System Identification Toolbox

For Use with MATLAB®

Lennart Ljung

Computation

Visualization

Programming

User's Guide

Version 5

How to Contact The MathWorks:

508-647-7000 Phone

508-647-7001 Fax

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098 Mail

`http://www.mathworks.com`	Web
`ftp.mathworks.com`	Anonymous FTP server
`comp.soft-sys.matlab`	Newsgroup

`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`subscribe@mathworks.com`	Subscribing user registration
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

System Identification Toolbox User's Guide

Printing History: April 1988 First printing
July 1991 Second printing
May 1995 Third printing
November 2000 Fourth printing for Version 5.0 (Release 12)

Contents

Preface

1 The System Identification Problem

2 The Graphical User Interface

3 Tutorial

4 Command Reference

Preface

What Is the System Identification Toolbox?

The System Identification Toolbox is for building accurate, simplified models of complex systems from noisy time-series data.

It provides tools for creating mathematical models of dynamic systems based on observed input/output data. The toolbox features a flexible graphical user interface that aids in the organization of data and models. The identification techniques provided with this toolbox are useful for applications ranging from control system design and signal processing to time-series analysis and vibration analysis.

Using This Guide

System Identification is about building mathematical models of dynamic systems based on measured data. Some knowledge about such models is therefore necessary for successful use of the toolbox.The topic is treated in several places in Chapter 3, “Tutorial” and there is a wide range of textbooks available for introductory and in-depth studies. For basic use of the toolbox, it is sufficient to have quite superficial insights about dynamic models. For review of basic knowledge, see “How do I get started?” on page 1-3.

If you are a beginner, browse through Chapter 2, “The Graphical User Interface” and try out a couple of the data sets that come with the toolbox. Use the graphical user interface (GUI) and check out the built-in help functions to understand what you are doing.

Typographical Conventions

We use some or all of these conventions in our manuals.

Item	Convention to Use	Example
Example code	Monospace font	To assign the value 5 to A, enter `A = 5`
Function names/syntax	Monospace font	The cos function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Keys	**Boldface** with an initial capital letter	Press the **Return** key.
Literal strings (in syntax descriptions in Reference chapters)	**`Monospace bold`** for literals.	`f = freqspace(n,'whole')`
Mathematical expressions	Variables in *italics* Functions, operators, and constants in standard text.	This vector represents the polynomial $p = x^2 + 2x + 3$

MATLAB output	Monospace font	MATLAB responds with
		`A =`
		`5`

Menu names, menu items, and controls	**Boldface** with an initial capital letter	Choose the **File** menu.
New terms	*Italics*	An *array* is an ordered collection of information.
String variables (from a finite list)	*Monospace italics*	`sysc = d2c(sysd, 'method')`

Related Products

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with the System Identification Toolbox. In particular, the Systems Identification Toolbox *requires* these products:

- MATLAB®

For more information about any of these products, see either:

- The online documentation for that product, if it is installed or if you are reading the documentation from the CD
- The MathWorks Web site, at `http://www.mathworks.com`; see the "products" section

Note The products listed below complement the functionality of the System Identification toolbox.

Product	Description
Simulink®	Interactive, graphical environment for modeling, simulating, and prototyping dynamic systems
Control System Toolbox	Tool for modeling, analyzing, and designing control systems using classical and modern techniques
Data Acquisition Toolbox	MATLAB functions for direct access to live, measured data from MATLAB
Financial Time Series Toolbox	Tool for analyzing time series data in the financial markets
Financial Toolbox	MATLAB functions for quantitative financial modeling and analytic prototyping
Fuzzy Logic Toolbox	Tool to help master fuzzy logic techniques and their application to practical control problems

Product	Description
-Analysis and Synthesis Toolbox	Computational algorithms for the structured singular value, μ, applicable to robustness and performance analysis for systems with modeling and parameter uncertainties
Neural Network Toolbox	Comprehensive environment for neural network research, design, and simulation within MATLAB
Optimization Toolbox	Tool for general and large-scale optimization of nonlinear problems, as well as for linear programming, quadratic programming, nonlinear least squares, and solving nonlinear equations
Robust Control Toolbox	Tools for modeling, analysis, and design of “robust” multivariable feedback control systems using H_∞ techniques
Signal Processing Toolbox	Tool for algorithm development, signal and linear system analysis, and time-series data modeling
Statistics Toolbox	Tool for analyzing historical data, modeling systems, developing statistical algorithms, and learning and teaching statistics

About the Author

Lennart Ljung received his PhD in Automatic Control from Lund Institute of Technology in 1974. Since 1976 he is Professor of the chair of Automatic Control in Linkoping, Sweden, and is currently Director of the Center for the “Information Systems for Industrial Control and Supervision” (ISIS). He has held visiting positions at Stanford and MIT and has written several books on System Identification and Estimation. He is an IEEE Fellow, an IFAC Advisor, a member of the Royal Swedish Academy of Sciences (KVA) and of the Royal Swedish Academy of Engineering Sciences (IVA), and has received honorary doctorates from the Baltic State Technical University in St Petersburg, and from Uppsala University.

1

The System Identification Problem

Basic Questions About System Identification

What is System Identification?

System Identification allows you to build mathematical models of a dynamic system based on measured data.

How is that done?

Essentially by adjusting parameters within a given model until its output coincides as well as possible with the measured output.

How do you know if the model is any good?

A good test is to take a close look at the model's output compared to the measured one on a data set that wasn't used for the fit ("Validation Data").

Can the quality of the model be tested in other ways?

It is also valuable to look at what the model couldn't reproduce in the data ("the residuals"). This should not be correlated with other available information, such as the system's input.

What models are most common?

The techniques apply to very general models. Most common models are difference equations descriptions, such as ARX and ARMAX models, as well as all types of linear state-space models.

Do you have to assume a model of a particular type?

For parametric models, you have to specify the structure. This could be as easy as just selecting a single integer, the model order, or may involve several choices.If you just assume that the system is linear, you can directly estimate its impulse or step response using Correlation Analysis or its frequency response using Spectral Analysis. This allows useful comparisons with other estimated models.

What does the System Identification Toolbox contain?

It contains all the common techniques to adjust parameters in all kinds of linear models. It also allows you to examine the models' properties, and to check if they are any good, as well as to preprocess and polish the measured data.

Isn't it a big limitation to work only with linear models?

No, actually not. Many common model nonlinearities are such that the measured data should be nonlinearly transformed (like squaring a voltage input if you think that it's the power that is the stimuli). Use physical insight about the system you are modeling and try out such transformations on models that are linear in the new variables, and you will cover a lot!

How do I get started?

If you are a beginner, browse through Chapter 2, "The Graphical User Interface." Then try out a couple of the data sets that come with the toolbox. Use the graphical user interface (GUI) and check out the built-in help functions to understand what you are doing.

Is this really all there is to System Identification?

Actually, there is a huge amount written on the subject. Experience with real data is the driving force to understand more. It is important to remember that any estimated model, no matter how good it looks on your screen, has only picked up a simple reflection of reality. Surprisingly often, however, this is sufficient for rational decision making.

Common Terms Used in System Identification

This section defines some of the terms that are frequently used in System Identification:

- **Estimation Data** is the data set that is used to fit a model to data. In the GUI this is the same as the **Working Data**.
- **Validation Data** is the data set that is used for model validation purposes. This includes simulating the model for these data and computing the residuals from the model when applied to these data.
- **Model Views** are various ways of inspecting the properties of a model. They include looking at zeros and poles, transient and frequency response, and similar things.
- **Data Views** are various ways of inspecting properties of data sets. A most common and useful thing is just to plot the data and scrutinize it. So-called *outliers* could be detected then. These are unreliable measurements, perhaps arising from failures in the measurement equipment. The frequency contents of the data signals, in terms of periodograms or spectral estimates, is also most revealing to study.
- **Model Sets** or **Model Structures** are families of models with adjustable parameters. **Parameter Estimation** amounts to finding the "best" values of these parameters. The System Identification problem amounts to finding both a good model structure and good numerical values of its parameters.
- **Parametric Identification Methods** are techniques to estimate parameters in given model structures. Basically it is a matter of finding (by numerical search) those numerical values of the parameters that give the best agreement between the model's (simulated or predicted) output and the measured one.
- **Nonparametric Identification Methods** are techniques to estimate model behavior without necessarily using a given parametrized model set. Typical nonparametric methods include **Correlation analysis**, which estimates a system's impulse response, and **Spectral analysis**, which estimates a system's frequency response.

- **Model Validation** is the process of gaining confidence in a model. Essentially this is achieved by "twisting and turning" the model to scrutinize all aspects of it. Of particular importance is the model's ability to reproduce the behavior of the Validation Data sets. Thus it is important to inspect the properties of the residuals from the model when applied to the Validation Data.

Basic Information About Dynamic Models

System Identification is about building **Dynamic Models**. Some knowledge about such models is therefore necessary for successful use of the toolbox. The topic is treated in several places in Chapter 3, "Tutorial." Also, there is a wide range of textbooks available for introductory and in-depth studies. For basic use of the toolbox, it is sufficient to have quite superficial insights about dynamic models. This section describes such a basic level of knowledge.

The Signals

Models describe relationships between measured signals. It is convenient to distinguish between **input** signals and **output** signals. The outputs are then partly determined by the inputs. Think for example of an airplane where the inputs would be the different control surfaces, ailerons, elevators, and the like, while the outputs would be the airplane's orientation and position. In most cases, the outputs are also affected by more signals than the measured inputs. In the airplane example it would be wind gusts and turbulence effects. Such "unmeasured inputs" will be called **disturbance** signals or **noise**. If we denote inputs, outputs, and disturbances by **u**, **y**, and **e**, respectively, the relationship can be depicted in the following figure.

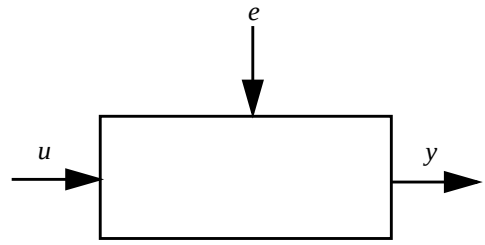


Figure 1-1: Input Signals *u*, Output Signals *y*, and Disturbances *e*

All these signals are functions of time, and the value of the input at time *t* will be denoted by *u(t)*. Often, in the identification context, only discrete-time points are considered, since the measurement equipment typically records the signals just at discrete-time instants, often equally spread in time with a **sampling interval** of *T* time units. The modeling problem is then to describe how the three signals relate to each other.

The Basic Dynamic Model

The basic relationship is the **linear difference equation**. An example of such an equation is the following one.

$$y(t) - 1.5y(t-T) + 0.7y(t-2T) = 0.9u(t-2T) + 0.5u(t-3T) \qquad (ARX)$$

Such a relationship tells us, for example, how to compute the output *y(t)* if the input is known and the disturbance can be ignored:

$$y(t) = 1.5y(t-T) - 0.7y(t-2T) + 0.9u(t-2T) + 0.5u(t-3T)$$

The output at time *t* is thus computed as a linear combination of past outputs and past inputs. It follows, for example, that the output at time *t* depends on the input signal at many previous time instants. This is what the word **dynamic** refers to. The identification problem is then to use measurements of *u* and *y* to figure out:

- The coefficients in this equation (i.e., -1.5, 0.7, etc.).
- How many delayed outputs to use in the description (two in the example: $y(t\text{-}T)$ and $y(t\text{-}2T)$).
- The **time delay** in the system is (*2T* in the example: you see from the second equation that it takes *2T* time units before a change in *u* will affect *y*).
- How many delayed inputs to use (two in the example: *u(t-2T)* and *u(t-3T)*). The number of delayed inputs and outputs are usually referred to as the **model order(s).**

Variants of Model Descriptions

The model given above is called an **ARX model**. There are a handful of variants of this model known as **Output-Error** (OE) models, **ARMAX** models, **FIR** models, and **Box-Jenkins** (BJ) models. These are described later on in the manual. At a basic level it is sufficient to think of them as variants of the ARX model allowing also a characterization of the properties of the disturbances *e*.

Linear state-space models are also easy to work with. The essential structure variable is just a scalar: the model order. This gives just one knob to turn when searching for a suitable model description. See below.

General linear models can be described symbolically by

$$y = Gu + He$$

which says that the measured output *y(t)* is a sum of one contribution that comes from the measured input *u(t)* and one contribution that comes from the noise *He*. The symbol *G* then denotes the dynamic properties of the system, that is, how the output is formed from the input. For linear systems it is called the **transfer function** from input to output. The symbol *H* refers to the noise properties, and is called the **disturbance model**. It describes how the disturbances at the output are formed from some standardized noise source *e(t)*.

State-space models are common representations of dynamical models. They describe the same type of linear difference relationship between the inputs and the outputs as in the ARX model, but they are rearranged so that only one delay is used in the expressions. To achieve this, some extra variables, the **state variables**, are introduced. They are not measured, but can be reconstructed from the measured input-output data. This is especially useful when there are several output signals, i.e., when *y(t)* is a vector. Chapter 3, "Tutorial", gives more details about this. For basic use of the toolbox it is sufficient to know that the **order** of the state-space model relates to the number of delayed inputs and outputs used in the corresponding linear difference equation. The state-space representation looks like

$$x(t+1) = Ax(t) + Bu(t) + Ke(t)$$

$$y(t) = Cx(t) + Du(t) + e(t)$$

Here *x(t)* is the vector of state variables. The model order is the dimension of this vector. The matrix *K* determines the disturbance properties. Notice that if *K = 0*, then the noise source *e(t)* affects only the output, and no specific model of the noise properties is built. This corresponds to *H = 1* in the general description above, and is usually referred to as an *Output-Error model*. Notice also that *D = 0* means that there is no direct influence from *u(t)* to *y(t)*. Thus the effect of the input on the output all passes via *x(t)* and will thus be delayed at least one sample. The first value of the state variable vector *x(0)* reflects the initial conditions for the system at the beginning of the data record. When dealing with models in state-space form, a typical option is whether to estimate *D*, *K*, and *x(0)* or to let them be zero.

How to Interpret the Noise Source

In many cases of system identification, the effects of the noise on the output are insignificant compared to those of the input. With good signal-to-noise ratios (SNR), it is less important to have an accurate disturbance model.

Nevertheless it is important to understand the role of the disturbances and the noise source *e(t)*, whether it appears in the ARX model or in the general descriptions given above.

There are three aspects of the disturbances that should be stressed:

- Understanding white noise
- Interpreting the noise source
- Using the noise source when working with the model

These aspects are discussed one by one.

How can we understand white noise? From a formal point of view, the noise source *e* will normally be regarded as *white noise*. This means that it is entirely unpredictable. In other words, it is impossible to guess the value of *e(t)* no matter how accurately we have measured past data up to time *t*-1.

How can we interpret the noise source? The actual disturbance contribution to the output, *H e*, has real significance. It contains all the influences on the measured *y*, known and unknown, that are not contained in the input *u*. It explains and captures the fact that even if an experiment is repeated with the same input, the output signal will typically be somewhat different. However, the noise source *e* need not have a physical significance. In the airplane example mentioned earlier, the disturbance effects are wind gusts and turbulence. Describing these as arising from a white noise source via a transfer function *H*, is just a convenient way of capturing their character.

How can we deal with the noise source when using the model? If the model is used just for simulation, i.e., the responses to various inputs are to be studied, then the disturbance model plays no immediate role. Since the noise source *e(t)* for new data will be unknown, it is taken as zero in the simulations, so as to study the effect of the input alone (a noise-free simulation). Making another simulation with *e* being arbitrary white noise will reveal how reliable the result of the simulation is, but it will not give a more accurate simulation result for the actual system's response. It is a different thing when the model is used for prediction: Predicting future outputs from inputs and previously measured outputs, means that also future disturbance contributions have to be predicted. A known, or estimated, correlation structure (which really is the disturbance model) for the disturbances, will allow predictions of future disturbances, based on the previously measured values.

The need and use of the noise model can be summarized as follows:

- It is, in most cases, required to obtain a better estimate for the dynamics, *G*.
- It indicates how reliable noise-free simulations are.
- It is required for reliable predictions and stochastic control design.

Terms to Characterize the Model Properties

The properties of an input-output relationship like the ARX model follow from the numerical values of the coefficients, and the number of delays used. This is however a fairly implicit way of talking about the model properties. Instead a number of different terms are used in practice:

Impulse Response

The impulse response of a dynamical model is the output signal that results when the input is an impulse, i.e., *u(t)* is zero for all values of *t* except *t=0*, where *u(0)*=1. It can be computed as in the equation following (ARX), by letting *t* be equal to 0, 1, 2, ... and taking *y(-T)=y(-2T)*=0 and *u(0)*=1.

Step Response

The step response is the output signal that results from a step input, i.e., *u(t)* is zero for negative values of *t* and equal to one for positive values of *t*. The impulse and step responses together are called the model's **transient response**.

Frequency Response

The frequency response of a linear dynamic model describes how the model reacts to sinusoidal inputs. If we let the input *u(t)* be a sinusoid of a certain frequency, then the output *y(t)* will also be a sinusoid of this frequency. The amplitude and the phase (relative to the input) will however be different. This frequency response is most often depicted by two plots; one that shows the amplitude change as a function of the sinusoid's frequency and one that shows the phase shift as function of frequency. This is known as a Bode plot.

Zeros and Poles

The zeros and the poles are equivalent ways of describing the coefficients of a linear difference equation like the ARX model. The poles relate to the "output-side" and the zeros relate to the "input-side" of this equation. The number of poles (zeros) is equal to the number of sampling intervals between the most and least delayed output (input). In the ARX example in the beginning of this section, there are consequently two poles and one zero.

The Basic Steps of System Identification

The System Identification problem is to estimate a model of a system based on observed input-output data. Several ways to describe a system and to estimate such descriptions exist. This section gives a brief account of the most important approaches.

The procedure to determine a model of a dynamical system from observed input-output data involves three basic ingredients:

- The input-output data
- A set of candidate models (the model structure)
- A criterion to select a particular model in the set, based on the information in the data (the identification method)

The identification process amounts to repeatedly selecting a model structure, computing the best model in the structure, and evaluating this model's properties to see if they are satisfactory. The cycle can be itemized as follows:

1. Design an experiment and collect input-output data from the process to be identified.
2. Examine the data. Polish it so as to remove trends and outliers, and select useful portions of the original data. Possibly apply filtering to enhance important frequency ranges.
3. Select and define a model structure (a set of candidate system descriptions) within which a model is to be found.
4. Compute the best model in the model structure according to the input-output data and a given criterion of fit.
5. Examine the obtained model's properties
6. If the model is good enough, then stop; otherwise go back to Step 3 to try another model set. Possibly also try other estimation methods (Step 4) or work further on the input-output data (Steps 1 and 2).

The System Identification Toolbox offers several functions for each of these steps.

For Step 2 there are routines to plot data, filter data, and remove trends in data, as well as to resample and reconstruct missing data.

For Step 3 the System Identification Toolbox offers a variety of nonparametric models, as well as all the most common black-box input-output and state-space structures, and also general tailor-made linear state-space models in discrete and continuous time.

For Step 4 general prediction error (maximum likelihood) methods, as well as instrumental variable methods and sub-space methods are offered for parametric models, while basic correlation and spectral analysis methods are used for nonparametric model structures.

To examine models in Step 5, many functions allow the computation and presentation of frequency functions and poles and zeros, as well as simulation and prediction using the model. Functions are also included for transformations between continuous-time and discrete-time model descriptions and to formats that are used in other MATLAB toolboxes, like the Control System Toolbox and the Signal Processing Toolbox.

A Startup Identification Procedure

There are no standard and secure routes to good models in System Identification. Given the number of possibilities, it is easy to get confused about what to do, what model structures to test, and so on. This section describes one route that often works well, but there are no guarantees. The steps refer to functions within the GUI, but you can also go through them in command mode. For the basic commands, see Chapter 4, "Command Reference."

Step 1: Looking at the Data

Plot the data. Look at them carefully. Try to see the dynamics with your own eyes. Can you see the effects in the outputs of the changes in the input? Can you see nonlinear effects, like different responses at different levels, or different responses to a step up and a step down? Are there portions of the data that appear to be "messy" or carry no information. Use this insight to select portions of the data for estimation and validation purposes.

Do physical levels play a role in your model? If not, detrend the data by removing their mean values. The models will then describe how changes in the input give changes in output, but not explain the actual levels of the signals. This is the normal situation.

The default situation, with good data, is that you detrend by removing means, and then select the first half or so of the data record for estimation purposes, and use the remaining data for validation. This is what happens when you apply **Quickstart** under the pop-up menu **Preprocess** in the main **ident** window.

Step 2: Getting a Feel for the Difficulties

Apply **Quickstart** under pop-up menu **Estimate** in the main **ident** window. This will compute and display the spectral analysis estimate and the correlation analysis estimate, as well as a fourth order ARX model with a delay estimated from the correlation analysis and a default order state-space model computed by `n4sid`. This gives three plots. Look at the agreement between the:

- Spectral Analysis estimate and the ARX and state-space models' frequency functions
- Correlation Analysis estimate and the ARX and state-space models' transient responses

- Measured Validation Data output and the ARX and state-space models' simulated outputs

If these agreements are reasonable, the problem is not so difficult, and a relatively simple linear model will do a good job. Some fine tuning of model orders, and noise models have to be made and you can proceed to Step 4. Otherwise go to Step 3.

Step 3: Examining the Difficulties

There may be several reasons why the comparisons in Step 2 did not look good. This section discusses the most common ones, and how they can be handled.

Model Unstable

The ARX or state-space model may turn out to be unstable, but could still be useful for control purposes. Change to a 5- or 10-step ahead prediction instead of simulation in the **Model Output View**.

Feedback in Data

If there is feedback from the output to the input, due to some regulator, then the spectral and correlations analysis estimates are not reliable. Discrepancies between these estimates and the ARX and state-space models can therefore be disregarded in this case. In the **Model Residuals View** of the parametric models, feedback in data can also be visible as correlation between residuals and input for negative lags.

Disturbance Model

If the state-space model is clearly better than the ARX model at reproducing the measured output, this is an indication that the disturbances have a substantial influence, and it will be necessary to model them carefully.

Model Order

If a fourth order model does not give a good **Model Output** plot, try eighth order. If the fit clearly improves, it follows that higher order models will be required, but that linear models could be sufficient.

Additional Inputs

If the **Model Output** fit has not significantly improved by the tests so far, think over the physics of the application. Are there more signals that have been, or

could be, measured that might influence the output? If so, include these among the inputs and try again a fourth order ARX model from all the inputs. (Note that the inputs need not at all be control signals, anything measurable, including disturbances, should be treated as inputs).

Nonlinear Effects

If the fit between measured and model output is still bad, consider the physics of the application. Are there nonlinear effects in the system? In that case, form the nonlinearities from the measured data and add those transformed measurements as extra inputs. This could be as simple as forming the product of voltage and current measurements, if you realize that it is the electrical power that is the driving stimulus in, say, a heating process, and temperature is the output. This is of course application dependent. It does not take very much work, however, to form a number of additional inputs by reasonable nonlinear transformations of the measured ones, and just test if inclusion of them improves the fit.

Still Problems?

If none of these tests leads to a model that is able to reproduce the Validation Data reasonably well, the conclusion might be that a sufficiently good model cannot be produced from the data. There may be many reasons for this. It may be that the system has some quite complicated nonlinearities, which cannot be realized on physical grounds. In such cases, nonlinear, black-box models could be a solution. Among the most used models of this character are the Artificial Neural Networks (ANN).

Another important reason is that the data simply do not contain sufficient information, e.g., due to bad signal to noise ratios, large and nonstationary disturbances, varying system properties, etc.

Otherwise, use the insights of which inputs to use and which model orders to expect and proceed to Step 4.

Step 4: Fine Tuning Orders and Disturbance Structures

For real data there is no such thing as a “correct model structure.” However, different structures can give quite different model quality. The only way to find this out is to try out a number of different structures and compare the

properties of the obtained models. There are a few things to look for in these comparisons.

Fit Between Simulated and Measured Output

Keep the **Model Output View** open and look at the fit between the model's simulated output and the measured one for the Validation Data. Formally, you could pick that model, for which this number is the highest. In practice, it is better to be more pragmatic, and also take into account the model complexity, and whether the important features of the output response are captured.

Residual Analysis Test

You should require of a good model that the cross correlation function between residuals and input does not go significantly outside the confidence region. Otherwise there is something in the residuals that originate from the input, and has not been properly taken care of by the model. A clear peak at lag *k* shows that the effect from input *u(t-k)* on *y(t)* is not correctly described. A rule of thumb is that a slowly varying cross correlation function outside the confidence region is an indication of too few poles, while sharper peaks indicate too few zeros or wrong delays.

Pole Zero Cancellations

If the pole-zero plot (including confidence intervals) indicates pole-zero cancellations in the dynamics, this suggests that lower order models can be used. In particular, if it turns out that the orders of ARX models have to be increased to get a good fit, but that pole-zero cancellations are indicated, then the extra poles are just introduced to describe the noise. Then try ARMAX, OE, or BJ model structures with an A or F polynomial of an order equal to that of the number of noncanceled poles.

What Model Structures Should be Tested?

Well, you can spend any amount of time to check out a very large number of structures. It often takes just a few seconds to compute and evaluate a model in a certain structure, so that you should have a generous attitude to the testing. However, experience shows that when the basic properties of the system's behavior have been picked up, it is not much use to fine tune orders in absurdum just to press the fit by fractions of percents.

Many ARX models: There is a very cheap way of testing many ARX structures simultaneously. Enter in the **Orders** text field many combinations of orders,

using the colon (":") notation. You can also press the **Order Selection** button. When you select **Estimate**, models for all combinations (easily several hundreds) are computed and their (prediction error) fit to Validation Data is shown in a special plot. By clicking in this plot the best models with any chosen number of parameters will be inserted into the Model Board, and evaluated as desired.

Many State-space models: A similar feature is also available for black-box state-space models, estimated using `n4sid`. When a good order has been found, try the PEM estimation method, which often improves on the accuracy.

ARMAX, OE, and BJ models: Once you have a feel for suitable delays and dynamics orders, if is often useful to try out ARMAX, OE, and/or BJ with these orders and test some different orders for the disturbance transfer functions (C and D). Especially for poorly damped systems, the OE structure is suitable.

There is a quite extensive literature on order and structure selection, and anyone who would like to know more should consult the references.

Multivariable Systems

Systems with many input signals and/or many output signals are called *multivariable*. Such systems are often more challenging to model. In particular systems with several outputs could be difficult. A basic reason for the difficulties is that the couplings between several inputs and outputs lead to more complex models. The structures involved are richer and more parameters will be required to obtain a good fit.

Available Models

The System Identification Toolbox as well as the GUI handle general, linear multivariable models. All earlier mentioned models are supported in the single output, multiple input case. For multiple outputs, ARX models and state-space models are covered. Multi-output ARMAX and OE models are covered via state-space representations: ARMAX corresponds to estimating the K-matrix, while OE corresponds to fixing K to zero. (These are pop-up options in the GUI model order editor.)

Generally speaking, it is preferable to work with state-space models in the multivariable case, since the model structure complexity is easier to deal with. It is essentially just a matter of choosing the model order.

Working with Subsets of the Input-Output Channels

In the process of identifying good models of a system, it is often useful to select subsets of the input and output channels. Partial models of the system's behavior will then be constructed. It might not, for example, be clear if all measured inputs have a significant influence on the outputs. That is most easily tested by removing an input channel from the data, building a model for how the output(s) depends on the remaining input channels, and checking if there is a significant deterioration in the model output's fit to the measured one. See also the discussion under Step 3 above.

Generally speaking, the fit gets better when more inputs are included and often gets worse when more outputs are included. To understand the latter fact, you should realize that a model that has to explain the behavior of several outputs has a tougher job than one that just must account for a single output. If you have difficulties obtaining good models for a multi-output system, it might be wise to model one output at a time, to find out which are the difficult ones to handle.

Models that are just to be used for simulations could very well be built up from single-output models, for one output at a time. However, models for prediction and control will be able to produce better results if constructed for all outputs simultaneously. This follows from the fact that knowing the set of all previous output channels gives a better basis for prediction, than just knowing the past outputs in one channel. Also, for systems, where the different outputs reflect similar dynamics, using several outputs simultaneously will help estimating the dynamics.

Some Practical Advice

Both the GUI and command line operation will do useful bookkeeping for you, handling different channels. You could follow the steps of this agenda:

- Import data and create a data set with all input and output channels of interest. Do the necessary preprocessing of this set in terms of detrending, etc., and then select a Validation Data set with all channels.
- Then select a Working Data set with all channels, and estimate state-space models of different orders using `n4sid` for these data. Examine the resulting model primarily using the **Model Output** view.
- If it is difficult to get a good fit in all output channels or you would like to investigate how important the different input channels are, construct new

data sets using subsets of the original input/output channels. Use the pop-up menu **Preprocess > Select Channels** for this. Don't change the Validation Data. The GUI will keep track of the input and output channels. It will "do the right thing" when evaluating the channel-restricted models using the Validation Data. It might also be appropriate to see if improvements in the fit are obtained for various model types, built for one output at a time.

- If you decide for a multi-output model, it is often easiest to use state-space models. Use `n4sid` as a primary tool and try out `pem` when a good order has been found.

Reading More About System Identification

There is substantial literature on System Identification. The following textbook deals with identification methods from a similar perspective as this toolbox, and also describes methods for physical modeling:

- Ljung L. and T. Glad. *Modeling of Dynamic Systems*, Prentice Hall, Englewood Cliffs, N.J. 1994.

For more details about the algorithms and theories of identification:

- Ljung L. *System Identification - Theory for the User*, Prentice Hall, Upper Saddle River, N.J. 2nd edition, 1999.
- Söderström T. and P. Stoica. *System Identification*, Prentice Hall International, London. 1989.

For more about system and signals:

- Oppenheim J. and A.S. Willsky. *Signals and Systems*, Prentice Hall, Englewood Cliffs, N.J. 1985.

The following textbook deals with the underlying numerical techniques for parameter estimation:

- Dennis, J.E. Jr. and R.B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice Hall, Englewood Cliffs, N.J. 1983.

2

The Graphical User Interface

The Big Picture

The System Identification Toolbox provides a graphical user interface (GUI) . The GUI covers most of the toolbox's functions and gives easy access to all variables that are created during a session. It is started by typing

```
ident
```

in the MATLAB command window.

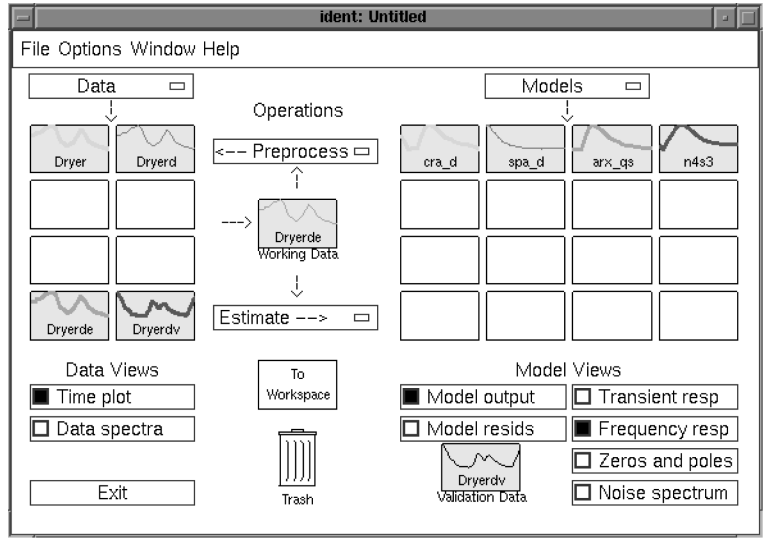


Figure 2-1: The Main ident Information Window

The Model and Data Boards

System Identification is about data and models and creating models from data. The main information and communication window **ident**, is therefore dominated by two tables:

- A table over available data sets, each represented by an icon
- A table over created models, each represented by an icon

These tables will be referred to as the *Model Board* and the *Data Board* in this chapter. You enter data sets into the Data Board by

- Opening earlier saved sessions.
- Importing them from the MATLAB workspace.
- Creating them by detrending, filtering, selecting subsets, etc., of another data set in the Data Board.

Imports are handled under the pop-up menu **Data** while creation of new data sets is handled under the pop-up menu **Preprocess**. “Handling Data” on page 2-7 deals with this in more detail.

The models are entered into the summary board by

- Opening earlier saved sessions.
- Importing them from the MATLAB workspace.
- Estimating them from data.

Imports are handled under the pop-up menu **Models**, while all the different estimation schemes are reached under the pop-up menu **Estimate**. More about this in “Estimating Models” on page 2-15.

The Data and Model Boards can be rearranged by dragging and dropping. More boards open automatically when necessary or when asked for (under menu **Options**).

The Working Data

All data sets and models are created from the Working Data set. This is the data that is given in the center of the **ident** window. To change the Working Data set drag and drop any data set from the Data Board on the Working Data icon.

The Views

Below the Data and Model Boards are buttons for different views. These control what aspects of the data sets and models you would like to examine, and are described in more detail in “Handling Data” on page 2-7 and in “Examining Models” on page 2-28. To select a data set or a model, so that its properties are displayed, click on its icon. A selected object is marked by a thicker line in the icon. To deselect, click again. An arbitrary number of data/

model objects can be examined simultaneously. To have more information about an object, double-click (or right-click or Ctrl-click) on its icon.

The Validation Data

The two model views **Model Output** and **Model Residuals** illustrate model properties when applied to the Validation Data set. This is the set marked in the box below these two views. To change the Validation Data, drag and drop any data set from the Data Board on the Validation Data icon.

It is good and common practice in identification to evaluate an estimated model's properties using a "fresh" data set, that is, one that was not used for the estimation. It is thus good advice to let the Validation Data be different from the Working Data, but they should of course be compatible with these.

The Work Flow

You start by importing data (under pop-up menu **Data**); you examine the data set using the **Data Views.** You probably remove the means from the data and select subsets of data for estimation and validation purposes using the items in the pop-up menu **Preprocess**. You then continue to estimate models, using the possibilities under the pop-up menu **Estimate**, perhaps first doing a quickstart. You examine the obtained models with respect to your favorite aspects using the different **Model Views**. The basic idea is that any checked view shows the properties of all selected models at any time. This function is "live" so models and views can be checked in and out at will in an online fashion. You select/deselect a model by clicking on its icon.

Inspired by the information you gain from the plots, you continue to try out different model structures (model orders) until you find a model you are satisfied with.

Management Aspects

Diary: It is easy to forget what you have been doing. By double-clicking on a data/model icon, a complete diary will be given of how this object was created, along with other key information. At this point you can also add comments and change the name of the object and its color.

Layout: To have a good overview of the created models and data sets, it is good practice to try rearranging the icons by dragging and dropping. In this way models corresponding to a particular data set can be grouped together, etc. You

can also open new boards (**Options** menu **Extra model/data boards**) to further rearrange the icons. These can be dragged across the screen between different windows. The extra boards are also equipped with notepads for your comments.

Sessions: The Model and Data Boards with all models and data sets together with their diaries can be saved (under menu item **File**) at any point, and reloaded later. This is the counterpart of save/load workspace in the command-driven MATLAB. The four most recent sessions are listed under **File** for immediate open.

Cleanliness: The boards will hold an arbitrary number of models and data sets (by creating clones of the board when necessary). It is however advisable to clear (delete) models and data sets that no longer are of interest. Do that by dragging the object to the **Trash Can**. (Double-clicking on the trash can will open it up, and its contents can be retrieved.) Empty the can if you run into memory problems.

Window Culture: Dialog and plot windows are best managed by the GUI's `close` function (submenu item under **File** menu, or select **Close**, or check/uncheck the corresponding View box). It is generally not suitable to iconify the windows – the GUI's handling and window management system is usually a better alternative.

Workspace Variables

The models and data sets created within the GUI are normally not available in the MATLAB workspace. Indeed, the workspace is not at all littered with variables during the sessions with the GUI. The variables can however be exported at any time to the workspace, by dragging and dropping the object icon on the **To Workspace** box. They will then carry the name in the workspace that marked the object icon at the time of export. You can work with the variables in the workspace, using any MATLAB commands, and then perhaps import modified versions back into the GUI. Note that models and data are exported as the toolbox's objects `idmodel`, `idfrd`, and `iddata`. For how to extract information and work with these objects, see Chapter 3, "Tutorial", and "Model Conversions" on page 4-5 of the "Command Reference "chapter.

The GUI's names of data sets and models are suggested by default procedures. Normally, you can enter any other name of your choice at the time of creation of the variable. Names can be changed (after double-clicking on the icon) at any

time. Unlike the workspace situation, two GUI objects can carry the same name (i.e., the same string in their icons).

Help Texts

The GUI contains some 100 help texts that are accessible in a nested fashion, when required. The main **ident** window contains general help topics under the **Help** menu. This is also the case for the various plot windows. In addition, every dialog box has a **Help** push button for current help and advice.

Handling Data

Data Representation

In the System Identification Toolbox, signals and observed data are represented as column vectors, e.g.,

$$u = \begin{bmatrix} u(1) \\ u(2) \\ \dots \\ \dots \\ u(N) \end{bmatrix}$$

The entry in row number k, i.e., $u(k)$, will then be the signal's value at sampling instant number k. It is generally assumed in the toolbox that data are sampled at equidistant sampling times, and the sampling interval T is supplied as a specific argument.

We generally denote the input to a system by the letter u and the output by y. If the system has several input channels, the input data is represented by a matrix, where the columns are the input signals in the different channels:

$$u = \begin{bmatrix} u1 & u2 & \dots & um \end{bmatrix}$$

The same holds for systems with several output channels.

The observed input-output data record is represented in the System Identification Toolbox by the `iddata` object, that is created from the input and output signals by

```
Data = iddata(y,u,Ts)
```

where `Ts` is the sampling time

The `iddata` object can also be created from the input and output signals when the data are inserted into the GUI.

Getting Input-Output Data into the GUI

The information about a data set that should be supplied to the GUI is as follows:

1. The input and output signals
2. The name you give to the data set
3. The sampling interval

In addition to this mandatory information, you may add further properties that will help in the bookkeeping:

4. The starting time for the sampling
5. Input and output channel names
6. Input and output channel units
7. Periodicity and intersample behavior of the input
8. Data notes: These are notes for your own information and bookkeeping that will follow the data and all models created from them.

As you select the pop-up menu **Data** and choose the item **Import**, a dialog box will open, where you can enter the information items 1 - 8, just listed. This box has five fields for you to fill in.

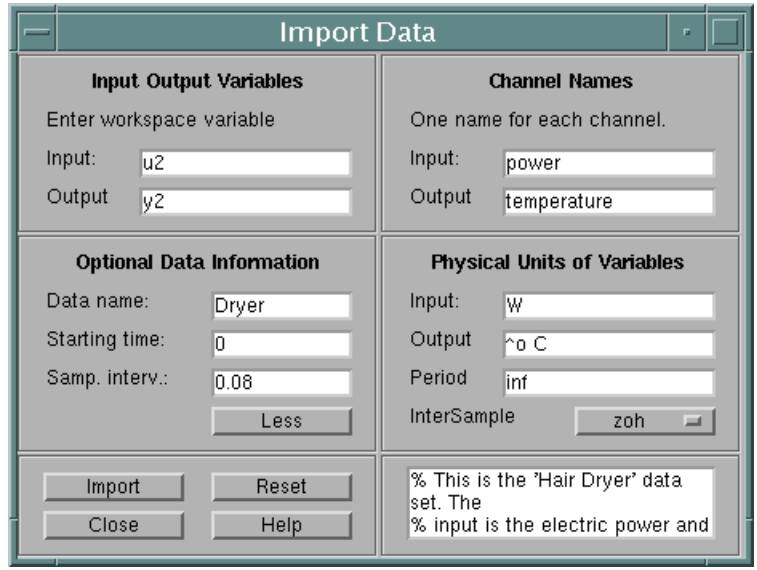


Figure 2-1: The Dialog for Importing Data into the GUI

By pressing **More,** six more fields will become visible.

Input and **Output**: Enter the variable names of the input and output respectively. These should be variables in your MATLAB workspace, so you may have to load some disk files first.

Actually, you can enter any MATLAB expressions in these fields, and they will be evaluated to compute the input and the output before inserting the data into the GUI.

Data name: Enter the name of the data set to be used by the GUI. This name can be changed later on.

Starting time and Sampling interval: Fill these out for correct time and frequency scales in the plots.

On the extra page you optionally can fill out

Channel names: Enter strings for the different input and output channels names. Separate the strings by comma. The number of names must be equal to the number of channels. If these entries are not filled out, default names, y1,y2, ..., u1, u2 ..., will be used.

Channel units: Enter, in analogous format, the units in which the measurements are made. These will follow to all models built from data, but are used only for plot information.

Period: If the input is periodic, enter here the period length. 'Inf' means a non-periodic input, which is default.

Intersample: Choose the intersample behavior of the input as one of ZOH (zero-order hold, i.e., the input signal piecewise constant between the samples) or FOH (first-order hold, i.e., the input signal is piecewise linear between the samples) or BL (Band-limited, i.e., the continuous time input signal has no power above the Nyquist frequency). ZOH is default.

The box at the bottom is for **Notes,** where you can enter any text you want to accompany the data for bookkeeping purposes.

Finally, select **Import** to insert the data into the GUI. When no more data sets are to be inserted, select **Close** to close the dialog box. **Reset** will empty all the fields of the box.

The procedure just described will create an `iddata` object, with all its properties. If you already have an `iddata` object available in the workspace, you can import that directly by selecting the data format **Iddata Object** in the pop-up menu at the top of the **Import Data** dialog.

Taking a Look at the Data

The first thing to do after having inserted the data set into the Data Board is to examine it. By checking the **Data View** item **Time plot**, a plot of the input and output signals will be shown for the data sets that are selected. You select/deselect the data sets by clicking on them. For multivariable data, the different combinations of input and output signals are chosen under menu item **Channel** in the plot window. Using the `zoom` function (drawing rectangles with the left mouse button down) different portions of the data can be examined in more detail.

To examine the frequency contents of the data, check the **Data View** item **Data spectra**. The function is analogous to **Time plot**, but the signals' spectra are shown instead. By default the periodograms of the data are shown, i.e., the absolute square of the Fourier transforms of the data. The plot can be changed to any chosen frequency range and a number of different ways of estimating spectra, by the **Options** menu item in the spectra window.

The purpose of examining the data in these ways is to find out if there are portions of the data that are not suitable for identification, if the information contents of the data is suitable in the interesting frequency regions, and if the data have to be preprocessed in some way, before using them for estimation.

Preprocessing Data

Detrending

Detrending the data involves removing the mean values or linear trends from the signals (the means and the linear trends are then computed and removed from each signal individually). This function is accessed under the pop-up menu **Preprocess**, by selecting item **Remove Means** or **Remove Trends**. More advanced detrending, such as removing piecewise linear trends or seasonal variations cannot be accessed within the GUI. It is generally recommended that you remove at least the mean values of the data before the estimation phase. There are however situations when it is not advisable to remove the sample means. It could for example be that the physical levels are built into the underlying model, or that integrations in the system must be handled with the right level of the input being integrated.

Selecting Data Ranges

It is often the case that the whole data record is not suitable for identification, due to various undesired features (missing or "bad" data, outbursts of disturbances, level changes etc.), so that only portions of the data can be used. In any case, it is advisable to select one portion of the measured data for estimation purposes and another portion for validation purposes. The pop-up menu item **Preprocess > Select Range...** opens a dialog box, which facilitates the selection of different data portions, by typing in the ranges, or marking them by drawing rectangles with the mouse button down.

For multivariable data it is often advantageous to start by working with just some of the input and output signals. The menu item **Preprocess > Select**

Channels... allows you to select subsets of the inputs and outputs. This is done in such a way that the input/output numbering and names remains consistent when you evaluate data and model properties, for models covering different subsets of the data.

Prefiltering

By filtering the input and output signals through a linear filter (the same filter for all signals) you can, e.g., remove drift and high frequency disturbances in the data, that should not affect the model estimation. This is done by selecting the pop-up menu item **Preprocess > Filter...** in the main window. The dialog is quite analogous to that of selecting data ranges in the time domain. You mark with a rectangle in the spectral plots the intended passband or stop band of the filter, you select a button to check if the filtering has the desired effect, and then you insert the filtered data into the GUI's Data Board.

Prefiltering is a good way of removing high frequency noise in the data, and also a good alternative to detrending (by cutting out low frequencies from the pass band). Depending on the intended model use, you can also make sure that the model concentrates on the important frequency ranges. For a model that will be used for control design, for example, the frequency band around the intended closed-loop bandwidth is of special importance.

If you intend to use the data to build models both of the system dynamics and the disturbance properties, it is recommended to do the filtering at the estimation phase. That is achieved by selection the pop-up menu item **Estimate > Parametric Models,** and then select the estimation **Focus** to be **Filter**. This opens the same filter dialog as above. The prefiltering will however apply only for estimating the dynamics from input to output. The disturbance model is determined from the original data.

Resampling

If the data turn out to be sampled too fast, they can be decimated, i.e., every k-th value is picked, after proper prefiltering (antialias filtering). This is obtained from menu item **Preprocess > Resample**.

You can also resample at a faster sampling rate by interpolation, using the same command, and giving a resampling factor less than one.

Quickstart

The pop-up menu item **Preprocess > Quickstart** performs the following sequence of actions: It opens the **Time plot Data view**, removes the means from the signals, and it splits these detrended data into two halves. The first one is made Working Data and the second one becomes Validation Data. All the three created data sets are inserted into the Data Board.

Multi-Experiment Data

The toolbox allows the handling of data sets that contain several different experiments. Both estimation and validation can be applied to such data sets. This is quite useful to deal with experiments that have been conducted at different occasions but describe the same system. It is also useful to be able to keep together pieces of data that have been obtained by cutting out "informative pieces" from a long data set. Multi-experiment data can be imported and used in the GUI as any `iddata` object. Selecting specific part of a multi-experiment data set is done from the pop-up menu item Preprocess > Select Experiment. To merge several data sets in the Data board (obtained, e.g., by cutting out nice portions from other data sets) use the pop-up menu item Preprocess > Merge Experiment.

Checklist for Data Handling

- Insert data into the GUI's Data Board.
- Plot the data and examine it carefully.
- Typically detrend the data by removing mean values.
- Select portions of the data for Estimation and for Validation. Drag and drop these data sets to the corresponding boxes in the GUI.

Simulating Data

The GUI is intended primarily for working with real data sets, and does not itself provide functions for simulating synthetic data. That has to be done in command mode, and you can use your favorite procedure in Simulink, the Signal Processing Toolbox, or any other toolbox for simulation and then insert the simulated data into the GUI as described above.

The System Identification Toolbox also has several commands for simulation. For example, should check `idinput` and `sim` in the "Command Reference" chapter for details. The following example shows how the ARMAX model

$$y(t) - 1.5y(t-1) + 0.7y(t-2) =$$
$$u(t-1) + 0.5u(t-2) + e(t) - e(t-1) + 0.2e(t-1)$$

is simulated with a random binary input u:

```
% Create an ARMAX model
model1 = idpoly([1 -1.5 0.7],[0 1 0.5],[1 -1 0.2]);
u = idinput(400,'rbs',[0 0.3]);
e = randn(400,1);
y = sim(model1,[u e]);
```

The input, u, and the output, y, can now be imported into the GUI as data, and the various estimation routines can be applied to them. By also importing the simulation model, `model1`, into the GUI, its properties can be compared to those of the different estimated models.

To simulate a continuous-time state-space model

$$\dot{x} = Ax + Bu + Ke$$
$$y = Cx + e$$

with the same input, and a sampling interval of 0.1 seconds, do the following in the System Identification Toolbox:

```
A = [-1 1;-0.5 0]; B = [1; 0.5]; C = [1 0]; D = 0; K = [0.5;0.5];
Model2 = idss(A,B,C,D,K,'Ts', 0) % Ts = 0 means continuous time
Data = iddata([],[u e]);
Data.Ts = 0.1
y=sim(Model2,Data);
```

Estimating Models

The Basics

Estimating models from data is the central activity in the System Identification Toolbox. It is also the one that offers the most variety of possibilities and thus is the most demanding one for the user.

All estimation routines are accessed from the pop-up menu **Estimate** in the **ident** window. The models are always estimated using the data set that is currently in the **Working Data** box.

One can distinguish between two different types of estimation methods:

- Direct estimation of the Impulse or the Frequency Response of the system. These methods are often also called nonparametric estimation methods, and do not impose any structure assumptions about the system, other than that it is linear.
- Parametric methods. A specific model structure is assumed, and the parameters in this structure are estimated using data. This opens up a large variety of possibilities, corresponding to different ways of describing the system. Dominating ways are state-space and several variants of difference equation descriptions.

Direct Estimation of the Impulse Response

A linear system can be described by the impulse response g_k, with the property that

$$y(t) = \sum_{k=1}^{\infty} g_k u(t-k)$$

The name derives from the fact that if the input *u(t)* is an impulse, i.e., *u(t)*=1 when *t*=0 and 0 when *t*>0, then the output *y(t)* will be $y(t) = g_t$. For a multivariable system, the impulse response g_k will be a *ny*-by-*nu* matrix, where *ny* is the number of outputs and *nu* is the number of inputs. Its *i-j* element thus describes the behavior of the i-th output after an impulse in the *j*-th input.

By choosing menu item **Estimate > Correlation Model** impulse response coefficients are estimated directly from the input/output data using so called

correlation analysis. The actual method is described under the command `impulse` in the "Command Reference" chapter. For a quick action, you can also just type the letter `c` in the **ident** window. This is the *hotkey* for correlation analysis.

The resulting impulse response estimate is placed in the Model Board, under the default name `imp`. (The name can be changed by double-clicking on the model icon and then typing in the desired name in the dialog box that opens.)

The best way to examine the result is to select the **Model View Transient Response**. This gives a graph of the estimated response. This view offers a choice between displaying the Impulse or the Step response. For a multivariable system, the different channels, i.e., the responses from a certain input to a certain output, are selected under menu item **Channel**.

The number of lags for which the impulse response is estimated, i.e., the length of the estimated response, is determined as one of the options in the Transient Response view.

Direct Estimation of the Frequency Response

The frequency response of a linear system is the Fourier transform of its impulse response. This description of the system gives considerable engineering insight into its properties. The relation between input and output is often written

$$y(t) = G(z)u(t) + v(t)$$

where G is the transfer function and v is the additive disturbance. The function

$$G(e^{i\omega T})$$

as a function of (angular) frequency ω is then the frequency response or frequency function. T is the sampling interval. If you need more details on the different interpretations of the frequency response, See "The System Identification Problem" on page 3-9 in the Tutorial chapter or any textbook on linear systems.

The system's frequency response is directly estimated using *spectral analysis* by the menu item **Estimate > Spectral Model**, and then selecting the **Estimate** button in the dialog box that opens. The result is placed on the Model Board under the default name spad. The best way to examine it is to plot it using the **Model View Frequency Response**. This view offers a number of

different options on how to graph the curves. The frequencies for which to estimate the response can also be selected as an option under the **Options** menu in this **View** window.

The Spectral Analysis command also estimates the spectrum of the additive disturbance *v(t)* in the system description. This estimated disturbance spectrum is examined under the **Model View** item **Noise Spectrum**.

The Spectral Analysis estimate is stored as an `idfrd` object. If you need to further work with the estimates, you can export the model to the MATLAB workspace and retrieve the responses directly from this object or by Nyquist and Bode. See `idfrd`, `bode`, and `nyquist` in the "Command Reference" chapter for more information. (A model is exported by dragging and dropping it over the **To Workspace** icon.)

Two options that affect the spectral analysis estimate can be set in the dialog box. The most important choice is a number, M, (the size of the lag window) that affects the frequency resolution of the estimates. Essentially, the frequency resolution is about $2\pi/M$ radians/(sampling interval). The choice of M is a trade-off between frequency resolution and variance (fluctuations). A large value of M gives good resolution but fluctuating and less reliable estimates. The default choice of M is good for systems that do not have very sharp resonances and may have to be adjusted for more resonant systems.

The options also offer a choice between the Blackman-Tukey windowing method `spa` (which is default) and a method based on smoothing direct Fourier transforms, `etfe`. `etfe` has an advantage for highly resonant systems, in that it is more efficient for large values of M. It however has the drawbacks that it requires linearly spaced frequency values, does not estimate the disturbance spectrum, and does not provide confidence intervals. The actual methods are described in more detail in the "Command Reference" chapter under `spa` and `etfe`. To obtain the spectral analysis model for the current settings of the options, you can just type the hotkey `s` in the **ident** window.

Estimation of Parametric Models

The System Identification Toolbox supports a wide range of model structures for linear systems. They are all accessed by the menu item **Estimate > Parametric Models...** in the **ident** window. This opens up a dialog box

Parametric Models, which contains the basic dialog for all parametric estimation as shown below.

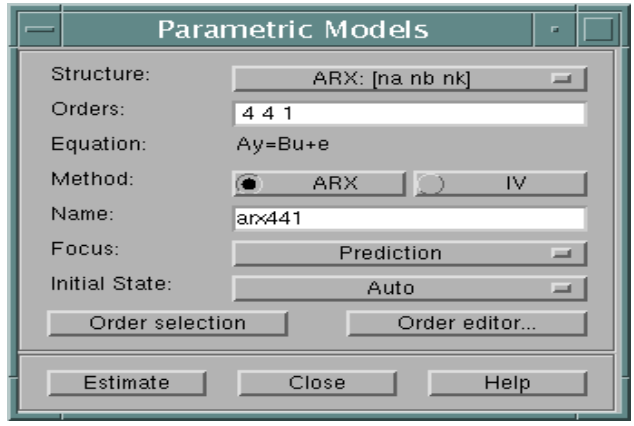


Figure 2-2: The Dialog Box for Estimating Parametric Models

The basic function of this box is as follows:

As you select **Estimate**, a model is estimated from the Working Data. The structure of this model is defined by the pop-up menu **Structure** together with the edit box **Orders**. It is given a name, which is written in the edit box **Name**.

The GUI will always suggest a default model name in the **Name** box, but you can change it to any string before selecting the **Estimate** button. (If you intend to export the model later, avoid spaces in the name.)

The interpretation of the model structure information (typically integers) in the **Order** box, depends on the selected **Structure** in the pop-up menu. This covers, typically, six choices:

- ARX models
- ARMAX model
- Output-Error (OE) models
- Box-Jenkins (BJ) models
- State-space models
- Model structure defined by Initial Model (User defined structures)

These are dealt with one by one shortly.

You can fill out the **Order** box yourself at any time, but for assistance you can select **Order Editor**. This will open up another dialog box, depending on the chosen **Structure**, in which the desired model order and structure information can be entered in a simpler fashion.

You can also enter a name of a MATLAB workspace variable in the order edit box. This variable should then have a value that is consistent with the necessary orders for the chosen structure.

Note For the state-space structure and the ARX structure, several orders and combination of orders can be entered. Then all corresponding models will be compared and displayed in a special dialog window for you to select suitable ones. This could be a useful tool to select good model orders. This option is described in more detail later in this section. When it is available, a button **Order selection** is visible.

Estimation Method

A common and general method of estimating the parameters is the *prediction error approach*, where simply the parameters of the model are chosen so that the difference between the model's (predicted) output and the measured output is minimized. This method is available for all model structures. Except for the ARX case, the estimation involves an iterative, numerical search for the best fit.

To obtain information from and interact with this search, select **Iteration control...** This is a button which is visible when an iterative estimation process has been selected. This also gives access to a number of options that govern the search process. (See "Algorithm Properties" on page 4-10 in the "Command Reference" chapter.)

For some model structures (the ARX model, and black-box state-space models) methods based on correlation are also available: Instrumental Variable (IV) and Sub-space (N4SID) methods. The choice between methods is made in the **Parametric Models** dialog box.

