SOFTWARE MANUAL FOR THE LTI SYSTEM IDENTIFICATION TOOLBOX

Filtering and System Identification: An Introduction to using Matlab Software

Michel Verhaegen, Vincent Verdult, and Niek Bergboer

August 1, 2007

Delft University of Technology
Delft Center for Systems and Control
Mekelweg 2, 2628 CD, Delft, The Netherlands
M.Verhaegen@moesp.org, I.Houtzager@TUDelft.nl

Contents

Chapter 1

Introduction

This book is a companion to the textbook "Filtering and System Identification, An Introduction" by Michel Verhaegen and Vincent Verdult. It describes and illustrates the use of Matlab programs for a number of algorithms presented in the textbook. The Matlab programs, bundled in a toolbox, can be found as download on the publishers website. The goal of this companion book is twofold. First, it helps to clarify some of the theoretical matter of the textbook using examples. Second, it describes how to perform many of the filtering and system identification operations using the toolbox software.

Chapters 2, 3 and 4 are companions to Chapters 8, 9 and 10 in the textbook. Chapter 2 describes parametric model identification and Chapter 3 describes subspace identification methods. These two chapters treat numerical methods both in the time domain and the frequency domain. Although, the textbook mostly deals with the time domain, we present also the frequency domain, because this extension is almost straightforward.

Chapter 4 provides a framework into which both topics fit. Chapter 5 of this companion book contains a comprehensive overview of the toolbox software, including a detailed reference page for every toolbox function.

1.1 How to use this book

It is assumed that the user of this book also has the textbook at his disposal, as references to equations, examples, sections and page-numbers in the textbook are often made. The theory in the textbook is assumed to be prior knowledge, although the examples in this companion book are designed to help one to better understand the theory. It is therefore recommended to read a section in the textbook, and to perform the examples in the corresponding section in this companion book after wards.

In order to repeat and understand the examples in this companion book, it is highly recommended to have MATLAB 6 or higher at one's disposal, since MATLAB commands are often used throughout the text. A MATLAB toolbox is provided on a CD-ROM or as download located on the publishers website. Section 1.2 contains more information on installing and using this toolbox.

MATLAB function name

Whenever a standard MATLAB command or a command from the toolbox provided with this book is used for the first time, its name is printed in a bold typeface in the margin.

A number of the MATLAB examples in this companion book use input and output data sequences from a linear system. In many of these cases, a data-file is provided so that the user can repeat the experiment himself to gain more understanding and confidence in using the toolbox software. If a data-file is available, a CD-ROM icon is shown in the margin. In the text near this CD-ROM icon, the filename of the datafile is mentioned.

1.2 Toolbox Software Features

The toolbox software provided with this book provides subspace identification and parametric model estimation functions for linear time-invariant (LTI) state-space models, based on both time-domain measurements and frequency response function (FRF) measurements.

The subspace identification framework uses comparatively simple linear algebra steps and operates in a noniterative way. This has the advantage of not requiring an initial model estimate. In addition, the time-domain subspace identification framework allows multiple individually recorded input-output batches —possible under different operating conditions— to be *concatenated*. The frequency-domain subspace identification framework also allows concatenation when estimating discrete-time models.

The results from the subspace identification framework can be used as initial estimates for the parametric model estimation framework. Models can be optimized based on both time-domain and frequency-domain measurements. The parametric estimation framework provides a simple one-command way of optimizing models. Both output-error and innovation model estimation is supported in a transparent way. In addition, the efficient handling of very large datasets and the maximum likelihood identification of state-space models is supported. These last two features provide interesting possibilities, but explaining them falls outside the scope of this book. However, the function reference pages in the final chapter of this book do show the use of these features.

In addition to these main two functionalities —subspace identification and parametric model estimation— the toolbox contains a host of small utilities that can be used to facilitate tasks ranging from model parameterization to model validation.

1.3 Toolbox Software and Hardware Requirements

The toolbox software is targeted at MATLAB version 6 (Release 12) or higher without requiring any additional MATLAB toolboxes to function. However, the toolbox also works on MATLAB version 5 (Release 11). It should be noted that although the toolbox software itself does not depend on other toolboxes, some

of the examples in this book do require additional toolboxes. These toolboxes are the MATLAB Control Systems Toolbox [1] and the MATLAB Identification Toolbox [2]. If an example depends on one of these toolboxes, this fact will be clearly noted in the text.

A number of numerical calculation functions have been implemented as MATLAB executables (MEX-files) for efficiency reasons. These MEX-files are available for Microsoft Windows and Linux on Intel-compatible CPUs. For platforms for which no MEX-files are available, corresponding MATLAB-scripts (M-files) provide the same functionality —although much slower— in a manner transparent to the user. Also the sourcecodes can be compiled for your specific architecture, if the BLAS, LAPACK and SLICOT libraries are avalaible.

The hardware requirements for using the toolbox software are the same as those for using MATLAB itself. On platforms for which no MEX-files are available, only the platform and CPU independent M-files of the toolbox are executed by MATLAB itself. The Linux MEX-files for MATLAB 5 require, like MATLAB itself, at least an Intel 80486 compatible CPU, while those for MATLAB 6 require at least an Intel Pentium compatible CPU. The Microsoft Windows MEX-files run on both MATLAB 5 and MATLAB 6 for Windows, and have therefore been targeted at the hardware requirements for MATLAB 5. This means that at least an Intel 80486 compatible CPU is required.

Section 5.2 contains details on how to install the toolbox software on a local system.

1.4 Acknowledgments

The toolbox software is based mainly of the Subspace Model Identification (SMI) toolbox version 1.0 [3], which has been developed by Michel Verhaegen, Tung Chou, Bert Haverkamp and Vincent Verdult of the Delft University of Technology, as well as by David Westwick of the Boston University. However, the conceptual framework for parametric model estimation has inherited some features of the SMI toolbox version 2.0 [4], which Bert Haverkamp developed during his PhD research and which is based on SMI version 1.0. Niek Bergboer extended the software to its current state during a temporary project at the SCE group at the University of Twente in Enschede.

1.5 Disclaimer

ACCURACY OR COMPLETENESS OF ANY STATEMENTS, INFORMATION OR MATERIALS CONCERNING THE SOFTWARE. IN NO EVENT WILL THE AUTHORS BE LIABLE FOR ANY INDIRECT, PUNITIVE, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES HOWEVER THEY MAY ARISE AND EVEN IF THE AUTHORS HAVE BEEN PREVIOUSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

1.6 License Agreement

- You are allowed to use the toolbox software for noncommercial (academic) research and educational purposes free of charge.
- You are not allowed to commercialize the software.
- If the use of the toolbox software is an essential part of a published work, you must give proper reference to the toolbox and its authors.
- If you wish to use the toolbox software for commercial projects, an additional agreement with the authors must be made.

1.7 Lines of Communication

The authors wish to establish an open line of communication with the users of both this companion book and the toolbox software. We strongly encourage all users to email the authors with comments and suggestions for this and future editions of both the companion book and the toolbox software. In this way we can keep improving our work and keep you informed of any interesting news concerning the book or software.

Delft Center for Systems and Control
Michel Verhaegen, m.verhaegen@moesp.org
Ivo Houtzager, i.houtzager@tudelft.nl

References

[1] The MathWorks Inc., Natick, Massachusetts, *Using the Control Systems Toolbox*, version 1 (release 12) ed., Nov. 2000.

[2] L. Ljung, *System Identification Toolbox User's Guide*. The MathWorks Inc., Natick, Massachusetts, version 5 (release 12) ed., Nov. 2000.

[3] B. Haverkamp, C. T. Chou, and M. Verhaegen, "SMI toolbox: A Matlab toolbox for state space model identification," *Journal A*, vol. 38, no. 3, pp. 34–37, 1997.

[4] B. Haverkamp, *Subspace Method Identification, Theory and Practice*. PhD thesis, Delft University of Technology, Delft, The Netherlands, 2000.

Chapter 2

Parametric Model Estimation

After studying this chapter you can:

- obtain the various LTI state-space model parameterizations, covered in Chapters 7 and 8 of the textbook, using the toolbox functions `dss2th`, `dth2ss`, `css2th` and `cth2ss`.
- define a cost-criterion, or cost-function for the parametric estimation of models based on time-domain and frequency-domain data.
- perform a numerical parameter search using the Moré-Hebden Levenberg-Marquardt algorithm in the `lmmore` toolbox function.
- perform parametric identification of an LTI state-space model based on time-domain data using the `doptlti` toolbox function.
- perform parametric identification of an LTI state-space model based on frequency response function (FRF) data using the `foptlti` toolbox function.

2.1 Introduction

This chapter is a companion to Chapters 7 and 8 of the textbook. In this chapter, we illustrate some of the theoretical concepts using simple examples that the reader can repeat himself using MATLAB and the toolbox software. In Section 2.2, the parameterization of MIMO state-space models will be covered, as well as the conversion of a parameter vector into a corresponding state-space model. In Section 2.3, different parameterizations will be used to identify state-space models based on time-domain measurements. In Section 2.4 a different approach is taken: rather than using time-domain measurements, frequency response function (FRF) measurements are taken as a starting point for state-space model identification.

2.2 Parameterizing MIMO State-Space Models

In this section the various model parameterizations derived in Section 7.3 of the textbook will be clarified by means of a number of examples. First, a canonical form for systems having one output are covered. Then, the output normal form, tridiagonal and full parameterizations are covered. The emphasis lies on illustrating the theory and showing how the toolbox can be used to obtain the parameterization in a quick and convenient way. Finally, the use of the toolbox to convert parameter vectors back into their corresponding state-space models is covered for multiple input, multiple output (MIMO) systems. This means that a parameter vector θ for a given parameterization is converted back into a state-space model (A, B, C, D).

2.2.1 Canonical Forms for Systems with One Output

In Section 7.3 of the textbook we covered the observer canonical form for systems having one output. For systems with one output, the C-matrix is a row vector. The essential step then is the realization that there always exists a similarity transformation of a state-space model such that the C-vector is transformed into the first unit vector. Furthermore, the A-matrix is transformed such that it contains only n free parameters.

In this section we will use a fourth-order system from [1] with the D matrix equal to zero. The system is defined as follows:

$$\begin{aligned} A &= \begin{bmatrix} -0.6129 & 0 & 0.0645 & 0 \\ 0 & 0.7978 & 0 & -0.4494 \\ -6.4516 & 0 & -0.7419 & 0 \\ 0 & 0.4494 & 0 & 0.8876 \end{bmatrix}, \qquad B = \begin{bmatrix} 0.0323 \\ 0.8989 \\ 0.1290 \\ 0.2247 \end{bmatrix}, \\ C &= \begin{bmatrix} 9.6774 & 0.1124 & 1.6129 & 0.4719 \end{bmatrix}, \qquad D = 0. \end{aligned} \tag{2.1}$$

The system is converted into observer canonical form. According to Subsection 3.4.4 of the textbook, the observer canonical form of a fourth-order system is given by

$$x(k+1) = \begin{bmatrix} 0 & 0 & 0 & -a_0 \\ 1 & 0 & 0 & -a_1 \\ 0 & 1 & 0 & -a_2 \\ 0 & 0 & 1 & -a_3 \end{bmatrix} x(k) + \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} u(k),$$
$$y(k) = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} x(k).$$

Such that,

$$y(k) = \frac{c_{n-1}q^{n-1} + \cdots + c_1 q + c_0}{q^n + a_{n-1}q^{n-1} + \cdots + a_1 q + a_0} u(k).$$

In order to build this state-space model, we first need the transfer function polynomials corresponding to (2.1). The MATLAB Control Systems Toolbox [2] functions `ss`, `tf` and `tfdata` are used to this end as follows:

ss
tf
tfdata

```
[num,den]=tfdata(tf(ss(A,B,C,D,-1)));
```

The variables `num` and `den` are cell arrays, such that `num{1}` contains $[c_n, c_{n-1}, \ldots, c_0]$ and `den{1}` contains $[1, a_{n-1}, a_n, \ldots, a_0]$. These polynomials can be used to build the state-space model corresponding to (2.1) in observer canonical form as follows:

```
>> n=4;
>> At=[[zeros(1,n-1);eye(n-1)],flipud(-den{1}(2:n+1)')]
At =

         0         0         0   -0.7925
    1.0000         0         0    0.2347
         0    1.0000         0    0.5025
         0         0    1.0000    0.3306
>> Bt=flipud(num{1}(2:n+1)')
Bt =

    0.0989
    0.4859
   -0.4881
    0.7277
>> Ct=[zeros(1,n-1),1]
Ct =
     0     0     0     1
>> Dt=0;
Dt =
     0
```

We can subsequently verify that the model (A_T, B_T, C_T, D_T) indeed has the same input-output behavior as the original model. This is equivalent to saying that the transfer functions of the original and transformed systems should be equal. The MATLAB Control Systems Toolbox functions `tf` and `ss` are used to calculate these transfer functions as follows:

```
>> tf(ss(A,B,C,D,-1))
Transfer function:
  0.7277 z^3 - 0.4881 z^2 + 0.4859 z + 0.09886
-----------------------------------------------
z^4 - 0.3306 z^3 - 0.5025 z^2 - 0.2347 z + 0.7925
Sampling time: unspecified

>> tf(ss(At,Bt,Ct,Dt,-1))
Transfer function:
  0.7277 z^3 - 0.4881 z^2 + 0.4859 z + 0.09886
-----------------------------------------------
z^4 - 0.3306 z^3 - 0.5025 z^2 - 0.2347 z + 0.7925
Sampling time: unspecified
```

The transfer functions are indeed equal.

2.2.2 The Output Normal Form

In this section we will illustrate the output normal parameterization using the same example from [3] that is used Section 7.3.1 of the textbook. Consider the second-order state-space model defined by the quadruple of system matrices $[A, B, C, D]$ equal to:

$$\begin{aligned} A &= \begin{bmatrix} 1.5 & -0.7 \\ 1 & 0 \end{bmatrix}, & B &= \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \\ C &= \begin{bmatrix} 1 & 0.5 \end{bmatrix}, & D &= 0. \end{aligned} \tag{2.2}$$

We will use this model at numerous places in this companion book as it serves as a simple yet interesting system. In Example 7.5 on page 194 of the textbook, we show how to derive the two parameters that parameterize the pair (A, C) of an output normal form of the system (2.2). However, rather than having to do this tedious work manually, we can use the toolbox-function `dss2th` (see manual on page 124) to obtain the output normal parameters. This function name is an abbreviation of "Discrete-time State-Space To Theta conversion". The input parameters to `dss2th` are the (A, C)-pair and the string `'on'`. The latter is an abbreviation of "output normal".

dss2th

```
>> th=dss2th(A,C,'on')
th =
    0.8824
   -0.7000
```

This parameter vector contains indeed the two parameters derived in Example 7.5 on page 194 of the textbook. We can also obtain the similarity transformation T that was used to transform the (A, C)-pair into lower triangular form. Note that this similarity transformation is denoted $(T_t T_h)$ in the example in the textbook. The similarity transformation is obtained as the third output argument of `dss2th`:

```
>> [th,params,T]=dss2th(A,C,'on')
```

The output argument `params` is a structure that contains information on the parameterized system that could not be put in the parameter vector itself. This structure contains the system dimensions, which matrices are parameterized and what type of parameterization has been used. We can subsequently check that T is indeed the correct similarity transformation by calculating the transformed $C_T = CT$ and $A_T = T^{-1}AT$ and verifying that $[C_T^T A_T^T]^T$ indeed is lower-triangular.

```
>> [C*T;T\A*T]
ans =
    0.7141         0
    0.6176    0.4706
   -0.3294    0.8824
```

After parameterizing the (A, C)-pair, the next step in the output normal parameterization is to transform B, after which B and D can be appended to the parameter vector. The transformed B is given by:

```
>> T\B
ans =
    1.4003
    4.1133
```

The D matrix does not change under the similarity transformation, and will remain 0. It can thus simply be appended to be parameter vector. However, rather than appending the B and D matrices to the output normal parameter vector manually, the entire process can be automated using one single call to the `dss2th` parameterization function in which all matrices are passed. Again, the string `'on'` indicates that an output normal parameterization should be used:

```
>> th=dss2th(A,B,C,D,'on')
th =
    0.8824
   -0.7000
    1.4003
    4.1133
         0
```

The number of parameters is $n\ell + nm + \ell m = 5$ which is the minimal number of parameters necessary to parameterize the state-space model.

2.2.3 The Tridiagonal Form

The toolbox function `dss2th` can be used for the tridiagonal parameterization as well, which will be shown in this section. However, the second-order system introduced in the previous section will not be used here, since the A matrix of a second-order system is 2×2 and thus by definition already tridiagonal.

A more educational example is the parameterization of a fourth-order system, since in this case the effect of the tridiagonalization can be inspected visually. Like in the SISO observer canonical form example, the model (2.1) is used.

The entire model can be parameterized in a single call to the `dss2th` parameterization function. We will also request the similarity transformation T that was used to transform A into tridiagonal form. The last input argument is now `'tr'`, which indicates that a tridiagonal parameterization is requested.

```
>> [th,params,T]=dss2th(A,B,C,D,'tr');
```

It is straightforward to check that this transformation indeed converts A into tridiagonal form by looking at the transformed matrix $A_T = T^{-1}AT$:

```
>> T\A*T
ans=
   -0.6774    0.6418         0         0
   -0.6418   -0.6774         0         0
         0         0    0.8427    0.4472
         0         0   -0.4472    0.8427
```

The number of parameters in the vector θ equals $3n - 2 + n\ell + \ell n + nm = 19$, which is 10 more than the minimum number $n\ell + \ell n + nm = 9$. This represents an excess of $3n - 2 = 10$ parameters, making the parameter mapping surjective. During a numerical search for the optimal parameter value, this means that regularization needs to be applied.

2.2.4 The Full Parameterization

In the full parameterization, the model is not transformed at all. Rather, the system matrices are simply vectorized as follows:

$$\theta = \operatorname{vec}\left(\begin{bmatrix} A & B \\ C & D \end{bmatrix}\right). \tag{2.3}$$

The simplicity of this parameterization is easily seen when parameterizing the second-order model (2.2) that was also used in the output normal form in Section 2.2.2. The call to `dss2th` is the same as in the output normal and tridiagonal parameterization cases, except for the last input argument, which now is `'fl'` to indicate that a full parameterization is requested:

```
>> th=dss2th(A,B,C,D,'fl')
th =
    1.5000
    1.0000
    1.0000
   -0.7000
         0
    0.5000
    1.0000
         0
         0
```

It is clear that the elements of A, B, C and D are simply stacked in order to obtain the parameter vector θ:

$$\theta = \mathrm{vec}\left(\begin{bmatrix} A & B \\ C & D \end{bmatrix}\right) = \mathrm{vec}\left(\left[\begin{array}{cc|c} 1.5 & -0.7 & 1 \\ 1 & 0 & 0 \\ \hline 1 & 0.5 & 0 \end{array}\right]\right).$$

Like the tridiagonal parameterization, this full parameterization is surjective: it is overparameterized. In the tridiagonal case, this over-parameterization is taken care of using regularization in the numerical parameter search. However, in the full parameterization described in this section, we have a more elegant method at our disposal. The gradient projection described in the textbook, which is based on [4, 5], is used to confine the possible parameter updates to the directions that do not correspond to similarity transformations on the system. The calculation of this projection will be illustrated in Example 2.5 on page 23.

2.2.5 Converting a Parameter Vector into a Corresponding State-Space Model

In a numerical search algorithm we need to be able to calculate for the current parameter estimate θ the value of the cost function

$$J_N(\theta) = \frac{1}{N}\sum_{k=0}^{N-1} \|y(k) - \widehat{y}(k,\theta)\|_2^2,$$

with $\widehat{y}(k,\theta)$ the output of the model. As shown in Section 8.2.3 of the textbook, this involves simulating a system that corresponds to the current value of θ. How this simulation takes place will be covered in the next section. In this section, we will discuss how to obtain the state-space model that corresponds to the current value of θ.

Whereas the `dss2th` function converts a state-space model into parameter vector, the "Discrete-time Theta To State-Space conversion" function `dth2ss` (see manual on page 126) does the opposite: it converts a parameter vector into a state-space model. We will now show how to use this function, taking the output normal parameterization of the system (2.2) on page 12 as an example. The function works completely analogous for the other parameterizations. We will first derive the output normal parameter vector including the `params` structure and similarity transformation:

dth2ss

```
>> [th,params,T]=dss2th(A,B,C,D,'on');
```

Subsequently, we will convert the parameter vector back into a state-space model. Note that `dth2ss` needs the same extra information that `dss2th` put into the `params` structure in order to perform the correct conversion:

```
>> [A2,B2,C2,D2]=dth2ss(th,params)
A2 =
    0.6176    0.4706
   -0.3294    0.8824
B2 =
    1.4003
```

```
    4.1133
C2 =
    0.7141         0
D2 =
     0
```

At first sight, it might seem that the conversion went wrong, since the model shown here is not the same as the model (2.2) that was parameterized in the first place. However, the difference is just a similarity transformation. This means that the above model is equivalent to the original model in terms of input-output behavior, but that the similarity transformation that was used to parameterize the model has not been taken into account in the conversion.

If required, the inverse of the similarity transformation that was used to parameterize the model can be applied to the model corresponding to the parameter vector, yielding the original model (2.2):

```
>> [A2,B2,C2,D2]=dth2ss(th,params,T)
A2 =
    1.5000   -0.7000
    1.0000   -0.0000
B2 =
    1.0000
    0.0000
C2 =
    1.0000    0.5000
D2 =
     0
```

2.2.6 Parameterizing continuous-time models

Up till now, all models involved have been discrete-time models. However, in the frequency-domain framework of section 2.4 it is possible to identify continuous-time models as well. However, the output normal parameterization is defined in a different way for continuous-time models [3]. Therefore, continuous-time counterparts of `dss2th` and `dth2ss` are included in the toolbox: these are the functions `css2th` (see manual on page 94) and `cth2ss` (see manual on page 96).

css2th
cth2ss

2.3 Identifying State-Space Models of Known Order in the Time Domain

In this section we will show how to identify state-space models based on an initial model estimate and time-domain measurements. An initial model is required to start up the identification procedure. In Chapter 3 we will see that subspace identification methods can be used to generate an initial model.

The cost function $J_N(\theta)$, which serves as a model mismatch criterion, will be discussed first for both output error models and prediction error models. Then, the numerical parameter estimation itself is discussed.

