error Variance Diagonal Terms as a Function of Model Order

We can compare the frequency response of the second-order system with that of the true system as follows:

1. Specify a model order of `2` in **Order A polynomial**.
2. Select **Validation»Frequency Response»Input - Output model (Magnitude)** or **(Phase)**.
3. Select **File»Compare With Model**, and specify `sys_true` to compare the real system with the fourth-order identified system.

   The results appear in Figure 4-24; the fit between the two is good.

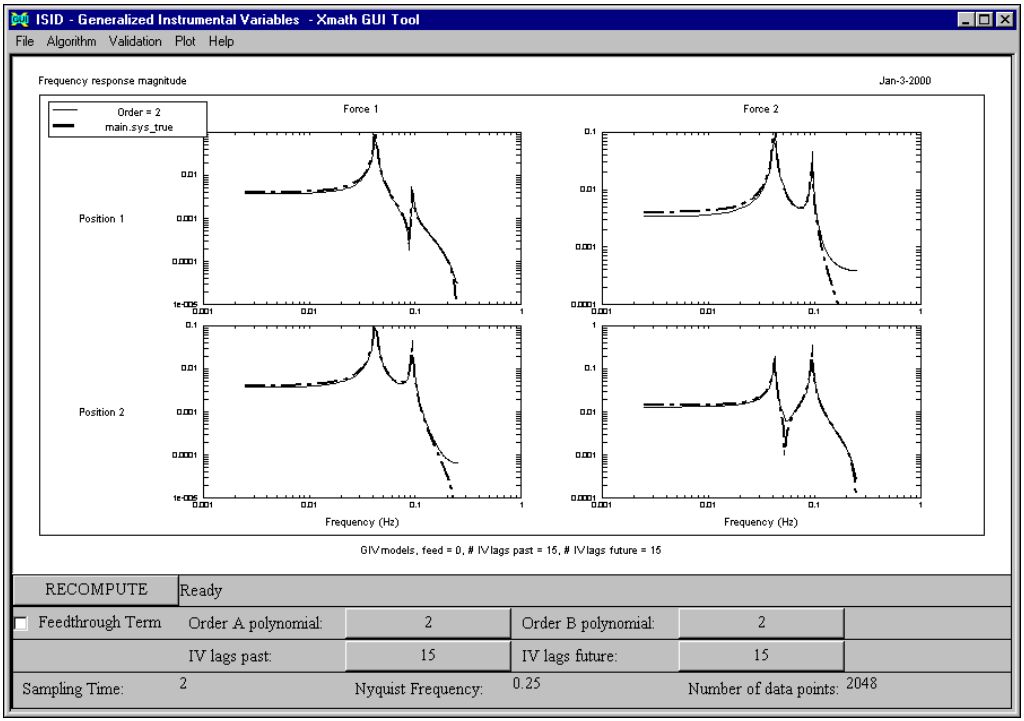

Figure 4-24. Frequency Magnitude Response for Second-Order Model Compared with the True System Response

Signal Analysis

The purpose of signal analysis is to get a rough impression of the model and data quality in a simple, robust, and efficient manner. Important quantities with respect to this are power spectral density (SDF) and coherence function estimates. The `sdf( )` function computes estimates for either of these functions. All functions are either based on the `fft( )` and `ifft( )` functions for the discrete Fourier transformation and its inverse, or on autoregressive modeling based on least squares; for more details, refer to the *Spectral Density Function Estimation* section of Chapter 3, *Identification Algorithms*.

First, we estimate the auto spectral density of `u_prbs2`:

```
suu2 = sdf(u_prbs2,u_prbs2,128);

mtxplt(abs(suu2),
   {axtxt ="Frequency (Hz)",
   bottxt = "Wide band PRBS",
   ultxt = "Spectral density function"})
```

Figure 4-25 shows that the bandwidth of this signal is half the highest frequency because the signal is kept constant during two consecutive samples. The results are a one-sided spectral density.

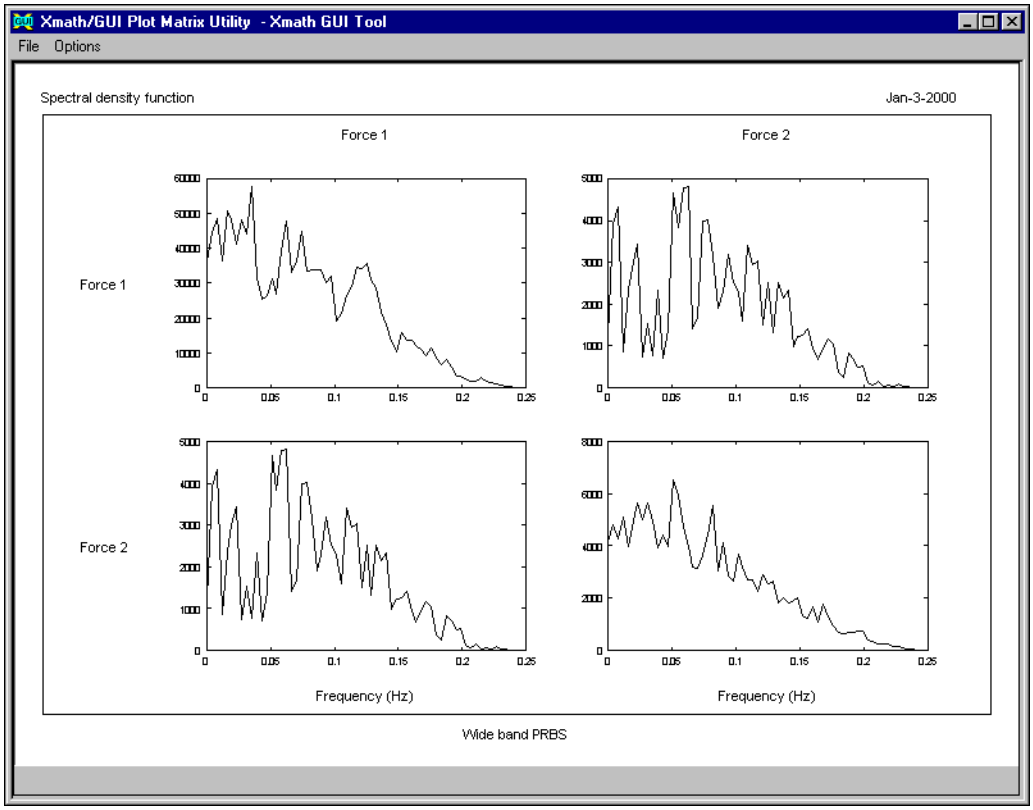

Figure 4-25. SDF of u_prbs2

If a two-sided spectral density function is desired, type:

```
suu2 = sdf(u_prbs2,u_prbs2,128,{fmin=0,fmax=0.5});
```

The SDF estimation of the sine sweep (`suu_ss` below) is a good example of how problematic spectral density estimation can be. As this is a nonstationary signal, we may expect problems; spectral density estimation is based on the assumption that the data is stationary.

```
ss = u_ss1(1,1); suu_ss = sdf(ss,ss,128);
mtxplt(abs(suu_ss),{axtxt = "Frequency (Hz)",
   bottxt = "Sine sweep, SDF averaging method",
   ultxt = "Spectral density function"})
```

The results are shown in Figure 4-26. In contrast with the expected flat spectral density, we see four sharp peaks because `sdf( )` systematically zeros out certain frequencies due to tapering. Each frequency appears only once in the time window due to the characteristics of sines sweep data.

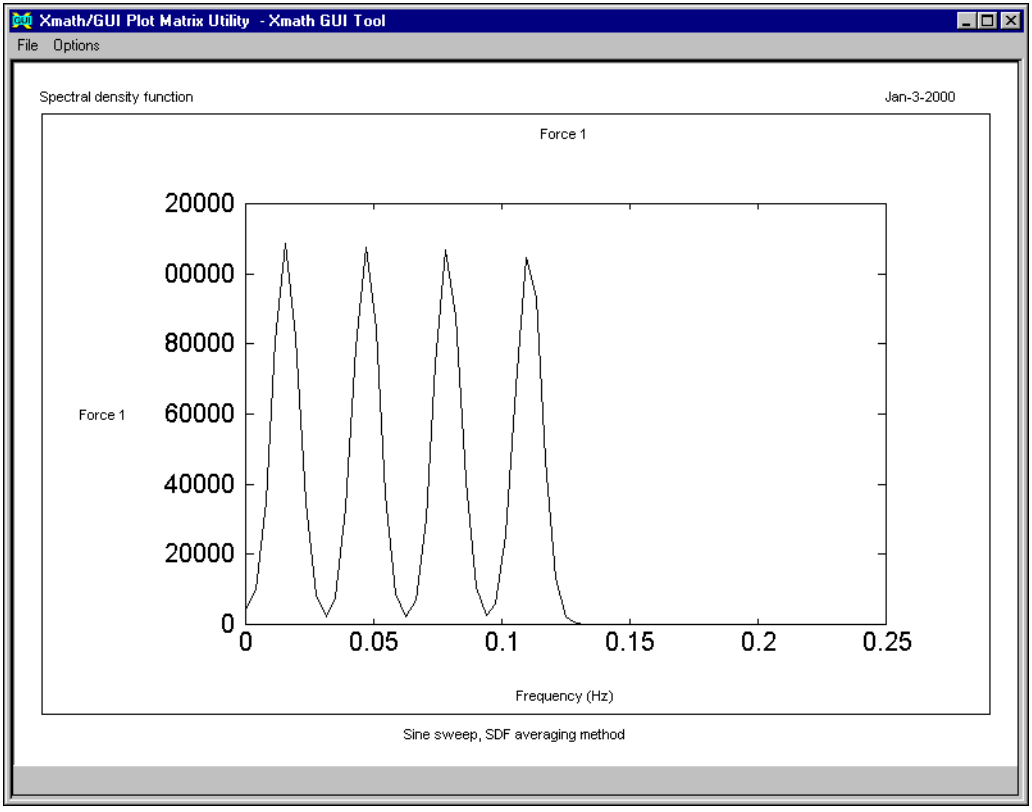

Figure 4-26. Sine Sweep SDF Showing Frequency Suppression

We can improve the results dramatically by tapering the data at several overlapping windows instead of the default non-overlapping case. This is achieved through the `overlap` keyword. The following example uses an eight-fold overlap. The computational work is eight times larger than with the default overlap parameter of 1.

```
suu_ss_tap = sdf(ss,ss,128,{overlap=8});
mtxplt(abs(suu_ss_tap),{axtxt = "Frequency (Hz)",
bottxt="Sine sweep, SDF averaging method "+...
"with 8-point overlap",
ultxt="Spectral density function"})
```

The resulting SDF estimate, which averages the raw SDF estimates over these windows, is displayed in Figure 4-27.