The dialog box also has two pop-up menus that offer further options**: Focus** allows you to choose between a frequency weighting that concentrates on the model's prediction or simulation performance. Another alternative is prefiltering, which was described on page 2-12. Moreover, the pop-up menu

InitialState gives options to estimate the initial state or to fix it to zero. The value **Auto** makes an automatic choice among these options.

Resulting Models

The estimated model is inserted into the GUI's Model Board. You can then examine its various properties and compare them with other models' properties using the **Model View** plots. More about that in "Examining Models" on page 2-28.

To take a look at the model itself, double-click on the model's icon (or right mouse button click or Ctrl-click). The **Data/Model Info** window that then opens gives you information about how the model was estimated. You can then also select the **Present** button, which will list the model, and its parameters with estimated standard deviations in the MATLAB command window.

If you need to work further with the model, you can export it by dragging and dropping it over the **To Workspace** icon, and then apply any MATLAB and toolbox commands to it. (See, in particular, the commands `ssdata`, `tfdata`, `d2c`, and `get` in the "Command Reference" chapter.)

How to Know Which Structure and Method to Use

There is no simple way to find out "the best model structure"; in fact, for real data, there is no such thing as a *best* structure. Some routes to find good and acceptable model are described in "A Startup Identification Procedure" on page 1-14. It is best to be generous at this point. It often takes just a few seconds to estimate a model, and by the different validation tools described in the next section, you can quickly find out if the new model is any better than the ones you had before. There is often a significant amount of work behind the data collection, and spending a few extra minutes trying out several different structures is usually worth while.

ARX Models

The Structure

The most used model structure is the simple linear difference equation

$$y(t) + a_1 y(t-1) + \ldots + a_{na} y(t-na) =$$
$$b_1 u(t-nk) + \ldots + b_{nb} u(t-nk-nb+1)$$

which relates the current output *y(t)* to a finite number of past outputs *y(t-k)* and inputs *u(t-k)*.

The structure is thus entirely defined by the three integers na, nb, and nk. na is equal to the number of poles and nb–1 is the number of zeros, while nk is the pure time-delay (the dead-time) in the system. For a system under sampled-data control, typically nk is equal to 1 if there is no dead-time.

For multi-input systems nb and nk are row vectors, where the i-th element gives the order/delay associated with the i-th input.

Entering the Order Parameters

The orders na, nb, and nk can either be directly entered into the edit box **Orders** in the **Parametric Models** window, or selected using the pop-up menus in the **Order Editor**.

Estimating Many Models Simultaneously

By entering any or all of the structure parameters as vectors, using MATLAB's colon notation, like na=1:10, etc., you define many different structures that correspond to all combinations of orders. When selecting **Estimate**, models corresponding to all of these structures are computed. A special plot window will then open that shows the fit of these models to Validation Data. By clicking in this plot, you can then enter any models of your choice into the Model Board.

Multi-input models: For multi-input models you can of course enter each of the input orders and delays as a vector. The number of models resulting from all combinations of orders and delays can however be very large. As an alternative, you may enter one vector (like nb=1:10) for all inputs and one vector for all delays. Then only such models are computed that have the same orders and delays from all inputs.

Estimation Methods

There are two methods to estimate the coefficients a and b in the ARX model structure:

Least Squares: Minimizes the sum of squares of the right-hand side minus the left-hand side of the expression above, with respect to *a* and *b*. This is obtained by selecting ARX as the **Method**.

Instrumental Variables: Determines *a* and *b* so that the error between the right- and left- hand sides becomes uncorrelated with certain linear combinations of the inputs. This is obtained by selecting `IV` in the **Method** box.

The methods are described in more detail in the "Command Reference" chapter under `arx` and `iv4`.

Multi-Output Models

For a multi-output ARX structure with ny outputs and nu inputs, the difference equation above is still valid. The only change is that the coefficients *a* are ny-by-ny matrices and the coefficients *b* are ny-by-nu matrices.

The orders `[NA NB NK]` define the model structure as follows:

`NA`: an *ny*-by-*ny* matrix whose *i*-*j* entry is the order of the polynomial (in the delay operator) that relates the *j*-th output to the *i*-th output

`NB`: an *ny*-by-*nu* matrix whose *i*-*j* entry is the order of the polynomial that relates the *j*-th input to the *i*-th output

`NK`: an *ny*-by-*nu* matrix whose *i*-*j* entry is the delay from the *j*-th input to the *i*-th output

The **Order Editor** dialog box allows the choices

```
NA = na*ones(ny,ny)
NB = nb*ones(ny,nu)
NK = nk*ones(ny,nu)
```

where `na`, `nb`, and `nk` are chosen by the pop-up menus.

For tailor-made order choices, construct a matrix `[NA NB NK]` in the MATLAB command window and enter the name of this matrix in the **Order** edit box in the **Parametric Models** window.

Note that the possibility to estimate many models simultaneously is not available for multi-output ARX models.

See "Defining Model Structures" on page 3-35 for more information on multi-output ARX models.

ARMAX, Output-Error and Box-Jenkins Models

There are several elaborations of the basic ARX model, where different disturbance models are introduced. These include well known model types, such as ARMAX, Output-Error, and Box-Jenkins.

The General Structure

A general input-output linear model for a single-output system with input u and output y can be written:

$$A(q)y(t) = \sum_{i=1}^{nu} [B_i(q)/F_i(q)]u_i(t - nk_i) + [C(q)/D(q)]e(t)$$

Here u_i denotes input #i, and A, B_i, C, D, and F_i, are polynomials in the shift operator (z or q). (Don't get intimidated by this: It is just a compact way of writing difference equations; see below.)

The general structure is defined by giving the time-delays nk and the orders of these polynomials *(i.e.*, the number of poles and zeros of the dynamic model from u to y, as well as of the disturbance model from e to y).

The Special Cases

Most often the choices are confined to one of the following special cases:

ARX: $A(q)y(t) = B(q)u(t - nk) + e(t)$

ARMAX: $A(q)y(t) = B(q)u(t - nk) + C(q)e(t)$

OE: $y(t) = [B(q)/F(q)]u(t - nk) + e(t)$ (Output-Error)

BJ: $y(t) = [B(q)/F(q)]u(t - nk) + [C(q)/D(q)]e(t)$ (Box-Jenkins)

The "shift operator polynomials" are just compact ways of writing difference equations. For example the ARMAX model in longhand would be:

$$y(t) + a_1 y(t-1) + \ldots + a_{na} y(t - na) = b_1 u(t - nk) + \ldots +$$
$$b_{nb} u(t - nk - nb + 1) + e(t) + c_1 e(t-1) + \ldots + c_{nc} e(t - nc)$$

Note that *A(q)* corresponds to poles that are common between the dynamic model and the disturbance model (useful if disturbances enter the system "close to" the input). Likewise $F_i(q)$ determines the poles that are unique for

the dynamics from input # *i*, and *D(q)* the poles that are unique for the disturbances.

The reason for introducing all these model variants is to provide for flexibility in the disturbance description and to allow for common or different poles (dynamics) for the different inputs.

Entering the Model Structure

Use the **Structure** pop-up menu in the **Parametric Models** dialog to choose between the ARX, ARMAX, Output-Error, and Box-Jenkins structures. Note that if the Working Data set has several outputs, only the first choice is available. For time series (data with no input signal) only AR and ARMA are available among these choices. These are the time series counterparts of ARX and ARMAX.

The orders of the polynomials are selected by the pop-up menus in the **Order Editor** dialog window, or by directly entering them in the edit box **Orders** in the **Parametric Models** window. When the order editor is open, the default orders, entered as you change the model structure, are based on previously used orders.

Estimation Method

The coefficients of the polynomials are estimated using a prediction error/ Maximum Likelihood method, by minimizing the size of the error term “e” in the expression above. Several options govern the minimization procedure. These are accessed by activating **Iteration Control** in the **Parametric Models** window, and selecting **Options**.

The algorithms are further described in Chapter 4, “Command Reference” under `armax`, `Algorithm Properties`, `bj`, `oe`, and `pem`. See also “Parametric Model Estimation” on page 3-25 and “Defining Model Structures” on page 3-35.

Note These model structures are available only for the scalar output case. For multi-output models, the state-space structures offer the same flexibility. Also note that it is not possible to estimate many different structures simultaneously for the input-output models.

State-Space Models

The Model Structure

The basic state-space model in innovations form can be written

$$x(t+1) = A\,x(t) + B\,u(t) + K\,e(t)$$
$$y(t) = C\,x(t) + D\,u(t) + e(t)$$

The System Identification Toolbox supports two kinds of parametrizations of state-space models: black-box, free parametrizations, and parametrizations tailor-made to the application. The latter is discussed below under the heading "User Defined Model Structures" on page 2-26. First we will discuss the black-box case.

Entering Black-Box State-Space Model Structures

The most important structure index is the model order; i.e., the dimension of the state vector *x*.

Use the pop-up menu in the **Order Editor** to choose the model order, or enter it directly into the **Orders** edit box in the **Parametric Models** window. Using the other pop-up menus in the **Order Editor**, you can further affect the chosen model structure:

- Fixing *K* to zero gives an Output-Error method; i.e., the difference between the model's simulated output and the measured one is minimized. Formally, this corresponds to an assumption that the output disturbance is white noise.

The delays from the input can be chosen independently for each input. It will be a row vector nk, with *nu* entries. When the delay is larger than or equal to one, the *D*-matrix in the discrete time model is fixed to zero. For physical systems, without a pure time delay, that are driven by piece-wise constant inputs, nk = 1 is a natural assumption This is also the default. Note also that the delays can be directly entered into the **Orders** edit box.

Estimating Many Models Simultaneously

By entering a vector for the model order, using MATLAB's colon notation, (such as "1:10") all indicated orders will be computed using a preliminary method.

You can then enter models of different orders into the Model Board by clicking in a special graph that contains information about the models.

Estimation Methods

There are two basic methods for the estimation:

`PEM`: Is a standard prediction error/maximum likelihood method, based on iterative minimization of a criterion. The iterations are started up at parameter values that are computed from `n4sid`. The parametrization of the matrices *A*, *B*, *C*, *D*, and *K* is free. The search for minimum is controlled by a number of options. These are accessed from the **Option** button in the **Iteration Control** window.

`N4SID`: Is a subspace-based method that does not use iterative search. The quality of the resulting estimates may significantly depend some options called `N4Weight` and `N4Horizon`. These options can be chosen in the **Order Editor** window. If `N4Horizon` is entered with several rows, the models corresponding to the horizons in each row are examined separately using the Working data. The best model in terms of prediction (or simulation, if K = 0) performance is selected. A figure is shown that illustrates the fit as a function of the horizon. If the `N4Horizon` box is left empty, a default choice is made.

See `n4sid` and `pem` in the "Command Reference" chapter for more information.

User Defined Model Structures

State-Space Structures

The System Identification Toolbox supports user-defined linear state-space models of arbitrary structure. Using the `idmodel idss,` known and unknown parameters in the *A*, *B*, *C*, *D*, *K*, and *X0* matrices can be easily defined both for discrete- and continuous-time models. The `idgrey` object allows you to use a completely arbitrary greybox structure, defined by an M-file. The model object properties can be easily manipulated. See `idss` and `idgrey` in the Chapter 4, "Command Reference" and "Structured State-Space Models with Free Parameters: the idss Model" on page 3-42

To use these structures in conjunction with the GUI, just define the appropriate structure in the MATLAB command window. Then use the **Structure** pop-up menu to select **By Initial Model** and enter the variable

name of the structure in the edit box **Initial Model** in the **Parametric Models** window and select **Estimate**.

Any Model Structure

Arbitrary model structures can be defined using the System Identification Toolbox model objects:

- `idpoly`: Creates Input-output structures for single-output models
- `idss`: Creates Linear State-space models with arbitrary, free parameters
- `idgrey`: Creates completely arbitrary parametrizations of linear systems
- `idarx`: Creates multivariable ARX structures

In addition, all estimation commands create model structures in terms of the resulting models.

Enter the name of any model structure in the box **Orders (or Initial model)** in the window **Parametric Models** and then select **Estimate**. Then the parameters of the model structure are adjusted to the chosen Working Data set. The method is a standard prediction error/maximum likelihood approach that iteratively searches for the minimum of a criterion. Options that govern this search are accessed by the **Option** button in the **Iteration Control** window.

The name of the initial model must be a variable either in the workspace or in the Model Board. In the latter case you can just drag and drop it over the **Orders/Initial model** edit box.

Examining Models

Having estimated a model is just a first step. It must now be examined, compared with other models, and tested with new data sets. This is primarily done using the six **Model View** functions, at the bottom of the main **ident** window:

- Frequency response
- Transient response
- Zeros and poles
- Noise spectrum
- Model output
- Model residuals

In addition, you can double-click on the model's icon to get **Text Information** about the model. Finally, you can export the model to the MATLAB workspace and use any commands for further analysis and model use.

Views and Models

The basic idea is that if a certain **View** window is open (checked), then *all models* in the Model Summary Board that are selected will be represented in the window. The curves in the **View** window can be clicked in and out by selecting and deselecting the models in an online fashion. You select and deselect a model by clicking on its icon. An selected model is marked with a thicker line in its icon.

On color screens, the curves are color coded along with the model icons in the Model Board. Before printing a plot it might be a good idea to separate the line styles (menu item under **Style**). This could also be helpful on black and white screens.

Note that models that are obtained by spectral analysis only can be represented as frequency response and noise spectra, and that models estimated by correlation analysis only can be represented as transient response.

The Plot Windows

The six views all give similar plot windows, with several common features. They have a common menu bar, which covers some basic functions.

First of all, note that there is a zoom function in the plot window. By dragging with the left mouse button down, you can draw rectangles, which will be enlarged when the mouse button is released. By double-clicking, the original axis scales are restored. For plots with two axes, the x-axes scales are locked to each other. A single click on the left mouse button zooms in by a factor of two, while the middle button zooms out. The zoom function can be deactivated if desired. Just select the menu item **Zoom** under **Style**.

Second, by pointing to any curve in the plot, and pressing the right mouse button, the curve will be identified with model name and present coordinates.

The common menu bar covers the following functions.

File

File allows you to copy the current figure to another, standard MATLAB figure window. This might be useful, e.g., when you intend to print a customized plot. Other **File** items cover printing the current plot and closing the plot window.

Options

Options first of all cover actions for setting the axes scaling. This menu item also gives a number of choices that are specific for the plot window in question, like a choice between step response or impulse response in the **Transient response** window.

A most important option is the possibility to show confidence intervals. Each estimated model property has some uncertainty. This uncertainty can be estimated from data. By checking **Show confidence intervals**, a confidence region around the nominal curve (model property) will be marked (by dash-dotted lines). The level of confidence can also be set under this menu item.

Note Confidence intervals are supported for most models and properties, except models estimated using `etfe`, and the k-step ahead prediction-property. For `n4sid`, the covariance properties are actually not fully known. The Cramer-Rao lower limit for the covariance matrix is then used instead.

Style

The style menu gives access to various ways of affecting the plot. You can add gridlines, turn the zoom on and off, and change the linestyles. The menu also covers a number of other options, like choice of units and scale for the axis.

Channel

For multivariate systems, you can choose which input-output channel to examine. The current choice is marked in the figure title.

Help

The **Help** menu has a number of items, which explain the plot and its options.

Frequency Response and Disturbance Spectra

All linear models that are estimated can be written in the form

$$y(t) = G(z)u(t) + v(t)$$

where *G(z)* is the (discrete-time) transfer function of the system and *v(t)* is an additive disturbance. The frequency response or frequency function of the system is the complex-valued function $G(e^{i\omega T})$ viewed as a function of angular frequency ω.

This function is often graphed as a Bode diagram, i.e., the logarithm of the amplitude (the absolute value) of $G(e^{i\omega T})$ as well as the phase (the argument) of $G(e^{i\omega T})$ are plotted against the logarithm of frequency ω in two separate plots. These plots are obtained by checking the **Model View Frequency Response** in the main **ident** window.

The estimated spectrum of the disturbance *v* is plotted as a power spectrum by choosing the **Model View Noise Spectrum**.

If the data is a time series *y* (with no input *u*), then the spectrum of *y* is plotted under **Noise Spectrum**, and no frequency functions are given.

Transient Response

Good and simple insight into a model's dynamic properties is obtained by looking at its step response or impulse response. This is the output of the model when the input is a step or an impulse. These responses are plotted when the **Model View Transient Response** is checked.

It is quite informative to compare the transient response of a parametric model, with the one that was estimated using correlation analysis. If there is good agreement between the two, you can be quite confident that some essentially correct features have been picked up. It is useful to check the confidence intervals around the responses to see what "good agreement" could mean quantitatively.

Many models provide a description of the additive disturbance *v(t)*:

$$v(t) = H(z)e(t)$$

Here *H(z)* is a transfer function that describes how the disturbance *v(t)* can be thought of as generated by sending white noise *e(t)* through it. To display the properties of *H*, you can choose channels (in the **Channel** menu) that have noise components as inputs. The names of these channels are like `e@ynam`, for the noise component that directly affects the output with name `ynam`.

Poles and Zeros

The poles of a system are the roots of the denominator of the transfer function *G(z)*, while the zeros are the roots of the numerator. In particular the poles have a direct influence on the dynamic properties of the system.

The poles and zeros of *G* (and *H*) are plotted by choosing the **Model View Poles and Zeros**.

It is useful to turn on the confidence intervals in this case. They will clearly reveal which poles and zeros could cancel (their confidence regions overlap). That is an indication that a lower order dynamic model could be used.

For multivariable systems it is the poles and zeros of the individual input/output channels that are displayed. To obtain the so called transmission zeros, you will have to export the model and then apply the command `tzero`, provided you have the Control Systems Toolbox.

Compare Measured and Model Output

A very good way of obtaining insight into the quality of a model is to simulate it with the input from a fresh data set, and compare the simulated output with the measured one. This gives a good feel for which properties of the system have been picked up by the model, and which haven't.

This test is obtained by checking the **Model View Model Output**. Then the data set currently in the **Validation Data** box will be used for the comparison.

The fit will also be displayed. This is computed as the percentage of the output variation that is reproduced by the model. So, a model that has a fit of 0% gives the same mean square error as just setting the model output to be the mean of the measured output.

If the model is unstable, or has integration or very slow time constants, the levels of the simulated and the measured output may drift apart, even for a model that is quite good (at least for control purposes). It is then a good idea to evaluate the model's predicted output rather than the simulated one. With a *prediction horizon* of *k*, the *k*-step ahead predicted output is then obtained as follows:

The predicted value *y(t)* is computed from all available inputs $u(s)(s \le t)$ (used according to the model) and all available outputs up to time *t-k*, $y(s)(s \le t-k)$. The simulation case, where no past outputs at all are used, thus formally corresponds to $k=\infty$. To check if the model has picked up interesting dynamic properties, it is wise to let the predicted time horizon (kT, T being the sampling interval) be larger than the important time constants.

Note here that different models use the information in past output data in their predictors in different ways. This depends on the disturbance model. For example, so called Output-Error models (obtained by fixing K to zero for state-space models and setting `na=nc=nd=0` for input output models, see the previous section) do not use past outputs at all. The simulated and the predicted outputs, for any value of k, thus coincide.

Residual Analysis

In a model

$$y(t) = G(z)u(t) + H(z)e(t)$$

the noise source *e(t)* represents that part of the output that the model could not reproduce. It gives the "left-overs" or, in Latin, the *residuals*. For a good model, the residuals should be independent of the input. Otherwise, there would be more in the output that originates from the input and that the model has not picked up.

To test this independence, the cross-correlation function between input and residuals is computed by checking the **Model View Model Residuals**. It is wise to also display the confidence region for this function. For an ideal model the correlation function should lie entirely between the confidence lines for positive

lags. If, for example, there is a peak outside the confidence region for lag *k*, this means that there is something in the output *y(t)* that originates from *u(t-k)* and that has not been properly described by the model. The test is carried out using the Validation Data. If these were not used to estimate the model, the test is quite tough. See also “Model Structure Selection and Validation” on page 3-63.

For a model also to give a correct description of the disturbance properties (i.e., the transfer function *H*), the residuals should be mutually independent. This test is also carried out by the view **Model Residuals**, by displaying the auto-correlation function of the residuals (excluding lag zero, for which this function by definition is 1). For an ideal model, the correlation function should be entirely inside the confidence region.

Text Information

By double-clicking (right mouse button or Ctrl-click) on the model icon, a **Data/ model Info** dialog box opens, which contains some basic information about the model. It also gives a diary of how the model was created, along with the notes that originally were associated with the estimation data set. At this point you can do a number of things:

Present

Selecting the **Present** button displays details of the model in the MATLAB command window. The model's parameters along with estimated standard deviations are displayed, as well as some other notes.

Modify

You can simply type in any text you want anywhere in the **Diary and Notes** editable text field of the dialog box. You can also change the name of the model just by editing the text field with the model name. The color, which the model is associated with in all plots, can also be edited. Enter RGB-values or a color name (like `'y'`) in the corresponding box.

LTI Viewer

If you have the Control System Toolbox, you will see an icon **To LTI Viewer** in the main window. By dragging and dropping a model onto this icon you will open the LTI Viewer. This viewer handles an arbitrary amount of models, but it requires all of them to have the same number of inputs and outputs.

Further Analysis in the MATLAB Workspace

Any model and data object can be exported to the MATLAB workspace by dragging and dropping its icon over the **To Workspace** box in the **ident** window.

Once you have exported the model to the workspace, there are many commands by which you can further transform it, examine it, and convert it to other formats for use in other toolboxes. Some examples of such commands are

`d2c`	Transform to continuous time.
`ss, idss, ssdata`	Convert to state-space representation.
`tf, tfdata`	Convert to transfer function form.
`zpk, zpkdata`	Convert to zeros and poles

Note that the commands `ss`, `tf`, and `zkp` transform the model to the Control System Toolbox's LTI models. Moreover, if you have that toolbox many of its LTI-commands can be applied directly to the model objects of the Identification Toolbox. See "Connections Between the Control System Toolbox and the System Identification Toolbox" on page 3-87

Also, if you need to prepare specialized plots that are not covered by the **Views**, all the System Identification Toolbox commands for computing and extracting simulations, frequency functions, zeros and poles, etc., are available. See the "Tutorial" and "Command Reference" chapters.

Some Further GUI Topics

This section discusses a number of different topics.

Mouse Buttons and Hotkeys

The GUI uses three mouse buttons. If you have fewer buttons on your mouse, the actions associated with the middle and right mouse buttons are obtained by shift-click, alt-click or control-click, depending on the computer.

The Main ident Window

In the main **ident** window the mouse buttons are used to drag and drop, to select/deselect models and data sets, and to double-click to get text information about the object. You can use the left mouse button for all of this. A certain speed-up is obtained if you use the left button for dragging and dropping, the middle one for selecting models and data sets, and the right one for double-clicking (actually for the right button, (Ctrl-click) a single click is sufficient). On a slow machine a double-click from the left button might not be recognized.

The **ident** window also has a number of hotkeys**.** By pressing a keyboard letter when it is the current window, some functions can be quickly activated. These are

- s: Computes **Spectral Analysis Model** using the current options settings. (These can be changed in the dialog window that opens when you choose **Spectral Model** in the **Estimate** pop-up menu.)
- c: Computes **Correlation Analysis Model** using the current options settings.
- q: Computes the models associated with the **Quickstart**.
- d: Opens a dialog window for importing **Data**

Plot Windows

In the various plot windows the action of the mouse buttons depends on whether the zoom is activated or not:

Zoom Active: Then the left and middle mouse buttons are associated with the zoom functions as in the standard MATLAB zoom. Left button zooms in and

the middle one zooms out. In addition, you can draw rectangles with the left button, to define the area to be zoomed. Double-clicking restores the original plot. The right mouse button is associated with special GUI actions that depend on the window. In the **View** plots, the right mouse button is used to identify the curves. Point and click on a curve, and a box will display the name of the model/data set that the curve is associated with, and also the current coordinate values for the curve. In the **Model Selection** plots the right mouse button is used to inspect and select the various models. In the **Prefilter** and **Data Range** plots, rectangles are drawn with this mouse button down, to define the selected range.

Zoom not active: The special GUI functions just mentioned are obtained by any mouse button.

The zoom is activated and deactivated under the menu item **Style.** The default setting differs between the plots. Don't activate the zoom from the command line! That will destroy the special GUI functions. (If you happen to do so anyway, "quit" the window and open it again.)

Troubleshooting in Plots

The function **Auto-range**, which is found under the menu item **Options**, sets automatic scales to the plots. It is also a good function to invoke when you think that you have lost control over the curves in the plot. (This may happen, for example, if you have zoomed in a portion of a plot and then change the data of the plot).

If the view plots don't respond the way you expect them to, you can always "quit" the window and open it again. By quit here we mean using the underlying window system's own quitting mechanism, which is called different things in the different platforms. The normal way to close a window is to use the **Close** function under the menu item **File**, or to uncheck the corresponding check box.

Layout Questions and idprefs.mat

The GUI comes with a number of preset defaults. These include the window sizes and positions, the colors of the different models, and the default options in the different **View** windows.

The window sizes and positions, as well as the options in the plot windows, can of course be changed during the session in the standard way. If you want the

GUI to start with your current window layout and current plot options, select menu item

```
Options > Save preferences
```

in the main **ident** window. This saves the information in a file `idprefs.mat`. This file also stores information about the four most recent sessions with **ident**. This allows the session **File** menu to be correctly initialized. The session information is automatically stored upon exit. The layout and preference information is only saved when the indicated option is selected.

The file `idprefs.mat` is created the first time you close the GUI. It is by default stored in the same directory as your `startup.m` file. If this default does not work, you are prompted for a directory where to store the file. This can be ignored, but then session and preference information cannot be saved.

To change or select a directory for `idprefs.mat`, use the command `midprefs`. See the "Command Reference" chapter for details.

To change model colors and default options to your own customized choice, make a copy of the M-file `idlayout.m` to your own directory (which should be *before* the basic `ident` directory in the `MATLABPATH`), and edit it according to its instructions.

Customized Plots

If you need to prepare hardcopies of your plots with specialized texts, titles and so on, make a copy of the figure first, using **Copy Figure** under the **File** menu item. This produces a copy of the current figure in a standard MATLAB figure format.

For plots that are not covered by the **View** windows, (e.g., Nyquist plots), you have to export the model to the MATLAB workspace and construct the plots there.

What Cannot be Done Using the GUI

The GUI covers primarily everything you would like to do to examine data, estimate models and evaluate and compare models. It does not cover:

- Generation (simulation) of data sets
- Model creation (other than by estimation)

- Model manipulation and conversions
- Recursive (on-line) estimation algorithms

To see what M-files are available in the toolbox for these functions, check the "Tutorial" chapter, as well as the "Simulation and Prediction", "Model Structure Creation", "Manipulating Model Structures", "Model Conversions", and "Recursive Parameter Estimation" tables in the beginning of the "Command Reference" chapter.

Note that at any point you can export a data set or a model to the MATLAB workspace (by dragging and dropping its icon on the **To Workspace** icon). There you can modify and manipulate it any way you want and then import it back into **ident**. You can, for example, construct a continuous-time model from an estimated discrete-time one (using `d2c`), and then use the model views to compare the two.

3

Tutorial

Overview

This chapter has three purposes.

- It gives an overview of System Identification theory, the basic models and disturbance descriptions used, and the character of the basic algorithms. It also provides some practical advice for a number of issues that are essential for a successful application.
- It describes the commands and objects of the System Identification Toolbox, their syntax and use. If you primarily use the graphical user interface (GUI), you will not have to bother about these aspects.
- It also describes the commands that are not reached from the GUI; i.e., simulation, the recursive algorithms and more advanced model structure definitions.

The Toolbox Commands

It may be useful to recognize several *layers* of the System Identification Toolbox. Initially concentrate on the first layer of basic tools, which contains the commands from the System Identification Toolbox that any user must master. You can proceed to the next levels whenever an interest or the need from the applications warrants it. The layers are described in the following paragraphs:

Layer 0: Help Functions. `Help ident` gives an overview of available commands. `idhelp` gives access to a "micromanual" of command-line help, with several subhelps like `idhelp`, `evaluate`, etc.

Layer 1: Basic Tools for Estimating Black-Box Models. The first layer contains the basic tools for estimating models from measured data. It is necessary to know the basics of the data representation and the simple commands to build and evaluate black-box models. The commands are:

- For data representation: `iddata`, `plot`
- For nonparametric estimation of impulse and frequency response: `impulse`, `step`, `spa`
- For estimating black-box models of state-space and input-output type: `pem`, `arx`
- For evaluating models: `compare`, `resid`
- For displaying model characteristics: `bode`, `nyquist`, `pzmap`, `step`, `view`
- Looking at parametric model characteristics: By field referencing, like `Mod.A`, `Mod.dA`, etc.

The corresponding background is given in the next few sections of this "Tutorial."

Layer 2: Creating Models for Simulation and Transforming Models. To define models, to generate inputs and simulate models

```
idarx, idpoly, idss, idinput, sim
```

To transform models to other representations

```
arxdata, polydata, ssdata, tfdata, zpkdata
```

Layer 3: Model Structure Selection. The third layer of the toolbox contains some useful techniques to select orders and delays.

```
arxstruc, selstruc
```

Layer 4: Structured Models and Further Model Conversions. The fourth layer contains transformations between continuous and discrete time, and functions for estimating completely general model structures for linear systems. The commands are

```
c2d, d2c, idss, idgrey, pe, predict
ss, tf, zp, frd
```
(to be used with the Control System Toolbox)

The corresponding material is covered in "Defining Model Structures" on page 3-35 and in "Examining Models" on page 3-49.

Layer 5: Recursive Identification. Recursive (adaptive, online) methods of parameter estimation are covered by the commands

```
rarmax, rarx, rbj, roe, rpem, rplr
```

They are covered in "Recursive Parameter Estimation" on page 3-78.

(See the beginning of the "Command Reference" chapter for a complete list of available functions.)

An Introductory Example to Command Mode

A demonstration M-file called `iddemo.m` provides several examples of what might be typical sessions with the System Identification Toolbox. To start the demo, execute `iddemo` from inside MATLAB.

Before giving a formal treatment of the capabilities and possibilities of the toolbox, this example is designed to get you started with the software quickly. This example is essentially the same as demo #2 in `iddemo`. You may want to invoke MATLAB at this time, execute the demo, and follow along.

Data have been collected from a laboratory scale process. (Feedback's Process Trainer PT326; see page 526 in Ljung, 1999.(For references, see "Reading More About System Identification" on page 1-21 in this book.) The process operates much like a common hand-held hair dryer. Air is blown through a tube after being heated at the inlet to the tube. The input to the process is the power applied to a mesh of resistor wires that constitutes the heating device. The output of the process is the air temperature at the outlet, measured in volts by a thermocouple sensor.

One thousand input-output data points were collected from the process as the input was changed in a random fashion between two levels. The sampling interval is 80 ms. The data were loaded into MATLAB in ASCII form and are now stored as the vectors `y2` (output) and `u2` (input) in the file `dryer2.mat`.

First load the data:

```
load dryer2
```

It contains the input vector `u2`, the output vector `y2`. First form the data object:

```
dry = iddata(y2,u2,0.08);
```

To get information about the data, just type the name:

```
dry
```

To get an overview of all the information contained in the `iddata` object `dry`, type

```
get(dry)
```

For better bookkeeping, give names to input and outputs:

```
dry.InputName = 'Power';

dry.OutputName = 'Temperature';
```

Select the 300 first values for building a model:

```
ze = dry(1:300);
```

Plot the interval from sample 200 to 300:

```
plot(ze(200:300)),
```

Remove the constant levels and make the data zero-mean:

```
ze = detrend(ze);
```

Let us first estimate the impulse response of the system by correlation analysis to get some idea of time constants and the like:

```
impulse(ze,'sd',3)
```

This gives a plot with dash-dotted lines marking a confidence region corresponding to three standard deviations (ca 99.9%). From this it is easy to see if there is a time delay in the system.

The simplest way to get started, is to build a state-space model where the order is automatically determined, using a prediction error method:

```
m1 = pem(ze)
```

When the calculations are finished, a display of the basic information about `m1` is shown. Anytime `m1` is typed, this display is shown. Typing `present(m1)` will give some more information about the model, including uncertainties.

To retrieve the properties of this model we could, e.g., find the `A` matrix of the state space representation by

```
A = m1.a
```

`m1` is a model object, and

```
get(m1)
```

gives a list of all information stored in the model.

How good is this model? One way to find out is to simulate it and compare the model output with measured output. We then select a portion of the original data that was not used to build the model, e.g., from sample 800 to 900:

```
zv = dry(800:900);
zv = detrend(zv);
compare(zv,m1);
```

The Bode plot of the model is obtained by

```
bode(m1)
```

An alternative is to consider the Nyquist plot, and mark uncertainty regions at certain frequencies with ellipses, corresponding to 3 (say) standard deviations:

```
nyquist(m1,'sd',3)
```

We can also compare the step response of the model with one that is directly computed from data (`ze`) in a non-parametric way:

```
step(m1,ze)
```

To study a model with prescribed structure, we compute a difference equation model with two poles, one zero, and three delays:

```
m2 = arx(ze,[2 2 3])
```

This gives a model of the form

$$y(t) + a_1 y(t-T) + a_2 y(t-2T) = b_1 u(t-3T) + b_2 u(t-4T)$$

where T is the sampling interval (here 0.08 seconds). This model, known as an ARX model, tries to explain or compute the value of the output at time t, given previous values of y and u. To compare its performance on validation data with `m1`, type

```
compare(zv,m1,m2);
```

Compute and plot the poles and zeros of the models:

```
pzmap(m1,m2)
```

The uncertainties of the poles and zeros can also be plotted

```
pzmap(m1,m2,'sd',3), % '3' denotes the number of standard
deviations
```

Estimate the frequency response by a nonparametric spectral analysis method:

```
gs = spa(ze);
```

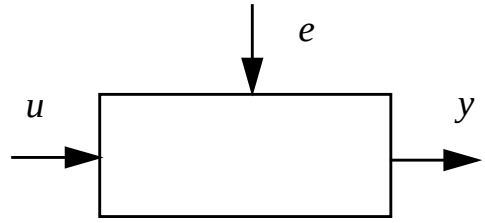
Compare with the frequency functions from the parametric models

```
bode(m1,m2,gs)
```

The System Identification Problem

This section discusses different basic ways to describe linear dynamic systems and also the most important methods for estimating such models.

Impulse Responses, Frequency Functions, and Spectra



The basic input-output configuration is depicted in the figure above. Assuming unit sampling interval, there is an input signal

$$u(t); \qquad t = 1, 2, \ldots, N$$

and an output signal

$$y(t); \qquad t = 1, 2, \ldots, N$$

Assuming the signals are related by a linear system, the relationship can be written

$$y(t) = G(q)u(t) + v(t) \tag{3-1}$$

where q is the shift operator and $G(q)u(t)$ is short for

$$G(q)u(t) = \sum_{k=1}^{\infty} g(k)u(t-k) \tag{3-2}$$

and

$$G(q) = \sum_{k=1}^{\infty} g(k)q^{-k}; \qquad q^{-1}u(t) = u(t-1) \tag{3-3}$$

The numbers $\{g(k)\}$ are called the *impulse response* of the system. Clearly, $g(k)$ is the output of the system at time k if the input is a single (im)pulse at time zero. The function $G(q)$ is called the *transfer function* of the system. This function evaluated on the unit circle $(q = e^{i\omega})$ gives the *frequency function*

$$G(e^{i\omega}) \tag{3-4}$$

In (3-1) $v(t)$ is an additional, unmeasurable disturbance (noise). Its properties can be expressed in terms of its (power) spectrum

$$\Phi_v(\omega) \tag{3-5}$$

which is defined by

$$\Phi_v(\omega) = \sum_{\tau=-\infty}^{\infty} R_v(\tau)e^{-i\omega\tau} \tag{3-6}$$

where $R_v(\tau)$ is the covariance function of $v(t)$

$$R_v(\tau) = Ev(t)v(t-\tau) \tag{3-7}$$

and E denotes mathematical expectation. Alternatively, the disturbance $v(t)$ can be described as filtered white noise

$$v(t) = H(q)e(t) \tag{3-8}$$

where $e(t)$ is white noise with variance λ and

$$\Phi_v(\omega) = \lambda\left|H(e^{i\omega})\right|^2 \tag{3-9}$$

Equations (3-1) and (3-8) together give a *time domain description* of the system

$$y(t) = G(q)u(t) + H(q)e(t) \tag{3-10}$$

where G is the *transfer function* of the system. Equations (3-4) and (3-5) constitute a *frequency domain description*:

$$G(e^{i\omega}); \qquad \Phi_v(\omega) \tag{3-11}$$

The impulse response (3-3) and the frequency domain description (3-11) are called *nonparametric model descriptions* since they are not defined in terms of a finite number of parameters. The basic description (3-10) also applies to the multivariable case; i.e., to systems with several (say *nu*) input signals and several (say *ny*) output signals. In that case $G(q)$ is an *ny*-by-*nu* matrix while $H(q)$ and $\Phi_v(\omega)$ are *ny*-by-*ny* matrices.

Polynomial Representation of Transfer Functions

Rather than specifying the functions *G* and *H* in (3-10) in terms of functions of the frequency variable ω, you can describe them as rational functions of q^{-1} and specify the numerator and denominator coefficients in some way.

A commonly used parametric model is the ARX model that corresponds to

$$G(q) = q^{-nk} \cdot \frac{B(q)}{A(q)}; \qquad H(q) = \frac{1}{A(q)} \tag{3-12}$$

where *B* and *A* are polynomials in the delay operator q^{-1}:

$$\begin{aligned} A(q) &= 1 + a_1 q^{-1} + \ldots\ldots + a_{na} q^{-na} \\ B(q) &= b_1 + b_2 q^{-1} + \ldots\ldots + b_{nb} q^{-nb+1} \end{aligned} \tag{3-13}$$

Here, the numbers *na* and *nb* are the orders of the respective polynomials. The number *nk* is the number of delays from input to output. The model is usually written

$$A(q)y(t) = B(q)u(t-nk) + e(t) \tag{3-14}$$

or explicitly

$$\begin{aligned} &y(t) + a_1 y(t-1) + \ldots\ldots + a_{na} y(t-na) = \\ &b_1 u(t-nk) + b_2 u(t-nk-1) + \ldots\ldots + b_{nb} u(t-nk-nb+1) + e(t) \end{aligned} \tag{3-15}$$

Note that (3-14) - (3-15) apply also to the multivariable case, with *ny* output channels and *nu* input channels. Then $A(q)$ and the coefficients a_i become *ny*-by-*ny* matrices, $B(q)$ and the coefficients b_i become *ny*-by-*nu* matrices.

Another very common, and more general, model structure is the ARMAX structure

$$A(q)y(t) = B(q)u(t-nk) + C(q)e(t) \quad \textbf{(3-16)}$$

Here, $A(q)$ and $B(q)$ are as in (3-13), while

$$C(q) = 1 + c_1 q^{-1} + \ldots + c_{nc} q^{-nc}$$

An *Output-Error* (OE) structure is obtained as

$$y(t) = \frac{B(q)}{F(q)} u(t-nk) + e(t) \quad \textbf{(3-17)}$$

with

$$F(q) = 1 + f_1 q^{-1} + \ldots + f_{nf} q^{-nf}$$

The so-called *Box-Jenkins* (BJ) model structure is given by

$$y(t) = \frac{B(q)}{F(q)} u(t-nk) + \frac{C(q)}{D(q)} e(t) \quad \textbf{(3-18)}$$

with

$$D(q) = 1 + d_1 q^{-1} + \ldots + d_{nd} q^{-nd}$$

All these models are special cases of the general parametric model structure:

$$A(q)y(t) = \frac{B(q)}{F(q)} u(t-nk) + \frac{C(q)}{D(q)} e(t) \quad \textbf{(3-19)}$$

The variance of the white noise $\{e(t)\}$ is assumed to be λ.

Within the structure of (3-19), virtually all of the usual linear black-box model structures are obtained as special cases. The ARX structure is obviously obtained for $nc = nd = nf = 0$. The ARMAX structure corresponds to $nf = nd = 0$. The ARARX structure (or the “generalized least squares model”) is obtained for $nc = nf = 0$, while the ARARMAX structure (or “extended matrix model”) corresponds to $nf = 0$. The Output-Error model is obtained with $na = nc = nd = 0$, while the Box-Jenkins model corresponds to $na = 0$. (See Section 4.2 in Ljung (1999) for a detailed discussion.)

The same type of models can be defined for systems with an arbitrary number of inputs. They have the form

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t-nk_1) + \ldots + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t-nk_{nu}) + \frac{C(q)}{D(q)}e(t) \quad \textbf{(3-20)}$$

State-Space Representation of Transfer Functions

A common way of describing linear systems is to use the *state-space form*:

$$\begin{aligned} x(t+1) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) + v(t) \end{aligned} \quad \textbf{(3-21)}$$

Here the relationship between the input $u(t)$ and the output $y(t)$ is defined via the *nx*-dimensional *state vector* $x(t)$. In transfer function form (3-21) corresponds to (3-1) with

$$G(q) = C(qI_{nx} - A)^{-1}B + D \quad \textbf{(3-22)}$$

Here I_{nx} is the *nx* by *nx* identity matrix. Clearly (3-21) can be viewed as one way of parametrizing the transfer function: Via (3-22) $G(q)$ becomes a function of the elements of the matrices *A*, *B*, *C*, and *D*.

To further describe the character of the noise term $v(t)$ in (3-21) a more flexible *innovations* form of the state-space model can be used:

$$\begin{aligned} x(t+1) &= Ax(t) + Bu(t) + Ke(t) \\ y(t) &= Cx(t) + Du(t) + e(t) \end{aligned} \quad \textbf{(3-23)}$$

This is equivalent to (3-10) with $G(q)$ given by (3-22) and $H(q)$ by

$$H(q) = C(qI_{nx} - A)^{-1}K + I_{ny} \quad \textbf{(3-24)}$$

Here *ny* is the dimension of $y(t)$ and $e(t)$.

It is often possible to set up a system description directly in the innovations form (3-23). In other cases, it might be preferable to describe first the nature of disturbances that act on the system. That leads to a stochastic state-space model

$$x(t+1) = Ax(t) + Bu(t) + w(t)$$
$$y(t) = Cx(t) + Du(t) + e(t) \qquad \textbf{(3-25)}$$

where $w(t)$ and $e(t)$ are stochastic processes with certain covariance properties. In stationarity and from an input-output view, (3-25) is equivalent to (3-23) if the matrix K is chosen as the steady-state *Kalman gain*. How to compute K from (3-25) is described in the Control System Toolbox documentation.

Continuous-Time State-Space Models

It is often easier to describe a system from physical modeling in terms of a continuous-time model. The reason is that most physical laws are expressed in continuous time as differential equations. Therefore, physical modeling typically leads to state-space descriptions like

$$\dot{x}(t) = Fx(t) + Gu(t)$$
$$y(t) = Hx(t) + Du(t) + v(t) \qquad \textbf{(3-26)}$$

Here, $\dot{x}$ means the time derivative of x. If the input is piece-wise constant over time intervals $kT \le t < (k+1)T$, then the relationship between $u[k] = u(kT)$ and $y[k] = y(kT)$ can be exactly expressed by (3-21) by taking

$$A = e^{FT}; \qquad B = \int_0^T e^{F\tau} G\, d\tau; \qquad C = H \qquad \textbf{(3-27)}$$

and associate $y(t)$ with $y[t]$, etc. If you start with a continuous-time innovations form

$$\dot{x}(t) = Fx(t) + Gu(t) + \tilde{K}e(t)$$
$$y(t) = Hx(t) + Du(t) + e(t) \qquad \textbf{(3-28)}$$

the discrete-time counterpart is given by (3-23) where still the relationships (3-27) hold. The exact connection between $\tilde{K}$ and K is somewhat more complicated, though. An *ad hoc* solution is to use

$$K = \int_0^T e^{F\tau} \tilde{K}\, d\tau; \qquad \textbf{(3-29)}$$

in analogy with G and B. This is a good approximation for short sampling intervals T.

Estimating Impulse Responses

Consider the descriptions (3-1) and (3-2). To directly estimated the impulse response coefficients, also in the multivariable case, it is suitable to define a high order Finite Impulse Response (FIR) model:

$$y(t) = g(0)u(t) + g(1)u(t-1) + \ldots + g(n)u(t-n) \qquad \textbf{(3-30)}$$

and estimate the g-coefficients by the linear least squares method. In fact, to check if there are non-causal effects from input to output, e.g., due to feedback from y in the generation of u (closed loop data), g for negative lags can also be estimated:

$$\begin{aligned} y(t) = {} & g(-m)u(t+m) + \ldots + g(-1)u(t+1) + g(0)u(t) + \\ & g(1)u(t-1) + \ldots + g(n)u(t-n) \end{aligned} \qquad \textbf{(3-31)}$$

If u is white noise, the impulse response coefficients will be correctly estimated, even if the true dynamics from u to y is more complicated than these models. Therefore it is natural to filter both the output and the input through a filter that makes the input sequence as white as possible, before estimating the g. This is the essence of *correlation analysis* for estimating impulse responses.

Estimating Spectra and Frequency Functions

This section describes methods that estimate the frequency functions and spectra (3-11) directly. The cross-covariance function $R_{yu}(\tau)$ between $y(t)$ and $u(t)$ is defined as analogously to (3-7). Its Fourier transform, the cross spectrum, $\Phi_{yu}(\omega)$ is defined analogously to (3-6). Provided that the input $u(t)$ is independent of $v(t)$, the relationship (3-1) implies the following relationships between the spectra:

$$\begin{aligned} \Phi_y(\omega) &= \left|G(e^{i\omega})\right|^2 \Phi_u(\omega) + \Phi_v(\omega) \\ \Phi_{yu}(\omega) &= G(e^{i\omega})\Phi_u(\omega) \end{aligned} \qquad \textbf{(3-32)}$$

By estimating the various spectra involved, the frequency function and the disturbance spectrum can be estimated as follows:

Form estimates of the covariance functions (as defined in (3-7)) $\hat{R}_y(\tau)$, $\hat{R}_{yu}(\tau)$, and $\hat{R}_u(\tau)$, using

$$\hat{R}_{yu}(\tau) = \frac{1}{N}\sum_{t=1}^{N} y(t+\tau)u(t) \qquad \textbf{(3-33)}$$

and analog expressions for the others. Then, form estimates of the corresponding spectra

$$\hat{\Phi}_y(\omega) = \sum_{\tau=-M}^{M} \hat{R}_y(\tau)W_M(\tau)e^{-i\omega\tau} \qquad \textbf{(3-34)}$$

and analogously for $\hat{\Phi}_u$ and $\hat{\Phi}_{yu}$. Here $W_M(\tau)$ is the so-called *lag window* and M is the width of the lag window. The estimates are then formed as

$$\hat{G}_N(e^{i\omega}) = \frac{\hat{\Phi}_{yu}(\omega)}{\hat{\Phi}_u(\omega)}; \qquad \hat{\Phi}_v(\omega) = \hat{\Phi}_y(\omega) - \frac{\left|\hat{\Phi}_{yu}(\omega)\right|^2}{\hat{\Phi}_u(\omega)} \qquad \textbf{(3-35)}$$

This procedure is known as *spectral analysis*. (See Chapter 6 in Ljung (1999).)

Estimating Parametric Models

Given a description (3-10) and having observed the input-output data u, y, the (prediction) errors $e(t)$ in (3-10) can be computed as:

$$e(t) = H^{-1}(q)[y(t) - G(q)u(t)] \qquad \textbf{(3-36)}$$

These errors are, for given data y and u, functions of G and H. These in turn are parametrized by the polynomials in (3-14)-(3-19) or by entries in the state-space matrices defined in (3-26)–(3-29). The most common parametric identification method is to determine estimates of G and H by minimizing

$$V_N(G, H) = \sum_{t=1}^{N} e^2(t) \qquad \textbf{(3-37)}$$

that is

$$[\hat{G}_N, \hat{H}_N] = \underset{}{argmin} \sum_{t=1}^{N} e^2(t) \qquad \textbf{(3-38)}$$

This is called a *prediction error method*. For Gaussian disturbances it coincides with the maximum likelihood method. (See Chapter 7 in Ljung (1999).)

A somewhat different philosophy can be applied to the ARX model (3-14). By forming filtered versions of the input

$$N(q)s(t) = M(q)u(t) \qquad \textbf{(3-39)}$$

and by multiplying (3-14) with $s(t-k), k = 1, 2, \ldots, na$ and $u(t-nk+1-k), k = 1, 2, \ldots, nb$ and summing over t, the noise in (3-14) can be correlated out and solved for the dynamics. This gives the *instrumental variable* method, and $s(t)$ are called the instruments. (See Section 7.6 in Ljung (1999).)

Subspace Methods for Estimating State-Space Models

The state-space matrices *A*, *B*, *C*, *D*, and *K* in (3-23) can be estimated directly, without first specifying any particular parametrization by efficient *subspace methods*. The idea behind this can be explained as follows: If the sequence of state vectors *x(t)* were known, together with *y(t)* and *u(t)*, Eq. (3-23) would be a linear regression, and *C* and *D* could be estimated by the least squares method. Then *e(t)* could be determined, and treated as a known signal in (3-23), which then would be another linear regression model for *A*, *B* and *K*. (One could also treat (3-21) as a linear regression for *A*, *B*, *C*, and *D* with *y(t)* and *x(t+1)* as simultaneous outputs, and find the joint process and measurement noises as the residuals from this regression. The Kalman gain *K* could then be computed from the Riccati equation.) Thus, once the states are known, the estimation of the state-space matrices is easy.

How to find the states *x(t)*? All states in representations like (3-23) can be formed as linear combinations of the *k*-step ahead predicted outputs (*k*=1,2,...,*n*). It is thus a matter of finding these predictors, and then selecting a basis among them. The subspace methods form an efficient and numerically reliable way of determining the predictors by projections directly on the observed data sequences. See Sections 7.3 and 10.6 in Ljung (1999). For more details, see the references under `n4sid` in the “Command Reference” chapter.

Data Representation and Nonparametric Model Estimation

This and the following sections give an introduction to the basic functions in the System Identification Toolbox. Not all of the options available when using the functions are described here; see the "Command Reference" chapter and the online Help facility.

Data Representation

The observed output and input signals, $y(t)$ and $u(t)$, are represented as *column vectors* `y` and `u`. Row `k` corresponds to sample number k. For multivariable systems, each input (output) component is represented as a column vector, so that `u` becomes an N-by-nu matrix (N = number of sampled observations, nu = number of input channels). The output-input data is collectively represented in the `iddata` format. This is the basic object for dealing with signals in the toolbox. It is used by most of the commands. It is created by

```
Data = iddata(y,u,Ts)
```

where `y` is a column vector or an N-by-ny matrix. The columns of `y` correspond to the different output channels. Similarly `u` is a column vector or an N-by-nu matrix containing the signals of the input channels. `Ts` is the sampling interval. This construction is sufficient for almost all purposes.

The data is then plotted by `plot(Data)` and portions of the data record are selected as in

```
ze = Data(1:300)
```

The signals in the output channels are retrieved by `Data.OutputData` or for short, `Data.y`. Similarly the input signals are obtained by `Data.InputData` or `Data.u`.

For a time series (no input channels) use `Data = iddata(y)`, or let `u = [ ]`. An `iddata` object can also contain just an input, by letting `y = [ ]`.

The sampling interval can be changed by `set(Data,'Ts',0.3)` or, simpler, by

```
Data.Ts = 0.3
```

More details about the `iddata` object is given at the end of this section.

Correlation Analysis

The correlation analysis procedure described in "Estimating Impulse Responses" on page 3-15 is implemented in the function `impulse`:

```
impulse(Data)
```

This function plots the estimated impulse response. Adding an argument `'sd'` as in:

```
impulse(Data,'sd',3)
```

it also marks a confidence region corresponding to (in this case) three standard deviations. The result can be stored and replotted:

```
ir = impulse(Data)
impulse(ir,'sd',3)
```

An alternative is the command `step` that plots the step response, calculated from the impulse estimate:

```
step(Data)
```

Spectral Analysis

The function `spa` performs spectral analysis according to the procedure in (3-35)–(3-37):

```
g = spa(Data)
```

Here `Data` contains the output-input data in the `iddata` object as above. `g` is returned as an `idfrd` (Identified frequency domain) model object, that contains the estimated frequency function $\hat{G}_N$ and the estimated disturbance spectrum $\hat{\Phi}_v$ in (3-37), as well as estimated uncertainty covariances. The `idfrd` object is described in the "Command Reference" chapter, but for normal use you do not have to bother about these details. The frequency function, or *frequency response*, G in `g` can be graphed by the commands `bode`, `ffplot`, or `nyquist`. The noise spectrum is retrieved by `g('n')` (`'n'` for "Noise") so

```
g = spa(Data)
bode(g)
bode(g('n'))
```

performs the spectral analysis, and plots first G and then Φ_v. `bode` gives logarithmic amplitude and frequency scales (in rad/sec) and linear phase scale,

while `ffplot` gives linear frequency scales (in Hz). The uncertainty of the estimates is displayed by adding the argument `'sd'` as in

```
bode(g,'sd',3)
```

which will display, by dash-dotted lines, a confidence region around the estimate, that corresponds to (in this case) three standard deviations. Adding an argument `'fill'` will show the uncertainty region instead as a filled region.

```
bode(g,'sd',3,'fill')
```

Similarly

```
nyquist(g)
```

gives a Nyquist plot of the frequency function, i.e. a plot of the real part versus the imaginary part of G

If `Data = y` is a time series, that is `Data` has no input channel, `spa` returns an estimate of the spectrum of that signal:

```
g= spa(y)
ffplot(g)
```

In the computations (3-35)-(3-37), `spa` uses as a lag window the Hamming window for $W(\tau)$ with a default length M equal to the minimum of 30 and a tenth of the number of data points. This window size M can be changed to an arbitrary number by

```
g = spa(Data,M)
```

The rule is that as `M` increases, the estimated frequency functions show sharper details, but are also more affected by random disturbances. A typical sequence of commands that test different window sizes is

```
g10 = spa(Data,10)
g25 = spa(Data,25)
g50 = spa(Data,50)
bode(g10, g25, g50)
```

An empirical transfer function estimate is obtained as the ratio of the output and input Fourier transforms with

```
g = etfe(Data)
```

This can also be interpreted as the spectral analysis estimate for a window size that is equal to the data length. For time series, `etfe` gives the *periodogram* as a spectral estimate. The function also allows some smoothing of the crude estimate; it can be a good alternative for signals and systems with sharp resonances. See the "Command Reference" chapter for more information.

More on the Data Representation in iddata

Some Bookkeeping Facilities

The input and output channels are given default names like `y1`, `y2`, `u1`, `u2`, etc. The channel names can be set by

```
set(Data,'InputName',{'Voltage','Current'},'OutputName','Temper
ature')
```

(two inputs and one output is this example) and these names will then follow the object and appear in all plots. The names will also be inherited by models that are estimated from the data.

Similarly, channel units can be specified using the properties `OutputUnit` and `InputUnit`. These units, when specified, will be used in plots.

The timepoints associated with the data samples are determined by the sampling interval `Ts` and the time of the first sample, `Tstart`:

```
Data.Tstart = 24
```

The actual time point values are given by the property, `SamplingInstants`, as in

```
plot(Data.sa,Data.u)
```

for a plot of the input with correct time points. Autofill is used for all properties, and they are case insensitive. For easy writing `'u'` is synonymous to `'Input'` and `'y'` to `'Output'` when referring to the properties.

Manipulating Channels

An easy way to set and retrieve channel properties is to use subscripting. The subscripts are defined as

```
Data(samples,outputs,inputs),
```

so `Dat(:,3,:)` is the data object obtained from `Dat` by keeping all input channels, but only output channel 3. (Trailing ':'s can be omitted so `Dat(:,3,:)= Dat(:,3)`.)