2.3.1 The Cost Function and its Gradient

The optimal parameter value $\widehat{\theta}$ is found by minimizing a cost function

$$J_N(\theta) = \frac{1}{N}\sum_{k=0}^{N-1} ||y(k) - \widehat{y}(k,\theta)||_2^2.$$

The estimated output $\widehat{y}(k,\theta)$ corresponding to the current value θ of the parameters can be defined in two different ways. The first corresponds to the output-error problem of Chapter 7 of the textbook. The estimated output $\widehat{y}(k,\theta)$ in the output-error problem is obtained from the predictor

$$\widehat{x}(k+1,\theta) = A(\theta)\widehat{x}(k,\theta) + B(\theta)u(k), \tag{2.4}$$
$$\widehat{y}(k,\theta) = C(\theta)\widehat{x}(k,\theta) + D(\theta)u(k). \tag{2.5}$$

The second way of defining the estimated output is by using the one step ahead predictor as in the prediction error problem of Chapter 8 of the textbook. Given a suitably parameterized Kalman gain $K(\theta)$, the predicted output is then obtained as follows:

$$\begin{aligned}\widehat{x}(k+1,\theta) &= (A(\theta) - K(\theta)C(\theta))\,\widehat{x}(k,\theta) \\ &\quad + (B(\theta) - K(\theta)D(\theta))\,u(k) + K(\theta)y(k),\end{aligned} \tag{2.6}$$
$$\widehat{y}(k,\theta) = C(\theta)\widehat{x}(k,\theta) + D(\theta)u(k). \tag{2.7}$$

Regardless of which of these definitions is used, a second-order approximation of the cost function is made in Section 7.5 of the textbook:

$$J_N(\theta) \approx J_N(\theta^{(i)}) + J_N'(\theta^{(i)})\left(\theta - \theta^{(i)}\right) + \frac{1}{2}\left(\theta - \theta^{(i)}\right)^T J_N''(\theta^{(i)})\left(\theta - \theta^{(i)}\right). \tag{2.8}$$

Furthermore, an error-vector $E_N(\theta)$ is defined, which is built up of the mismatches between the model output and the measured output $\epsilon(k,\theta) = y(k) - \widehat{y}(k,\theta)$:

$$E_N(\theta) = \begin{bmatrix} \epsilon(0,\theta) \\ \epsilon(1,\theta) \\ \vdots \\ \epsilon(N-1,\theta) \end{bmatrix}.$$

In addition the Jacobian, or gradient, $\Psi_N(\theta)$ of this error-vector is defined:

$$\Psi_N(\theta) = \frac{\partial E_N(\theta)}{\partial \theta^T} = \begin{bmatrix} \frac{\partial \epsilon(0,\theta)}{\partial \theta^T} \\ \frac{\partial \epsilon(1,\theta)}{\partial \theta^T} \\ \vdots \\ \frac{\partial \epsilon(N-1,\theta)}{\partial \theta^T} \end{bmatrix}.$$

Both the error-vector $E_N(\theta)$ and the Jacobian $\Psi_N(\theta)$ are required to perform a numerical minimization of the cost function. The next section on numerical

dfunlti

parameter estimation will show how $E_N(\theta)$ and $\Psi_N(\theta)$ are obtained using the toolbox function `dfunlti` (see manual on page 103). This name is an abbreviation of "Discrete-time cost Function for LTI systems".

Example 2.1 (Calculating an error-vector and its Jacobian)

The second-order system (2.2) will be an example in this case. The parameter vector θ will be taken equal to the actual output normal parameter vector of the system. The input signal $u(k)$ is a unit-variance Gaussian white-noise signal having $N = 1024$ samples. The output $y(k)$ is simulated based on the actual system and the input signal $u(k)$. The system matrices and signals described above can be loaded from the file `examples/SysData2ndOrder.mat` on the CD-ROM; they can be used to repeat the experiments in this example.

The following code-fragment will first parameterize the model in the output normal form after which both the error-vector $E_N(\theta)$ and its Jacobian $\Psi_N(\theta)$ are obtained. The toolbox function `dfunlti` is used to this end. This function needs the current parameter vector and the measured input and output signals. In addition, it needs the `params` structure generated by `dss2th` in order to determine the problem dimensions and properties.

```
>> [th,params]=dss2th(A,B,C,D,'on');
>> [EN,PsiN]=dfunlti(th,u,y,params);
```

Following the conventions in the textbook, the costs for a given θ can be calculated as $E_N^T E_N/(2N)$. Since in this example θ equals the parameter vector of the true system, the costs are very small, which can be verified as follows:

```
>> VN=EN'*EN/(2*N)
VN =
   2.4086e-29
```

This value is almost zero, but not exactly due to numerical round-off errors in the computer. The exact value one obtains when repeating this experiment may vary on different CPU architectures and operating systems.

The Jacobian $\Psi_N(\theta)$ contains 9 columns; one for each of the parameters. The ith column equals $-\partial\widehat{y}(k,\theta)/\partial\theta_i$. In other words, it shows the negative error vector differentiated with respect to θ_i. The signals for $i = 1,\ldots,9$ have been plotted in Figure 2.1. The large oscillating signal corresponds to the first column in $\Psi_N(\theta)$. The fact that this signal is rather large illustrates that the sensitivity of the error-vector —and thus that of the costs— to the first output normal parameter is large.

Note that Figure 2.1 shows that the Jacobian Ψ_N is not zero in the global minimum of the cost function. This is expected, since only the gradient $\Psi_N E_N$ of the costs must be zero at a minimum.

2.3.2 Numerical Parameter Estimation

This section will show how the actual numerical parameter estimation can be done using the toolbox software. Section 7.5 of the textbook covers both the

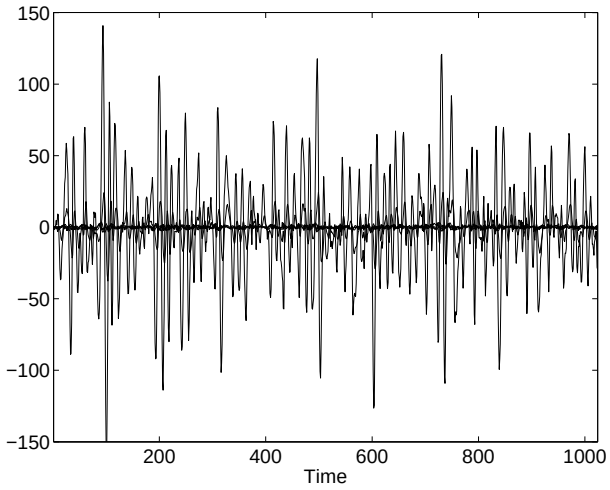


Figure 2.1: Signals in the Jacobian matrix $\Psi_N(\theta)$.

Gauss-Newton search method and the steepest-descent method. If the parameter vector at iteration i is given by $\theta^{(i)}$, then the parameter vector for the Gauss-Newton method at iteration $(i+1)$ is given as follows:

$$\theta^{(i+1)} = \theta^{(i)} - \mu^{(i)} \left(\Psi_N(\theta^{(i)})^T \Psi_N(\theta^{(i)}) \right)^{-1} \Psi_N(\theta^{(i)})^T E_N(\theta^{(i)}). \tag{2.9}$$

For the steepest-descent method, the parameter vector at iteration $(i+1)$ is the following:

$$\theta^{(i+1)} = \theta^{(i)} - \mu^{(i)} \Psi_N(\theta^{(i)})^T E_N(\theta^{(i)}). \tag{2.10}$$

The advantage of the Gauss-Newton update is that the parameter search converges quadratically if the search is performed near a minimum in the cost function. However, Gauss-Newton updates are based on approximating the cost function by a quadratic function. In practice, the cost function is generally non-quadratic, and this may result in the iterative Gauss-Newton update process to become unstable. The steepest-descent method, on the other hand, does not depend on the cost function being close to quadratic.The steepest-descent method attempts to update the parameters in the opposite direction of the gradient of the cost function. Its stability depends on the choice of the step size that is used in the update direction. The disadvantage of the steepest-descent method is that it converges slowly.

One would therefore like to use a method that uses the "safer" slower convergent steepest-descent updates in areas where the cost function behaves in a non-quadratic way, and that uses the "faster". Gauss-Newton updates wherever the cost function is close to quadratic. Such methods, which basically are methods in-between the Gauss-Newton and the steepest-descent method, indeed exist and are called Levenberg-Marquardt methods. In Levenberg-Marquardt methods, the inversion of the matrix $\left(\Psi_N(\theta^{(i)})^T \Psi_N(\theta^{(i)})\right)$ is regularized by adding

a constant λ times the identity to the matrix product before inversion. The Levenberg-Marquardt update can be written as follows:

$$\theta^{(i+1)} = \theta^{(i)} - \mu^{(i)} \left(\Psi_N(\theta^{(i)})^T \Psi_N(\theta^{(i)}) + \lambda^{(i)} I \right)^{-1} \Psi_N(\theta^{(i)})^T E_N(\theta^{(i)}). \quad (2.11)$$

It should be noted at this point that for $\lambda \downarrow 0$, the above expression approaches a Gauss-Newton update. For $\lambda \to \infty$, the regularization starts to dominate the matrix that is to be inverted, and the above expression approaches $(1/\lambda^{(i)})$ times a steepest-descent update. A Levenberg-Marquardt method thus indeed is a method in-between Gauss-Newton and steepest-descent. The division between Gauss-Newton and steepest-descent behavior roughly lies at $\lambda^{(i)} = \|\Psi_N^T \Psi_N\|_2$.

There exist several different Levenberg-Marquardt methods, and they differ in how the regularization parameter $\lambda^{(i)}$ is determined. In the toolbox software provided with this book, a trust-region based Moré-Hebden Levenberg-Marquardt implementation is used [6], which determines $\lambda^{(i)}$ such that the updated parameter lies in a region of the cost function, around the current estimate $\theta^{(i)}$, that is "trusted" to be sufficiently quadratic. The toolbox function `lmmore` (see manual on page 144) implements this Moré-Hebden method.

Immore

Example 2.2 (Parameter-optimization)

The functions `dss2th`, `dth2ss` and `lmmore` together provide enough functionality to identify a state-space model. This example will show how to use these functions to identify the system (2.2).

The input signal $u(k)$ has $N = 1024$ samples and is a unit-variance Gaussian white-noise signal. The output signal $y(k)$ is simulated in a noise-free fashion given the input signal $u(k)$ and the system matrices. The system matrices and signals described above can be loaded from the file `examples/SysData2ndOrder.mat` on the CD-ROM; they can be used to repeat the experiments in this example.

We will first parameterize a *disturbed* system in order to obtain an initial parameter estimate `th0` that does not already coincides with the cost function's global minimum. However, the disturbance is chosen such that `th0` does not lie too far away from the true parameter value, since in that case one might end up in a local minimum.

```
>> [th0,params]=dss2th(A+0.1,B,C,D,'on');
```

Subsequently, the numerical parameter search function `lmmore` is used to find the optimal parameter vector `th1`. The function `lmmore` has to be told which cost function to use, and this is done in the toolbox function `dfunlti`. In addition, the initial estimate `th0` and some default matrices are passed, as well as the arguments that are passed on to the cost function `dfunlti`:

```
>> th1=lmmore('dfunlti',th0,[],[],[],u,y,params)
Optimization terminated successfully:
 Search direction less than tolX
Optimization terminated successfully:
```

```
 Gradient in the search direction less than tolFun
th1 =
    0.8824
   -0.7000
    1.4003
    4.1133
    0.0000
```

The two messages state the reason why the optimization terminated successfully. A quick inspection of the optimized vector `th1` shows that it is equal to the output normal parameter vector that was obtained in Section 2.2.2. A next step might be to convert this parameter vector back into an optimized state-space model using the function `dth2ss`:

```
>> [A2,B2,C2,D2]=dth2ss(th1,params)
A2 =
    0.6176    0.4706
   -0.3294    0.8824
B2 =
    1.4003
    4.1133
C2 =
    0.7141         0
D2 =
   1.2485e-16
```

The toolbox function `lmmore` has been made syntax-compatible with the MATLAB 6 Optimization Toolbox function `lsqnonlin`. **lsqnonlin**

Although the optimization method in the previous example is rather straightforward, a wrapper-function "Discrete-time Optimization of LTI models", or `doptlti` (see manual on page 113), is available in the toolbox. In addition to providing a one-command optimization, this function provides extra functionality such as optimizing innovation models, unstable models and performing maximum likelihood optimizations. **doptlti**

Example 2.3 (Automated state-space model optimization)

The results obtained in the previous example can be obtained in one simple call to the `doptlti` wrapper-function as follows. Again, the datafile `examples/SysData2ndOrder.mat` on the CD-ROM can be used.

```
>> [A2,B2,C2,D2]=doptlti(u,y,A,B,C,D,[],[],'on')
Optimization terminated successfully:
 Search direction less than tolX
Optimization terminated successfully:
 Gradient in the search direction less than tolFun
A2 =
    0.6176    0.4706
   -0.3294    0.8824
B2 =
```

```
    1.4003
    4.1133
C2 =
    0.7141         0
D2 =
   1.2485e-16
```

When comparing these results to those in Example 2.2, it is clear that the resulting model is exactly the same.

Whereas the examples up to this point have mainly shown how to perform certain operations in MATLAB, the next example will graphically show how the parameter search in an optimization is carried out.

Example 2.4 (Graphical parameter search)

Figure 2.2 shows an actual parameter search for a slightly different system. A first-order SISO system was used here since its output-normal parameter vectors contains 3 elements. This is the maximum that can be visualized in three dimensions. The system is given by the following matrices:

$$A = 0.9048, \quad B = 0.2500, \quad C = 0.3807, \quad D = 0.$$

The input signal is a unit-variance Gaussian white-noise signal having $N = 512$ samples. The noise-free output signal is simulated given the input signal and the system matrices. Figure 2.2 shows level-contours (iso-surfaces) for the cost function. The black line shows the parameter search: each cross shows the parameter vector value for a given iteration.

In addition to providing a one-command LTI model optimization facility, the `doptlti` wrapper function also supports the optimization of innovation models. This implies that a prediction error identification is performed. A demonstration of these capabilities will be postponed until Section 2.3.4. The next section described the issues in using a full parameterization when identifying models.

2.3.3 Identifying Fully Parameterized State-Space Models

When using the full parameterization, the cost function calculation is a little different from that in Example 2.1. As stated in the textbook, the full parameterization is surjective, and a gradient projection is carried out in order to counteract this fact. The theory for this projection is described in Section 7.5.4 of the textbook, and its implication is described here. At every iteration, a unitary matrix U_2 is calculated that restricts parameters updates to directions that lie in its column space. In this way, only parameter updates that do not correspond to similarity transformations of the state-space model are allowed. The parameter-update rule (2.11) is modified as follows when incorporating the gradient projection:

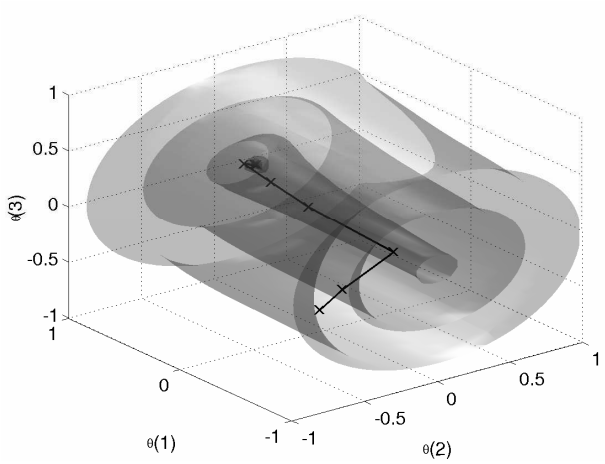


Figure 2.2: Contour-surfaces of the cost function and optimization trajectory in the parameter space Example 2.4. The translucent shapes represent surfaces on which the cost function is constant; the inner surfaces correspond to low costs while the outer surfaces correspond to high costs. From the small roughly spherical surface at the left to the outer cylindrical surface, the costs are 0.05, 0.1, 1, 5 and 12 respectively. The parameter-search is started at the cross on the lower end of the black line (high value of the cost function), and it converges to the global minimum (low value of the cost function) on the left side of the figure.

$$\theta^{(i+1)} = \theta^{(i)} - \mu^{(i)} U_2(\theta^{(i)}) \Phi(\theta^{(i)})^{-1} U_2(\theta^{(i)})^T \Psi_N(\theta^{(i)})^T E_N(\theta^{(i)}), \tag{2.12}$$

with

$$\Phi(\theta^{(i)}) = U_2(\theta^{(i)})^T \Psi_N(\theta^{(i)})^T \Psi_N(\theta^{(i)}) U_2(\theta^{(i)}) + \lambda^{(i)} I.$$

In this so-called projected gradient method, the product $\Psi_N U_2$ is calculated. Because of the well-chosen set of directions in U_2, this product $\Psi_N U_2$ is regular. The product is calculated directly, rather than first calculating Ψ_N and multiplying by U_2 later, for efficiency reasons.

Using (2.12), the `lmmore` optimization function is able to calculate parameter updates based on E_N, $\Psi_N U_2$ and U_2. The next example will show how this triple of matrices is calculated using the `dfunlti` toolbox function.

Example 2.5 (Calculating an error-vector and its projected Jacobian)

In this example, the same system and data as in Example 2.1 is used. However, we will now use the full parameterization. This changes the syntax of the cost function `dfunlti` as three output parameters are now returned: the error-vector E_N, its *projected* Jacobian $\Psi_N U_2$ and a matrix U_2. The three arguments are obtained from `dfunlti` as follows:

```
>> [th,params]=dss2th(A,B,C,D,'fl');
>> [EN,PsiNU2,U2]=dfunlti(th,u,y,params);
```

A visual inspection of the `U2` matrix shows that its size if 9×5. This is what one would expect: there is a total of 9 parameters, from which $n^2 = 4$ degrees of freedom correspond to similarity transformation. The columns of `U2` correspond to the remaining 5 directions in the parameter space. It is also apparent that the entries of the D-matrix —the 9th parameter in this case— are not influenced by similarity transformations at all.

```
U2 =
   -0.0648   -0.4762    0.6973   -0.1867         0
    0.3261    0.2634   -0.0778   -0.3119         0
    0.1342    0.4039    0.3917    0.3351         0
   -0.5086   -0.2104    0.2428    0.1622         0
    0.7580   -0.3926    0.1490   -0.0332         0
   -0.0598    0.2321    0.1842   -0.3968         0
    0.1043    0.5199    0.4838    0.1367         0
    0.1539   -0.1346   -0.0712    0.7429         0
         0         0         0         0    1.0000
```

2.3.4 Identification of Innovation-Type State-Space Models

This section describes additional capabilities of the `doptlti` LTI state-space model optimization function in the toolbox software. The following two examples will show how to identify innovation models and how to identify unstable models.

Example 2.6 (Identifying an innovation model)

In this example we will optimize an innovation model. The data is generated by the same second-order model that was used previously in Example 2.2. However, noise is added as follows:

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k) + w(k), \\ y(k) &= Cx(k) + Du(k) + v(k), \end{aligned}$$

with $w(k)$ and $v(k)$ zero-mean random sequences with covariance matrices

$$E\begin{bmatrix} w(k) \\ v(k) \end{bmatrix}\begin{bmatrix} w(j)^T & v(j)^T \end{bmatrix} = \begin{bmatrix} Q & S \\ S^T & R \end{bmatrix}\Delta(k-j) = \begin{bmatrix} \begin{bmatrix} 10^{-2} & 10^{-4} \\ 10^{-4} & 10^{-2} \end{bmatrix} & \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 \end{bmatrix} & 10^{-2} \end{bmatrix}\Delta(k-j).$$

A total of $N = 2048$ samples of input-output data are generated. These signals can be loaded from the datafile `examples/SysInnovData2ndOrder.mat` on the CD-ROM. They can be used to repeat the experiments in this example.

dlqe

Given the covariances of the noises, a Kalman gain K can be calculated using the function `dlqe` (from the MATLAB Control Systems Toolbox [2]). Note that `dlqe` also supports systems in which $x(k+1) = Ax(k) + Bu(k) + Gw(k)$. However, in our case $G = I_2$:

```
>> K=dlqe(A,eye(2),C,Q,R)
K =
    0.6693
    0.3417
```

First we will try to fit a normal output error state-space model to this data. We take a disturbed version of the actual system as initial estimate and optimize this model. Note that in this example we will use the full parameterization. This is indicated by either passing `'fl'` as parameterization type to `doptlti`, or by not specifying any parameterization type at all; `doptlti` uses the full parameterization by default. We will choose the latter option:

```
>> [Ao,Bo,Co,Do]=doptlti(u,y,A+0.1,B,C,D);
Optimization terminated:
 Relative decrease of cost function less than TolFun
```

It should be noted at this point that the same results would have been obtained with the output normal or tridiagonal parameterization. However, the full parameterization usually is slightly more accurate, and the syntax is simpler since no parameterization type needs to be specified.

A prediction error model is optimized by simply adding a nonempty Kalman gain matrix `K` to the parameter list as follows:

```
>> [Ak,Bk,Ck,Dk,x0k,Kk]=doptlti(u,y,A+0.1,B,C,D,[],K);
Optimization terminated:
 Relative decrease of cost function less than TolFun
```

Note that following the calling syntax specification on page 113, the parameter order is A, B, C, D, x_0 and K. That is why an empty matrix "`[]`" has to be passed between `D` and `K`; the initial state is assumed to be zero. For the output parameters the same ordering applies.

The estimated models can now be assessed using the variance accounted for (VAF) figure of merit as described in Chapter 10 of the textbook. First the output the output error and the innovation model are calculated, after which the toolbox function `vaf` (see manual on page 157) is used to calculate their VAFs:

vaf

```
>> yo=dlsim(Ao,Bo,Co,Do,u);
>> yk=dlsim(Ak-Kk*Ck,[Bk-Kk*Dk Kk],Ck,[Dk 0],[u y]);
>> vaf(y,yo)
ans =
   98.4387
>> vaf(y,yk)
ans =
   99.6534
```

It is clear that the innovation model calculates a better one-step ahead prediction of the system's output than the output error model.

If a model is unstable, then the calculation of the error-vector and the Jacobian presents a problem; both are simulated and will contain entries with extreme large magnitudes. The same problem may exist for the measured data itself if the actual system is unstable. However, we will assume that unstable systems are captured in a feed-forward or feedback control system such that the measured input and output sequences are finite.

Another problem is that output normal parameters can be calculated for the pair (A, C) only if this pair corresponds to a stable model. However, it is readily proven that for any observable pair (A, C), a matrix K can be found such that the eigenvalues of $(A - KC)$ lie within the unit circle. Therefore, the innovation form model can be stabilized.

An immediate problem in this approach is that the matrix K now is no longer a proper Kalman-gain, that is, there is no relation to the noise properties anymore. This causes the optimization problem with respect to the entries of the K-matrix to become ill-conditioned. The `doptlti` function therefore provides the possibility for supplying a matrix K that stabilizes A, but which is not optimized itself in the parameter search. In other words: the entries of K are fixed. This of course is only possible if the dynamics matrix of the optimized model, namely $(A_o - KC_o)$, is stable as well. If the initial guess is good, then generally both the initial $(A - KC)$ and the final $(A_o - KC_o)$ will be stable. The following example shows how to use this feature.

Example 2.7 (Identifying a marginally stable model)

Models encountered in the process industry often contain integrators. These system modes have poles at $z = 1$ and thus correspond to marginally stable models. In order to show the optimization of such a model, the second-order model (2.2) used throughout this chapter will be extended by adding an integrator at the input. This yields the following quadruple of state-space matrices (A, B, C, D):

$$A = \begin{bmatrix} 1.5 & -0.7 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix},$$
$$C = \begin{bmatrix} 1 & 0.5 & 0 \end{bmatrix}, \quad D = 0.$$

We will assume that our initial model estimate is disturbed, such that the disturbed A_p matrix equals the actual A matrix plus an additional 0.01 on all elements. First, a K-matrix that stabilizes $(A_p - KC)$ will be calculated using the function `place` (from the MATLAB Control Systems Toolbox [2]). The stabilized poles will be placed at $z = 0.7$ and $z = 0.7e^{\pm 0.1j\pi}$.

place

```
>> thepoles=[0.7 0.7*exp(j*pi*0.1) 0.7*exp(-j*pi*0.1)];
>> K=place((A+1e-2)',C',thepoles)'
place: ndigits= 15
K =
    0.3198
    0.3575
    0.0461
```

The system's input signal $u(k)$ has $N = 512$ samples and is a unit-variance Gaussian white-noise signal. The output signal $y(k)$ is simulated in a noise-free fashion based on the input signal $u(k)$ and the undisturbed actual system matrices. These signals can be loaded from the datafile `examples/SysData3rdOrder.mat` on the CD-ROM if one wishes to repeat the experiments in this example.