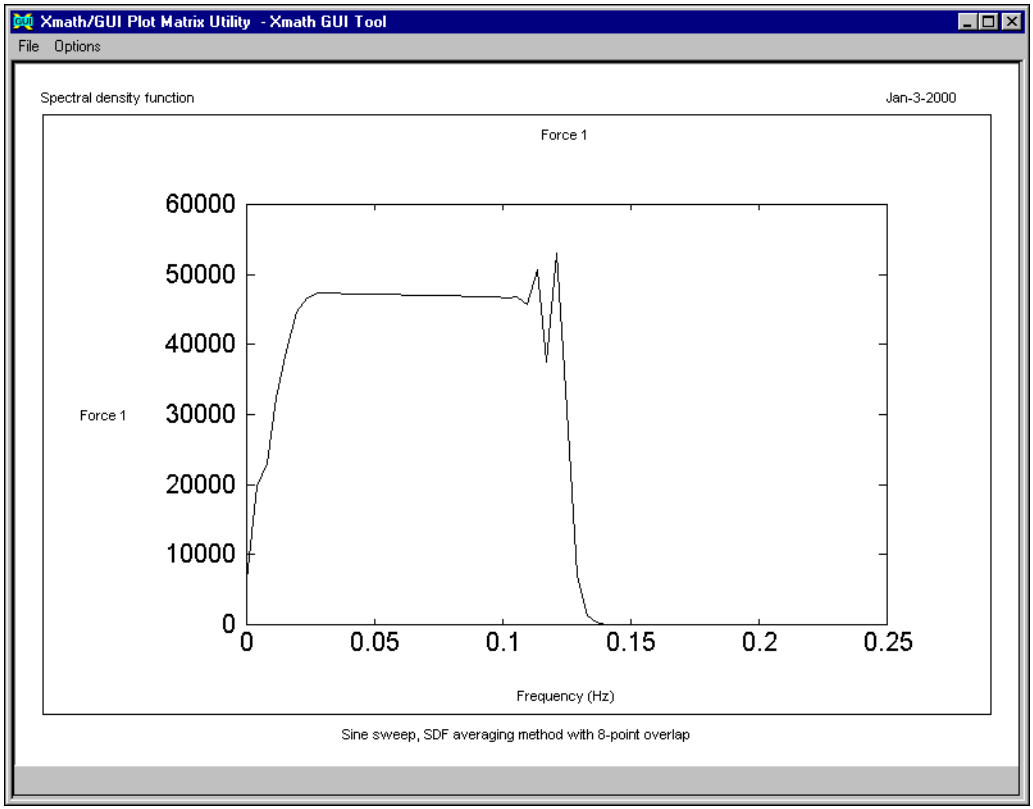

Figure 4-27. Sine Sweep SDF - Averaging Over SDFs with Overlap

An alternative method to achieve similar results is to average unwindowed covariance functions over parts of the data, which is done by specifying the `{bt}` (Blackman-Tukey) keyword.

```
suu_ss_bt = sdf(ss,ss,128,{bt});
```

The results are comparable to the previous one and is therefore not shown.

You can obtain the coherence of `y_prbs2` and `u_prbs2` by calling `sdf( )` with the `{coh}` keyword as follows:

```
cyu = sdf(y_prbs2,u_prbs2,128,
   {wintype = "Hamming",coh, fmin=0,fmax=0.25});
mtxplt(abs(cyu),{axtxt="Frequency (Hz)",
   bottxt="Narrow band PRBS",
   ultxt = "Coherence"})
```

The `sdf( )` keyword `{wintype}` specifies the type of data windowing to be used. Window options are discussed in more detail in the *Spectral Density Function Estimation* section of Chapter 3, *Identification Algorithms*, as well as in the *sdf( )* topic of the *Xmath Help*.The results are displayed in Figure 4-27. The low coherence beyond 0.15 Hz is due to the limited system bandwidth. In row 2 of this matrix plot, we can see that where the coherence with the first input (Force 1) is high, the coherence with the second input is low, and vice versa. This is due to the multivariable nature of our data: Because the coherence is based on SDF computations of scalar signals, the influence of input 2 on the coherence between the output and input 1 (and vice versa) is interpreted by the algorithm as noise.

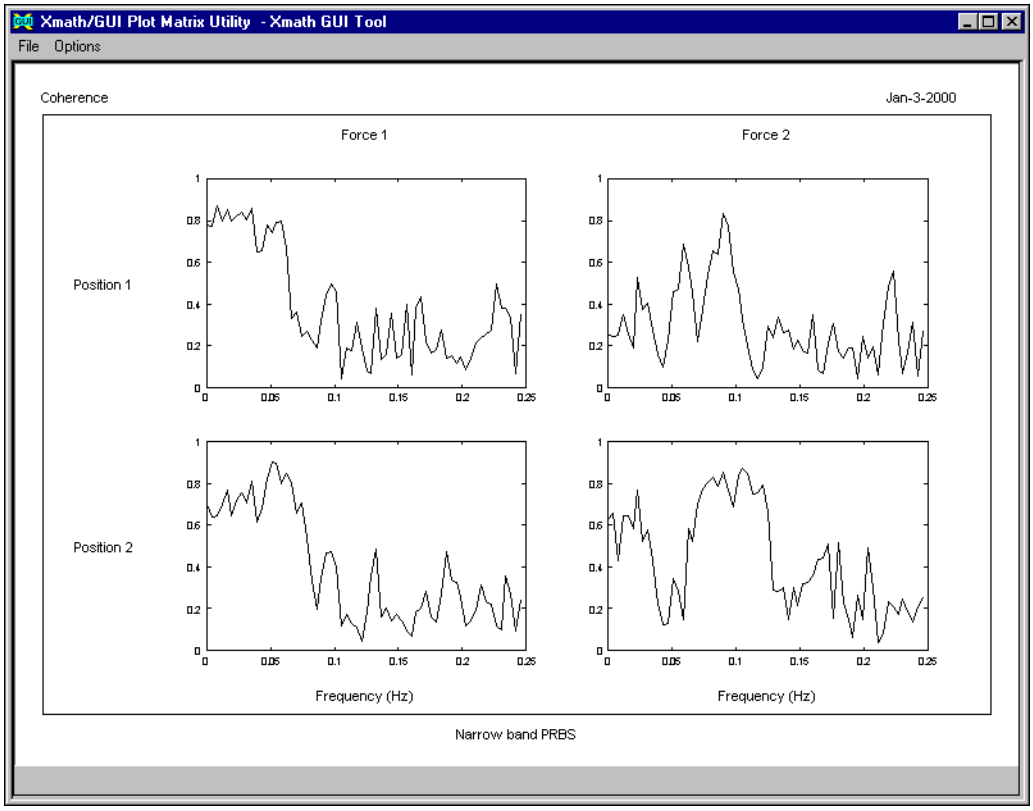

Figure 4-28. Coherence Narrow Band PRBS Data

The last `sdf( )` keyword option to be discussed in this section is `{nar}`, which specifies that autoregressive modeling is used to estimate the SDFs. `{nar}` is set to a value indicating the number of lags in the autoregressive model as shown below. This keyword has the same effect as the window width with the FFT-based methods, but, in general, it is considerably smaller.

```
[suu_ar] = sdf(u_prbs2,u_prbs2,300,{nar=10})
```

With the `{nar}` keyword specified, the number 300 refers to the number of frequency points over which the SDF is computed.

```
mtxplt(abs(suu_ar),{axtxt="Frequency (Hz)",
   bottxt="Narrow band PRBS, "+...
   "autoregressive-based SDF",
   ultxt="Spectral density function"})
```

The results shown in Figure 4-29 are quite comparable to the FFT-based results shown in Figure 4-25.

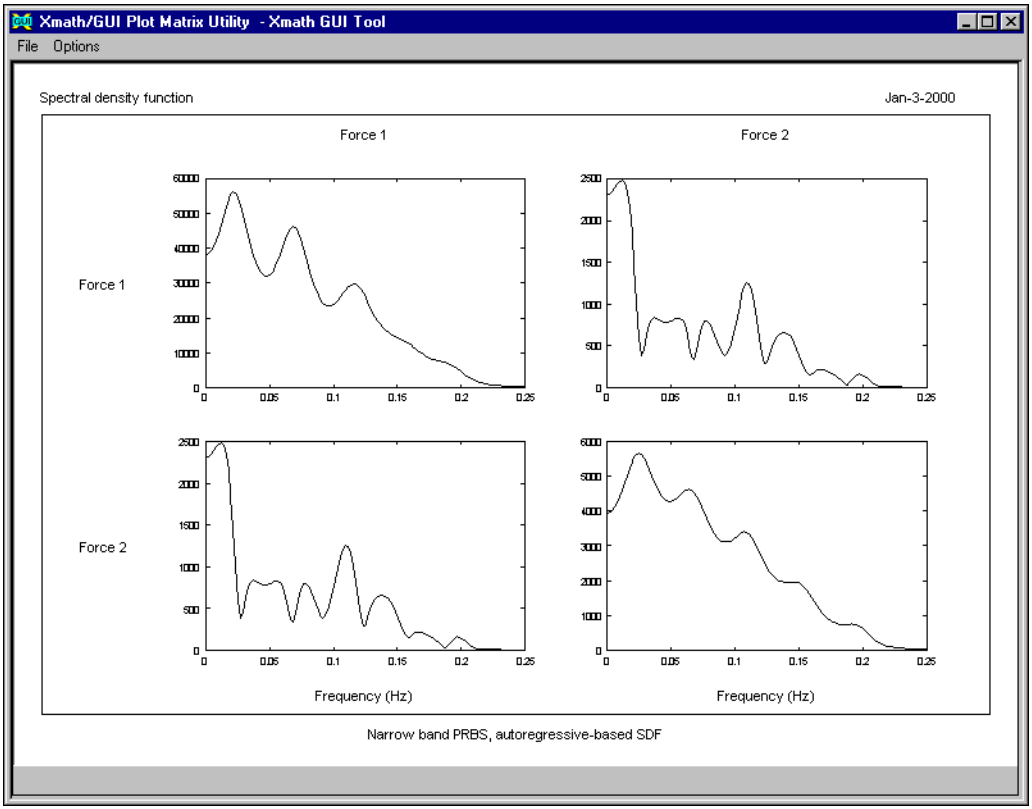

Figure 4-29. AR Based SDF Estimate

Empirical Transfer Function Estimation

The empirical transfer function estimate (ETFE) is implemented with the `etfe( )` function. It has a corresponding interactive tool. The `etfe( )` function calls `sdf( )` internally and has a similar list of associated keywords. Refer to the *etfe( )* topic of the *Xmath Help* for more information. These keyword-based options also can be implemented within the interactive tool.

If you have not already done so, load the data you created in the *Tutorial Data* section.

Call `etfe( )` as follows:

```
[g_etfe,sdf_noise]=etfe(y_prbs2,u_prbs2,256,{gui})
```

Specifying the `{gui}` keyword brings up the graphical user interface for `etfe( )`, shown in Figure 4-30, and also creates the hidden partition `_etfe_gui` to store data changed interactively through the tool.

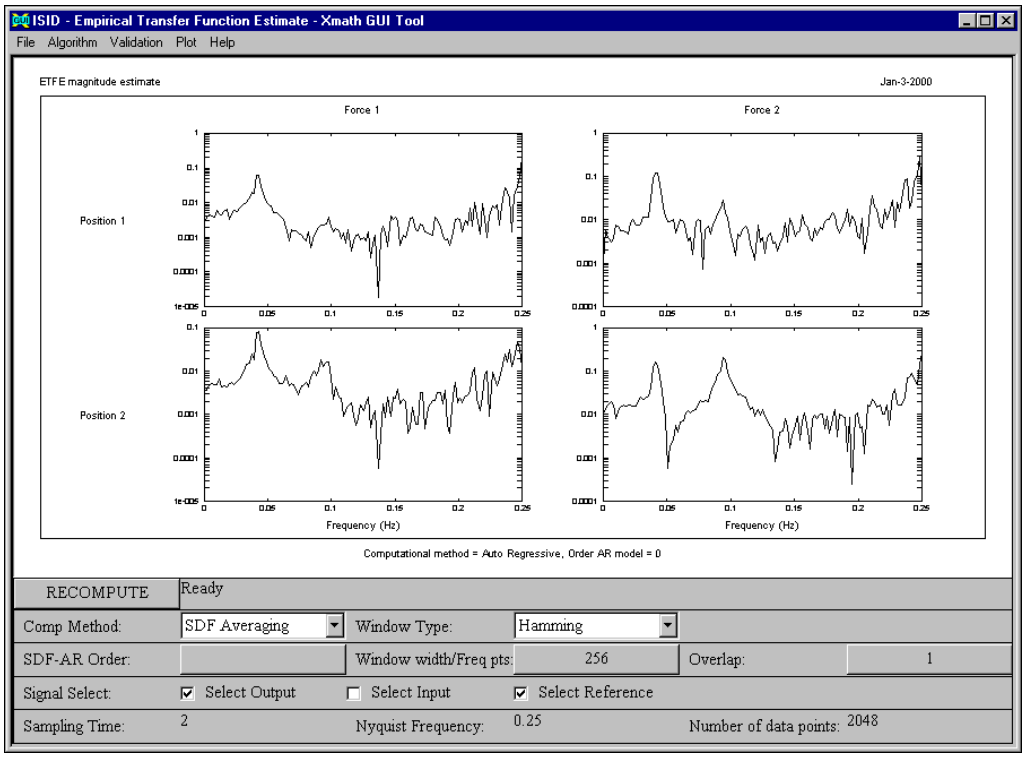

Figure 4-30. etfe GUI Tool

The estimated frequency response has a much higher gain than we would expect beyond about 0.12 Hz. This is a consequence of the bad signal conditioning in that frequency region. You can confirm this by selecting **Algorithm»Coherence»Magnitude**, as shown in Figure 4-31.