The channels can also be retrieved by their names, so that

```
Dat(:,{'speed','flow'},[ ])
```

is the data object where the indicated output channels have been selected and no input channels are selected.

Moreover

```
Dat1(101:200,[3 4],[1 3]) = Dat2(1001:1100,[1 2],[6 7])
```

will change samples 101 to 200 of output channels 3 and 4 and input channels 1 and 3 in the `iddata` object `Dat1` to the indicated values from `iddata` object `Dat2`. The names and units of these channels will then also be changed accordingly.

To add new channels, use horizontal concatenation of `iddata` objects

```
Dat =[Dat1, Dat2];
```

(see “Adding Channels” on page 3-24) or add the data record directly, so that

```
Dat.u(:,5) = u
```

will add a fifth input to `Dat`.

Nonequal Sampling

The property `SamplingInstants` gives the sampling instants of the data points. It can always be retrieved by `get(Dat,'SamplingInstants')` (or `Dat.s`) and is then computed from `Dat.Ts` and `Dat.Tstart`. `SamplingInstants` can also be set to an arbitrary vector of the same length as the data, so that nonequal sampling can be handled. `Ts` is then automatically set to [ ]. Most of the estimation routines, though, do not handle unequally sampled data.

Multiple Experiments

The `iddata` object can also store data from separate experiments. The property `ExperimentName` is used to separate the experiments. The number of data as well as the sampling properties can vary from experiment to experiment, but the input and output channels must be the same. (Use `NaN` to fill unmeasured

channels in certain experiments.) The data records will be cell arrays, where the cells contain data from each experiment.

Multiple experiments can be defined directly by letting the `'y'` and `'u'` properties as well as `'Ts'` and `'Tstart'` be cell arrays.

It is normally easier to create multi-experiment data by merging experiments as in

```
Dat = merge(Dat1,Dat2)
```

See `merge (iddata)` in the "Command Reference." Storing multiple experiments as one `iddata` object may be very useful to handle experimental data that have been collected on different occasions, or when a data set has been split up to remove "bad" portions of the data. All the toolbox's routines accept multiple experiment data.

Experiments can be set and retrieved by subscripting with curly brackets. `Dat{3}` is experiment number 3 and `Dat{'Day1','Day4'}` retrieves the two experiments with the indicated names.

The subscripting can be combined: `Dat{3}(1:100,[2,3],[4:8])` gives the 100 first samples of output channels 2 and 3 and input channels 4 to 8 of experiment number 3. It can also be used for subassignment

```
Dat{'Run4'} = Dat2
```

adds the data in `Dat2` as a new experiment with name `'Run4'`. See iddemo # 8 for an illustration of how multiple experiments can be used.

iddata Properties

Type `get(Dat)` or see `iddata` in the "Command Reference" for a complete list of `iddata` properties.

Subreferencing

The samples, outputs and input channels can be referenced according to

```
Data(samples,outputs,inputs)
```

Use colon (:) to denote all samples/channels and the empty matrix ([ ]) to denote no samples/channels. The channels can be referenced by number or by name. For several names a cell array must be used:

```
Dat2 = Dat(:,'y3',{'u1','u4'})
```

```
Dat2 = Dat(:,3,[1 4])
```

Logical expressions will also work.

```
Dat3 = Dat2(Dat2.sa>1.27&Dat2.sa<9.3)
```

will select the samples with time marks between 1.27 and 9.3.

Subreferencing with curly brackets refers to the experiment:

```
Data{Experiment}(samples,outputs,inputs)
```

Any subreferenced variable can also be assigned.

```
Data{'Exp3'}.y = flow(1:700,:)
Data(1:10,1,1) = Dat1(101:110,2,3)
```

Adding Channels

```
Dat = [Dat1,Dat2,...,DatN]
```

creates an iddata object `Dat`, consisting of the input and output channels in `Dat1,... DatN`. Default channel names (`'u1', 'u2', 'y1' , 'y2'` etc) will be changed so that overlaps in names are avoided, and the new channels will be added.

If `Datk` contains channels with user-specified names that are already present in the channels of `Datj, j<k`, these new channels will be ignored.

Adding Samples

```
Dat = [Dat1;Dat2;... ;DatN]
```

creates an `iddata` object `Dat` whose signals are obtained by stacking those of `Datk` on top of each other. That is

```
Dat.y = [Dat1.y;Dat2.y; ... DatN.y]
```

and similarly for the inputs. The `Datk` objects must all have the same number of channels and experiments.

Parametric Model Estimation

The System Identification Toolbox contains several functions for parametric model estimation. They all share the same command structure

```
m = function(Data,modstruc)
m =   ...
function(Data,modstruc,'Property1',Value1,...'PropertyN',ValueN
)
```

The argument `Data` is an `iddata` object that contains the output and input data sequences, while `modstruc` specifies the particular structure of the model to be estimated. The resulting estimated model is contained in `m`. It is a model object that stores various information. The model objects will be described in "Defining Model Structures" on page 3-35, but for most use of the toolbox, you do not have to consider the details of these objects. Just typing the model name

```
m
```

will give a concise display of the model. The command

```
present(m)
```

gives some more details, while

```
get(m)
```

gives a complete list of the model's properties. The property values can be easily retrieved just by dot-referencing; for example,

```
m.par
```

retrieves the estimated parameters

In the function call `(...,'Property1', Value1,...,'PropertyN',ValueN)` is a list of properties that can be assigned to affect the model structure, as well as the estimation algorithm. A list of typical properties is given at the end of this section. The model `m` is also immediately prepared for displaying and analyzing its characteristics as well as for transforming it to other representations, as in

```
bode(m)
compare(Data,m)
[A,B,C,D, K] = ssdata(m)
```

See"Examining Models" on page 3-49 for a detailed discussion of these possibilities.

In the following, `Data` denotes an `iddata` object that contains the input output data as described in the previous section. It may also just contain an output signal, i.e., a time series.

ARX Models

To estimate the parameters a_i and b_i of the ARX model (3-14), use the function `arx`:

```
m  = arx(Data,[na nb nk])
```

Here `na`, `nb`, and `nk` are the corresponding orders and delays in (3-15) on page 3-11 that define the exact model structure. The function `arx` implements the least squares estimation method, using QR-factorization for overdetermined linear equations.

An alternative is to use the Instrumental Variable (IV) method described in connection with (3-39). This is obtained with

```
m = iv4(Data,[na nb nk])
```

which gives an automatic (and approximately optimal) choice of the filters N and M in (3-39). (See the procedure (15.21)-(15.26) in Ljung (1999).)

Both `arx` and `iv4` are applicable to arbitrary multivariable systems. If you have *ny* outputs and *nu* inputs, the orders are defined accordingly: `na` is an *ny*-by-*ny* matrix whose *i* -*j* entry gives the order of the polynomial that relates past values of y_j to the current value of y_i (i.e., past values of y_j up to $y_j(t - na(i, j))$are used when predicting $y_i(t)$) . Similarly, the *i* -*j* entries of the *ny*-by-*nu* matrices `nu` and `nk`, respectively, give the order and delay from input number *j* when predicting output number *i*. (See "Multivariable ARX Models: The idarx Model" on page 3-37 and the"Command Reference" chapter for exact details.)

AR Models

For a single output signal $y(t)$ the counterpart of the ARX model is the AR model:

$$A(q)y(t) = e(t) \qquad \textbf{(3-40)}$$

The arx command also covers this special case

```
m = arx(y,na)
```

but for scalar signals more options are offered by the command

```
m = ar(y,na)
```

which has an option that allows you to choose the algorithm from a group of several popular techniques for computing the least squares AR model. Among these are Burg's method, a geometric lattice method, the Yule-Walker approach, and a modified covariance method. (See the "Command Reference" chapter for details.) The counterpart of the iv4 command is

```
m = ivar(y,na)
```

which uses an instrumental variable technique to compute the AR part of a time series.

General Polynomial Black-Box Models

Based on the prediction error method (3-38), you can construct models of basically any structure. For the general model (3-19), there is the function

```
m = pem(Data,nn)
```

where nn gives all the orders and delays

```
nn = [na nb nc nd nf nk]
```

The nonzero orders of the model can also be defined as property name/property value pairs as in

```
m = pem(Data,'na',na,'nb',nb,'nc',nc,'nk',nk)
```

The input parameters are defined on page 3-12. The pem command covers all cases of black-box linear system models. For the common special cases

```
m = armax(Data,[na nb nc nk])
m = oe(Data,[nb nf nk])
m = bj(Data,[nb nc nd nf nk])
```

can be used. These handle the model structures (3-16), (3-17) and (3-18), respectively.

All the routines also cover single-output, multi-input systems of the type

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t-nk_1) + \ldots + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t-nk_{nu}) + \frac{C(q)}{D(q)}e(t) \quad \textbf{(3-41)}$$

where `nb`, `nf`, and `nk` are row vectors of the same lengths as the number of input channels containing each of the orders and delays

```
nb = [nb1 ...  nbnu];
nf = [nf1 ...  nfnu];
nk = [nk1 ...  nknu];
```

These parameter estimation routines require an iterative search for the minimum of the function (3-39). This search uses a special start-up procedure based on least squares and instrumental variables (the details are given as Equation (10.79) in Ljung (1999)). From the initial estimate, a Gauss-Newton minimization procedure is carried out until the norm of the Gauss-Newton direction is less than a certain tolerance. See Ljung (1999), Section 11.2 or Dennis and Schnabel(1983) for details. See also the entry at the end of this section on optional variables associated with the search.

The estimation routines also return the estimated covariance matrix of the estimated parameter vector as part of `m`. This reflects the reliability of the estimates. The covariance matrix estimate is computed under the assumption that it is possible to obtain a "true" description in the given structure.

The routines `pem`, `armax`, `oe`, and `bj` can also be started at any initial value `mi` that is a model object by replacing `nn` by `mi`. For example

```
m = pem(Data,mi)
```

While the search is typically initialized using the built-in start-up procedure giving just orders and delays (as described above), the ability to force a specific initial condition is useful in several contexts. Some examples are mentioned in "Initial Parameter Values" on page 3-90.

Information about how the minimization progresses can be supplied to the MATLAB command window by the property `trace`. See the list at the end of this section.

State-Space Models

Black-Box, Discrete Time Parametrizations

Suppose first that there is no particular knowledge about the internal structure of the discrete-time state-space model (3-15). Any linear model is sought. A simple approach then is to use

```
m = pem(Data)
```

This estimates a state-space model of an order (among 1 to 10) that seems reasonable.

To find a black-box model of a certain order `n`, use

```
m = pem(Data,n)
```

To get a plot, from which the order can be determined among a list of orders `nn = [n1,n2,...,nN]`, use

```
m = pem(Data,'nx',nn)
```

All these black-box models are initialized by the subspace method `n4sid`. To obtain the estimate from this routine, use

```
m = n4sid(Data,n)
```

Arbitrarily Structured Models in Discrete and Continuous Time

For state-space models of given structure, most of the effort involved relates to defining and manipulating the structure. This is discussed in “Structured State-Space Models with Free Parameters: the idss Model” on page 3-42 and onwards. Once the structure is defined as `ms`, you can estimate its parameters with

```
m = pem(Data,ms)
```

When the systems are multi-output, the following criterion is used for the minimization,

$$\min \det \sum_{t=1}^{N} e(t)e^{T}(t) \qquad \textbf{(3-42)}$$

which is the maximum likelihood criterion for Gaussian noise with unknown covariance matrix.

The numerical minimization of the prediction error criterion (3-39) or (3-42) can be a difficult problem for general model parametrizations. The criterion, as a function of the free parameters, can define a complicated surface with many local minima, narrow valleys, and so on. This may require substantial interaction from the user, in providing reasonable initial parameter values, and also by freezing certain parameter values (using the property `FixedParameters`) while allowing others to be free. Note that `pem` easily allows the freezing of any parameters to their current/nominal values. The model structure can also be directly manipulated as will be described in "Structured State-Space Models with Free Parameters: the idss Model" on page 3-42. A procedure that is often used for state-space models is to allow the noise parameter in the *K* matrix free, only when a reasonable model of the dynamic part has been obtained.

Optional Variables

The estimation functions accept a list of property name/property value pairs that may affect both the model structure and the estimation algorithm. For complete lists of these properties, see `idmodel`, algorithm properties, `idarx`,`idpoly`, `idss`, and `idgrey` in the "Command Reference" chapter. Here we list some of them. Note that any property, as well as values that are strings, can be entered as any unambiguous, case insensitive, abbreviation, as is

```
m = pem(Data,mi,'fo','si').
```

Note 1: 'Algorithm' is a property of `idmodel`. Any algorithm property can be separately set as above. Also, if you have a standard algorithm setup, that you prefer, you can set those properties simultaneously as in `m = pem(Data,mi,'alg',myalg)`

Note 2: The algorithm properties, like all other model properties, will be inherited by the resulting model `m`. If you continue the estimation using `m` as initial model, all previously set algorithm features will thus apply, unless you specify otherwise.

Applying to All Estimation Methods

The following properties apply to all estimation methods:

- Focus
- MaxSize
- FixedParameter

`Focus`: This property affects the weighting applied to the fit between the model and the data. It can be used to assure that the model approximates the true system well over certain frequency intervals. `Focus` can assume the following values:

- *Prediction*: This is the default and means that the model is determined by minimizing the prediction errors. It corresponds to a frequency weighting that is given by the input spectrum times the inverse noise model. Typically, this favors a good fit at high frequencies. From a statistical variance point of view, this is the optimal weighting, but then the approximation aspects (bias) of the fit are neglected.
- *Simulation:* This means that frequency weighting of the transfer function fit is given by the input spectrum. Frequency ranges where the input has considerable power will thus be better described by the model. In other words, the model approximation is such that the model will produce as good simulations as possible, when applied to inputs with the same spectra as used for the estimation. For models that have no disturbance model (A=C=D for `idpoly` models and K=0 for `idss` models) there is no difference between the Simulation and Prediction values. For models with a disturbance description, this is estimated by a prediction error method, keeping the estimated transfer function from input to output fixed. The resulting model is guaranteed to be stable.
- *Stability:* The algorithm is modified so that a stable model is guaranteed, but the weighing still correspond to prediction.

- *Any SISO linear filter.* Then the transfer function from input to output is determined by a frequency fit with this filter times the input spectrum as weighting function. The noise model is determined by a prediction error method, keeping the transfer function estimate fixed. To obtain a good model fit over a special frequency range, the filter should thus be chosen with a passband over this range. For a model with no disturbance model, the result is the same as first applying prefiltering to data using `idfilt`. The filter can be specified in a few different ways:
  - as any single-input-single-output `idmodel`
  - as a `ss`, `tf`, or `zpk` model from the Control System Toolbox
  - as `{A,B,C,D}` with the state-space matrices for the filter
  - as `{numerator, denominator}` with the transfer function numerator/ denominator of the filter

`MaxSize`**:** No matrix with more than `MaxSize` elements is formed by the algorithm. Instead, `for` loops will be used. `MaxSize` will thus decide the memory/speed trade-off, and can prevent slow use of virtual memory. `MaxSize` can be any positive integer, but it is of course required that the input-output data themselves contain less than `MaxSize` elements. The default value of `MaxSize` is `Auto`, which means that the value is determined in the M-file `idmsize`. This file may be edited by the user to optimize speed on a particular computer. See also “Memory - Speed Trade-Offs” on page 3-89.

`FixedParameter`: A list of parameters that will be kept fixed to the nominal/ initial values and not estimated. This is a vector of integers containing the indices of the fixed parameters, or a cell array of parameter names. If names are used, wildcard entries apply, which may be convenient if you have groups of parameters in your model. See the reference page of `Algorithm properties`.

Algorithm Properties That Apply to n4sid, Estimating State-Space Models

The properties that apply to sub-space model estimation are

- N4Weight
- N4Horizon

These properties then also apply to `pem` for estimating black-box state-space models, since `pem` is then initialized by the `n4sid` estimate

`N4Weight`: This property determines some weighting matrices used in the singular-value decomposition that is a central step in the algorithm. Two choices are offered: `moesp` that corresponds to the MOESP algorithm by Verhaegen and `cva` which is the canonical variable algorithm by Larimore. The default value is `N4Weight = Auto`, which gives an automatic choice between the two options.

`N4Horison`: Determines the prediction horizons forward and backward, used by the algorithm. This is a row vector with three elements: `N4Horison =[r sy su]`, where `r` is the maximum forward prediction horizon, i.e., the algorithms uses up to `r`-step ahead predictors. `sy` is the number of past outputs, and `su` is the number of past inputs that are used for the predictions. See Ljung (1999), pages 345-348. These numbers may have a substantial influence of the quality of the resulting model, and there are no simple rules for choosing them. Making `N4Horizon` a k-by-3 matrix means that each row of `N4Horison` will be tried out, and the value that gives the best (prediction) fit to data will be selected. If you specify only one column in `N4Horizon`, the interpretation is `r=sy=su`. The default choice is
`N4Horizon = Auto`, which uses the Akaike Information Criterion (AIC) for the selection of `sy` and `su`. See the reference page for `n4sid` for literature references.

Properties That Apply to Estimation Methods Using Iterative Search for Minimizing a Criterion

The properties that governs the iterative search are:

- Trace
- LimitError
- MaxIter
- Tolerance
- SearchDirection
- Advanced

These properties apply to `armax`, `bj`, `oe`, and `pem`

`Trace`: This property determines the information about the iterative search that is provided to the MATLAB command window.

- *Trace = Off:* No information is written to the screen

- *Trace = On:* Information about criterion values and the search process is given for each iteration.
- *Trace= Full*: The current parameter values and the search direction are also given (except in the "free" `SSParameterization` case for `idss` models)

`LimitError`: This variable determines how the criterion is modified from quadratic to one that gives linear weight to large errors. Errors larger than `LimitError` times the estimated standard deviation will carry a linear weight in the criteria. The default value of `LimitError` is 1.6. `LimitError =0` disables the robustification and leads to a purely quadratic criterion. The standard deviation is estimated robustly as the median of the absolute deviations from the median, divided by 0.7. (See Eq. (15.9)-(15.10) in Ljung (1999).)

`MaxIter`: The maximum number of iterations performed during the search for minimum. The iterations will stop when `MaxIter` is reached, or some other stopping criterion is satisfied. The default value of `MaxIter` is 20. Setting `MaxIter=0` will return the result of the start-up procedure. The actual number of used iterations is given by the property `EstimationInfo.Iterations`.

`Tolerance`: Based on the Gauss-Newton vector computed at the current parameter value, an estimate is made of the expected improvement of the criterion at the next iteration. When this expected improvement is less than `Tolerance %`, the iterations are stopped. Default value: 0.01.

`SearchDirection`: The direction along which a line search is performed to find a lower value of the criterion function. It may assume the following values:

- *gn:* The Gauss-Newton direction (inverse of the Hessian times the gradient direction) If no improvement is found along this direction, the gradient direction is also tried out.
- *gns:* A regularized version of the Gauss-Newton direction. Eigenvalues less than `pinvtol` of the Hessian are neglected, and the Gauss-Newton direction is computed in the remaining subspace. (`pinvtol` is part of the 'advanced' field: See `Algorithm Properties` in the "Command Reference" Chapter.
- *lm:* The Levenberg-Marquard method is used. This means that the next parameter value is `-pinv(H+d*I)*grad` from the previous one, where `H` is the Hessian, I is the identity matrix, `grad` is the gradient. `d` is a number that is increased until a lower value of the criterion is found.

- *Auto*: A choice between the above is made in the algorithm. This is the default choice.

One property of the returned model is `EstimationInfo`. That is a structure that contains useful information about the estimation process. See `EstimationInfo` in the "Command Reference" chapter for a list of fields.

Another important option is `InitialState`. See "Initial State" on page 3-91.

For the spectral analysis estimate, you can compute the frequency functions at arbitrary frequencies. If the frequencies are specified in a row vector `w`, then

```
g = spa(z,M,w)
```

results in `g` being computed at these frequencies. You can generate logarithmically spaced frequencies using the MATLAB `logspace` function. For example

```
w = logspace(-3,pi,128)
```

Defining Model Structures

Since the System Identification Toolbox handles a wide variety of different model structures, it is important that these can be defined in a flexible way. In the previous section you saw how models are automatically produced in the right form by the various estimation routines `arx`, `iv4`, `oe`, `bj`, `armax`, and `pem`, if you just specify model orders and delays.

This section describes how model structures and models can be directly defined. This may be required, for example, when creating a model for simulation. Also, it many be necessary to define model structures that are not of black-box type, but contain more detailed internal structure, reflecting some physical insights into how the system works.

The general way of representing models and model structures in the System Identification Toolbox is by various model objects. This section introduces the commands (apart from the parametric estimation functions themselves) that create these different models.

The model objects will contain a number of properties. For any model you can type

```
get(m)
```

to see a list of the model's properties, and

```
set(m)
```

to see what the assignable values are. See `get` and/or `set` in the "Command Reference" chapter. Each property value can easily also be retrieved by subreferencing as in

```
m.A
```

and set as in

```
m.b(3) = 27
```

See the "Command Reference" chapter for complete property lists. Here only examples are given. Note that it is sufficient to use any case insensitive, unambiguous abbreviation of the property names.

Polynomial Black-Box Models: The idpoly Model

The general input-output form (3-19)

$$A(q)y(t) = \frac{B(q)}{F(q)}u(t-nk) + \frac{C(q)}{D(q)}e(t) \qquad \textbf{(3-43)}$$

is defined by the five polynomials $A(q)$, $B(q)$, $C(q)$, $D(q)$, and $F(q)$. These are represented in the standard MATLAB format for polynomials. Polynomial coefficients are stored as row vectors ordered by descending powers. For example, the polynomial

$$A(q) = 1 + a_1 q^{-1} + a_2 q^{-2} + \ldots + a_n q^{-n}$$

is represented as

```
A = [1 a1 a2 ...  an]
```

Delays in the system are indicated by leading zeros in the $B(q)$ polynomial. For example, the ARX model

$$y(t) - 1.5y(t-1) + 0.7y(t-2) = 2.5u(t-2) + 0.9u(t-3) \qquad \textbf{(3-44)}$$

is represented by the polynomials

```
A = [1 -1.5 0.7]
B = [0 0 2.5 0.9]
```

The `idpoly` representation of (3-43) is now created by the command

```
m = idpoly(A,B,C,D,F,lam,T)
```

`lam` is here the variance of the white noise source $e(t)$ and `T` is the sampling interval. Trailing arguments can be omitted for default values. The system (3-44) can for example be represented by

```
m = idpoly([1 -1.5 0.7],[0 0 2.5 0.9])
```

In the multi-input case (3-41) `B` and `F` are matrices, whose row number `k` corresponds to the `k`-th input. The command `idpoly` can also be used to define time-continuous systems. See Chapter 4, “Command Reference” for details.

When `m` is defined, the polynomials and their orders can be easily retrieved and changed, as in

```
m.a % for the A-polynomial
roots(m.a)
m.a(3)=0.95
```

Multivariable ARX Models: The idarx Model

A multivariable ARX model with *nu* inputs and *ny* outputs is given by

$$A(q)y(t) = B(q)u(t) + e(t) \quad \textbf{(3-45)}$$

Here *A(q)* is an *ny*-by-*ny* matrix whose entries are polynomials in the delay operator q^{-1}. You can represent it as

$$A(q) = I_{ny} + A_1 q^{-1} + \ldots + A_{na} q^{-na} \quad \textbf{(3-46)}$$

as well as the matrix

$$A(q) = \begin{bmatrix} a_{11}(q) & a_{12}(q) & \ldots & a_{1ny}(q) \\ a_{21}(q) & a_{22}(q) & \ldots & a_{2ny}(q) \\ \ldots & \ldots & \ldots & \ldots \\ a_{ny1}(q) & a_{ny2}(q) & \ldots & a_{nyny}(q) \end{bmatrix} \quad \textbf{(3-47)}$$

where the entries a_{kj} are polynomials in the delay operator q^{-1}:

$$a_{kj}(q) = \delta_{kj} + a_{kj}^1 q^{-1} + \ldots + a_{kj}^{na_{kj}} q^{-na_{kj}} \quad \textbf{(3-48)}$$

This polynomial describes how old values of output number *j* affect output number *k*. Here δ_{kj} is the Kronecker-delta; it equals 1 when $k = j$, otherwise, it is 0. Similarly, $B(q)$ is an *ny*-by-*nu* matrix

$$B(q) = B_0 + B_1 q^{-1} + \ldots B_{nb} q^{-nb} \quad \textbf{(3-49)}$$

or

$$B(q) = \begin{bmatrix} b_{11}(q) & b_{12}(q) & \dots & b_{1nu}(q) \\ b_{21}(q) & b_{22}(q) & \dots & b_{2nu}(q) \\ \dots & \dots & \dots & \dots \\ b_{nu1}(q) & b_{nu2}(q) & \dots & b_{nunu}(q) \end{bmatrix} \quad \textbf{(3-50)}$$

with

$$b_{kj}(q) = b_{kj}^{1} q^{-nk_{kj}} + \dots + b_{kj}^{nb_{kj}} q^{-nk_{kj}-nb_{kj}+1}$$

The delay from input number j to output number k is nk_{kj}. To link with the structure definition in terms of `na`, `nb`, `nk` in the `arx` and `iv4` commands, note that `na` is a matrix whose kj-element is na_{kj}, while the kj-elements of `nb` and `nk` are nb_{kj} and nk_{kj} respectively.

The `idarx` representation of the model (3-45) can be created by

```
m = idarx(A,B)
```

where `A` and `B` are 3-D arrays of dimensions `ny-by-ny-by-(na+1)` and `ny-by-nu-by-(nb+1)`, respectively, that define the matrix polynomials (3-46) and (3-49):

```
A(:,:,k+1) = Ak
B(:,:,k+1) = Bk
```

Note that `A(:,:,1)` is always the identity matrix, and that the leading coefficients in `B` matrices define the delays.

Consider the following system with two outputs and three inputs:

$$y_1(t) - 1.5y_1(t-1) + 0.4y_2(t-1) + 0.7y_1(t-2) =$$
$$0.2u_1(t-4) + 0.3u_1(t-5) + 0.4u_2(t) - 0.1u_2(t-1) + 0.15u_2(t-2) + e_1(t)$$
$$y_2(t) - 0.2y_1(t-1) - 0.7y_2(t-2) + 0.01y_1(t-2) =$$
$$u_1(t) + 2u_2(t-4) + 3u_3(t-1) + 4u_3(t-2) + e_2(t)$$

which in matrix notation can be written as

$$y(t) + \begin{bmatrix} -1.5 & 0.4 \\ -0.2 & 0 \end{bmatrix} y(t-1) + \begin{bmatrix} 0.7 & 0 \\ 0.01 & -0.7 \end{bmatrix} y(t-2) = \begin{bmatrix} 0 & 0.4 & 0 \\ 1 & 0 & 0 \end{bmatrix} u(t) +$$

$$\begin{bmatrix} 0 & -0.1 & 0 \\ 0 & 0 & 3 \end{bmatrix} u(t-1) + \begin{bmatrix} 0 & 0.15 & 0 \\ 0 & 0 & 4 \end{bmatrix} u(t-2) + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} u(t-3) +$$

$$\begin{bmatrix} 0.2 & 0 & 0 \\ 0 & 2 & 0 \end{bmatrix} u(t-4) + \begin{bmatrix} 0.3 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} u(t-5)$$

This system is defined and simulated for a certain input u, and then estimated in the correct ARX structure by the following string of commands:

```
A(:,:,1) = eye(2);
A(:,:,2) = [-1.5 0.4; -0.2 0];
A(:,:,3) = [0.7 0 ; 0.01 -0.7];
B(:,:,1) = [0 0.4 0;1 0 0];
B(:,:,2) = [0 -0.1 0;0 0 3];
B(:,:,3) = [0 0.15 0;0 0 4];
B(:,:,4) = [0 0 0;0 0 0];
B(:,:,5) = [0.2 0 0;0 2 0];
B(:,:,6) = [0.3 0 0;0 0 0];
m0 = idarx(A,B);
u = iddata([], idinput([200,3]));
e = iddata([], randn(200,2));
y = sim(m0, [u e]);
na = [2 1;2 2];
nb = [2 3 0;1 1 2];
nk = [4 0 0;0 4 1];
me = arx([y u],[na nb nk])
me.a % The estimated A-polynominal
```

Black-Box State-Space Models: the idss Model

The basic state-space models are the following ones: (See also “State-Space Models” on page 3-28.).

Discrete-Time Innovations Form

$$\begin{aligned} x(kT+T) &= Ax(kT) + Bu(kT) + Ke(kT) && (a) \\ y(kT) &= Cx(kT) + Du(kT) + e(kT) && (b) \\ x(0) &= x0 && (c) \end{aligned} \qquad \textbf{(3-51)}$$

Here T is the sampling interval, $u(kT)$ is the input at time instant kT, and $y(kT)$ is the output at time kT. (See Ljung (1999) page 99.)

System Dynamics Expressed in Continuous Time

$$\begin{aligned} \dot{x}(t) &= Fx(t) + Gu(t) + \tilde{K}w(t) \\ y(t) &= Hx(t) + Du(t) + w(t) \\ x(0) &= x0 \end{aligned} \qquad \textbf{(3-52)}$$

(See Ljung (1999) page 93.) It is often easier to define a parameterized state-space model in continuous time because physical laws are most often described in terms of differential equations. The matrices F, G, H, and D contain elements with physical significance (for example, material constants). The numerical values of these may or may not be known. To estimate unknown parameters based on sampled data (assuming T is the sampling interval) first transform (3-52) to (3-51) using the formulas (3-27). The value of the Kalman gain matrix K in (3-51) or $\tilde{K}$ in (3-52) depends on the assumed character of the additive noises $w(t)$ and $e(t)$ in (3-25), and its continuous-time counterpart. Disregard that link and view K in (3-51) (or $\tilde{K}$ in (3-52)) as the basic tool to model the disturbance properties. This gives the *directly parametrized innovations form*. (See Ljung (1999) page 99.) If the internal noise structure is important, you could use user-defined greybox structures (the `idgrey` object) as in the example on page 3-47.

The discrete time model (3-51) can be put into the `idss` model by

```
m = idss(A,B,C,D,K,X0,'Ts',T)
```

and for the continuous-time model (3-52) use

```
m = idss(F,G,H,D,Kt,X0,'Ts',0)
```

Setting the sampling interval `Ts` to zero means a continuous-time model. The model `m` can now be used for simulation and it can be examined by the various commands. The parameterization of the matrices is by default “free” that is,

any elements in the matrices are freely adjustable by the estimation routines. The parameters will be adjusted to data by

```
me = pem(Data,m)
```

The iterative search for the best fit is then initialized in the nominal matrices `A`, `B`, **`C`**, `D`, `K`, `X0`. Note that the command `me = pem(Data,4)`, which just defines the model order, first estimates (using `n4sid`) a starting model `m`, from which the search is initialized.

In this free parameterization, you can decide how to deal with the disturbance model matrix *K*. Letting

```
m.DisturbanceModel = 'None'
```

(rather than `'Estimate'`) fixes the *K*-matrix to zero, thereby creating an Output-Error model.

Letting

```
m.InitialState ='zero'
```

(rather than `'Estimate'`) sets the initial state vector `x0` to zero.

The property `nk` determines the delays from the different inputs just as for `idpoly` models. Thus

```
m.nk = [0,0,...,0]
```

(no delays) means that all elements of the *D*-matrix should be estimated, while

```
m.nk = [1,1,..,1]
```

fixes the *D*-matrix to zero.

With the parameterization of `A`, `B`, and `C` being completely free, a basis for the state-space realization is automatically selected to give well-conditioned calculations. An alternative is to specify an observer canonical form for `A`, `B`, `C` by

```
m.sspar = 'Canonical'
```

(rather than `'Free'`). This is still a black-box model, since the canonical form covers all models of a certain order. The structure modifications can all be combined at the estimation call

```
me = pem(Data,m,'sspar','can','dist','none','ini','z')
```

which is the same as

```
set(m,'sspar','can','dist','none','ini','z')
me = pem(Data,m);
```

Structured State-Space Models with Free Parameters: the idss Model

The System Identification Toolbox allows you to define arbitrary parameterizations of the matrices in (3-51) or (3-52). To define the structure, so called **structure matrices** are used. These are "shadow matrices" to A, B, C, D, K, and X0, and have the same sizes and coincide with these at all matrix elements that are known. The structure matrices are denoted by `As`, `Bs`, `Cs`, `Ds`, `Ks`, and `X0s` and have the entry `NaN` at those elements that correspond to unknown parameters to be estimated.

For example

```
m.As = [NaN 0;0 NaN]
```

sets the structure matrix for `A`, called `As`, to a diagonal matrix, where the diagonal elements are freely adjustable. Defining

```
m.A = [2 0; 0 3]
```

sets the nominal/initial values of these diagonal elements to 2 and 3, respectively.

Example 3.1: A Discrete-Time Structure. Consider the discrete-time model

$$x(t+1) = \begin{bmatrix} 1 & \theta_1 \\ 0 & 1 \end{bmatrix} x(t) + \begin{bmatrix} \theta_2 \\ \theta_3 \end{bmatrix} u(t) + \begin{bmatrix} \theta_4 \\ \theta_5 \end{bmatrix} e(t)$$

$$y(t) = \begin{bmatrix} 1 & 0 \end{bmatrix} x(t) + e(t)$$

$$x(0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

with five unknown parameters θ_i, i=1,...,5. Suppose the nominal/initial values of these parameters are -1, 2, 3, 4 and 5. This structure is then defined by

```
m = idss([1, -1;0, 1],[2;3],[1,0],0,[4;5])
m.As = [1, NaN; 0 ,1];
m.Bs = [NaN;NaN];
m.Cs = [1, 0];
m.Ds = 0;
m.Ks = [NaN;NaN];
m.x0s = [0;0];
```

The definition thus follows in two steps. First the nominal model is defined. Then the structure (known and unknown parameter values) is defined by the structure matrices, As, Bs, etc.

Example 3.2: A Continuous-Time Model Structure. Consider the following model structure

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 \\ 0 & \theta_1 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \theta_2 \end{bmatrix} u(t)$$

$$y(t) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(t) + e(t)$$

$$x(0) = \begin{bmatrix} \theta_3 \\ 0 \end{bmatrix}$$

This corresponds to an electrical motor, where $y_1(t) = x_1(t)$ is the angular position of the motor shaft and $y_2(t) = x_2(t)$ is the angular velocity. The parameter $-\theta_1$ is the inverse time constant of the motor and $-\theta_2/\theta_1$ is the static gain from the input to the angular velocity. (See page Example 4.1 in Ljung (1999).) The motor is at rest at time 0 but at an unknown angular position. Suppose that θ_1 is around -1 and θ_2 is around 0.25. If you also know that the variance of the errors in the position measurement is 0.01 and in the angular velocity measurements is 0.1, you can then define an idss model using

```
m = idss([0 1;0
-1],[0;0.25],eye(2),[0;0],zeros(2,2),[0;0],'Ts',0)
m.as = [0 1; 0 NaN]
m.bs = [0 ;NaN]
m.cs = m.c
m.ds = m.d
m.ks = m.k
```

```
m.x0s = [NaN;0]
m.noisevar = [0.01 0; 0 0.1]
```

The structure `m` can now be used to estimate the unknown parameters θ_i from observed data

```
Data = iddata([y1 y2], u, 0.1)
```

by

```
model = pem(Data,m)
```

The iterative search for minimum is then initialized at the parameters in the nominal model `m`. The continuous time model is automatically sampled to agree with the sampling interval of the data. The structure can also be used to simulate the system above with sampling interval `T=0.1` for input `u` and noise realization `e`:

```
e = randn(300,2)
u = idinput(300);
simdat = iddata([],[u e],'Ts',0.1);
y = sim(m,simdat) % The continuous system will automatically be
                  % sampled using Ts =0.1
```

The nominal parameter values are then used, and the noise sequence is scaled according to the matrix `m.noisevar`.

When estimating models, you can try a number of "neighboring" structures, such as "what happens if I fix this parameter to a certain value" or "what happens if I let loose these parameters." This is easily handled by the structure matrices `As`, `Bs`, etc. For example, to free the parameter $x_2(0)$ (perhaps the motor wasn't at rest after all), you can use

```
model = pem(Data,m,'x0s',[NaN;NaN])
```

To manipulate initial conditions, the function `init` is also useful.

State-Space Models with Coupled Parameters: the idgrey Model

In some situations you may want the unknown parameters in the matrices in (3-51) or (3-52) to be linked to each other. Then the `NaN` feature is not sufficient to describe these links. Instead you need to do some "greybox"-modelling and write an M-file that describes the structure. The format is

```
[A,B,C,D,K,x0] = mymfile(par,T,aux);
```

where `mymfile` is the user-defined name for the M-file, `par` contains the parameters as a column vector, `T` is the sampling interval, and `aux` contains auxiliary variables. The latter variables are used to house options, so that some different cases can be tried out without having to edit the M-file. The matrices `A`, `B`, `C`, `D`, `K`, and `x0` refer either to the continuous time description (3-52) or to the discrete-time description (3-51). When a continuous time description is fitted to sampled data, the estimation routines perform the necessary sampling of the model. To obtain the same structure as in the Example 3.2, you can do the following:

```
function [A,B,C,D,K,x0] = mymfile(par,T,aux)
A = [0 1; 0 par(1)];
B = [0;par(2)];
C = eye(2);
D = zeros(2,2);
K = zeros(2,1);
x0 =[par(3);0];
```

Once the M-file has been written, the idgrey model `m` is defined by the command

```
m = idgrey('mymfile',par,'c',aux);
```

where `par` contains the nominal (initial) values of the corresponding entries in the structure. `'c'` signals that the underlying parametrization is continuous time. `aux` contains the values of the auxiliary parameters. Note that `T` and `aux` must be given as input arguments, even if they are not used by the code.

From here on, estimate models and evaluate results as for any other model structure. Some further examples of user-defined model structures are given below.

Some Examples of idgrey Model Structures

With user-defined structures, you have complete freedom in the choice of models of linear systems. This section gives two examples of such structures.

Example 3.3: Heat Diffusion. Consider a system driven by the heat-diffusion equation (see also Example 4.3 in Ljung (1999)).

This is a metal rod with a heat-diffusion coefficient κ, which is insulated at the near end and heated by the power u (W) at the far end. The output of the

system is the temperature at the near end. This system is described by a partial differential equation in time and space. Replacing the space-second derivative by a corresponding difference approximation gives a time-continuous state-space model (3-52), where the dimension of x depends on the grid-size in space used in the approximation. It is also desirable to be able to work with different grid sizes without having to edit the model file. This is described by the following M-file:

```
function [A,B,C,D,K,x0] = heatd(pars,T,aux)
Ngrid = aux(1); % Number of points in the space-discretization
L = aux(2); % Length of the rod
temp = aux(3); % Assuming uniform initial temperature of the rod
deltaL = L/Ngrid;    % Space interval
kappa = pars(1); % The heat-diffusion coefficient
htf = pars(2); % Heat transfer coefficient at far end of rod
A = zeros(Ngrid,Ngrid);
for kk = 2:Ngrid-1
A(kk,kk-1) = 1;
A(kk,kk) = -2;
A(kk,kk+1) = 1;
end
A(1,1) = -1; A(1,2) = 1; % Near end of rod insulated
A(Ngrid,Ngrid-1) = 1;
A(Ngrid,Ngrid) = -1;
A = A*kappa/deltaL/deltaL;
B = zeros(Ngrid,1);
B(Ngrid,1) = htf/deltaL;
C = zeros(1,Ngrid);
C(1,1) = 1;
D = 0;
K = zeros(Ngrid,1);
x0 = temp*ones(Ngrid,1);
```

You can then define the model by

```
m = idgrey('heatd',[0.27 1],'c',[10,1,22])
```

for a 10th order approximation of a heat rod one meter in length, with an initial temperature of 22 degrees. The initial estimate of the heat conductivity is 0.27, and of the heat transfer coefficient is 1.

The model parameters are estimated by

```
me = pem(Data,m)
```

If you would like to try a finer grid, that is, take `Ngrid` larger, you can do this easily with

```
me = pem(Data,m,'Filearg',[20,1,22])
```

Example 3.4: Parametrized Disturbance Models. Consider a discrete-time model

$$x(t+1) = Ax(t) + Bu(t) + w(t)$$
$$y(t) = Cx(t) + e(t)$$

where w and e are independent white noises with covariance matrices $R1$ and $R2$, respectively. Suppose that you know the variance of the measurement noise $R2$, and that only the first component of $w(t)$ is nonzero. This can be handled by the following M-file:

```
function [A,B,C,D,K,x0] = mynoise(par,T,aux)
R2 = aux(1); % The assumed known measurement noise variance
A = [par(1) par(2);1 0];
B = [1;0];
C = [par(3) par(4)];
D = 0;
R1 = [par(5) 0;0 0];
K = A*dlqe(A,eye(2),C,R1,R2); % from the Control System Toolbox
x0 = [0;0];
```

State-Space Structures: Initial Values and Numerical Derivatives

For a structured state-space model it is sometimes difficult to find good initial parameter values at which to start the numerical search for a minimum of (3-38). It is always best to use physical insight, whenever possible, to suggest such values. For random initialization, the command `init` is useful. Since there is always a risk that the numerical minimization may get stuck in a local minimum, it is advisable to try several different initialization values for θ.

In the search for the minimum, the gradient of the prediction errors $e(t)$ is computed by numerical differentiation. The step-size is determined by the M-file `nuderst`. In its default version the step-size is simply 10^{-4} times the absolute value of the parameter in question (or the number 10^{-7} if this is larger). When the model structure contains parameters with different orders of

magnitude, try to scale the variables so that the parameters are all roughly the same magnitude. You may need to edit the M-file `nuderst` to address the problem of suitable step sizes for numerical differentiation.

Examining Models

Once you have estimated a model, you need to investigate its properties. You have to simulate it, test its predictions, and compute its poles and zeros and so on. You thus have to transform the model to various ways of representing and presenting it. This section deals with how this is done. The following topics will be covered

- Parametric models: basic use, accessing properties, simulation and prediction. Also manipulating channels, in particular the noise input channels
- Frequency domain models
- Graphing model properties
- Transformations to other representations
- Transformations between continuous and discrete time

Parametric Models: idmodel and its children

`idmodel` is an object that the user does not deal with directly. It contains all the common properties of the model objects `idarx`, `idgrey`, `idpoly`, and `idss`, which are returned by the different estimation routines.

Basic Use

If you just estimate models from data, the model objects should be transparent. All parametric estimation routines return `idmodel` results:

```
m = arx(Data,[2 2 1])
```

The model `m` contains all relevant information. Just typing `m` will give a brief account of the model. `present(m)` also gives information about the uncertainties of the estimated parameters. `get(m)` gives a complete list of model properties.

Most of the interesting properties can be directly accessed by subreferencing:

```
m.a
m.da
```

See the property list obtained by `get(m)`, as well as the property lists of `idgrey`, `idarx`, `idpoly`, and `idss` in the “Command Reference” for more details on this.

The characteristics of the model `m` can be directly examined and displayed by commands like `impulse`, `step`, `bode`, `nyquist`, `pzmap`. The quality of the model is assessed by commands like `compare`, and `resid`. If you have the Control System Toolbox, just typing `view(m)` gives access to various display functions. More details about this will be given below.

To extract state-space matrices, transfer function polynomials, etc., there are the commands

```
arxdata, polydata, tfdata, ssdata, zpkdata
```

and by `idfrd` and `freqresp`, the frequency response of the model can be computed.

Simulation and Prediction

Any `idmodel` `m` can be simulated with

```
y = sim(m,Data)
```

where `Data` is an `iddata` object with just input channels.

```
Data = iddata([ ],[u v])
```

The number of input channels must either be equal to the number of measured channels in `m`, in which case a noise free simulation is obtained, or equal to the sum of the number of input and output channels in `m`. In the latter case the last input signals (v) are interpreted as white noise. They are then scaled by the `NoiseVariance` matrix of `m` and added to the output via the disturbance model

$$y = Gu + He$$

$$e = Lv$$

where the matrix L is given from the noise covariance Λ by $\Lambda = LL^T$:

```
L=chol(m.NoiseVariance)'
```

The output is returned as an `iddata` object with just output channels. Here is a typical string of commands:

```
A = [1 -1.5 0.7];
B = [0 1 0.5];
m0 = idpoly(A,B,[1 -1 0.2]);
u = iddata([],idinput(400,'rbs',[0 0.3]));
v= iddata([],randn(400,1));
```

```
y = sim(m0, [u v]);
plot(y)
```

The "inverse model" (3-38), which computes the prediction errors from given input-output data, is simulated with

```
e = pe(m,[y u])
```

To compute the k-step ahead prediction of the output signal based on a model `m`, the procedure is as follows:

```
yhat = predict(m,[y u],k
```

The predicted value $\hat{y}(t|t-k)$ is computed using the information in $u(s)$ up to time $s = t$ and information in $y(s)$ up to time $s = t-k$. The actual way that the information in past outputs is used depends on the disturbance model in `m`. For example, an output error model (that is, $H = 1$ in (3-10) maintains that there is no information in past outputs, therefore, predictions and simulations coincide.

`predict` can evaluate how well a time-series model is capable of predicting future values of the data. Here is an example, where `y` is the original series of, say, monthly sales figures. A model is estimated based on the first half, and then its ability to predict half a year ahead is checked out on the second half of the observations:

```
plot(y)
y1 = y(1:48), y2 = y(49:96)
m4 = ar(y1,4)
yhat = predict(m4,y2,6)
plot(y2,yhat)
```

The command `compare` is useful for any comparisons involving `sim` and `predict`.

Dealing with Input and Output Channels

For multivariable models, you construct submodels containing a subset of inputs and outputs by simple subreferencing. The outputs and input channels can be referenced according to

```
m(outputs,inputs)
```

Use colon (:) to denote all channels and the empty matrix ([ ]) to denote no channels. The channels can be referenced by number or by name. For several names, a cell array must be used.

```
m3 = m('position',{'power','speed'})
```

or

```
m3 = m(3,[1 4])
```

Thus `m3` is the model obtained from `m` by considering the transfer functions from input numbers 1 and 4 (with input names `'power'` and `'speed'`) to output number 3 (with name `'position'`)

For a single output model `m`

```
m4 = m(inputs)
```

will select the corresponding input channels, and for a single input model

```
m5 = m(outputs)
```

will select the indicated output channels.

Subreferencing is quite useful, e.g., when a plot of just some channels is desired.

The Noise Channels

The estimated models have two kinds of input channels: the measured inputs u and the noise inputs e. For a general linear model `m`, we have

$$y(t) = G(q)u(t) + H(q)e(t) \qquad \textbf{(3-53)}$$

where u is the nu-dimensional vector of measured input channels and `e` is the ny-dimensional vector of noise channels. The covariance matrix of `e` is given by the property `'NoiseVariance'`. Occasionally this matrix Λ will be written in factored form

$$\Lambda = LL^T$$

This means that `e` can be written as

$$e = Lv$$

where v is white noise with identity covariance matrix (independent noise sources with unit variances).

If `m` is a time series ($nu = 0$), G is empty and the model is given by

$$y(t) = H(q)e(t) \qquad \textbf{(3-54)}$$

For the model `m` in (3-53), the restriction to the transfer function matrix G is obtained by

```
m1 = m('measured')
```
or just
```
m1 = m('m')
```

Then e is set to `0` and H is removed.

Analogously

```
m2 = m('noise')
```
or just
```
m2 = m('n')
```

creates a time-series model `m2` from `m` by ignoring the measured input. That is `m2` is given by (3-54).

For a system with measured inputs, `bode`, `step`, and many other transformation and display functions just deal with the transfer function matrix G. To obtain or graph the properties of the disturbance model H, it is therefore important to make the transformations `m('n')`. For example,

```
bode(m('n'))
```

will plot the additive noise spectra according to the model `m`, while

```
bode(m)
```

just plots the frequency responses of G.

To study the noise contributions in more detail, it may be useful to convert the noise channels to measured channels, using the command `noisecnv`:

```
m3 = noisecnv(m)
```

This creates a model `m3` with all input channels, both measured `u` and noise sources `e`, being treated as measured signals,. That is, `m3` is a model from `u` and `e` to `y`, describing the transfer functions G and H. The information about the variance of the innovations `e` is then lost. For example, studying the step response from the noise channels, will then not take into consideration how large the noise contributions actually are.

To include that information, e should first be normalized $e = Lv$, so that v becomes white noise with an identity covariance matrix:

```
m4 = noisecnv(m,'Norm')
```

This will create a model `m4` with u and v treated as measured signals:

$$y(t) = G(q)u(t) + H(q)Lv(t) = \begin{bmatrix} G & HL \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

For example, the step responses from v to y will now also reflect the typical size of the disturbance influence, due to the scaling by L. In both these cases, the previous noise sources, that have become regular inputs will automatically get input names that are related to the corresponding output. The unnormalized noise sources `e` have names like `'e@y1'` (noise `e` at output channel with name `y1`), while the normalized sources `v` are called `'v@y1'`.

Retrieving Transfer Functions

The functions that retrieve transfer function properties, `ssdata`, `tfdata`, and `zpkdata` will thus work as follows for a model (3-53) with measured inputs: (*fcn* is any of `ssdata`, `tfdata`, or `zpkdata`)

fcn`(m)` returns the properties of G (ny outputs and nu inputs)

fcn`(m('n'))` returns the properties of the transfer function H (ny outputs and ny inputs)

fcn`(noisecnv(m))` returns the properties of the transfer function $[G\ H]$ (ny outputs and $ny+nu$ inputs).

fcn`(noisecnv(m,'Norm'))` returns the properties of the transfer function $[G\ HL\}$ (ny outputs and $ny+nu$ inputs. Analogously

```
fcn(noisecnv(m('n'),'Norm'))
```

returns the properties of the transfer function HL. (ny outputs and ny inputs).

If `m` is a time series model, *fcn*`(m)` returns the properties of H, while

```
fcn(noisecnv(m,'Norm'))
```

returns the properties of HL.

Note that the estimated covariance matrix `NoiseVariance` itself is uncertain. This means that the uncertainty information about *H* is different from that of *HL*.

idmodel Properties

See the "Command Reference" chapter for a complete list of `idmodel` properties.

Adding Channels

```
m = [m1,m2,...,mN]
```

creates an idmodel object `m`, consisting of all the input channels in `m1, ... mN`. The output channels of `mk` must be the same. Analogously

```
m = [m1;m2;... ;mN]
```

creates an `idmodel` object `m` consisting of all the output channels in `m1`, `m2`,..., `mN`. The input channels of `mk` must all be the same.

If you have the Control System Toolbox, interconnections between idmodels, like `G1+G2, G1*G2, append(G1,G2), feedback(G1,G2)`, etc, can be performed just as for LTI-objects. However, covariance information is typically lost.

Frequency Function Format: the idfrd model

Frequency functions and spectra are stored as an `idfrd (I`dentified Frequency Response Data) model object (which is not a child of `idmodel`). This model format is used by `spa` and `etfe` to deliver their results. Moreover, any `idmodel` can be transformed to an `idfrd` object.

The frequency function and the disturbance spectrum corresponding to an `idmodel m` is computed by

```
h = idfrd(m)
```

This gives *G* and $\hat{\Phi}_v$ in (3-11) along with their estimated covariances, which are translated from the covariance matrix of the estimated parameters. If `m` corresponds to a time-continuous model, the frequency functions are computed accordingly.

The functions are retrieved by

`h.ResponseData, h.CovarianceData, h.SpectrumData, and h.NoiseCovariance`

or any case-insensitive abbreviation of the names. The frequency vector is contained in `h.Frequency`.

In addition, an `idfrd` model can be defined directly from the frequency functions. See the "Command Reference" chapter, which also contains a list of `idfrd` properties. The channels of an `idfrd` model can be manipulated analogously to `idmodels`.

An alternative is to compute the response functions without storing them as an `idfrd` object:

```
[Response,Frequency,Covariance] = freqresp(m)
```

Graphs of Model Properties

There are several commands in the toolbox for graphing model characteristics like

- `bode`
- `compare`
- `ffplot`
- `impulse`
- `nyquist`
- `pzmap`
- `step`

They have all the same basic syntax. To look at one model use

```
command(Model)
```

where `command` is any of the functions listed above.

```
command(Mod1,Mod2, ...,ModN)
```

shows a comparison of several models. `Modk` can be any `idmodel` models. They can be used with any of the Control System Toolbox's LTI models. For some commands `Modk` can also be `idfrd` and `iddata` objects. For multivariable models, the plots are grouped so that each input/output channel (for all models) are plotted together. The `InputName` and `OutputName` properties of the models are used for this. The number of channels need not be the same in the different

models, which is quite useful when trying to find a good model of a multivariable system.

```
command(Mod1,PlotStyle1,...,ModN,PlotStyleN)
```

allows you to define colors, linestyles and markers associated with the different models. `PlotStyle` takes values such as `'b'` (for blue), `'b:'` (for a blue dotted line) or `'b*-'` (for a blue solid line with the points marked by a star). This is the same as for the usual plot command.

To also show the uncertainty of the model characteristics, use

```
command(Mod1,...,ModN,'sd',SD)
```

This will mark, using dash-dotted lines, a confidence region around the nominal model corresponding to `SD` standard deviations (for the Gaussian distribution). This region is computed using the estimated covariance matrix for the estimated parameters.

```
command(Mod1,...,ModN,'sd',SD,'fill')
```

shows the uncertainty region as a filled region instead.

The different commands have some further options to select time or frequency ranges and similar. See the “Command Reference” chapter.

If `Model` contains measured input channels, the plot shows just the transfer functions from these measured inputs to the outputs, that is *G* in (3-53). To graph the response from the noise sources, use

```
command(Model('n'))
```

For the frequency response graphs, this shows the additive disturbance spectra, i.e., the spectra of the signal *H(q)e(t)* in Equation (3-53), so that the properties of the noise source *e* are included in the plot.

For the other graphs, the properties of the transfer function *H* are shown, i.e., no noise normalization is done. The same is true if `Model` is a time series and has no measured input channels. That means that, for example, `step` shows the step response of the transfer function *H*, without accounting for the size (covariance matrix) of *e*. To include such effects, the disturbances should first be converted to normalized noise sources, using the command `noisecnv`. See “The Noise Channels” on page 3-52.

Model Output

An important and visually suggestive plot, is to compare the measured output signal with the models' simulated or predicted outputs. This is achieved by

```
compare(Data,model)
```

The input signal in `Data` is used by the model(s) to simulate the output. This simulated output is shown together with the measured output, which reveals what features in the data the model can and cannot reproduce. Also a legend shows the fit between the signals, in terms of how much of the output variation is reproduced by the model(s).

Frequency Response

Three functions offer graphic display of the frequency functions and spectra: `bode`, `ffplot`, and `nyquist`.

```
bode(G)
```

plots the Bode diagram of `G` (logarithmic scales and frequencies in rad/sec). If `G` is a spectrum, only an amplitude plot (the power spectrum) is given. Here `G` can be any `idmodel` or `idfrd` object.

The command `ffplot` has the same syntax as `bode` but works with linear frequency scales and Hertz as the unit. The command `nyquist` also has the same syntax, but produces Nyquist plots; i.e., graphs of the frequency function in the complex plane.

Transient Response

The impulse and step responses of the models are shown by

```
impulse(Model) and step(Model)
```

`impulse` and `step` follow the general syntax, but can also accept `iddata` objects as arguments. For direct estimation of step and impulse responses from data, the procedure described in "Estimating Impulse Responses" on page 3-15 is used.

Zeros and Poles

The zeros and poles are graphed by

```
pzmap(Model)
```

This gives a plot with 'x' marking poles and 'o' marking zeros. Otherwise, pzmap follows the general syntax.