We need to tell `doptlti` that we wish to keep the K matrix constant. This is accomplished by passing an `optimset` options structure to `doptlti` (see manual on page 113). This options structure can be used to modify the optimization function's behavior. If MATLAB 6 is installed, this structure can be made using the `optimset` function. If an older version of MATLAB is used, the toolbox function `mkoptstruc` (see manual on page 150) provides a very rudimentary work-alike. After creating a structure, the field `options.OEMStable` is set to `'on'` to indicate that a stabilized output error model is optimized:

doptlti

optimset
mkoptstruc

```
>> options=mkoptstruc;
>> options.OEMStable='on';
```

The `options` variable should then be specified as an additional parameter to `doptlti` as follows:

```
>> [A2,B2,C2,D2,x02,K2]=doptlti(u,y,A+0.01,B,C,D,[],K,[],options);
Optimization terminated successfully:
 Search direction less than tolX
Optimization terminated successfully:
 Gradient in the search direction less than tolFun
```

We can now check whether the optimization has indeed converged to the marginally stable system. To this end, the eigenvalues of the optimized model are compared to those of the actual system:

```
>> eig(A2)-eig(A)
ans =
   1.0e-15 *
   0.1110 + 0.0555i
   0.1110 - 0.0555i
   0.2220
```

It is clear that the eigenvalues of the marginally stable system have been identified very accurately; the difference between the estimated and actual eigenvalues lies in the same order of magnitude as the machine precision $\epsilon_P = 2 \cdot 10^{-16}$. It should be noted at this point that the differences calculated above are due to rounding errors in the computer. The differences may vary between different CPU types and operating systems.

The `options.OEMStable` feature prevented any simulation problem because of instability, while keeping the problem well-conditioned by fixing the entries of the K-matrix.

2.4 Identifying State-Space Models of Known Order in the Frequency Domain

The previous section dealt with the optimization of state-space models based on time-domain measurements. In many cases this is convenient, since time-signal measurements can generally be obtained using relatively simple equipment. However, the time-domain framework requires all the samples to be equidistant in time. Stiff systems, that have both very slow and very fast modes, are difficult to handle in this respect: one needs a long measurement time to capture the slow modes, and a high sampling frequency to capture to fast modes. The result is that one needs very large data batches, which make the optimization computationally burdensome.

Another alternative is to use measurements of a system's frequency response function (FRF). This implies choosing a suitable set of frequencies, after which the FRF is measured for all input-output combinations at these frequencies. Since both magnitude and phase response are measured, the FRF is a complex-valued function; see Chapter 3 of the textbook. The advantage of these frequency-domain measurements is that the frequencies do not need to be equidistant. For stiff systems, this implies that one can use a loosely-spaced frequency-grid of just a small number of points. The system can be accurately identified by additionally clustering a larger number of frequencies around the interesting system modes. This keeps the amount of measurement data small. Another advantage of the frequency-domain framework is that it allows for the optimization of both discrete-time and continuous-time models.

In contrast to the time-domain method, the FRF method described in this section only handles output error models. The discrete-time model structure is given by

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k), \\ y(k) &= Cx(k) + Du(k). \end{aligned}$$

The continuous-time model structure is given by

$$\begin{aligned} \frac{\mathrm{d}x(t)}{\mathrm{d}t} &= Ax(t) + Bu(t), \\ y(t) &= Cx(t) + Du(t). \end{aligned}$$

Despite the differences in description in the time-domain, discrete-time and continuous-time models can be described in practically the same way in the frequency domain. For a given system with state-space matrices (A, B, C, D), the complex-valued FRF is given by

$$H(\xi_i) = C(\xi_i I_n - A)^{-1} B + D. \tag{2.13}$$

In this expression ξ_i denotes the ith complex frequency. Given FRF measurements for discrete-time systems at actual radial frequencies ω_i, the complex frequencies are given by

$$\xi_i = e^{j\omega_i T},$$

in which T is the sampling time to which the desired state-space model should correspond. These complex frequencies correspond to those used by the discrete-time Fourier transformation (DTFT) discussed in Section 3.3.2 of the textbook. Given FRF measurements for continuous-time systems at actual radial frequencies ω_i, the complex frequencies are given by

$$\xi_i = j\omega_i.$$

The next section will describe how a cost function is defined for FRF-based model optimizations. Subsequently, a numerical parameter estimation example is shown.

2.4.1 The Cost Function and its Gradient

Like in the time-domain, frequency-domain optimization is done by minimizing a cost function. However, the FRF is a complex matrix $H(\xi_i) \in \mathbb{C}^{\ell\times m}$. Since the FRF is assumed to be measured at N frequencies, the FRF will consists of N matrices $H(\xi_i) \in \mathbb{C}^{\ell\times m}$. Because it is a complex matrix, and the cost function $J_N(\theta)$ used for parameter identification is taken to be real valued, special precautions have to be taken.

First, all the FRF matrices are vectorized. Then, following the strategy for complex functions proposed in [7], the real and imaginary parts of the resulting vector are stacked. Finally, the elements of this real vector are squared and summed to obtain the actual costs. This gives rise to the following cost function:

$$\begin{aligned} J_N(\theta) &= \frac{1}{N}\sum_{i=1}^{N}\left\|H(\xi_i) - \widehat{H}(\xi_i,\theta)\right\|_F^2 \\ &= \frac{1}{N}\sum_{i=1}^{N}\sum_{p=1}^{\ell}\sum_{q=1}^{m} \mid H_{p,q}(\xi_i) - \widehat{H}_{p,q}(\xi_i,\theta) \mid^2 \\ &= \frac{1}{N}\sum_{i=1}^{N}\sum_{p=1}^{\ell}\sum_{q=1}^{m}\Big\{\mathrm{Re}\left(H_{p,q}(\xi_i) - \widehat{H}_{p,q}(\xi_i,\theta)\right)^2 \\ &\qquad + \mathrm{Im}\left(H_{p,q}(\xi_i) - \widehat{H}_{p,q}(\xi_i,\theta)\right)^2\Big\}. \end{aligned} \tag{2.14}$$

The predicted FRF $\widehat{H}(\xi_i,\theta)$ is obtained from the system matrices that correspond to the current parameter vector.

$$\widehat{H}(\xi_i,\theta) = C(\theta)(\xi_i I_n - A(\theta))^{-1}B(\theta) + D(\theta). \tag{2.15}$$

Like in the time-domain framework discussed in Section 2.3, a second-order approximation of the cost function is made, and an error-vector $E_N(\theta)$ and its Jacobian $\Psi_N(\theta)$ are defined. We will now show how these quantities are calculated using the toolbox function `ffunlti` (see manual on page 139). This name is an abbreviation of "Frequency-domain cost Function for LTI models". Like in the time-domain optimization framework, the output normal, tridiagonal and

ffunlti

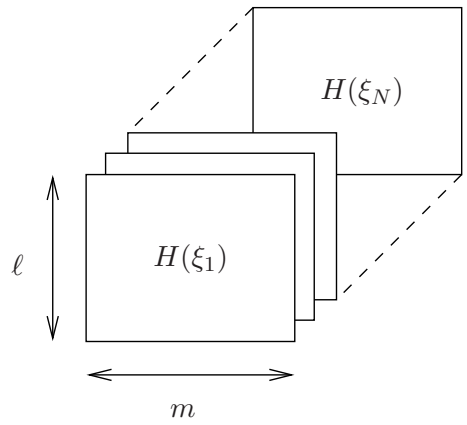


Figure 2.3: Layout of the 3D-array returned by `ltifrf`.

full parameterization types can be used. In this section we will use the output normal parameterization as an example.

The second-order system (2.2) will be an example in this case. The parameter vector θ will be taken equal to the actual output normal parameter vector of the system. A total of $N = 512$ complex frequencies are distributed over the upper part of the unit circle in the complex plane. The corresponding FRF is calculated in a noise-free fashion. This frequency-vector and FRF can be loaded from the datafile `examples/SysData2ndOrder.mat` on the CD-ROM. The following code-fragment will first parameterize the model in the output normal form after which both the error-vector $E_N(\theta)$ and its Jacobian $\Psi_N(\theta)$ are obtained. The toolbox function `ffunlti` is used to this end. This function needs the current parameter vector, the measured FRF and the frequencies at which it is measured. In addition, it needs the `params` structure generated by `dss2th` in order to determine certain dimensions and properties. The measured FRF is stored in the MATLAB 3D-array `H`, which is formatted as shown in Figure 2.3. The complex frequencies are stored in a MATLAB vector `w`:

```
>> [th,params]=dss2th(A,B,C,D,'on');
>> [EN,PsiN]=ffunlti(th,H,w,params);
```

As the parameter vector `th` corresponds to the actual system, the costs should be very small. This can be verified by calculating the costs $E_N^T E_N/(2N)$ at the current parameter vector `th`:

```
>> VN=EN'*EN/(2*N)
VN =
   2.2232e-29
```

This value of the cost function is very small. Ideally, it would have been zero, but it is slightly different from zero because of rounding errors in the computer. The exact value may vary between different CPU types and operating systems.

2.4.2 Numerical Parameter Estimation

This section will show how the actual numerical parameter estimation based on frequency-domain data can be carried out using the toolbox software. Like in the time-domain framework, the Moré-Hebden Levenberg-Marquardt parameter search algorithm implemented in `lmmore` is used.

The functions `dss2th`, `dth2ss`, `ffunlti` and `lmmore` together would provide enough functionality to identify a state-space model based on FRF data. The procedure is fully analogous to the one used in Example 2.2 on page 20 for time-domain data. The only line of code that needs to be changed is the call to `lmmore`, because `lmmore` needs to be told to use the frequency-domain cost function `ffunlti` and the frequency-domain data from `H` and `w`:

```
>> th1=lmmore('ffunlti',th0,[],[],[],H,w,params)
```

In this section, we will show an example in which a *continuous*-time model is optimized.

Example 2.8 (Continuous-time state-space model optimization)

In this example a continuous-time state-space model will be identified. The following state-space model defined by the quadruple of system matrices (A, B, C, D) will be considered [8]:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & -0.2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -25 & -0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -9 & -0.12 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}, \quad D = 0. \tag{2.16}$$

The FRF is calculated at $N = 512$ frequencies uniformly spaced in the band $[0.01, 9]$ rad/s. This frequency-vector and FRF can be loaded from the datafile `examples/SysCData6thOrder.mat` on the CD-ROM in order to repeat the experiments in this example.

We will first parameterize a *disturbed* system that is close to the actual system. The parameter vector will then be optimized. Since this is a *continuous*-time system, the continuous-time `css2th` parameterization function has to be used:

```
>> [th0,params]=css2th(A+1e-2,B,C,D,'on');
```

The parameter search function `lmmore` is subsequently used to find the optimal parameter vector `th1`. The information that is passed to `lmmore` is similar to the information used in Example 2.2: namely, (1) which cost function to use, (2) the FRF and the frequencies at which it is measured and (3) the `params` structure that was generated by `css2th`. In addition, an extra parameter `'cont'` is passed to indicate to `ffunlti` that continuous-time model is used. This is important since the definition of the output normal form is different between

continuous-time and discrete-time models [3]. In addition, the optimization framework's internal stability checking functions differ between discrete-time and continuous-time models.

```
>> th1=lmmore('ffunlti',th0,[],[],[],H,w,params,[],'cont')
Optimization terminated successfully:
 Gradient in the search direction less than tolFun
th1 =
    1.2806
    4.1200
    2.5751
    1.9805
    2.0680
    1.8383
   -0.0000
    0.5686
    0.1207
    1.3174
    0.0425
    0.9721
    0.0000
```

We can verify that this parameter vector is indeed the right one because the developed parameterization of the output normal form is unique. We will subtract the output normal parameter vector corresponding to the undisturbed system and take the norm of this vector: this norm should be very small:

```
>> norm(th1-css2th(A,B,C,D,'on'))
ans =
   2.7405e-09
```

Ideally, one would like to see a value in the order of 10^{-15} here. However, although the output normal parameterization is unique, it is not the most accurate of the available parameterizations. The above value is sufficiently small. Like in the time-domain case, we can subsequently use `cth2ss` to convert the optimized parameter vector back into a model:

```
>> [A2,B2,C2,D2]=cth2ss(th1,params);
```

The shape of the cost function is shown in Figure 2.4. It is clear that apart from the global minimum, there exist strong local minima. This example illustrates that a good initial model guess is of paramount importance.

---

foptlti

Like in the time-domain case, an optimization wrapper function is incorporated in the toolbox. The "Frequency-domain Optimization of LTI models" function `foptlti` (see manual on page 141) can be used as a one-command optimization for both continuous-time and discrete-time state-space models. For the model in the above example, the entire optimization can be performed using only one command:

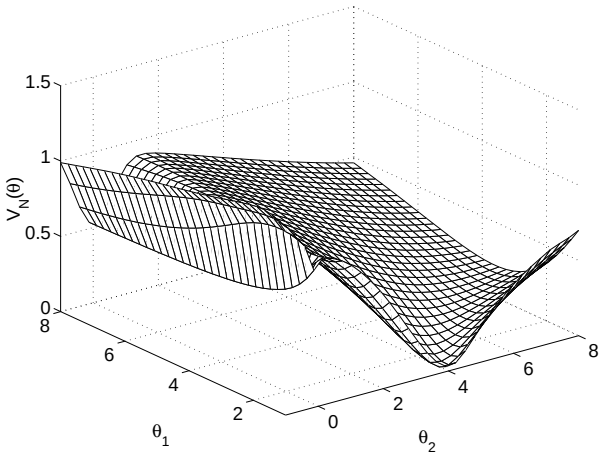


Figure 2.4: Shape of the cost function $J_N(\theta)$. Parameters θ_1 and θ_2 are varied. These parameters correspond to the pair (A, C) of the output normal parameterization.

```
>> [A2,B2,C2,D2]=foptlti(H,w,A+1e-2,B,C,D,'on',[],'cont');
Optimization terminated successfully:
 Gradient in the search direction less than tolFun
```

References

[1] T. McKelvey, "Frequency domain system identification with instrumental variable based subspace algorithm," in *Proceedings of the 16th Biennal Conference on Mechanical Vibaration and Noise*, (Sacramento, California), Sept. 1997.

[2] The MathWorks Inc., Natick, Massachusetts, *Using the Control Systems Toolbox*, version 1 (release 12) ed., Nov. 2000.

[3] B. Haverkamp, *Subspace Method Identification, Theory and Practice*. PhD thesis, Delft University of Technology, Delft, The Netherlands, 2000.

[4] L. H. Lee and K. Poolla, "Identification of linear parameter-varying systems using nonlinear programming," *Journal of Dynamic Systems, Measurement and Control*, vol. 121, pp. 71–78, Mar. 1999.

[5] V. Verdult, *Nonlinear System Identification: A State-Space Approach*. PhD thesis, University of Twente, Twente, The Netherlands, 2002.

[6] J. J. Moré, "The Levenberg-Marquardt algorithm: Implementation and theory," in *Numerical Analysis* (G. A. Watson, ed.), vol. 630 of *Lecture Notes in Mathematics*, pp. 106–116, Springer Verlag, 1978.

[7] M. Hayes, *Statistical Digital Signal Processing and Modeling*. New York: John Wiley and Sons, 1996.

[8] P. van Overschee and B. De Moor, "Continuous-time frequency domain subspace system identification," *Signal Processing*, vol. 52, no. 2, pp. 179–194, 1996.

Chapter 3

Subspace Model Identification

After studying this chapter you can:

- implement a simple MOESP subspace algorithm in less than 20 lines of MATLAB code.
- perform time-domain subspace identification in MATLAB using the toolbox functions `dordpi`, `dordpo`, `dmodpi`, `dmodpo`, `dac2bd` and `dinit`.
- concatenate time-domain data batches from different experiments to identify discrete-time state-space models.
- identify discrete-time state-space models based on frequency-domain data using the toolbox functions `fdordom`, `fdmodom` and `fac2bd`.
- identify continuous-time state-space models based on frequency-domain data using the toolbox functions `fcordom`, `fcmodom` and `fac2bd`.
- concatenate frequency-domain data batches from different experiments to identify discrete-time state-space models.

3.1 Introduction

This chapter is a companion to Chapter 9 of the textbook. In this chapter, we illustrate how subspace identification can be performed within MATLAB using the toolbox software. In Section 3.2 we will show that the simple linear algebra steps of Ordinary MOESP can be implemented in less than 20 lines of MATLAB code. Section 3.3 explains how PI-MOESP and PO-MOESP subspace identification can be carried out using the toolbox software. Section 3.4 describes an extension of the subspace methods that can be used to identify discrete-time and continuous-time state-space models based on frequency response function (FRF) measurements.

3.2 Subspace Identification using Arbitrary Inputs: Ordinary MOESP

This section will illustrate the Ordinary MOESP subspace algorithm that has been developed in Section 9.2.4 of the textbook. Although the theory will be summarized in this section, it should be stressed that the emphasis will not be on the theory. Rather, given the fact that the algorithm steps in subspace identification are linear algebra operations, we will show how an Ordinary MOESP algorithm can be implemented in just a few lines of MATLAB code. Although a MIMO case is more or less equivalent from the theoretical point of view, a MIMO implementation contains a rather large number of practical details that would distract us from the simplicity of the Ordinary MOESP algorithm. Therefore, we will confine ourselves to a SISO implementation and leave the MIMO implementation as an exercise to the reader.

Step 1: Data Compression and Order Estimation

The Ordinary MOESP algorithm is a linear subspace identification algorithm that identifies systems based on measured input and output data, where the input data can be arbitrary, provided that it is sufficiently exciting.

The algorithm's starting point is the data equation which, for a noise-free case, can be written as

$$Y_{0,s,N} = \mathcal{O}_s X_{0,N} + \mathcal{T}_s U_{0,S,N}. \tag{3.1}$$

The next step is to project the output Hankel matrix $Y_{0,s,N}$ onto the orthogonal complement of the row space of the input Hankel matrix $U_{0,s,N}$, which cancels the second term on the right-hand side of the data equation:

$$Y_{0,s,N}\Pi^{\perp}_{U_{0,s,N}} = \mathcal{O}_s X_{0,N}\Pi^{\perp}_{U_{0,s,N}}. \tag{3.2}$$

As we have shown in (9.21) on page 265 of the textbook, this projection can be efficiently implemented by using an RQ-factorization

$$\begin{bmatrix} U_{0,s,N} \\ Y_{0,s,N} \end{bmatrix} = \begin{bmatrix} R_{11} & 0 \\ R_{21} & R_{22} \end{bmatrix} \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix}, \tag{3.3}$$

after which the R_{22} matrix theoretically has the same column space as the unknown extended observability matrix $\mathcal{O}_s$. The order of the system is determined from the rank of R_{22}. This rank is obtained from the following singular value decomposition:

$$R_{22} = U_n \Sigma_n V_n^T. \tag{3.4}$$

Step 2: Estimation of A and C

The order n is taken equal to the number of "large" singular values, while the "small" singular values are attributed to noise or round-off errors. It has been shown in Lemma 9.3 on page 268 of the textbook, that under the appropriate noise conditions, the matrix U_n forms a consistent estimate of the system's extended observability matrix $\mathcal{O}_s$ up to an unknown similarity transformation. The $\widehat{A}$ and $\widehat{C}$ estimates are then obtained exploiting the shift-structure of $\mathcal{O}_s$:

$$\widehat{A} = U_n(1 : (s-l)\ell, :)^{\dagger} U_n(\ell + 1 : s\ell, :), \tag{3.5}$$

$$\widehat{C} = U_n(1 : \ell, :). \tag{3.6}$$

Step 3: Estimation of B, D and the Initial State.

Technically, only steps 1 and 2 are specific to Ordinary MOESP subspace identification. The estimation of the B and D matrices is done using a method common to Ordinary MOESP, PI-MOESP and PO-MOESP. The estimation of the B and D matrices and the initial state is a least squares regression problem of which the solution is given by

$$\begin{bmatrix} x_0 \\ \mathrm{vec}(\widehat{B}) \\ \mathrm{vec}(\widehat{D}) \end{bmatrix} = \Phi^{\dagger} Y_{0,N,1}.$$

The regression matrix Φ and data vector $Y_{0,N,1}$ are given by:

$$\Phi = \begin{bmatrix} C & 0 & u(1)^T \otimes I_\ell \\ CA & u(1)^T \otimes C & u(2)^T \otimes I_\ell \\ \vdots & \vdots & \vdots \\ CA^{N-1} & \sum_{\tau=0}^{N-2} u(\tau+1)^T \otimes CA^{N-2-\tau} & u(N)^T \otimes I_\ell \end{bmatrix},$$

$$Y_{0,N,1} = \begin{bmatrix} y(1) \\ y(2) \\ \vdots \\ y(N) \end{bmatrix}.$$

The next two examples will show how the above three steps of Ordinary MOESP subspace identification can be implemented in a few lines of MATLAB code for a SISO system.

Example 3.1 (Simple Ordinary MOESP implementation: Steps 1 and 2)

We will now show how the Ordinary MOESP algorithm can be implemented in MATLAB for a SISO system. The MATLAB implementation will be posed first, after which a line-by-line walk-through of the code is given.

Given an input sequence `u` and output sequence `y`, both of length `N`, and a block-size parameter `s`, the following code-fragment implements steps 1 and 2 of the Ordinary MOESP algorithm: it estimates `A` and `C` based on time-domain measurements:

```
U=hankel(u(1:s),u(s:N));
Y=hankel(y(1:s),y(s:N));
R=(triu(qr([U;Y]')))';
[Un,Sn,Vn]=svd(R(s+1:2*s,s+1:2*s));
A=Un(1:s-1,1:n)\Un(2:s,1:n);
C=Un(1,1:n);
```

The first major "step" in Ordinary MOESP subspace identification consists of three components that together correspond to the first four lines of MATLAB code in the above implementation.

The first component is the formation of the Hankel matrices $U_{0,s,N}$ and $Y_{0,s,N}$. In the MATLAB implementation, these matrices are called `U` and `Y`. Two immediate problems are that MATLAB matrix indices start at 1 rather than 0, and that the theoretical definition of the Hankel matrices requires $N + s - 1$ samples to be available, while only N are available in practice. We therefore create Hankel matrices $U_{1,s,N-s+1}$ and $Y_{1,s,N-s+1}$, which solves both problems. The MATLAB function `hankel` is used for the formation of the Hankel matrices. This is shown in the first two lines of the MATLAB implementation.

The second component is the projection onto the orthogonal complement of the row-space of $U_{1,s,N-s+1}$. This is theoretically accomplished using a RQ-factorization, but MATLAB provides only a QR-factorization function, so the following factorization is carried out in practice:

$$\begin{bmatrix} U_{1,s,N-s+1}^T & Y_{1,s,N-s+1}^T \end{bmatrix} = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}. \tag{3.7}$$

It should be noted as this point that only the R-matrix of this factorization is needed. MATLAB can be told not to form the Q matrix explicitly. As Q is a large matrix of size $N\ell \times N\ell$, not calculating Q saves a considerable amount of storage space and calculation time. If MATLAB's `qr` function is called with one output argument, it will call the LAPACK function `DGEQRF` internally. This returns a full matrix of which the upper triangle equals the required R-factor. The function `triu` is used to retrieve this upper-triangular part, after which the entire R-factor has to be transposed in order to correspond to the R-matrix of an RQ-factorization. The third line of the MATLAB implementation performs this action.

qr

triu

The final component in the first step in Ordinary MOESP is the singular value decomposition of the R_{22} matrix. This decomposition is performed in a straightforward way by MATLAB's internal `svd` function. As the algorithm described in this section is a SISO algorithm, the expression `R(s+1:2*s,s+1:2*s)` equals the R_{22} matrix. The fourth line of the MATLAB implementation performs this decomposition.

svd

The second step in Ordinary MOESP is the estimation of A and C. However, the system order has to be selected first by inspecting the singular values. A more detailed example on how to estimate the order will be given in Section 3.3.2. However in this case we will assume that this order is `n`. Subsequently, A and C, which are are called `A` and `C` in MATLAB, are obtained exactly as shown in the theory above. Lines 5 and 6 of the MATLAB implementation perform this estimation.