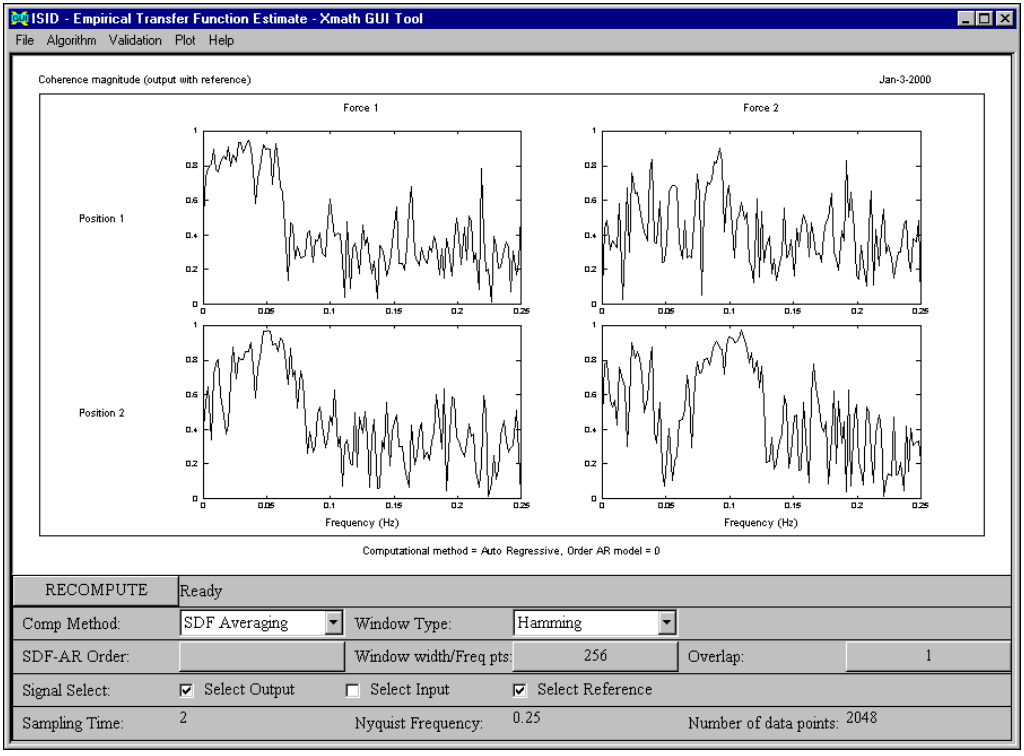

Figure 4-31. Coherence Estimate

Additional `etfe( )` options can improve the results. For instance, you can change the window width, the overlap parameter, and other parameters displayed in the `etfe( )` GUI. Because these parameters are identical to the ones discussed in the *Signal Analysis* section, we do not display the results here.

The Algorithm pull-down menu is quite comprehensive; it offers the interactive computation of covariance, coherence, and spectral density estimates. You can determine which of the three kinds of signals—output, input, and reference—to use for these menu options with checkboxes on the GUI.

Examine the time-domain impulse response obtained from inverse Fourier transforming the ETFE. This can be done in combination with a weighting of the ETFE to remove unwanted components related to the low coherence beyond the bandwidth of the input signal and/or the system.

You can edit the ISID weighting function interactively. Select **Algorithm»Impulse response»Weighted»Enter weight function**. Then hold the <Shift> key down and click the left mouse button over the data line, dragging it until the value beyond 0.12 is very small (that is where the coherence is small). It should resemble Figure 4-32.

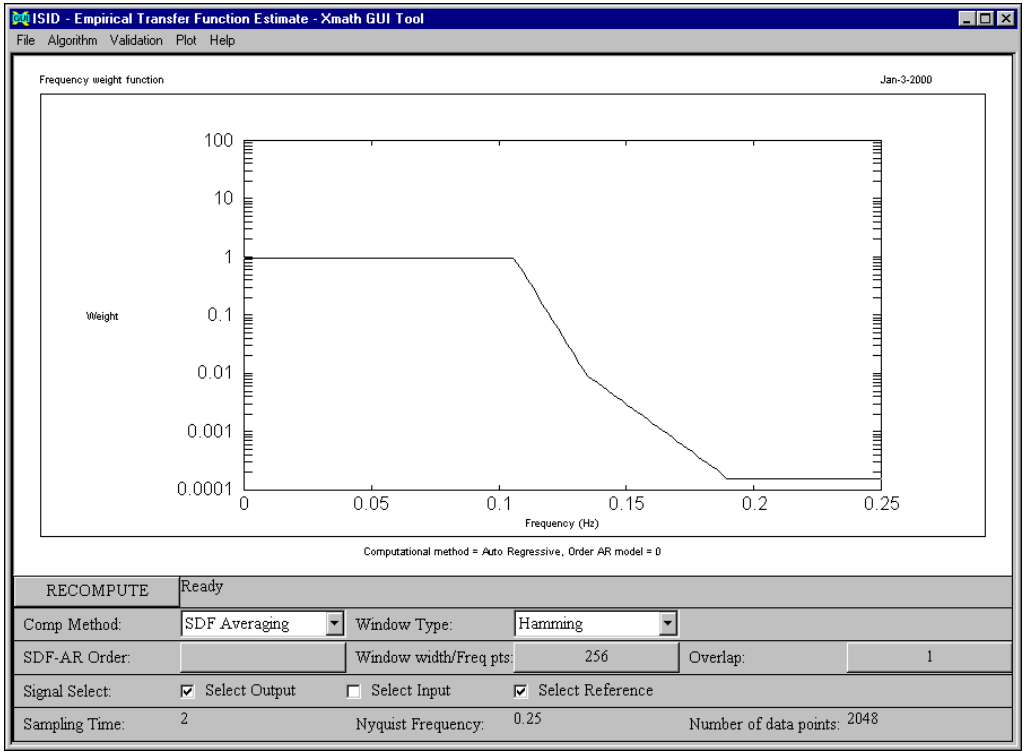

Figure 4-32. Weight Function

To see a comparison of the unweighted and weighted impulse response estimates, select **Algorithm»Impulse response»Weighted»Compute impulse response**. The results are displayed in Figure 4-33.

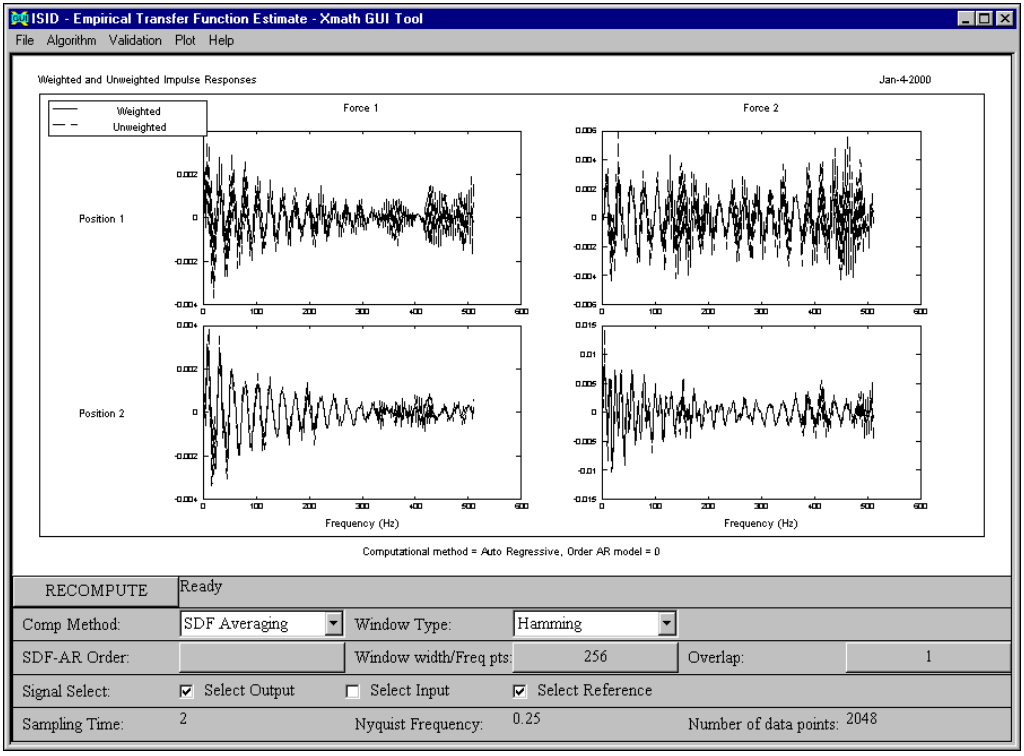

Figure 4-33. Unweighted and Weighted Impulse Response Estimates

Save the impulse response as an Xmath variable by selecting **File»Save Plot»Xmath**. Enter the name `main.etfe_impulse` in the popup that appears. You will use this impulse response data in the *Impulse Realization* section, which discusses the `irea( )` identification function.

Impulse Realization

You use the impulse response `main.etfe_impulse` (generated in the *Empirical Transfer Function Estimation* section using `etfe( )`) as input to `irea`. Alternatively, you can obtain it from the command line as follows.

```
g_etfe = etfe(y_prbs2,u_prbs2,256,{fmin=0,fmax=0.5});
g_etfe(129-66:129+66) = 0;
etfe_impulse = real(ifft(g_etfe,{channels}));
etfe_impulse = pdm(etfe_impulse, [0:255]*dt);
```

The first call uses `etfe( )` in such a way that it produces a two-sided result, and the second call implements the frequency weighting that you did graphically earlier. The PDM call is required because `ifft( )` does not automatically update the domain from frequency (Hz) to time(sec).

We select the first 100 impulse response parameters. Notice that we might even take a smaller number, since the theory does not require that the selected part of the impulse response be damped out completely. We call irea with 10 as the initial order. Because you are using the interactive tool for `irea`, you can interactively change this later based on the results shown in a singular value plot.

```
sys_imp = irea(main.etfe_impulse(1:100),10,{gui})
```

The `irea` interactive tool comes up displaying the Hankel singular values of the identified state-space system `sys_imp`, as shown in Figure 4-34. It simultaneously creates the hidden `_irea_gui` partition to store data relating to the tool.

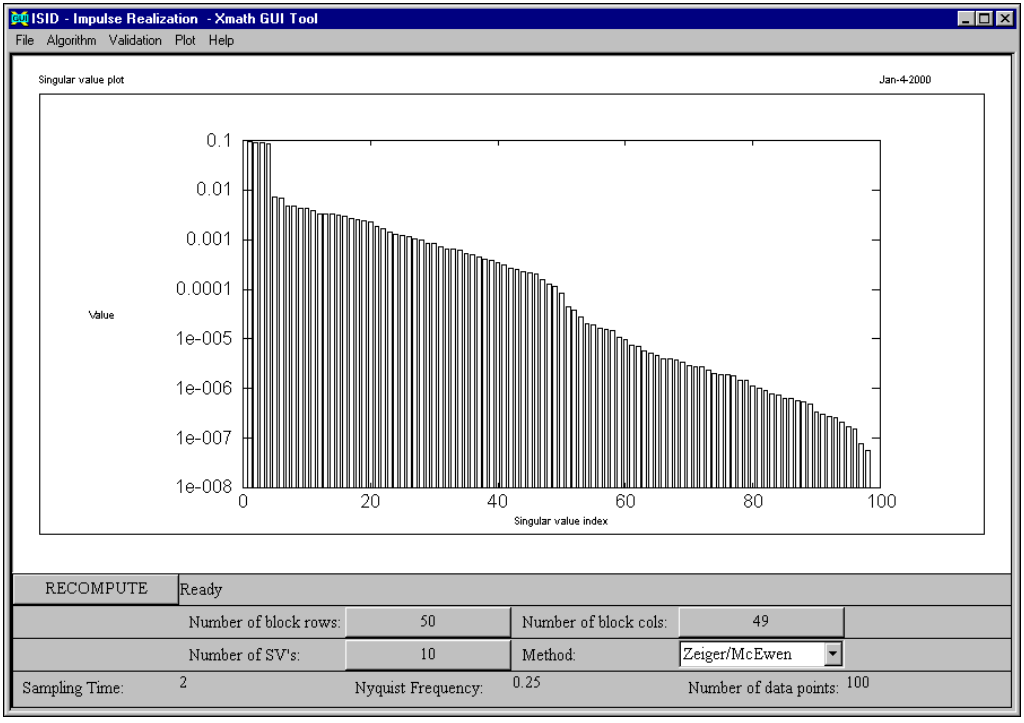

Figure 4-34. IREA - Hankel Singular Values

There are only four significant components in the Hankel matrix, which means that we can obtain a good fourth-order state-space model. You can compare the fourth-order impulse response with the original. Change the number of retained singular values (**Number of SV's**) to `4` and click **RECOMPUTE**. Select **Validation»Impulse Response»Input - Output model**. The comparison of the original and reduced-order impulse responses, shown in Figure 4-35, shows the two to be remarkably close.