General

If you have the Control System Toolbox

```
view(Model)
```

will open the LTI-viewer with access to a number of model displays. No uncertainty information can be shown, though.

Transformations to Other Model Representations

Within the structure in which the model was created, you can extract parametric information by get or by subscripting. For example, for a state-space model, Mod.A is the A-matrix, while Mod.dA contains its standard deviations. For a polynomial model, Mod.a and Mod.da are the A-polynomial and its standard deviation, etc.

In addition, regardless of the particular model structure, there are a number of commands that compute various model representations. These all have the basic syntax

```
[G, dG] = command(Model)
```

where G contains model characteristics and dG their standard deviation or covariance. The transformation commands are

```
[A,B,C,D,K,X0,dA,dB,dC,dD,dK,dX0] = ssdata(Model)

[a,b,c,d,f,da,db,dc,dd,df] = polydata(Model)

[A,B,dA,dB] =arxdata(Model)

[Num,Den,dNum,dDen] = tfdata(Model)

[Z,P,K,CovZ,CovP,covK] = zpkdata(Model)

G = idfrd(Model)

[H,w,CovH] = freqresp(Model)
```

The two last commands were described on page 3-55. The three first commands clearly transform to the state-space, the polynomial, and the multivariable

ARX representations. See “Defining Model Structures” on page 3-35. `tfdata` and `zpkdata` compute the transfer functions and zeros, poles and transfer function gains. See the reference pages in Chapter 4 for details.

Discrete and Continuous Time Models

Continuous-Time Models

Continuous-time models are created and recognized by the property `'Ts' = 0`. All `idmodel` objects can be created and analyzed as continuous-time models by setting `Ts` equal to zero at the time of creation, as in

```
m = idpoly(1,[0 1 1],1,1,[1 2 3],'Ts',0)
```

for the model

$$y = \frac{s+1}{s^2+2s+3}u + e$$

All model characteristics are then computed and graphed for the continuous-time representation. Time and frequency scales are determined using the sampling interval of the data, from which the model was estimated. For a nonestimated model, a default choice is made, which may make it necessary to supply frequency and time ranges to the commands.

For simulation and prediction, the continuous-time models are first converted to discrete time, using the sampling interval and intersample behavior of the data.

Estimating Continuous-Time Models

The estimation routines support the estimation of continuous-time state-space models in several different ways. The easiest is to use

```
mc = n4sid(Data,nx,'Ts',0)
```

This creates a continuous-time model in a free parameterization, based on the `n4sid` estimate. Further iterations from this estimate can be achieved by

```
mc = pem(Data,mc,'ss','can')
```

or directly by

```
mc = pem(Data,nx,'Ts',0,'ss','can')
```

The search for the continuous-time model must be carried out in a canonical (or any other structured) parameterization. The fit is still made to the sampled signals in `Data`. The model is sampled with the data's sampling interval for the fit. The information about the input intersample behavior in `Data`. `InterSample` is used to determine whether the sampling should be zero-order-hold (zoh, piecewise constant input) or first-order-hold (foh, piecewise linear input). All this gives black-box state-space models without any prescribed internal structure. In these cases, and for a zoh input, it may be easier to first estimate a black-box model in discrete time and then transform it to continuous time with `d2c` as described below. For a foh input it might be better to directly estimate the continuous-time model, since the mapping from discrete to continuous under a foh assumption is somewhat tricky.

The major reason for identifying continuous-time model is to secure a particular structure of the continuous-time state-space matrices. This would typically reflect a physical interpretation or some greybox modeling work done. This situation is handled by defining the structure as a continuous time `idss` or `idgrey` model, as described in"Black-Box State-Space Models: the idss Model" on page 3-39 and onwards. The resulting structure `mi` is fitted to data in the usual way.

```
m = pem(Data,mi)
```

Transformations

Transformations between continuous-time and discrete-time model representations are achieved by `c2d` and `d2c`. Note that it is not sufficient to just assign a new value of `Ts` to the model object. The corresponding uncertainty measure (the estimated covariance matrix of the internal parameters) is also transformed in most cases. The syntax is

```
modc = d2c(modd)
modd = c2d(mc,T)
```

If the discrete-time model has some pure time delays ($nk > 1$) the default command removes them before forming the continuous-time model, and appends them using the property `InputDelay` in model `modc`. This property is used to add appropriate phase lag and shift the data, whenever the model is used. `d2c` also offers as an option to approximate the dead time by a finite dimensional system. Note that the disturbance properties are translated by the somewhat questionable formula (3-29). The covariance matrix is translated by Gauss' approximation formula using numerical derivatives. The M-file

`nuderst` is then invoked. You may want to edit it for applications where the parameters have very different orders of magnitude. See the comments in "State-Space Structures: Initial Values and Numerical Derivatives" on page 3-47.

Here is an example that compares the Bode plots of an estimated model and its continuous-time counterpart.

```
m= armax(Data,[2 3 1 2]);
mc = d2c(m); bode(m,mc)
```

Model Structure Selection and Validation

After you have been analyzing data for some time, you typically end up with a large collection of models with different orders and structures. You need to decide which one is best, and if the best description is an adequate model for your purposes. These are the problems of *model validation*.

Model validation is the heart of the identification problem, but there is no absolute procedure for approaching it. It is wise to be equipped with a variety of different tools with which to evaluate model qualities. This section describes the techniques you can use to evaluate model qualities using the System Identification Toolbox.

Comparing Different Structures

It is natural to compare the results obtained from model structures with different orders. For state-space models this is easily obtained by using a vector argument for the order in `n4sid` or `pem`

```
m = n4sid(Data,1:10)
m = pem(Data,'nx',3:15)
```

This invokes a plot from which a "best" order is chosen. Just omitting the order argument, `m = n4sid(Data)` or `pem(Data)` makes a default choice of the best order.

For models of ARX type, various orders and delays can be efficiently studied with the command `arxstruc`. Collect in a matrix `NN` all of the ARX structures you want to investigate, so that each row of `NN` is of the type

```
[na nb nk]
```

With

```
V = arxstruc(Date,Datv,NN)
```

an ARX model is fitted to the data set `Date` for each of the structures in `NN`. Next, for each of these models, the sum of squared prediction errors is computed, as they are applied to the data set `Datv`. The resulting loss functions are stored in `V` together with the corresponding structures.

To select the structure that has the smallest loss function for the validation set `Datv`, use

```
nn = selstruc(V,0)
```

Such a procedure is known as *cross validation* and is a good way to approach the model selection problem.

It is usually a good idea to visually inspect how the fit changes with the number of estimated parameters. A graph of the fit versus the number of parameters is obtained with

```
selstruc(V)
```

This routine prompts you to choose the number of parameters to estimate, based upon visual inspection of the graph, and then it selects the structure with the best fit for that number of parameters.

The command `struc` helps generate typical structure matrices `NN` for single-input systems. A typical sequence of commands is

```
V = arxstruc(Date,Datv,struc(2,2,1:10));
nn = selstruc(V,0);
nk = nn(3);
V = arxstruc(Date,Datv,struc(1:5,1:5,nk-1:nk+1));
selstruc(V)
```

where you first establish a suitable value of the delay `nk` by testing second order models with delays between one and ten. The best fit selects the delay, and then all combinations of ARX models with up to five *a* and *b* parameters are tested with delays around the chosen value (a total of 75 models).

If the model is validated on the same data set from which it was estimated; i.e., if `Date = Datv`, the fit always improves as the flexibility of the model structure increases. You need to compensate for this automatic decrease of the loss functions. There are several approaches for this. Probably the best known technique is Akaike's Final Prediction Error (FPE) criterion and his closely related Information Theoretic Criterion (AIC). Both simulate the cross validation situation, where the model is tested on another data set.

The FPE is formed as

$$FPE = \frac{1 + \frac{d}{N}}{1 - \frac{d}{N}} V$$

where d is the total number of estimated parameters and N is the length of the data record. V is the loss function (quadratic fit) for the structure in question. The AIC is formed as

$$AIC = \log\left(V\left(1 + 2\frac{d}{N} \right) \right)$$

(See Section 16.4 in Ljung (1999).)

According to Akaike's theory, in a collection of different models, choose the one with the smallest FPE (or AIC). The FPE values are displayed with the model parameters, by just typing the model name. It is also one of the fields in `EstimationInfo`, and can be accessed by

```
FPE = fpe(m)
```

Similarly, the AIC value of an estimated model is obtained as

```
AIC = aic(m)
```

The structure that minimizes the AIC is obtained with

```
nn = selstruc(V,'AIC')
```

where `V` was generated by `arxstruc`.

A related criterion is Rissanen's Minimum Description Length (MDL) approach, which selects the structure that allows the shortest over-all description of the observed data. This is obtained with

```
nn = selstruc(V,'MDL')
```

If substantial noise is present, the ARX models may need to be of high order to describe simultaneously the noise characteristics and the system dynamics. (For ARX models the disturbance model $1/A(q)$ is directly coupled to the dynamics model $B(q)/A(q)$.)

Impulse Response to Determine Delays

The command `impulse` applied to a dataset

```
impulse(Data,'sd',3)
```

shows a nonparametric estimate of the impulse response. In the call above, also a confidence region around zero is shown, corresponding to three standard deviations (ca 99.9%). Any part of the impulse response that is outside this region is thus significant. The first sample after *t=0*, at which the impulse response estimate crosses the confidence band is thus a good estimate of the delay in the channel in question.

Significant impulse response estimates for negative time lags are indications of feedback in the data.

Checking Pole-Zero Cancellations

A near pole-zero cancellation in the dynamics model is an indication that the model order may be too high. To judge if a "near" cancellation is a real cancellation, take the uncertainties in the pole- and zero-locations into consideration

```
pzmap(mod,'sd',1)
```

where the `1` indicates how many standard-deviations wide the confidence interval is. If the confidence regions of a zero and a pole overlap, try lower model orders.

This check is especially useful when the models have been generated by `arx`. As mentioned on page 3-66, the orders can be pushed up because of the requirement that $1/A(q)$ describe the disturbance characteristics. Checking cancellations in $B(q)/A(q)$ then gives a good indication of which orders to chose from model structures like `armax`, `oe`, and `bj`.

Residual Analysis

The residuals associated with the data and a given model, as in (3-38), are ideally white and independent of the input for the model to correctly describe the system. The function

```
resid(Model,Data)
```

computes the residuals (prediction errors) *e* from the model when applied to `Data`, and performs whiteness and independence analyses. The auto correlation function of `e` and the cross-correlation function between `e` and `u` are computed and displayed for up to lag 25. Also displayed are 99% confidence intervals for these variables, assuming that `e` is indeed white and independent of `u`.

The rule is that if the correlation functions go significantly outside these confidence intervals, do not accept the corresponding model as a good description of the system. Some qualifications of this statement are necessary:

- Model structures like the OE structure (3-17) and methods like the IV method (3-41) focus on the dynamics *G* and less about the disturbance properties *H*. If you are interested primarily in *G*, focus on the independence of `e` and `u` rather than the whiteness of `e`.
- Correlation between `e` and `u` for negative lags, or current $e(t)$ affecting future $u(t)$, is an indication of output feedback. This is not a reason to reject the model. Correlation at negative lags is of interest, since certain methods do not work well when feedback is present in the input-output data, (see "Feedback in Data" on page 3-76), but concentrate on the positive lags in the cross-correlation plot for model validation purposes.
- When using the ARX model (3-14), the least squares procedure automatically makes the correlation between $e(t)$ and $u(t-k)$ zero for $k = nk, nk+1, \ldots nk+nb-1$, for the data used for the estimation.

The residuals `e` together with the input `u` are returned by

```
E= resid(Model,Data)
```

as an `iddata` object. As part of the validation process, you can graph the residuals using

```
plot(E)
```

for a simple visual inspection of irregularities and outliers. (See also "Outliers and Bad Data; Multi-Experiment Data" on page 3-74.)

Model Error Models

The residual call

```
E= resid(Model,Data)
```

returns the iddata object e which has the inputs in Data as inputs and the prediction errors (residuals) as outputs. Building models using e will thus reveal if there is any significant influence from u to e left in the data. Such models are called Model Error Models, and examining them is a good complement to traditional residual analysis:

```
E= resid(Model,Data)
impulse(E,'sd',3) % An alternative to residual analysis
bode(spa(E),'sd',3) % Shows the frequency ranges
                    % with significant model errors
m = arx(E,[0 10 0])
bode(m,'sd',3)
```

Note that the resid command has several options to display model error properties rather than correlation functions.

Noise-Free Simulations

To check whether a model is capable of reproducing the observed output when driven by the actual input, you can run a simulation:

```
u = Data(:,[],:) % Extracting the input from the data
yh = sim(Model,u)
y = Data(:,:,[]) % extracting the output from the data
plot(y,yh)
```

The same result is obtained by

```
compare(Data,Model)
```

It is a much tougher and revealing test to perform this simulation, as well as the residual tests, on a fresh data set Data that was not used for the estimation of the model Model. This is called *cross validation*.

Assessing the Model Uncertainty

The estimated model is always uncertain, due to disturbances in the observed data, and due to the lack of an absolutely correct model structure. The variability of the model that is due to the random disturbances in the output is estimated by most of the estimation procedures, and it can be displayed and illuminated in a number of ways. This variability answers the question of how different can the model be if the identification procedure is repeated, using the same model structure, but with a different data set that uses the same input

sequence. It does not account for systematic errors due to an inadequate choice of model structure. There is no guarantee that the “true system” lies in the confidence interval.

The uncertainty in the different model views is displayed if the argument 'sd' is included in the argument list

```
command(Model,'sd',sd)
```

as explained in “Graphs of Model Properties” on page 3-56.

The uncertainty in the time response is displayed by

```
simsd(Model,u)
```

Ten possible models are drawn from the asymptotic distribution of the model `Model`. The response of each of them to the input `u` is graphed on the same diagram.

The uncertainty of these responses concerns the external, input-output properties of the model. It reflects the effects of inadequate excitation and the presence of disturbances.

You can also directly get the standard deviation of the simulated result by

```
[ysim,ysimsd] = sim(Model,u)
```

The uncertainty in internal representations is manifested in the covariance matrix of the estimated parameters,

```
Model.CovarianceMatrix
```

which is used to give the standard deviations of all model characteristics. The parametric uncertainty is directly available as

`Model.da` for the standard deviations of `Model.a`

Note that state-space models, estimated in a free parameterization do not have well defined standard deviations of the matrix elements. The model still has stored the uncertainty of the input-output behavior, so other model representations and graphs will show the uncertainty. For a state-space model in a free parameterization, it is possible to first transform it to a canonical parameterization and then display the matrix parameter uncertainties:

```
Modelc = Model
Modelc.ss = 'canon'
Modelc.da
```

All routines for computing and displaying model characteristics also offer to calculate and show the uncertainties. See “Transformations to Other Model Representations” on page 3-59.

Large uncertainties in these representations are caused by excessively high model orders, inadequate excitation, or bad signal-to-noise ratios.

Comparing Different Models

It is a good idea to display the model properties in terms of quantities that have more physical meaning than the parameters themselves. Bode plots, pole-zero plots, and model simulations all give a sense of the properties of the system that have been picked up by the model.

If several models of different characters give very similar Bode plots in the frequency range of interest, you can be fairly confident that these must reflect features of the true, unknown system. You can then choose the simplest model among these.

A typical identification session includes estimation in several different structures, and comparisons of the model properties. Here is an example.

```
a1 = arx(Data,[1 2 1]);
g = spa(Data);
bode(g,a1)
bode(g('n'),a1('n'))% the output disturbance spectra
am2 = armax(Data,[2 2 2 1]);
bode(g,am2)
pzmap(a1,am2,'sd',3)
```

Selecting Model Structures for Multivariable Systems

A multivariable (MIMO) system is a system with several input and output channels. All model structures in the toolbox support models with one output and several inputs. Polynomial models, `idpoly`, do not handle multi-output models, however.

Model Structures

Multivariable system offer a potentially richer internal structure. The easiest approach, in the black-box situation, is to think just in terms of input delays and state-space model order. A recommended approach is to get an idea of input delays from the nonparametric impulse response estimate, and determine the vector `nk = [nk1,nk2,...,nkm]` where `nkj` is the minimal delay from input `j` to any of the output channels, and then try state-space models with several orders and with these delays:

```
impulse(Data,'sd',3)
Model = n4sid(Data(1:500),'nx',1:10,'nk',nk)
compare(Data(501:1000),Model)
```

The `compare` plot will reveal which output channels are easy and which are difficult to reproduce.

An alternative to find the delays is to first estimate a parametric model with delays `1`, and then examine the impulse responses of this model and determine the delays:

```
Model = pem(Data) % This uses 'best' model order
impulse(Model,'sd',3)
Model = pem(Data,'nx',1:10,'nk',nk)
```

(To test models with delay 0 in a similar way, use `Model = pem(Data,'best','nk',zeros(size(nk)))`. Significant responses at delay 0 must be examined with care, since they may be caused by feedback.)

Note that delays `nk` larger than 1 will be incorporated in the model structure, and thus increase the state-space model order from the nominal one with `sum(max(nk-1,zeros(size(nk))))`. An alternative is to use the property `'InputDelay'`. This leads to a model that has the same delays as for `'nk'`, but these are not explicitly shown in the model matrices, but stored as a property to be used when necessary. See `idmodel` properties in the “Command Reference” chapter.

If you have detailed knowledge about which orders and delays that are reasonable in the different input/output channels, you can use multivariable ARX models, in the `idarx` model format. This allows you to define the orders of the input and output lags, as well as the delays, independently for the different channels.

Black-box parameterizations of multi-variable systems require many parameters. Therefore, it may be important to incorporate any essential structure knowledge based on physical insight. This is typically done by continuous-time, taylor-made model parameterizations using structured `idss` models, or `idgrey` models. See "Structured State-Space Models with Free Parameters: the idss Model" on page 3-42 and "State-Space Models with Coupled Parameters: the idgrey Model" on page 3-44.

Channel Selection

A particular aspect of multivariable models regards the selection of channels. Models for subselections of input-output channels may be quite useful and informative. Generally speaking the models become "better" when more input channels are used, and "worse" when more output channels are used. The latter observation is due to the fact that such models have "more to explain."

If you build models with several outputs and find, using `compare`, a certain output channel to be difficult to reproduce, then try to build model of this channel alone. This will reveal if there are inherent difficulties with this output, or that it is just too difficult to handle it together with other outputs.

Analogously, if you see that, using, e.g., step or impulse, a certain input channel seems to have an insignificant influence on the outputs, then remove that channel, and examine if the corresponding model becomes any worse, e.g., in the `compare` plots.

A main feature of the toolbox's data and model objects is that it gives full support for the bookkeeping required for these channel subselections. Channels are selected by direct subreferencing, and the `InputName` and `OutputName` properties form the basis for a correct combination of channels. The subreferencing follows

```
Data(Samples,Outputs,Inputs)
Model(Outputs,Inputs)
```

and typical command sequences may be

```
Date = Data(1:500)
Datv = Data(501:1000)
m = pem(Date)
compare(Datv,m)
m1 = pem(Date(:,3,4))
compare(Datv,m,m1)
bode(m,m1)
compare(Datv,m(:,4),m1)
```

Dealing with Data

Extracting information from data is not an entirely straightforward task. In addition to the decisions required for model structure selection and validation, the data may need to be handled carefully. This section gives some advice on handling several common situations.

Offset Levels

When the data have been collected from a physical plant, they are typically measured in physical units. The levels in these raw input and output measurements may not match in any consistent way. This will force the models to waste some parameters correcting the levels.

Typically, linearized models are sought around some physical equilibrium. In such cases offsets are easily dealt with: subtract the mean levels from the input and output sequences before the estimation. It is best if the mean levels correspond to the physical equilibrium, but if such values are not known, use the sample means:

```
Data = detrend(Data);
```

Section 14.1 in Ljung (1999) discusses this in more detail. There are situations when it is not advisable to remove the sample means. It could for example be that the physical levels are built into the underlying model, or that integrations in the system must be handled with the right level of the input being integrated.

With the `detrend` command, you can also remove piece-wise linear trends.

Outliers and Bad Data; Multi-Experiment Data

Real data are also subject to possible bad disturbances; an unusually large disturbance, a temporary sensor or transmitter failure, etc. It is important that such outliers are not allowed to affect the models too strongly.

The robustification of the error criterion (described under `LimitError` on page 3-33) helps here, but it is always good practice to examine the residuals for unusually large values, and to go back and critically evaluate the original data responsible for the large values. If the raw data are obviously in error, they can be smoothed, and the estimation procedure repeated.

Often the data has portions with bad behavior. This may, e.g., be due to big disturbances or sensor failures over a period of time. It can also be that there are time periods where "nothing happens," the input is not exciting, etc. Then the best alternative is to break up the data into pieces of informative portions. By merging the pieces into a multiexperiment `iddata` object, they can still be used together to build models. Another situation when multiexperiment data is useful is when several different experiments have been performed on the same plant. The estimation routines take proper action to handle the different pieces. All estimation, simulation, and validation routines in the toolbox handle multi-experiment data in a transparent fashion. A typical string of commands could be

```
plot(Data)
Datam = merge(Data(1:340),Data(500:897), ...
              Data(1001:1200),Data(1550:2000))
m =pem(Datam{[1,2,4]}) % Portions 1,2 and 4 for estimation
compare(Datam{3},m) % Portion 3 for validation
```

Missing Data

In practice it is often the case that certain measurement samples are missing. The reason may be sensor failures or data acquisition failures. It may be that the data are directly reported as missing, or that plots reveal that some values are obviously in error. This may apply both to inputs and outputs. In these cases, replace the missing data by `NaN` when forming the signal matrices and the `iddata` object. The routine `misdata` can then be applied to reconstruct the missing data in a reasonable way:

```
dat = iddata(y,u,0.2) % y and/or u contain NaN for missing data.
dat1 = misdata(dat);
plot(dat,dat1) % Checking how the missing data
               % have been estimated in dat1
m = pem(dat1) % Model estimated using reconstructed missing data
```

See Section 14.2 in Ljung(1999) for a discussion on missing data.

Filtering Data: Focus

Depending upon the application, interest in the model can be focused on specific frequency bands. Filtering the data before the estimation, through filters that enhance these bands, improves the fit in the interesting regions.

This is accomplished in the System Identification Toolbox by the property `'Focus'`. For example, to enhance the fit in the frequency band between 0.02π and 0.1π, (assuming a unit sampling interval) execute

```
[B,A] = butter(5,[0.02 0.1])
m = pem(Data,3,'Foc',{B,A})
ma = arx(Data,[2 3 1],'Foc',{B,A})
```

This computes and uses a fifth order Butterworth bandpass filter with passband between the indicated frequencies. The data is filtered through this filter before fitting the transfer function from the measured inputs (G in Equation (3-53)) to the outputs. The disturbance model (H) is however estimated using the unfiltered data. Chapter 14 in Ljung (1999) discusses the role of filtering in more detail.

The command `butter` is from the Signal Processing Toolbox. If you do not have that toolbox, the filter can be computed using `idfilt` from the System Identification Toolbox:

```
[Df,Mfilt] = idfilt(Data,5,[0.02 0.1])
m = pem(Data,3,'Foc',Mfilt)
```

For a model that does not use a disturbance description (that is, $H = 1$ in (3-53), which corresponds to `K = 0` for state-space, and `na=nc=nd=0` for polynomial models), the `Focus` effect is the same as applying the routine to filtered data. That is,

```
m = pem(Data,3,'Foc',Mfilt','dist','none')
m = pem(Df,3,'dist','none')
```

give the same model.

The System Identification Toolbox contains other useful commands for related problems. For example, if you want to lower the sampling rate by a factor of 5, use

```
Dat5 = resample(Data,1,5);
```

Feedback in Data

If the system was operating in closed loop (feedback from the past outputs to the current input) when the data were collected, some care has to be exercised.

Basically, all the prediction error methods work equally well for closed-loop data. Note, however, that the Output-Error model (3-17) and the Box-Jenkins model (3-18) are normally capable of giving a correct description of the dynamics *G*, even if *H* (which equals 1 for the output error model) does not allow a correct description of the disturbance properties. This is no longer true for closed-loop data. You then need to model the disturbance properties more carefully. Another thing to be cautious about is that impulse response effects at delay 0 very well could be traced to the feedback mechanism and not to the system itself.

The spectral analysis method and the instrumental variable techniques (with default instruments) as well as `n4sid` may give unreliable results when applied to closed-loop data. These techniques should be avoided when feedback is present.

To detect if feedback is present, use the basic method of applying `impulse` to estimate the impulse response. Significant values of the impulse response at negative lags is a clear indication of feedback. When a parametric model has been estimated and the `resid` command is applied, a graph of the correlation between residuals and inputs is given. Significant correlation at negative lags again indicates output feedback in the generation of the input. Testing for feedback is, therefore, a natural part of model validation.

Delays

The selection of the delay `nk` in the model structure is a very important step in obtaining good identification results. You can get an idea about the delays in the system by the impulse response estimate from `impulse`.

Incorrect delays are also visible in parametric models. Underestimated delays (`nk` too small) show up as small values of leading b_k estimates, compared to their standard deviations. Overestimated delays (`nk` too large) are usually visible as a significant correlation between the residuals and the input at the lags corresponding to the missing b_k terms in the `resid` plot.

A good procedure is to start by using `arxstruc` to test all feasible delays together with a second-order model. Use the delay that gives the best fit for further modeling. When you have found an otherwise satisfactory structure, vary `nk` around the nominal value within the structure, and evaluate the results.

Recursive Parameter Estimation

In many cases it may be necessary to estimate a model on line at the same time as the input-output data is received. You may need the model to make some decision on line, as in adaptive control, adaptive filtering, or adaptive prediction. It may be necessary to investigate possible time variation in the system's (or signal's) properties during the collection of data. Terms like *recursive identification*, *adaptive parameter estimation*, *sequential estimation*, and *on-line algorithms* are used for such algorithms. Chapter 11 in Ljung (1999) deals with such algorithms in some detail.

The Basic Algorithm

A typical recursive identification algorithm is

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)(y(t) - \hat{y}(t)) \qquad \textbf{(3-55)}$$

Here $\hat{\theta}(t)$ is the parameter estimate at time t, and $y(t)$ is the observed output at time t. Moreover, $\hat{y}(t)$ is a prediction of the value $y(t)$ based on observations up to time $t-1$ and also based on the current model (and possibly also earlier ones) at time $t-1$. The gain $K(t)$ determines in what way the current prediction error $y(t) - \hat{y}(t)$ affects the update of the parameter estimate. It is typically chosen as

$$K(t) = Q(t)\psi(t) \qquad \textbf{(3-56)}$$

where $\psi(t)$ is (an approximation of) the gradient with respect to θ of $\hat{y}(t|\theta)$. The latter symbol is the prediction of $y(t)$ according the model described by θ. Note that model structures like AR and ARX that correspond to linear regressions can be written as

$$y(t) = \psi^T(t)\theta_0(t) + e(t) \qquad \textbf{(3-57)}$$

where the *regression vector* $\psi(t)$ contains old values of observed inputs and outputs, and $\theta_0(t)$ represents the true description of the system. Moreover, $e(t)$ is the noise source (the innovations). Compare with (3-14). The natural prediction is $\hat{y}(t) = \psi^T(t)\hat{\theta}(t-1)$ and its gradient with respect to θ becomes exactly $\psi(t)$.

For models that cannot be written as linear regressions, you cannot recursively compute the exact prediction and its gradient for the current estimate $\hat{\theta}(t-1)$. Then approximations $\hat{y}(t)$ and $\psi(t)$ must be used instead. Section 11.4 in Ljung (1999) describes suitable ways of computing such approximations for general model structures.

The matrix $Q(t)$ that affects both the adaptation gain and the direction in which the updates are made, can be chosen in several different ways. This is discussed in the following.

Choosing an Adaptation Mechanism and Gain

The most logical approach to the adaptation problem is to assume a certain model for how the "true" parameters θ_0 change. A typical choice is to describe these parameters as a random walk:

$$\theta_0(t) = \theta_0(t-1) + w(t) \tag{3-58}$$

Here $w(t)$ is assumed to be white Gaussian noise with covariance matrix

$$Ew(t)w^T(t) = R_1 \tag{3-59}$$

Suppose that the underlying description of the observations is a linear regression (3-57). An optimal choice of $Q(t)$ in (3-55)-(3-56) can then be computed from the Kalman filter, and the complete algorithm becomes

$$\begin{aligned}
\hat{\theta}(t) &= \hat{\theta}(t-1) + K(t)(y(t) - \hat{y}(t)) \\
\hat{y}(t) &= \psi^T(t)\hat{\theta}(t-1) \\
K(t) &= Q(t)\psi(t) \\
Q(t) &= \frac{P(t-1)}{R_2 + \psi(t)^T P(t-1)\psi(t)} \\
P(t) &= P(t-1) + R_1 - \frac{P(t-1)\psi(t)\psi(t)^T P(t-1)}{R_2 + \psi(t)^T P(t-1)\psi(t)}
\end{aligned} \tag{3-60}$$

Here R_2 is the variance of the innovations $e(t)$ in (3-57): $R_2 = Ee^2(t)$ (a scalar). The algorithm (3-60) will be called the **Kalman filter (KF) approach** to adaptation, with *drift matrix* R_1. See eq (11.66)-(11.67) in Ljung (1999). The algorithm is entirely specified by $R_1, R_2, P(0), \theta(0)$, and the sequence of data

$y(t), \psi(t)$, $t = 1, 2, \ldots$. Even though the algorithm was motivated for a linear regression model structure, it can also be applied in the general case where $\hat{y}(t)$ is computed in a different way from (3-60b).

Another approach is to discount old measurements exponentially, so that an observation that is τ samples old carries a weight that is λ^{τ} of the weight of the most recent observation. This means that the following function is minimized rather than (3-39):

$$\sum_{k=1}^{t} \lambda^{t-k} e^2(k) \qquad \textbf{(3-61)}$$

at time t. Here λ is a positive number (slightly) less than 1. The measurements that are older than $\tau = 1/(1-\lambda)$ carry a weight in the expression above that is less than about 0.3. Think of $\tau = 1/(1-\lambda)$ as the *memory horizon* of the approach. Typical values of λ are in the range 0.97– 0.995.

The criterion (3-61) can be minimized exactly in the linear regression case giving the algorithm (3-60abc) with the following choice of $Q(t)$:

$$\begin{aligned} Q(t) &= P(t) = \frac{P(t-1)}{\lambda + \psi(t)^T P(t-1)\psi(t)} \\ P(t) &= \frac{1}{\lambda}\left(P(t-1) - \frac{P(t-1)\psi(t)\psi(t)^T P(t-1)}{\lambda + \psi(t)^T P(t-1)\psi(t)} \right) \end{aligned} \qquad \textbf{(3-62)}$$

This algorithm will be called the **Forgetting Factor (FF) approach** to adaptation, with the *forgetting factor* λ. See eq (11.63) in Ljung (1999). The algorithm is also known as *recursive least squares* (RLS) in the linear regression case. Note that $\lambda = 1$ in this approach gives the same algorithm as $R_1 = 0, R_2 = 1$ in the Kalman filter approach.

A third approach is to allow the matrix $Q(t)$ to be a multiple of the identity matrix:

$$Q(t) = \gamma I \qquad \textbf{(3-63)}$$

It can also be normalized with respect to the size of ψ:

$$Q(t) = \frac{\gamma}{|\psi(t)|^2} I \qquad \textbf{(3-64)}$$

See eqs (11.45) and (11.46), respectively in Ljung (1999). These choices of $Q(t)$ move the updates of $\hat{\theta}$ in (3-55) in the negative gradient direction (with respect to θ) of the criterion (3-39). Therefore, (3-63) will be called the **Unnormalized Gradient (UG) approach** and (3-64) the **Normalized Gradient (NG) approach** to adaptation, with gain γ. The gradient methods are also known as *least mean squares* (LMS) in the linear regression case.

Available Algorithms

The System Identification Toolbox provides the following functions that implement all common recursive identification algorithms for model structures in the family (3-43): `rarmax`, `rarx`, `rbj`, `rpem`, `rplr`, and `roe`. They all share the following basic syntax:

```
[thm,yh] = rfcn(z,nn,adm,adg)
```

Here `z` contains the output-input data as usual. `nn` specifies the model structure, exactly as for the corresponding offline algorithm. The arguments `adm` and `adg` select the adaptation mechanism and adaptation gain listed above.

```
adm = 'ff'; adg = lam
```

gives the forgetting factor algorithm (3-62), with forgetting factor `lam`.

```
adm = 'ug'; adg = gam
```

gives the unnormalized gradient approach (3-63) with gain `gam`. Similarly,

```
adm = 'ng'; adg = gam
```

gives the normalized gradient approach (3-64). To obtain the Kalman filter approach (3-60) with drift matrix `R1`, enter

```
adm = 'kf'; adg = R1
```

The value of R_2 is always 1. Note that the estimates $\hat{\theta}$ in (3-60) are not affected if all the matrices R_1, R_2 and $P(0)$ are scaled by the same number. You can therefore always scale the original problem so that R_2 becomes 1.

The output argument `thm` is a matrix that contains the current models at the different samples. Row `k` of `thm` contains the model parameters, in alphabetical order at sample time k, corresponding to row `k` in the data matrix `z`. The ordering of the parameters is the same as `m.par` would give when applied to a corresponding off-line model.

The output argument yh is a column vector that contains, in row k, the predicted value of $y(k)$, based on past observations and current model. The vector yh thus contains the adaptive predictions of the outputs, and is useful also for noise cancelling and other adaptive filtering applications.

The functions also have optional input arguments that allow the specification of $\theta(0)$, $P(0)$, and $\psi(0)$. Optional output arguments include the last value of the matrix P and of the vector ψ.

Now, rarx is a recursive variant of arx; similarly rarmax is the recursive counterpart of armax and so on. Note, though that rarx does not handle multi-output systems, and rpem does not handle state-space structures.

The function rplr is a variant of rpem, and uses a different approximation of the gradient ψ. It is known as the *recursive pseudo-linear regression approach*, and contains some well known special cases. See Equation (11.57) in Ljung (1999). When applied to the output error model (nn=[0 nb 0 0 nf nk]) it results in methods known as HARF ('ff'-case) and SHARF ('ng'-case). The common *extended least squares* (ELS) method is an rplr algorithm for the ARMAX model (nn=[na nb nc 0 0 nk]).

The following example shows a second order output error model, which is built recursively and its time varying parameter estimates plotted as functions of time:

```
thm = roe(z,[2 2 1],'ff',0.98);
plot(thm)
```

The next example shows how a second order ARMAX model is recursively estimated by the ELS method, using Kalman filter adaptation. The resulting static gains of the estimated models are then plotted as a function of time:

```
[N,dum]=size(z);
thm = rplr(z,[2 2 2 0 0 1],'kf',0.01*eye(6));
nums = sum(thm(:,3:4)')';
dens = ones(N,1)+sum(thm(:,1:2)')';
stg = nums./dens;
plot(stg)
```

So far, the examples of applications where a batch of data is examined cover studies of the variability of the system. The algorithms are, however, also prepared for true on-line applications, where the computed model is used for some on-line decision. This is accomplished by storing the update information

in $\hat{\theta}(t-1)$, $P(t-1)$ and information about past data in $\phi(t-1)$ (and $\psi(t-1)$) and using that information as initial data for the next time step. The following example shows the recursive least squares algorithm being used on line (just to plot one current parameter estimate):

```
%Initialization, first i/o pair y,u (scalars)
[th,yh,P,phi] = rarx([y u],[2 2 1],'ff',0.98);
axis([1 50 -2 2])
plot(1,th(1),'*'),hold
%The online loop:
for k = 2:50
% At time k receive y,u
[th,yh,P,phi] = rarx([y u],[2 2 1],'ff',0.98,th',P,phi);
plot(k,th(1),'*')
end
```

Execute `iddemo #10` to illustrate the recursive algorithms.

Segmentation of Data

Sometimes the system or signal exhibits abrupt changes during the time when the data is collected. It may be important in certain applications to find the time instants when the changes occur and to develop models for the different segments during which the system does not change. This is the *segmentation problem*. Fault detection in systems and detection of trend breaks in time series can serve as two examples of typical problems.

The System Identification Toolbox offers the function `segment` to deal with the segmentation problem. The basic syntax is

```
thm = segment(z,nn)
```

with a format like `rarx` or `rarmax`. The matrix `thm` contains the piecewise constant models in the same format as for the algorithms described earlier in this section.

The algorithm that is implemented in `segment` is based on a model description like (3-58), where the change term $w(t)$ is zero most of the time, but now and then it abruptly changes the system parameters $\theta_0(t)$. Several Kalman filters that estimate these parameters are run in parallel, each of them corresponding to a particular assumption about when the system actually changed. The relative reliability of these assumed system behaviors is constantly judged, and

unlikely hypotheses are replaced by new ones. Optional arguments allow the specification of the measurement noise variance R_2 in (3-57), of the probability of a jump, of the number of parallel models in use, and also of the guaranteed lifespan of each hypothesis. See `segment` in the "Command Reference" chapter.

Some Special Topics

This section describes a number of miscellaneous topics. Most of the information here is also covered in other parts of the manual, but since manuals seldom are read from the beginning, you can also check if a particular topic is brought up here.

Time Series Modeling

When there is no input present, the general model (3-43) reduces to the ARMA model structure:

$$A(q)y(t) = C(q)e(t)$$

With $C(q) = 1$ you have an AR model structure.

Similarly, a state-space model for a time series is given by

$$x(t+1) = Ax(t) + Ke(t)$$
$$y(t) = Cx(t) + e(t)$$

so that the matrices B and D are empty.

Basically all commands still apply to these time-series models, but with natural modifications. They are listed as follows:

```
m= idpoly(A,[ ],C)
e = iddata([],idinput(300,'rgs'))
y = sim(m,e)
```

Spectral analysis (`etfe` and `spa`) returns results in the `idfrd` model format, that now just contains `SpectrumData` and its variance. `bode` will only plot these signal spectra and, if required, the confidence intervals.

```
g = spa(y)
p= etfe(y)
bode(g,p,'sd',3)
```

Note that `etfe` gives the *periodogram* estimate `p` of the spectrum.

`armax` and `arx` work the same way, but need no specification of `nb` and `nk`:

```
th = arx(y,na)
th = armax(y,[na nc])
```

Note that arx also handles multivariable signals, and so do n4sid and pem:

```
m = n4sid(y) %default order
bode(m)
compare(y,m,10) % 10-step ahead predictions being evaluated.
```

Structured state-space models of time series can be built simply by specifying B = [], D = [] in idss and idgrey. resid works the same way for time series models, but does not provide any input-residual correlation plots:

```
resid(m,y)
```

In addition there are two commands that are specifically constructed for building scalar AR models of time series. One is

```
m = ar(y,na)
```

which has an option that allows you to choose the algorithm from a group of several popular techniques for computing the least squares AR model. Among these are Burg's method, a geometric lattice method, the Yule-Walker approach, and a modified covariance method. See the "Command Reference" chapter for details. The other command is

```
m = ivar(y,na)
```

which uses an instrumental variables technique to compute the AR part of a time series.

Finally, when no input is present, the functions bj, iv, iv4, and oe are not of interest.

Here is an example where you can simulate a time series, compare spectral estimates and covariance function estimates, and also the predictions of the model.

```
ts0 = idpoly([1 -1.5 0.7],[]);
ir = sim(ts0,[1;zeros(24,1)]);
Ry0 = conv(ir,ir(25:-1:1)); % The true covariance function
e = idinput(200,'rgs');
y = sim(ts0,e);
plot(y)
per = etfe(y);
speh = spa(y);
ffplot(per,speh,ts0)
```

```
ts2 = ar(y,2); % A second order AR model:
ffplot(speh,ts2,ts0,'sd',3)
% The covariance function estimates:
Ryh = covf(y,25);
Ryh = [Ryh(end:-1:2),Ryh]';
ir2 = sim(ts2,[1;zeros(24,1)]);
Ry2 = conv(ir2,ir2(25:-1:1));
plot([-24:24]'*ones(1,3),[Ryh,Ry2,Ry0])
% The prediction ability of the model:
compare(y,ts2,5)
```

Periodic Inputs

It is often an advantage to use a periodic input for identification, whenever possible. See Section 13.3 in Ljung(1999). If you import or create a periodic input, as in

```
u = idinput([300 2 5]) % Period 300, 2 inputs, 5 periods
```

you should set the corresponding period in the iddata object:

```
u = iddata([],u,'Period',[300; 300]);
```

Normally, an even number of periods should be represented in the data. That will allow the estimation routines to do the right things. For example, `etfe` when called with data with periodic inputs, will honor the period and compute the frequency response on a suitably chosen frequency grid. Try this:

```
m0 =idpoly([1 -1.5 0.7],[0 1 0.5]);
u = idinput([10 1 150],'rbs');
u = iddata([],u,'Period',10);
e = iddata([],randn(1500,1));
y = sim(m0, [u e])
g = etfe([y u])
bode(g,'x',m0) % Good fit at the 5 excited frequencies
```

Connections Between the Control System Toolbox and the System Identification Toolbox

The objects and functions of the Control System Toolbox, are quite similar to those of the System Identification Toolbox. This means that the two toolboxes can be run efficiently together.

Function Calls

The function calls are the same for many essential functions. `bode`, `nyquist`, `step`, `impulse`, `ssdata`, `tfdata`, `zpkdata`, `freqresp`, `minreal`, etc., all do the same things with essentially the same syntax. The System Identification Toolbox commands however also handle model uncertainty. The System Identification Toolbox commands will be used whenever at least one of the objects in the argument list is an `idmodel` or `idfrd` object.

Also, subreferencing channels and concatenations follow the same syntax.

Moreover, most of the LTI-commands for model manipulation, like `G1+G2`, `G1*G2`, `feedback`, `append`, `balreal`, `augstate`, `canon`, etc, will work (using the Control System Toolbox) in the expected way, returning `idmodel` objects. However, covariance information is in most cases lost.

Object Relations

Since the System Identification Toolbox can be run without the Control System Toolbox, there are no formal parent/child relations between the objects in the two toolboxes. There are however easy transformations between them. The command that creates `idmodel`, `idss`, and `idpoly` will accept any LTI object, `zpk`, `tf` or `ss`. `idfrd` can similarly be created from `frd` objects. If the LTI object has an `InputGroup` named `'noise'` these input will be treated as normalized white noise, when creating the `idmodel` object with correct disturbance model information.

Analogously, `ss`, `zpk`, `tf`, and `frd` accept any `idmodel` or `idfrd` (in case of `frd`) object. The covariance information is then not stored in the LTI objects, but all disturbance information will be translated to a group of extra input channels with the group name `'noise'`. If these are interpreted as normalized white noise, the LTI objects have the same disturbance properties as the original `imdmodel` object.

These simple relations also mean that it is easy to use any LTI command in the Control System Toolbox and return to System Identification Toolbox objects:

```
Mb = idss(balreal(ss(M)))
```

Plot Relations

Although the calls `bode`, `step` etc., have essentially the same syntax, the plots look different. The System Identification Toolbox commands show, when required, confidence regions, and typically show the different input/output

channels as separate plots. The sorting of the channels is based on the `InputName` and `OutputName` properties. Therefore the System Identification Toolbox commands allow any mix of models not necessarily of the same sizes.

The System Identification Toolbox plot commands do not offer the same options and plot interaction facilities as `ltiview`. However, applying `view` to one or several `idmodel` objects invokes the LTI Viewer.

Here is an example of the interplay between the functions in the two toolboxes:

```
m0 = drss(4,3,2)
m0 = idss(m0,'NoiseVar',0.1*eye(3))
u = iddata([], idinput([800 2],'rbs'));
e = iddata([], randn(800, 3));
y = sim(m0, [u e])
Data = [y u];
m = pem(Data(1:400))
tf(m)
compare(Data(401:800),m)
view(m)
```

Memory - Speed Trade-Offs

On machines with no formal memory limitations, it is still of interest to monitor the sizes of the matrices that are formed. The typical situation is when an overdetermined set of linear equations is solved for the least squares solution. The solution time depends, of course, on the dimensions of the corresponding matrix. The number of rows corresponds to the number of observed data, while the number of columns corresponds to the number of estimated parameters. The property `MaxSize` used with all the relevant M-files, guarantees that no matrix with more than `MaxSize` elements is formed. Larger data sets and/or higher order models are handled by `for` loops. `for` loops give linear increase in time when the data record is increased, plus some overhead.

If you regularly work with large data sets and/or high order models, it is advisable to tailor the memory and speed trade-off to your machine by choosing `MaxSize` carefully. You could also change the default value of `MaxSize` in the M-file `idmsize`. Then the default value of `MaxSize` (that is `'Auto'`) will be tailored to your needs. Note that this value is allowed to depend on the number of rows and columns of the matrices formed.

Local Minima

The iterative search procedures in `pem`, `armax`, `oe`, and `bj` lead to models corresponding to a local minimum of the criterion function (3-39). Nothing guarantees that this local minimum is also a global minimum. The start-up procedure for black-box models in these routines is, however, reasonably efficient in giving initial estimates that lead to the global minimum.

If there is an indication that a minimum is not as good as you expected, try starting the minimization at several different initial conditions, to see if a smaller value of the loss function can be found. The function `init` can be used for that.

Initial Parameter Values

When only orders and delays are specified, the functions `armax`, `bj`, `oe`, and `pem` use a start-up procedure to produce initial values. The start-up procedure goes through two to four least squares and instrumental variables steps. It is reasonably efficient in that it usually saves several iterations in the minimization phase. Sometimes it may, however, pay to use other initial conditions. For example, you can use an `iv4` estimate computed earlier as an initial condition for estimating an output-error model of the same structure:

```
m1 = iv4(Data,[na nb nk]);
set(m1,'a',1,'f',m1.a)
m2= oe(Data,m1);
```

Another example is when you want to try a model with one more delay (for example, three instead of two) because the leading *b*-coefficient is quite small:

```
m1 = armax(Data,[3 3 2 2]);
m1.b(3) = 0
m2 = armax(Data,m1);
```

If you decrease the number of delays, remember that leading zeros in the B-polynomial are treated as delays. Suppose you go from three to two delays in the above example:

```
m1 = armax(z,[3 3 2 3]);
m1.b(3) = 0.00001;
m2 = armax(Data,m1);
```

Note that when constructing homemade initial conditions, the conditions must correspond to a stable predictor (*C* and *F* being Hurwitz polynomials), and they should not contain any exact pole-zero cancellations.

For user defined structured state-space and multi-output models, you must provide the initial parameter values (initial model) when defining the structure in `idss` or `idgrey`. The basic approach is to use physical insight to choose initial values of the parameters with physical significance, and try some different (randomized) initial values for the others. The routine `init` can be used for that.

Initial State

The filter that computes the prediction errors in (3-36) needs to be properly initialized. For input-output (polynomial) models, values of inputs, outputs and predictions prior to time *t = 0* are required, and state-space models need the initial state *x(0).* There are several ways to handle these unknown states. A simple one is to take all unknown values as zero. If the model predictor has slow dynamics (i.e. the poles of *CF*, or the eigenvalues of *A-KC* are close to the unit circle), this could have a very bad effect on the parameter estimates. It is particularly pronounced for output-error models, where the noise model cannot be adjusted to handle slow transients form initial conditions.

The toolbox offers a number of options how to deal with the initial state of the predictor. They are handled by the model property `InitialState`. The unknown state can be treated as a vector of unknown parameters (`InitialState = 'Estimate'`), they can be set to zero (`InitialState = 'Zero'`), or estimated by a backwards prediction method (`InitialState = 'Backcast'`). It can also be fixed to any user defined value. The default value is `InitialState = 'Auto'`, which makes an automatic choice between the options, guided by the estimation data. For details, see `idss` and `idpoly` in the "Command Reference" chapter. Basically, the effect of the initial conditions on the prediction errors are tested and if they seem to be negligible, `'zero'` is chosen, which gives a fast and efficient algorithm. Otherwise the initial state is estimated or "backcasted." `EstimationInfo` will contain information about which method was chosen in this case.

Proper handling of the initial state is necessary both when models are estimated, and when predictions and simulations are compared. The commands `predict`, `pe`, `sim`, and `compare` all offer options for how to deal with this.

The Estimated Parameter Covariance Matrix

The estimated parameters are uncertain. The amount of uncertainty is measured and described by the covariance matrix of the estimated parameter vector, (this vector is a random variable, since it depends on the random noise that has affected the output). This covariance (uncertainty) can also be estimated from data, as described, e.g. in Chapter 9 of Ljung (1999). The estimated covariance matrix is contained in the estimated model as the property `Model.CovarianceMatrix`. It is used to compute all relevant uncertainty measures of various model input-output properties (Bode plots, uncertain model output, zeros and poles, etc.)

The estimate of the covariance matrix is based on the assumption that the model structure is capable of giving a correct description of the system. For models that contain a disturbance model (*H* is estimated) it, thus, assumed that the model will produce white residuals, for the uncertainty estimate to be correct.

However, for output-error models (*H* fixed to 1, corresponding to $K = 0$ for state space models, and $C = D = A = 1$ for polynomial models), it is not assumed that the residuals are white. Instead, their color is estimated and a correct estimate of the covariance estimate is used. This corresponds to eq (9.42) in Ljung (1999).

No Covariance

Evaluating and visualizing the uncertainty of the estimated models is a very important aspect of system identification. Handling, and translating covariance information takes a major part of the time in many of the routines of the System Identification Toolbox. For example, in `n4sid`, calculating the Cramer-Rao bound (which in this case is used and an indication of the covariance properties) takes much longer than estimating the actual model. In `d2c` and `c2d`, most of the time is spent on covariance handling. If you build models that are of a preliminary nature, and you would like to speed up the calculations, you can add the property name/property value pair `'Covariance'/'None'` to the list of arguments in most relevant routines. This will prevent covariance calculations and set a flag not to spend time on this in future use of the model. This flag can also be set in the model at any time by

```
Model.cov = 'no'
```

nk and InputDelay

What's the difference between the properties `nk` and `InputDelay`? `InputDelay` is defined for all `idmodel` and `idfrd` objects, while `nk` is defined for `idarx`, `idpoly` as well as for `'Free'` and `'Canonical'` `idss` models. Both properties indicate a delay from the input channels to the outputs. For `idarx`, `nk` is a matrix, describing the delays in the different input/output channels, but otherwise both `nk` and `InputDelay` describe the delay from a certain input channel to all the output channels.

`InputDelay` is really a flag that tells the model to append the input delays as time lags, when the model is simulated, or as phase lags when the frequency functions are computed. The `InputDelay` does not show up when the model is represented in state-space form, nor as transfer functions, nor in the input-output polynomials. `InputDelay` can be used both for continuous and discrete time models. In the latter case, the `InputDelay` is measured in number of samples. Moreover `InputDelay` may assume negative values, in order to handle noncausal models.

The property `nk`, on the other hand, is a model structure property, requiring the model to contain the indicated number of delays whatever the parameter values. This means that the state-space matrices, the transfer functions, etc., will show these delays in an explicit manner. Consequently, `nk` is not defined for continuous-time models.

Otherwise the two properties can be used in the same way

```
m1 = pem(Data,4,'InputDelay',[3 2 0])
m2 = pem(Data,4,'nk',[3 2 0])
bode(m1,m2)
A1 = m1.A
A2 = m2.A
```

give identical bode-plots (up to minor variations due to end-effects in the data records), while `A1` and `A2` are different. In fact while `A1` is of size 4-by-4, the matrix `A2` is of size 7-by-7, since three extra states are required to accommodate the extra 2+1 input delays.

Note that setting `nk` to a certain value for a given model gives a model structure that has the indicated delay for any parameter values. The impulse response of the model may however change (not only be shifted) by this assignment.

Linear Regression Models

A linear regression model is of the type

$$y(t) = \theta^T \varphi(t) + e(t) \qquad \textbf{(3-65)}$$

where $y(t)$ and $\varphi(t)$ are measured variables and $e(t)$ represents noise. Such models are very useful in most applications. They allow, for example, the inclusion of nonlinear effects in a simple way. The System Identification Toolbox function `arx` allows an arbitrary number of inputs. You can therefore handle arbitrary linear regression models with `arx`. For example, if you want to build a model of the type

$$y(t) = b_0 + b_1 u(t) + b_2 u^2(t) + b_3 u^3(t) \qquad \textbf{(3-66)}$$

let

```
Data = iddata(y,[ones(size(u)), u, u.^2, u.^3]);
m= arx(Data,'na',0,'nb',[1 1 1 1],'nk',[ 0 0 0 0])
```

This is formally a model with one output and four inputs, but all the model testing in terms of `compare`, `sim`, and `resid` operate in the natural way for the model (3-65), once the data set `Data` is defined as above.

Note that when `pem` is applied to linear regression structures, by default a robustified quadratic criterion is used. The search for a minimum of the criterion function is carried out by iterative search. Normally, use this robustified criterion. If you insist on a quadratic criterion, then set the argument `LimitError` in `pem` to zero. Then `pem` also converges in one step.

Spectrum Normalization and the Sampling Interval

In the function `spa` the spectrum estimate is normalized with the sampling interval T as

$$\Phi_y(\omega) = T \sum_{k=-M}^{M} R_y(kT) e^{-i\omega T} W_M(k) \qquad \textbf{(3-67)}$$

where

$$\hat{R}_y(kT) = \frac{1}{N}\sum_{l=1}^{N} y(lT - kT)y(lT)$$

(See also (3-3)). The normalization in `etfe` is consistent with (3-67). This normalization means that the unit of $\Phi_y(\omega)$ is "power per radians/time unit" and that the frequency scale is "radians/time unit." You then have

$$Ey^2(t) = \frac{1}{2\pi}\int_{-\pi/T}^{\pi/T} \Phi_y(\omega)d\omega \qquad \textbf{(3-68)}$$

In MATLAB, therefore, you have $S1 \approx S2$ where

```
y.ts = T
sp = spa(y);
phiy = squeeze(sp.spec) % squeeze takes out the spurios dimensions
S1 = sum(phiy)/length(phiy)/T;
S2 = sum(y.^2)/size(y,1);
```

Note that `PHIY` contains $\Phi_y(\omega)$ between $\omega = 0$ and $\omega = \pi/T$ with a frequency step of $\pi/(T$ `length(phiy)`$)$. The sum $S1$ is, therefore, the rectangular approximation of the integral in (3-68). The spectrum normalization differs from the one used by `spectrum` in the Signal Processing Toolbox, and the above example shows the nature of the difference.

The normalization with T in (3-67) also gives consistent results when time series are decimated. If the energy above the Nyquist frequency is removed before decimation (as is done in `resample`), the spectral estimates coincide; otherwise you see folding effects.

Try the following sequence of commands:

```
m0 = idpoly(1,[ ],[1 1 1 1]);
       % 4th order MA-process
e = idinput(2000,'rgs')
e = iddata([], e, 'Ts', 1);
y = sim(m0, e);
g1 = spa(y);
g2 = spa(y(1:4:2000)); % This code automatically sets Ts to 4.
ffplot(g1,g2) % Folding effects
```

```
g3 = spa(resample(y,1,4)); % Prefilter applied
ffplot(g1,g3) % No folding
```

For a parametric noise (time series) model

$$y(t) = H(q)e(t); \qquad Ee^2(t) = \lambda$$

the spectrum is computed as

$$\Phi_y(\omega) = \lambda T \left| H(e^{i\omega T}) \right|^2 \tag{3-69}$$

which is consistent with (3-67) and (3-68). Think of λT as the spectral density of the white noise source $e(t)$.

When a parametric disturbance model is transformed between continuous time and discrete time and/or resampled at another sampling rate, the functions `c2d` and `d2c` in the System Identification Toolbox use formulas that are formally correct only for piecewise constant inputs. (See (3-29)). This approximation is good when T is small compared to the bandwidth of the noise. During these transformations the variance λ of the innovations $e(t)$ is changed so that the spectral density $T \cdot \lambda$ remains constant. This has two effects:

- The spectrum scalings are consistent, so that the noise spectrum is essentially invariant (up to the Nyquist frequency) with respect to resampling.
- Simulations with noise using `sim` has a higher noise level when performed at faster sampling.