The next example will show how the third step of Ordinary MOESP, the estimation of B and D, can be implemented. It should again be noted that the implementation can be programmed in just a few lines of MATLAB code.

Example 3.2 (Simple estimation of B and D)

Given an input sequence u and output sequence y, both of length N, and the A and C estimates obtained earlier, the following code-fragment estimates B and D for SISO systems.

```
Phi=zeros(N,n+1);
for i=1:n,
  dB=zeros(n,1);
  dB(i,1)=1;
  Phi(:,i)=dltisim(A,dB,C,0,u);
end;
Phi(:,i+1)=u;
BD=Phi\y;
B=BD(1:n);
D=BD(n+1);
```

Estimating B and D from the regression problem start with setting up the matrices for this problem. In the MIMO case, this procedure is rather involved. However, in the SISO case the Φ matrix simplifies considerably. If we, in addition, assume that the initial state is zero, then the Φ matrix and regression problem reduce to

$$\Phi = \begin{bmatrix} 0 & u(1) \\ u(1)C & u(2) \\ \vdots & \vdots \\ \sum_{\tau=0}^{N-2} u(\tau+1)CA^{N-2-\tau} & u(N) \end{bmatrix},$$

$$\begin{bmatrix} \widehat{B} \\ \widehat{D} \end{bmatrix} = \Phi^{\dagger} Y_{0,N,1}.$$

Filling the first block-column of Φ is the most complicated part of practically estimating B and D. However, a closer inspection of each of the block-columns reveals that its ith column is the output of an LTI system with $D = 0$ and B equal to zero except for its ith element. The function `dltisim` (see manual on page 108) can be used to simulate these auxiliary systems. The `for`-loop in the MATLAB implementation fills the first block-column of Φ.

dltisim

Filling the last block-column of Φ is a simple matter of copying the input signal `u` to this column. The remaining code-lines in the MATLAB implementation solve the least squares problem and extract the `B` and `D` matrices.

We have thus shown that estimating the B and D matrices of a SISO system is possible in just 10 lines of MATLAB code. Obviously, the MIMO case is more involved.

3.3 Subspace Identification with Instrumental Variables

In the previous section we have shown that the implementation of a basic subspace identification algorithm can be done in a few lines of MATLAB code. However, in practice the code in the previous section is not used. First, because one often wishes to identify MIMO systems. Second because the Ordinary MOESP algorithm is not used extensively in practice because of the limited noise-conditions under which it delivers consistent model estimates. Finally the code given in the previous section contains no checks or usable function-interfaces.

The toolbox software contains efficient routines to perform both PI-MOESP [1] and PO-MOESP [2] subspace identification, for which the theory is described in Sections 9.5 and 9.6 of the textbook. This section contains a brief tutorial on how to use these toolbox functions, as well as a number of examples.

3.3.1 General Subspace Identification Procedure

Figure 3.1 shows the general steps that have to be taken in order to identify a discrete-time state-space model using the PI-MOESP or PO-MOESP algorithms. Subspace identification in the toolbox is built up of four parts: the order estimation, the estimation of A and C, the estimation of B and D and the estimation of the initial state x_0.

The order estimation functions compress the data and obtain singular values based on which the required model order can be determined. The functions performing these tasks have names containing "`ord`" in the toolbox. The `dordpo` (see manual on page 119) function, for example, performs a "Discrete-time Order estimation for PO-MOESP". The function `dordpi` (see manual on page 117) function does the same for PI-MOESP. In addition to the measured input and output signals, the "`ord`" functions need a model structure specification. However, in the toolbox framework this structure specification reduced to just one scalar: an upper bound s on the order of the system. A call to an "`ord`" function

dordpo

dordpi

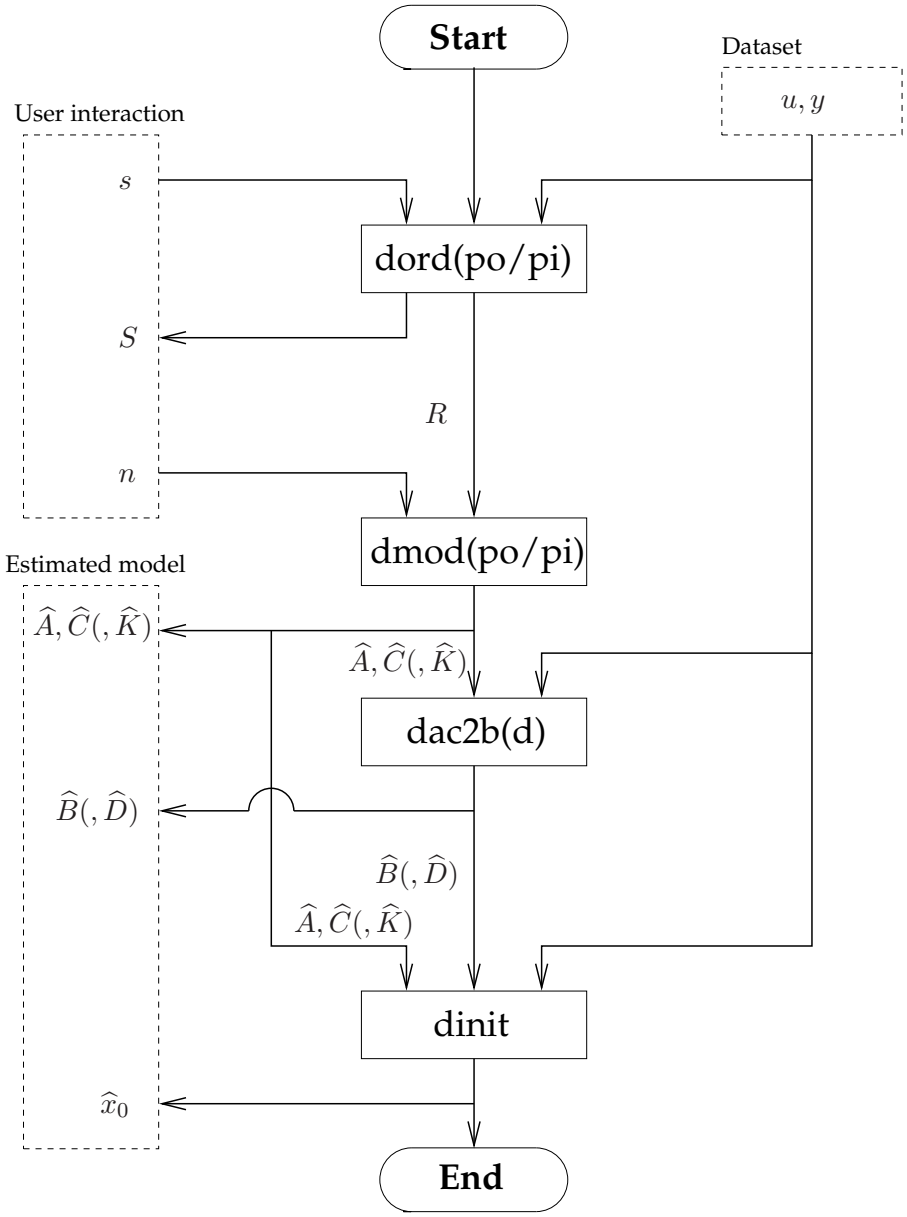


Figure 3.1: Flow-graph for PI-MOESP and PO-MOESP time-domain subspace identification using the toolbox software. Note that a Kalman gain estimate $\widehat{K}$ can be obtained only in the PO-MOESP algorithm; the PI-MOESP algorithm is an output error algorithm that does not provide a Kalman gain estimate.

yields a singular value vector S and a compressed data matrix R. The selection of the model order is done based on the singular values stored in S. A number of simple examples on order selection are given in this section, Section 3.4 and in Example 9.4 on page 270 of the textbook. A more elaborate example is provided in the case study in Section 4.6. The "ord" functions allow multiple data batches from separate experiments to be concatenated. This functionality will be discussed in Section 3.3.3.

Once the data has been compressed, the "mod" functions estimate the A and C matrices from the extended observability matrix estimate $\widehat{\mathcal{O}}_s$ that is part of the compressed data matrix R. The dmodpo (see manual on page 110) and dmodpi (see manual on page 109) function perform these tasks for PO-MOESP and PI-MOESP respectively. The user should specify the desired model order n. In PO-MOESP, a Kalman gain K can be estimated as well. It should be noted that R matrices are incompatible among the different algorithms, that is dmodpo cannot be used on an R matrix generated by dordpi: dmodpi should be used for that.

dmodpo
dmodpi

Once the estimates $\widehat{A}$ and $\widehat{C}$ have been calculated, the matrices B and D can be estimated. For time-domain subspace identification either the function dac2b (see manual on page 98) or the function dac2bd (see manual on page 100) can be used. The first function assumes that D is known to be zero and estimates only B. The second function estimates both B and D. Like the "ord"-functions, the dac2b and dac2bd function can be used with multiple data batches. This functionality will be discussed in Section 3.3.3. Note that dac2b and dac2bd can be used with both PI-MOESP and PO-MOESP.

dac2b
dac2bd

Finally, the user may wish to estimate the initial state x_0. The function dinit (see manual on page 106) can be used for this purpose. It needs all system matrix estimates $\widehat{A}, \widehat{B}, \widehat{C}, \widehat{D}$ and a data batch u, y.

dinit

3.3.2 Using the Toolbox Software: PI-MOESP

In this section we will give an example of how the subspace identification framework in the toolbox software is used in a practical situation. The PI-MOESP scheme functions dordpi and dmodpi will be used to estimate A and C, after which dac2bd and dinit are used to estimate B, D and the initial state. Finally, a simple model validation is performed.

First we generate data for the second-order system (2.2) on page 12. A pseudo-random binary sequence will be generated as identification input using the toolbox function prbn (see manual on page 152). A pseudo-random binary sequence, or PRBN, is an often-used identification input signal. The theory behind this kind of signal is described in [3]. In rather loose terms, samples of a PRBN are either 0 or 1, and after each sample there is a certain probability $\rho \in [0, 1]$ that the signal switches state (from 0 to 1 or from 1 to 0). In the following example ρ is set to 0.1 in order to get a signal that changes state rather infrequently and which therefore mainly contains signal energy in the lower frequency range.

prbn

```
>> A=[1.5 -0.7;1 0]; B=[1;0]; C=[1 0.5]; D=0;
>> u=prbn(300,0.1)-0.5;
>> y=dltisim(A,B,C,D,u);
```

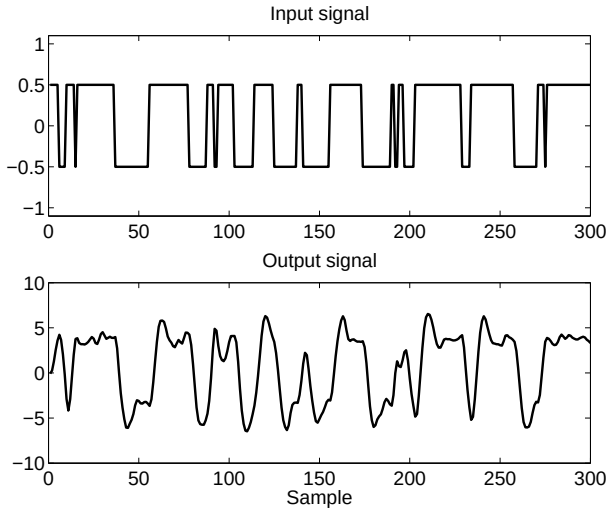


Figure 3.2: Input and disturbed output signals of the system.

In order to make the example more realistic, the output signal is disturbed by colored measurement noise such that a 20 dB SNR is obtained. The `b` and `a` vectors correspond to the numerator and denominator polynomials of a low-pass filter.

```
>> b=[0.17 0.50 0.50 0.17];
>> a=[1.0 0 0.33 0];
>> y=y+0.1*std(y)*filter(b,a,randn(300,1));
```

The generated input and output signals are shown in Figure 3.2. The exact data sequences used in this example can be loaded from the datafile `examples/PI-MOESP-tutorial.mat` on the CD-ROM.

We will now try to identify the system from the generated data set by assuming the following state-space model structure:

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k), \qquad x(0) = x_0, \\ y(k) &= Cx(k) + Du(k) + v(k). \end{aligned}$$

This model, in which $v(k)$ is a colored noise signal, falls within the class of models for which PI-MOESP can provide consistent estimates, as was illustrated in Section 9.5 of the textbook. In identifying this model we will use the general flow-graph that was shown in Figure 3.1.

Step 1: Data Compression and Order Estimation

In this first step we use the "`ord`" function —`dordpi` in this PI-MOESP case— in order to compress the available data and to generate a model order estimate. The *only* model structure selection parameter that we need to pass is the block-size `s`, which should be larger than the expected system order. We will use the rather

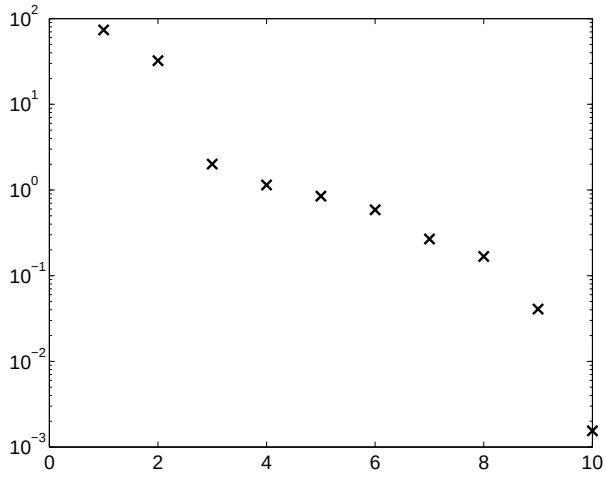


Figure 3.3: Singular values generated by `dordpi` PI-MOESP data compression function.

high value `s=10` in this example in order to show the order selection mechanism more clearly.

```
>> s=10;
>> [S,R]=dordpi(u,y,s);
```

The function `dordpi` returns a vector `S` containing singular values based on which the model order can be determined. In addition, a compressed data matrix `R` is returned that is used by `dmodpi` to estimate A and C in the next step. The singular values in `S` are plotted in Figure 3.3 using the following command:

```
>> semilogy(1:10,S,'x')
```

In a noise-free case, only the first n singular values would have been nonzero. However, the singular values that would have been zero are now disturbed because of the noise. Still, a *gap* is visible between singular values 2 and 3, and so the model order will be chosen equal to `n=2`.

Step 2: Estimation of A and C

In this step we will obtain estimates for A and C. As the `A` and `C` variables have already been defined, we will call the estimates for A and C, `Ae` and `Ce` respectively. The function `dmodpi` is used to determine `Ae` and `Ce` based on the `R` matrix from `dordpi` and the model order `n` determined from the singular value plot.

```
>> n=2;
>> [Ae,Ce]=dmodpi(R,n);
```

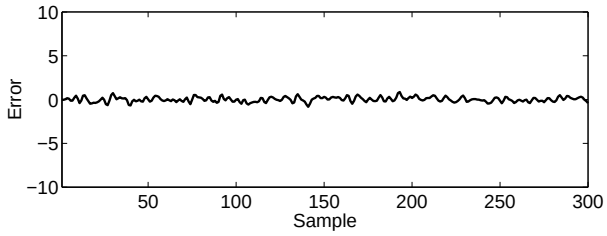


Figure 3.4: The error signal $y(k) - \widehat{y}(k)$.

Step 3: Estimation of B, D and the Initial State.

Once estimates `Ae` and `Ce` for A and C have been determined, the toolbox function `dac2bd` will be used to estimate B and D, as `Be` and `De` respectively. The function `dac2bd` requires the estimates for A and C and the measured input-output data.

Subsequently, the toolbox function `dinit` is used to obtain the initial state x_0 corresponding to the current data set. This function needs estimates for all system matrices as well as the measured input-output data.

```
>> [Be,De]=dac2bd(Ae,Ce,u,y);
>> x0e=dinit(Ae,Be,Ce,De,u,y);
```

Model Validation

The quality of the model `(Ae,Be,Ce,De)` that has been identified will now be assessed. To this end, we will compare the output predicted by the identified model to the measured output signal. As a figure of merit, we use the variance accounted for (VAF), which is described in more detail in Section 4.5.4. If the model is good, the VAF should be close to 100%. The following code fragment simulated the estimated model using the measured input signal in order to obtain the estimated output `ye`. Subsequently, the VAF is calculated using the toolbox function `vaf` (see manual on page 157). **vaf**

```
>> ye=dltisim(Ae,Be,Ce,De,u,x0e);
>> vaf(y,ye)
ans =
   99.4479
```

It is clear that the estimated model described the actual system behavior well. The error signal $y(k) - \widehat{y}(k)$ is plotted in Figure 3.4. This error is very small compared to the output signal in Figure 3.2.

3.3.3 Using the Toolbox Software with Multiple Data Sets.

In this section we show how the toolbox software can be used with multiple data batches that each result from a separate experiment on the same system. This functionality is important for two reasons. First, rather than performing

one large experiment on a system, a number of experiments that are unconnected in time can be performed. Second, very large data batches can be split into smaller batches in order to prevent excessive memory usage. It should be noted that although PO-MOESP is used in this multiple-batch experiment, the toolbox-functions for PI-MOESP can also be used with multiple batches in exactly the same way as the PO-MOESP functions. A flow-graph example for two data batches is given in Figure 3.5. In general, the "`ord`"-functions deliver a vector `S` containing the singular values and a compressed data matrix `R`. This data matrix can be used to estimate A and C using the "`mod`"-function immediately. However, it can also be used as the fourth input argument in a next call to the "`ord`"-function with an additional data batch. This mechanism is shown for two data batches in Figure 3.5. However, an arbitrary number of batches can be concatenated in this way.

The B,D-estimation functions `dac2b(d)` can also be used with multiple data batches. The mechanism is different from that of the "`ord`"-functions; all data batches can be processed in one call to `dac2b(d)`. Again, the mechanism is shown graphically in Figure 3.5 for two batches.

The system under consideration in this section is a three degree-of-freedom mass spring system stated in [4]. In this article, the following state-space model is derived:

$$A = \begin{bmatrix} 0.9856 & 0.1628 & 0 & 0 & 0 & 0 \\ -0.1628 & 0.9856 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.8976 & 0.4305 & 0 & 0 \\ 0 & 0 & -0.4305 & 0.8976 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.8127 & 0.5690 \\ 0 & 0 & 0 & 0 & -0.5690 & 0.8127 \end{bmatrix},$$

$$B = \begin{bmatrix} 0.0011 & 0.0134 & -0.0016 & -0.0072 & 0.0011 & 0.0034 \end{bmatrix}^T,$$

$$C = \begin{bmatrix} 1.5119 & 0 & 2 & 0 & 1.5119 & 0 \\ 1.3093 & 0 & 0 & 0 & -1.3093 & 0 \end{bmatrix}.$$

The system's D-matrix is assumed to be zero. In addition, the covariance matrices of the process noise and measurement noise are given by Q and R respectively:

$$Q = 10^{-4} \cdot \mathrm{diag}\left(\begin{bmatrix} 0.0242 & 3.5920 & 0.0534 & 1.034 & 0.0226 & 0.2279 \end{bmatrix}\right),$$
$$R = 10^{-2} \cdot \mathrm{diag}\left(\begin{bmatrix} 2.785 & 2.785 \end{bmatrix}\right).$$

The problem defined as such falls within the class of innovation model problems, for which the PO-MOESP algorithm provides consistent estimates [2]. The toolbox functions `dordpo` and `dmodpo` are used to perform the data compression and estimation of A and C.

Three batches of input-output data are generated. The first input batch $u_1(k)$ is a pink-noise signal. The second batch $u_2(k)$ is a multi-sine input and the third batch $u_3(k)$ is a pseudo-random binary sequence. A fourth pink-noise batch $u_4(k)$ is used for validation purposes. Figure 3.6 shows the input signals of the first three batches.

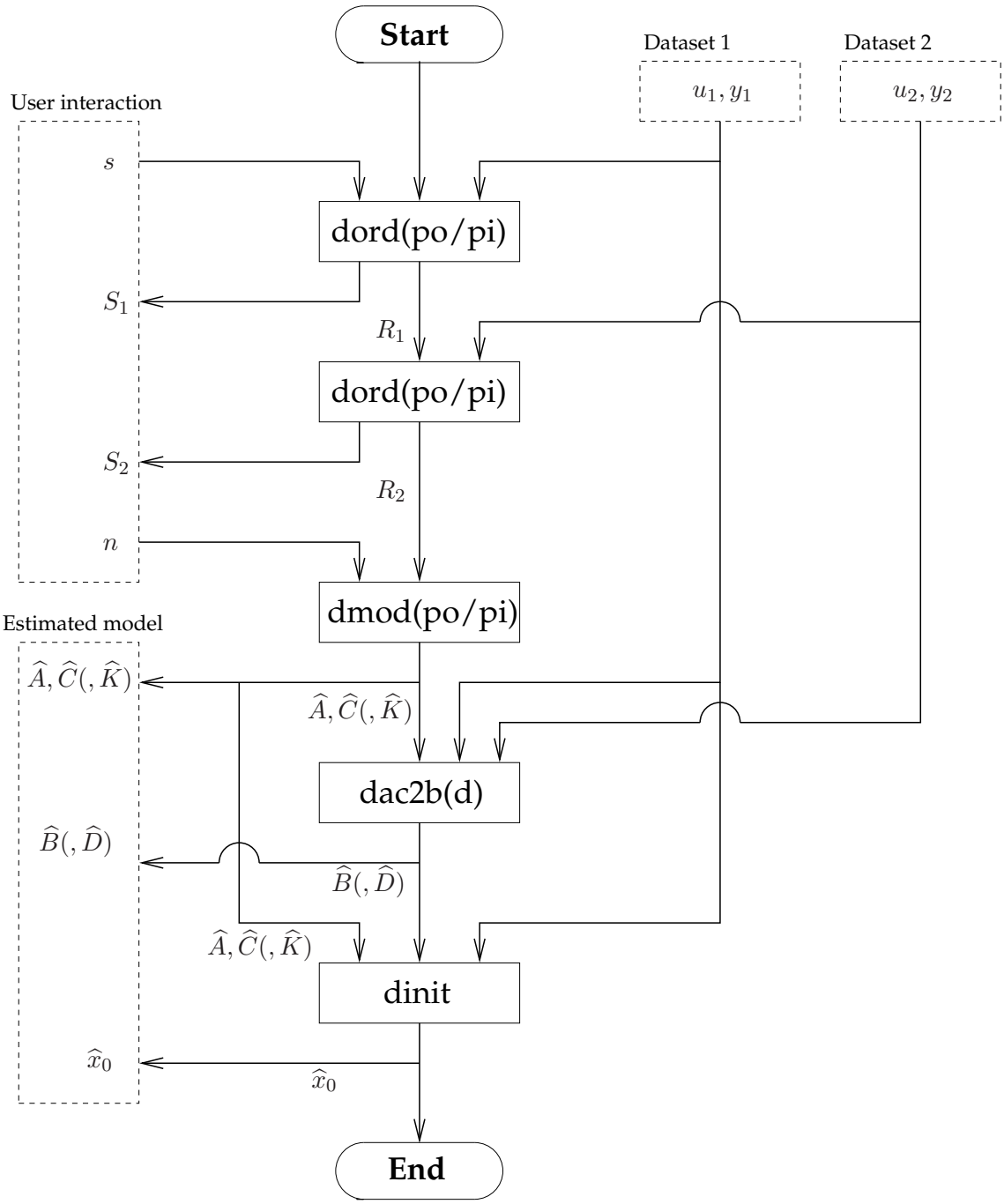


Figure 3.5: Flow-graph for multiple-batch PI-MOESP and PO-MOESP time-domain subspace identification using the toolbox software. This example shows the concatenation of two data batches. Note that a Kalman gain estimate $\widehat{K}$ can be obtained only in the PO-MOESP algorithm; the PI-MOESP algorithm is an output error algorithm that does not provide a Kalman gain estimate.