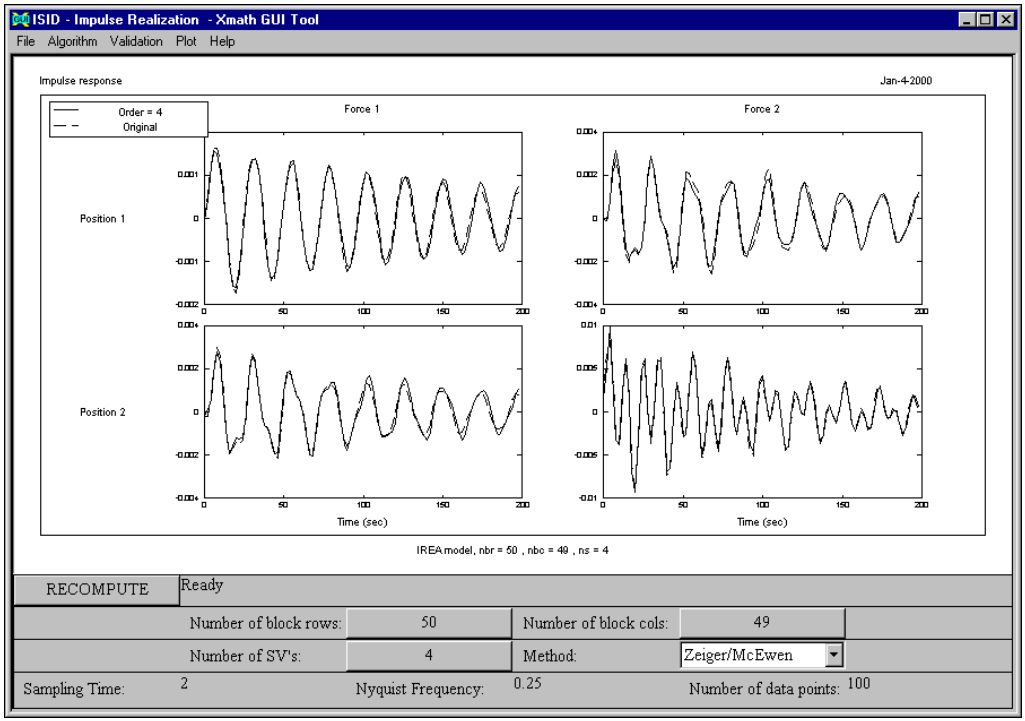

Figure 4-35. Comparison of the Original Order (10) and Reduced-Order (4) Impulse Responses

You now can recompute the frequency response `g_etfe` for the fourth-order model and display it by selecting **Validation»Frequency Response»Input-Output Model (Magnitude)**. Then, select **File» Compare With Data** to load in the known frequency response, `g_true`.

The fit in terms of frequency response is good, as shown in Figure 4-36.

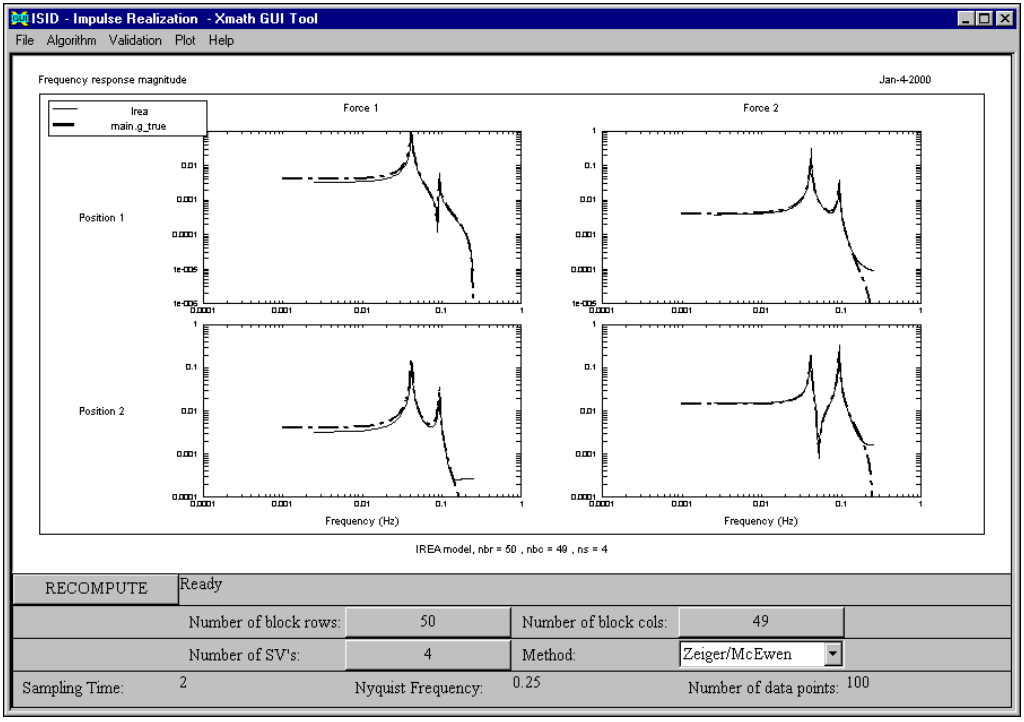

Figure 4-36. Comparison of Fourth Order Model and True Model Impulse Realizations

Least Squares in the Frequency Domain

The least-squares method is not limited to the time domain only. You also can make a least-squares fit to a frequency response obtained by `etfe( )` or a high order least-squares estimate as discussed in the *Least Squares-Frequency Domain* section of Chapter 3, *Identification Algorithms*. Because this method provides the option of weighting frequency bands of interest, we refer to this method as the Frequency Weighted Least Squares (FWLS) method. If you have not already done so, load the data you created in the *Tutorial Data* section. We illustrate how to use the `fwls( )` function with its interactive tool.

```
[g_etfe, sdf_noise] = etfe(y_prbs2,u_prbs2,256);

nmax = 10;
[sys_fwls,sr_fwls] = fwls(g_etfe,nmax,dt,{gui});
```

The parameter `dt` must be passed to `fwls( )` since `fwls( )` cannot determine the model time step from frequency domain data.

When the interactive tool is first created, it displays a bar plot of the prediction error variance matrix diagonal terms for each output as a function of model order, as shown in Figure 4-37. The maximal model order is `nmax`, or 10, which is displayed in the **Order A polynomial** and **Order B polynomial** fields. Because `fwls( )`, like `ls( )`, creates a square root object, you can obtain all lower-order models without recomputing the square root information.

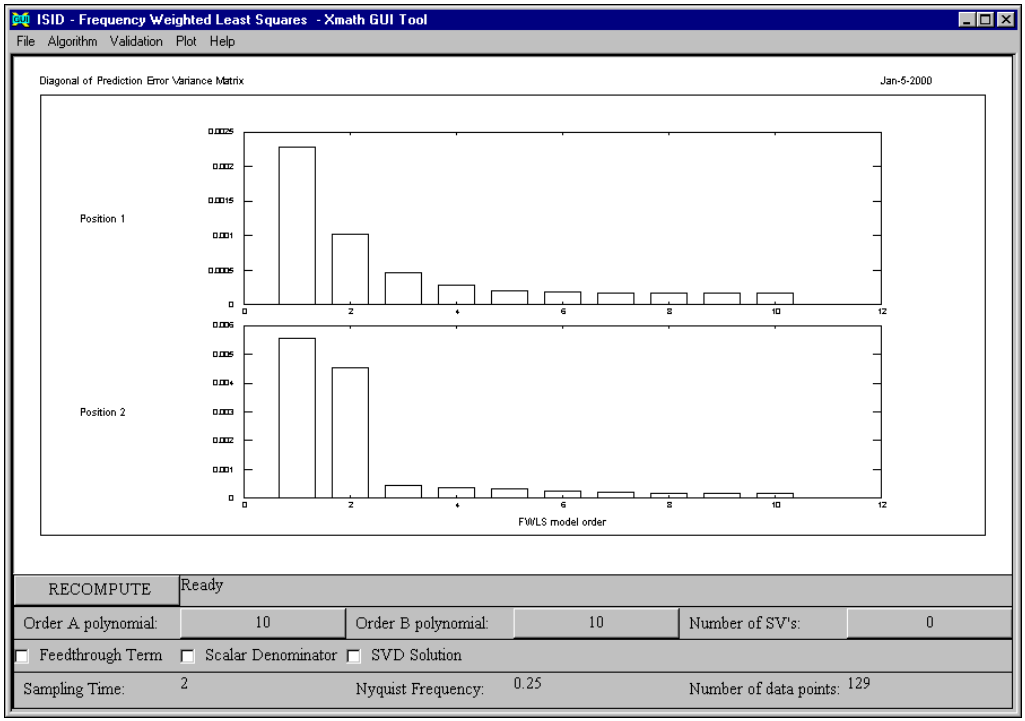

Figure 4-37. FWLS Errors, Uniform Weight

Particularly for the first output, the values do not drop sharply as a function of model order. One approach is to see how well the frequency response of the identified model matched the measured data for a variety of model orders.

You can examine the second-order model responses by entering `2` for the **Order A polynomial** and selecting **Validation»Frequency Response» Input-Output Model (Magnitude)**. The fit is poor; `fwls( )` seems to

have problems fitting the high frequency part. That makes sense because the ETFE is ill-conditioned there. You can take advantage of the frequency weighting feature to improve the results.

To use the frequency weighting feature, complete the following steps.

1. Select **Algorithm»Frequency Weight**.

   A dialog box appears.

2. When asked which input you want to weigh, click **OK** to weigh all inputs equally.

   The plot area then shows the default unity weight function.

   You want the weighting window to resemble Figure 4-38.