This latter effect is well in line with the standard description that the underlying continuous-time model is subject to continuous-time white noise disturbances (which have infinite, instantaneous variance), and appropriate low-pass filtering is applied before sampling the measurements. If this effect is unwanted in a particular application, scale the noise source appropriately before applying `sim`.

Note the following cautions relating to these transformations of disturbance models. Continuous-time disturbance models must have a white noise component. Otherwise the underlying state-space model, which is formed and used in `c2d` and `d2c`, is ill-defined. Warnings about this are issued by `idpoly` and these functions. Modify the *C*-polynomial accordingly. Make the degree of

the monic *C*-polynomial in continuous time equal to the sum of the degrees of the monic *A*- and *D*-polynomials; i.e., in continuous time

```
length(C)-1 = (length(A)-1)+(length(D)-1).
```

Interpretation of the Loss Function

The value of the quadratic loss function is given as the field `LossFcn` in the `EstimationInfo` of the model:

```
m.es.LossFcn
```

For multi-output systems, this is equal to the determinant of the estimated covariance matrix of the noise source *e*

For most models the estimated covariance matrix of the innovations is obtained by forming the corresponding sample mean of the prediction errors (squared), computed (using `pe`) from the model with the data for which the model was estimated.

Note the discrepancy between this value and the values shown during the minimization procedure (in `pem`, `armax`, `bj`, or `oe`), since these are the values of the *robustified* loss function (see under `LimitError` on page 3-33). Note also that it is the non-robustified residuals that are used to estimate the variance of *e*, as stored in `Model.NoiseCovariance`. It is also this value that is used to estimate the covariance matrix of the estimated parameters. Outliers may thus influence the estimate of `NoiseVariance` and the covariance matrix, while the parameter estimates are made robust against them.

Be careful when comparing loss function values between different structures that use very different disturbance models. An Output-Error model may have a better input-output fit, even though it displays a higher value of the loss function than, say, an ARX model.

Note that for ARX models computed using `iv4`, the covariance matrix of the innovations is estimated using the provisional disturbance model that is used to form the optimal instruments. The loss function therefore differs from what would be obtained if you computed the prediction errors using the model directly from the data. It is still the best available estimate of the innovations covariance. In particular, it is difficult to compare the loss function in an ARX model estimated using `arx` and one estimated using `iv4`.

Enumeration of Estimated Parameters

In some cases the parameters of a model are given just as an ordered list. This is the case for `m.ParameterVector` and also when online information from the minimization routines are given:

- For the input-output model (3-19) or its multi-input variant (3-41), you have the following alphabetical ordering

$$pars = [a_1, \ldots, a_{na}, b_1^1, \ldots, b_{nb1}^1, b_1^2, \ldots b_{nb2}^2, \ldots$$

$$b_1^{nu}, \ldots, b_{nbnu}^{nu}, c_1, \ldots, c_{nc}, d_1, \ldots, d_{nc},$$

$$f_1^1, \ldots f_{nf1}^1, \ldots, f_1^{nu}, \ldots, f_{nfnu}^{nu}]$$

Here superscript refers to the input number:

- For a state-space structure, defined by `idss` the parameters in `m.ParameterValues` are obtained in the following order. The *A* matrix is first scanned row by row for free parameters. Then the *B* matrix is scanned row by row, and so on for the *C*, *D*, *K*, and *X*0 matrices.
- For a state-space matrix that is defined by `idgrey`, the ordering of the parameters is the same as in the user-written M-file.

Multivariable ARX models are internally represented in state-space form. The parameter ordering follows the one described above. The ordering of the parameters may, however, not be transparent so it is better to use `idarx` and `arxdata`.

Note that the property `PName` (for Parameter Name) may be useful to help with the bookkeeping in these cases, and when fixing certain parameters using `FixedParameter`. The routine `setpname` may be helpful in automatically setting mnemonic parameter names for black-box models.

Complex-Valued Data

Some applications of system identification work with complex-valued data, and thus create complex-valued models. Most of the routines in the System Identification Toolbox support complex data and models. This is true for the estimation routines `ar`, `armax`, `arx`, `bj`, `covf`, `ivar`, `iv4`, `oe`, `pem`, `spa`, and `n4sid`.

The transformation routines, like `freqresp`, `zpkdata`, etc., also work for complex-valued models, but no pole-zero confidence regions are given. Note also that the parameter variance-covariance information then refers to the complex valued parameters, so no separate information about the accuracy of the real and imaginary parts will be given. Some display functions like `compare` and `plot` do not work for the complex case. Use `sim` and plot real and imaginary parts separately.

Strange Results

Strange results can of course be obtained in any number of ways. We only point out two cautions: It is tempting in identification applications to call the residuals `eps`. *Don't do that.* This changes the machine ε, which certainly will give you strange results.

It is also natural to use names like `step`, `phase`, etc., for certain variables. Note though that these variables take precedence over M-files with the same name so be sure you don't use variable names that also are names of M-files.

4

Command Reference

This chapter contains detailed descriptions of all of the functions of user interest in the System Identification Toolbox. It begins with a list of functions grouped by subject area and continues with the entries in alphabetical order. Information is also available through the on-line Help facility.

By typing a function name without arguments, you also get immediate syntax help about its arguments for most functions

For ease of use, most functions have several default arguments. The Syntax first lists the function with the necessary input arguments and then with all the possible input arguments. The functions can be used with any number of arguments between these extremes. The rule is that missing, trailing arguments are given default values, as defined in the manual. Default values are also obtained by entering the arguments as the empty matrix [ ].

MATLAB does not require that you specify all of the output arguments; those not specified are not returned. For functions with several output arguments in the System Identification Toolbox, missing arguments are, as a rule, not computed, in order to save time.

	Help Functions
`help ident`	Lists the commands.
`idhelp`	A micro-manual.
`idprops,` `help idprops`	Lists and explains the object properties.

	The Graphical User Interface
`ident`	Open the interface.
`midprefs`	Set directory where to store start-up information.

	Simulation and Prediction
idinput	Generate input signals.
pe	Compute prediction errors.
predict	Compute predictions according to model.
sim	Simulate a general linear system.

	Data Manipulation
detrend	Remove trends from data.
get/set	Retrieve and modify iddata properties.
iddata	Construct a data object.
idfilt	Filter data.
merge (iddata)	Merge data sets into a multiple experiment set.
misdata	Reconstruct missing input and output data.
plot (iddata)	Plot data.
resample	Resample data.

	Nonparametric Estimation
covf	Estimate covariance function.
cra	Estimate impulse response and covariance functions using correlation analysis.
impulse, step	Estimate impulse and step responses using high order parametric models.
etfe	Estimate spectra and transfer functions using direct Fourier techniques.
spa	Estimate spectra and transfer functions using spectral analysis.

	Parameter Estimation
ar	Estimate AR model.
armax	Estimate ARMAX model.
arx	Estimate ARX model using least squares.
bj	Estimate Box-Jenkins model.
ivar	Estimate AR model using instrumental variable methods.
iv4	Estimate ARX model using four-stage instrumental variable method.
oe	Estimate Output-Error model.
n4sid	Estimate state-space model using subspace method.
pem	Estimate general linear model.

	Model Structure Creation
idarx	Create multivariable ARX-models.
idfrd	Create Identified Frequency Response Data object.
idgrey	Create a greybox linear model using an M-file that you write.
idpoly	Create a model structure for input-output models defined as numerator and denominator polynomials.
idss	Create model structure for linear state-space models with known and unknown parameters.

	Manipulating Model Structures
`get/set`	Retrieve and modify model structures.
`init`	Select or randomize initial parameter values.
`merge (idmodel)`	Merge estimated models.

	Model Conversions
`arxdata`	Compute ARX parameters.
`idmodred`	Reduce a model to lower order.
`c2d`	Transform from continuous to discrete time.
`d2c`	Transform from discrete to continuous time.
`freqresp`	Compute frequency response.
`idfrd`	Convert `idmodel` to the IDFRD object that contains frequency functions and spectra.
`noisecnv`	Convert noise inputs to measured channels
`polydata`	Compute transfer function polynomials.
`ssdata`	Compute state-space matrices.
`tfdata`	Compute transfer functions.
`ss, tf, zpk, frd`	Conversion of `idmodel` to the LTI-objects of the Control Systems Toolbox.
`zpkdata`	Compute zeros, poles, and gains.

	Model Analysis
`bode`	Plot Bode diagrams.
`compare`	Compare measured and simulated outputs.
`ffplot`	Plot frequency functions and spectra.

	Model Analysis (Continued)
`impulse, step`	Plot impulse and step responses.
`nyquist`	Plot Nyquist diagrams.
`present`	Display model on screen.
`pzmap`	Plot zeros and poles.
`view`	Plot model characteristics using the LTI Viewer in the Control System Toolbox.

	Model Validation
`aic, fpe`	Compute model selection criteria
`arxstruc, selstruc`	Select ARX-structure
`compare`	Compare model's simulated or predicted output with actual output
`pe`	Compute prediction errors
`predict`	Predict future outputs
`resid`	Compute and test model residuals
`sim`	Simulate a model

	Assessing Model Uncertainty
`simsd`	Simulate responses from several possible models
`bode, nyquist`	Frequency responses with confidence regions
`impulse, step, sim`	Time responses with confidence regions
`pzmap`	Pole/zero plot with confidence regions
`arxdata, polydata, ssdata, tfdata, zpkdata`	Model data with variance information

	Model Structure Selection
`arxstruc`	Compute loss functions for sets of ARX model structure.
`ivstruc`	Compute loss functions for sets of output error model structures.
`n4sid, pem`	State-space model order can be give as a range.
`selstruc`	Select structure.
`struc`	Generate sets of structures.

	Recursive Parameter Estimation
`rarmax`	Estimate ARMAX or ARMA models recursively.
`rarx`	Estimate ARX or AR models recursively.
`rbj`	Estimate Box-Jenkins models recursively.
`roe`	Estimate Output-Error models (IIR-filters) recursively.
`rpem`	Estimate general input-output models using a recursive prediction error method.
`rplr`	Estimate general input-output models using a recursive pseudo-linear regression method.
`segment`	Segment data and estimate models for each segment.

	General
`get`	Retrieve object properties.
`set`	Set object properties.
`setpname`	Set default, mnemonic parameter names.
`size`	Give sizes of the different objects.
`timestamp`	Show object's time of creation.

Purpose

Compute the Akaike Information Criterion for an estimated model

Syntax

```
am = aic(Model)
```

Description

`Model` is any estimated `idmodel (idarx, idgrey, idpoly, idss)`.

`am` is returned as the value of Akaike's Information theoretic Criterion

$$AIC = \log(V) + \frac{2d}{N}$$

where V is the loss function, d is the number of estimated parameters and N is the number of estimation data.

See Also

`EstimationInfo, fpe`

Reference

Sections 7.4 and 16.4 in Ljung (1999)

Algorithm Properties

Purpose

Describe the algorithm properties that affect the estimation process.

Syntax

```
idprops algorithm
m.algorithm
```

Description

All the `idmodel` objects in the toolbox, `idarx`, `idss`, `idpoly`, and `idgrey`, have a property `Algorithm`, which is a structure that contains a number of options that govern the estimation algorithms. The fields of this structure can be individually set and retrieved in the usual way, such as `get(m,'MaxIter')` or `m.SearchDirection = 'gn'`. Also, autofill applies and the names are case insensitive.

Note 1: 'Algorithm' is a property of `idmodel`. Any algorithm property can be separately set as above. Also, if you have a standard algorithm setup, that you prefer, you can set those properties simultaneously as in `m = pem(Data,mi,'alg',myalg)`

Note 2: The algorithm properties, like all other model properties, will be inherited by the resulting model `m`. If you continue the estimation using `m` as initial model, all previously set algorithm features will thus apply, unless you specify otherwise.

The fields of `Algorithm` are as follows.

1. Applying to all estimation methods:

- **Focus:** This property affects the weighting applied to the fit between the model and the data. It can be used to assure that the model approximates the true system well over certain frequency intervals. **Focus** can assume the following values:
  - `'Prediction'`: This is the default and means that the model is determined by minimizing the prediction errors. It corresponds to a frequency weighting that is given by the input spectrum times the inverse noise model. Typically, this favours a good fit at high frequencies. From a

statistical variance point of view, this is the optimal weighting, but then the approximation aspects (bias) of the fit are neglected.

- `'Simulation'`: This means that frequency weighting of the transfer function fit is given by the input spectrum. Frequency ranges where the input has considerable power will thus be better described by the model. In other words, the model approximation is such that the model will produce as good simulations as possible, when applied to inputs with the same spectra as used for the estimation. For models that have no disturbance model, that is $y = G u + e$, (`A=C=D=1` for `idpoly` models and `K=0` for `idss` models) there is no difference between `'Simulation'` and `'Prediction'`. For models with a disturbance description, i.e. $y = Gu + H e$, G is first estimated with $H = 1$ and then H is estimated by a prediction error method, keeping the estimated transfer function $\hat{G}$ fixed. This option will also guarantee a stable transfer function G.
- 'Stability': The resulting model is guaranteed to be stable, but a prediction weighing is still maintained. Note that forcing the model to be stable could mean that a bad model is obtained. Use only when you know the system to be stable.
- Any SISO linear filter. Then the transfer function from input to output is determined by a frequency fit with this filter times the input spectrum as weighting function. The disturbance model is determined by a prediction error method, keeping the transfer function estimate fixed, as in the simulation case. To obtain a good model fit over a special frequency range, the filter should thus be chosen with a passband over this range. For a model with no disturbance model, the result is the same as first applying prefiltering to data using `idfilt`. The filter can be specified in a few different ways as:
- Any single-input-single-output `idmodel`
- An `ss, tf` or `zpk` model from the Control System Toolbox
- `{A,B,C,D}` with the state-space matrices for the filter
- `{numerator, denominator}` with the transfer function numerator/ denominator of the filter

- **`MaxSize`**: No matrix with more than `MaxSize` elements is formed by the algorithm. Instead, `for`-loops will be used. `MaxSize` will thus decide the memory/speed trade-off, and can prevent slow use of virtual memory. `MaxSize` can be any positive integer, but it is required that the input-output

data contain less than MaxSize elements. The default value of MaxSize is 'Auto', which means that the value is determined in the M-file idmsize. You can edit this file to optimize speed on a particular computer.

- **FixedParameter**: A list of parameters that will be kept fixed to the nominal/ initial values and not estimated. This is a vector of integers containing the indices of the fixed parameters. The numbering of the parameters is the same as in the model property 'ParameterVector'. The parameter names from the property 'PName' can also be used. For structured state-space models, it is easier to fix/unfix parameters by the structure matrices, As, Bs, etc. See idss. When using parameter names to specify the fixed parameters, Fixedparameter is a cell array of strings. The strings may contain the wildcards '*' (meaning any continuation of the given string) and '?' (meaning any character). For example, if all disturbance model parameters start with 'k', FixedParameter = {'k*'} will fix all these parameters. The function setpname may be useful in this context.

2. Algorithm properties that apply to n4sid, estimating state-space models. They then also apply to pem for estimating black-box state-space models, since these are initialized by the n4sid estimate

- **N4Weight:** This property determines some weighting matrices used in the singular-value decomposition that is a central step in the algorithm. Two choices are offered: 'MOESP', which corresponds to the MOESP algorithm by Verhaegen, and 'CVA', which is the canonical variable algorithm by Larimore. See the reference page for n4sid. The default value is 'N4Weight' = 'Auto', which gives an automatic choice between the two options.
- **N4Horison**: Determines the prediction horizons forward and backward, used by the algorithm. This is a row vector with three elements: N4Horison =[r sy su], where r is the maximum forward prediction horizon, i.e., the algorithms uses up to r-step ahead predictors. sy is the number of past outputs, and su is the number of past inputs that are used for the predictions. For an exact definition of these integers, see pages 209-210 in Ljung(1999), where they are called *r, s1* and *s2*. These numbers may have a substantial influence of the quality of the resulting model, and there are no simple rules for choosing them. Making 'N4Horizon' a k-by-3 matrix means that each row of 'N4Horison' will be tried out, and the value that gives the best (prediction) fit to data will be selected. (This option cannot be combined with selection of model order.) If you specify only one column in

'N4Horizon', the interpretation is r=sy=su. The default choice is 'N4Horizon' = 'Auto', which uses an Akaike Information Criterion (AIC) for the selection of sy and su.

4. Properties that apply to estimation methods using iterative search for minimizing a criterion, i.e., armax, bj, oe, and pem:

- **Trace:** This property determines the information about the iterative search that is provided to the MATLAB command window.
  - 'Trace' = 'Off': No information is written to the screen.
  - 'Trace' = 'On': Information about criterion values and the search process is given for each iteration.
  - 'Trace'= 'Full': The current parameter values and the search direction are also given (except in the 'Free' SSParameterization case for idss models)
- **LimitError**: This variable determines how the criterion is modified from quadratic to one that gives linear weight to large errors. Errors larger than LimitError times the estimated standard deviation will carry a linear weight in the criterions.The default value of LimitError is 1.6. LimitError = 0 disables the robustification and leads to a purely quadratic criterion. The standard deviation is estimated robustly as the median of the absolute deviations from the median, divided by 0.7. (See Eq. (15.9)-(15.10) in Ljung (1999)).
- **MaxIter**: The maximum number of iterations performed during the search for minimum. The iterations will stop when MaxIter is reached, or some other stopping criterion is satisfied. The default value of MaxIter is 20. Setting MaxIter = 0 will return the result of the start-up procedure. The actual number of used iterations is given by the property EstimationInfo.Iterations.
- **Tolerance**: Based on the Gauss-Newton vector computed at the current parameter value, an estimate is made of the expected improvement of the criterion at the next iteration. When this expected improvement is less than Tolerance, measured in percent, the iterations are stopped. Default value: 0.01.
- **SearchDirection**: The direction along which a line search is performed to find a lower value of the criterion function. It may assume the following values:

- 'gn': The Gauss-Newton direction (inverse of the Hessian times the gradient direction). If no improvement is found along this direction, the gradient direction is also tried out.
- 'gns': A regularized version of the Gauss-Newton direction. Eigenvalues less than pinvtol (see "Advanced" below) of the Hessian are neglected, and the Gauss-Newton direction is computed in the remaining subspace.
- 'lm': The Levenberg-Marquard method is used. This means that the next parameter value is -pinv(H+d*I)*grad from the previous one, where H is the Hessian, I is the identity matrix, grad is the gradient. d is a number that is increased until a lower value of the criterion is found.
- 'Auto': A choice between the above is made in the algorithm. This is the default choice.

• **Advanced**: This is a structure that contains detailed algorithm choices, that normally the user does not need to get involved in. For detailed explanations, the code will have to be examined**.** 'Advanced' has the following fields:

- Search: Contains fields with relevance for the iterative search:
  a GnsPinvTol: The tolerance for the pseudoinverse used to compute the gns direction. See above. Default 10^-9.
  b LmStep: The next value of d in the LM method is lmstep times the previous one. Default lmstep = 2.
  c StepReduction: In the line search used for other directions than LM, the step is reduced by the factor stepred in each try. Default: StepReduction = 2.
  d MaxBisection: The maximum number of bisections used by the line search along the search direction. Default 10.
  e LmStartValue: The starting value of d in the LM method. Default 0.001.
  f RelImprovement: The iterations are stopped if the relative improvement of the criterion is less than relimp. Default RelImprovement= 0.
- Threshold: Contains fields with thresholds for several tests:
  a Sstability: used for stability test of continuous time models. Model is considered stable if its rightmost pole is to the left of Sstability. Default 0.
  b Zstability: used for stability test of discrete time models. Model is considered stable if all poles are within the distance Zstability from the origin. Default 1.0001.

- `AutoInitialState`: When `InitialState` = 'Auto', the state will be estimated if the ratio of the prediction error norm with zero initial state, to the norm with estimated initial state exceeds `AutoInitialState`. Default 1.2.

- `lmstart`: The starting value of d in the LM method.
- `relimp`: The iterations are stopped if the relative improvement of the criterion is less than `relimp`. Default `relimp = 0`.

See Also

`armax`, `bj`, `EstimationInfo`, `n4sid`, `oe`, `pem`

Reference

For the iterative minimization, see

Dennis, J.E. Jr. and R.B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice Hall, Englewood Cliffs, N.J. 1983.

For a general reference to the identification algorithms, see Ljung (1999), Chapter 10.

Purpose

Estimate the parameters of an AR model for scalar time series.

Syntax

```
m = ar(y,n)
[m ,refl] = ar(y,n,approach,window,maxsize)
```

Description

The parameters of the AR model structure

$$A(q)y(t) = e(t)$$

are estimated using variants of the least-squares method.

The `iddata` object `y` contains the time-series data (just one output channel). The scalar `n` specifies the order of the model to be estimated (the number of *A* parameters in the AR model).

Note that the routine is for scalar time series only. For multivariate data use `arx`.

The estimate is returned in `m` and stored as an `idpoly` model. For the two lattice-based approaches, `'burg'` and `'gl'` (see below), the variable `refl` is returned containing the reflection coefficients in the first row, and the corresponding loss function values in the second. The first column is the zero-th order model, so that the (2,1) element of `refl` is the norm of the time series itself.

Variable `approach` allows you to choose an algorithm from a group of several popular techniques for computing the least-squares AR model. Available methods are as follows:

`approach = 'fb'`: The forward-backward approach. This is the default approach. The sum of a least-squares criterion for a forward model and the analogous criterion for a time-reversed model is minimized.

`approach = 'ls'`: The least-squares approach. The standard sum of squared forward prediction errors is minimized.

`approach = 'yw'`: The Yule-Walker approach. The Yule-Walker equations, formed from sample covariances, are solved.

`approach = 'burg'`: Burg's lattice-based method. The lattice filter equations are solved, using the harmonic mean of forward and backward squared prediction errors.

`approach = 'gl'`: A geometric lattice approach. As in Burg's method, but the geometric mean is used instead of the harmonic one.

The computation of the covariance matrix can be suppressed in any of the above methods by ending the `approach` argument with 0 (zero), for example, `'burg0'`.

Windowing, within the context of AR modeling, is a technique for dealing with the fact that information about past and future data is lacking. There are a number of variants available:

`window = 'now'`: No windowing. This is the default value, except when `approach = 'yw'`. Only actually measured data are used to form the regression vectors. The summation in the criteria starts only at time `n`.

`window = 'prw'`: Pre-windowing. Missing past data are replaced by zeros, so that the summation in the criteria can be started at time zero.

`window = 'pow'`: Post-windowing. Missing end data are replaced by zeros, so that the summation can be extended to time `N + n`. (`N` being the number of observations.)

`window = 'ppw'`: Pre- and post-windowing. This is used in the Yule-Walker approach.

The combinations of approaches and windowing have a variety of names. The least-squares approach with no windowing is also known as the *covariance method*. This is the same method that is used in the `arx` routine. The MATLAB default method, forward-backward with no windowing, is often called the *modified covariance method*. The Yule-Walker approach, least-squares plus pre- and post-windowing, is also known as the *correlation method*.

See `Algorithm Properties` for an explanation of the input argument `maxsize`.

Examples

Compare the spectral estimates of Burg's method with those found from the forward-backward nonwindowed method, given a sinusoid in noise signal.

```
y = sin([1:300]') + 0.5*randn(300,1);
y = iddata(y);
mb = ar(y,4,'burg');
mfb = ar(y,4);
bode(mb,mfb)
```

See Also `arx`, `etfe`, `ivar`, `spa`

References Marple, Jr., S. L. *Digital Spectral Analysis with Applications*, Prentice Hall, Englewood Cliffs, 1987, Chapter 8.

armax

Purpose

Estimate the parameters of an ARMAX or ARMA model.

Syntax

```
m = armax(data,orders)
m = armax(data,'na',na,'nb',nb,'nc',nc,'nk',nk)
m = armax(data,orders,'Property1',Value1,...,'PropertyN',ValueN)
```

Description

`armax` returns `m` as an `idpoly` object with the resulting parameter estimates, together with estimated covariances.

`armax` estimates the parameters of the ARMAX model structure

$$A(q)y(t) = B(q)u(t-nk) + C(q)e(t)$$

using a prediction error method.

`data` is an `iddata` object containing the output-input data. The model orders can be specified as `(...,'na',na,'nb',nb,...)` or by setting the argument `orders` to

```
orders = [na nb nc nk]
```

The parameters `na`, `nb`, and `nc` are the orders of the ARMAX model, and `nk` is the delay. Specifically,

$$na: \quad A(q) = 1 + a_1 q^{-1} + \ldots + a_{na} q^{-na}$$

$$nb: \quad B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$$

$$nc: \quad C(q) = 1 + c_1 q^{-1} + \ldots + c_{nc} q^{-nc}$$

Alternatively, you can specify the vector as

```
orders = mi
```

where `mi` is an initial guess at the ARMAX model given in `idpoly` format. See "Polynomial Representation of Transfer Functions" on page 3-11 in the "Tutorial" chapter for more information.

For multi-input systems, `nb` and `nk` are row vectors, such that the `k`-th entry corresponds to the order and delay associated with the `k`-th input.

If `data` has no input channels and just one output channel (i.e., it is a time series) then

```
orders = [na nc],
```

and `armax` calculates an ARMA model for the time series

$$A(q)y(t) = C(q)e(t)$$

The structure and the estimation algorithm are affected by any property name/property value pairs that are set in the input argument list. Useful properties are `'Focus'`, `'InitialState'`, `'Trace'`, `'MaxIter'`, `'Tolerance'`, `'LimitError'`, and `'FixedParameter'`.

See `Algorithm Properties`, `idpoly` and `idmodel` for details of these properties and their possible values.

`armax` does not support multi-output models. Use the state-space model for this case (see `n4sid` and `pem`).

Algorithm

A robustified quadratic prediction error criterion is minimized using an iterative search algorithm, whose details are governed by the properties `'SearchDirection'`, `'MaxIter'`,`'Tolerance'` and `'Advanced'`. The iterations are terminated when `MaxIter` is reached, when the expected improvement is less than `Tolerance`, or when a lower value of the criterion cannot be found. Information about the search is contained in `m.EstimationInfo`.

The initial parameter values for the iterative search, if not specified in `orders`, are constructed in a special four-stage LS-IV algorithm.

The cut-off value for the robustification is based on the property `LimitError` as well as on the estimated standard deviation of the residuals from the initial parameter estimate. It is not recalculated during the minimization.

A stability test of the predictor is performed, so as to assure that only models corresponding to stable predictors are tested. Generally, both $C(q)$ and $F_i(q)$ (if applicable) must have all their zeros inside the unit circle.

Information about the minimization is furnished to the screen in case the property `'Trace'` is set to `'On'` or `'Full'`. With `'Trace' = 'Full'`, current and previous parameter estimates (in column vector form, listing parameters in alphabetical order) as well as the values of the criterion function are given. The Gauss-Newton vector and its norm are also displayed. With `'Trace' = 'On'` just criterion values are displayed.

See Also `arx`, `bj`, `idmodel`, `idpoly`, `oe`, `pem`, `Algorithm Properties`, `EstimationInfo`

References Ljung (1999), Section 10.2.

Purpose

Estimate the parameters of an ARX or AR model.

Syntax

```
m = arx(data,orders)
m = arx(data,'na',na,'nb',nb,'nk',nk)
m= arx(data,orders,'Property1',Value1,...,'PropertyN',ValueN)
```

Description

The parameters of the ARX model structure

$$A(q)y(t) = B(q)u(t-nk) + e(t)$$

are estimated using the least-squares method.

`data` is an `iddata` object that contains the output-input data. `orders` is given as

```
orders = [na nb nk]
```

defining the orders and delay of the ARX model. Specifically,

$$na: \quad A(q) = 1 + a_1 q^{-1} + \ldots + a_{na} q^{-na}$$

$$nb: \quad B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$$

See “Polynomial Representation of Transfer Functions” on page 3-11 in the “Tutorial” chapter for more information. The model orders can also be defined by explicit pairs `(...,'na',na,'nb',nb,'nk',nk,...)`.

`m` is returned as the least-squares estimates of the parameters. For single-output data this is an `idpoly` object, otherwise an `idarx` object.

For a time series, `data` contains no input channels and `orders = na`. Then an AR model of order `na` for `y` is computed.

$$A(q)y(t) = e(t)$$

Models with several inputs

$$A(q)y(t) = B_1(q)u_1(t-nk_1) + \ldots B_{nu}u_{nu}(t-nk_{nu}) + e(t)$$

are handled by allowing `nb` and `nk` to be row vectors defining the orders and delays associated with each input.

Models with several inputs and several outputs are handled by allowing na, nb, and nk to contain one row for each output number. See "Multivariable ARX Models: The idarx Model" on page 3-38 in the "Tutorial" chapter for exact definitions.

The algorithm and model structure are affected by the property name/property value list in the input argument.

Useful options are reached by the properties 'Focus', 'InputDelay', and 'MaxSize'.

See Algorithm Properties for details of these properties and possible values

When the true noise term $e(t)$ in the ARX model structure is not white noise and na is nonzero, the estimate does not give a correct model. It is then better to use armax, bj, iv4, or oe.

Examples

Here is an example that generates data and estimates an ARX model.

```
A = [1  -1.5  0.7]; B = [0 1 0.5];
m0 = idpoly(A,B);
u = iddata([],idinput(300,'rbs'));
e = iddata([],randn(300,1));
y = sim(m0, [u e]);
z = [y,u];
m = arx(z,[2 2 1]);
```

Algorithm

The least-squares estimation problem is an overdetermined set of linear equations that is solved using QR-factorization.

The regression matrix is formed so that only measured quantities are used (no fill-out with zeros). When the regression matrix is larger than MaxSize, the QR-factorization is performed in a for-loop.

See Also

ar, ivx, iv4, Algorithm Properties, EstimationInfo

Purpose

Extract the ARX parameters from `idmodel` models.

Syntax

```
[A,B] = arxdata(m)
[A,B,dA,dB] = arxdata(m)
```

Description

`m` is the model as an `idarx` or `idpoly` model object. `arxdata` will work on any `idarx` model. For `idpoly` it will give an error unless the underlying model is an ARX model, i.e., the orders `nc=nd=nf=0`. (See the reference page for `idpoly`.)

`A` and `B` are returned in the standard multivariable ARX format (see `idarx`), describing the model.

$$y(t) + A_1 y(t-1) + A_2 y(t-2) + \ldots + A_{na} y(t-na) =$$

$$B_0 u(t) + B_1 u(t-1) + \ldots + B_{nb} u(t-nb) + e(t)$$

Here A_k and B_k are matrices of dimensions *ny*-by-*ny* and *ny*-by-*nu*, respectively (*ny* is the number of outputs, i.e., the dimension of the vector $y(t)$ and *nu* is the number of inputs). See “Multivariable ARX Models: The idarx Model” on page 3-38 in the “Tutorial” chapter.

The arguments `A` and `B` are 3-D arrays that contain the A matrices and the B matrices of the model in the following way:

A is an `ny-by-ny-by-(na+1)` array such that

```
A(:,:,k+1) = Ak
A(:,:,1) = eye(ny)
```

Similarly B is an `ny-by-nu-by-(nb+1)` array with

```
B(:,:,k+1) = Bk
```

Note that `A` always starts with the identity matrix, and that leading entries in `B` equal to zero means delays in the model. For a time series `B = []`.

`dA` and `dB` are the estimated standard deviations of `A` and `B`.

See Also

`idarx`

arxstruc

Purpose

Compute loss functions for a set of different model structures of single-output ARX type.

Syntax

```
V = arxstruc(ze,zv,NN)
V = arxstruc(ze,zv,NN,maxsize)
```

Description

`NN` is a matrix that defines a number of different structures of the ARX type. Each row of `NN` is of the form

```
nn = [na nb nk]
```

with the same interpretation as described for `arx`. See `struc` for easy generation of typical `NN` matrices for single-input systems.

Each of `ze` and `zv` are `iddata` objects containing output-input data. Models for each of the model structures defined by `NN` are estimated using the data set `ze`. The loss functions (normalized sum of squared prediction errors) are then computed for these models when applied to the validation data set `zv`. The data sets, `ze` and `zv`, need not be of equal size. They could, however, be the same sets, in which case the computation is faster.

Note that `arxstruc` is intended for single-output systems only.

The output argument `V` is best analyzed using `selstruc`. It contains the loss functions in its first row. The remaining rows of `V` contain the transpose of `NN`, so that the orders and delays are given just below the corresponding loss functions. The last column of `V` contains the number of data points in `ze`. The selection of a suitable model structure based on the information in `v` is normally done using `selstruc`. See "Model Structure Selection and Validation" on page 3-64 in the "Tutorial" chapter for advice on model structure selection and cross-validation.

See `Algorithm Properties` for an explanation of `maxsize`.

Examples

Compare first to fifth order models with one delay using cross-validation on the second half of the data set. Then select the order that gives the best fit to the validation data set.

```
NN = struc(1:5,1:5,1);
V = arxstruc(z(1:200),z(201:400),NN);
nn = selstruc(V,0);
m = arx(z,nn);
```

See Also

`arx`, `ivstruc`, `n4sid`, `selstruc`, `struc`

bj

Purpose

Estimate the parameters of a Box-Jenkins model.

Syntax

```
m = bj(data,orders)
m = bj(data,'nb',nb,'nc',nc,'nd',nd,'nf',nf,'nk',nk)
m = bj(data,orders,'Property1',Value1,'Property2',Value2,...)
```

Description

`bj` returns `m` as an `idpoly` object with the resulting parameter estimates, together with estimated covariances. The `bj` function estimates parameters of the Box-Jenkins model structure

$$y(t) = \frac{B(q)}{F(q)}u(t-nk) + \frac{C(q)}{D(q)}e(t)$$

using a prediction error method.

`data` is an `iddata` object containing the output-input data. The model orders can be specified by setting the argument `orders` to

```
orders = [na nb nc nk]
```

The parameters `nb`, `nc`, `nd`, and `nf` are the orders of the Box-Jenkins model and `nk` is the delay. Specifically,

$$na: \quad A(q) = 1 + a_1 q^{-1} + \ldots + a_{na} q^{-na}$$

$$nb: \quad B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$$

$$nd: \quad D(q) = 1 + c_1 q^{-1} + \ldots + c_{nc} q^{-nc}$$

$$nf: \quad F(q) = 1 + f_1 q^{-1} + \ldots + f_{nf} q^{-nf}$$

The orders can also be defined as property name/property value pairs `(...,'nb',nb,...)`. Alternatively, you can specify the vector as

```
orders = mi
```

where `mi` is an initial guess at the Box-Jenkins model given in `idpoly` format. See “Polynomial Representation of Transfer Functions” on page 3-11 in the “Tutorial” chapter for more information.

For multi-input systems, `nb`, `nf`, and `nk` are row vectors with as many entries as there are input channels. Entry number `i` then describes the orders and delays associated with the `i`-th input.

The structure and the estimation algorithm are affected by any property name/property value pairs that are set in the input argument list. Useful properties are `'Focus'`, `'InitialState'`, `'Trace'`, `'MaxIter'`, `'Tolerance'`, `'LimitError'`, and `'FixedParameter'`.

See Algorithm Properties and the reference pages for idmodel and idpoly for details of these properties and their possible values.

bj does not support multi-output models. Use state-space model for this case (see n4sid and pem).

Examples

Here is an example that generates data and stores the results of the startup procedure separately.

```
B = [0 1 0.5];
C = [1 -1 0.2];
D = [1 1.5 0.7];
F = [1 -1.5 0.7];
m0 = idpoly(1,B,C,D,F,0.1);
e = idddat([],randn(200,1));
u = iddata([],idinput(200));
y = sim(m0,[u e]);
z = [y u];
mi = bj(z,[2 2 2 2 1],'MaxIter',0)
m = bj(z,mi)
m.EstimationInfo
m = bj(z,m); % Continue if m.es.WhyStop shows that maxiter has
             % been reached.
compare(z,m,mi)
```

Algorithm

bj uses essentially the same algorithm as armax with modifications to the computation of prediction errors and gradients.

See Also

armax, idmodel, idpoly, oe, pem

Purpose

Plot frequency functions in Bode diagram form.

Syntax

```
bode(m)
[mag,phase,w] = bode(m)
[mag,phase,w,sdmag,sdphase] = bode(m)
bode(m1,m2,m3,...,w)
bode(m1,'PlotStyle1',m2,'PlotStyle2',...)
bode(m1,m2,m3,..'sd',sd,'mode',mode,'ap',ap)
bode(m1,m2,m3,..'sd',sd,'mode',mode,'ap',ap,'fill')
```

Description

`bode` computes the magnitude and phase of the frequency response of `idmodel` and `idfrd` models. When invoked without left-hand arguments, `bode` produces a Bode plot on the screen.

`bode(m)` plots the Bode response of an arbitrary `idmodel` or `idfrd` model `m`. This model can be continuous or discrete, and SISO or MIMO. The `InputNames` and `OuputNames` of the models are used to plot the responses for different I/O channels in separate plots. Pressing the **Enter** key advances the plot from one input-output pair to the next one.

If `m` contains information about both I/O channels and output noise spectra, only the I/O channels are shown. To show the output noise spectra enter `m('n')` ('n' for 'noise') in the model list. Analogously, specific I/O channels can be selected by the normal subreferencing: `m(ky,ku)`.

Arguments sd, ap, mode and w.

The arguments `sd, ap, mode` and `w` can appear in any order, or be omitted.

`sd`: If `sd` is specified as a number larger than zero, confidence intervals for the functions are added to the graph as dash-dotted curves (of the same color as the estimate curve). They indicate the confidence regions corresponding to `sd` standard deviations. If an argument `'fill'` is included in the argument list, the confidence region is marked as a filled band instead.

`ap`: By default, amplitude and phase plots are shown simultaneously for each I/O channel present in `m`. For spectra, phase plots are omitted. To show amplitude plots only, use `'ap'= 'A'`. For phase plots only, use `'ap'= 'P'`. The default is `'ap' = 'B'` for both plots.

`mode`: To obtain all plots on the same diagram use `mode = 'same'`.

`w`: `bode(m,w)` explicitly specifies the frequency range or frequency points to be used for the plot or for computing the response. To focus on a particular frequency interval `[wmin,wmax]`, set `w = {wmin,wmax}` (Notice the curly brackets). To use particular frequency points, set `w` to the vector of desired frequencies. Use `logspace` to generate logarithmically spaced frequency vectors. All frequencies should be specified in radians/sec.

Note that the frequencies cannot be specified for `idfrd` objects. For those the plot and response are calculated for the internally stored frequencies.

Several Models

`bode(m1,m2,...,mN)` or `bode(m1,m2,...mN,w)` plots the Bode response of several `idmodel` or `idfrd` models on a single figure. The models may be mixes of different sizes and continuous/discrete. The sorting of the plots is made based on the `InputNames` and `OutputNames`. If the frequencies `w` are specified, these will apply to all non-`idfrd` models in the list. If you want different frequencies for different models, you should thus first convert them to `idfrd` objects using the `idfrd` command.

`bode(m1,'PlotStyle1',...,mN,'PlotStyleN')` further specifies which color, linestyle and/or marker should be used to plot each system, as in

```
bode(m1,'r--',m2,'gx')
```

Arguments

The output argument `w` contains the frequencies for which the response is given, whether specified among the input arguments or not. The output arguments `mag` and `phase` are 3-D arrays with dimensions

```
(number of outputs)x(number of inputs)x(length of w)
```

For SISO systems `mag(1,1,k)` and `phase(1,1,k)` give the magnitude and phase (in degrees) at the frequency ω_k = `w(k)`. To obtain the result as a normal vector of responses use `mag = mag(:)` and `phase = phase(:)`.

For MIMO systems `mag(i,j,k)` is the magnitude of the frequency response at frequency `w(k)` from input `j` to output `i`, and similairly for `phase(i,j,k)`.

If sdmag and sdphase are specified, the standard deviations of the magnitude and phase are also computed. Then sdmag is an array of the same size as mag, containing the estimated standard deviations of the response, and analogously for sdphase.

See Also

etfe, ffplot, freqresp, idfrd, nyquist, spa

compare

Purpose

Compare measured outputs with model outputs.

Syntax

```
compare(data,m);
compare(data,m,k,sampnr,init)
compare(data,m1,m2,...,mN,Yplots)
compare(data,m1,'PlotStyle1',...,mN,'PlotStyleN',k,sampnr,init)
[yh,fit] =
   compare(data,m1,'PlotStyle1',...,mN,'PlotStyleN',k,sampnr,init)
```

Description

`data` is the output-input data in the usual `iddata` object format.

`compare` computes the output `yh` that results when the model `m` is simulated with the input `u`. The result is plotted together with the corresponding measured output `y`. The percentage of the ouput variation that is explained by the model

```
fit = 100*(1 - norm(yh - y)/norm(y-mean(y)))
```

is also computed and displayed. For multi-output systems this is done separately for each output.

When the argument `k` is specified, the *k*-step ahead prediction of `y` according to the model `m` are computed instead of the simulated output. In the calculation of $yh(t)$, the model can use outputs up to time $t-k$: $y(s), s = t-k, t-k-1, \ldots$ (and inputs up to the current time t). The default value of `k` is `inf`, which gives a pure simulation from the input only.

A last argument `Yplots` may be given as a cell array of strings. Only the outputs with `OutputName` in this array will be plotted, while all are used for the necessary computations. If `Yplots` is not specified, all outputs will be plotted.

The argument `sampnr` indicates that only the sample numbers in this row vector are plotted and used for the calculation of the fit. The whole data record is used for the simulation/prediction, though.

The argument init determines how to handle initial conditions in the models:

`init = 'e' (for 'estimate')` estimates the initial conditions for best fit.

`init = 'm' (for 'model')` used the model's internally stored initial state.

`init = 'z' (for 'zero')` uses zero initial conditions.

`init = x0`, where `x0` is a column vector of the same size as the state vector of the models, uses `x0` as the initial state.

`init = 'e'` is default.

When several models are specified, as in `compare(data,m1,m2,...,mN)`, the plots show responses and fits for all models. In that case `data` should contain all inputs and outputs that are required for the different models. However, some models may very well correspond to subselections of channels and may not need all channels in `data`. In that case the proper handling of signals is based on the `InputNames` and `OutputNames` of `data` and the models.

With `compare(data,m1,'PlotStyle1',...mN,'PlotStyle2')` the color, linestyle, and/or marker can be specified for the curves associated with the different models. The markers are the same as for the regular `plot` command. For example,

```
compare(data,m1,'g_*',m2,'r:')
```

If `data` contains several experiments, separate plots are given for the different experiments. In this case `sampnr`, if specified, must be a cell array with as many entries as there are experiments.

Arguments

When output arguments `[yh,fit] = compare(data,m1,..,mN)` are specified, no plots are produced.

`yh` is a cell array of length equal to the number of models. Each cell contains the corresponding model output as an `iddata` object.

`fit` is in the general case a 3-D array with `fit(kexp,kmod,ky)` containing the fit (computed as above) for output `ky`, model `kmod`, and experiment `kexp`.

Examples

Split the data record into two parts. Use the first one for estimating a model and the second one to check the model's ability to predict six steps ahead:

```
ze = z(1:250);
zv = z(251:500);
m= armax(ze,[2 3 1 0]);
compare(zv,m,6);
```

See Also

`sim`, `predict`

covf

Purpose

Estimate time-series covariance functions.

Syntax

```
R = covf(data,M)
R = covf(data,M,maxsize)
```

Description

`data` is an `iddata` object and `M` is the maximum delay -1 for which the covariance function is estimated.

Let `z` contain the output and input channels:

```
z = [data.OutputData, data.InputData]
```

with a total of `nz` channels.

`R` is returned as an nz^2 -by- M matrix with entries

$$R(i+(j-1)nz,k+1) = \frac{1}{N}\sum_{t=1}^{N} z_i(t)z_j(t+k) = \hat{R}_{ij}(k)$$

where z_j is the `j`-th column of `z` and missing values in the sum are replaced by zero.

The optional argument `maxsize` controls the memory size as explained under `Algorithm Properties`.

The easiest way to describe and unpack the result is to use

```
reshape(R(:,k+1),nz,nz) = E z(t)*z'(t+k)
```

Here `'` is complex conjugate transpose, which also explains how complex data are handled. The expectation symbol `E` corresponds to the sample means.

Algorithm

When `nz` is at most two, and when permitted by `maxsize`, a fast Fourier transform technique is applied. Otherwise, straightforward summing is used.

See Also

`spa`

Purpose

Perform prewhitening based correlation analysis and estimate impulse response.

Syntax

```
cra(data);
[ir,R,cl] = cra(data,M,na,plot);
cra(R);
```

Description

`data` is the output-input data given as an `iddata` object.

The routine only handles single-input-single-output data pairs. (For the multivariate case, apply `cra` to two signals at a time, or use `impulse`.) `cra` prewhitens the input sequence, i.e., filters `u` through a filter chosen so that the result is as uncorrelated (white) as possible. The output `y` is subjected to the same filter, and then the covariance functions of the filtered `y` and `u` are computed and graphed. The cross correlation function between (prewhitened) input and output is also computed and graphed. Positive values of the lag variable then corresponds to an influence from `u` to later values of `y`. In other words, significant correlation for negative lags is an indication of feedback from `y` to `u` in the data.

A properly scaled version of this correlation function is also an estimate of the system's impulse response `ir`. This is also graphed along with 99% confidence levels. The output argument `ir` is this impulse response estimate, so that its first entry corresponds to lag zero. (Negative lags are excluded in `ir`.) In the plot, the impulse response is scaled, so that it corresponds to an impulse of height $1/T$ and duration T, where T is the sampling interval of the data.

The output argument `R` contains the covariance/correlation information as follows: The first column of `R` contains the lag indices. The second column contains the covariance function of the (possibly filtered) output. The third column contains the covariance function of the (possibly prewhitened) input, and the fourth column contains the correlation function. The plots can be redisplayed by `cra(R)`.

The output argument `cl` is the 99% confidence level for the impulse response estimate.

The optional argument `M` defines the number of lags for which the covariance/correlation functions are computed. These are from –`M` to `M`, so that the length

of R is 2M+1. The impulse response is computed from 0 to M. The default value of M is 20.

For the prewhitening, the input is fitted to an AR model of order na. The third argument of cra can change this order from its default value na = 10. With na = 0 the covariance and correlation functions of the original data sequences are obtained.

plot: plot = 0 gives no plots. plot = 1 (default) gives a plot of the estimated impulse response together with a 99% confidence region. plot = 2 gives a plot of all the covariance functions.

An often better alternative to cra are the functions impulse and step, which use a high order FIR model to estimate the impulse response.

Examples

Compare a second order ARX model's impulse response with the one obtained by correlation analysis.

```
ir = cra(z);
m = arx(z,[2 2 1]);
imp = [1;zeros(19,1)];
irth = sim(m,imp);
subplot(211)
plot([ir irth])
title('impulse responses')
subplot(212)
plot([cumsum(ir),cumsum(irth)])
title('step responses')
```

See Also

impulse, step

Purpose

Convert a model from continuous time to discrete time.

Syntax

```
md = c2d(mc,T)
md = c2d(mc,T,method)
```

Description

`mc` is a continuous-time model as any `idmodel` object (`idgrey`, `idpoly`, or `idss`). `md` is the model that is obtained when it is sampled with sampling interval `T`.

Note that the covariance matrix of `mc` is not translated.

`method = 'zoh'` (default) makes the translation to discrete time under the assumption that the input is piecewise constant (zero-order hold).

`method = 'foh'` assumed the input to be piecewise linear between the sampling instants (first-order-hold).

Note that the innovations variance λ of the continuous-time model is interpreted as the intensity of the spectral density of the noise spectrum. The noise variance in `md` will thus be given as λ/T.

`idpoly` and `idss` models are returned in the same format. `idgrey` structures will be preserved if their `CDMfile` property is equal to 'cd'. Otherwise they will be transformed to `idss` objects.

Examples

Define a continuous-time system and study the poles and zeros of the sampled counterpart.

```
mc = idpoly(1,1,1,1,[1 1 0],'Ts',0);
md = c2d(mc,0.5);
pzmap(md)
```

See Also

`d2c`

detrend

Purpose

Remove trends from output-input data.

Syntax

```
zd = detrend(z)
zd = detrend(z,o,brkp)
```

Description

`z` is an `iddata` object containing the input-output data. `detrend` removes the trend from each signal and returns the result as an `iddata` object `zd`.

The default (`o = 0`) removes the zero-th order trends, i.e., the sample means are subtracted.

With `o = 1`, linear trends are removed, after a least-squares fit. With `brkp` not specified, one single line is subtracted from the entire data record. A continuous piecewise linear trend is subtracted if `brkp` contains breakpoints at sample numbers given in a row vector.

Note that `detrend` for `iddata` objects differs somewhat from `detrend` in the Signal Processing Toolbox.

Examples

Remove a V-shaped trend from the output with its peak at sample number 119, and remove the sample mean from the input.

```
zd(:,1,[]) = detrend(z(:,1,[]),1,119);
zd(:,[],1) = detrend(z(:,[],1));
```

Purpose

Convert a model from discrete to continuous time.

Syntax

```
mc = d2c(md)
mc = d2c(md,'CovarianceMatrix',cov,'InputDelay',inpd)
```

Description

The discrete-time model `md`, given as any `idmodel` object, is converted to a continuous-time counterpart `mc`. The covariance matrix of the parameters in the model is also translated using Gauss' approximation formula and numerical derivatives of the transformation. The step-sizes in the numerical derivatives are determined by the function `nuderst`. To inhibit the translation of the covariance matrix and save time, enter among the input arguments `(...,'CovarianceMatrix,'None',....)` (any abbreviations will do).

If the discrete-time model contains pure time delays, i.e., $nk > 1$, then these are first removed before the transformation is made. These delays are appended as pure time-delay (deadtime) to the continuous-time model as the property `InputDelay`. To have the time delay approximated by a finite-dimensional continuous system, enter among the input arguments `(...,'InputDelay',0,...)`.

If the noise variance is λ in `md`, and its sampling interval is T, then the continuous-time model has an indicated level of noise spectral density equal to $T\,\lambda$.

While `idpoly` and `idss` models are returned in the same format, `idarx` models are returned as `idss` models `mc`. The reason is that the transformation does not preserve the special structure of `idarx`. `idgrey` structures will be preserved if their `CDMfile` property is equal to 'cd'. Otherwise they will be transformed to `idss` objects.

Note The transformation from discrete to continuous time is not unique. `d2c` selects the continuous-time counterpart with the slowest time constants, consistent with the discrete-time model. The lack of uniqueness also means that the transformation may be ill-conditioned or even singular. In particular, poles on the negative real axis, in the origin, or in the point 1, are likely to cause problems. Interpret the results with care.

Examples

Transform an identified model to continuous time and compare the frequency responses of the two models.

```
m = n4sid(data,3)
mc = d2c(m);
bode(m.mc,'sd',3)
```

Note that the transformation to continuous time can be included in the `n4sid` command by specifying the model to be continuos time.

```
mc = n4sid(data,3,'Ts',0)
```

See Also

`c2d`, `nuderst`

References

See "Discrete and Continuous Time Models" on page 3-61 and "Spectrum Normalization and the Sampling Interval" on page 3-95" in the "Tutorial" chapter.

Purpose

To provide information about the results of the estimation process

Syntax

```
m.EstimationInfo
m.es
m.es.DataLength, etc
```

Description

Any estimated model has the property `EstimationInfo`, which is a structure whose fields give information about the results of the estimation. The model structure will contain the properties `ParameterVector`, `CovarianceMatrix`, and `NoiseVariance`, which are all calculated in the estimation process (see the reference page for `idmodel`). In addition, `EstimationInfo` contains the following fields.

- **`Status:`** Information whether the model has been estimated, or modified after being estimated.
- **`Method:`** The name of the estimation command that produced the model.
- **`LossFcn:`** The value of the identification criterion at the estimate. Normally equal to the determinant of the covariance matrix of the prediction errors, i.e., the determinant of `NoiseVariance`. Note that the loss function for the minimization might be different due to `LimitError`. In `LossFcn`, always the value of the non-robustified loss function is stored.
- **`FPE:`** Akaikes Final Prediction Error, defined as `LossFcn *(1+d/N}/(1-d/N)` where `d` is the number of estimated parameters and `N` is the length of the data record.
- **`DataName`**: Name of the data set from which the model was estimated. This is equal to the property `name` of the `iddata` object. If this was not defined, the name of the MATLAB `iddata` variable is used.
- **`DataLength`**: The length of the data record
- **`DataTs`**: The sampling interval of the data
- **`DataInterSample:`** The intersample behavior of the data from which the model was estimated. This equals the property `InterSample` of the `iddata` object. (See `iddata`.)
- **`WhyStop:`** For models that have been estimated by iterative search. The stopping rule that caused the iterations to terminate. Assumes values like `'MaxIter reached','No improvement possible along the search`

vector' or 'Near (local) minimum'. The latter means that the expected improvement is less than Tolerance (see Algorithm Properties).

- **UpdateNorm**: The norm of the Gauss-Newton vector at the last iteration
- **LastImprovement**: The relative improvement of the criterion value at the last iteration.
- **Iterations**: The number of iterations used in the search.
- **InitialState**: The actually used option when Model.InitialState = 'auto'.
- **N4Weight**: For n4sid estimates, or estimates that have been initialized by n4sid: the actual value of N4Weight used.
- **N4Horizon**: For n4sid estimates, or estimates that have been initialized by n4sid: the actual value of N4Horizon used. See n4sid and Algorithm Properties.

See also idfrd

Purpose

Estimate empirical transfer functions and periodograms.

Syntax

```
g = etfe(data)
g = etfe(data,M,N)
```

Description

`etfe` estimates the transfer function `g` of the general linear model

$$y(t) = G(q)u(t) + v(t)$$

`data` contains the output-input data and is an `iddata` object.

`g` is given as an `idfrd` object with the estimate of $G(e^{i\omega})$ at the frequencies

```
w = [1:N]/N*pi/T
```

The default value of `N` is 128.

In case `data` contains a time series (no input channels), `g` is returned as the periodogram of `y`.

When `M` is specified other than the default value `M = []`, a smoothing operation is performed on the raw spectral estimates. The effect of `M` is then similar to the effect of `M` in `spa`. This can be a useful alternative to `spa` for narrowband spectra and systems, which require large values of `M`.

When `etfe` is applied to time series, the corresponding spectral estimate is normalized in the way that is defined in the section "Spectrum Normalization and the Sampling Interval" on page 3-95 in the *Tutorial*. Note that this normalization may differ from the one used by `spectrum` in the Signal Processing Toolbox.

If the (input) data is marked as periodic (`data.Period = integer`) and contains an even number of periods, the response is computed at the frequencies `k*2*pi/period` for `k=0` up to the Nyquist frequency.

Examples

Compare an empirical transfer function estimate to a smoothed spectral estimate.

```
ge = etfe(z);
gs = spa(z);
bode(ge,gs)
```

Generate a periodic input, simulate a system with it, and compare the frequency response of the estimated model with the true system at the excited frequency points.

```
m = idpoly([1 -1.5 0.7],[0 1 0.5]);
u = iddata([],idinput([50,1,10],'sine'));
u.Period = 50;
y = sim(u,m);
me = etfe([y u])
bode(me,'b*',m)
```

Algorithm

The empirical transfer function estimate is computed as the ratio of the output Fourier transform to the input Fourier transform, using `fft`. The periodogram is computed as the normalized absolute square of the Fourier transform of the time series.

The smoothed versions (`M` less than the length of `z`) are obtained by applying a Hamming window to the output fast Fourier transform (FFT) times the conjugate of the input FFT, and to the absolute square of the input FFT, respectively, and subsequently forming the ratio of the results. The length of this Hamming window is equal to the number of data points in `z` divided by `M`, plus one.