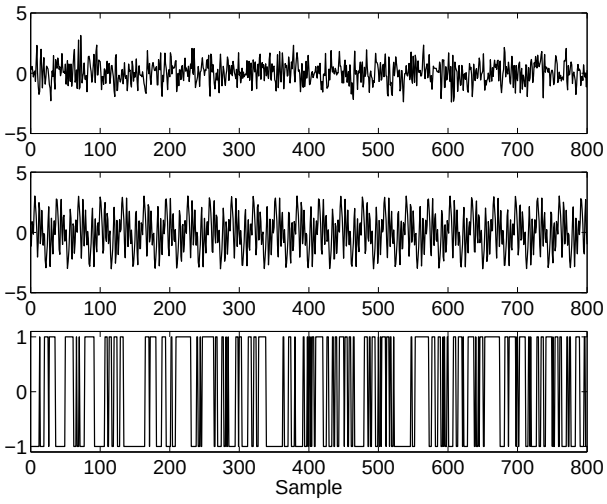


Figure 3.6: Three input sequences for the system.

The output batches corresponding to the four input batches are generated using the system matrices shown above. The system is excited by both the generated data batches $u_i(k)$ and a process noise signal $w_i(k)$ with covariance matrix Q. The output measurement noise signal $v_i(k)$ has covariance matrix R. The input and noise-signals for the first batch are generated using the following MATLAB code:

```
u1 = filter(b,a,randn(N,1));
w1 = randn(N,6)*sqrt(Q);
v1 = randn(N,2)*sqrt(R);
```

The input and output signals for the four data batches can be loaded from the datafile `examples/PO-MOESP-tutorial.mat` on the CD-ROM.
We will now fit a state-space model onto the generated data.

Step 1: Data Compression and Order Estimation

The toolbox function `dordpo` is used to compress the available data and to provide a model order estimate. Because the three input-output batches have been obtained in unconnected experiments, the data cannot just be concatenated and fed to `dordpo`; since the data batches are unconnected, we would introduce two discontinuities in the system's state sequence.

Rather, we use `dordpo`'s capability to take these discontinuities into account. Initially, only the first data batch `u1,y1` is supplied to `dordpo` together with a block-size `s=12`. This yields a singular value vector `S1` and a data matrix `R1`. The second data batch is concatenated by supplying the batch `u2,y2` and `s=12` to `dordpo`, *with* `R1` *as a fourth input argument.* In this way, `dordpo` *does* concatenate the data batches while taking the state discontinuity into account. The same procedure is then followed for the third data batch.

Prior to identification, the signals are again detrended.

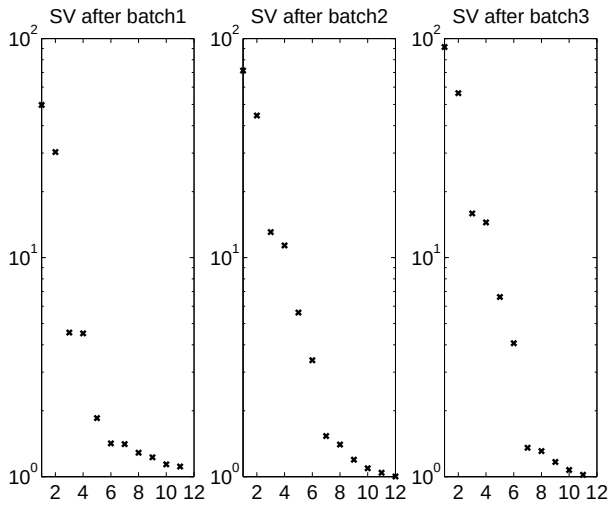


Figure 3.7: Singular values after adding each of the three data batches.

```
>> u1=detrend(u1); y1=detrend(y1);
>> u2=detrend(u2); y2=detrend(y2);
>> u3=detrend(u3); y3=detrend(y3);
>> [S1,R1]=dordpo(u1,y1,12);
>> [S2,R2]=dordpo(u2,y2,12,R1);
>> [S3,R3]=dordpo(u3,y3,12,R2);
```

After each call to `dordpo` we can observe the singular value vector to actually see the increasing amount of information on the system: The singular values after adding each of the three batches are shown in Figure 3.7. After the first batch, the sixth singular value is still buried in noise, but after the second and third batches the gap between dynamics-related and noise-related singular values becomes larger, and it becomes more clear that the model order should be `n=6`.

Step 2: Estimation of A and C

In theory, each of the data matrices `R1`, `R2` and `R3` could be used to estimate the A and C matrices. This provides us with a convenient mechanism, since after adding any data batch we can look at the singular value gap and decide that we have obtained enough information to move on to the estimation of A and C. On the other hand, we may decide to add another batch. In this case we will use the data matrix `R3`, because it contains information on all three data batches and therefore is more informative than `R1` or `R2`: these last two batches contains information on only one and two batches respectively.

```
>> [Ae,Ce]=dmodpo(R3,6);
```

Step 3: Estimation of B and D.

When estimating B and D based on multiple data batches, there exists a concatenation problem like in data compression step: the state is discontinuous in the transition from one batch to another.

Multiple data batches can be concatenated in one single call to the `dac2bd` function as follows:

```
>> [Be,De]=dac2bd(Ae,Ce,u1,y1,u2,y2,u3,y3);
```

This concatenation mechanism is different from the one used in the data compression step in that there are no intermediate results for B and D after adding the first and second batch. However, one should realize that in the data compression step, one does not know beforehand how many data batches will be required, so that one should be able to continue estimating A and C after adding any batch. When estimating B and D, one already knows how many data batches are used, so *one* ($\widehat{B}$,$\widehat{D}$) estimate is calculated based on *all* data batches.

Model Validation

Now that an estimated model `(Ae,Be,Ce,De)` is available, it would seem natural to simulate the output of this model based on the validation input batch, and to compare this prediction to the actual output. However, the system under consideration in this section is only marginally stable. Because of the noise, the poles of the estimated model are not exactly equal to those of the actual system. Because the actual system poles are so close to the unit circle edge, it is in fact non unlikely that the estimated poles are just outside the unit circle because of the disturbances. This would mean that the estimated model is unstable, and simulation would result in an unbounded output signal. Figure 3.8 illustrates how close to the unit circle edge the actual system poles lie.

Simulating an unstable model in order to obtain a predicted output is impossible since the predicted output would become infinite. However, it is possible to use a one step ahead predictor to obtain the model output. In this way, the following *stable* model is simulated:

$$\begin{aligned} \widehat{x}(k+1) &= (\widehat{A}-\widehat{K}\widehat{C})\widehat{x}(k)+(\widehat{B}-\widehat{K}\widehat{D})u(k)+\widehat{K}y(k), && (3.8)\\ y(k) &= \widehat{C}\widehat{x}(k)+\widehat{D}u(k). && (3.9) \end{aligned}$$

Obviously, we need a proper Kalman gain estimate $\widehat{K}$ to simulate this innovation model. This Kalman gain is obtained from `dmodpo` as follows:

```
>> [Ae,Ce,Ke]=dmodpo(R3,6);
```

It should be noted at this point that the `Ae` and `Ce` matrices are equal to those estimated before: these matrices are independent of whether a Kalman gain is requested. The last element of information needed to simulate the predicted output is the initial state of the validation batch. This initial state is obtained using the `dinit` function on the validation batch. Since we will simulate an innovation model, the initial state corresponding to the innovation model is estimated.

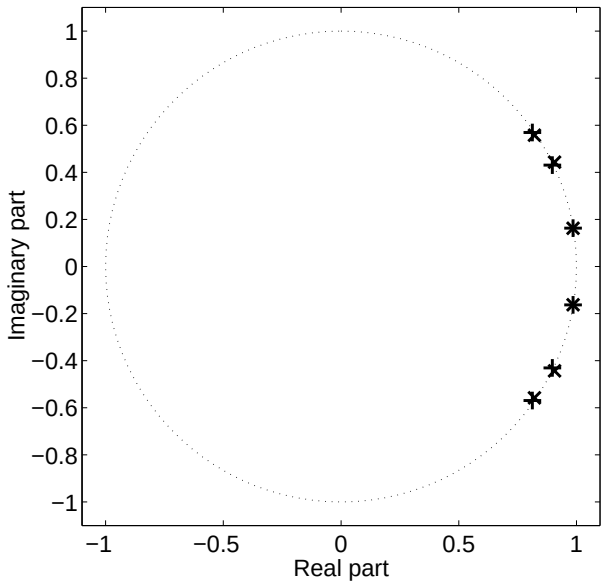


Figure 3.8: Estimated pole locations ($\times$) and true pole locations ($+$).

It should again be stressed at this point that in order to validate the model, we use a fourth validation batch rather than one of the three batches based on which the model was identified. The main reason for using a separate validation batch is that it reduces the chance of over-fitting the model; a concept which is explained in more detail in Section 4.5.3.

```
>> x04e=dinit([Ae-Ke*Ce],[Be-Ke*De],Ce,[De zeros(2,2)],[u4 y4],y4);
>> yek=dltisim(Ae-Ke*Ce,[Be-Ke*De Ke],Ce,[De zeros(2,2)],...
               [u4 y4],x04e);
```

We calculate the VAF between `y4` and `yek` in order to validate the estimated model:

```
>> vaf(y4,yek)
ans =
   66.9799
   50.2648
```

These VAF values are surprisingly low. However, this does not imply that the estimated model is bad; if noise-levels are high compared to the signals, the maximum attainable VAF will be low, even if the model matrices would be exactly equal to the actual system matrices.

Another way the assess the estimated model's quality is by using the residual test in Section 4.5.1. The residual `y4` $-$ `yek` should be white. The output signal $y(k)$ and error signal $\epsilon(k) =$ `y4-yek` are plotted in Figure 3.9. The auto-correlation of the residuals on the first and second output is shown in Figure 3.10. The auto-correlation function more or less equals a pulse for the two residuals. The estimated model is valid.

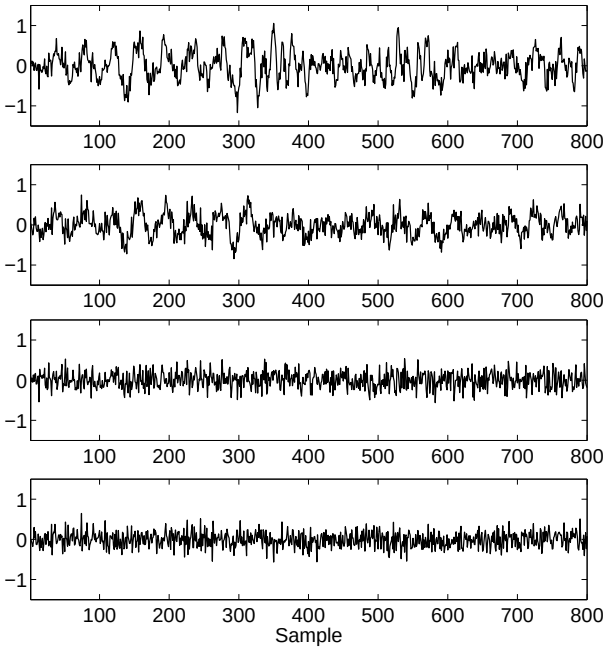


Figure 3.9: The output signals and error signal $\epsilon(k) = y(k) - \widehat{y}(k)$ corresponding to the fourth data batch. The top two signals are the two output signals of the fourth batch. The bottom two signals are the two prediction errors.

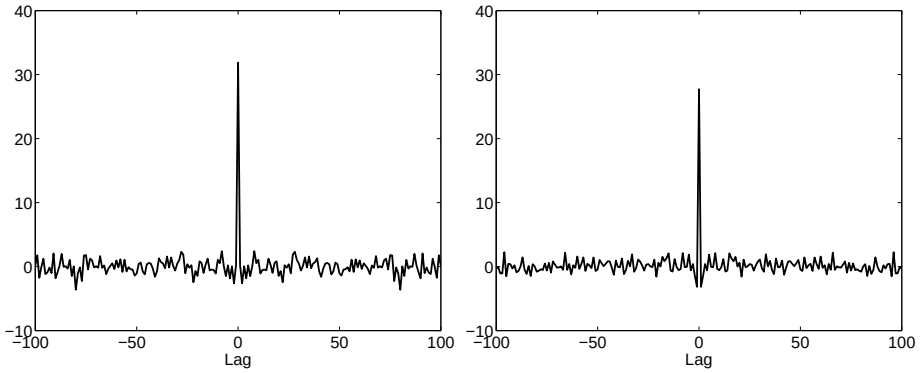


Figure 3.10: The output signals and error signal $\epsilon(k) = y(k) - \widehat{y}(k)$ corresponding to the fourth data batch. The top two signals are the two output signals of the fourth batch. The bottom two signals are the two prediction errors.

3.4 Subspace Identification in the Frequency Domain

The subspace algorithms and implementation discussed in the previous section all dealt with time-domain measurements. The subspace implementation discussed in this section identifies models based on frequency response function (FRF) measurements. Although measuring a system's FRF usually is a more complex operation than measuring time-series, the FRF approach has a number of advantages, as was already discussed in Section 2.4. To be able to use the frequency domain subspace identification methods in the toolbox, when instead of FRF measurements, input and output spectra are available, the FRF needs to be estimated from these spectra first.This can however only be done if the input spectrum matrix is nonsingular at each of the measured frequencies.

It is possible to estimate the FRF from measured input-output data. To this end, the MATLAB Identification Toolbox [5] functions `spa` and `etfe` can be used.

spa
etfe

3.4.1 Identifying Discrete-Time Models

The toolbox software provides a number of MATLAB functions that enable the identification of discrete-time state-space models from FRF data. Figure 3.11 shows how a state-space model is obtained starting from data. When comparing this flowchart to Figure 3.1, it is obvious that the general method is the same. Like in the time-domain, multiple data batches can be concatenated. This method is very similar to that of the time-domain case in Figure 3.5, and it is shown in Figure 3.12.

First there is an "`ord`" function that compresses the data. This function performs a frequency-domain compression of data in an Ordinary MOESP like fashion, hence the name `fdordom` (see manual on page 137). The inputs to `fdordom` are a vector of complex frequencies `w`, the FRF measurements `H` and an upper bound `s` on the model order. The outputs of `fdordom` are a vector of singular values and a compressed data matrix `R`. Like in the time-domain case, multiple data batches can be concatenated using this `R`-matrix. A "`mod`" function is subsequently used to obtain the estimates $\widehat{A}$ and $\widehat{C}$ from the estimate of the extended observability matrix estimate $\widehat{\mathcal{O}}_s$ that is incorporated in the data matrix `R`. The user needs to specify the desired system order `n` as well.

fdordom

Example 3.3 (Estimating A and C from FRF data)

In this example we will identify the second-order model (2.2) on page 12. First, a dataset is generated. As the model is discrete-time, we need to specify complex frequencies on to unit circle. We will specify $N = 512$ frequencies equidistantly spaced on the upper part of the unit circle and calculate the corresponding FRF using the toolbox function `ltifrf` (see manual on page 148). containing the FRF, as shown in Figure 2.3 on page 30. In this case, the array has dimensions $1 \times 1 \times 512$. In practice, H will be measured using a network analyzer. These measurements are assumed to be disturbed by a Gaussian random variable, which is emulated by adding white noise to the calculated FRF.

ltifrf

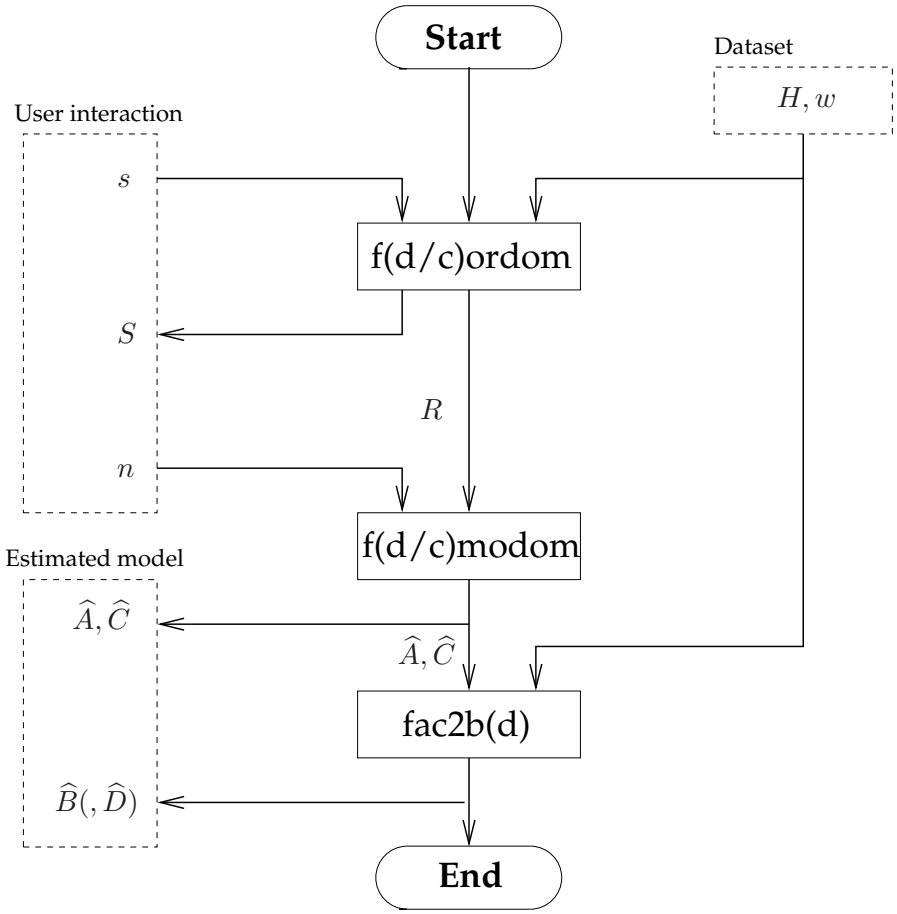


Figure 3.11: Flow-graph for frequency-domain subspace identification of discrete-time and continuous-time models using the toolbox software.

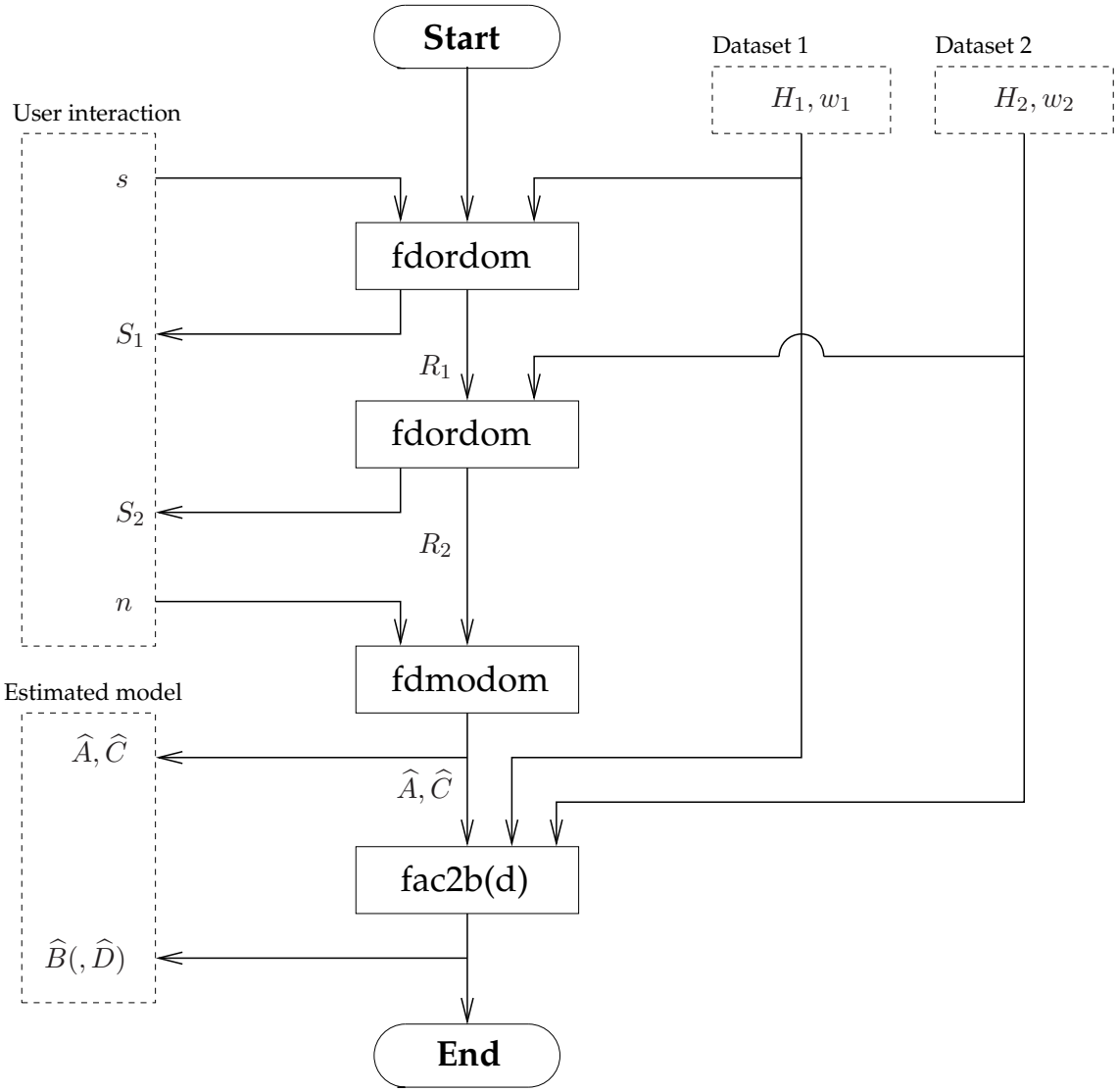


Figure 3.12: Flow-graph for frequency-domain subspace identification of discrete-time models using the toolbox software. This example shows the concatenation of two data batches.

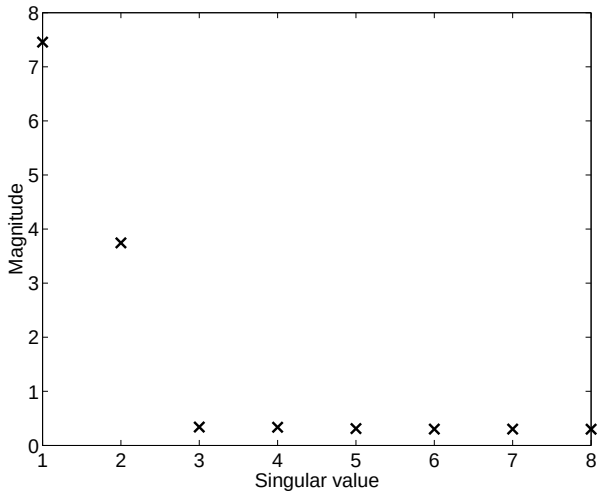


Figure 3.13: Singular values for the frequency-domain subspace identification example.

```
>> w=exp(j*pi*linspace(0,1,512)).';
>> H=ltifrf(A,B,C,D,[],w,[]);
>> H=H+0.3*randn(1,1,512);
```

The frequency-vector and disturbed FRF data can be loaded from the datafile `examples/FDDiscSub.mat` on the CD-ROM. The first step in the identification is the data compression, for which the `fdordom` function is called. As we assume that we don't know the system order in advance, we will choose a rather large upper bound $s = 8$:

```
>> [S,R]=fdordom(H,w,8);
```

A plot of the eight singular values in the vector `S` is shown in Figure 3.13. From the singular value plot it is clear that the system order `n` should be chosen equal to two. The next step is to obtain the estimates $\widehat{A}$ and $\widehat{C}$ using the function `fdmodom` (see manual on page 134):

fdmodom

```
>> [Ae,Ce]=fdmodom(R,2);
```

At this point we would like to assess the quality of the estimated `Ae` and `Ce` matrices. However, we cannot just compare `Ae` and `Ce` to `A` and `C`, since subspace identification estimates the state-space matrices up to an unknown similarity transformation. As the eigenvalues of a state-space model's A matrix are invariant under such a transformation, we can assess `Ae` by comparing its eigenvalues to those of `A`.

```
>> eig(Ae)-eig(A)
ans =
   1.0e-03 *
  -0.8792 + 0.7270i
  -0.8792 - 0.7270i
```

There is a clear difference between the eigenvalues of `Ae` and `A`, which is caused by the noise that was added to the FRF "measurement". However, this difference is small.