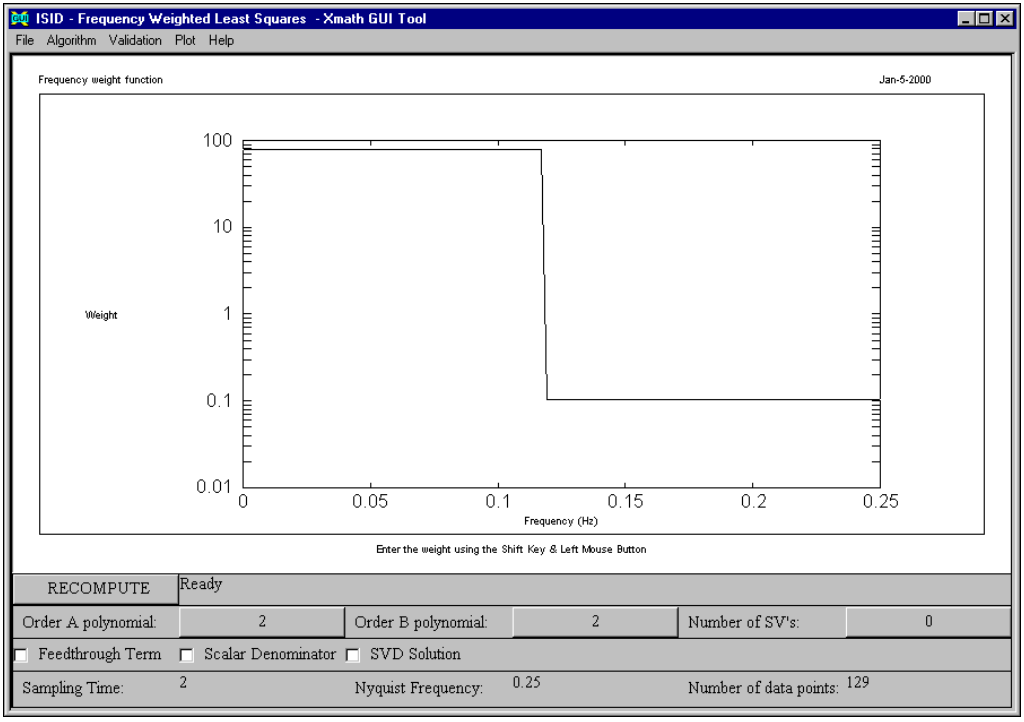

Figure 4-38. Weight Function

3. Change the weight function as follows:

   a. Holding down the <Shift> key, click the mouse button at the left end of the function line and drag it to the right to a frequency of about 0.12 Hz. Without releasing the mouse button, move the mouse until the weight is about 80.

b. Holding down the <Shift> key, click the mouse button at about 0.12 Hz and drag the mouse to the right end of the frequency range. Without releasing the mouse button, move the mouse until the weight is about 0.1.

c. Click **RECOMPUTE** to regenerate the models.

d. Select **Validation»Frequency Response»Input - Output Model (Magnitude)**.

   This action allows you to compare the magnitude response of the weighted model with the ETFE, as shown in Figure 4-39.

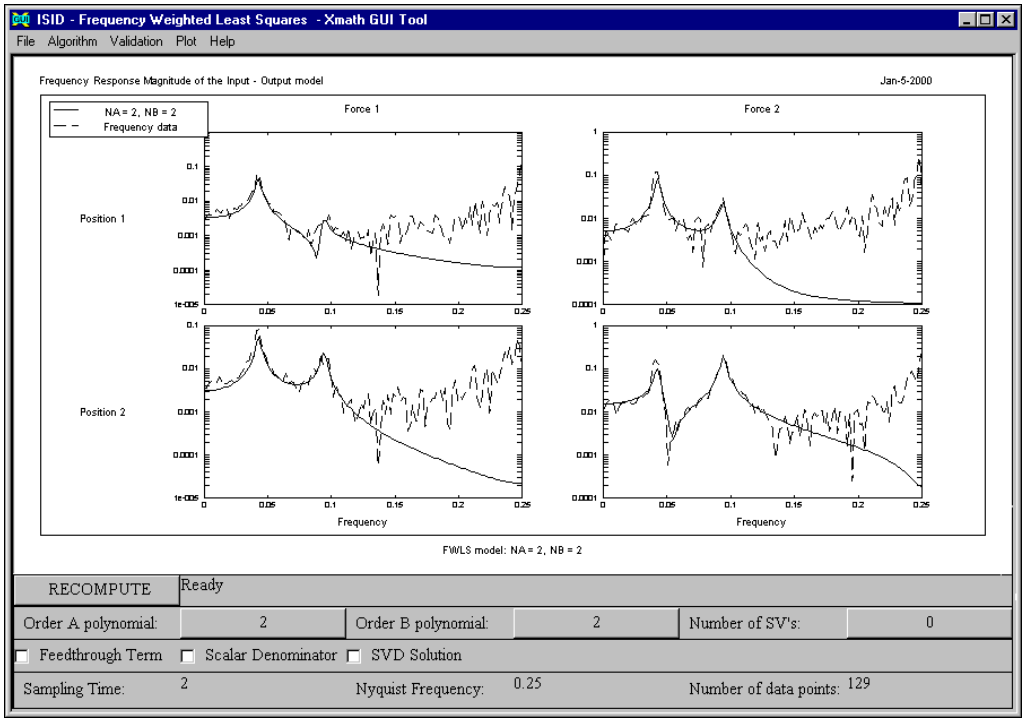

Figure 4-39. Comparison of Second-Order Reduced Model Response and True Response

The result is acceptable. The use of `fwls( )` to reduce frequency responses of high order models is particularly helpful for the `ls( )` case, which frequently requires you to identify high-order ARX models. You may find it informative to try this out on the LS example in the *Least-Squares in the Time Domain* section.

SISO Transfer Function Identification from Frequency Response Data

The `tfid( )` function provides a method for continuous-time frequency-domain identification. `tfid( )` is limited to the identification of single-input, single-output (SISO) systems, but it is a particularly useful approach when some prior information, such as the number of system zeros and poles, is known. `tfid( )` identifies a standard Xmath system in transfer function form using Chebyshev polynomials as basis functions; you can find details of the algorithm in [AD87]. You can specify an optional weighting PDM of the same size as the frequency response to indicate areas of particular interest.

If you have not already done so, load the data you created in the *Tutorial Data* section.

`tfid( )` does not handle the complete multiple-input, multiple-output (MIMO) PRBS identification data used in other examples, but we can extract a channel of the true frequency response and examine how well `tfid( )` can match it.

```
g_siso = g_true(1,1);
```

In calling `tfid( )`, you take advantage of the fact that you have a fairly good idea of the system order, thanks to the previous identifications you have performed. By default, `tfid( )` displays a bode-format magnitude and phase plot in the Xmath Graphics window; for consistency, however, you will suppress this plot and use `mtxplt( )` to compare the responses here. Before making the fit, you truncate `g_siso` to the first 128 samples; near the Nyquist frequency, discrete-to-continuous fits usually give problems so you want to avoid that area.

```
g_siso = g_siso(1:128);
```

```
[tf,sys,g_tfid] = tfid(g_siso,{np=4,nz=4,!graph});
```

```
mtxplt([abs(g_siso),abs(g_tfid)],{y_log,x_log,
   columns=1,ultxt="Frequency Response",
   axtxt="Frequency (Hz)",
   bottxt="Measured response (solid} vs."+...
   "tfid response (dashed)"})
```

Figure 4-40 shows that the fit is very close for this frequency range.

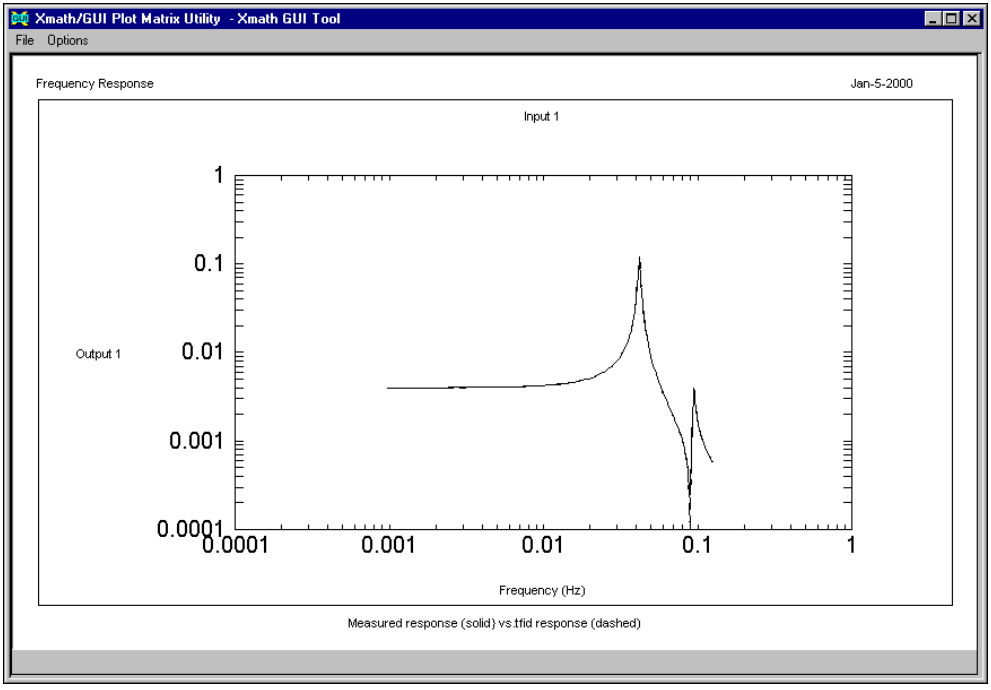

Figure 4-40. Comparison of Measured and tfid-Identified Response with Default Weighting

Instead of using the first part of the frequency response, you might have passed an appropriately defined weight function instead to achieve the same effect. You may want to experiment with iteratively improving the fit using the inverse of the model denominator polynomial as weight function; the reason for this choice is that least squares systematically suppresses the information near the zeros of the denominator polynomial.

The identified SISO system is returned as both a transfer function (`tf( )`) and a state-space system (`sys`); the two results are calculated using different representations. They generally describe exactly the same system, but the state-space system might be more numerically reliable for high-order identifications.

Validation

Validation means determining the model quality. It is the most difficult part of the whole identification cycle and sometimes is more of an art than a science. Many questions are still unresolved, such as: what is "good?" A model might have a large uncertainty in certain frequency bands, yet turn out to be good enough for control design. This section summarizes several validation topics and provides some guidelines.

Innovations Models

Because most model validation utilities deal with prediction errors, a model in innovations form is often more appropriate than an input/output model. The functions `ls( )`, `sds( )`, `oe( )`, `armax( )`, `bj( )`, and `pem( )` produce innovations models when the parameter `{inn}` is passed:

```
[sys,sr] = ls(y_prbs2,u_prbs2,8,{inn});
```

This model has additional inputs for the stochastic parts. Refer to information about innovations models in Chapter 2, *Identification Process*. The input/output part can be extracted by indexing:

```
nu=2;
sys_io = sys(:, 1:nu);
```

The functions `idsim( )`, `idimpulse( )`, and `idfreq( )` can deal with both innovations and input/output models. For example, the following function calls are equivalent:

```
g = idfreq(sys,f);
g = freq(sys_io,f);
```

Computing Prediction Errors

Using the `inn2pe( )` function, prediction errors of an identified model can be obtained as follows:

```
e = inn2pe(sys,y_prbs2,u_prbs2);
```

The more general simulation function `idsim( )` also can be used for this purpose:

```
[yhat, e] = idsim(y_prbs2,u_prbs2,sys,{mode=1});
```

This produces both the prediction and the prediction error. It has a `{mode}` keyword to indicate whether you want a prediction based on a

Kalman-Filter or on the input/output model only; the default is `mode=0`, which corresponds to input/output.

Signal Analysis

Spectral density computation and coherence are useful quantities for determining the input signal bandwidth and the correlation between input and output as a function of frequency:

```
[suu] = sdf(u_prbs2,u_prbs2,256);
[cyu] = sdf(y_prbs2,u_prbs2,256,{coh});
```

Use `sdf( )` to produce spectral density functions, covariance functions, and coherences by passing the appropriate keywords. The ETFE GUI has several menu options that you can use to inspect these quantities interactively; you can obtain covariance and SDF estimates of the noise, as well.

Stochastic Properties of Innovations Models

The functions `idimpulse( )` and `idfreq( )` produce the noise covariance and spectral density functions as additional outputs:

```
[g,sdf_n]=idfreq(sys,f);
```

```
[imp,cov_n]=idimpulse(sys,[0:2:200]);
```

The estimates are based on the estimated innovations model only, not on signal analysis. You can invoke the `etfe( )` function, however, to produce a nonparametric noise SDF estimate:

```
[g,sdf_n] = etfe(y_prbs2,u_prbs2,512);
```

Model Uncertainty Estimates

The functions `ls2unc( )` and `inn2unc( )` produce estimated parametric model uncertainties based on the Fisher information matrix. Assuming correctness of the model assumptions, the one-sigma model frequency response model errors are estimated by conversion of the estimated parameter variance to frequency response magnitude:

```
[g,deltag] = ls2unc(8,sr,f);
```

```
[deltag] = inn2unc(y_prbs2,u_prbs2,sys);
```

You can apply `inn2unc( )` to both innovations models and input/output models. If an innovations model is not available, `inn2unc( )` internally estimates a stochastic model to make the computation of the Fisher information matrix possible.