See Also

spa

Purpose

Plot frequency functions and spectra.

Syntax

```
ffplot(m)
[mag,phase,w] = ffplot(m)
[mag,phase,w,sdmag,sdphase] = ffplot(m)
ffplot(m1,m2,m3,...,w)
ffplot(m1,'PlotStyle1',m2,'PlotStyle2',...)
ffplot(m1,m2,m3,..'sd',sd,'mode',mode,'ap',ap)
```

Description

This function has exactly the same syntax as bode. The only difference is that it gives graphs with linear frequency scales and Hz as the frequency unit.

See Also

bode, nyquist

freqresp

Purpose

Compute the frequency function for a model.

Syntax

```
H = freqresp(m)
[H,w,covH] = freqresp(m,w)
```

Description

`m` is any `idmodel` or `idfrd` object.

`H = freqresp(m,w)` computes the frequency response `H` of the `idmodel` model `m` at the frequencies specified by the vector `w`. These frequencies should be real and in radians/second.

If `m` has `ny` outputs and `nu` inputs, and `w` contains `Nw` frequencies, the output `H` is a `ny`-by-`nu`-by-`Nw` array such that `H(:,:,k)` gives the complex valued response at the frequency `w(k)`.

For a SISO model, `H(:)` to obtain a vector of the frequency response.

If `w` is not specified, a default choice is made. For a discrete-time model **`w`** will be 128 equally spaced frequency points from 0 (excluded) to the Nyquist frequency. For a continuous-time model, the default is

```
w=logspace(log10(pi/abs(Ts)/100),log10(10*pi/abs(Ts)),128)'
```

where `Ts` is the sampling interval of the data from which the model was estimated. If the model is not estimated, `Ts` is taken as 1, which may make it necessary to specify `w` as in input argument in this case.

```
[H,w,covH] = freqresp(M,w)
```

also returns the frequencies `w` and the covariance `covH` of the response. `covH` is a 5-D array where `covH(ky,ku,k,:,:)` is the 2-by-2 covariance matrix of the response from input `ku` to output `ky` at frequency `w(k)`. The 1,1 element is the variance of the real part, the 2,2 element the variance of the imaginary part and the 1,2 and 2,1 elements the covariance between the real and imaginary parts. `squeeze(covH(ky,ku,k,:,:))` gives the covariance matrix of the corresponding response.

If `m` is a time series (no input channels), `H` is returned as the (power) spectrum of the outputs; an `ny`-by-`ny`-by-`Nw` array. Hence `H(:,:,k)` is the spectrum matrix at frequency `w(k)`. The element `H(k1,k2,k)` is the cross spectrum between outputs `k1` and `k2` at frequency `w(k)`. When `k1=k2`, this is the real-valued power spectrum of output `k1`.

`covH` is then the covariance of the estimated spectrum H, so that `covH(k1,k1,k)` is the variance of the power spectrum estimate of output `k1` at frequency `W(k)`. No information about the variance of the cross spectra is normally given i.e., `covH(k1,k2,k) = 0` for `k1` not equal to `k2`.)

If the model `m` is not a time series, use `freqresp(m('n'))` to obtain the spectrum information of the noise (output disturbance) signals.

Note that `idfrd` computes the same information as `freqresp`, and stores it in the `idfrd` object.

See Also

`bode`, `idfrd`, `nyquist`

fpe

Purpose

Compute the Akaike Final Prediction Error for an estimated model

Syntax

```
am = fpe(Model)
```

Description

`Model` is any estimated `idmodel (idarx, idgrey, idpoly, idss)`.

`am` is returned as the value of the Akaike Final Prediction Error:

$$FPE = V\frac{1 + d/N}{1 - d/N}$$

where V is the loss function, d is the number of estimated parameters and N is the number of estimation data.

See Also

`EstimationInfo, aic`

Reference

Sections 7.4 and 16.4 in Ljung (1999)

Purpose

Access/query idmodel, idfrd, and iddata properties

Syntax

```
Value = get(m,'PropertyName')
get(m)
Struct = get(m)
```

Description

`value = get(m,'PropertyName')` returns the current value of the property `PropertyName` of the `iddata` set or `idfrd`, or `idmodel` (`idgrey`, `idarx`, `idpoly`, `idss`) `m`. The string `'PropertyName'` can be the full property name (e.g., `'SSParameterization'`) an any unambiguous case-insensitive abbreviation thereof (e.g., `'ss'`). You can specify any generic idmodel property or any property specific to `idgrey`, `idarx`, etc. (see `iddata`, `idmodel`, `idgrey`, `idarx`, `idpoly`, `idss`, and Algorithm Properties for lists of properties that can be accessed directly).

`Struct = get(m)` converts the object `m` into a standard MATLAB structure with the property names as field names and the property values as field values.

Without left-hand argument

```
get(m)
```

displays all properties of `m` and their values.

Remark

An alternative to the syntax

```
Value = get(m,'PropertyName')
```

is the structure-like referencing

```
Value = m.PropertyName
```

See Also

`arxdata`, `iddata`, `idfrd`, `idmodel`, `polydata`, `set`, `ssdata`, `tfdata`, `zpkdata`, `Algorithm Properties`, `EstimationInfo`

idarx

Purpose

Construct idarx model from ARX polynomials.

Syntax

```
m = idarx(A,B,Ts)
m = idarx(A,B,Ts,'Property1',Value1,...,,'PropertyN',ValueN)
```

Description

`idarx` creates an object containing parameters that describe the general multi-input, multi-output model structure of ARX type.

$$y(t) + A_1 y(t-1) + A_2 y(t-2) + \ldots + A_{na} y(t-na) =$$

$$B_0 u(t) + B_1 u(t-1) + \ldots + B_{nb} u(t-nb) + e(t)$$

Here A_k and B_k are matrices of dimensions *ny*-by-*ny* and *ny*-by-*nu*, respectively (*ny* is the number of outputs, i.e., the dimension of the vector $y(t)$ and *nu* is the number of inputs). See "Multivariable ARX Models: The idarx Model" on page 3-38 in the "Tutorial" chapter.

The arguments `A` and `B` are 3-D arrays that contain the A matrices and the B matrices of the model in the following way.

A is an `ny-by-ny-by-(na+1)` array such that

```
A(:,:,k+1) = Ak
A(:,:,1) = eye(ny)
```

Similairly B is an `ny-by-nu-by-(nb+1)` array with

```
B(:,:,k+1) = Bk
```

Note that `A` always starts with the identity matrix, and that delays in the model are defined by setting the corresponding leading entries in `B` to zero. For a multivariate time series take `B = []`.

The optional property `NoiseVariance` sets the covariance matrix of the driving noise source $e(t)$ in the model above. The default value is the identity matrix.

The argument Ts is the sampling interval.

The use of idarx is twofold. You can use it to create models that are simulated (using sim) or analyzed (using bode, pzmap, etc.). You can also use it to define initial value models that are further adjusted to data (using arx). The free parameters in the structure are consistent with the structure of A and B, i.e., leading zeros in the rows of B are regarded as fixed delays, and trailing zeros in A and B are regarded as a definition of lower order polynomials. These zeros are fixed, while all other parameters are free.

For a model with one output, ARX models can be descried both as idarx and idpoly models. The internal representation is however different.

idarx Properties

- **A, B**: The A and B polynomials as 3-D arrays, described above
- **dA, dB**: The standard deviations of A and B. Same format as A and B. Cannot be set.
- **na, nb, nk**: The orders and delays of the model. na is a *ny*-by-*ny* matrix whose *i-j* entry is the order of the polynomial corresponding to the *i-j* entry of A. Similarly nb is an *ny*-by-*nu* matrix with the orders of the B. nk is also an *ny*-by-*nu* matrix, whose *i-j* entry is the delay from input *j* to output *i*, that is, the number of leading zeros in the *i-j* entry of B.

In addition to these properties, idarx objects also have all the properties of the idmodel object. See idmodel, Algorithm Properties, and EstimationInfo.

Note that all properties can be set and retrieved either by set/get or by subscripts. Autofill applies to all properties and values, and these are case insensitive.

For a complete list of property values, use get(m). To see possible value assignments, use set(m). See also idprops idarx.

Examples

Simulate a second order ARX model with one input and two outputs, and then estimate a model using the simulated data.

```
A = zeros(2,2,3);
B = zeros(2,1,3)
A(:,:,1) =eye(2);
A(:,:,2) = [-1.5 0.1;-0.2 1.5];
A(:,:,3) = [0.7 -0.3;0.1 0.7];
B(:,:,2) = [1;-1];
B(:,:,3) = [0.5;1.2];
m0 = idarx(A,B,1);
u = iddata([],idinput(300));
e = iddata([],randn(300,2));
y = sim(m0,[u e]);
m = arx([y u],[[2 2;2 2],[2;2],[1;1]]);
```

See Also

`arx`, `arxdata`, `idmodel`, `idpoly`

Purpose

Package input-output data into the `iddata` object

Syntax

```
data = iddata(y,u)
data = iddata(y,u,Ts,'Property1',Value1,...,'PropertyN',ValueN)
```

Description

`iddata` is the basic object for dealing with signals in the toolbox. It is used by most of the commands.

Basic Use

Let `y` be a column vector or an *N*-by-*ny* matrix. The columns of `y` correspond to the different output channels. Similarly, `u` is a column vector or an *N*-by-*nu* matrix containing the signals of the input channels.

```
data = iddata(y,u,Ts)
```

creates an `iddata` object containing these output and input channels. `Ts` is the sampling interval. This construction is sufficient for most purposes.

The data is then plotted by `plot(data)` (see `plot`), and portions of the data record are selected as in `ze = data(1:300)` or `zv = data(501:700)`.

The signals in the output channels are retrieved by `data.OutputData`, or for `short data.y`. Similarly the input signals are obtained by `data.InputData` or `data.u`.

For a time series (no input channels) use `data = iddata(y)`, or let `u = []`.

An `iddata` object can also contain just an input, by letting `y = []`.

The sampling interval can be changed by `set(data,'Ts',0.3)` or, simpler, by `data.Ts = 0.3`.

The input and output channels are given default names like `'y1'`, `'y2'`, `'u1'`,`'u2'`, etc. The channel names can be set by

```
set(data,'InputName',{'Voltage','Current'},'OutputName','Temper
ature')
```

(two inputs and one output is this example) and these names will then follow the object and appear in all plots. The names will also be inherited by models that are estimated from the data.

Similarly, channel units can be specified using the properties `'OutputUnit'` and `'InputUnit'`. These units, when specified, will be used in plots.

The time points associated with the data samples are determined by the sampling interval `Ts` and the time of the first sample, `Tstart`.

```
data.Tstart = 24
```

The actual time point values are given by the property `'SamplingInstants'`, as in

```
plot(data.sa,data.u)
```

for a plot of the input with correct time points. Autofill is used for all properties, and they are case insensitive.

Manipulating Channels

An easy way to set and retrieve channel properties is to use subscripting. The subscripts are defined as

```
data(Samples,Outputs,Inputs)
```

so `dat(:,3,:)` is the data object obtained from `dat` by keeping all input channels, but only output channel 3. (Trailing `':'`s can be omitted so `dat(:,3,:)= dat(:,3).`).

The channels can also be retrieved by their names, so that

```
dat(:,{'speed','flow'},[])
```

is the data object where the indicated output channels have been selected and no input channels are selected.

Moreover

```
dat1(101:200,[3 4],[1 3]) = dat2(1001:1100,[1 2],[6 7])
```

will change samples 101 to 200 of output channels 3 and 4 and input channels 1 and 3 in the `iddata` object `DAT1` to the indicated values from `iddata` object `DAT2`. The names and units of these channels will then also be changed accordingly.

To add new channels, use horizontal concatenation of `iddata` objects:

```
dat =[dat1, dat2];
```

(see “horizontal concatenation” below) or add the data record directly. Thus

```
dat.u(:,5) = U
```

will add a fifth input to dat.

Nonequal Sampling

The property 'SamplingInstants' gives the sampling instants of the data points. It can always be retrieved by get(DAT,'SamplingInstants') (or dat.s) and is then computed from dat.Ts and dat.Tstart. 'SamplingInstants' can also be set to an arbitrary vector of the same length as the data, so that nonequal sampling can be handled. Ts is then automatically set to [ ]. Most of the estimation routines, though, do not handle unequally sampled data.

Multiple Experiments

The iddata object can also store data from separate experiments. The property 'ExperimentName' is used to separate the experiments. The number of data as well as the sampling properties can vary from experiment to experiment, but the input and output channels must be the same. (Use NaN to fill possibly unmeasured channels in certain experiments.) The data records will be cell arrays, where the cells contain data from each experiment.

Multiple experiments can be defined directly by letting the 'y' and 'u' properties as well as 'Ts' and 'Tstart' be cell arrays.

It is normally easier to create multi-experiment data by merging experiments as in

```
dat = merge(dat1,dat2).
```

See the reference page for merge (data). Storing multiple experiments as one iddata object may be very useful to handle experimental data that have been collected on different occasions, or when a data set has been split up to remove “bad” portions of the data. All the toolbox’s routines accept multiple experiment data.

Experiments can be set and retrieved by subscripting with curly brackets: DAT{3} is experiment number 3 and dat{'Day1','Day4'} retrieves the two experiments with the indicated names.

The subscripting can be combined: `dat{3}(1:100,[2,3],[4:8])` gives the 100 first samples of output channels 2 and 3 and input channels 4 to 8 of experiment number 3. It can also be used for subassignment, like

```
dat{'Run4'} = dat2
```

which adds the data in `dat2` as a new experiment with name `'Run4'`. See `iddemo` number 8 for an illustration of how multiple experiments can be used.

iddata Properties

In the list below, `N` denotes the number of samples of the signals, `ny` the number of output channels, `nu` the number of input channels, and `Ne` the number of experiments.

- **`Domain`**: Assumes the values `'Time'` or `'Frequency'` and denotes whether the data are time domain or frequency domain data.
- **`Name`**: An optional name for the data set. An arbitrary string.
- **`OutputData, InputData`**: The data matrices `y` and `u`. In the single experiment case `y` is an `N`-by-`ny` matrix and `u` is an `N`-by-`nu` matrix. For multiple experiments `y` and `u` are 1-by-`Ne` cell arrays, with each cell containing the data for the different experiments.
- **`OutputName, InputName`**: Cell arrays of length `ny`-by-1 and `nu`-by-1 containing the names of the output and input channels. If not specified, default names, `{'y1';'y2';...}` and `{'u1';'u2';...}` are given.
- **`OutputUnit, InputUnit`**: Cell arrays of length `ny`-by-1 and `nu`-by-1 containing the units of the output and input channels.
- **`TimeUnit`**: The unit for the sampling instants.
- **`Ts:`** Sampling interval. A positive scalar. For multiexperiment data, `Ts` is a 1-by-`Ne` cell array, with each cell containing the sampling interval of the corresponding experiment. For nonequally sampled data, `Ts = []`.
- **`Tstart`**: The starting time of the data record. A scalar. For multiexperiment data, `Tstart` is a 1-by-`Ne` cell array, with each cell containing the starting time for the corresponding experiment
- **`SamplingInstants`**: The time values of the sample points. A `N`-by-1 vector. For multiple experiment data, `SamplingInstants` is a 1-by-`Ne` cell array, with each cell containing the sampling instants of the corresponding experiment. For equally sampled data, `SamplingInstants` is generated from `Ts` and `Tstart`.

- **Period**: The period of the input. A nu-by-1 vector, where the k-th entry contains the period of the k-th input. `Period = inf` means nonperiodic data. For multiexperiment data, `Period` is a 1-by-Ne cell array with each cell containing the period(s) for the input of the corresponding experiment.
- **InterSample**: Describes the intersample behavior of the input channels. An nu-by-1 cell array where the `(k,1)` element is `'zoh'`, `'foh'`, or `'bl'`, denoting that input number k is piecewise constant, piecewise linear, or band limited. For multiple experiment data, `InterSample` is an nu-by-Ne cell array.
- **ExperimentName**: A string containing the name of the experiment. For multiple experiment data `ExperimentName` is a 1-by-Ne cell array with each cell containing the name of the corresponding experiment. It can be freely set, and is by default given names `{'Exp1', 'Exp2',...}`.
- **Notes**: An arbitrary field to store extra information and notes about the object.
- **UserData**: An arbitrary field for any possible use.

Note that all properties can be set or retrieved either by `set/get` or by subscripts. Autofill applies to all properties and values, and are case insensitive:. `'y'` and `'u'` can be used as short for `'OutputData'` and `'InputData'`. `'y'` and `'u'` can also replace `'Output'` and `'Input'` in the other properties.

```
data.y=randn(100,2)
data.una = 'Voltage'
set(data,'tim','minute')
p = data.per
```

For a complete list of property values, use `get(data)`. To see possible value assignments, use `set(data)`.

Subreferencing

The samples, outputs and input channels can be referenced according to

```
data(samples,outputs,inputs)
```

Use colon (:) to denote all samples/channels and the empty matrix ([ ]) to denote no samples/channels. The channels can be referenced by number or by name. For several names, a cell array must be used.

```
dat2 = dat(:,'y3',{'u1','u4'})
dat2 = dat(:,3,[1 4])
```

Logical expressions will also work:

```
dat3 = dat2(dat2.sa>1.27&dat2.sa<9.3)
```

will select the samples with time marks between 1.27 and 9.3.

Subreferencing with curly brackets refers to the experiment:

```
data{Experiment}(samples,outputs,inputs)
```

Any subreferenced variable can also be assigned

```
data{'Exp3'}.y = flow(1:700,:)
data(1:10,1,1) = dat1(101:110,2,3)
```

Horizontal Concatenation

```
dat = [dat1,dat2,...,datN]
```

creates an `iddata` object `dat`, consisting of the input and output channels in `dat1,... datN`. Default channel names (`'u1', 'u2', 'y1', 'y2'`, etc.) will be changed so that overlaps in names are avoided, and the new channels will be added.

If `datk` contains channels with user specified names, that are already present in the channels of `Datj, j<k`, these new channels will be ignored.

Vertical Concatenation

```
dat = [dat1;dat2;... ;datN]
```

creates an `iddata` object `dat` whose signals are obtained by stacking those of `datk` on top of each other. That is

```
dat.OutputData = [dat1.Ouputdata;dat2.OutputData; ...
datN.OutputData]
```

and similarly for the inputs. The `datk` objects must all have the same number of channels and experiments.

Online Help Functions

See `help iddata`, `idprops iddata`, `help iddata/subsref`, `help iddata/subsasgn`, `help iddata/horzcat`, and `help iddata/vertcat`.

See Also

`plot`, `size`

Purpose

Open the graphical user interface

Syntax

```
ident
ident(session,directory)
```

Description

`ident` by itself opens the main interface window, or brings it forward if it is already open.

`session` is the name of a previous session with the graphical interface, and typically has extension `.sid`. `directory` is the complete path for the location of this file. If the session file is on the `MATLABPATH`, `directory` can be omitted.

When the session is specified, the interface will open with this session active. Typing `ident(session,directory)` on the MATLAB command line, when the interface is active, will load and open the session in question.

For more information about the graphical user interface, see Chapter 2, “The Graphical User Interface.”

Examples

```
ident('iddata1.sid')
ident('mydata.sid','\matlab\data\cdplayer\')
```

idfilt

Purpose

Filter data using general filters or Butterworth filters.

Syntax

```
zf = idfilt(z,filter)
zf = idfilt(z,ord,Wn)
zf = idfilt(z,ord,causality)
[zf,mf] = idfilt(z,ord,Wn,hs)
```

Description

`z` is the data, defined as an `iddata` object. `zf` contains the filtered data as an `iddata` object. The filter can be defined in two ways:

- As an explicit argument `filter`. This in turn can be given either as any SISO `idmodel` or LTI model object, or as a cell array `{A,B,C,D}` of SISO state-space matrices or as a cell array `{num,den}` of numerator/denominator filter coefficients.
- As a triplet of arguments `...,ord,Wn,hs,...`, which defines a Butterworth filter of order `ord`. If `hs` is not specified and `Wn` contains just one element, a low pass filter with cutoff frequency `Wn` (measured as a fraction of the Nyquist frequency) is obtained. If `hs =' high'` a high pass filter with this cutoff frequency is obtained instead. If `Wn = [Wnl Wnh]` is a vector with two elements, a filter (of order `2*ord`) with passband between `Wnl` and `Wnh` is obtained is `hs` is not specified. If `hs = 'stop'` a bandstop filter with stop band between these two frequencies is obtained instead.

The output argument `mf` is the filter given as an `idmodel` object.

With `causality = 'causal'` (default) causal filtering is used. With `causality = 'noncausal'`, a noncausal, zero-phase filter is used for the filtering.

It is common practice in identification to select a frequency band where the fit between model and data is concentrated. Often this corresponds to bandpass filtering with a pass band over the interesting breakpoints in a Bode diagram. For identification where a disturbance model also is estimated, it is better to achieve the desired estimation result by using the property `'Focus'` (see `Algorithm Properties`) than just to prefilter the data.

Algorithm

The Butterworth filter is the same as `butter` in the Signal Processing Toolbox. Also, the zero-phase filter is equivalent to `filtfilt` in that toolbox.

References

Ljung (1999), Chapter 14.

idfrd

Purpose

Create the `idfrd` (Identified Frequency Response Data) object that stores frequency function and spectrum data along with covariance information.

Syntax

```
h = idfrd(Response,Freqs,Ts)
h = idfrd(Response,Freqs,Ts,'CovarianceData',Covariance, ...
   'SpectrumData',Spec,'NoiseCovariance',Speccov,'property1', ...
    Value1,'PropertyN',ValueN)
h = idfrd(mod)
h = idfrd(mod,Freqs)
```

Description

`idfrd` creates the `idfrd` model object.

For a model

$$y(t) = G(q)u(t) + H(q)e(t)$$

`it stores` the transfer function estimate G (see equation (Equation 3-4) an onwards in the *Tutorial*)

$$G(e^{i\omega})$$

as well as the spectrum of the additive noise (Φ_v) at the outputs

$$\Phi_v(\omega) = \lambda T \left| H(e^{i\omega T}) \right|^2$$

where λ is the estimated variance of *e(t)*, and T is the sampling interval.

Creating idfrd from given responses.

`Response` is a 3-D array of dimension `ny-by-nu-by-Nf` with `ny` being the number of outputs, `nu` the number of inputs, and `Nf` the number of frequencies (i.e., the length of `Freqs`). `Response(ky,ku,kf)` is thus the complex-valued frequency response from input `ku` to output `ky` at frequency ω=`Freqs(kf)`. When defining the response of a SISO system, `Response` can be given as a vector.

`Freqs` is a column vector of length `Nf` containing the frequencies of the response.

`Ts` is the sampling interval. `T = 0` means a continuous time model.

`Covariance` is a 5-D array containing the covariance of the frequency response. It has dimension `ny-by-nu-by-Nf-by-2-by-2`. The structure is such that

`Covariance(ky,ku,kf,:,:)` is the 2-by-2 covariance matrix of the response `Response(ky,ku,kf)`. The 1-1 element is the variance of the real part, the 2-2 element is the variance of the imaginary part and the 1-2 and 2-1 elements is the covariance between the real and imaginary parts. `squeeze(Covariance(ky,ku,kf,:,:))` thus gives the covariance matrix of the corresponding response.

The information about spectrum is optional. The format is as follows:

`spec` is a 3-D array of dimension `ny-by-ny-by-Nf`, such that `spec(ky1,ky2,kf)` is the cross spectrum between the noise at output `ky1` and the noise at output `ky2`, at frequency `Freqs(kf)`. When `ky1=ky2` the (power) spectrum of the noise at output `ky1` is thus obtained. For a single output model, `spec` can be given as a vector.

`speccov` is a 3-D array of dimension `ny-by-ny-by-Nf`, such that `speccov(ky1,ky1,kf)` is the variance of the corresponding power spectrum. Normally, no information is included about the covariance of the non-diagonal spectrum elements.

If only `SpectrumData` is to be packaged in the `idfrd` object, set `Response = []`.

Creating idfrd from a given model.

`idfrd` can also be computed from a given model `mod` (defined as any `idmodel` object).

The default values of the frequencies in the discrete-time case are

```
Freqs = [1:128]'/128*pi/Ts
```

where `Ts` is the sampling interval specified by `mod` and for the continuous-time case

```
Freqs = logspace(log10(pi/Ts/100),log10(10*pi/ Ts),128)
```

where `Ts` is the sampling interval of the data from which the model was estimated. If the model is not estimated, a simple default choice of `Freqs` is made. In this case it may be necessary to supply the argument `Freqs` explicitly.

If `mod` has `InputDelay` different from zero, these are appended as phase lags, and `h` will then have an `InputDelay` 0.

The estimated covariances are computed using the Gauss approximation formula from the uncertainty information in `mod`. For models with complicated parameter dependencies, numerical differentiation is applied. The step-sizes for the numerical derivatives are determined by `nuderst`.

Frequency responses for submodels can be obtained by the standard subreferencing: `h = idfrd(m(2,3))`. See `idmodel`. In particular, `h= idfrf(m('measured'))` gives `h` that just contains the `ResponseData (G)` and no spectra. Also `h = idfrd(m('noise'))` gives a `h` that just contains `SpectrumData`.

The `idfrd` models can be graphed with `bode`, `ffplot`, and `nyquist`, which all accept mixtures of `idmodel` and `idfrd` models as arguments. Note that `spa` and `etfe` return their estimation results as `idfrd` objects.

idfrd Properties

To summarize the properties of `idfrd`:

- **`ResponseData`**: A 3-D array of the complex-valued frequency response as described above. For SISO system use `Response(1,1,:)` to obtain a vector of the response data.
- **`Frequency`**: A column vector containing the frequencies as which the responses are defined.
- **`CovarianceData`**: A 5-D array of the covariance matrices of the response data as described above.
- **`SpectrumData;`** A 3-D array containing power spectra and cross spectra of the output disturbances (noise) of the system.
- **`NoiseCovariance:`** A 3-D array containing the variances of the power spectra, as explained above.
- **`Units:`** the unit of the frequency vector. Can assume the values `'rad/s'` and `'Hz'`.
- **`Ts:`** A scalar denoting the sampling interval of the model whose frequency response is stored. `'Ts' = 0` means a continuous-time model.
- **`Name:`** An optional name for the object
- **`InputName:`** A string or a cell array containing the names of the input channels. It has as many entries as there are input channels.
- **`OutputName:`** Correspondingly for the output channels.

- **InputUnit:** The units in which the input channels are measured. It has the same format as `'InputName'`.
- **OutputUnit:** Correspondingly for the output channels.
- **InputDelay**: A row vector of length equal to the number of input channels. Contains the delays from the input channels. These should thus be appended as phase lags when the response is calculated. This is done automatically by `freqresp`, `bode`, `ffplot`, and `nyquist`. Note that if the `idfrd` is calculated form an `idmodel`, possible input delays in that model are converted to phase lags, and `InputDelay` of the `idfrd` model is set to zero.
- **Notes:** An arbitrary field to store extra information and notes about the object.
- **UserData:** An arbitrary field for any possible use.
- **EstimationInfo:** A structure that contains information about the estimation process that is behind the frequency data. It contains the following fields:
  - `Status`: Gives the status of the model, e.g., `'Not estimated'`.
  - `Method`: The identification routine that created the model.
  - `WindowSize`: If the model was estimated by `spa` or `etfe`, the size of window (input argument `M`) that was used.
  - `DataName`: The name of the data set from which the model was estimated.
  - `DataLength`: The length of this data set.

Note that all properties can be set or retrieved either by `set/get` or by subscripts. Autofill applies to all properties and values, and these are case insensitive:

```
h.ts = 0
loglog(h.fre,squeeze(h.spe(2,2,:)))
```

For a complete list of property values, use `get(m)`. To see possible value assignments, use `set(m)`. See also `idprops idfrd`.

Subreferencing

The different channels of the `idfrd` are retrieved by subreferencing.

```
h(outputs,inputs)
```

`h(2,3)` thus contains the response data from input channel 3 to output channel 2, and, if applicable, the output spectrum data for output channel 2. The channels can also be referred to by their names: `h('power',{'voltage',''speed'})`.

```
h('m')
```

contains the information for measured inputs only, that is, just `ResponseData`, while

```
h('n')
```

(`'n'` for `'noise'`) just contains `SpectrumData`.

Horizontal Concatenation

Adding input channels

```
h = [h1,h2,...,hN]
```

creates an `idfrd` model `h`, with `ResponseData` containing of all the input channels in `h1, ... hN`. The output channels of `hk` must be the same as well as the frequency vectors. `SpectrumData` will be ignored.

Vertical Concatenation

Adding output channels

```
h = [h1;h2;... ;hN]
```

creates an `idfrd` model `h` with `ResponseData` containing all the output channels in `h1, h2,...,hN`. The input channels of `hk` must all be the same, as well as the frequency vectors. `SpectrumData` will also be appended for the new outputs. The cross spectrum between output channels will then be set to zero.

Examples

Compare the results from spectral analysis and an ARMAX model.

```
m = armax(z,[2 2 2 1]);
g = spa(z)
bode(g,m)
```

Compute separate `idfrd` models, one containing `g` and the other the noise spectrum.

```
g = idfrd(m('m'))
phi = idfrd(m('n'))
```

See Also

`bode`, `etfe`, `ffplot`, `freqresp`, `nyquist`, `spa`

idgrey

Purpose

Package a greybox model structure defined by a user-written M-file into an idgrey model.

Syntax

```
m = idgrey(MfileName,ParameterVector,CDmfile)
m = idgrey(MfileName,ParameterVector,CDmfile,FileArgument,Ts,...
'Property1',Value1,...,'PropertyN',ValueN)
```

Description

The function idgrey is used to create arbitrarily parameterized state-space models as idgrey objects.

MfileName is the name of an M-file that defines how the state-space matrices depend on the parameters to be estimated. The format of this M-file is given by

```
[A,B,C,D,K,X0] = mymfile(pars,Tsm,Auxarg)
```

and is further discussed below.

ParameterVector is a column vector of the nominal/initial parameters. Its length must be equal to the number of free parameters in the model (that is, the argument pars in the example below).

The argument CDmfile describes how the user-written M-file handles continuous/discrete-time model. It takes the following values:

- CDmfile = 'cd': The M-file returns the continuous-time state-space matrices when called with the argument Tsm=0. When called with a value Tsm>0 the M-file returns the discrete-time state-space matrices, obtained by sampling the continuous-time system with sampling interval Tsm. The M-file must consequently in this case include the sampling procedure.
- CDmfile = 'c'. The M-file always returns the continuous-time state-space matrices, no matter the value of Tsm. In this case the toolbox's estimation routines will provide the sampling when fitting the model to discrete-time data.
- CDmfile='d'. The M-file always returns discrete-time state-space matrices, that may or may not depend on Tsm.

The argument FileArgument corresponds to the auxiliary argument Auxarg in the user-written M-file. It can be used to handle several variants of the model

structure, without having to edit the M-file. If it is not used, enter `FileArgument = []`. (Default).

`Ts` denotes the sampling interval of the model. Its default value is `Ts = 0`, that is, a continuous-time model.

The `idgrey` object is a child of `idmodel`. Therefore any `idmodel` properties can be set as property name/property value pairs in the `idgrey` command. They can of course also be set by the command `set`, or by subassignment, like

```
m.InputName = {'speed','voltage'}
m.FileArgument = 0.23
```

There are also two properties, `DisturbanceModel` and `InitialState` that can be used to affect the parameterizations of *K* and *X0*, thus overriding the outputs from the M-file. See below.

idgrey Properties

To summarize, the properties of `idgrey` are the following ones:

- **`MfileName:`** The name of the user-written M-file. See below for details
- **`CDmfile:`** How this file handles continuous/discrete models, depending on its second argument `T`.
  - `CDmfile = 'cd'` means that the mfile returns the continuous time state space model matrices when the argument T = 0, and the discrete time model, obtained by sampling with sampling interval T when T > 0.
  - `CDmfile= 'c'` means that the mfile always returns continuous time model matrices, no matter the value of T.
  - `CDmfile = 'd'` means that the mfile always returns discrete time model matrices that may or may not depend on the value of T.
- **`FileArgument:`** Possible extra input arguments to the user-written M-file
- **`DisturbanceModel`**: affects the parameterization of the K-matrix. It can assume the following values:
  - `'Model'`: This is the default. It means that the K matrix obtained from the user-written M-file is used.
  - `'Estimate'`: The K-matrix is treated as unknown and its elements are estimated as free parameters.
  - `'Fixed'`: The K-matrix is fixed to a given value.

- `'Zero'`: The K-matrix is fixed to zero, thus producing an output error model.

Note that in the three last cases the output `K` from the user written M-file is ignored. The estimated/fixed value is stored internally and does not change when the model is sampled/resampled/or converted to continuous time. Note also that this estimated value is tailored only to the sampling interval of the data.

- **`InitialState`**: affects the parameterization of the X0-vector. It assumes the same values as DisturbanceModel, with analogous interpretations. In addition `InitialState` can assume the value '`Backcast`', with the same interpretation as for an `idss` object.
- **`A, B, C, D, K,`** and **`X0`**: The state-space matrices. For `idgrey` models only `'K'` and `'X0'` can be set, the others only retrieved. The set `'K'` and `'X0'` are relevant only when `DisturbanceModel/InitialState` are `Estimate` or `Fixed`.
- **`dA, dB, dC, dD, dK,`** and **`dX0`**: The estimated standard deviations of the state-space matrices. These cannot be set, only retrieved.

In addition, any `idgrey` object also has all the properties of `idmodel`. See `Algorithm Properties` and the reference page for `idmodel`.

Note that all properties can be `set/get` either by these commands or by subscripts. Autofill applies to all properties and values, and are case insensitive:

```
m.fi = 10;
set(m,'search','gn')
p = roots(m.a)
```

For a complete list of property values, use `get(m)`. To see possible value assignments, use `set(m)`. See also `idprops idgrey`.

M-File Details

The model structure corresponds to the general linear state-space structure

$$\tilde{x}(t) = A(\theta)x(t) + B(\theta)u(t) + K(\theta)e(t)$$

$$x(0) = x_0(\theta)$$

$$y(t) = C(\theta)x(t) + D(\theta)u(t) + e(t)$$

Here $\tilde{x}(t)$ is the time derivative $\dot{x}(t)$ for a continuous time model and $x(t + Ts)$ for a discrete time model.

The matrices in this time-discrete model can be parameterized in an arbitrary way by the vector θ. Write the format for the M-file as follows:

```
[A,B,C,D,K,x0] = mymfile(pars,T,Auxarg)
```

Here the vector `pars` contains the parameters θ, and the output arguments `A`, `B`, `C`, `D`, `K`, and `x0` are the matrices in the model description that correspond to this value of the parameters and this value of the sampling interval `T`.

`T` is the sampling interval, and `Auxarg` is any variable of auxiliary variables with which you want to work. (In that way you can change certain constants and other aspects in the model structure without having to edit the M-file.) Note that the two arguments `T` and `Auxarg` must be included in the function head of the M-file, even if they are not utilized within the M-file.

Section "State-Space Models with Coupled Parameters: the idgrey Model" on page 3-45 of the *Tutorial* contains several examples of typical M-files that define model structures.

A comment about **`CDmfile`**: If a continuous time model is sought, it is of course most easy to let the mfile deliver just the continuous time model, i.e., have `CDmfile = 'c'`, and rely upon the Toolbox's routines for the proper sampling. Similarly, if the underlying parameterization is indeed discrete time, it is natural to deliver the discrete time model matrices, and let `CDmfile = 'd'`. If the underlying parameterization is continuous, but you prefer for some reason to do your own sampling inside the mfile, in accordance with the value of T, then it is preferable to let your mfile deliver the continuous time model when called with `T = 0`, that is, the alternative `CMmfile = 'cd'`. This will avoid sampling, and then transforming back (using `d2c`) to find the continuous time model.

Examples

Use the M-file `'mynoise'` given in "Example 3.4: Parametrized Disturbance Models" on page 3-48 in the *Tutorial* to obtain a physical parametrization of the Kalman gain.

```
mn = idgrey('mynoise',[0.1,-2,1,3,0.2]','d')
m = pem(z,mn)
```

Purpose

Generate signals, typically to be used as inputs for identification.

Syntax

```
u = idinput(N)
u = idinput(N,type,band,levels)
[u,freqs] = idinput(N,'sine',band,levels,sinedata)
```

Description

`idinput` generates input signals of different kinds, that are typically used for identification purposes. `u` is returned as a matrix or column vector.

For further use in the toolbox it is recommended to create an `iddata` object from `u`, indicating sampling time, input names, periodicity, and so on: `u = iddata([],u);`

`N` determines the number of generated input data. If `N` is a scalar, `u` will be a column vector with this number of rows.

`N = [N nu]` gives an input with `nu` input channels each of length `N`.

`N = [P nu M]` gives a periodic input with `nu` channels, each of length `M*P` and periodic with period `P`.

Default is `nu=1` and `M =1`.

`type` defines the type of input signal to be generated. This argument takes one of the following values:

- `type = 'rgs'` : This gives a random, Gaussian signal.
- `type = 'rbs'`: This gives a random, binary signal.
- `type = 'prbs'`: This gives a pseudo-random, binary signal.
- `type = 'sine'`: This gives a signal which is a sum of sinusoids.

Default is `type = 'rbs'`.

The frequency contents of the signal is determined by the argument `band`. For the choices `type = 'rs'`, `'rbs'`, and `'sine'`, this argument is a row-vector with two entries

```
band = [wlow, whigh]
```

that determine the lower and upper bound of the pass-band. The frequencies `wlow` and `whigh` are expressed in fractions of the Nyquist frequency. A white noise character input is thus obtained for `band = [0 1]`, which also is the default value.

For the choice `type = 'prbs'`

```
band = [0, B]
```

where `B` is such that the signal is constant over intervals of length `1/B (the clock period)`. Also in this case the default is `band = [0 1]`.

The argument `levels` defines the input level. It is a row vector

```
levels = [minu, maxu]
```

such that the signal u will always be between the values `minu` and `maxu` for the choices `type = 'rbs'`, `'prbs'` and `'sine'`. For `type = 'rgs'`, the signal level is such that `minu` is the mean value of the signal, minus one standard deviation, while `maxu` is the mean value plus one standard deviation. Gaussian white noise with zero mean and variance one is thus obtained for `levels = [-1, 1]`, which is also the default value.

In the `'sine'` case, the sinusoids are chosen from the frequency grid

`freq = 2*pi*[1:Grid_Skip:fix(P/2)]/P` intersected with `pi*[band(1) band(2)]`.

(for `Grid_Skip`, see below.) For multi-input signals, the different inputs use different frequencies from this grid. An integer number of full periods is always delivered. The selected frequencies are obtained as the second output argument, `freqs`, where row `ku` of `freqs` contains the frequencies of input number `ku`. The resulting signal is affected by a fifth input argument `sinedata`

```
sinedata = [No_of_Sinusoids, No_of_Trials, Grid_Skip],
```

meaning that `No_of_Sinusoids` is equally spread over the indicated `band`. `No_of_Trials` (different, random, relative phases) are tried until the lowest amplitude signal is found.

```
Default: sinedata = [10,10,1];
```

`Grid-skip` may be useful for controlling odd and even frequency multiples, e.g., to detect nonlinearities of various kinds.

Algorithm

Very simple algorithms are used. The frequency contents is achieved for `'rgs'` by an eighth order Butterworth, noncausal filter, using `idfilt`. This is quite reliable. The same filter is used for the `'rbs'` case, before making the signal

binary. This means that the frequency contents is not guaranteed to be precise in this case.

For the `'sine'` case, the frequencies are selected to be equally spread over the chosen grid, and each sinusoid is given a random phase. A number of trials are made, and the phases that give the smallest signal amplitude are selected. The amplitude is then scaled so as to satisfy the specifications of `levels`.

Reference

See Söderström and Stoica (1989), Chapter C5.3. For a general discussion of input signals, see Ljung (1999), Section 13.3.

idmodel

Purpose

Package all common model properties.

Description

`idmodel` is an object that the user does not deal with directly. It contains all the common properties of the model objects `idarx`, `idgrey`, `idpoly`, and `idss`, which are returned by the different estimation routines.

Basic Use

If you just estimate models from data, the model objects should be transparent. All parametric estimation routines return `idmodel` results:

```
m = arx(Data,[2 2 1])
```

The model `m` contains all relevant information. Just typing `m` will give a brief account of the model. `present(m)` also gives information about the uncertainties of the estimated parameters. `get(m)` gives a complete list of model properties.

Most of the interesting properties can be directly accessed by subreferencing:

```
m.a
m.da
```

See the property list obtained by `get(m)`, as well as the property lists of `idgrey`, `idarx`, `idpoly`, and `idss` in the "Command Reference" for more details on this. See also `help idprops`.

The characteristics of the model `m` can be directly examined and displayed by commands like `impulse`, `step`, `bode`, `nyquist`, `pzmap`. The quality of the model is assessed by commands like `compare`, and `resid`. If you have the Control System Toolbox, just typing `view(m)` gives access to various display functions.

To extract state-space matrices, transfer function polynomials, etc., use the commands

```
arxdata, polydata, tfdata, ssdata, zpkdata
```

and by `idfrd` and `freqresp`, the frequency response of the model can be computed.

Creating and Modifying Model Objects

If you want to define a model to use, e.g., for simulating data, you need to use the model creator functions:

- idarx, for multivariable arx models
- idgrey, for user-defined greybox state-space models
- idpoly, for single output polynomial models
- idss, for state-space models

Also, if you want to estimate a state-space model with a specific internal parameterization, you need to create an idss model or a idgrey model. See the respective reference pages for these functions.

Dealing with Input and Output Channels

For multivariable models, you construct submodels containing a subset of inputs and outputs by simple subreferencing. The outputs and input channels can be referenced according to

```
m(outputs,inputs)
```

Use colon (:) to denote all channels and the empty matrix ([ ]) to denote no channels. The channels can be referenced by number or by name. For several names, a cell array must be used.

```
m3 = m('position',{'power','speed'})
```

or

```
m3 = m(3,[1 4])
```

Thus m3 is the model obtained from m by looking at the transfer functions from input numbers 1 and 4 (with input names 'power' and 'speed') to output number 3 (with name position)

For a single output model m

```
m4 = m(inputs)
```

will select the corresponding input channels, and for a single input model

```
m5 = m(outputs)
```

will select the indicated output channels.

Subreferencing is quite useful; e.g., when a plot of just some channels is desired.

The Noise Channels

The estimated models have two kinds of input channels: the measured inputs u and the noise inputs e. For a general linear model `m`, we have

$$y(t) = G(q)u(t) + H(q)e(t) \qquad \textbf{(4-1)}$$

where u is the nu-dimensional vector of measured input channels and e is the ny-dimensional vector of noise channels. The covariance matrix of e is given by the property `'NoiseVariance'`. Occasionally this matrix Λ will be written in factored form

$$\Lambda = LL^T$$

This means that e can be written as

$$e = Lv$$

where v is white noise with identity covariance matrix (independent noise sources with unit variances).

If `m` is a time series ($nu = 0$), G is empty and the model is given by

$$y(t) = H(q)e(t)$$

For the model `m`, the restriction to the transfer function matrix G is obtained by

`m1 = m('measured')` or just `m1 = m('m')`

Then e is set to `0` and H is removed.

Analogously

`m2 = m('noise')` or just `m2 = m('n')`

creates a time-series model `m2` from `m` by ignoring the measured input. That is, `m2` describes the signal He.

For a system with measured inputs, `bode`, `step`, and many other transformation and display functions just deal with the transfer function matrix G. To obtain or graph the properties of the disturbance model H, it is therefore important to make the transformations `m('n')`. For example,

```
bode(m('n'))
```

will plot the additive noise spectra according to the model `m`, while

```
bode(m)
```

just plots the frequency responses of G.

To study the noise contributions in more detail, it is useful to convert the noise channels to measured channels, using the command `noisecnv`:

```
m3 = noisecnv(m)
```

This creates a model `m3` with all input channels, both measured `u` and noise sources `e`, being treated as measured signals,. That is, `m3` is a model from `u` and `e` to `y`, describing the transfer functions G and H. The information about the variance of the innovations `e` is then lost. For example, studying the step response from the noise channels, will then not take into consideration how large the noise contributions actually are.

To include that information, e should first be normalized $e = Lv$, so that v becomes white noise with an identity covariance matrix:

```
m4 = noisecnv(m,'Norm')
```

This will create a model `m4` with u and v treated as measured signals:

$$y(t) = G(q)u(t) + H(q)Lv(t) = \begin{bmatrix} G & HL \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

For example, the step responses from v to y will now also reflect the typical size of the disturbance influence, due to the scaling by L. In both these cases, the previous noise sources, that have become regular inputs will automatically get input names that are related to the corresponding output. The unnormalized noise sources `e` have names like `'e@ynam1'` (noise `e` at output channel `ynam1`), while the normalized sources `v` are called `'v@ynam1'`.

Retrieving Transfer Functions

The functions that retrieve transfer function properties, `ssdata`, `tfdata`, and `zpkdata` will thus work as follows for a model (Equation 4-1) with measured inputs: (`fcn` is any of `ssdata`, `tfdata`, or `zpkdata`)

`fcn(m)` returns the properties of G (ny outputs and nu inputs)

`fcn(m('n'))` returns the properties of the transfer function H (ny outputs and ny inputs)

`fcn(noiseconv(m,'Norm'))` returns the properties of the transfer function [*G HL*} (*ny* outputs and *ny*+*nu* inputs). Analogously

```
m1 = m('n'). fcn(noisecnv(m1,'Norm'))
```

returns the properties of the transfer function *HL* (*ny* outputs and *ny* inputs).

If `m` is a time series model, `fcn(m)` returns the properties of *H*, while

```
fcn(noisecnv(m,'Norm'))
```

returns the properties of *HL*.

Note that the estimated covariance matrix `NoiseVariance` itself is uncertain. This means that the uncertainty information about *H* is different from that of *HL*.

idmodel Properties

In the list below, ny is the number of output channels, and nu is the number of input channels:

- **`Name:`** An optional name for the data set. An arbitrary string.
- **`OutputName, InputName:`** Cell arrays of length ny-by-1 and nu-by-1 containing the names of the output and input channels. For estimated models, these are inherited from the data. If not specified, they will be given default names: `{'y1','y2',...}` and `{'u1','u2',...}`.
- **`OutputUnit, InputUnit:`** Cell arrays of length ny-by-1 and nu-by-1 containing the units of the output and input channels. Inherited from data for estimated models.
- **`TimeUnit:`** The unit for the sampling interval.
- **`Ts:`** Sampling interval. A non-negative scalar. `Ts = 0` denotes a continuous-time model. Note that changing just `Ts` will not recompute the model parameters. Use `c2d` and `d2c` for recomputing the model to other sampling intervals.
- **`ParameterVector:`** The vector of adjustable parameters in the model structure. Initial/nominal values or estimated values, depending on the status of the model. A column vector.
- **`PName:`** The names of the parameters. A cell array of the length of the parameter vector. If not specified, it will contain empty strings. See also `setpname`.

- **`CovarianceMatrix:`** The estimated covariance matrix of the parameter vector. For a nonestimated model this is the empty matrix. For state-space models in the `'Free'` parameterization the covariance matrix is also the empty matrix, since the individual matrix elements are not identifiable then. Instead, in this case, the covariance information is hidden (in the hidden property `'Utility'`) and retrieved by the relevant functions when necessary. Setting `CovarianceMatrix` to `'None'` will inhibit calculation of covariance and uncertainty information. This may save substantial time for certain models. See “No Covariance” on page 3-93 in the “Tutorial” chapter.
- **`NoiseVariance:`** The covariance matrix of the noise source e. An ny-by-ny matrix.
- **`InputDelay:`** A vector of size nu-by-1, containing the input delay from each input channel. For a continuous-time model (`Ts = 0`) the delay is measured in `TimeUnit`, while for discrete-time models (`Ts > 0`) the delay is measured as the number of samples. Note the difference between `InputDelay` and `nk` (which is a property of `idarx`, `idss`, and `idpoly`). `'Nk'` is a model structure property that tells the model structure to include such an input delay. In that case, the corresponding state-space matrices and polynomials will explicitly contain `Nk` input delays. The property `InputDelay`, on the other hand, is an information that in addition to the model as defined, the inputs should be shifted by the given amount. `InputDelay` is used by `sim` and the estimation routines to shift the input data. When computing frequency responses, the InputDelay is also respected. Note that `InputDelay` can be both positive and negative.
- **`Algorithm:`** See the reference page for `Algorithm Properties`.
- **`EstimationInfo:`** See the reference page for `EstimationInfo`.
- **`Notes:`** An arbitrary field to store extra information and notes about the object.
- **`UserData:`** An arbitrary field for any possible use.

Note All properties can be set or retrieved either by these commands or by subscripts. Autofill applies to all properties and values, and is case insensitive.

For a complete list of property values, use `get(m)`. To see possible value assignments, use `set(m)`.

Subreferencing

The outputs and input channels can be referenced according to

```
m(outputs,inputs)
```

Use colon (:) to denote all channels and the empty matrix ([ ]) to denote no channels. The channels can be referenced by number or by name. For several names, a cell array must be used.

```
m2 = m('y3',{'u1','u4'})
```

```
m3 = m(3,[1 4])
```

For a single output model `m`

```
m4 = m(inputs)
```

will select the corresponding input channels, and for a single input model

```
m5 = m(outputs)
```

will select the indicated output channels.

The string `'measured'` (or any abbreviation like `'m'`) means the measured input channels.

```
m4 = m(3,'m')
```
`m('m')` is the same as `m(:,'m')`

Similarily the string `'noise'` (or any abbreviation) refers to the noise input channels. See above under “The Noise Channels” for more details.

Horizontal Concatenation

Adding input channels

```
m = [m1,m2,...,mN]
```

creates an `idmodel` object `m`, consisting of all the input channels in `m1, ... mN`. The output channels of `mk` must be the same.

Vertical Concatenation

Adding output channels

```
m = [m1;m2;... ;mN]
```

creates an `idmodel` object m consisting of all the output channels in `m1`, `m2`, `..mN`. The input channels of `mk` must all be the same.

Online Help Functions

See `idhelp idprops idmodel`, `help idmodel/subsref`, `help idmodel/subsasgn`, `help idmodel/horzcat`, and `help idmodel/vertcat`.

See Also

`noisecnv`, `nkshift`, `plot`, `size`

idmodred

Purpose

Reduce the order of a model (requires the Control System Toolbox).

Syntax

```
MRED = idmodred(M)
MRED = idmodred(M,ORDER,'DisturbanceModel','None')
```

Description

This function reduces the order of any model `M` given as an `idmodel` object. The resulting reduced order model `MRED` is an idss model.

The function requires several routines in the Control System Toolbox.

`ORDER`: The desired order (dimension of the state-space representation). If `ORDER = []`, which is the default, a plot will show how the diagonal elements of the observability and controllability Gramians of a balanced realization decay with the order of the representation. You will then be prompted to select an order based on this plot. The idea is that such a small element will have a negligible influence on the input-output behavior of the model. It is thus suggested that an order is chosen, such that only large elements in these matrices are retained.

`'DisturbanceModel'`: If the property `DisturbanceModel` is set to `'None'`, then an Output- Error model `MRED` is produced, that is, one with the Kalman gain equal to zero (see Equation (Equation 3-23) in the *Tutorial*). Otherwise (default), also the disturbance model is reduced.

The function will recognize whether `M` is a continuous- or discrete-time model and perform the reduction accordingly. The resulting model `MRED` will be of the same kind in this respect as `M`.

Algorithm

The functions `balreal` and `modred` from the Control System Toolbox are used. The plot, in case `ORDER = []`, shows the vector `g` as returned from `balreal`.

Examples

Build a high order multivariable ARX model, reduce its order to 3 and compare the frequency responses of the original and reduced models.

```
M = arx(data,[4*ones(3,3),4*ones(3,2),ones(3,2)]);
MRED = idmodred(M,3);
bode(M,MRED)
```

Use the reduced order model as initial condition for a third order state-space model:

```
M2= pem(data,MRED);
```

Purpose

Construct an idpoly model for input-output models.

Syntax

```
m = idpoly(A,B)
m = idpoly(A,B,C,D,F,NoiseVariance,Ts)
m = idpoly(A,B,C,D,F,NoiseVariance,Ts,'Property1',Value1,...
           'PropertyN',ValueN)
m = idpoly(mi)
```

Description

idpoly creates a model object containing parameters that describe the general multi-input-single-output model structure.

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t - nk_1) + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t - nk_{nu}) + \frac{C(q)}{D(q)}e(t)$$

A, B, C, D, and F specify the polynomial coefficients.

For single-input systems, these are all row vectors in the standard MATLAB format.

```
A = [1 a1 a2 ...  ana]
```

consequently describes

$$A(q) = 1 + a_1 q^{-1} + \ldots + a_{na} q^{-na}$$

A, C, D, and F all start with 1, while B contains leading zeros to indicate the delays. See ""Polynomial Representation of Transfer Functions" on page 3-11 in the "Tutorial" chapter.

For multi-input systems, B and F are matrices with one row for each input.

For time series, B and F are entered as empty matrices.

```
B = [];   F = [];
```

NoiseVariance is the variance of the white noise sequence $e(t)$, while Ts is the sampling interval.

Trailing arguments C, D, F, NoiseVariance, and Ts can be omitted, in which case they are taken as 1. (If B=[], then F is taken as [].) The property name/property value pairs can start directly after B.

`Ts = 0` means that the model is a continuous-time one. Then the interpretation of the arguments is that

```
A = [1 2 3 4]
```

corresponds to the polynomial $s^3 + 2s^2 + 3s + 4$ in the Laplace variable s, and so on. For continuous-time systems `NoiseVariance` indicates the level of the spectral density of the innovations. A sampled version of the model has the innovations variance `NoiseVariance/Ts`, where `Ts` is the sampling interval. The continuous-time model must have a white noise component in its disturbance description. See the section "Spectrum Normalization and the Sampling Interval" on page 3-95 in the "Tutorial" chapter.

For discrete-time models (`Ts`>0), note the following: `idpoly` strips any trailing zeros from the polynomials when determining the orders. It also strips leading zeros from the `B` polynomial to determine the delays. Keep this in mind when you use `idpoly` and `polydata` to modify earlier estimates to serve as initial conditions for estimating new structures. See the section "Initial Parameter Values" on page 3-91 in the "Tutorial" chapter.

`idpoly` can also take any single-output `idmodel` or LTI-object `mi` as input argument. If an LTI-system has an input group with name '`Noise`', these inputs will be interpreted as white noise with unit variance, and the noise model of the `idpoly` model will be computed accordingly.

Idpoly Properties

The properties of the `idpoly` object can be summarized as follows:

- **`na, nb, nc, nd, nf, nk`**: The orders and delays of the polynomials. Integers or row vectors of integers.
- **`a, b, c, d, f`**: The polynomials, described by row vectors and matrices as detailed above.
- **`da, db, dc, dd, df`**: The estimated standard deviation of the polynomials. Cannot be set.
- **`InitialState`**: How to deal with the initial conditions that are required to compute the prediction of the output: Possible values
  - `'Estimate'`: The necessary initial states are estimated from data as extra parameters
  - `'Backcast'`: The necessary initial states are estimated by a backcasting (backwards filtering) process, described in Knudsen (1994)

- 'Zero': all initial states are taken as zero
- 'Auto': An automatic choice between the above is made, guided by the data.

In addition, any idpoly object also has all the properties of idmodel. See idmodel properties and Algorithm Properties.

Note that all properties can be set or retrieved either by set/get or by subscripts. Autofill applies to all properties and values, and are these case insensitive:

```
m.a=[1 -1.5 0.7];
set(m,'ini','b')
p = roots(m.a)
```

For a complete list of property values, use get(m). To see possible value assignments, use set(m). See also idprops idpoly.