After having obtained the $\widehat{A}$ and $\widehat{C}$ estimates, the B and D matrices can be estimated. To this end, the functions `fac2b` (see manual on page 129) and `fac2bd` (see manual on page 131) can be used. The former will assume that D is zero, and estimates only B. The latter estimates both B and D. Like in the time-domain implementation, multiple data batches can be concatenated in estimating B and D.

fac2b
fac2bd

Example 3.4 (Estimating B and D from FRF data)

This example is a continuation of the previous example. Again, the datafile `examples/FDDiscSub.mat` on the CD-ROM can be used. Given the estimates $\widehat{A}$ and $\widehat{C}$, called `Ae` and `Ce` in MATLAB, the function `fac2bd` can be used to obtain estimates $\widehat{B}$ and $\widehat{D}$ as follows:

```
>> [Be,De]=fac2bd(Ae,Ce,H,w)
Be =
   -0.1634
   -0.0849
De =
    0.0102
```

The `De` estimate can be assessed in a straightforward way, since a system's D matrix is invariant under the similarity transformation that is inherent to subspace identification. We know that the actual $D = 0$ and we can thus conclude that the estimated `De` is rather close to its expected value.

In order to assess all the model matrices, we can compare the frequency-responses of the actual system and the estimated model. Figure 3.14 shows the transfer function of the actual system and of the estimated model. It is clear that the differences are very small. The MATLAB commands for plotting this figure are the following:

```
>> Ha=ltifrf(A,B,C,D,[],w,[]);
>> He=ltifrf(Ae,Be,Ce,De,[],w,[]);
>> f=linspace(0,0.5,512);
>> subplot(2,1,1);
>> semilogy(f,abs(Ha(:)),f,abs(He(:)),'--');
>> subplot(2,1,2);
>> plot(f,unwrap(angle(H(:))),f,unwrap(angle(He(:))),'--');
```

First, the FRF for the actual and estimated system matrices is calculated using the `ltifrf` toolbox function. Then a 512-point discrete frequency grid $f \in [0, 0.5]$ is calculated, corresponding to frequencies ranging from 0 to 0.5 times the sampling frequency. The absolute values of the measured and estimated FRF are plotted on this frequency axis using a logarithmic y-axis. Subsequently, the phase is plotted. Note that the `unwrap` function is used to prevent jumps in

unwrap

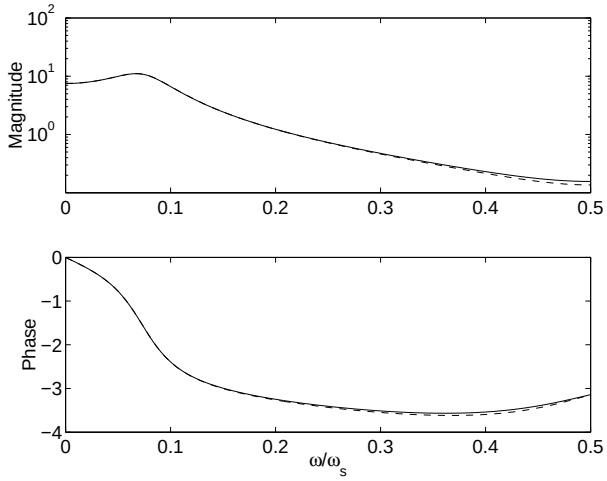


Figure 3.14: Transfer functions of the actual system (*solid*) and the estimated model (*dashed*) resulting from frequency-domain subspace identification.

the phase from $-\pi$ to π. In all cases, the notation `H(:)` is used to convert the 3D FRF array into a vector.

3.4.2 Identifying Continuous-Time Models

In this section we will show how the frequency-domain subspace identification functions in the toolbox can be used to identify continuous-time state-space models. The mechanism is basically the same as in the discrete-time case in Figure 3.11. However, in contrast to the mechanism shown in Figure 3.12, multiple data batches are not supported when identifying continuous-time models. This is not an implementation shortcoming but an algorithm shortcoming; the continuous-time algorithm uses Forsythe-recursions [6] to prevent ill-conditioning in the data-compression stage. These Forsythe-recursions do not allow the concatenation of multiple batches. The second character of the function names for the "`ord`" and "`mod`" function are a "`c`" to indicate that the functions generate data suitable for estimating continuous-time models.

Example 3.5 (Identifying a continuous-time model)

In this example we will identify the continuous-time system (2.16) from [6].

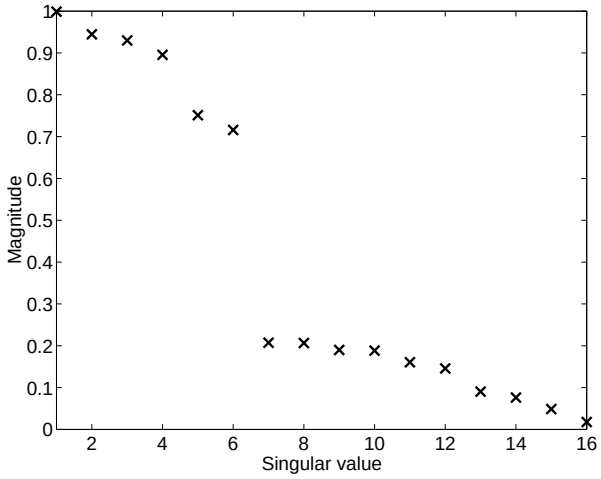


Figure 3.15: Singular values for the continuous-time frequency-domain subspace identification example.

This system is repeated here for convenience:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & -0.2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -25 & -0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -9 & -0.12 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$
$$C = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}, \quad D = 0.$$

The FRF is calculated on an equidistant frequency-grid of $N = 900$ points in the frequency-band $[0.01, 9]$ rad/s. A disturbance is subsequently added to the FRF:

```
>> w=j*(0.01:0.01:9)';
>> H=ltifrf(A,B,C,D,[],w,[]);
>> H=H+0.01*randn(1,1,900);
```

This frequency-vector and FRF can be loaded from the datafile `examples/FDContSub.mat` on the CD-ROM. The first step in identifying this model is the compression of data in order to generate an order-estimate. Since we are dealing with a continuous-time model here, the continuous-time compression function `fcordom` (see manual on page 135) is used. The upper bound `s` on the model order, or block-size, is chosen equal to 16 in this case.

fcordom

```
>> [S,R]=fcordom(H,w,16);
```

The 16 singular values that are returned by `fcordom` are plotted in Figure 3.15. It is obvious that a 6th order model should be identified. The $\widehat{A}$ and $\widehat{C}$ matrices corresponding to a 6th order model can be obtained using the continuous-time "`mod`" function `fcmodom` (see manual on page 133). This func-

fcmodom

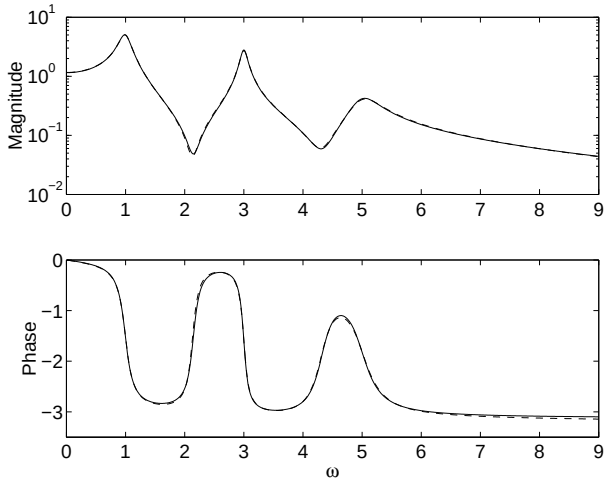


Figure 3.16: Transfer functions of the actual system (*solid*) and the estimated model (*dashed*) resulting from continuous-time frequency-domain subspace identification.

tion needs the data matrix `R` that was obtained using `fcordom` and the desired model order.

```
>> [Ae,Ce]=fcmodom(R,6);
```

The next step is the determination of the $\widehat{B}$ and $\widehat{D}$ estimates. Like for the discrete-time system in Example 3.4, the functions `fac2b` and `fac2bd` can be used to this end.

```
>> [Be,De]=fac2bd(Ae,Ce,H,w)
Be =
    0.0075
   -0.0063
   -0.0043
   -0.0052
    0.0057
    0.0147
De =
    0.0020
```

The `De` matrix can be directly compared with the actual system's D-matrix, since a state-space model's D matrix is invariant under the similarity transformation that is inherent to subspace identification. The matrix `De` should be 0, and is indeed very small.

The actual system and estimated model can be compared by comparing their frequency response functions. These are plotted in Figure 3.16. The magnitude of the difference between the actual and estimated FRF is shown in Figure 3.17. The MATLAB commands for this plot and the previous are similar to those used in the discrete-time case on page 57. When comparing Figures 3.16 and 3.17, it

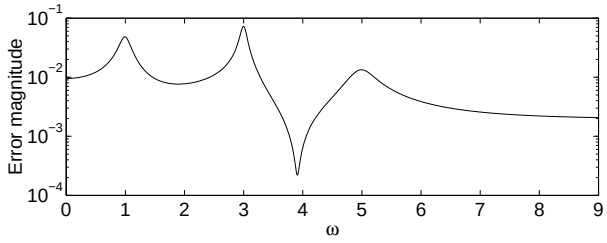


Figure 3.17: Magnitude of the difference between the actual and estimated FRF.

is clear that the magnitude of the error is much smaller than the FRF magnitude itself. We can therefore conclude that a good model has been estimated.

References

[1] M. Verhaegen, "Subspace model identification part 3. Analysis of the ordinary output-error state-space model identification algorithm," *International Journal of Control*, vol. 56, no. 3, pp. 555–586, 1993.

[2] M. Verhaegen, "Identification of the deterministic part of MIMO state space models given in innovations form from input-output data," *Automatica*, vol. 30, no. 1, pp. 61–74, 1994.

[3] H. J. A. F. Tulleken, "Generalized binary noise test-signal concept for improved identification-experiment design," *Automatica*, vol. 26, no. 1, pp. 37–49, 1990.

[4] J. N. Juang, M. Phan, L. G. Horta, and R. W. Longman, "Identification of observer/Kalman filter Markov parameters: Theory and experiments," in *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, (New Orleans, Louisiana), pp. 1195–1206, Aug. 1991.

[5] L. Ljung, *System Identification Toolbox User's Guide*. The MathWorks Inc., Natick, Massachusetts, version 5 (release 12) ed., Nov. 2000.

[6] P. van Overschee and B. De Moor, "Continuous-time frequency domain subspace system identification," *Signal Processing*, vol. 52, no. 2, pp. 179–194, 1996.

Chapter 4

The Identification Cycle

After studying this chapter you can:

- use MATLAB to generate commonly used identification input signals.
- remove trends and spikes from measured signals.
- use prefiltering to improved data for identification.
- estimate and remove delays from measured data.
- select the model structure in identification procedures.
- validate identified models.

4.1 Introduction

This chapter is a companion to Chapter 10 of the textbook. In this chapter we show some of the practical considerations in the identification cycle. This mainly concerns how certain operations are performed using MATLAB and the toolbox software. The textbook itself covers the theoretical and engineering background.

The identification cycle is discussed in the textbook, and is displayed in Figure 4.1 for convenience. The most important message that is conveyed by this figure is that identifying a system is an inherently iterative process. None of the steps —the experiment design, the experiment itself, the choice of model structure and the model fitting itself— should be taken for granted. Rather, the result of each step should be inspected carefully, and the consequences for each of the other steps should be considered.

Section 4.2 contains a number of practical notes to accompany the theoretical background on experiment design covered in Section 10.2 of the textbook. In Section 4.3 the practical points of data preprocessing in MATLAB are covered. Section 4.4 shows how model structures can be selected in a number of identification schemes. Section 4.5 shows how an estimated model can be validated using MATLAB and toolbox functions.

Finally, Section 4.6 shows a case-study example in which all concepts discussed in this chapter are illustrated.

4.2 Experiment Design

In this section the design of an identification experiment will be illustrated. The theory and many guidelines on experiment as described in Section 10.2 of the textbook, and the textbook should be consulted for most of the decisions. In this section we will show how to generate the various input signals discussed in the textbook.

Figure 4.2 contains the identification input sequences that have been discussed in the textbook. On the left hand side, the inputs and their characteristic parameters are shown. On the right hand side, the MATLAB code to generate signals with these characteristics is shown. It is important to note that while the textbook describes the theoretical properties of the various signals, the code that is shown here is very practically oriented. We assume that the user wishes to generate an input sequence of N samples, with the discrete-time vector `t` ranging from 1 up to and including N.

prbn

The signal generation commands that are shown do not require any MATLAB toolboxes to be installed. An exception is the last command: `prbn` (see manual on page 152). It is a function that is part of the toolbox software provided with this book. The frequency sweep can also be generated using the `chirp` function, provided that the Signal Processing Toolbox [1] is installed.

chirp

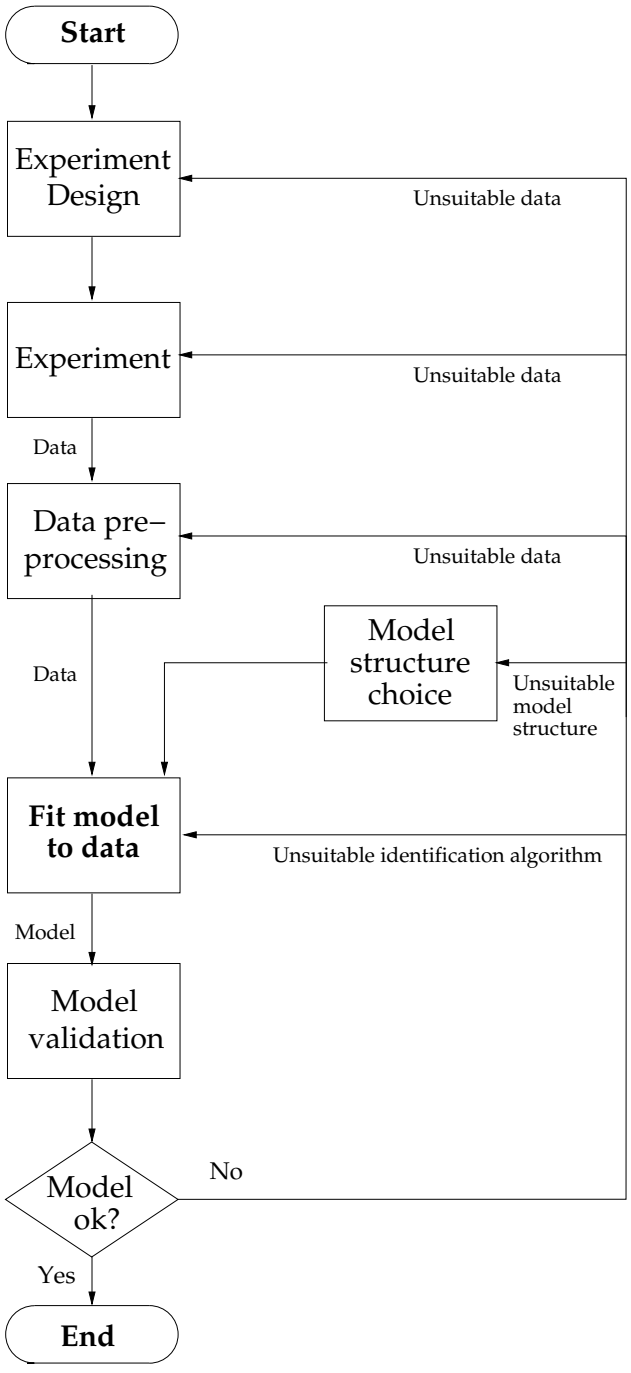
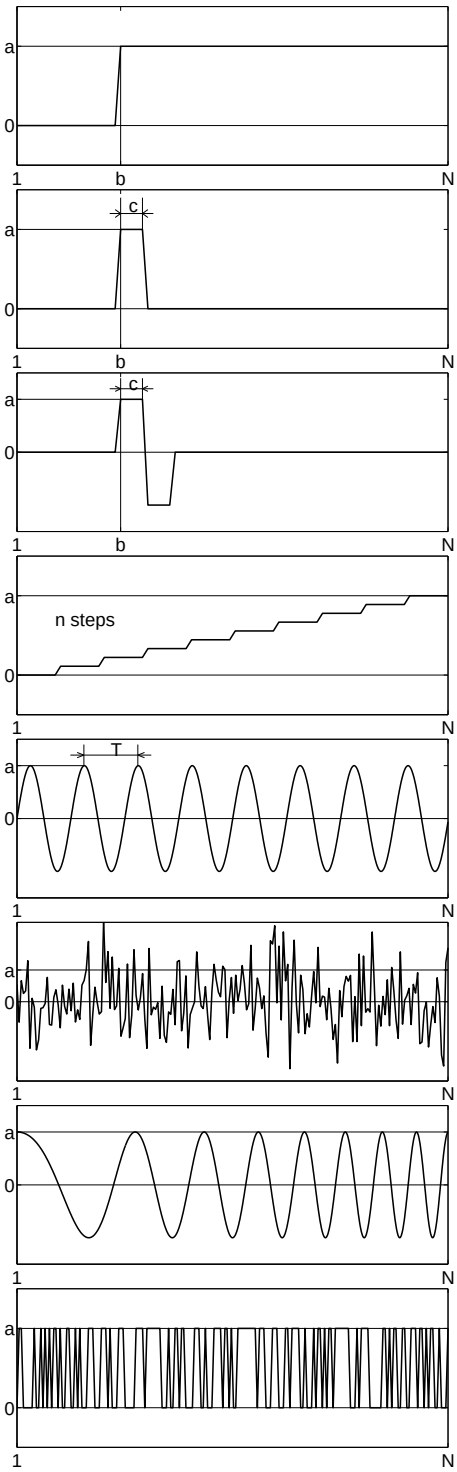


Figure 4.1: The system identification cycle



Step
```
u=a*[zeros(1,b-1),ones(1,N-b+1)]
```

Short pulse
```
u=a*[zeros(1,b-1),ones(1,c),
 zeros(1,N-b-c+1)]
```

Doublet
```
u=a*[zeros(1,b-1),ones(1,c),
 -ones(1,c),
 zeros(1,N-b-2*c+1)]
```

Staircase
```
u=kron(linspace(0,a,n),
 ones(1,N/n))
```

Harmonic
```
t=1:N
u=a*sin(2*pi*(t-1)/T)
```

Gaussian white noise
Standard deviation `a`
```
u=a*randn(N,1)
```

Frequency sweep
Period `T1` to `T2`
```
t=1:N
u=a*sin(2*pi*(t-1)/T1.*(N-t)/N
 +2*pi*(t-1)/T2.*t/N )
```

Pseudo-random binary sequence
Change probability is `rho`
```
u=prbn(N,rho)
```

Figure 4.2: Different types of standard input sequences used in system identification. The characteristic parameters are shown visually in the figures on the left. The code on the right shown how to generate these signals in MATLAB.

4.3 Data Preprocessing

This section will illustrate the issues in data preprocessing that were covered in Section 10.3 of the textbook. Data obtained from an actual system may have been sampled at too high a sampling frequency. In addition, the data may contain unwanted trends or spikes. Data may be prefiltered based on prior knowledge of the disturbance spectra in order to obtain more accurate models. Finally, data from different experiments can be concatenated to increase the amount of available information.

4.3.1 Decimation

Selecting a sampling frequency for measuring data sequences is an important step in identification. In Section 10.2.1 of the textbook we explained that if a system has a bandwidth of interest of ω_B rad/s, a rough engineering rule is to take the sampling frequency $f_s = 10\omega_B/2\pi$ Hz. However, in practice one might not know the bandwidth of interest in advance. In that case, one can first sample a signal at a high sampling frequency, after which one can find the bandwidth of interest by inspecting the signal's spectrum. Once the bandwidth of interest has been found, one can either install an analog anti-aliasing filter and measure the signals at a lower sampling frequency, or one can *digitally* resample the already measured sequence.

When resampling a sampled time-signal at a lower sampling frequency, one has to use a *digital* low-pass filter to prevent aliasing. In addition, such a filter causes delays, and these have to be taken into account as well. The MATLAB Signal Processing Toolbox [1] function `resample` takes care of both issues. If one wishes to resample the signal `x` at one-fifth the original sampling frequency to obtain the signal `y`, the following MATLAB command can be used:

resample

```
>> y=resample(x,1,5);
```

In this command, the numbers 1 and 5 indicate that the signal should be resampled at $1/5$ times the original sampling frequency. Any other pair of integers p and q can be used to resample the signal at p/q times the original sampling frequency.

4.3.2 Detrending and Shaving

As described in Section 10.3.2 of the textbook, a system's behavior can be linearized around a certain operating point, or offset, $(\overline{u}, \overline{y})$. Usually, these offsets are unknown, and there are two ways to deal with their presence. The first method —subtracting estimates of these offsets from the input-output data— will be described in this section. The second method —incorporating the offset into the model as an unknown parameter— will not be treated in the section.

A widely used method to estimate offsets is to calculate the sample mean of the measured sequences:

$$\overline{y} = \frac{1}{N}\sum_{k=1}^{N} y(k), \qquad \overline{u} = \frac{1}{N}\sum_{k=1}^{N} u(k).$$

Although the toolbox software does not include functions for this, MATLAB itself contains a function that detrends data by subtracting the sample mean. The
detrend function `detrend` is convenient since it processes both univariate and multivariable signals. Removing a sample mean $\overline{y}$ from a signal `y` is accomplished as follows:

```
>> yd=detrend(y,'constant');
```

In addition to removing an unknown offset from a signal, `detrend` also allow the removal of linear trends from signals. These may be caused by amplifiers in the measurement setup that suffer from an offset drift. In order to remove linear trends, the following command can be used:

```
>> yd=detrend(y);
```

The removal of unknown offsets and/or linear trends from signals can be seen as a special case of signal prefiltering. A more general case of signal prefiltering for SISO systems is considered in the next section.

However, before continuing it should be noted that actually measured signals often contain spikes and other artifacts due to temporary sensor failures or other unforeseen external influences. As these artifacts do not correspond to the underlying system dynamics, they should be removed before starting the identification. A method to "shave" spikes off signals has been developed in [2], and
shave has been implemented in the toolbox function `shave` (see manual on page 155).

Since the function `shave` correctly handles signals with rather complex trends, it should preferably be called *before* using the `detrend` function. However, in contrast to `detrend`, `shave` only supports univariate signals. This means that in the multivariable case, `shave` should be used on each of the signal components separately.

A full calling syntax specification for `shave` can be found on page 155.