These estimates take some time to compute. They only are valid under the assumption that the model is close to the true system and under the regular assumptions of prediction error methods. The `ls( )` and `sds( )` GUIs use these functions internally and represent the model error estimate as a band around the estimated model frequency response.

Least Squares Prediction Error Norms

For the special case of least squares, computation of error norms on both the identification or the validation data set can efficiently be done using the square root:

```
[sysi,sr_id] = ls(y_prbs2(1:1024), u_prbs2(1:1024),10);
[sysv,sr_val] = ls(y_prbs2(1025:2048),
u_prbs2(1025:2048),10);
var_id = ls2var(sr_id)
var_val = ls2var(sr_id,{srval=sr_val})
```

A similar function is available for the instrumental variables method: `giv2var`.

Pole/Zero Inspection

You can compute poles and zeros with a plot showing their locations using the `polezero( )` function:

```
[pls, zrs] = polezero(sys);
```

Non-square systems generically have no zeros.

Interactive Validation Tool

`val( )` provides an interactive tool that you can use for general validation of input-output data with a system model or for general response analysis of a given system. It is instantiated by calling the `val( )` function with either a single discrete-time system model (ARMA, backward polynomial, state-space, or state-space innovations) or a model and input and output data. Within the tool, you have the facility to validate any saved system model with any data set through the standard options available through the **Validation** menu. `val( )` is particularly helpful for validating models obtained with identification routines (including your own) that do not include their own interactive tools.

To validate a least squares model, complete the following steps.

1. Provide the inputs:

   ```
   [sys,sr] = ls(y_prbs2,u_prbs2,8,{inn});
   val(sys)
   ```

   The interactive tool comes up displaying the frequency response of `sys`. Examining the **Validation** menu, notice that only the frequency response, impulse response, and pole-zero plot options are currently available. Validate `sys` with data from different data sets.

2. Enter `sys` in **System Name**, `u_prbs2` in **Input Data**, and `y_prbs2` in **Output Data**, remembering to press <Return> or <Enter> after each entry.

   Notice that all the options in the **Validation** menu are now enabled.

3. As a partial validation of the noise model, select **Validation» Prediction Errors»Input - Output model**.

   Notice the percentage errors (34.77% and 30.10%) shown at the top of the plot.

4. Change the **Input Data** to `u_ss2` and **Output Data** to `y_ss2`.

5. Click **RECOMPUTE**.

   The errors increase to 49.81% and 26.71% when the `sys_ls` model is validated with data it was not identified with.

6. Select **Validation»Frequency Response»Input - Output model (Magnitude)** to plot the magnitude response of `sys_ls`.

7. Select **File»Compare With Data** and enter `main.g_etfe` as the data variable to compare this response with that generated with `etfe( )` by s.

This example illustrates the flexibility of `val( )` as a utility for comparing different models validated over different data sets.

Guidelines

- You should check assumptions on the data or filtered quantities that are used as a basis for the identification algorithm. In the case of prediction error methods, for instance, check conditions like whiteness of prediction errors. For the open-loop case, you would expect that the prediction errors are uncorrelated with the input and in the closed-loop case, uncorrelated with past inputs only.
- The most reliable method of determining the optimal order for a given model structure is *cross validation*, where two data sets are available;

use one set for identification and the other one for validation. If the model order is too large, then the additional freedom in the parameter vector results in an increased prediction error norm on the validation data set. This functionality is available through the general validation interactive tool provided by `val( )`. Refer to the *Interactive Validation Tool* section for more information.

- Comparing the consistency of models obtained by different identification algorithms and/or using different data sets can be useful. If, for instance, an empirical transfer function estimate gives an entirely different result than a high order least-squares estimate, then that is an indication that something is wrong with one or both of the models. However, the opposite does not necessarily hold true.
- Validation is a lot easier in the case of model reduction. It can simply be done by comparing the result with the known original model as opposed to the identification case.
- Because least-squares, instrumental variables, and subspace models of different orders are obtained easily, these methods offer a lot of validation opportunities. You might compare the variances of prediction errors for different model orders. The usual plot of variance versus order shows a sharp decline for the first couple of model orders, from which a minimal order can be deduced. You can compare frequency responses of models for different orders and hope to observe a convergence as the model order increases. This is most easily done using the GUIs.

Input Design

Input sequence design is directly related to the signal-to-noise ratio and is therefore an important factor in the identification procedure. Input design is impossible without prior information. Sometimes that prior information is obtained from earlier identification tests. As the theory of input design is not quite ready for translation to practical design algorithms, take a more pragmatic approach to the problem; you want to construct input signals which have a fairly constant power over a certain frequency band. Typically, this frequency band covers the system bandwidth, which is assumed to be roughly known. More general kinds of input sequence can be obtained by adding several of these signals or filtering them.

In engineering, *sine sweeps* or *chirps* are commonly used for the excitation of one input component at a time. They can be considered as sine functions

based on frequency, which is a slowly increasing linear function of time. The function is called as follows:

```
dt = 2;
[ss,t] = sweep(0,0.125,1*dt,2048*dt,dt,512*dt);
plot(ss)
```

The easiest way to explain how this function works is to consider `ss`, a discretized continuous-time signal where the time is measured in seconds and where the discretization interval is `dt` seconds. This call generates a sine sweep that goes from 0 to 0.125 Hz over the time interval `dt` to `2048*dt` where the frequency increases from its minimum value to its maximum in `512*dt` seconds; then goes down again. The result is shown in Figure 4-41.

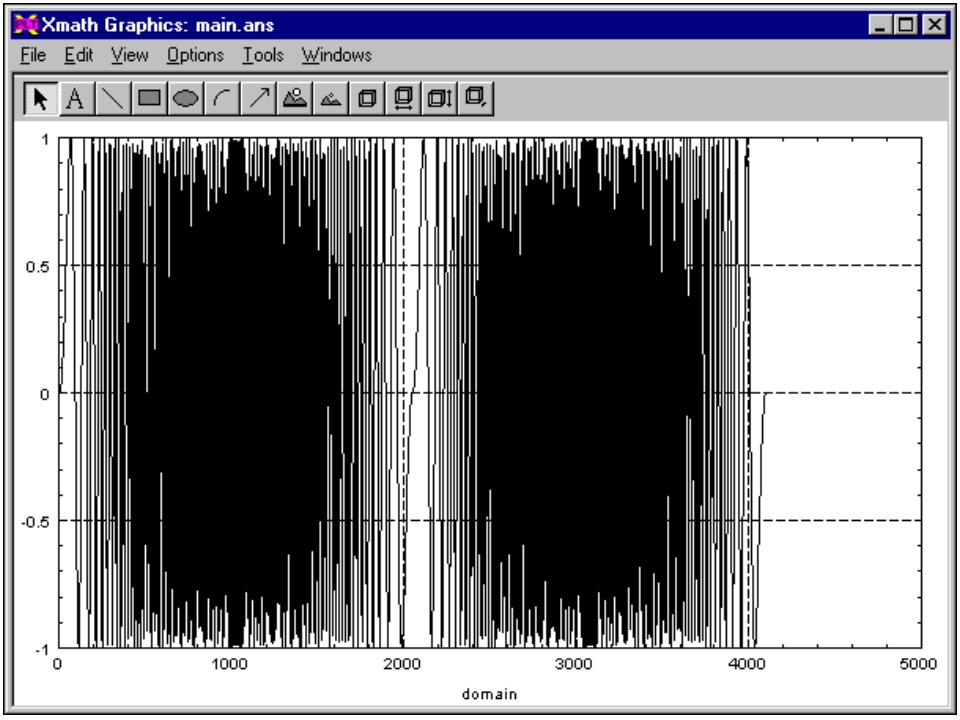

Figure 4-41. Sine Sweep

Another popular signal is the so-called pseudo-random binary sequence (PRBS), generated by the `prbs` function:

```
p = 2*(prbs(9,[1,1,1,1,1,0,0,0,0])-.5); plot(p)?
```

This call generates a PRBS of $2^9 - 1$ samples which is uncorrelated over its entire data length. The original values of the PRBS are 0 and 1, so those of `p` are –1 and 1. The second input parameter is optional and determines the initial state of the PRBS register. This initial sequence can be recognized as the last nine values, due to the periodicity of the signal. You can refer to this plot in Figure 4-42.

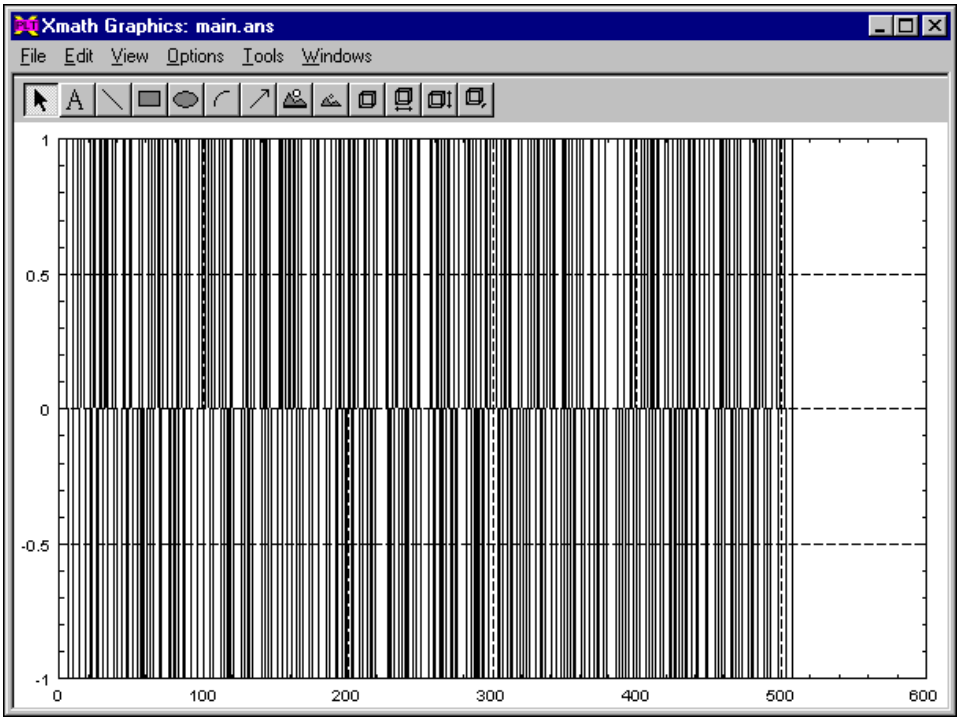

Figure 4-42. PRBS

Because of its whiteness, the PRBS has constant power over all frequencies. However, it is possible to make the bandwidth `k` times smaller by keeping the signal constant over `k` constant values (for example, `k = 2`) as follows:

```
nsamp = 2**9-1;
p_lf(1:nsamp:2) = p(1:nsamp/2+1);
p_lf(2:nsamp:2) = p(1:nsamp/2);
```

Another important input signal, especially with simulations, is random Gaussian noise. This can be generated using the core Xmath function `random( )`:

```
set distribution normal
r = random(nsamp,1)
```

A

List Data Structures

This appendix contains the list objects created to hold data from special models.

The preferred object storage method in MathScript is the MathScript Object (MSO) method described in *Xmath User Guide*. The objects described in this appendix were created before MSOs were available.

ARMA Models

ARMA systems are currently represented as 10-element list objects. Each element field contains a particular data variable associated with the ARMA system as listed in Table A-1.

Table A-1. ARMA Systems

Field	Variable	Definition
1	"ARMA"	A string used within the module functions to identify this list as an ARMA system.
2	outputnames	A vector of strings representing the names of the outputs of the system.
3	inputnames	A vector of strings representing the names of the inputs of the system.
4	dt	The sampling interval of the system data.
5	ϑ_B	A matrix containing the B_k matrices in the format $\vartheta_B = [B_0, B_1, B_2, ..., B_{n_B}]$.
6	ϑ_A	A matrix ϑ_A containing the A_k matrices in the format $\vartheta_A = [I, A_1, A_2, ..., A_{n_A}]$.
7	n_y	Number of system outputs.
8	n_u	Number of system inputs.