Examples

To create a system of ARMAX, type

```
A = [1 -1.5 0.7];
B = [0 1 0.5];
C = [1 -1 0.2];
m0 = idpoly(A,B,C);
```

This gives a system with one delay (nk = 1).

Create the continuous-time model

$$y(t) = \frac{1}{s(s+1)}u_1(t) + \frac{s+3}{s^2+2s+4}u_2(t) + e(t)$$

Sample it with T=0.1 and then simulate it without noise.

```
B=[0 1;1 3];
F=[1 1 0;1 2 4]
m = idpoly(1,B,1,1,F,1,0)
md = c2d(m,0.1)
y = sim(md,[u1 u2]);
```

Note that the continuous time model will automatically be sampled to the sampling interval of the data, when simulated, so the above is also achieved by

```
u = iddata([],[u1 u2],0.1)
y = sim(m,u)
```

See Also

`sim`, `idss`

References

See Ljung (1999) Section 4.2 for the model structure family. See

T. Knudsen (1994): A new method for estimating ARMAX models. *In Proc. 10th IFAC Symposium on System Identification*, pp 611-617. Copenhagen, Denmark.

for the Backcast method

idss

Purpose

Package state-space parameterizations into an `idss` model.

Syntax

```
m = idss(A,B,C,D)
m =
   idss(A,B,C,D,K,x0,Ts,'Property1',Value1,...,'PropertyN',ValueN)
mss = idss(m1)
```

Description

The function `idss` is used to construct state-space model structures with various parameterizations. It is a complement to `idgrey` and deals with parameterizations that do not require the user to write a special M-file. Instead it covers parameterizations that either a completely `'Free'`, that is, all parameters in the A, B and C matrices can be adjusted freely, or `'Canonical'`, meaning that the matrices are parameterized as canonical forms. The parameterization can also be `'Structured'`, which means that certain elements in the state-space matrices are free to be adjusted, while other are fixed. This is explained below.

`Ts` is the sampling interval. `Ts = 0` means a continuous time model.

The `idss` object `m` describes state-space models in innovations form, of the following kind.

$$\tilde{x}(t) = A(\theta)x(t) + B(\theta)u(t) + K(\theta)e(t)$$

$$x(0) = x_0(\theta)$$

$$y(t) = C(\theta)x(t) + D(\theta)u(t) + e(t)$$

Here $\tilde{x}(t)$ is the time derivative $\dot{x}(t)$ for a continuous time model and $x(t + Ts)$ for a discrete time model.

The model `m` will contain information both about the nominal/initial values of the A, B, C, D, K, and X0-matrices and about how these matrices are parameterized by the parameter vector θ (to be estimated).

The nominal model is defined by `idss(A,B,C,D,K,X0)`. If `K` and `X0` are omitted they are taken as zero matrices of appropriate dimensions.

Defining an `idss` object from a given model

```
mss = idss(m1)
```

constructs an `idss` model from any `idmodel` or LTI-system `m1`.

If `m1` is an LTI system (`ss, tf` or `zpk`) which has no `InputGroup` called `'Noise'`, the corresponding state-space matrices *A,B,C,D* are used to define the `idss` object. The Kalman gain *K* is then set to zero.

If the LTI-system has an `InputGroup` called `'Noise',` these inputs will be interpreted as white noise with a covariance matrix equal to the identity matrix. The corresponding Kalman gain and noise variance is then computed and entered into the `idss` model together with *A,B,C,* and *D*.

Parameterizations

There are several different ways to define the parameterization of the state-space matrices. The parameterization will determine which parameters can be adjusted to data by the parameter estimation routine `pem`:

- **Free Black-Box parameterizations:** This is the default situation and corresponds to letting all parameters in *A, B,* and *C* be freely adjustable. This is obtained by setting the property `'SSParameterization' = 'Free'`. The parameterizations of *D, K,* and *X0* are then determined by the following properties:
  - `'nk'`: A row vector of the same length as the number of inputs. The `ku`-th element is the delay from input channel no `ku`. `nk =[0,...,0]`, thus means that there is no delay from any of the inputs, and that consequently all elements of the *D* matrix should be estimated. `nk =[1,...,1]` means that there is a delay of 1 from each input, so that the *D* matrix is fixed to be zero.
  - `'DisturbanceModel'`: This property affects the parametrization of K and can assume the values:

    `'Estimate'` which means that all elements of the *K* matrix are to be estimated.

    `'None'`: all elements of *K* are fixed to zero.

    `'Fixed'`: all elements of *K* are fixed to their nominal/initial values.

- `'InitialState'`: This properly affects the parameterization of *X0* and can assume the following values:

  `'Auto'`: An automatic choice of the below is made, depending on data (default).

  `'Estimate'`: All elements of *X0* are to be estimated.

  `'Zero'`: All elements of *X0* are fixed to zero.

  `'Fixed'`: All elements of *X0* are fixed to their nominal/initial values.

  `'Backcast'`: The vector *X0* is adjusted, during the parameter estimation step, to a suitable value, but it is not stored as an estimation result.

- **Canonical Black-Box parameterizations**: This is obtained by setting the property `'SSParameterization' = 'Canonical'`. The matrices *A*, *B* and *C* are then parameterized as an observer canonical form, which means that `ny` (= the number of output channels) rows of *A* are fully parameterized while the others contain 0's and 1's in a certain pattern. The *C* matrix is built up of 0's and 1's while the *B* matrix is fully parameterized. See Equation(A.16) in Ljung(1999) for details. The exact form of the parameterization is affected by the property `'CanonicalIndices'`. The default value `'Auto'` is a good choice. The parameterization of the *D*, *K* and *X0*-matrices in this case is determined by the properties `'nk'`, `'DisturbanceModel'` and `'InitialState'` exactly as above.
- **Arbitrarily structured parameterizations:** To cover the general case where arbitrary elements of the state-space matrices may be fixed and others be freely adjusted, corresponds to the case `'SSParameterization' = 'Structured'`. Then the parameterization is determined by the `idss` properties `As, Bs, Cs, Ds, Ks,` and `X0s`. These are the *structure matrices* that are "shadows" of the state-space matrices, so that an element in these matrices that is equal to `NaN` indicates a freely adjustable parameter, while a numerical value in these matrices indicates that the corresponding system matrix element is fixed (nonadjustable) to this value.

  See the Examples below.

idss Properties

To summarize the properties of the `idss` object we have:

- **`SSParameterization`** with possible values
  - `'Free'`: Means that all parameters in *A*,*B* and *C* are freely adjustable and the parameterizations of *D*, *K* and *X0* depend on the properties `'nk'`, `'DisturbanceModel'` and `'InitialState'`
  - `'Canonical'`: Means that *A* and *C* are parameterized as an observer canonical form. The details of this parameterization depends on the property `'CanonicalIndices'`. The *B*-matrix is always fully parameterized, and the parameterizations of *D*, *K*, and *X0* depends on the properties `'nk'`, `'DisturbanceModel'`, and `'InitialState'`.
  - `'Structured'`: Means that the parametrization is determined by the properties (the structure matrices) `'As'`, `'Bs'`, `'Cs'`, `'Ds'`, `'Ks'`, and `'X0s'`. A `NaN` in any position in these matrices denotes a freely adjustable parameter and a numeric value denotes a fixed and nonadjustable parameter.
- **`nk:`** A row vector with as many entries as the number of input channels. The entry number `k` denotes the time delay from `uk(t)` to `y(t)`. This property is relevant only for `'Free'` and `'Canonical'` parameterizations. If any delay is larger than 1, the structure of the *A*, *B*, and *C* matrices will accommodate this delay, at the price of a higher order model.
- **`DisturbanceModel`** with possible values
  - `'Estimate'`: Means that the *K* matrix is fully parameterized.
  - `'None'`: Means that the *K* matrix is fixed to zero. This gives a so-called Output-Error model, since the model output depends on past inputs only.
  - `'Fixed'`: Means that the *K* matrix is fixed to the current nominal values
- **`InitialState`** with possible values
  - `'Estimate'`: Means that *X0* is fully parameterized.
  - `'Zero'`: Means that *X0* is fixed to zero.
  - `'Fixed'`: Means that *X0* is fixed to the current nominal value.
  - `'Backcast'`: The value of *X0* is estimated by the identification routines as the best fit to data, but it is not stored.
  - `'Auto'`: Gives an automatic, and data-dependent choice between `'Estimate'`, `'Zero'` and `'Backcast'`.

- **A, B, C, D, K,** and **X0**: The state-space matrices that can be set and retrieved at any time. These contain both fixed values and estimated/ nominal values.
- **dA, dB, dC, dD, dK,** and **dX0**: The estimated standard deviations of the state-space matrices. These cannot be set, only retrieved. Note that these are not defined for an `idss` model with `'Free' SSParameterization`. You can then convert the parameterization to `'Canonical'` and study the uncertainties of the matrix elements in that form.
- **As, Bs, Cs, Ds, Ks,** and **X0s**: These are the structure matrices that have the same sizes as `A, B, C` etc. and show the freely adjustable parameters as `NaN` in the corresponding position. These properties are used to define the model structure for `'SSParameterization' = 'Structured'`. They are however always defined and can be studied also for the other parametrizations.
- **CanonicalIndices**: Determines the details of the canonical parameterization. It is a row vector of integers with as many entries as there are outputs. They sum up to the system order. This is the so-called pseudo-canonical multi-index, with an exact definition, e.g., on page 132 in Ljung (1999). A good default choice is `'Auto'`. This property is relevant only for the canonical parameterization case. Note however, that for `'Free'` parameterizations, the estimation algorithms also store a canonically parameterized model, to handle the model uncertainty.

In addition to these properties, `idss` objects also have all the properties of the `idmodel` object. See `idmodel` properties, `Algorithm Properties`, and `EstimationInfo`.

Note that all properties can be set and retrieved either by `set/get` or by subscripts. Autofill applies to all properties and values, and are case insensitive.

```
m.ss='can'
set(m,'ini','z')
p = eig(m.a)
```

For a complete list of property values, use `get(m)`. To see possible value assignments, use `set(m)`. See also `idprops idss`.

Examples

Define a continuous-time model structure corresponding to

$$\dot{x} = \begin{bmatrix} \theta_1 & 0 \\ 0 & \theta_2 \end{bmatrix} x + \begin{bmatrix} \theta_3 \\ \theta_4 \end{bmatrix} u$$

$$y = \begin{bmatrix} 1 & 1 \end{bmatrix} x + e$$

with initial values

$$\theta = \begin{bmatrix} -0.2 & -0.3 & 2 & 4 \end{bmatrix}$$

and estimate the free parameters:

```
A = [-0.2, 0; 0, -0.3]; B = [2;4]; C=[1, 1]; D = 0
m0 = idss(A,B,C,D);
m0.As = [NaN,0;0,NaN];
m0.Bs = [NaN;NaN];
m0.Cs = [1,1];
m0.Ts = 0;
m = pem(z,m0);
```

Estimate a model in free parameterization. Convert it to continuous time, then convert it to canonical form and continue to fit this model to data.

```
m1 = n4sid(data,3);
m1 = d2c(m1);
m1.ss ='can';
m = pem(data,m1);
```

All of this can be done at once by

```
m = pem(data,3,'ss','can','ts',0)
```

See Also

`n4sid`, `pem`

impulse

Purpose

Estimate/compute/display impulse response.

Syntax

```
impulse(m)
impulse(data)
impulse(data,'sd',sd,'pw',na,Time)
impulse(m,'sd',sd,Time)
impulse(m1,m2,...,dat1, ...,mN,Time,'sd',sd)
impulse(m1,'PlotStyle1',m2,'PlotStyle2',...,dat1,'PlotStylek',...,
   mN,'PlotStyleN',Time,'sd',sd)
[y,t,ysd] = impulse(m)
mod = impulse(data)
```

Description

`impulse` can be applied both to `idmodels` and to `iddata` sets, as well as to any mixture.

For a discrete time `idmodel m`, the impulse response `y` and, when required, its estimated standard deviation `ysd`, is computed using `sim`. When called with output arguments, `y`, `ysd` and the time vector `t` are returned. When impulse is called without output arguments, a plot of the impulse response is shown. If `sd` is given a value larger than zero, a confidence region around zero is drawn. It corresponds to the confidence of `sd` standard deviations. In the plots, the impulse is inversely scaled with the sampling interval, so that it has the same energy regardless of the sampling.

Adding an argument `'fill'` among the input arguments gives an uncertainty region marked by a filled area, rather than by dash-dotted lines.

The start time `T1` and the end time `T2` can be specified by `Time= [T1 T2]`. If `T1` is not given, it is set to `-T2/4`. The negative time lags (the impulse is always assumed to occur at time 0) show possible feedback effects in the data, when the impulse is estimated directly from data. If `Time` is not specified, a default value is used.

For an `iddata` set data, `impulse(data)` estimates a high order, noncausal FIR model after first having prefiltered the data so that the input is “as white as possible.” The impulse response of this FIR model and, when asked for, its confidence region is then plotted. When called with an output argument, `impulse`, in the `iddata` case, returns this FIR model, stored as an `idarx` model.The order of the prewhitening filter can be specified as `na`. The default value is `na = 10`.

Any number and any mixture of models and data sets can be used as input arguments. The responses are plotted with each input/output channel (as defined by the models' and data sets' `InputName` and `OutputName`) as a separate plot. Colors, linestyles, and marks can be defined by `PlotStyle` values. These are the same as for the regular `plot` command, like

```
impulse(m1,'b-*',m2,'y--',m3,'g')
```

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `m.NoiseVariance` = Λ. The model can also be described with unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

- `impulse(m)` plots the impulse response of the transfer function G.
- `impulse(m('n'))` plots the impulse response of the transfer function H. (ny inputs and ny outputs).The input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- If `m` is a time series, that is $nu = 0$, `impulse(m)` plots the impulse response of the transfer function H.
- `impulse(noisecnv(m))` plots the impulse response of the transfer function $[G\ H]$ ($nu+ny$ inputs and ny outputs). The noise input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- `impulse(noisecnv(m,'norm'))` plots the impulse response of the transfer function $[G\ HL]$ ($nu+ny$ inputs and ny outputs. The noise input channels have names `v@yname`, where `yname` is the name of the corresponding output.

Arguments

If `impulse` is called with a single `idmodel m`, the output argument `y` is a 3-D array of dimension `Nt`-by-`ny`-by-`nu`. Here `Nt` is the length of the time vector `t`, `ny` is the number of output channels and `nu` is the number of input channels. Thus `y(:,ky,ku)` is the response in ouput `ky` to an impulse in the `ku`-th input channel.

`ysd` has the same dimensions as `y` and contains the standard deviations of `y`.

If `impulse` is called with an output argument and a single data set in the input arguments, the output is returned as an `idarx` model `mod` containing the high order FIR model, and its uncertainty. By calling impulse with `mod`, the responses can be displayed and returned without having to redo the estimation.

Example

`impulse(data,'sd',3)` estimates and plots the impulse response

```
mod = impulse(data)
impulse(mod,'sd',3)
```

See Also

`cra`, `step`

Purpose

Set initial values for the parameters to be estimated.

Syntax

```
m = init(m0)
m = init(m0,R,pars,sp)
```

Description

This function randomizes initial parameter estimates for model structures `m0` for any idmodel type. `m` is the same model structure as `m0`, but with a different nominal parameter vector. This vector is used as the initial estimate by `pem`.

The parameters are randomized around `pars` with variances given by the row vector `R`. Parameter number *k* is randomized as `pars(k)` + `e*sqrt(R(k))`, where `e` is a normal random variable with zero mean and a variance of 1. The default value of `R` is all ones, and the default value of `pars` is the nominal parameter vector in `m0`.

Only models that give stable predictors are accepted. If `sp = 'b'`, only models that are both stable and have stable predictors are accepted.

`sp = 's'` requires stability only of the model, and `sp = 'p'` requires stability only of the predictor. `Sp = 'p'` is the default.

Sufficiently free parameterizations can be stabilized by direct means without any random search. To just stabilize such an initial model, set `R =0`. With `R>0` randomization is also done.

For model structures where a random search is necessary to find a stable model/predictor, a maximum of 100 trials are made by `init`. It may be difficult to find a stable predictor for high order systems just by trial and error.

See Also

`idss`, `n4sid`, `pem`

ivar

Purpose

Estimate the parameters of an AR model using an approximately optimal choice of instrumental variable procedure.

Syntax

```
m = ivar(y,na)
m = ivar(y,na,nc,maxsize)
```

Description

The parameters of an AR model structure

$$A(q)y(t) = v(t)$$

are estimated using the instrumental variable method. `y` is the signal to be modeled, entered as an `iddata` object (outputs only). `na` is the order of the A polynomial (the number of A parameters). The resulting estimate is returned as an `idpoly` model `m`. The routine is for scalar signals only.

In the above model, $v(t)$ is an arbitrary process, assumed to be a moving average process of order `nc`, possibly time varying. (Default is `nc = na`.) Instruments are chosen as appropriately filtered outputs, delayed `nc` steps.

The optional argument `maxsize` is explained under `Algorithm Properties`.

Examples

Compare spectra for sinusoids in noise, estimated by the IV method and estimated by the forward-backward least-squares method.

```
y = iddata(sin([1:500]'*1.2) + sin([1:500]'*1.5) +
0.2*randn(500,1),[]);
miv = ivar(y,4);
mls = ar(y,4);
bode(miv,mls)
```

See Also

`ar`, `etfe`, `spa`

References

Stoica, P. et al., *Optimal Instrumental variable estimates of the AR-parameters of an ARMA process*, IEEE Trans. Autom. Control, Vol AC-30, 1985, pp. 1066-1074.

Purpose

Compute fit between simulated and measured output for a group of model structures.

Syntax

```
v = ivstruc(ze,zv,NN)
v = ivstruc(ze,zv,NN,p,maxsize)
```

Description

`NN` is a matrix that defines a number of different structures of the ARX type. Each row of `NN` is of the form

```
nn = [na nb nk]
```

with the same interpretation as described for `arx`. See `struc` for easy generation of typical `NN` matrices for single-input systems.

Each of `ze` and `zv` are iddata objects containing output-input data. Models for each model structure defined in `NN` are estimated using the instrumental variable (IV) method on data set `ze`. The estimated models are simulated using the inputs from data set `zv`. The normalized quadratic fit between the simulated output and the measured output in `zv` is formed and returned in `v`. The rows below the first row in `v` are the transpose of `NN`, and the last row contains the logarithms of the condition numbers of the IV matrix

$$\sum \zeta(t)\varphi^T(t)$$

A large condition number indicates that the structure is of unnecessarily high order (see page 498 in Ljung (1999)).

The information in `v` is best analyzed using `selstruc`.

If `p` is equal to zero, the computation of condition numbers is suppressed. For the use of `maxsize`, see `Algorithm Properties`.

The routine is for single-output systems only.

Note The IV method used does not guarantee that the obtained models are stable. The output-error fit calculated in `v` may then be misleading.

Examples

Compare the effect of different orders and delays, using the same data set for both the estimation and validation.

```
v = ivstruc(z,z,struc(1:3,1:2,2:4));
nn = selstruc(v)
m = iv4(z,nn);
```

Algorithm

A maximum order ARX model is computed using the least-squares method. Instruments are generated by filtering the input(s) through this model. The models are subsequently obtained by operating on submatrices in the corresponding large IV matrix.

See Also

`arxstruc`, `iv4`, `n4sid`, `selstruc`, `struc`,

Purpose

Estimate the parameters of an ARX model using the instrumental variable (IV) method with arbitrary instruments.

Syntax

```
m = ivx(data,orders,x)
m = ivx(data,orders,x,maxsize)
```

Description

`ivx` is a routine analogous to the `iv4` routine, except that you can use arbitrary instruments. These are contained in the matrix `x`. Make this the same size as the output, `data.y`. In particular, if data contains several experiments x must be a cell array with one matrix/vector for each experiment. The instruments used are then analogous to the regression vector itself, except that `y` is replaced by `x`.

Note that `ivx` does not return any estimated covariance matrix for `m`, since that requires additional information. `m` is returned as an `idpoly` object for single output systems and as an `idarx` object for multi-output systems.

Use `iv4` as the basic IV routine for ARX model structures. The main interest in `ivx` lies in its use for nonstandard situations; for example when there is feedback present in the data, or when other instruments need to be tried out. Note that there is also an IV version that automatically generates instruments from certain filters you define (type `help iv`).

See Also

`iv4`, `ivar`

References

Ljung (1999), page 222.

iv4

Purpose

Estimate the parameters of an ARX model using an approximately optimal four-stage instrumental variable (IV) procedure.

Syntax

```
m = iv4(data,orders)
m = iv4(data,'na',na,'nb',nb,'nk',nk)
m= iv4(data,orders,'Property1',Value1,...,'PropertyN',ValueN)
```

Description

This routine is an alternative to `arx` and the use of the arguments are entirely analogous to the `arx` function. The main difference is that the procedure is not sensitive to the color of the noise term $e(t)$ in the model equation.

For an interpretation of the loss function (innovations covariance matrix), consult "Interpretation of the Loss Function" on page 3-98 in the *Tutorial*.

Examples

Here is an example of a two-input, one-output system with different delays on the inputs u_1 and u_2.

```
z = iddata(y, [u1 u2]);
nb = [2 2];
nk = [0 2];
m= iv4(z,[2 nb nk]);
```

Algorithm

The first stage uses the `arx` function. The resulting model generates the instruments for a second-stage IV estimate. The residuals obtained from this model are modeled as a high-order AR model. At the fourth stage, the input-output data are filtered through this AR model and then subjected to the IV function with the same instrument-filters as in the second stage.

For the multi-output case, optimal instruments are obtained only if the noise sources at the different outputs have the same color. The estimates obtained with the routine are reasonably accurate though even in other cases.

See Also

`arx`, `oe`

References

Ljung (1999), equations (15.21)-(15.26).

Purpose

To allow direct calls to typical LTI-commands from idmodel objects. The Control System Toolbox is required for these commands.

Syntax

```
append, augstate, balreal, canon, d2d, feedback, inv, minreal,
  modred, norm, parallel, series, ss2ss
```

Description

If you have the control system toolbox, most of the relevant LTI-commands, as listed under Syntax, can be directly applied to any `idmodel` (`idarx`, `idgrey`, `idpoly`, `idss`). You can also use the overloaded operations +, -, and *. The same operations are performed and the result is delivered as an idmodel. The original covariance information is however lost, most of the time.

Example

You have two more or less identical processes connected in series. Estimate a model for one of them, and use that to form an initial estimate for a model of the connected process

```
m = pem(data) % data concerns one of the processes
m2 = pem(data2,m*m) % data2 are from the whole connected process
```

merge (iddata)

Purpose

Merge different data sets into one iddata object.

Syntax

```
dat = merge(dat1,dat2,....,datN)
```

Description

dat collects the data sets in dat1,.. datN into one iddata object, with several *experiments*. The number of experiments in dat will the sum of the number of experiments in datk. For the merging to be allowed a number of conditions must be satisfied:

- All of datk must have the same number of input channels, and the InputNames must be the same.
- All of datk must have the same number of output channels, and the OutputNames must be the same. If some input or output channel is lacking in one experiment, it can be replaced by a vector of NaN's to conform with these rules.
- If the ExperimentNames of datk have been specified to something else than the default 'Exp1', 'Exp2', etc., they must all be unique. If default names overlap, they will be modified, so that dat will have a unique list of ExperimentNames.

The sampling intervals, the number of observations, and the input properties (Period, InterSample) may be different in the different experiments.

The individual experiments can be retrieved by subreferencing with curly brackets.

```
dat1 = dat{1}
```

They can also be referenced by experiment name.

```
dat1 = dat{'Exp1'}
```

An alternative to merge is to subassign in the corresponding way.

```
dat{1} = dat1, dat{2} = dat2, ..., dat{N} = datN.
```

Using names within brackets simultaneously sets the experiment name.

```
dat = dat1; dat{'Try2'} = dat2; dat{Day3'} = dat3 etc
```

Storing multiple experiments as one iddata object may be very useful to handle experimental data that have been collected on different occasions, or

when a data set has been split up to remove “bad” portions of the data. All the toolbox’s routines accept multiple experiment data.

Example

Bad portions of data have been detected around sample 500 and between samples 720 - 730. Cut out these bad portions and form a multiple experiment data set that can be used to estimate models, without the bad data destroying the estimate.

```
dat = merge(dat(1:498),dat(502:719),dat(719:1000))
m = pem(dat)
```

Use the first two parts to estimate the model and the third one for validation.

```
m = pem(dat{1:2});
compare(dat{3},m)
```

See also iddemo #8

See Also

`iddata`

merge (idmodel)

Purpose

Merge different models into one.

Syntax

```
m = merge(m1,m2,....,mN)
[m,tv] = merge(m1,m2)
```

Description

The models `m1,m2,...,mN` must all be of the same structure, just differing in parameter values and covariance matrices. `m` is then the merged model, where the parameter vector is a statistically weighted mean (using the covariance matrices to determine the weights) of the parameters of `mk`.

When two models are merged,

```
[m, tv] = merge(m1,m2)
```

returns a test variable tv. It is χ^2 distributed with `n` degrees of freedom, if the parameters of `m1` and `m2` have the same means. Here `n` is the length of the parameter vector. A large value of `tv` thus indicates that it might be questionable to merge the models.

Merging models is an alternative to merging data sets, and estimating a model for the merged data. Consequently

```
m1 = arx(z1,[2 3 4]);
m2 = arx(z2,[2 3 4]);
ma = merge(m1,m2);
```

and

```
mb = arx(merge(z1,z2),[2 3 4]);
```

lead to models `ma` and `mb` that are related and should be close. The difference is that merging the data sets assumes that the signal-to-noise ratios are about the same in the two experiments. Merging the models allows one model to much more uncertain, e.g, due to more disturbances in that experiment. If the conditions are about the same, it is recommended to merge data rather than models, since this is more efficient and typically involves better conditioned calculations.

Purpose

Select a directory for `idprefs.mat`, a file that stores the graphical user interface's start-up information.

Syntax

```
midprefs
midprefs(path)
```

Description

The graphical user interface `ident` allows a large number of variables for customized choices. These include the window layout, the default choices of plot options, and names and directories of the four most recent sessions with `ident`. This information is stored in the file `idprefs.mat`, which should be placed on the user's `MATLABPATH`. The default, automatic location for this file is in the same directory as the user's `startup.m` file.

`midprefs` is used to select or change the directory where you store `idprefs.mat`. Either type `midprefs`, and follow the instructions, or give the directory name as the argument. Include all directory delimiters as in the PC case

```
midprefs('c:\matlab\toolbox\local\')
```

or in the UNIX case

```
midprefs('/home/ljung/matlab/')
```

misdata

Purpose

To reconstruct missing input and output data

Syntax

```
Datae = misdata(Data)
Datae = misdata(Data,Model)
Datae = misdata(Data,Maxiter,Tol)
```

Description

`Data` is input-output data in the `iddata` object format. Missing data samples (both in inputs and in outputs) are entered as `NaN`.

`Datae` is an `iddata` object where the missing data have been replaced by reasonable estimates.

`Model` is any `idmodel (idarx, idgrey, idpoly, idss)` used for the reconstruction of missing data.

If no suitable model is known, it will be estimated in an iterative fashion using default order state-space models.

`Maxiter` is the maximum number of iterations carried out (default 10). The iterations will be terminated when the difference between two consecutive data estimates differ by less than `tol`%. The default value of `tol` is 1.

Algorithm

For a given model, the missing data are estimated as parameters so as to minimize the output prediction errors obtained from the reconstructed data. See Section 14.2 in Ljung (1999). Treating missing outputs as parameters is not the best approach from a statistical point of view, but is a good approximation in many cases.

When no model is given, the algorithm alternates between estimating missing data and estimating models, based on the current reconstruction.

Purpose

To shift data sequences

Syntax

```
Datas = nkshift(Data,nk)
```

Description

`Data` contains input-output data in the iddata format.

`nk` is a row vector with the same length as the number of input channels in `Data`.

`Datas` is an `iddata` object where the input channels in `Data` have been shifted according to `nk`. A positive value of `nk(ku)` means that input channel number `ku` is delayed `nk(ku)` samples.

`nkshift` lives in symbiosis with the `InputDelay` property of `idmodel`:

```
m1 = pem(dat,4,'InputDelay',nk)
```

is related to

```
m2 = pem(nkshift(dat,nk),4);
```

such that `m1` and `m2` are the same models, but `m1` stores the delay information for use when frequency responses etc. are computed.

Note the difference with the `idss` and `idpoly` property `nk`:

```
m3 = pem(dat,4,'nk',nk)
```

gives a model which itself explicitly contains a delay of `nk` samples

See also

`idss, Algorithm Properties`

noisecnv

Purpose

Convert an `idmodel` with noise channels to a model with only measured channels.

Syntax

```
mod1 = noisecnv(mod)
mod2 = noisecnv(mod,'norm')
```

Description

`mod` is any `idmodel, idarx, idgrey, idpoly or idss`.

The noise input channels in `mod` are converted as follows: Consider a model with both measured input channels u (*nu* channels) and noise channels e (*ny* channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `mod.NoiseVariance` = Λ. The model can also be described with unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

- `mod1 = noisecnv(mod)` converts the model to a representation of the system [G H] with *nu*+*ny* inputs and *ny* outputs. All input are treated as measured, and `mod1` does not have any noise model. The former noise input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- `mod2 = noisecnv(mod,'norm')` converts the model to a representation of the system [G HL] with *nu*+*ny* inputs and *ny* outputs. All input are treated as measured, and `mod2` does not have any noise model. The former noise input channels have names `v@yname`, where `yname` is the name of the corresponding output. Note that the noise variance matrix factor L typically is uncertain (has a non-zero covariance). This is taken into account in the uncertainty description of `mod2`.
- If `mod` is a time series, that is *nu* = 0, `mod1` is a model that describes the transfer function H with measured input channels. Analogously, `mod2` describes the transfer function HL.

Note the difference with subreferencing:

- mod('m') gives a description of G only.

- `mod('n')` gives description of the noise model characteristics as a time series model, i.e it describes *H* and also the covariance of *e*. In contrast, `noisecnv(m('n'))` describes just the transfer function *H*. To obtain a description of the normalized transfer function *HL*, use `noisecnv(m('n'),'norm')`

Converting the noise channels to measured inputs is useful to study the properties of the individual transfer functions from noise to output. It is also useful for transforming `idmodel` objects to representations that do not handle disturbance descriptions explicitly.

nuderst

Purpose

Select the step-size for numerical differentiation.

Syntax

```
nds = nuderst(pars)
```

Description

The function `pem` uses numerical differentiation with respect to the model parameters when applied to state-space structures. The same is true for many functions that transform model uncertainties to other representations.

The step-size used in these numerical derivatives is determined by the M-file `nuderst`. The output argument `nds` is a row vector whose k-th entry gives the increment to be used when differentiating with respect the k-th element of the parameter vector `pars`.

The default version of `nuderst` uses a very simple method. The step-size is the maximum of 10^{-4} times the absolute value of the current parameter and 10^{-7}. You can adjust this to the actual value of the corresponding parameter by editing `nuderst`. Note that the nominal value, for example 0, of a parameter may not reflect its normal size.

Purpose

Plot Nyquist curve of frequency function.

Syntax

```
nyquist(m)
[fr,w] = nyquist(m)
[fr,w,covfr] = nyquist(m)
nyquist(m1,m2,m3,...,w)
nyquist(m1,'PlotStyle1',m2,'PlotStyle2',...)
nyquist(m1,m2,m3,..'sd',sd,'mode',mode)
```

Description

`nyquist` computes the complex-valued frequency response of `idmodel` and `idfrd` models. When invoked without left-hand arguments, `nyquist` produces a Nyquist plot on the screen, that is, a graph of the frequency response's imaginary part against its real part.

The argument `m` is an arbitrary *idmodel* or `idfrd` model. This model can be continuous or discrete, and SISO or MIMO. The `InputNames` and `OuputNames` of the models are used to plot the responses for different I/O channels in separate plots. Pressing the **Enter** key advances the plot from one input-output pair to the next one. Specific I/O channels can be selected by the normal subreferencing: `m(ky,ku)`. With `mode = 'same'` all plots are given in the same diagram.

`nyquist(m,w)` explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval `[wmin,wmax]`, set `w = {wmin,wmax}`. (Notice the curly brackets.) To use particular frequency points, set `w` to the vector of desired frequencies. Use `logspace` to generate logarithmically spaced frequency vectors. All frequencies should be specified in radians/sec.

`nyquist(m1,m2,...,mN)` or `nyquist(m1,m2,...mN,w)` plots the Bode responses of several `idmodels` or `idfrd` models on a single figure. The models may be mixes of different sizes and continuous/discrete. The sorting of the plots is made based on the `InputNames` and `OutputNames`.

`nyquist(m1,'PlotStyle1',...,mN,'PlotStyleN')` further specifies which color, linestyle, and/or marker should be used to plot each system, as in

```
nyquist(m1,'r--',m2,'gx')
```

When `sd` is specified as a number larger than zero, confidence regions will also be plotted. These are ellipses in the complex plane and correspond to the region

where the true response at the frequency in question is to be found with a confidence, corresponding to sd standard deviations (of the Gaussian distribution).

If the argument indicating standard deviations is given as in 'sd+5', a confidence region is plotted for every 5:th frequency, marking the center point by '+'. The default is 'sd+10'.

Note that the frequencies cannot be specified for idfrd objects. For those, the plot and response are calculated for the internally stored frequencies. If the frequencies w are specified when several models are treated, they will apply to all non-idfrd models in the list. If you want different frequencies for different models, you should thus first convert them to idfrd objects using the idfrd command.

For time-series models (no input channels) the Nyquist plot is not defined.

Arguments

When nyquist is called with a single system and output arguments

`fr = nyquist(m,w)` or `[fr,w,covfr] = nyquist(m)`

no plot is drawn on the screen. If m has ny outputs and nu inputs, and w contains nw frequencies, then fr is an ny-by-nu-by-Nw array such that `fr(ky,ku,k)` gives the complex-valued frequency response from input ku to output ky at the frequency `w(k)`. For a SISO model, use `fr(:)` to obtain a vector of the frequency response. The uncertainty information covfr is a 5-D array where `covfr(ky,ku,k,:,:))` is the 2-by-2 covariance matrix of the response from input ku to output ky at frequency `w(k)`. The 1,1 element is the variance of the real part, the 2,2 element the variance of the imaginary part and the 1,2 and 2,1 elements the covariance between the real and imaginary parts.

`squeeze(covfr(ky,ku,k,:,:))` gives the covariance matrix of the corresponding response.

If m is a time series (no input), fr is returned as the (power) spectrum of the outputs; an ny-by-ny-by-Nw array. Hence `fr(:,:,k)` is the spectrum matrix at frequency `w(k)`. The element `fr(k1,k2,k)` is the cross spectrum between outputs k1 and k2 at frequency `w(k)`. When k1=k2, this is the real-valued power spectrum of output k1. covfr is then the covariance of the spectrum fr, so that `covfr(k1,k1,k)` is the variance of the power spectrum of output k1 at frequency `w(k)`. No information about the variance of the cross spectra is normally given. (That is, `covfr(k1,k2,k) = 0` for k1 not equal to k2.)

If the model `m` is not a time series, use `fr = nyquist(m('n'))` to obtain the spectrum information of the noise (output disturbance) signals.

Examples

```
g = spa(data)
m = n4sid(data,3)
nyquist(g,m,3)
```

See Also

`bode`, `etfe`, `ffplot`, `idfrd`, `spa`

n4sid

Purpose

Estimate state-space models using a subspace method.

Syntax

```
m = n4sid(data)
m = n4sid(data,order,'Property1',Value1,...,'PropertyN',ValueN)
```

Description

The function `n4sid` estimates models in state-space form, and returns them as an `idss` object `m`. It handles an arbitrary number of input and outputs, including the time-series case (no input). The state-space model is in the innovations form:

$$x(t + Ts) = Ax(t) + Bu(t) + Ke(t)$$
$$y(t) = Cx(t) + Du(t) + e(t)$$

`m`: The resulting model as an `idss` object.

`data`: An `iddata` object containing the output-input data.

`order`: The desired order of the state-space model. If `order` is entered as a row vector (like `order = [1:10]`), preliminary calculations for all the indicated orders are carried out. A plot will then be given that shows the relative importance of the dimension of the state vector. More precisely, the singular values of the Hankel matrices of the impulse response for different orders are graphed. You will be prompted to select the order, based on this plot. The idea is to choose an order such that the singular values for higher orders are comparatively small. If `order = 'best'`, a model of "best" (default choice) order is computed, among the orders 1:10. This is the default choice of `order`.

The list of property name/property value pairs may contain any `idss` and algorithm properties. See `idss` and `Algorithm Properties`.

`idss` properties that are of particular interest for `n4sid` are

- **`nk:`** The delays from the inputs to the outputs, a row vector with the same number of entries as the number of input channels. Default is `nk = [1 1 ... 1]`. Note that delays being 0 or 1 show up as zeros or estimated parameters in the D matrix. Delays larger than 1 means that a special structure of the A, B and C matrices are used to accommodate the delays. This also means that the actual order of the state-space model will be larger than `order`.
- **`CovarianceMatrix`** (can be abbreviated to `'co'`): Setting `CovarianceMatrix` to `'None'` will block all calculations of uncertainty measures. These may take the major part of the computation time. Note that, for a `'Free'`

parameterization, the individual matrix elements cannot be associated with any variance (these parameters are not identifiable). Instead, the resulting model `m` stores a hidden state-space model in canonical form, that contains covariance information. This is used when the uncertainty of various input-output properties are calculated. It can also be retrieved by `m.ss = 'can'`. The actual covariance properties of `n4sid` estimates are not known today. Instead the Cramer-Rao bound is computed and stored as an indication of the uncertainty.

Algorithm properties that are special interest are:

- **`Focus:`** Assumes the values `'Prediction'` (default), `'Simulation'`, or any SISO linear filter (given as an LTI or `idmodel` object, or as filter coefficients. See `Algorithm Properties`.) Setting `'Focus'` to `'Simulation'` chooses weights that should give a better simulation performance for the model. In particular, a stable model is guaranteed. Selecting a linear filter will focus the fit to the frequency ranges that are emphasized by this filter.
- **`N4Weight:`** This property determines some weighting matrices used in the singular-value decomposition that is a central step in the algorithm. Two choices are offered: `'MOESP'` that corresponds to the MOESP algorithm by Verhaegen, and `'CVA'` which is the canonical variable algorithm by Larimore. The default value is `'N4Weight' = 'Auto'`, which gives an automatic choice between the two options. `m.EstimationInfo.N4Weight` tells you what the actual choice turned out to be.
- **`N4Horizon:`** Determines the prediction horizons forward and backward, used by the algorithm. This is a row vector with three elements: `N4Horizon =[r sy su]`, where `r` is the maximum forward prediction horizon, i.e., the algorithms uses up to r-step ahead predictors. `sy` is the number of past outputs, and `su` is the number of past inputs that are used for the predictions. These numbers may have a substantial influence of the quality of the resulting model, and there are no simple rules for choosing them. Making `'N4Horizon'` a `k`-by-3 matrix, means that each row of `'N4Horizon'` will be tried out, and the value that gives the best (prediction) fit to data will be selected. (This option cannot be combined with selection of model order.) If the property `'Trace'` is `'On'`, information about the results will be given in the MATLAB command window.

If you specify only one column in `'N4Horizon'`, the interpretation is `r=sy=su`. The default choice is `'N4Horizon' = 'Auto'`, which uses an Akaike Information Criterion (AIC) for the selection of `sy` and `su`. If `'DisturbanceModel' = 'None'`, `sy` is set to 0. Typing `m.EstimationInfor.N4Horizon` will tell you what the final choice of horizons were.

Algorithm

The variants of the implemented algorithm are described in Section 10.6 in Ljung (1999).

Examples

Build a fifth order model from data with three inputs and two outputs. Try several choices of auxiliary orders. Look at the frequency response of the model.

```
z = iddata([y1 y2],[ u1 u2 u3]);
m = n4sid(z,5,'n4h',[7:15]','trace','on');
bode(m,'sd',3)
```

Estimate a continuous-time model, in a canonical form parameterization, focusing on the simulation behavior. Determine the order yourself after seeing the plot of singular values.

```
m = n4sid(m,[1:10],'foc','sim','ssp','can','ts',0)
bode(m)
```

See Also

`idss`, `pem`, `Algorithm Properties`

References

P. vanOverschee and B. DeMoor: *Subspace Identification of Linear Systems: Theory, Implementation, Applications*. Kluwer Academic Publishers, 1996.

M. Verhaegen: Identification of the deterministic part of MIMO state space models. *Automatica*, Vol 30, pp 61-74, 1994.

W.E. Larimore: Canonical variate analysis in identification, filtering and adaptive control. In *Proc. 29th IEEE Conference on Decision and Control*, pp 596-604, Honolulu, 1990.

Purpose

Estimate the parameters of an Output-Error model.

Syntax

```
m = oe(data,orders)
m = oe(data,'nb',nb,'nf',nf,'nk',nk)
m = oe(data,orders,'Property1',Value1,'Property2',Value2,...)
```

Description

oe returns m as an idpoly object with the resulting parameter estimates, together with estimated covariances.The parameters of the Output-Error model structure

$$y(t) = \frac{B(q)}{F(q)}u(t - nk) + e(t)$$

are estimated using a prediction error method.

data is an iddata object containing the output-input data. The structure information can either be given explicitly as `(...,'nb',nb,'nf',nf,'nk',nk,...)`, or in the argument orders given as

```
orders = [nb nf nk]
```

The parameters nb and nf are the orders of the Output-Error model and nk is the delay. Specifically,

$$nb: \quad B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$$

$$nf: \quad F(q) = 1 + f_1 q^{-1} + \ldots + f_{nf} q^{-nf}$$

Alternatively, you can specify the vector as

```
orders = mi
```

where mi is an initial guess at the Output-Error model given in idpoly format. See “Polynomial Representation of Transfer Functions” on page 3-11 in the “Tutorial” chapter for more information.

For multi-input systems, nb, nf, and nk are row vectors with as many entries as there are input channels. Entry number i then describes the orders and delays associated with the i-th input.

The structure and the estimation algorithm are affected by any property name/property value pairs that are set in the input argument list. Useful properties are `'Focus'`, `'InitialState'`, `'InputDelay'`, `'SearchDirection'`, `'MaxIter'`, `'Tolerance'`, `'LimitError'`, `'FixedParameter'`, and `'Trace'`.

See `Algorithm Properties`, `idpoly` and `idmodel` for details of these properties and their possible values.

`oe` does not support multi-output models. Use state-space model for this case (see `n4sid` and `pem`).

Algorithm

`oe` uses essentially the same algorithm as `armax` with modifications to the computation of prediction errors and gradients.

See Also

`armax`, `bj`, `idpoly`, `pem`

Purpose

Compute the prediction errors associated with a model and a data set.

Syntax

```
e = pe(m,data)
[e,x0] = pe(m,data,init)
```

Description

`data` is the output-input data set, given as an `iddata` object, and `m` is any idmodel object.

`e` is returned as an `iddata` object, so that `e.OutputData` contains the prediction errors that result when model `m` is applied to the data:

$$e(t) = H^{-1}(q)[y(t) - G(q)u(t)]$$

The argument `init` determines how to deal with the initial conditions:

- `init ='estimate'` means that the initial state is chosen so that the norm of prediction error is minimized. This initial state is returned as x0.
- `init = 'zero'` sets the initial state to zero.
- `init = 'model'` used the model's internally stored initial state.
- `init = x0`, where `x0` is a column vector of appropriate dimension uses that value as initial state.

If `init` is not specified, the model property `m.InitialState` is used, so that `'Estimate'`, `'Backcast'` and `'Auto'` sets `init = 'Estimate'`, while `m.InitialState = 'Zero'` sets `init = 'zero'`, and `'Fixed'` and `'Model'` sets `init = 'model'`.

The output argument `x0` is the used value of the initial state. If `data` contains several experiments, `x0` will be a matrix, containing the initial states from each experiment.

See Also

`idmodel`, `resid`

pem

Purpose

Estimate the parameters of general linear models.

Syntax

```
m = pem(data)
m = pem(data,mi)
m = pem(data,mi,'Property1',Value1,...,'PropertyN',ValueN)
m = pem(data,orders)
m = pem(data,'nx',ssorder)
m = pem(data,'na',na,'nb',nb,'nc',nc,'nd',nd,'nf',nf,'nk',nk)
m = pem(data,orders,'Property1',Value1,...,'PropertyN',ValueN)
```

Description

pem is the basic estimation command in the toolbox and covers a variety of situations.

data is always an iddata object that contains the input/output data.

With Initial Model

mi is any idmodel object, idarx, idpoly, idss, or idgrey. It could be a result of another estimation routine, or constructed and modified by the constructors (idpoly, idss, idgrey) and set. The properties of mi can also be changed by any property name/property value pairs in pem as indicated in the syntax.

m is then returned as the best fitting model in the model structure defined by mi.The iterative search is initialized at the parameters of the initial/nominal model mi. m will be of the same class as mi.

Black-Box State-Space Models

With m = pem(data,n), where n is a positive integer, or m = pem(data,'nx',n) a state-space model of order n is estimated. The default situation is that it is estimated in a 'Free' parameterization, that can be further modified by the properties 'nk', 'DisturbanceModel', and 'InitialState' (see the reference pages for idss and n4sid). The model is initialized by n4sid, and then further adjusted by optimizing the prediction error fit.

You can choose between several different orders by

```
m = pem(data,'nx',[n1,n2,...nN])
```

and you will then be prompted for the 'best' order. By

```
m = pem(data,'best')
```

an automatic choice of order among 1:10 is made.

```
m = pem(data)
```

is short for `m = pem(data,'best')`. To work with other delays use, e.g. `m = pem(data,'best','nk',[0,...0])`.

In this case `m` is returned as an `idss` model.

Black-box Multiple-Input-Single-Output Models

The function `pem` also handles the general multi-input-single-output structure

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t-nk_1) + \ldots + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t-nk_{nu}) + \frac{C(q)}{D(q)}e(t)$$

The orders of this general model are given either as

```
orders = [na nb nc nd nf nk]
```

or with `(...'na',na,'nb',nb,...)` as shown in the syntax. Here, `na`, `nb`, `nc`, `nd`, and `nf` are the orders of the model and `nk` is the delay(s). For multi-input systems, `nb`, `nf`, and `nk` are row vectors giving the orders and delays of each input. (See "Polynomial Representation of Transfer Functions" on page 3-11 in the *Tutorial* for exact definitions of the orders.) When the orders are specified with separate entries, those not given are taken as zero.

In this case `m` is returned as an `idpoly` object.

Typical Properties to set

In all cases the algorithm is affected by the properties (see `Algorithm Properties` for details):

- **`Focus:`** with possible values `'Prediction'` (Default), `'Simulation'` or a SISO filter (given as an LTI or `idmodel` object or as filter coefficients)
- **`MaxIter`** and **`Tolerance`** that govern the stopping criteria for the iterative search.
- **`LimitError`**, that deals with how the criterion can be made less sensitive to outliers and bad data
- **`MaxSize`** that determines the largest matrix ever formed by the algorithm. The algorithm goes into `for`-loops to avoid larger matrices, which may be more efficient than using virtual memory.

- **Trace**, with possible values 'Off', 'On', 'Full', that governs the information sent to the MATLAB command window.

For black-box state-space models, also 'N4Weight' and 'N4Horizon' will affect the result, since these models are initialized with n4sid estimate. See the reference page for n4sid.

Typical idmodel properties to affect are (see idmodel properties for more details):

- **Ts,** the sampling interval. Set 'Ts'=0 to obtain a continuous-time state-space model. For discrete-time models, 'Ts' is automatically set to sampling interval of the data. Note that, in the black box case, it is usually better to first estimate a discrete-time model, and then convert that to continuous time by d2c.
- **nk,** the time delays from the inputs (not applicable to structured state-space models). Time delays specified by 'nk', will be included in the model.
- **DisturbanceModel**, determines the parameterization of K for free and canonical state-space parameterizations, as well as for idgrey models.
- **InitialState**. The initial state may have a substantial influence on the estimation result for system with slow responses. It is most pronounced for Output-Error models (K=0 for state-space, and na=nc=nd=0 for input/output models). The default value 'Auto', estimates the influence of the initial state and sets the value to 'Estimate', 'Backcast' or 'Zero', based on this effect. Possible values of 'InitialState' are 'Auto', 'Estimate', 'Backcast', 'Zero' and 'Fixed'. See "Initial State" on page 3-92 in the "Tutorial" chapter.

Examples

Here is an example of a system with three inputs and two outputs. A canonical form state-space model of order 5 is sought.

```
z = iddata([y1 y2],[ u1 u2 u3]);
m = pem(z,5,'ss','can')
```

Building an ARMAX model for the response to output 2.

```
ma = pem(z(:,2,:),'na',2,'nb',[2 3 1],'nc',2,'nk',[1 2 0])
```

Comparing the models (compare automatically matches the channels using the channel names).

```
compare(z,m,ma)
```

Algorithm

pem uses essentially the same algorithm as armax with modifications to the computation of prediction errors and gradients.

See Also

armax, bj, oe, idss, idpoly, idgrey, idmodel, Algorithm Properties, EstimationInfo

plot (iddata)

Purpose

Plot input-output `iddata`.

Syntax

```
plot(data)
plot(d1,...,dN)
plot(d1,PlotStyle1,...,dN,PlotStyleN)
```

Description

`data` is the output-input data to be graphed, given as an `iddata` object. A split plot is obtained with the outputs on top and the inputs at the bottom.

One plot for each I/O channel combination is produced. Pressing the **Return** key advances the plot.

To plot a specific interval, use `plot(data(200:300))`. To plot specific input/output channels, use `plot(data(:,ky,ku))`, consistent with the subreferencing of `iddata` objects. (See `iddata`).

If `data.intersample = 'zoh'`, the input is piecewise constant between sampling points, and it is then graphed accordingly.

To plot several `iddata` sets `d1,...,dN, use plot(d1,...,dN)`. I/O channels with the same experiment name, input name and output name, will always be plotted in the same plot.

With `PlotStyle`, the color, linestyle and marker of each data set can be specified.

```
plot(d1,'y:*',d2,'b')
```

just as in the regular `plot` command.

See Also

`iddata`

Purpose

Plot a variety of model characteristics (requires the Control System Toolbox).

Syntax

```
plot(m)
plot(m('n'))
plot(m1,...,mN,Plottype)
plot(m1,PlotStyle1,...,mN,PlotStyleN)
```

Description

`m` is the output-input data to be graphed, given as any `idfrd` or `idmodel` object. After appropriate model transformations, the LTI Viewer of the Control System Toolbox is invoked. This allows e.g., `bode`, `nyquist`, `impulse`, `step`, `zero/poles` plots.

To compare several models `m1,...,mN`, use `plot(m1,...,mN)`. With `PlotStyle`, the color, linestyle and marker, of each model can be specified.

```
plot(m1,'y:*',m2,'b')
```

Adding `Plottype` as a last argument specifies the type of plot in which plot is initialized. `Plottype` is any of `'impulse'`, `'step'`, `'bode'`, `'nyquist'`, `'nichols'`, `'sigma'`, or `'pzmap'`. It can also be given as a cell array containing any collection of these strings (up to 6) in which case a multiplot is shown.

`plot` will not display confidence regions. For that use `bode`, `nyquist`, `impulse`, `step`, and `pzmap`.

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `m.NoiseVariance` = Λ. The model can also be described with unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

- `plot(m)` plots the characteristics of the transfer function G.

- `plot(m('n'))` plots the characteristics of the transfer function *HL*. (*ny* inputs and *ny* outputs).The input channels have names `v@yname`, where `yname` is the name of the corresponding output.
- If `m` is a time series, that is *nu* = 0, `plot(m)` plots the characteristics of the transfer function *HL*.
- `plot(noisecnv(m))` plots the characteristics of the transfer function [*G H*] (*nu*+*ny* inputs and *ny* outputs). The noise input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- `plot(noisecnv(m,'norm'))` plots the characteristics of the transfer function *[G HL]* (*nu*+*ny* inputs and *ny* outputs. The noise input channels have names `v@yname`, where `yname` is the name of the corresponding output.

`plot` does not give access to all of the features of `ltiview`. Use

```
ml = ss(m), ltiview(Plottype,ml,...)
```

to reach these options.

See Also

`bode`, `impulse`, `nyquist`, `step`, `pzmap`

Purpose

Convert a model to input-output polynomials.

Syntax

```
[A,B,C,D,F] = polydata(m)
[A,B,C,D,F,dA,dB,dC,dD,dF] = polydata(m)
```

Description

This is essentially the inverse of the `idpoly` constructor. It returns the polynomials of the general model

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t - nk_1) + \ldots + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t - nk_{nu}) + \frac{C(q)}{D(q)}e(t)$$

as contained in the model `m`.

`dA`, `dB` etc. are the standard deviations of `A`, `B`, etc.

`m` can be any single output idmodel, i.e., not just `idpoly`. For multi-output models you can use `[A,B,C,D,F] = polydata(m(ky,:))` to obtain the polynomials for the `ky`-th output.

See Also

`idmodel`, `idpoly`, `tfdata`

predict

Purpose

Predict the output *k* steps ahead.

Syntax

```
yp = predict(m,data)
[yp,mpred] = predict(m,data,k,init)
```

Description

`data` is the output-input data as an iddata object, and `m` is any `idmodel` object (`idpoly`, `idss`, `idgrey`, or `idarx`)

The argument `k` indicates that the *k*-step ahead prediction of `y` according to the model `m` is computed. In the calculation of *yp(t)* the model can use outputs up to time

$$t-k: y(s), s = t-k, t-k-1, \ldots$$

and inputs up to the current time *t*. The default value of `k` is 1.

The output `yp` is an `iddata` object containing the predicted values as `OutputData`. The output argument `mpred` contains the *k*-step ahead predictor as an `idss` model object. (The predictor is a system with $ny + nu$ inputs and ny outputs, ny being the number of outputs and nu the number of inputs to `th`.)

`init` determines how to deal with the initial state:

- `init ='estimate'`: The initial state is set to value that minimizes the norm of the prediction error associated with the model and the data.
- `init = 'zero'` sets the initial state to zero.
- `init = 'model'` used the model's internally stored initial state
- `init = x0`, where `x0` is a column vector of appropriate dimension uses that value as initial state

If `init` is not specified, the model property `m.InitialState` is used, so that `'Estimate'`, `'Backcast'` and `'Auto'` sets `init = 'Estimate'`, while `m.InitialState = 'Zero'` sets `init = 'zero'`, and `'Model'` and `'Fixed'` sets `init = 'model'`.

An important use of `predict` is to evaluate a model's properties in the mid-frequency range. Simulation with `sim` (which conceptually corresponds to `k = inf`) can lead to levels that drift apart, since the low frequency behavior is emphasized. One step ahead prediction is not a powerful test of the model's properties, since the high frequency behavior is stressed. The trivial predictor

$\hat{y}(t) = y(t-1)$ can give good predictions in case the sampling of the data is fast.

Another important use of `predict` is to evaluate models of time series. The natural way of studying a time-series model's ability to reproduce observations is to compare its *k*-step ahead predictions with actual data.

Note that for Output-Error models, there is no difference between the *k*-step ahead predictions and the simulated output, since, by definition, Output-Error models only use past inputs to predict future outputs.

Algorithm

The model is evaluated in state-space form, and the state equations are simulated *k*-steps ahead with initial value $x(t-k) = \hat{x}(t-k)$, where $\hat{x}(t-k)$ is the Kalman filter state estimate.