Example 4.1 (Shaving and detrending a signal)

In this example we show the practical use of the functions `shave` and `detrend` in removing artifacts and trends from signals. The starting point of this example is a signal on which a trend and two large spikes have been superposed. This signal can be loaded from the datafile `examples/ShaveData.mat` on the CD-ROM. The function `shave` is used without output arguments, which generates Figure 4.3.

```
>> shave(y,3);
```

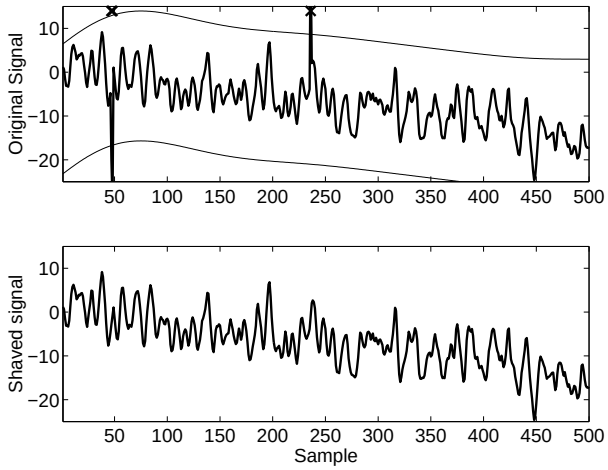


Figure 4.3: The result of removing spikes from a signal using `shave`.

The additional argument "3" specifies the width of the amplitude band in which signals are considered to be actual signals rather than artifacts. The band between the curving lines above and below the signal in Figure 4.3 represents the band of signal-values that are assumed to be "good". Samples outside this band are assumed to be spikes, and these are removed and replaced by a linear interpolation of the neighboring samples. The band of "good" samples follows the trend of the signal. Selecting the bandwidth, "3" in this case, it a matter of trial and error. On the one hand, one might take a very small value. This produces a very smooth signal, but also removes a lot more than just the spike artifacts. On the other hand, a very large value does not destroy the actual signal, but does not remove spikes either. In practice, the bandwidth should be selected such that obvious spikes are removed while most of the signal is retained. From a visual inspection of the result it is clear that there is a strong trend present in the signal. This trend is removed using the function `detrend`, after first having obtained the shaved signal:

```
>> ys=shave(y,3);
>> yd=detrend(ys);
```

The detrended signal `yd` is plotted in Figure 4.4.

4.3.3 Prefiltering the Data

In addition to removing spikes and trends from signals, signals related to SISO systems can be prefiltered in order to reduce the influence of noise in certain areas of the spectrum. As explained in Section 10.3.3 of the textbook, prefiltering can be used for two reasons. First, if the disturbance's frequency-band is known, the filtering may be specified such that this frequency-band does not have a large influence in the identification algorithm's cost function. Second, the

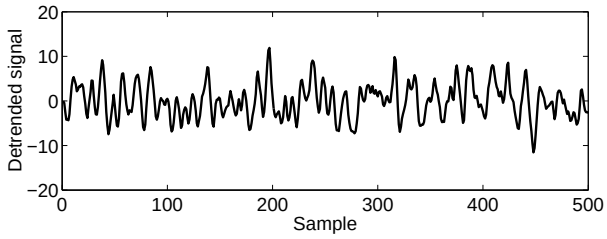


Figure 4.4: The result of removing a linear trend from the shaved signal.

filtering can be chosen such that the mismatch between the system's and model's transfer function is minimized in certain important areas of the spectrum, while the mismatch may be larger in other areas.

Example 4.2 (Prefiltering)

In this example we use data from the system (2.2) on page 12 to illustrate the effect of prefiltering data batches. The input to the system is a $N = 4096$ sample unit-variance Gaussian white-noise signal. The output $y(k)$ of the system consist of a deterministic part simulated by `dltisim` (see manual on page 108), and a stochastic part which is generated by feeding a unit-variance Gaussian white-noise signal through a fifth-order Butterworth high-pass filter with a cut-off at 0.25 times the sampling frequency. The signals are generated using the following MATLAB commands. Note that the function `butter` from the MATLAB Signal Processing Toolbox [1] is used. The function `filter` is part of the standard MATLAB installation.

dltisim

butter

filter

```
>> u=randn(4096,1);
>> ynf=dltisim(A,B,C,D,u);
>> [bn,an]=butter(5,2*0.25,'high',randn(4096,1));
>> v=filter(bn,an,e);
>> y=ynf+v;
```

The above signals can be loaded from the datafile `examples/PrefilterData.mat` on the CD-ROM. Singular values for the data are obtained using the PI-MOESP subspace identification algorithm discussed in the previous chapter. These singular values are plotted in the top halve of Figure 4.5. The required model order is unclear.

However, in this example the useful signal and the disturbance can be separated because the signal has signal power mainly in the low frequencies while the noise contains mostly high frequencies. This can be seen by plotting the spectrum of the signal as follows:

```
>> spectrum(y,[],[],[],[],1);
```

The spectrum has been plotted in Figure 4.6. The high-frequency content is mainly due to noise. A low-pass filter will thus remove most of the noise while retaining most of the actual signal. Therefore, the input and output signal

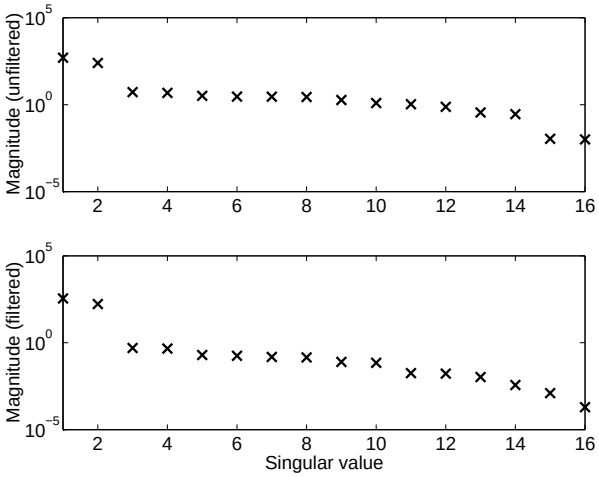


Figure 4.5: The effect of prefiltering on the estimated model order.

are fed through a third-order Butterworth low-pass filter with a cut-off at 0.15 times the sampling frequency. A low-pass filter is chosen because we know that the disturbance resides mainly in the higher frequency-region. A low-pass filter therefore removes much of this disturbance. On the other hand, we should not choose the cut-off frequency of the prefilter too low, since in that case information corresponding to the system's dynamics is lost. The system (2.2) has poles in $z = 0.8367e^{\pm 0.1462j\pi}$. This implies that its peak magnitude response lies at $0.1462/2$ times the sampling frequency. A low-pass filter with a cut-off frequency at 0.15 times the sampling frequency therefore is unlikely to destroy information corresponding to the system's dynamics. The signals are prefiltered using the following MATLAB commands:

```
>> [bf,af]=butter(3,2*0.15);
>> uf=filter(bf,af,u);
>> yf=filter(bf,af,y);
```

The singular values for the filtered data are also obtained using PI-MOESP, and these are plotted in the lower halve of Figure 4.5. It is clear that the model order should be chosen equal to $n = 2$.

Subsequently, the $\widehat{A}$ matrix estimates are calculated based on both the unfiltered and the filtered data. These eigenvalues are compared with the true system's eigenvalues. The results as obtained in MATLAB are:

```
unfiltered_diff =
   1.0e-03 *
   0.3223 - 0.2717i
   0.3223 + 0.2717i
filtered_diff =
   1.0e-04 *
   0.1509 + 0.2409i
   0.1509 - 0.2409i
```

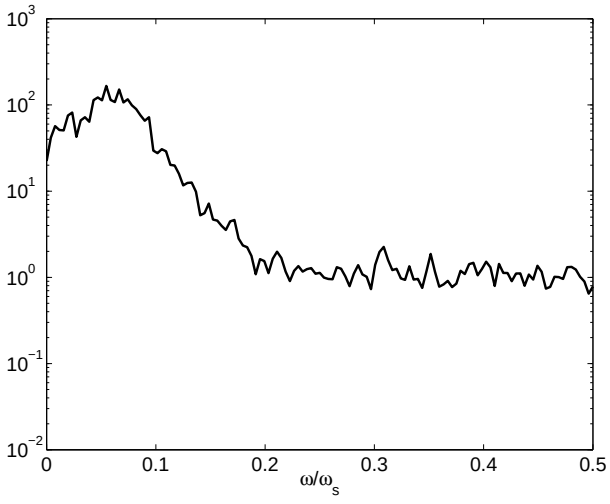


Figure 4.6: The power spectrum of the disturbed output signal.

It is clear that prefiltering the data has made the eigenvalue estimates substantially more accurate.

4.3.4 Concatenation of Data Batches

As stated in the textbook, it might be important to concatenate data batches collected from several experiments. In this way, all available information extracted from the system is used. However, as these data batches are usually unconnected, the initial state of the previous batch does not correspond to the final state of the current batch, which makes pasting data batches together a nontrivial job.

dordpi **dordpo** **dordrs** **dac2b** **dac2bd**

All time-domain subspace data compression functions —`dordpi` (see manual on page 117),`dordpo` (see manual on page 119) and `dordrs` (see manual on page 122)— and B/D estimation function —`dac2b` (see manual on page 98) and `dac2bd` (see manual on page 100)— contain mechanisms that correctly take this state-discontinuity into account [3, Chap. 2 and Sec. 5.4]. Section 3.3.3 contains an example on how to concatenate batches for these algorithms.

fdordom **fac2b** **fac2bd** **fcordom**

In frequency-domain identification, data batches can be more easily concatenated, since FRFs at different frequencies can be calculated independently from one another. However, proper care has to be taken in internal weighting routines. The discrete-time model frequency-domain data compression function `fdordom` (see manual on page 137) correctly takes these issues into account. The B/D estimation functions `fac2b` (see manual on page 129) and `fac2bd` (see manual on page 131) also take proper care when concatenating batches. However, when estimating continuous-time models using the functions `fcordom` (see manual on page 135) and `fac2b`/`fac2bd`, concatenating batches is possible neither in the data compression routines nor in the B/D estimation functions.

The state-space optimization framework does not support the concatenation of unrelated data batches in the time-domain function `doptlti` (see manual on page 113). In the frequency-domain, the data batches can be concatenated before providing them as measurement data to the optimization function `foptlti` (see manual on page 141).

doptlti

foptlti

4.4 Model Structure Selection

An important decision in the system identification cycle is the determination of a model structure. Structure in this respect concerns both delays ("dead time") and the description of the model's dynamic behavior. We will first show how delay-estimation can be carried out. Then, the selection of dynamics structure in both the identification of ARMAX models and the subspace identification of state-space models is illustrated.

4.4.1 Delay Estimation

The delay in a SISO system is obtained from the data by estimating the system's impulse response. We will show that this is a linear regression problem which can be easily solved within MATLAB. The system's impulse response will be modeled as a finite impulse response (FIR). Although a state-space model generally has in infinite impulse response (IIR), this IIR can be approximated by a FIR by taking the order m sufficiently large. An mth order FIR model is given by the following equation:

$$y(k) = h_0 u(k) + h_1 u(k-1) + \cdots + h_m u(k-m). \tag{4.1}$$

This one relation between the output signal and input signal does not yield sufficient information to estimate the h_i. However, we can stack the above equation at different time-instances such that as available data is used. This yields the following Toeplitz-system from which the h_i can be solved:

$$\begin{bmatrix} u(m+1) & u(m) & \cdots & u(1) \\ u(m+2) & u(m+1) & \cdots & u(2) \\ \vdots & \vdots & \ddots & \vdots \\ u(N) & u(N-1) & \cdots & u(N-m) \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ h_m \end{bmatrix} = \begin{bmatrix} y(m+1) \\ y(m+2) \\ \vdots \\ y(N) \end{bmatrix}.$$

In a MATLAB program, the $m+1$ coefficients h_i of an mth-order FIR can be estimated as follows:

```
>> h=toeplitz(u(m+1:N),u(m+1:-1:1)) \ y(m+1:N);
```

Given the h_i, we can visually inspect at which sample instant the impulse response starts to deviate from zero significantly. The earlier samples, which are more or less equal to zero, are caused by the delay present in the system. If the delay is denoted d, it can be removed from the measured signals as follows:

```
>> d=2;
>> u=u(1:N-d);
>> y=y(d+1:N);
```

If this code is used in a MATLAB script, one should take into account that all signals will have only $N - d$ samples after removing the delay.

4.4.2 Structure Selection in ARMAX Models

After removing a system's delay from the measured input-output sequences, its structure can be estimated. In this section we will illustrate Example 10.11 on page 330 of the textbook. We will mainly show which MATLAB commands can be used to reproduce the results of this textbook example. Note that in order to use the commands of this section in MATLAB, the System Identification Toolbox [4] and the Control Systems Toolbox [5] are required.

First, the actual data generating ARMAX model is defined, along with the 1000-sample dataset as described in the textbook. The numerator and denominator of the deterministic system will be called `sysnum` and `sysden` respectively. The numerator and denominator of the stochastic system will be called `noinum` and `noiden` respectively. The input to both is a $N = 1000$ sample long unit-variance Gaussian white-noise sequence.

```
>> sysnum=[1];
>> sysden=[1 -0.95];
>> noinum=[1 -0.8];
>> noiden=[1 -0.95];
>> u=randn(1000,1);
>> e=randn(1000,1);
>> y=dlsim(sysnum,sysden,u)+dlsim(noinum,noiden,e);
```

The resulting input and output data sequences can be loaded from the datafile `examples/ARMAX-example.mat` on the CD-ROM. We will approximate this second-order ARMAX model with a 21th-order ARX model. For this to be possible, the $C(q)$ polynomial —called `noinum` in the MATLAB code— must be Hurwitz. Whether its roots indeed lie within the unit circle can be checked using the `roots` function:

roots

```
>> roots(noinum)
ans =
    0.8000
```

We conclude that $C(q)$ is indeed Hurwitz, and proceed to estimating a 21th-order ARX model. For this step, the Identification Toolbox function `arx` will be used. The function `iddata` is used to generate a data structure suitable for the `arx` function. The vector `[21 21 0]` denotes that 21th-order ARX polynomials $A(q)$ and $B(q)$ should be estimated, and that the delay should be 0.

arx
iddata

```
>> ARXmodel=arx(iddata(y,u),[21 21 0]);
```

The poles and zeros of this model can be plotted using the `pzmap` command, which yields a figure similar to Figure 10.10 on page 332 of the textbook. The structure selection now proceeds by visual inspection of the poles and zeros; poles and zeros that practically cancel each other are assumed to be caused by the overparameterization in the high-order model. Only poles and zeros that do not overlap are assumed to correspond to actual system dynamics. Figure 10.10 on page 332 of the textbook shows that there is only one pole which is not canceled by a zero, and thus the system order is taken equal to 1. The noncanceling pole provides an accurate estimate of the actual system pole in $z = 0.95$, which can be verified by looking at the real roots of the $A(q)$-polynomial:

pzmap

```
>> TheRoots=roots(ARXmodel.A);
>> TheRoots(imag(TheRoots)==0)
ans =
    0.9495
```

The actual ARMAX model can be identified using the `armax` function in the Identification Toolbox. We will specify first-order polynomials for $A(q)$, $B(q)$ and $C(q)$, while the delay is assumed to be zero. This yields the options-vector $[1, 1, 1, 0]$.

armax

```
>> ARMAXmodel=armax(iddata(y,u),[1 1 1 0])
Discrete-time IDPOLY model: A(q)y(t) = B(q)u(t) + C(q)e(t)
A(q) = 1 - 0.9655 qˆ-1
B(q) = 0.7987
C(q) = 1 - 0.8713 qˆ-1
Estimated using ARMAX
Loss function 1.96054 and FPE 1.97238
Sampling interval: 1
```

The actual system polynomials are $A(q) = 1 - 0.95q^{-1}$, $B(q) = 1$ and $C(q) = 1 - 0.8q^{-1}$. The identification has thus yielded a moderately accurate model.

4.4.3 Structure Selection in Subspace Identification

Whereas structure selection for ARMAX models is a rather complex operation, it is much simpler for subspace identification. Like in the ARMAX case, a delay first needs to be eliminated from the signals. Subsequently, the only parameter that needs to be supplied to the first data compression "`ord`" function is the block size `s`. This block-size limits the model order to a maximum of $s - 1$. The block size `s` thus needs to be chosen larger than the expected model order.

The singular values from the data compression step can then be used to determine the required model order. Examples of this procedure can be found in Section 3.3.2 and in Example 9.4 on page 270 of the textbook. A more involved procedure can be found in the case-study in Section 4.6.

4.5 Model Validation

In this section, the model validation methods introduced in Section 10.5 of the textbook will be illustrated. First the MATLAB commands needed to perform the auto-correlation and cross-correlation tests will be given. Then, the cross-validation test will be described. Finally, the MATLAB commands needed to calculate the VAF for will be explained.

4.5.1 Auto-correlation Test

The auto-correlation test, as well as the cross-correlation test in the next section, fall within the class of residual tests. These tests are applicable to innovation models (5.55)–(5.56) on page 138 of the textbook, in which an optimal model means that the variance of the one-step ahead predictor error $\widehat{\epsilon}(k,\theta)$ is minimal. This error is defined as:

$$\widehat{\epsilon}(k,\theta) = y(k) - \widehat{y}(k,\theta). \tag{4.2}$$

In the textbook we explained that if the system under consideration falls within the model class defined by a certain innovation model, the properties of $\widehat{\epsilon}(k,\theta)$ for θ equal to the global optimum calculated using data from an *open-loop* experiment should be the following:

1. The sequence $\widehat{\epsilon}(k,\theta)$ is a zero-mean white-noise sequence.
2. The sequence $\widehat{\epsilon}(k,\theta)$ is independent of the input sequence $u(k)$.

The vector $\widehat{\epsilon}(k,\theta)$ is defined as the difference between the actually measured output $y(k)$ and the innovation model output $\widehat{y}(k,\theta)$ from (2.6)–(2.7). This model can be simulated using the toolbox function `dltisim`.

Example 4.3 (Obtaining a residual vector)

Given an arbitrary innovation state-space model defined by the matrices (A,B,C,D,K), a residual vector $\epsilon(k)$ can be obtained using the toolbox function `dltisim` to simulate an innovation model:

```
>> epsilon=dltisim(A-K*C,B-K*D,C,[D zeros(l,l)],[u y]);
```

For MIMO systems, the residuals $\widehat{\epsilon}_1(k,\theta)$ to $\widehat{\epsilon}_\ell(k,\theta)$ for each of the outputs are stored in the columns of the matrix `epsilon`.

Given the residual vector $\widehat{\epsilon}(k,\theta)$ for each of the outputs, the whiteness of the residual can be checked by calculating its auto-correlation function. This is done using the MATLAB Signal Processing Toolbox function `xcorr`. The function `xcorr` calculates the cross-correlation between two signals, and the cross-correlation of $\widehat{\epsilon}(k,\theta)$ and $\widehat{\epsilon}(k,\theta)$ equals an auto-correlation. The auto-correlation is a signal of length $2N+1$, corresponding to the auto-correlation from lag $-N$ up to and including lag $+N$. However, in practice one assesses a signal's whiteness

xcorr

based on a smaller lag-interval of say $-d$ to $+d$. The function `xcorr` supports the calculation of limited lag-intervals by specifying d as a third input argument. The auto-correlation vector `ac` can therefore be calculated as follows:

```
>> ac=xcorr(epsilon,epsilon,d);
```

The auto-correlation should look like a pulse signal, as shown in the lower left plot of Figure 10.14 on page 339 of the textbook. A plot can be made using the following MATLAB commands:

```
>> plot(-d:d,ac);
```

For a model having more than one output, the auto-correlation function can be calculated analogously for each of the vectors `epsilon1` to `epsilonl`.

4.5.2 Cross-Correlation Test

Like the auto-correlation test, the cross-correlation test is a residual test that can be used on innovation state-space models. In fact, the cross-correlation test checks the second property of the residual specified in the previous section: the residual should be independent of the input signal $u(k)$. The same residual vector `epsilon` as calculated in the previous section can be reused here. Again, the `xcorr` function is used to calculate the cross-correlation between the residual and the input signal as follows:

```
>> xc=xcorr(epsilon,u,d);
```

This command yields the cross-correlation `xc` from lag $-d$ to up and including lag $+d$. The auto-correlation should be very small, as shown in the lower right plot of Figure 10.14 on page 339 of the textbook. A plot can be made using the following MATLAB commands:

```
>> plot(-d:d,xc);
```

For MIMO systems, this situation is more involved, since the residual for each of the outputs should be uncorrelated with each of the inputs. This involves calculating the cross-correlation for each of the ℓm input-output combinations.

4.5.3 Cross-Validation Test

The residual tests discussed so far, as well the the variance accounted for discussed in the next section, do not make sure the model adequately describes the system dynamics. This is a consequence of overfitting. In case of overfitting the model complexity or the number of model parameters has become so large with respect to the length of the data sequence that the predicted output very accurately matches the the identification data. On a different data set, such an overfitted model may not predict the output well. This allows the detection of overfitting.

Therefore, a measured data batch is usually split into an identification part and a validation part. The identification is then performed on the first part (say the first $2/3$ of all samples), after which the tests described in this section can be reevaluated on the second part (the remaining $1/3$ of all samples) of the measured data. In MATLAB code, this is done as follows, assuming that N samples of input-output data `u/y` are available.

```
>> u_id=u(1:floor(2*N/3),:);
>> y_id=y(1:floor(2*N/3),:);
```

A model should now be identified based on `u_id` and `y_id`.

```
>> u_val=u(floor(2*N/3)+1:N,:);
>> y_val=y(floor(2*N/3)+1:N,:);
```

The output should now be predicted with `u_val` as input signal. How this prediction is carried out depends on which model type has been identified. Assuming that a state-space model (A, B, C, D) has been identified, the estimated validation output `ye_val` is calculated as follows:

```
>> ye_val=dltisim(A,B,C,D,u_val);
```

Having obtained a prediction for the validation output, the tests in this section should be performed again.

4.5.4 Variance Accounted For

The variance accounted for (VAF) is a scaled variant of the cost function $J_N(\theta)$ that is throughout both the textbook and this companion book. It is defined as

$$\mathrm{VAF}\left(y(k), \widehat{y}(k,\theta)\right) = \left(1 - \frac{\frac{1}{N}\sum_{k=1}^{N} \|y(k) - \widehat{y}(k,\theta)\|_2^2}{\frac{1}{N}\sum_{k=1}^{N} \|y(k)\|_2^2}\right) 100\%. \tag{4.3}$$

In practice, the VAF is calculated for each output separately. For an output signal $y(k)$ having ℓ components, the VAF for the ith output is given by

$$\mathrm{VAF}_i\left(y_i(k), \widehat{y}_i(k,\theta)\right) = \left(1 - \frac{\frac{1}{N}\sum_{k=1}^{N} |\, y_i(k) - \widehat{y}_i(k,\theta)\,|^2}{\frac{1}{N}\sum_{k=1}^{N} |\, y_i(k)\,|^2}\right) 100\%. \tag{4.4}$$

vaf

VAFs for all components of an output signal can be conveniently calculated using the toolbox function `vaf` (see manual on page 157). This function requires the measured output $y(k)$ and the predicted output $\widehat{y}(k)$, which are called `y` and `ye` in the following MATLAB code:

```
>> vaf(y,ye)
```

The case-study in the next section shows a practical example of using the VAF.

4.6 Case Study: Identifying an Acoustical Duct

In this section we show how the subspace identification framework from Chapter 3 can be used together with the parametric model estimation framework from Chapter 2 to yield a very accurate model. In essence, subspace identification will be used to obtain an accurate initial model estimate, which is then refined using parametric optimization.