Table A-1. ARMA Systems (Continued)

Field	Variable	Definition
9	n_A	Order of the A polynomial.
10	n_B	Order of the B polynomial.

ARMA objects can be created directly from data and operated on using ISID functions. They also can be converted to and from standard Xmath state-space system objects. For more details on functions using or returning ARMA objects, refer to the *Xmath Help*.

Backwards-Polynomial Innovations Model Implementation

The ISID Module uses a list-based object to represent any of the varieties of backward polynomial models. Because this list is designed to accommodate the most general model of this structure, it has seventeen fields, not all of which need to be explicitly specified when you are creating one of the more specific model types (the C, D, and F matrices default to identity as necessary). The structure is listed in Table A-2.

Table A-2. Model Structure

Field	Variable	Definition
1	BP Innovations Model	A string used within the module functions to identify this list as a backward polynomial model.
2	outputnames	A vector of strings representing the names of the outputs of the system.
3	inputnames	A vector of strings representing the names of the inputs of the system.
4	dt	The sampling interval of the system data.
5	ϑ_B	A matrix ϑ_A containing the A_k matrices in the format: $\vartheta_A = [I, A_1, A_2, \ldots, A_{n_A}]$.

Table A-2. Model Structure (Continued)

Field	Variable	Definition
6	ϑ_A	A matrix ϑ_B containing the B_k matrices in the format: $\vartheta_B = [B_0, B_1, B_2, \ldots, B_{n_B}]$.
7	ϑ_F	A matrix ϑ_F containing the F_k matrices in the format $\vartheta_F = [I, F_1, F_2, \ldots, F_{n_F}]$.
8	ϑ_C	A matrix ϑ_C containing the C_k matrices in the format $\vartheta_C = [I, C_1, C_2, \ldots, C_{n_C}]$.
9	ϑ_D	A matrix ϑ_D containing the D_k matrices in the format $\vartheta_D = [I, D_1, D_2, \ldots, D_{n_D}]$.
10	var	A square $n_y \times n_y$ matrix indicating the noise variance on the outputs.
11	n_y	Number of outputs.
12	n_u	Number of inputs.
13	n_A	Order of the A polynomial.
14	n_B	Order of the B polynomial.
15	n_F	Order of the F polynomial.
16	n_C	Order of the C polynomial.
17	n_D	Order of the D polynomial.

Figure A-1 illustrates the connections between ISID functions that create these different system models and convert from one representation to another.

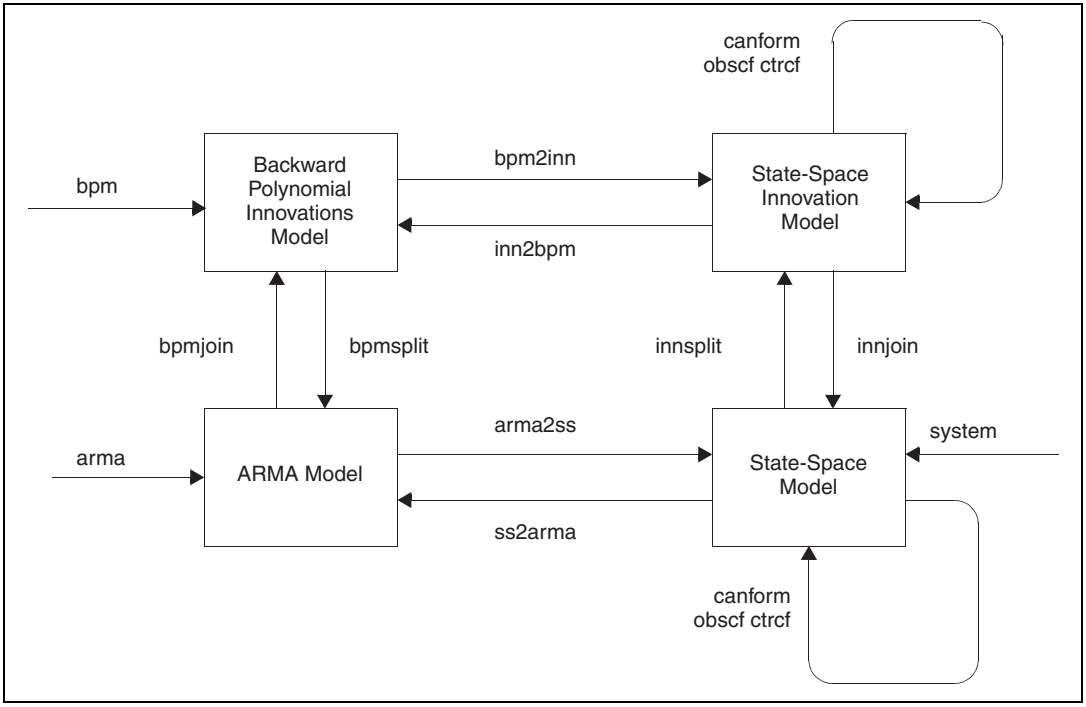

Figure A-1. Model Conversion

LS Square Root

The output parameter `sr` is a list object containing the S matrix as well as relevant structural parameters pertaining to the identification including the number of inputs and outputs, orders of A and B polynomials, and the number of regressions used.

Square root objects are 10-element lists structured as described in Table A-3.

Table A-3. ARMA Systems

Field	Definition
1	"LS Square Root" or "Lattice Square Root" (label indicating the type of square root object).
2	Output names.
3	Input names.

Table A-3. ARMA Systems (Continued)

Field	Definition
4	Sampling interval dt.
5	S matrix.
6	Number of outputs n_y.
7	Number of inputs n_u.
8	Order of B polynomial.
9	Order of A polynomial.
10	Number of input/output data samples. These parameters contain all information required for ARX models up to the minimum of the orders of the A and B polynomials.
11	Vector containing indices of the output regressions.
12	Vector containing indices of the input regressions.

B

Loading Data with the read( ) Function

In a typical scenario, data is obtained as discrete-time input and output measurements from a computer associated with the system to be identified. Depending on the data-measuring program, the data can be in ASCII or binary format.

Pure numeric data is typically most easily brought into Xmath through the `read( )` function. This is a generic function available with the Xmath core that allows you to read any type of formatted numeric data into an Xmath matrix variable. You can find specific information on this function in the *Xmath Help* or the *Xmath User Guide*.

To use `read( )`, you need to know the name of the file containing the data, the dimensions of the matrix you want the data to occupy, and the format of the data file. By default, `read( )` begins reading data from the beginning of the file. You have the option of specifying an amount of data to be skipped if you want to begin reading in data from the middle of the file.

The `read( )` function, shown in Example B-1, is better suited to reading in numeric data than text data. If you have multiple data files with a standard format, you may want to create your own script files or functions calling `read( )` to load in the data from these files.

The file `$XMATH/demos/ship.ascii` contains raw data corresponding to the steering angle theta of a boat (the system input), a vector of the time points at which each measurement was made, and x and y measurements of the boat position (the system output).

`ship.ascii` contains all data in ASCII format. Each information set corresponds to 101 time points. While this is a very small data set in terms of most identification problems, it is fairly typical in that you know the length of the data records ahead of time.

Example B-1 formats the data for the angle theta and the time as 101×1 vectors and the position data as a 101×2 matrix.

Example B-1 Calls to Read to Obtain the Data from ship.ascii

```
theta = read("$XMATH/demos/ship.ascii", 101, 1, "ascii");
time = read("$XMATH/demos/ship.ascii", 101, 1, "ascii",13);
position = read("$XMATH/demos/ship.ascii", 101, 2, "ascii",27);
```

Note The second and third calls to `read( )` include an offset indicating the number of lines to skip before beginning to read from the file.

C

Tool-Specific GUI Features

This appendix lists the approach-specific features of each tool:

- Least Squares GUI (LS)
- Instrumental Variables GUI (GIV)
- Empirical Transfer Function GUI (ETFE)
- Impulse Realization GUI (IREA)
- Frequency Domain Least Squares GUI (FWLS)
- Deterministic/Stochastic Subspace GUI (SDS)
- Stochastic Subspace GUI (SST)
- Validation GUI (VAL)

Least Squares GUI

Algorithm options include the following:

- **Error Norms**—Plots the diagonal terms of the prediction error variance matrix.
- **SDF Prediction Errors**—Depending on your selection from a walking submenu, plots the magnitude or phase of the spectral density function prediction errors. Magnitude is plotted as a direct ratio (no units); phase is plotted in degrees.
- **Frequency Weight**—This option is valid only for the frequency-weighted least squares function `fwls( )`. Refer to the *Least Squares in the Frequency Domain* section of Chapter 4, *Tutorial*, for more information. It is inactive for `ls( )`.
- **SV Selection**—Brings up a bar plot showing the singular values. This can be a useful guide in determining how many of the singular values should be retained in re-identifying the data with an SVD-based solution.
- **Defaults**—Allows you to modify the computational method used, the type, width, and overlap of the windows used over the data, the order of the SDFs and autoregressions, the frequency vector over which the model's frequency response is computed, and finally, the maximum and minimum values of any scaling weight function.

- **Recompute**—Provides the same functionality as the **RECOMPUTE** button but includes a key binding.

You can change a number of modeling options interactively with the toggle buttons and editable labels beneath the plot area. After you use them to make the desired changes to the identification, you can press **RECOMPUTE** to re-identify the data and update the displayed plots.

Below the plots, on the left side of the `ls` interactive tool are three VarEdit widgets:

- **Order A polynomial**—Allows you to modify the order of the A polynomial to any value no greater than the value with which the tool was launched. When you enter a value in this widget and press <Return> or <Enter>, that value is used for the orders of both the A and the B polynomials. This is done because identifications are frequently performed using the same order for both polynomials. However, you can set the B polynomial order separately, as described below.
- **Order B polynomial**—Allows you to modify the order of the B polynomial to any value no greater than the value with which the tool was launched. When you enter a value in this widget and press <Return> or <Enter>, that value is used for the order of the B polynomial only.
- **Number of SVs**—Allows you to set the number of singular values to be retained in using an SVD-based solution. Refer to the *Singular Value-Based Solutions* section of Chapter 3, *Identification Algorithms*, for more information. When you enter the number n, the n largest values are retained and used in the solution, while the rest are set to zero.

Four checkboxes appear near the bottom of the interactive tool:

- **Feedthrough Term**—If enabled, the model is recomputed using a feedthrough term B_0 in Equation 3-1. If the tool is called with the `{feed}` keyword specified, this checkbox is enabled.
- **Scalar Denominator**—If enabled, the model is recomputed with the assumption that all the elements of the multivariable transfer function have the same poles. Refer to the *Least Squares with Scalar Denominator* section of Chapter 3, *Identification Algorithms*, for more information. If the tool is called with the `{scden}` keyword specified, this checkbox is enabled.

- **SVD Solution**—If enabled, the model is recomputed using a solution based on a singular-value decomposition of the LS square root. If the `{nsvd}` keyword is set equal to some nonzero value when the tool is called, this checkbox is enabled.
- **Cross Validation**—If you use the `{yval}` and `{uval}` or `{srval}` keywords when calling `ls` with the interactive tool, this checkbox is enabled, indicating that a validation data set is available.

  The **Cross Validation** checkbox in the interactive tool is enabled in our examples because we specified validation data when we called `ls( )`.

Generalized Instrumental Variables Tool

The generalized instrumental variables (GIV) interactive tool is fairly simple. The Algorithm menu contains the following options:

- **Error Norms**—Plots the diagonal terms of the prediction error variance matrix.
- **Defaults**—Allows you to specify the frequency vector to be used in calculating the frequency response of the identified system.
- **Recompute**—Provides the same functionality as the **RECOMPUTE** button; includes a keyboard accelerator.

One checkbox appears in the interactive tool:

- **Feedthrough Term**—If enabled, the model is recomputed using a feedthrough term B_0 in the *Generalized Instrumental Variables* section of Chapter 3, *Identification Algorithms*.