Examples

Simulate a time series, estimate a model based on the first half of the data, and evaluate the four step ahead predictions on the second half.

```
m0 = idpoly([1 -0.99],[],[1 -1 0.2]);
e = iddata([],randn(400,1));
y = sim(m0,e);
m = armax(y(1:200),[1 2]);
yp = predict(m,y,4);
plot(y(201:400),yp(201:400))
```

Note that the last two commands also are achieved by

```
compare(y,m,4,201:400);
```

See Also

`compare`, `sim`, `pe`

present

Purpose

Display the information in an `idmodel` model, including uncertainty.

Syntax

```
present(m)
```

Description

This function displays the model `m`, together with the estimated standard deviations of the parameters, loss function, and Akaike's Final Prediction Error (FPE) Criterion (which essentially equals the AIC). It also displays information about how `m` was created.

`present` thus gives more detailed information about the model than the standard display function.

Purpose

Plot zeros and poles.

Syntax

```
pzmap(m)
pzmap(m,'sd',sd)
pzmap(m1,m2,m3,...)
pzmap(m1,'PlotStyle1',m2,'PlotStyle2',...,'sd',sd)
pzmap(m1,m2,m3,..,'sd',sd,mode,axis)
```

Description

`m` is any `idmodel` object: `idarx`, `idgrey`, `idss`, or `idpoly`.

The zeros and poles of `m` are graphed, with `o` denoting zeros and `x` denoting poles. Poles and zeros at infinity are ignored. For discrete-time models, zeros and poles at the origin are also ignored.

If `sd` has a value larger than zero, confidence regions around the poles and zeros are also graphed. The regions corresponding to `sd` standard deviations are marked. The default value is `sd = 0`. Note that the confidence regions may sometimes stretch outside the plot, but they are always symmetric around the indicated zero or pole.

If the poles and zeros are associated with a discrete-time model, a unit circle is also drawn. For continuous-time models the real and imaginary axes are drawn

When `mi` contain information about several different input/output channels, there are some options:

`mode = 'sub'` splits the screen into several plots, one for each input/output channel. These are based on the `InputName` and `OutputName` properties associated with the different models.

`mode = 'same'` gives all plots in the same diagram. Pressing the **Return** key advances the plots.

`mode = 'sep'` erases the previous plot before the next channel pair is treated.

The default value is `mode = 'sub'`.

`axis = [x1 x2 y1 y2]` fixes the axis scaling accordingly. `axis = s` is the same as

```
axis = [-s s -s s]
```

The colors associated with the different models can be selected by the arguments `PlotStyle`. Use `PlotStyle = 'b'`, `'g'`, etc. Markers and line styles are not used.

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `m.NoiseVariance` = Λ. The model can also be described with unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

Then

- `pzmap(m)` plots the zeros and poles of the transfer function G.
- `pzmap(m('n'))` plots the zeros and poles of the transfer function H. (ny inputs and ny outputs). The input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- If `m` is a time series, that is $nu = 0$, `pzmap(m)` plots the zeros and poles of the the transfer function H.
- `pzmap(noisecnv(m))` plots the zeros and poles of the transfer function $[G\ H]$ ($nu+ny$ inputs and ny outputs). The noise input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- `pzmap(noisecnv(m,'norm'))`lots the zeros and poles of the transfer function $[G\ HL]$ ($nu+ny$ inputs and ny outputs. The noise input channels have names `v@yname`, where `yname` is the name of the corresponding output.

Examples

```
mbj = bj(data,[2 2 1 1 1]);
mar = armax(data,[2 2 2 1]);
pzmap(mbj,mar,'sd',3)
```

shows all zeros and poles of two models along with the confidence regions corresponding to three standard deviations.

See Also

`idmodel`, `zpkdata`

Purpose

Estimate recursively the parameters of an ARMAX or ARMA model.

Syntax

```
thm = rarmax(z,nn,adm,adg)
[thm,yhat,P,phi,psi] = rarmax(z,nn,adm,adg,th0,P0,phi0,psi0)
```

Description

The parameters of the ARMAX model structure

$$A(q)y(t) = B(q)u(t-nk) + C(q)e(t)$$

are estimated using a recursive prediction error method.

The input-output data are contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. `nn` is given as

```
nn = [na nb nc nk]
```

where `na`, `nb`, and `nc` are the orders of the ARMAX model, and `nk` is the delay. Specifically,

na: $A(q) = 1 + a_1 q^{-1} + \ldots + a_{na} q^{-na}$

nb: $B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$

nc: $C(q) = 1 + c_1 q^{-1} + \ldots + c_{nc} q^{-nc}$

See “Polynomial Representation of Transfer Functions” on page 3-11 in the “Tutorial” chapter for more information.

If `z` represents a time series `y` and `nn = [na nc]`, `rarmax` estimates the parameters of an ARMA model for `y`.

$$A(q)y(t) = C(q)e(t)$$

Only single-input, single-output models are handled by `rarmax`. Use `rpem` for the multi-input case.

The estimated parameters are returned in the matrix `thm`. The `k`-th row of `thm` contains the parameters associated with time `k`, i.e., they are based on the data in the rows up to and including row `k` in `z`. Each row of `thm` contains the estimated parameters in the following order.

```
thm(k,:) = [a1,a2,...,ana,b1,...,bnb,c1,...,cnc]
```

`yhat` is the predicted value of the output, according to the current model, i.e., row *k* of `yhat` contains the predicted value of `y(k)` based on all past data.

The actual algorithm is selected with the two arguments `adm` and `adg`. These are described under `rarx`.

The input argument `th0` contains the initial value of the parameters, a row vector, consistent with the rows of `thm`. The default value of `th0` is all zeros.

The arguments `P0` and `P` are the initial and final values, respectively, of the scaled covariance matrix of the parameters. See `rarx`. The default value of `P0` is 10^4 times the unit matrix. The arguments `phi0`, `psi0`, `phi`, and `psi` contain initial and final values of the data vector and the gradient vector, respectively. The sizes of these depend in a rather complicated way on the chosen model orders. The normal choice of `phi0` and `psi0` is to use the outputs from a previous call to `rarmax` with the same model orders. (This call could of course be a *dummy* call with default input arguments.) The default values of `phi0` and `psi0` are all zeros.

Note that the function requires that the delay `nk` be larger than 0. If you want `nk=0`, shift the input sequence appropriately and use `nk=1`.

Algorithm

The general recursive prediction error algorithm (11.44), (11.47)-(11.49) of Ljung (1999) is implemented. See “Recursive Parameter Estimation” on page 3-79 in the “Tutorial” chapter for more information.

Examples

Compute and plot, as functions of time, the four parameters in a second order ARMA model of a time series given in the vector `y`. The forgetting factor algorithm with a forgetting factor of 0.98 is applied.

```
thm = rarmax(y,[2 2],'ff',0.98);
plot(thm)
```

Purpose

Estimate recursively the parameters of an ARX or AR model.

Syntax

```
thm = rarx(z,nn,adm,adg)
[thm,yhat,P,phi] = rarx(z,nn,adm,adg,th0,P0,phi0)
```

Description

The parameters of the ARX model structure

$$A(q)y(t) = B(q)u(t-nk) + e(t)$$

are estimated using different variants of the recursive least-squares method.

The input-output data are contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. `nn` is given as

```
nn = [na nb nk]
```

where `na` and `nb` are the orders of the ARX model, and `nk` is the delay. Specifically,

$$na: \quad A(q) = 1 + a_1 q^{-1} + \ldots + a_{na} q^{-na}$$

$$nb: \quad B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$$

See equation (Equation 3-13) in the “Tutorial” chapter for more information.

If `z` is a time series `y` and `nn = na`, `rarx` estimates the parameters of an AR model for `y`.

$$A(q)y(t) = e(t)$$

Models with several inputs

$$A(q)y(t) = B_1(q)u_1(t-nk_1) + \ldots B_{nu}u_{nu}(t-nk_{nu}) + e(t)$$

are handled by allowing `u` to contain each input as a column vector,

```
u = [u1 ...  unu]
```

and by allowing `nb` and `nk` to be row vectors defining the orders and delays associated with each input.

Only single-output models are handled by `rarx`.

The estimated parameters are returned in the matrix `thm`. The `k`-th row of `thm` contains the parameters associated with time `k`, i.e., they are based on the data in the rows up to and including row `k` in `z`. Each row of `thm` contains the estimated parameters in the following order:

```
thm(k,:) = [a1,a2,...,ana,b1,...,bnb]
```

In the case of a multi-input model, all the *b* parameters associated with input number 1 are given first, and then all the *b* parameters associated with input number 2, and so on.

`yhat` is the predicted value of the output, according to the current model, i.e., row `k` of `yhat` contains the predicted value of `y(k)` based on all past data.

The actual algorithm is selected with the two arguments `adg` and `adm`. These are described in "Recursive Parameter Estimation" on page 3-79 in the *Tutorial*. The options are as follows:

With `adm ='ff'` and `adg = lam` the *forgetting factor* algorithm (Equation 3-60abd)+(Equation 3-62) is obtained with forgetting factor $\lambda =$ `lam`. This is what is often referred to as Recursive Least Squares, RLS. In this case the matrix `P` (see below) has the following interpretation: $R_2/2 *$ `P` is approximately equal to the covariance matrix of the estimated parameters. Here R_2 is the variance of the innovations (the true prediction errors *e(t)* in (Equation 3-57)

With `adm ='ug'` and `adg = gam`, the *unnormalized gradient* algorithm (Equation 3-60abc) + (Equation 3-63) is obtained with gain *gamma*= `gam`. This algorithm is commonly known as normalized Least Mean Squares, LMS.

Similarly, `adm ='ng'` and `adg = gam` give the *normalized gradient* or Normalized Least Mean Squares, NLMS algorithm (Equation 3-60abc) + (Equation 3-64). In these cases, `P` is not defined or applicable.

With `adm ='kf'` and `adg = R1` the *Kalman Filter Based* algorithm (Equation 3-60) is obtained with $R_2 = 1$ and $R_1 =$ `R1`. If the variance of the innovations *e(t)* is not unity but R_2; then $R_2 *$ `P` is the covariance matrix of the parameter estimates, while $R_1 =$`R1` / R_2 is the covariance matrix of the parameter changes in (Equation 3-58).

The input argument th0 contains the initial value of the parameters; a row vector, consistent with the rows of thm. (See above.) The default value of th0 is all zeros.

The arguments P0 and P are the initial and final values, respectively, of the scaled covariance matrix of the parameters. The default value of P0 is 10^4 times the identity matrix.

The arguments phi0 and phi contain initial and final values, respectively, of the data vector.

$$\varphi(t) = [y(t-1), \ldots, y(t-na), u(t-1), \ldots u(t-nb-nk+1)]$$

Then, if

```
z = [y(1),u(1); ...  ;y(N),u(N)]
```

you have phi0 = $\varphi(1)$ and phi = $\varphi(N)$. The default value of phi0 is all zeros. For online use of rarx, use phi0, th0, and P0 as the previous outputs phi, thm (last row), and P.

Note that the function requires that the delay nk be larger than 0. If you want nk=0, shift the input sequence appropriately and use nk=1. See nkshift.

Examples

Adaptive noise canceling: The signal *y* contains a component that has its origin in a known signal *r*. Remove this component by estimating, recursively, the system that relates *r* to *y* using a sixth order FIR model together with the NLMS algorithm.

```
z = [y r];
[thm,noise] = rarx(z,[0 6 1],'ng',0.1);
%noise is the adaptive estimate of the noise
%component of y
plot(y-noise)
```

If the above application is a true online one, so that you want to plot the best estimate of the signal y - noise at the same time as the data *y* and *u* become available, proceed as follows.

```
phi = zeros(6,1); P=1000*eye(6);
th = zeros(1,6); axis([0 100 -2 2]);
plot(0,0,'*'), hold on
%The loop:
```

```
while ~abort
[y,r,abort] = readAD(time);
[th,ns,P,phi] = rarx([y r],'ff',0.98,th,P,phi);
plot(time,y-ns,'*')
time = time +Dt
end
```

This example uses a forgetting factor algorithm with a forgetting factor of 0.98. `readAD` represents an M-file that reads the value of an A/D converter at the indicated time instant.

Purpose

Estimate recursively the parameters of a Box-Jenkins model.

Syntax

```
thm = rbj(z,nn,adm,adg)
[thm,yhat,P,phi,psi] = ... rbj(z,nn,adm,adg,th0,P0,phi0,psi0)
```

Description

The parameters of the ARMAX model structure

$$y(t) = \frac{B(q)}{F(q)}u(t-nk) + \frac{C(q)}{D(q)}e(t)$$

are estimated using a recursive prediction error method.

The input-output data are contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. `nn` is given as

```
nn = [nb nc nd nf nk]
```

where `nb`, `nc`, `nd`, and `nf` are the orders of the Box-Jenkins model, and `nk` is the delay. Specifically,

nb: $B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$

nc: $C(q) = 1 + c_1 q^{-1} + \ldots + c_{nc} q^{-nc}$

nd: $D(q) = 1 + d_1 q^{-1} + \ldots + d_{nd} q^{-nd}$

nf: $F(q) = 1 + f_1 q^{-1} + \ldots + f_{nf} q^{-nf}$

See “Polynomial Representation of Transfer Functions” on page 3-11 in the “Tutorial” chapter for more information.

Only single-input, single-output models are handled by `rbj`. Use `rpem` for the multi-input case.

The estimated parameters are returned in the matrix `thm`. The `k`-th row of `thm` contains the parameters associated with time `k`, i.e., they are based on the data in the rows up to and including row `k` in `z`. Each row of `thm` contains the estimated parameters in the following order:

```
thm(k,:) = [b1,...,bnb,c1,...,cnc,d1,...,dnd,f1,...,fnf]
```

yhat is the predicted value of the output, according to the current model, i.e., row k of yhat contains the predicted value of y(k) based on all past data.

The actual algorithm is selected with the two arguments adm and adg. These are described under rarx.

The input argument th0 contains the initial value of the parameters, a row vector, consistent with the rows of thm. (See above.) The default value of th0 is all zeros.

The arguments P0 and P are the initial and final values, respectively of the scaled covariance matrix of the parameters. See rarx. The default value of P0 is 10^4 times the unit matrix. The arguments phi0, psi0, phi, and psi contain initial and final values of the data vector and the gradient vector, respectively. The sizes of these depend in a rather complicated way on the chosen model orders. The normal choice of phi0 and psi0 is to use the outputs from a previous call to rbj with the same model orders. (This call could, of course, be a dummy call with default input arguments.) The default values of phi0 and psi0 are all zeros.

Note that the function requires that the delay nk is larger than 0. If you want nk=0, shift the input sequence appropriately and use nk=1.

Algorithm

The general recursive prediction error algorithm (11.44) of Ljung (1900) is implemented. See also “Recursive Parameter Estimation” on page 3-79 in the “Tutorial” chapter.

Purpose

Resample data by interpolation and decimation.

Syntax

```
zr = resample(z,R)
[zr, R_act] = resample(z,R,filter_order,tol)
```

Description

`z`: The data to be resampled, given as an `iddata` object

`zr`: The resampled data returned as an `iddata` object

`R`: The resampling factor. The new data record will correspond to a new sampling interval of `R` times the original one. `R > 1` thus corresponds to decimation and `R < 1` corresponds to interpolation. Any positive real number for `R` is allowed, but it will be replaced by a rational approximation (`R_act`).

`R_act`: The actually achieved resampling factor.

`filter_order`: The order of the presampling filters used before interpolation and decimation. Default is `8`.

`tol`: The tolerance in the rational approximation of `R`. Default is `0.1`.

Note To use the `resample` function, you must have the Signal Processing Toolbox installed.

Caution

For signals that have much energy around the Nyquist frequency (like piece-wise constant inputs), the resampled waveform may look "very different," due to the prefiltering effects. The frequency and information contents for identification is, however, not mishandled.

Example

Resample by a factor 1.5 and compare the signals.

```
plot(u)
ur = resample(u,1.5);
plot(u,ur)
```

resid

Purpose

Compute and test the residuals (prediction errors) of a model.

Syntax

```
resid(m,data)
resid(m,data,Type)
resid(m,data,Type,M)
e = resid(m,data);
```

Description

`data` contains the output-input data as an `iddata` object

`m` is the model to be evaluated on the given data set. It is any `idmodel` object.

In all cases the residuals *e* associated with the data and the model are computed. This is done as in the command `pe` with a default choice of `init`.

When called without output arguments, `resid` produces a plot. The plot can be of three kinds depending on the argument `Type`:

- `Type = 'Corr'` (default): The autocorrelation function of `e` and the cross correlation between `e` and the input(s) `u` are computed and displayed. The 99% confidence intervals for these values are also computed and shown as a yellow region. The computation of the confidence region is done assuming `e` to be white and independent of `u`. The functions are displayed up to lag `M`, which is 25 by default.
- `Type = 'ir'`: The impulse response (up to lag `M`, which is 25 by default) from the input to the residuals is plotted with a 99% confidence region around zero marked as a yellow area. Negative lags up tp `M`/4 are also included to investigate feedback effects. (The result is the same as `impulse(e,'sd',2.58,'fill',M)`.)
- `Type = 'fr'`: The frequency response from the input to the residuals (based on a high order FIR model) is shown as a Bode plot. A 99% confidence region around zero is also marked as a yellow area.

With an output argument, no plot is produced, and `e` is returned with the residuals (prediction errors) associated with the model and the data. It is an `iddata` object with the residuals as outputs and the input in `data` as inputs. That means that `e` can be directly used to build model error models, i.e., models that describe the dynamics from the input to the residuals (which should be negligible if `m` is a good description of the system).

See “Model Structure Selection and Validation” on page 3-64 in the “Tutorial” chapter for more information.

Examples

Here are some typical model validation commands:

```
e = resid(m,data);
plot(e)
compare(data,m);
```

To compute a “model error model,” that is, a model to input to the residuals to see if any essential unmodeled dynamics are left,

```
e = resid(m,data);
me = arx(e,[10 10 0]);
bode(me,'sd',3,fill')
```

See Also

`compare`, `idgrey`, `idarx`, `idpoly`, `idss`, `pem`

References

Ljung (1999), Section 16.6.

Purpose

Estimate recursively the parameters of an Output-Error model.

Syntax

```
thm = roe(z,nn,adm,adg)
[thm,yhat,P,phi,psi] = roe(z,nn,adm,adg,th0,P0,phi0,psi0)
```

Description

The parameters of the Output-Error model structure

$$y(t) = \frac{B(q)}{F(q)}u(t-nk)$$

are estimated using a recursive prediction error method.

The input-output data are contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. `nn` is given as

```
nn = [nb nf nk]
```

where `nb` and `nf` are the orders of the Output-Error model, and `nk` is the delay. Specifically,

nb: $B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$

nf: $F(q) = 1 + f_1 q^{-1} + \ldots + f_{nf} q^{-nf}$

See “Polynomial Representation of Transfer Functions” on page 3-11 in the *Tutorial* for more information.

Only single-input, single-output models are handled by `roe`. Use `rpem` for the multi-input case.

The estimated parameters are returned in the matrix `thm`. The `k`-th row of `thm` contains the parameters associated with time `k`, i.e., they are based on the data in the rows up to and including row `k` in `z`.

Each row of `thm` contains the estimated parameters in the following order:

```
thm(k,:) = [b1,...,bnb,f1,...,fnf]
```

`yhat` is the predicted value of the output, according to the current model, i.e., row `k` of `yhat` contains the predicted value of `y(k)` based on all past data.

The actual algorithm is selected with the two arguments adg and adm. These are described under rarx.

The input argument th0 contains the initial value of the parameters, a row vector, consistent with the rows of thm. (See above.) The default value of th0 is all zeros.

The arguments P0 and P are the initial and final values, respectively, of the scaled covariance matrix of the parameters. See rarx. The default value of P0 is 10^4 times the unit matrix. The arguments phi0, psi0, phi, and psi contain initial and final values of the data vector and the gradient vector, respectively. The sizes of these depend in a rather complicated way on the chosen model orders. The normal choice of phi0 and psi0 is to use the outputs from a previous call to roe with the same model orders. (This call could be a dummy call with default input arguments.) The default values of phi0 and psi0 are all zeros.

Note that the function requires that the delay nk is larger than 0. If you want nk=0, shift the input sequence appropriately and use nk=1.

Algorithm

The general recursive prediction error algorithm (11.44) of Ljung (1999) is implemented. See also “Recursive Parameter Estimation” on page 3-79 in the “Tutorial” chapter.

See Also

oe, rarx, rbj, rplr, rpem, nkshift

rpem

Purpose

Estimate recursively the parameters of a general multi-input single-output linear model.

Syntax

```
thm = rpem(z,nn,adm,adg)
[thm,yhat,P,phi,psi] = rpem(z,nn,adm,adg,th0,P0,phi0,psi0)
```

Description

The parameters of the general linear model structure

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t-nk_1) + \ldots + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t-nk_{nu}) + \frac{C(q)}{D(q)}e(t)$$

are estimated using a recursive prediction error method.

The input-output data are contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. (In the multi-input case `u` contains one column for each input). `nn` is given as

```
nn = [na nb nc nd nf nk]
```

where `na`, `nb`, `nc`, `nd`, and `nf` are the orders of the model, and `nk` is the delay. For multi-input systems `nb`, `nf`, and `nk` are row vectors giving the orders and delays of each input. See "Polynomial Representation of Transfer Functions" on page 3-11 in the "Tutorial" chapter for an exact definition of the orders.

The estimated parameters are returned in the matrix `thm`. The `k`-th row of `thm` contains the parameters associated with time `k`, i.e., they are based on the data in the rows up to and including row `k` in `z`. Each row of `thm` contains the estimated parameters in the following order.

```
thm(k,:) = [a1,a2,...,ana,b1,...,bnb,...
  c1,...,cnc,d1,...,dnd,f1,...,fnf]
```

For multi-input systems the *B* part in the above expression is repeated for each input before the *C* part begins, and also the *F* part is repeated for each input. This is the same ordering as in `m.par`.

`yhat` is the predicted value of the output, according to the current model, i.e., row *k* of `yhat` contains the predicted value of `y(k)` based on all past data.

The actual algorithm is selected with the two arguments `adg` and `adm`. These are described under `rarx`.

The input argument th0 contains the initial value of the parameters;, a row vector, consistent with the rows of thm. (See above.) The default value of th0 is all zeros.

The arguments P0 and P are the initial and final values, respectively, of the scaled covariance matrix of the parameters. See rarx. The default value of P0 is 10^4 times the unit matrix. The arguments phi0, psi0, phi, and psi contain initial and final values of the data vector and the gradient vector, respectively. The sizes of these depend in a rather complicated way on the chosen model orders. The normal choice of phi0 and psi0 is to use the outputs from a previous call to rpem with the same model orders. (This call could be a dummy call with default input arguments.) The default values of phi0 and psi0 are all zeros.

Note that the function requires that the delay nk is larger than 0. If you want nk=0 , shift the input sequence appropriately and use nk=1.

Algorithm

The general recursive prediction error algorithm (11.44) of Ljung (1999) is implemented. See also “Recursive Parameter Estimation” on page 3-79 in the “Tutorial” chapter.

For the special cases of ARX/AR models, and of single-input ARMAX/ARMA, Box-Jenkins, and Output-Error models, it is more efficient to use rarx, rarmax, rbj, and roe.

See Also

pem, rarmax, rarx, rbj, roe, rplr, nkshift

rplr

Purpose

Estimate recursively the parameters of a general multi-input single-output linear model.

Syntax

```
thm = rplr(z,nn,adm,adg)
[thm,yhat,P,phi] = rplr(z,nn,adm,adg,th0,P0,phi0)
```

Description

This is a direct alternative to `rpem` and has essentially the same syntax. See `rpem` for an explanation of the input and output arguments.

`rplr` differs from `rpem` in that it uses another gradient approximation. See Section 11.5 in Ljung (1999). Several of the special cases are well known algorithms.

When applied to ARMAX models, (`nn = [na nb nc 0 0 nk]`), `rplr` gives the Extended Least Squares (ELS) method. When applied to the output error structure (`nn = [0 nb 0 0 nf nk]`) the method is known as HARF in the `adm = 'ff'` case and SHARF in the `adm = 'ng'` case.

`rplr` can also be applied to the time-series case in which an ARMA model is estimated with

```
z = y
nn = [na nc]
```

You can thus use `rplr` as an alternative to `rarmax` for this case.

See Also

`pem`, `rarmax`, `rarx`, `rbj`, `roe`, `rpem`

Purpose

Segment data and estimate models for each segment.

Syntax

```
segm = segment(z,nn)
[segm,V,thm,R2e] = segment(z,nn,R2,q,R1,M,th0,P0,ll,mu)
```

Description

segment builds models of AR, ARX, or ARMAX/ARMA type,

$$A(q)y(t) = B(q)u(t-nk) + C(q)e(t)$$

assuming that the model parameters are piece-wise constant over time. It results in a model that has split the data record into segments over which the model remains constant. The function models signals and systems that may undergo abrupt changes.

The input-output data are contained in z, which is either an iddata object or a matrix z = [y u] where y and u are column vectors. If the system has several inputs, u has the corresponding number of columns.

The argument nn defines the model order. For the ARMAX model

```
nn = [na nb nc nk]
```

where na, nb, and nc are the orders of the corresponding polynomials. See "Polynomial Representation of Transfer Functions" on page 3-11 in the "Tutorial" chapter. Moreover nk is the delay. If the model has several inputs, nb and nk are row vectors, giving the orders and delays for each input.

For an ARX model (nc = 0) enter

```
nn = [na nb nk]
```

For an ARMA model of a time series

```
z = y
nn = [na nc]
```

and for an AR model

```
nn = na
```

The output argument segm is a matrix, whose k-row contains the parameters corresponding to time k. This is analogous to the output argument thm in rarx and rarmax. The output argument thm of segment contains the corresponding model parameters that have not yet been segmented. That is, they are not

piecewise constant, and therefore correspond to the outputs of the functions `rarmax` and `rarx`. In fact, `segment` is an alternative to these two algorithms, and has a better capability to deal with time variations that may be abrupt.

The output argument `V` contains the sum of the squared prediction errors of the segmented model. It is a measure of how successful the segmentation has been.

The input argument `R2` is the assumed variance of the innovations *e(t)* in the model. The default value of `R2`, `R2 = []`, is that it is estimated. Then the output argument `R2e` is a vector whose `k`-th element contains the estimate of `R2` at time `k`.

The argument `q` is the probability that the model undergoes at an abrupt change at any given time. The default value is 0.01.

`R1` is the assumed covariance matrix of the parameter jumps when they occur. The default value is the identity matrix with dimension equal to the number of estimated parameters.

`M` is the number of parallel models used in the algorithm (see below). Its default value is 5.

`th0` is the initial value of the parameters. Its default is zero. `P0` is the initial covariance matrix of the parameters. The default is 10 times the identity matrix.

`ll` is the guaranteed life of each of the models. That is, any created candidate model is not abolished until after at least `ll` time steps. The default is `ll = 1`. `Mu` is a forgetting parameter that is used in the scheme that estimates `R2`. The default is `0.97`.

The most critical parameter for you to choose is `R2`. It is usually more robust to have a reasonable guess of `R2` than to estimate it. Typically, you need to try different values of `R2` and evaluate the results. (See the example below.) `sqrt(R2)` corresponds to a change in the value *y(t)* that is normal, giving no indication that the system or the input might have changed.

Algorithm

The algorithm is based on `M` parallel models, each recursively estimated by an algorithm of Kalman filter type. Each is updated independently, and its posterior probability is computed. The time varying estimate `thm` is formed by weighting together the `M` different models with weights equal to their posterior probability. At each time step the model (among those that have lived at least `ll` samples) that has the lowest posterior probability is abolished. A new model is started, assuming that the system parameters have jumped, with probability `q`, a random jump from the most likely among the models. The covariance matrix of the parameter change is set to `R1`.

After all the data are examined, the surviving model with the highest posterior probability is tracked back and the time instances where it jumped are marked. This defines the different segments of the data. (If no models had been abolished in the algorithm, this would have been the maximum likelihood estimates of the jump instances.) The segmented model `segm` is then formed by smoothing the parameter estimate, assuming that the jump instances are correct. In other words, the last estimate over a segment is chosen to represent the whole segment.

Examples

Check how the algorithm segments a sinusoid into segments of constant levels. Then use a very simple model $y(t) = b_1 * 1$, where 1 is a fake input and b_1 describes the piecewise constant level of the signal $y(t)$ (which is simulated as a sinusoid).

```
y = sin([1:50]/3)';
thm = segment([y,ones(y)],[0 1 1],0.1);
plot([thm,y])
```

By trying various values of `R2` (0.1 in the above example), more levels are created as `R2` decreases.

selstruc

Purpose

Select model order (structure).

Syntax

```
[nn,vmod] = selstruc(v)
[nn,vmod] = selstruc(v,c)
```

Description

`selstruc` is a function to help choose a model structure (order) from the information contained in the matrix `v` obtained as the output from `arxstruc` or `ivstruc`.

The default value of `c` is `'plot'`. The plot shows the percentage of the output variance that is not explained by the model, as a function of the number of parameters used. Each value shows the best fit for that number of parameters. By clicking in the plot you can examine which orders are of interest. By clicking on 'Select' the variable `nn` is returned in the workspace as the optimal model structure for your choice of number of parameters. Several choices can be made.

`c = 'aic'` gives no plots, but returns in `nn` the structure that minimizes Akaike's Information Criterion (AIC),

$$V_{mod} = V\left(1 + \frac{2d}{N}\right)$$

where V is the loss function, d is the total number of parameters in the structure in question, and N is the number of data points used for the estimation.

`c = 'mdl'` returns in `nn` the structure that minimizes Rissanen's Minimum Description Length (MDL) criterion.

$$V_{mod} = V\left(1 + \frac{d\log(N)}{N}\right)$$

When `c` equals a numerical value, the structure that minimizes

$$V_{mod} = V\left(1 + \frac{cd}{N}\right)$$

is selected.

The output argument `vmod` has the same format as `v`, but it contains the logarithms of the accordingly modified criteria.

Example

```
V = arxstruc(data(1:200),data(201:400),struc(1:10,1:10,1:10))
nn = selstruc(V,0); %best fit to validation data
m = arx(data,nn)
```

set

Purpose

Set or modify the properties of models and iddata sets

Syntax

```
set(m,'Property',Value)
set(m,'Property1',Value1,...'PropertyN',ValueN)
set(m,'Property')
set(m)
```

Description

set is used to set or modify the properties of any of the objects in the toolbox (iddata, idmodel, idgrey, idarx, idpoly, idss). See the corresponding reference entries for a complete list of properties.

`set(m,'Property',Value)` assigns the value `Value` to the property of the object `m`, specified by the string `'Property'`. This string can be the full property name (e.g., `'SSParameterization'`) or any unambiguous case-insensitive abbreviation (e.g., `'ss'`).

`set(m,'Property1',Value1,...'PropertyN',ValueN)` sets multiple properties with a single statement. In certain cases this may be necessary, since the model `m` must, e.g., have state-space matrices of consistent dimensions after each set statement.

`set(m,'Property')` displays admissible values for the property specified by `'Property'`.

`set(m)` displays all assignable values of `m` and their admissible values.

The same result is also obtained by subassignment:

```
m.Property = Value
```

Purpose

Assign mnemonic parameter names to black box model structures.

Syntax

```
model = setpname(model)
```

Description

`model` is an `idmodel` object of `idarx`, `idpoly` or `idss` type. The returned model has the 'PName' property set to a cell array of strings that correspond to the symbols used in this manual to describe the parameters.

For single input `idpoly` models, the parameters are called `'a1'`, `'a2'`, `...`,`'fn'`, just as defined in Section "Polynomial Representation of Transfer Functions" on page 3-11.

For multi-input `idpoly` models, the *b*- and *f*-parameters have the output/input channel number in parenthesis as in `'b1(1,2)'`, `'f3(1,2)'` etc.

For `idarx` models, the parameter names are as in `'Ar(ky,ku)'` for the *ky*-*ku* entry of the matrix in (Equation 3-46) and similarly for the *B*-parameters.

For `idss` models the parameters are named for the matrix entries they represent, like `'A(4,5)'`, `'K(2,3)'` etc.

This function is particularly useful when certain parameters are to be fixed. See the property `'FixedParameter'` under `Algorithm Properties`.

sim

Purpose

Simulate linear models.

Syntax

```
y = sim(m,ue)
[y, ysd] = sim(m,ue,init)
```

Description

`m` is an arbitrary `idmodel` object.

`ue` is an `iddata` object, containing inputs only. The number of input channels in `ue` must either be equal to the number of inputs of the model `m`, or equal to the sum of the number of inputs and noise sources (= number of outputs). In the latter case the last inputs in `ue` are regarded as noise sources and a noise-corrupted simulation is obtained. The noise is scaled according to the property `m.NoiseVariance` in `m`, so in order to obtain the right noise level according to the model, the noise inputs should be white noise with zero mean and unit covariance matrix. If no noise sources are contained in `ue`, a noise-free simulation is obtained.

`sim` returns `y` containing the simulated output, as an `iddata` object.

`init` gives access to the initial states:

- `init = 'm'` (default) uses the model `m`'s internally stored initial state.
- `init = 'z'` uses zero initial state.
- `init = x0`, where `x0` is a column vector of appropriate length uses this value as the initial state.

The second output argument `ysd` is the standard deviation of the simulated output.

If `m` is a continuous-time model, it is first converted to discrete time with the sampling interval given by `ue` taking into account the intersample behavior of the input (`ue.InterSample`). See the section"Discrete and Continuous Time Models" on page 3-61 in the "Tutorial" chapter.

Examples

Simulate a given system `m0` (for example created by `idpoly`).

```
e = randn(500,1);
u = iddata([],idinput(500,'prbs'));
y = sim(m0,[u e]);
z = [y u]; % An iddata object with y as output and u as input.
```

Validate a model by comparing a measured output y with one simulated using an estimated model m.

```
yh = sim(m,u);
plot(y,yh)
```

See Also

iddata, idpoly, idarx, idss, idgrey, simsd

simsd

Purpose

Simulate models with uncertainty.

Syntax

```
simsd(m,u)
simsd(m,u,N,noise,Ky)
```

Description

`u` is an `iddata` object containing the inputs. `m` is a model given as any `idmodel` object. `N` random models are created, according to the covariance information given in `m`. The responses of each of these models to the input `u` are computed and graphed in the same diagram. If `noise = 'noise'`, noise is added to the simulation, in accordance with the noise model of `m`, and its own uncertainty. `Ky` denotes the output numbers to be plotted (default all)

The default values are

```
N = 10
noise = 'nonoise'
```

Examples

Plot the step response of the model `m` and evaluate how it varies in view of the model's uncertainty.

```
step1 = [zeros(5,1); ones(20,1)];
simsd(m,step1)
```

See Also

`sim`

Purpose

Dimensions of `iddata`. `idfrd` and `idmodel` objects

Syntax

```
d = size(m)
[ny,nu,Npar,Nx] = size(model)
[N, ny, nu, Nexp] = size(data)
ny = size(data,2)
ny = size(data,'ny')
size(model)
```

Description

`size` describes the dimensions of both model and `iddata` objects.

For `iddata` objects, the sizes returned are, in this order:

`N` = the length of the data record. For multiexperiment data, `N` is a row vector with as many entries as there are experiments.

`ny` = the number of output channels

`ny` = the number of input channels

`Ne` = the number of experiments

To access just one of these sizes use `size(data,k)` for the *k*-th dimension or `size(data,'N'), size(data,'ny')`, etc.

When called with one output argument `d = size(data)` returns:

`d = [N ny nu]` if the number of experiments is 1

`d = [sum(N) ny nu Ne]` if the number of experiments is `Ne>1`

For `idmodel` objects the sizes returned are, in this order:

`ny` = the number of output channels

`nu` = the number of input channels

`Npar` = the length of the `ParameterVector` (= the number of estimated parameters)

`Nx` = the number of states for `idss` and `idgrey` models.

Also in this case the individual dimensions are obtained by `size(mod,2)`, `size(mod,'Npar')`, etc.

When `size` is called with one output argument `d = size(mod)`, `d` is given by

[ny nu Npar]

For idfrd models, the sizes are:

ny = number of output channels

nu = number of input channels

Nf = number of frequencies

Ns = number of spectrum channels

Also in this case the individual dimensions are obtained by size(mod,2), size(mod,'Nf'), etc.

When size is called with one output argument d = size(fre), d is given by

```
[ny nu Nf]
```

When size is called with no output arguments in any of these cases, the information is displayed in the MATLAB command window.

Purpose

Estimate frequency response and spectrum by spectral analysis.

Syntax

```
g = spa(data)
g = spa(data,M,w,maxsize)
[g,phi,spe] = spa(data)
```

Description

spa estimates the transfer function g and the noise spectrum Φ_v of the general linear model

$$y(t) = G(q)u(t) + v(t)$$

where $\Phi_v(\omega)$ is the spectrum of $v(t)$.

data contains the output-input data as an iddata object. The data may be complex-valued.

g is returned as an idfrd object (see idfrd) with the estimate of $G(e^{i\omega})$ at the frequencies ω specified by row vector w. The default value of w is

```
w = [1:128]/128*pi/Ts
```

Here Ts is the sampling interval of data.

g also includes information about the spectrum estimate of $\Phi_v(\omega)$ at the same frequencies. Both outputs are returned with estimated covariances, included in g. See idfrd.

M is the length of the lag window used in the calculations. The default value is

```
M = min(30,length(data)/10)
```

Changing the value of M exchanges bias for variance in the spectral estimate. As M is increased, the estimated functions show more detail, but are more corrupted by noise. The sharper peaks a true frequency function has, the higher M it needs. See etfe as an alternative for narrowband signals and systems. See also “Estimating Spectra and Frequency Functions” on page 3-15 in the “Tutorial” chapter.

maxsize controls the memory-speed trade-off (see Algorithm Properties).

For time series, where data contains no input channels, g is returned with the estimated output spectrum and its estimated standard deviation.

When spa is called with two or three output arguments:

- g is returned as an idfrd model with just the estimated frequency response from input to output and its uncertainty.
- phi is returned as an idfrd model, containing just the spectrum data for the output spectrum $\Phi_y(\omega)$ and its uncertainty.
- spe is returned as an idfrd model containing spectrum data for all output-input channels in data. That is if z = [data.OutputData, data.InputData], spe contains as spectrum data the matrix-valued power spectrum of z.

$$S = \sum_{m=-M}^{M} Ez(t+m)z(t)'\exp(-iWmT)win(m)$$

Here *win(m)* is weight at lag m of an M-size Hamming window and *W* is the frequency value i rad/s. Note that ' denotes complex-conjugate transpose.

The normalization of the spectrum differs from the one used by spectrum in the Signal Processing Toolbox. See “Spectrum Normalization and the Sampling Interval” on page 3-95 in the “Tutorial” chapter for a more precise definition.

Examples

With default frequencies

```
g = spa(z);
bode(g)
```

With logarithmically spaced frequencies

```
w = logspace(-2,pi,128);
g= spa(z,[],w); % (empty matrix gives default)
bode(g,3)
bode(g('noise'),3) % The noise spectrum with confidence interval
of 3 standard deviations.
```

Algorithm

The covariance function estimates are computed using `covf`. These are multiplied by a Hamming window of lag size `M` and then Fourier transformed. The relevant ratios and differences are then formed. For the default frequencies, this is done using FFT, which is more efficient than for user-defined frequencies. For multi-variable systems, a straightforward for-loop is used.

Note that `M` = γ is in Table 6.1 of Ljung (1999). The standard deviations are computed as on pages 184 and 188 in Ljung (1999).

See Also

`bode`, `etfe`, `ffplot`, `idfrd`, `nyquist`

ss, tf, zpk, frd

Purpose

Transform the model objects of the System Identification Toolbox to the LTI models of the Control System Toolbox.

Syntax

```
sys = ss(mod)
sys = tf(mod)
sys = zpk(mod)
sys = frd(mod)
```

Description

`mod` is any `idmodel` object: `idgrey`, `idarx`, `idpoly`, `idss`, `idmodel`, or `idfrd`. However, `idfrd` objects can only be converted to `frd` objects.

`sys` is returned as the indicated LTI model. The noise input channels in `mod` are treated as follows:

Consider a model `mod` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `mod.NoiseVariance` = Λ. The model can also be described with unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

For `ss`, `tf`, and `zpk`, both measured input channels u and normalized noise input channels v in `mod` will be input channels in `sys`. The noise input channels will belong to the `InputGroup 'Noise'`, while the others belong to the `InputGroup 'Measured'`. The names of the noise input channels will be `v@yname`, where `yname` is the name of the corresponding output channel. This means that the LTI object realizes the transfer function $[G\ HL]$.

For `frd`, no noise input information is included.

To transform only the measured input channels in `sys`, use

```
sys = ss(mod('m'))
```

and analogously for `tf` and `zpk`. This will give a representation of just G.

For a time series, (no measured input channels, *nu* = 0), the LTI representations in `ss`, `tf`, `zpk`, and `frd` contain the transfer functions from the normalized noise sources *v* to the outputs., that is *HL*. If the model `m` has both measured and noise inputs, `sys = ss(mod('n'))` will give a representation of the additive noise. For `frd` no output is given for a time-series model.

In addition, the normal subreferencing can be used.

```
sys = ss(mod(1,[3 4]))
```

If you want to describe [*G H*] or *H* (unnormalized noise), from *e* to *y*, use first

```
mod = noisecnv(mod)
```

to convert the noise channels *e* to regular input channels. These channels will have the names `e@yname`.

ssdata

Purpose

Transform a model to state-space form.

Syntax

```
[A,B,C,D,K,X0] = ssdata(m)
[A,B,C,D,K,X0,dA,dB,dC,dD,dK,dX0] = ssdata(m)
```

Description

`m` is the model given as any `idmodel` object. `A`, `B`, `C`, `D`, `K`, and `X0` are the matrices in the state-space description

$$\tilde{x}(t) = Ax(t) + Bu(t) + Ke(t)$$

$$x(0) = x0$$

$$y(t) = Cx(t) + Dx(t) + e(t)$$

where $\tilde{x}(t)$ is $\dot{x}(t)$ or $x(t + Ts)$ depending on whether `m` is a continuous or discrete-time model.

`dA`, `dB`, `dC`, `dD`, `dK`, and `dX0` are the standard deviations of the state-space matrices.

If the underlying model itself is a state-space model, the matrices correspond to the same basis. If the underlying model is an input-output model, an observer canonical form representation is obtained.

For a time-series model (no measured input channels, u= [ ]), B and D are returned as the empty matrices.

Subreferencing models in usual way (see `idmodel` properties) will give the state-space representation of the chosen channels. Notice in particular that

```
[A,B,C,D] = ssdata(m('m'))
```

gives the response from the measured inputs. This is a model without a disturbance description. Moreover

```
[A,B,C,D,K] = ssdata(m('n'))
```

(`'n'` as in 'noise') gives the disturbance description, i.e, a time-series description of the additive noise with no measured inputs, so that `B=[]` and `D=[]`.

To obtain state-space descriptions that treat all input channels, both u and e as measured inputs, first apply the conversion

```
m = noisecnv(m)
```

or

```
m = noisecnv(m,'norm')
```

where the latter case first normalizes *e* to *v*, where *v* has a unit covariance matrix. See the reference page for `noisecnv`.

Algorithm

The computation of the standard deviations in the input-output case assumes that an *A* polynomial is not used together with a *F* or *D* polynomial in (Equation 3-43). For the computation of standard deviations in the case that the state-space parameters are complicated functions of the parameters, Gauss approximation formula is used together with numerical derivatives. The step-sizes for this differentiation are determined by `nuderst`.

See Also

`idmodel`, `idss`, `nuderst`

step

Purpose

Estimate/compute/display step response.

Syntax

```
step(m)
step(data)
step(data,'sd',sd,'PW',na,Time)
step(m,'sd',sd,Time)
step(m1,m2,...,dat1, ...,mN,Time,'sd',sd)
step(m1,'PlotStyle1',m2,'PlotStyle2',...,dat1,'PlotStylek',...,mN,
   'PlotStyleN',Time,'sd',sd)
[y,t,ysd] = step(m)
mod = step(data)
```

Description

`step` can be applied both to `idmodels` and to `iddata` sets, as well as to any mixture.

For a discrete-time `idmodel` `m`, the step response `y` and, when required, its estimated standard deviation `ysd`, is computed using `sim`. When called with output arguments, `y`, `ysd` and the time vector `t` are returned. When `step` is called without output arguments, a plot of the step response is shown. If `sd` is given a value larger than zero, a confidence region around the response is drawn. It corresponds to the confidence of `sd` standard deviations. If the input argument list contains `'fill'`, this region is plotted as a filled area.

The start time `T1` and the end time `T2` can be specified by `Time= [T1 T2]`. If `T1` is not given, it is set to `-T2/4`. The negative time lags (the step is always assumed to occur at time 0) show possible feedback effects in the data, when the step is estimated directly from data. If `Time` is not specified, a default value is used.

For an `iddata` set data, `step(data)` estimates a high order, noncausal FIR model after first having prefiltered the data so that the input is “as white as possible.” The step response of this FIR model and, when asked for, its confidence region is then plotted. When called with an output argument, `step`, in the `iddata` case, returns this FIR model, stored as an `idarx` model.The order of the prewhitening filter can be specified as `na`. The default value is `na = 10`.

Any number and any mixture of models and data sets can be used as input arguments. The responses are plotted with each input/output channel (as defined by the models and data sets `InputName` and `OutputName`) as a separate plot. Colors, linestyles, and marks can be defined by `PlotStyle` values, as in

```
step(m1,'b-*',m2,'y--',m3,'g')
```

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$
$$\text{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `m.NoiseVariance` = Λ. The model can also be described with unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$\text{cov}(v) = I$$

- `step(m)` plots the step response of the transfer function G.
- `step(m('n'))` plots the step response of the transfer function H. (ny inputs and ny outputs).The input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- If `m` is a time series, that is $nu = 0$, `step(m)` plots the step response of the transfer function H.
- `step(noisecnv(m))` plots the step response of the transfer function $[G\ H]$ ($nu+ny$ inputs and ny outputs). The noise input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- `step(noisecnv(m,'norm'))` plots the step response of the transfer function $[G\ HL]$ ($nu+ny$ inputs and ny outputs). The noise input channels have names `v@yname`, where `yname` is the name of the corresponding output.

Arguments

If `step` is called with a single `idmodel` `m`, the output argument `y` is a 3-D array of dimension `Nt`-by-`ny`-by-`nu`. Here `Nt` is the length of the time vector `t`, `ny` is the number of output channels and `nu` is the number of input channels. Thus `y(:,ky,ku)` is the response in the `ky`-th output channel to a step in the `ku`-th input channel. No plot is produced when output arguments are used.

`ysd` has the same dimensions as y and contains the standard deviations of `y`.

If `step` is called with an output argument and a single data set in the input arguments, the output is returned as an `idarx` model `mod` containing the high order FIR model, and its uncertainty. By calling `step` with `mod`, the responses can be displayed and returned without having to redo the estimation.

Example `step(data,'sd',3)` estimates and plots the step response

```
mod = step(data)
step(mod,'sd',3)
```

See Also `cra`, `impulse`

Purpose

Generate model structure matrices.

Syntax

```
NN = struc(NA,NB,NK)
```

Description

`struc` returns in `NN` the set of model structures comprised of all combinations of the orders and delays given in row vectors `NA`, `NB`, and `NK`. The format of `NN` is consistent with the input format used by `arxstruc` and `ivstruc`. The command is intended for single-input systems only.

Examples

The statement

```
NN = struc(1:2,1:2,4:5);
```

produces

```
NN =
 1  1  4
 1  1  5
 1  2  4
 1  2  5
 2  1  4
 2  1  5
 2  2  5
```

timestamp

Purpose

Bookkeeping of created objects.

Syntax

```
timestamp(obj)
ts = timestamp(obj)
```

Description

`obj` is any `idmodel`, `iddata` or `idfrd` object. `timestamp` returns or displays a string with information about when the object was created and last modified.

Purpose

Transform a model to transfer function form.

Syntax

```
[num,den] = tfdata(m)
[num,den,sdnum,sdden] = tfdata(m)
[num,den,sdnum,sdden] = tfdata(m,'v')
```

Description

`m` is a model given as any `idmodel` object with `ny` output channels and `nu` input channels

`num` is a a cell array of dimension `ny`-by-`nu`. `num{ky,ku}` (note the curly brackets) contains the numerator of the transfer function from input `ku` to output `ky`. This numerator is a row vector whose interpretation is described below.

Similarily `den` is an `ny`-by-`nu` cell array of the denominators.

`sdnum` and `sdden` have the same formats as `num` and `den`. They contain the standard deviations of the numerator and denominator coefficients.

If `m` is a SISO model, adding an extra input argument `'v'` (for vector) will return `num` and `den` as vectors rather than cell arrays.

The formats of `num` and `den` are the same as those used by the Signal Processing Toolbox and the Control System Toolbox, both for continuous-time and discrete-time models. See “Examining Models” in the “Tutorial” chapter and the examples below.

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `m.NoiseVariance` = Λ. The model can also be described with unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

- `tfdata(m)` returns the transfer function *G*.
- `tfdata(m('n'))` returns the transfer function *H*. (*ny* inputs and *ny* outputs)
- If `m` is a time series, that is *nu* = 0, `tfdata(m)` returns the transfer function *H*.
- `tfdata(noisecnv(m))` returns the transfer function [*G H*] (*nu*+*ny* inputs and *ny* outputs)
- `tfdata(noisecnv(m,'norm'))` returns the transfer function *[G HL]* (*nu*+*ny* inputs and *ny* outputs.

Examples

For a continuous-time model

```
num = [1 2]
den = [1 3 0]
```

corresponds to the transfer function

$$G(s) = \frac{s+2}{s^2+3s}$$

For a discrete-time model

```
num = [2 4 0]
den = [1 2 3 5]
```

corresponds to the transfer function

$$H(z) = \frac{2z^2+4z}{z^3+2z^2+3z+5}$$

which is the same as

$$H(q) = \frac{2q^{-1}+4q^{-2}}{1+2q^{-1}+3q^{-2}+5q^{-3}}$$

Note that for discrete time, `idpoly` and `polydata` has a different interpretation of the numerator vector, in case it does not have the same length as the denominator vector. To avoid confusion, it is advised to fill out with zeros to make numerator and denominator vectors the same length. That is done by `tfdata`.

See Also

`idpoly`, `noisecnv`

Purpose

Plot a variety of model characteristics (Requires the Control System Toolbox).

Syntax

```
view(m)
view(m('n'))
view(m1,...,mN,Plottype)
view(m1,PlotStyle1,...,mN,PlotStyleN)
```

Description

`m` is the output-input data to be graphed, given as any `idfrd` or `idmodel` object. After appropriate model transformations, the LTI Viewer of the Control System Toolbox is invoked. This allows e.g., `bode`, `nyquist`, `impulse`, `step`, `zero/poles` plots.

To compare several models `m1,...,mN`, use `view(m1,...,mN)`. With `PlotStyle`, the color, linestyle and marker, of each model can be specified.

```
view(m1,'y:*',m2,'b')
```

Adding `Plottype` as a last argument specifies the type of plot in which view is initialized. `Plottype` is any of `'impulse'`, `'step'`, `'bode'`, `'nyquist'`, `'nichols'`, `'sigma'`, or `'pzmap'`. It can also be given as a cell array containing any collection of these strings (up to 6) in which case a multiplot is shown.

`view` will not display confidence regions. For that use `bode`, `nyquist`, `impulse`, `step`, and `pzmap`.

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix

$$y = Gu + He$$

$$cov(v) = \Lambda = LL^T$$

where L is a lower triangular matrix. Note that `m.NoiseVariance` = Λ. The model can also be described with unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$cov(v) = I$$

- `view(m)` plots the characteristics of the transfer function G.

- `view(m('n'))` plots the characteristics of the transfer function *HL*. (*ny* inputs and *ny* outputs).The input channels have names `v@yname`, where `yname` is the name of the corresponding output.
- If `m` is a time series, that is *nu* = 0, `view(m)` plots the characteristics of the transfer function *HL*.
- `view(noisecnv(m))` plots the characteristics of the transfer function [*G H*] (*nu*+*ny* inputs and *ny* outputs). The noise input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- `view(noisecnv(m,'norm'))` plots the characteristics of the transfer function *[G HL]* (*nu*+*ny* inputs and *ny* outputs. The noise input channels have names `v@yname`, where `yname` is the name of the corresponding output.

`view` does not give access to all of the features of `ltiview`. Use

```
ml = ss(m), ltiview(Plottype,ml,...)
```

to reach these options.

See Also

`bode`, `impulse`, `nyquist`, `step`, `pzmap`

Purpose

Compute zeros, poles, and transfer function gains of models.

Syntax

```
[z,p,k] = zpkdata(m)
[z,p,k,dz,dp,dk] = zpkdata(m)
[z,p,k,dz,dp,dk] = zpkdata(m,'v')
```

Description

`m` is a model given as any `idmodel` object with `ny` output channels and `nu` input channels.

`z` is a a cell array of dimension `ny`-by-`nu`. `z{ky,ku}` (note the curly brackets) contains the zeros of the transfer function from input `ku` to output `ky`. This is a column vector of possibly complex numbers.

Similarily `p` is an `ny`-by-`nu` cell array containing the poles.

`k` is a `ny`-by-`nu` matrix whose `ky`-`ku` entry is the transfer function gain of the transfer function from input `ku` to output `ky`. Note that the transfer function gain is value of the leading coefficient of the numerator, when the leading coefficient of the denominator is normalized to 1. It thus differs from the static gain. The static gain can be retrieved as `Ks = freqresp(m,0)`.

`dz` contains the covariance matrices of the zeros in the following way: `dz` is a `ny`-by-`nu` cell array. `dz{ky,ku}` contains the covariance information about the zeros of the transfer function from `ku` to `ky`. It is a 3-D array of dimension 2-by-2-by-`Nz`, where `Nz` is the number of zeros. `dz{ky,ku}(:,:,kz)` is the covariance matrix of the zero `z{ky,ku}(kz)`, so that the 1-1 element is the variance of the real part, the 2-2 element is the variance of the imaginary part and the 1-2 and 2-1 elements contain the covariance between the real and imaginary parts.

`dp` contains the covariance matrices of the poles in the same way.

`dk` is a matrix containing the variances of the elements of `k`.

If `m` is a SISO model, adding an extra input argument `'v'` (for vector) will return `z` and `p` as vectors rather than cell arrays.

Note that the zeros and the poles are associated with the different channels combinations. To obtain the so-called transmission zeros, use `tzero`.

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels *u* (*nu* channels) and noise channels *e* (*ny* channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where *L* is a lower triangular matrix. Note that `m.NoiseVariance` = Λ. The model can also be described with unit variance, normalized noise source *v*:

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

Then

- `zpkdata(m)` returns the zeros and poles of *G*.
- `zpkdata(m('n'))` returns the zeros and poles of *H*. (*ny* inputs and *ny* outputs)
- If `m` is a time series, that is *nu* = 0, `zpkdata(m)` returns the zeros and poles of *H*.
- `zpkdata(noisecnv(m))` returns the zeros and poles of the transfer function [*G H*] (*nu*+*ny* inputs and *ny* outputs)
- `zpkdata(noisecnv(m,'norm'))` returns the zeros and poles of the transfer function *[G HL]* (*nu*+*ny* inputs and *ny* outputs.

The procedure handles both models in continuous and discrete time.

Note that you cannot rely on information about zeros and poles at the origin and at infinity for discrete time models. (This is a somewhat confusing issue anyway.)

Algorithm

The poles and zeros are computed using `ss2zp`. The covariance information is computed using Gauss's approximation formula, using the parameter covariance matrix contained in `m`. When the transfer function depends on the parameters in a complicated way, numerical differentiation is applied. The step-sizes for the differentiation are determined in the M-file `nuderst`.

Index

F

G

H

I

K

L

M

N

O

U

V

W

Y

Z