There are three VarEdit widgets below the plots in the GIV interactive tool:

- **Model Order**—The order of the current ARMA system.
- **IV Lags Past**—The number of past lags used in the instrumental variables least squares fit.
- **IV Lags Future**—The number of future lags used in the instrumental variables least squares fit.

Empirical Transfer Function GUI

The menu layout of the `etfe` tool is as discussed in the *General Features of ISID Interactive Tools* section of Chapter 4, *Tutorial*; however, the Validation menu is inactive for this tool because `etfe` returns PDMs for the system frequency response and spectral density noise rather than one of the system models described in *Model Structures* section of Chapter 4, *Tutorial*. Validation of the `etfe`-identified response is accomplished through the Algorithm menu. The Algorithm menu options are as follows:

- **ETFE**—A walking submenu lets you select whether to plot the magnitude or phase of the empirical transfer function estimate.
- **SDF**—Computes and plots the spectral density function of the output with the input data. A walking submenu lets you select a magnitude or phase plot of the SDF.
- **SDF Additive Noise**—Plots the spectral density function of the additive model noise. A walking submenu lets you select a magnitude or phase plot.
- **Coherence**—Plots the magnitude or phase (depending on submenu selection) of the coherence of the output with the input data.
- **Covariance**—Computes and plots the covariance of the output with the input data; performs the ETFE identification using covariance-averaging instead of SDF-averaging.
- **Covariance Additive Noise**—Plots the noise computed with an ETFE obtained by covariance averaging.
- **Impulse Response**—Plots the impulse response obtained as the real part of the inverse Fourier transform of the frequency-domain ETFE. A walking submenu allows you to plot the unweighted response directly or to weight the ETFE interactively using the mouse and then recompute and plot the weighted impulse response.
- **Display Signals**—Plots the output and input data signals in strip plot format.
- **Recompute**—Provides the same functionality as the **RECOMPUTE** button, including a keyboard accelerator.

Two more menu options appear below the plotting area:

- **Computational Method**—As discussed in the *Spectral Density Function Estimation* section of Chapter 3, *Identification Algorithms*, the spectral density functions from which the ETFE is computed can be obtained in several ways: by averaging spectral density function estimates, averaging covariances, or by an autoregressive method.

Changing the computation method through this menu and clicking **RECOMPUTE** recomputes the ETFE and updates the plot.

- **Window Type**—Allows you to change the type of data windowing interactively. The mathematical formulas for the window types (Hamming, Hanning, Blackman, Triangular, and Rectangular) are given in the *Spectral Density Function Estimation* section of Chapter 3, *Identification Algorithms*.

Below the plot are three checkboxes labeled **Select input**, **Select output**, and **Select reference**. You can enable any combination of these to determine which signals to use in computing the ETFE. If the system is open-loop, the reference signal is taken to be the same as the input signal; otherwise, you can explicitly specify a reference signal with the keyword `{ref}` when you call `etfe( )`.

Three VarEdits are located near the bottom of the tool:

- **Window width/Frequency points**—Allows you to change the number of points in each window over the data when you are using the SDF or covariance computational method. If you are using the autoregressive approach, the value in this VarEdit is used as the number of frequency points to use in the computation.
- **SDF/AR order**—This VarEdit should be set to the number of lags desired when using the autoregressive approach.
- **Overlap**—Allows you to change the number of overlapping windows in which each data point appears.

Impulse Realization GUI

The Algorithm menu has two `irea`-specific options:

- **SV Selection**—Brings up a bar plot of the system Hankel singular values.
- **Defaults**—Brings up a popup window thorough which you can change either the computational method used (refer to the *Identification from Impulse Response Data* section of Chapter 3, *Identification Algorithms*) or the vector of frequencies to be used in computing the system frequency response as part of the identification validation.

Three VarEdit widgets are located beneath the plot area of the tool:

- **Number of block rows**—Allows you to modify the number of block Hankel rows used in the identification.
- **Number of block columns**—Allows you to modify the number of block Hankel columns used in the identification.
- **Number of SVs**—Allows you to choose the number of singular values `nsv` to retain in forming the identified state-space system. All singular values smaller than the `nsv` largest singular values are set to zero. You can change the vector frequency range over which to evaluate the frequency response of the system.

A combo box labeled beneath these VarEdits shows and allows you to change the current computational **Method** (Zeiger/McEwen). The computation method also can be changed through the **Algorithm»Defaults** menu.

Frequency Domain Least Squares GUI

The interactive tool for `fwls` resembles that for `ls( )` and offers similar options through the **Algorithm** menu. Refer to the *Least Squares GUI* section for a listing and description of these options. The key difference is that the **Algorithm»Frequency weight option** is enabled for the `fwls( )` tool.

Deterministic/Stochastic Subspace GUI

The options listed on the **Algorithm** menu are as follows:

- **Select Order**—Can be used to regenerate the bar plot of singular values or principal angles shown when the interactive tool is first instantiated.
- **Defaults**—Brings up a popup in which you can modify the identification bias, the type of basis to use, scaling factors, and whether or not to use the lattice algorithm. The validation parameters you can change are the range and number of points to use in computing the system's frequency response, as well as the number of lags to use in computing the covariance of the prediction error and the impulse response.
- **Recompute**—Provides the same functionality as does the **RECOMPUTE** button with an associated keybinding.

Two VarEdit widgets appear beneath the plotting area:

- **Model Order**—Allows you to change the order of the model to be identified.
- **Number of Block Rows**—Allows you to change the number of block rows to use in forming the Hankel matrix used in the identification.

Stochastic Subspace GUI

The options listed on the **Algorithm** menu are as follows:

- **Select Order**—Can be used to regenerate the bar plot of singular values or principal angles shown when the interactive tool is first instantiated.
- **Defaults**—Brings up a popup in which you can modify the type of basis to use, scaling factors, and whether or not to use the lattice algorithm. The validation parameters you can change are the range and number of points to use in computing the system's frequency response, as well as the number of lags to use in computing the covariance of the prediction error and the impulse response.
- **Recompute**—Provides the same functionality as does the **RECOMPUTE** button with an associated keybinding.

Two VarEdit widgets appear beneath the plotting area:

- **Model Order**—Allows you to change the order of the model to be identified.
- **Number of Block Rows**—Allows you to change the number of block rows to use in forming the Hankel matrix used in the identification.

Validation GUI

The **Algorithm** menu for the `val( )` tool contains two options:

- **Defaults**—Selecting this option brings up a popup in which you can select the minimum and maximum frequencies and the number of points to be used in computing frequency responses, as well as the number of covariance and impulse lags.
- **Recompute**—This option and the associated keyboard accelerator have the identical function as the **RECOMPUTE** button beneath the plot area.

Three VarEdits appear beneath the plotting area of the interactive tool:

- **System Name**—Allows you to enter the name of a system model to be validated. Data is assumed to be in the current partition; specify the variable name in `partitionName.variableName` form if you want to use a variable in another partition.
- **Input Data**—Allows you to enter the name of a variable containing the input data with which to validate the current system model. Data is assumed to be in the current partition; specify the variable name in `partitionName.variableName` form if you want to use a variable in another partition.
- **Output Data**—Allows you to enter the name of a variable containing the output data with which to validate the current system model. Data is assumed to be in the current partition; specify the variable name in *partitionName.variableName* form if you want to use a variable in another partition.

D

Bibliography

[AD87] J. L. Adcock: *Curve Fitter for Pole-Zero Analysis*, Hewlett-Packard Journal, January 1987, p. 33.

[AKA] H. Akaike: *Markovian representation of stochastic processes by canonical variables*, SIAM J. Control, Vol. 13, No. 1, pp. 162–173, 1975.

[ALING] H. Aling: *A Fast Least Squares Lattice Algorithm*, Proc. IEEE-CDC Tucson, 1993, pp. 3709–3710.

[AndGev] B.D.O. Anderson, M. Gevers: *Identifiability of linear stochastic systems operating under feedback*, Automatica Vol. 18, No. 2, pp. 195–213, 1982.

[AndMo] B.D.O. Anderson, J.B. Moore: Optimal Filtering. ISBN 0-13-638122-7, Prentice Hall, New Jersey, 1979.

[ARUN] K.S. Arun, S.Y. Kung: *Balanced Approximation of Stochastic Systems.* SIAM Matrix Analysis and Applications, 11, 1990, pp. 42–68.

[BIER] G.J. Bierman: *Factorization methods for discrete sequential estimation*, Academic Press, New York.

[DAV] Davies, W.D.T., *Generation and properties of maximum-length sequences*, parts 1–3, Control, June, July, and August 1966.

[Enns] Enns, Dale: *Model Reduction for Control System Design, A Report to NASA Ames-Dryden Flight Research Facility.* Stanford Electronics Laboratories, Dept. of Electrical Engineering, Stanford University, Stanford, CA, 1984.

[GMW] Gill, P.E., Murray, W. and Wright, M.H. Practical Optimization, Academic Press Inc., N.Y., N.Y., 1981, pp. 105–115.

[GP] G.C. Goodwin, R.L. Payne: *Dynamic system identification: Experiment design and data analysis*, ISBN 0-12-289750-1, Academic Press, 1977.

[KAI] T. Kailath: Linear Systems, ISBN 0-536961-4, Prentice-Hall, 1980.

[LARI] W.E. Larimore: *System identification, reduced order filtering and modelling via canonical variate analysis*, Proc. ACC 1983, San Francisco, 1984.

[LARI2] W.E. Larimore: *Canonical Variate Analysis in Identification, Filtering, and Adaptive Control.* 29th IEEE Conference On Decision and Control, Honolulu, Hawaii, December 1990, pp. 596–604.

[LJU1] L. Ljung, T. Söderström: *Theory and practice of recursive identification*, ISBN 0-262-12095-X, MIT Press, 1983.

[LJU2] L. Ljung: *System identification – theory for the user*, ISBN 0-13-881640-9, Prentice-Hall, 1987.

[MOON] M. Moonen, B. De Moor, L. Vandenberghe, J. Vandewalle: *On- and off-line identification of linear state-space models*, Int. J. Control, 1989, Vol. 49, No. 1, pp. 219–232.

[MOOR] B. De Moor: *Mathematical concepts and techniques for modelling of static and dynamic systems*, Ph.D. thesis, Katholieke Universiteit Leuven, Belgium, UDC 519.17.

[PET] V. Peterka: *A square root filter for real time multivariate regression*, Kybernetika, Vol. 11, No. 1, pp. 53–67, 1975.

[POR] B. Porat, B. Friedlander, M. Morf: *Square root covariance ladder algorithms*, IEEE-AC, Vol. 27, No. 4, pp. 813–829, 1982.

[PRIES] M.B. Priestley: *Spectral analysis and time series*, ISBN 0-12-564922-3, Academic Press, 1981.

[Söd] T. Söderström, L. Ljung, I. Gustavsson: *Identifiability conditions of linear multivariable systems operating under feedback*, IEEE-AC Vol. 21, No. 6, pp. 837, 840, 1976.

[VODM1] P. Van Overschee, B. De Moor: *Subspace Algorithms for the Identification of Combined Deterministic-Stochastic Systems.* Automatica, Special Issue on Statistical Processing and Control, Vol. 30, No. 1, pp. 75–93.

[VODM2] P. Van Overschee, B. De Moor: *Subspace Algorithms for the Stochastic Identification Problem.* Automatica, Vol. 29, No. 3, pp. 649–660.

[WAH] B. Wahlberg, L. Ljung: *Design variables for bias distribution in transfer function estimation*, IEEE-AC, Vol. 31, No. 2, pp. 134–144, 1986.

E

Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Online technical support resources at ni.com/support include the following:
  - **Self-Help Resources**—For immediate answers and solutions, visit the award-winning National Instruments Web site for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on.
  - **Free Technical Support**—All registered users receive free Basic Service, which includes access to hundreds of Application Engineers worldwide in the NI Developer Exchange at ni.com/exchange. National Instruments Application Engineers make sure every question receives an answer.
- **Training and Certification**—Visit ni.com/training for self-paced training, eLearning virtual classrooms, interactive CDs, and Certification program information. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, NI Alliance Program members can help. To learn more, call your local NI office or visit ni.com/alliance.

If you searched ni.com and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Index

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z