The system under consideration will be the 19th-order ARX model in Table 8.1 on page 244 of the textbook. This model describes an acoustical duct. The goal is to fit a relatively low-order state-space model to data of this "system". The input and output data for the case-study in this section can be loaded from the datafile `examples/CaseStudy.mat` on the CD-ROM.

4.6.1 Experiment Design

Since the system model already is a discrete-time model, no sampling frequency needs to be chosen. Since we will be conducting the experiment under white output measurement noise conditions, and the model is rather complex, we will take $N = 10000$ samples. As we need a persistently exciting input signal, we will choose a unit-variance Gaussian white-noise signal.

4.6.2 The Experiment

The "experiment", between quotes because a model is actually simulated rather than measuring data from a physical system, will be described in this section. A white measurement noise is added to the system's output such that a SNR of 20 dB is attained. The following MATLAB code describes the "experiment". Note that the Control Systems Toolbox function `dlsim` is used to simulate the ARX model. The variables `num` and `den` are assumed to contain the 19th-order model's numerator and denominator polynomials. The vectors `num` and `den` are equal to the second and first column of Table 8.1 on page 244 of the textbook respectively.

dlsim

```
>> N=10000;
>> u=randn(N,1);
>> y=dlsim(num,den,u)+1e-1*std(y)*randn(size(y));
```

4.6.3 Data Preprocessing

In this section we preprocess the data. Decimation is unnecessary in this case since we have the luxury of knowing beforehand that the 19th-order model is

sampled properly. In practice, of course, one does not have this luxury! Subsequently, the data should be detrended. Although the data would normally have to be shaved, shaving is not necessary in this case: we obtained the data through simulation, so no spikes due to sensor failures or unforeseen external influences are present. Detrending the data is done as follows:

```
>> u=detrend(u);
>> y=detrend(y);
```

4.6.4 Model Structure Selection

The system under consideration is an acoustical duct. The fact that sound travels through this duct at a finite speed, implies that we can expect a delay to be present in the system.

The delay is obtained from the data by estimating the system's impulse response. To this end, the theory and MATLAB commands from Section 4.4.1 are used to estimate a 250th-order FIR model.

The resulting impulse response estimate $\widehat{h}_k$ is shown in Figure 4.7. If we assume that there is no direct feed-through, then the delay d should be taken equal to 2. This assumption corresponds to the D matrix of the system being zero, which for discrete-time systems often is a valid assumption.

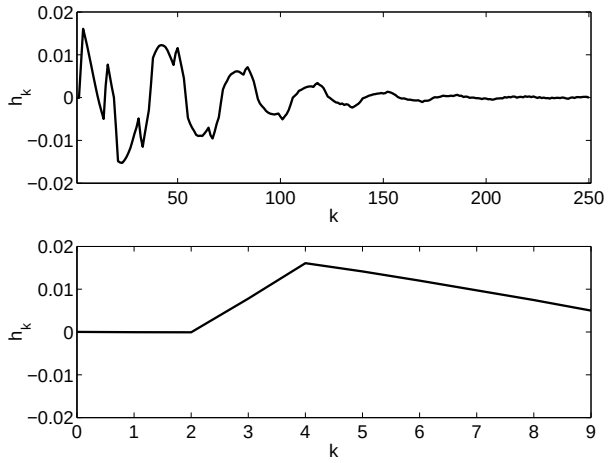


Figure 4.7: The estimated impulse response weights $\widehat{h}_k$ for the acoustical duct. All weights (*top-halve*) and the first few weights (*bottom-halve*).

Rather than estimating an unnecessarily complex state-space model, in which the delays are modeled as states, we will shift the output sequence $y(k)$ two samples back in time, so that a more simple state-space model can be estimated. The MATLAB code from Section 4.4.1 is used with $d = 2$ to remove the delay.

Now that the delay has been removed from the signals, the second part of the model structure should be selected: its order. In subspace identification the only selection that has to be made prior to starting the identification is the selection of

a block-size s which should be two or three times larger than the expected model order. Therefore, in this case we will choose $s = 60$.

4.6.5 Fitting the Model to the Data

With the preliminary steps completed, the actual identification can take place. We will use PO-MOESP to identify the model using the `dordpo`-function:

```
>> [S,R]=dordpo(u,y,60);
```

The singular values in the vector `S` have been plotted in Figure 4.8. Looking purely at the "gaps" in the singular values, one could choose a model order of 2, 5, 8, 10, 12 or 16.

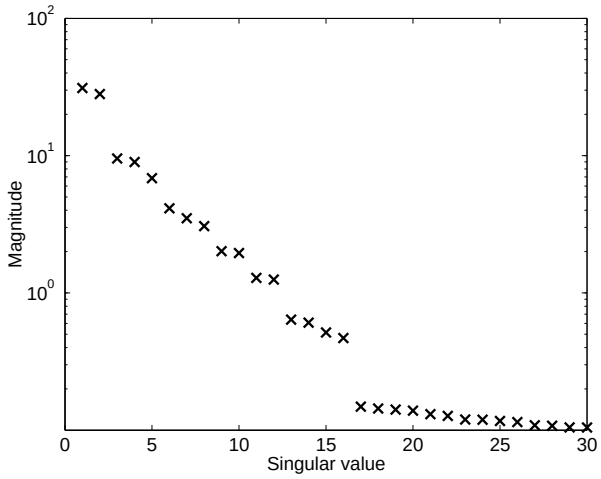


Figure 4.8: The singular values for the acoustical duct data obtained using PO-MOESP.

From the singular value plot alone it is not immediately obvious which of the orders should be chosen. Since the final goal of the identification is to produce a model that predicts the measured output best, we wish to identify a model that produces a good VAF, while not being overly complex. In order to determine which model order we should choose, we identify a model for each system order $n = 1$ to $n = 16$. Note that the *same* `R` matrix from `dordpo` can be used to obtain the pair (A, C) of *all* these models. For a given model order `n`, the corresponding model and VAF are calculated using the toolbox functions `dmodpo`, `dac2bd` and `vaf`:

```
>> [Ai,Ci]=dmodpo(R,n);
>> [Bi,Di]=dac2bd(Ai,Ci,u,y);
>> yi=dltisim(Ai,Bi,Ci,Di,u);
>> vaf(y,yi)
```

The VAFs for the different model orders have been plotted in Figure 4.9. The lowest possible model order for which a good VAF (that is, $> 90\%$) is obtained apparently is $n = 6$.

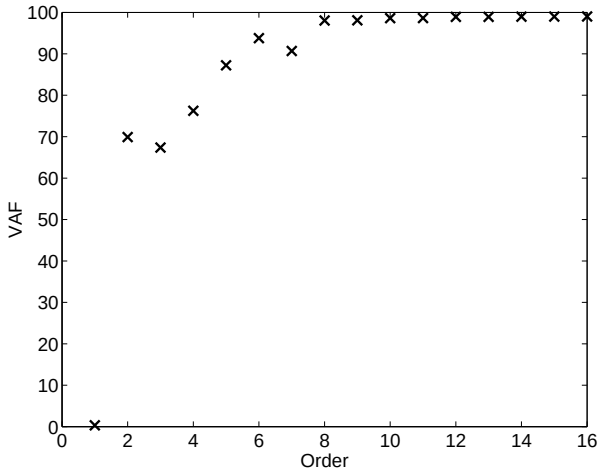


Figure 4.9: The variance accounted for at different model order estimated using PO-MOESP.

The obtained model of order $n = 6$ will be refined using the optimization framework function `doptlti`:

```
>> [Ao,Bo,Co,Do]=doptlti(u,y,Ai,Bi,Ci,Di);
>> yo=dltisim(Ao,Bo,Co,Do,u);
```

4.6.6 Model Validation

In the preceding sections we have estimated two models: an initial model (A_i, B_i, C_i, D_i) and an optimized model (A_o, B_o, C_o, D_o). We now wish to validate these models using the techniques described in Section 10.5 of the textbook. In the cross-validation and auto-correlation approaches, the model is accepted as a good model if the residual is white and uncorrelated with the input signal. However, these approaches do assume that the estimated model is complex enough to describe all relevant system dynamics. In this case, we deliberately estimated a rather low-order model. Therefore, we can expect the residual to be nonwhite even though the low-order model is optimal. The auto-correlation and cross-correlation functions have been calculated using the MATLAB function `xcorr` using the following MATLAB code:

```
>> maxlag=1000;
>> aci=xcorr(y-yi,y-yi,maxlag);
>> aco=xcorr(y-yo,y-yo,maxlag);
>> xci=xcorr(y-yi,u,maxlag);
>> xco=xcorr(y-yo,u,maxlag);
```

Figure 4.10 shows the resulting functions. As expected, the auto-correlation functions are nonwhite due to the unmodeled dynamics. For the same reason, the cross-correlation functions are nonzero. However, there is more that can be said about these functions. Figure 4.11 shows the *spectrum* of the prediction

error. From this figure we can conclude that the residual of the optimized model is considerable "whiter" in the low frequency range.

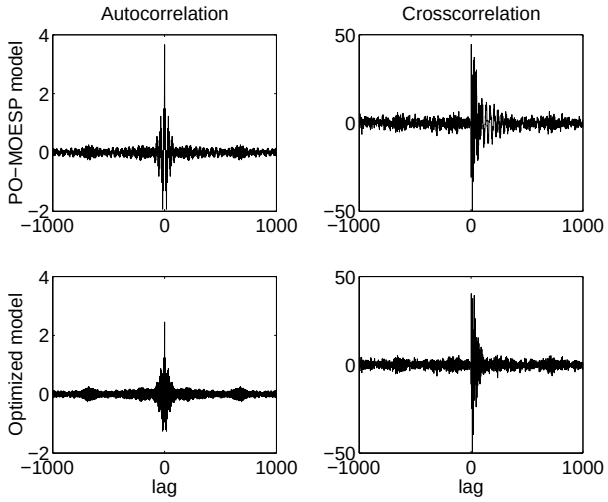


Figure 4.10: Auto-correlation functions of $\widehat{y}(k) - y(k)$ and the cross-correlation functions between $\widehat{y}(k) - y(k)$ and $u(k)$ for both the PO-MOESP model and the optimized model.

The auto-correlation and cross-correlation approaches that have been used in the above treatment share the property that their results are unique to a given data set. We will now compare the actual system's frequency response function (FRF) to that of the initial and optimized models. The result is shown in Figure 4.12. It is clear that the optimized model corresponds much better to the actual system behavior especially in the lower frequency-range and the first two oscillating modes.

Subsequently, the variance accounted for between the estimated and actual output signals can be calculated:

```
>> vaf(y,yi)
ans =
   93.7969
>> vaf(y,yo)
ans =
   95.8464
```

It is clear that the subspace identification yielded an already satisfactory model. However, this model has been improved significantly using the optimization framework.

Finally, a cross-validation check is performed. To this end, a fresh data set is generated for the actual system, the PO-MOESP model and the optimized model. Note that since the estimated models describe the system *without* delay, the delay first has to be removed from the actual system data before a cross-validation can be performed. The actual cross-validation is performed by calculating the variance accounted for.

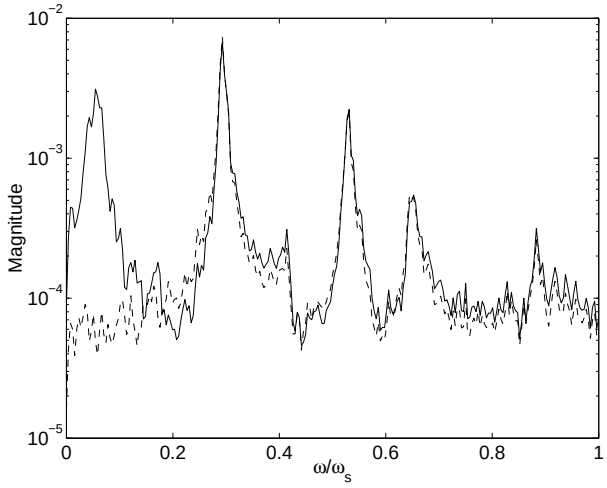


Figure 4.11: Spectra of the prediction errors of $\widehat{y}(k) - y(k)$ for both the PO-MOESP model (*solid*) and the optimized model (*dashed*).

```
>> uv=randn(N,1);
>> yv=dlsim(dense,numse,uv);
>> d=2;
>> uv=uv(1:N-d);
>> yv=yv(d+1:N);
>> yvi=dltisim(Ai,Bi,Ci,Di,uv);
>> yvo=dltisim(Ao,Bo,Co,Do,uv);
>> vaf(yv,yvi)
ans =
   93.4893
>> vaf(yv,yvo)
ans =
   95.5700
```

We can thus draw the conclusion that the estimated models describe the actual system sufficiently well. Although the PO-MOESP model is good, the optimized model is significantly better.

References

[1] The MathWorks Inc., Natick, Massachusetts, *Signal Processing Toolbox User's Guide*, second ed., Sept. 2000.

[2] A. Backx, *Identification of an Industrial Process: A Markov Parameter Approach*. PhD thesis, University of Eindhoven, Eindhoven, The Netherlands, 1987.

[3] B. Haverkamp, *Subspace Method Identification, Theory and Practice*. PhD thesis, Delft University of Technology, Delft, The Netherlands, 2000.

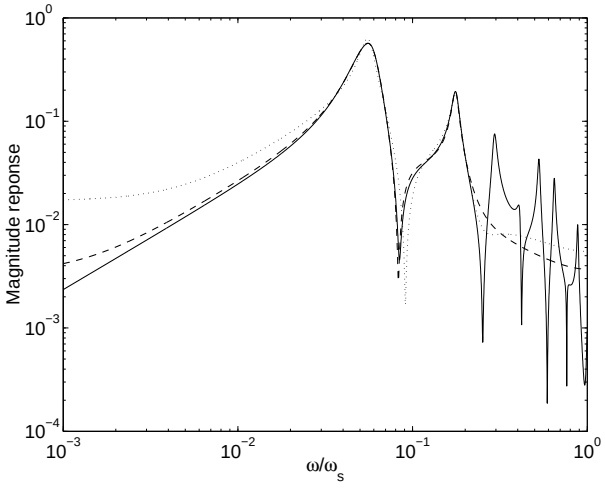


Figure 4.12: The magnitude of the frequency response function (FRF) for the actual system (*solid*), the initial PO-MOESP model (*dotted*) and the optimized model (*dashed*).

[4] L. Ljung, *System Identification Toolbox User's Guide*. The MathWorks Inc., Natick, Massachusetts, version 5 (release 12) ed., Nov. 2000.

[5] The MathWorks Inc., Natick, Massachusetts, *Using the Control Systems Toolbox*, version 1 (release 12) ed., Nov. 2000.

Chapter 5

Toolbox Software Manual

After studying this chapter you can:

- obtain and install the toolbox software.
- look up the manual page of each toolbox function.

5.1 Introduction

This chapter provides an installation and user's guide for the toolbox software. Section 5.2 explains how to obtain and install the software on a computer system. Section 5.3 provides an overview of the toolbox functions and a detailed reference page for every toolbox function.

5.2 Obtaining and Installing the Toolbox Software

The toolbox software is available on a CD-ROM provided with this book or as download from the publishers website. Two compressed files have been put in the `toolbox` directory on the CD-ROM; one for Windows and one for Linux (Fedora). Installing the toolbox is a two-step process. In the first step, the software is copied to the appropriate directory on the local system. In the second step, the directory to which the toolbox was copied is added to the MATLAB path so that the toolbox functions can be used from M-files in any directory.

The toolbox software can be installed system-wide, which makes it available to all users on the system. However, administrative privileges are required to do this. On Windows 95/98/Millennium systems this generally is no problem since any user is allowed to write files anywhere on the system. On Windows NT/2000/XP and on UNIX an ordinary user is generally not allowed to write to system directories, and the toolbox should be installed as "administrator" or "root".

Alternatively, the toolbox can be installed on a per-user basis. In this scenario, a user copies the toolbox to a directory in which he is allowed to write; generally his home-directory. After this, the toolbox directory can be added to the MATLAB path in a personal startup-file.

The next two sections describe the installation process on UNIX and Windows.

5.2.1 Installation on Linux

Copying the toolbox files to the right location on the local file-system is done in almost the same way in a system-wide and per-user installation, the only difference being the directory to which the files are copied. In the system-wide case this generally is a directory with a name similar to `/usr/local/matlab/toolbox`, whereas in the per-user case a directory `/home/username/toolbox` is suggested. In both cases, one should change the directory to one of those mentioned above, and issue the following command to unpack the toolbox:

```
tar xzf /home/username/LTIToolbox-2.0-Linux.tar.gz
```

Note that on UNIX systems other than Linux, the GNU tar command `gtar` must be used rather than `tar`.

The second installation step concerns adding the toolbox directory to MATLAB's search-path. In the system-wide case, the following line should be added to `/usr/local/matlab/toolbox/local/pathdef.m`:

```
matlabroot,'/toolbox/LTIToolbox',...
```

In the per-user installation, the following line should be added to `/home/username/matlab/startup.m`:

```
addpath /home/username/toolbox/LTIToolbox;
```

5.2.2 Installation on Windows

A decompression utility like for example WinZip can be used to decompress the toolbox zip-file `LTIToolbox-2.0-Windows.zip` to the appropriate directory. In a system-wide installation, this directory-name will be similar to `C:\MATLAB\toolbox`. In a per-user installation, this directory can reside anywhere as long as the user is allowed to write to the location at hand.

Adding the toolbox directory to MATLAB's search-path can be done in either the `pathdef.m` or `startup.m` files in the same way as in the Linux-installation. However, both files reside in the MATLAB-tree (for example `C:\MATLAB\toolbox\local`) under Windows. This means that a per-user path is not possible. If the user has no write-permission in the MATLAB-tree either the systems administrator should be contacted, or the following command should be issued each time MATLAB is started:

```
addpath C:\toolbox\LTIToolbox
```

Obviously, the directory-name in the above command should be replaced with the directory in which the toolbox was actually installed.

5.3 Toolbox Overview and Function Reference

0.4pt0pt

The toolbox software is based partly on the SMI-1.0 toolbox, and forms a superset of most of its functionality. Some function names and their syntax are changed to make them more consistent with the current function capabilities. Table 5.1 lists which functions have become obsolete, and which functions should be used instead. Table 5.2 lists all functions that are part of the toolbox software.

0.4pt0.4pt 0pt0pt

Table 5.1: Depreciated functions from SMI-1.0 and their equivalent in the new toolbox (if any).

SMI-1.0	*Description*	*New*	*Description*	*See page*
`dgnlisls`	Separable Least Squares optimization of discrete-time LTI systems	`doptlti`	Least Squares optimization of discrete-time LTI systems	113
`gnlisls`	Separable Least Squares optimization of continuous-time LTI systems	-	-	-
`gnwisls`	Separable Least Squares optimization of continuous-time Wiener systems	-	-	-
`dsmisim`	LTI system simulation	`dltisim`	LTI system simulation	108
`ss2thon`	State-space to Output Normal parameterization	`css2th`	Continuous-time parameterization of state-space systems	94
`th2sson`	Output Normal parameter vector to state-space conversion	`cth2ss`	Continuous-time parameter vector state-space conversion	96
`tchebest`	Estimate static nonlinear behavior	-	-	-
`tchebsim`	Simulate static nonlinear behavior	-	-	-

Table 5.2: List of functions in the toolbox software.

Time-domain subspace identification in discrete-time		
Function	*Description*	*See page*
`dordpi`	PI-MOESP preprocessing and order estimation	117
`dordpo`	PO-MOESP preprocessing and order estimation	119
`dordrs`	RS-MOESP preprocessing and order estimation	122
`dmodpi`	PI-MOESP estimation of A and C	109
`dmodpo`	PO-MOESP estimation of A, C and K	110
`dmodrs`	RS-MOESP estimation of A and C	112
`dac2b`	Estimation of B	98
`dac2bd`	Estimation of B and D	100
`dinit`	Estimation of the initial state	106
Frequency-domain subspace identification		
`fdordom`	Discrete-time preprocessing and order estimation	137
`fcordom`	Continuous-time preprocessing and order estimation	135
`fdmodom`	Discrete-time estimation of A and C	134
`fcmodom`	Continuous-time estimation of A and C	133
`fac2b`	Estimation of B	129
`fac2bd`	Estimation of B and D	131
Time-domain parametric model refinement		
`doptlti`	User-level LTI model optimization driver	113
`dfunlti`	Cost-function for `doptlti`	103
Frequency-domain parametric model refinement		
`foptlti`	User-level LTI model optimization driver	141
`ffunlti`	Cost-function for `foptlti`	139
Common function for parametric model refinement		
`dss2th`	Parameterization of discrete-time LTI models	124
`css2th`	Parameterization of continuous-time LTI models	94
`dth2ss`	Parameter vector to discrete-time LTI model conversion	126
`cth2ss`	Parameter vector to continuous-time LTI model conversion	96
`simlns`	Local parameter subspace calculation	153
`lmmore`	Moré-Hebden Levenberg-Marquardt optimization	144
Maximum likelihood utilities		
`destmar`	Multivariable AR model estimation	102
`cholicm`	Calculations of a Cholesky factor of the inverse covariance matrix of a multivariable AR noise model	92
Low-level simulation and calculation facilities		
`ltiitr`	State trajectory calculation	147
`ltifrf`	Frequency response function (FRF) calculation	148
Miscellaneous utilities		
`prbn`	Pseudo random binary sequence generation	152
`shave`	Peak and outlier removal	155
`vaf`	Variance-accounted-for calculation	157
`example`	Toolbox usage example	128
`mkoptstruc`	Optimization options generation	150
`optim5to6`	Translation of MATLAB 5 to MATLAB 6 optimization options	151
`dltisim`	Simulation of discrete-time LTI state-space systems	108

Purpose

Calculates a Cholesky-factor of the inverse covariance matrix (ICM) of a multivariable autoregressive process.

Syntax

```
C=cholicm(Af,Ab,Sf,Sb,N)
```

Description

The Inverse Covariance Matrix S of a multivariable autoregressive noise process according to [1] is calculated. A Cholesky factor C is returned such that $C^T C = S$

The noise model contains a causal and an auticausal part, both of which describe the actual noise $v(k)$. If $e(k)$ is a Gaussian white innovation, the model is given by:

$$v(k) = \vec{e}\ (k) - \vec{A}_1\ v(k-1) - ... - \vec{A}_d\ v(k-d),$$
$$v(k) = \overleftarrow{e}\ (k) - \overleftarrow{A}_1\ v(k+1) - ... - \overleftarrow{A}_d\ v(k+d).$$

Arrows $\rightharpoonup$ denote the causal (Forward) components while arrows $\leftharpoondown$ denote the anti-causal (Backward) ones.

The multivariable noise model can be obtained using the `destmar` function.

Inputs

`Af`	An $\ell \times \ell d$ matrix containing the causal part of the noise process. $\vec{A} = [\vec{A}_1\ \vec{A}_2\ \cdots\ \vec{A}_d]$.
`Ab`	An $\ell \times \ell d$ matrix containing the anti-causal part of the noise process. $\overleftarrow{A} = [\overleftarrow{A}_1\ \overleftarrow{A}_2\ \cdots\ \overleftarrow{A}_d]$.
`Sf`	An $\ell \times \ell$ matrix describing the covariance $E[\vec{e}\,\vec{e}^T]$
`Sb`	An $\ell \times \ell$ matrix describing the covariance $E[\overleftarrow{e}\,\overleftarrow{e}^T]$
`N`	The number of samples.

Outputs

`C`	A Cholesky factor of the ICM. This matrix is stored in LAPACK/BLAS band-storage; its size is $(d+1)\ell \times N$, and the bottom row contains the diagonal of C. The row above contains a zero, and then the first superdiagonal of C. The row above contains two zeros, and then the second superdiagonal, etc. The top row contains $(d+1)\ell - 1$ zeros, and then the $((d+1)\ell - 1)^{\text{th}}$ superdiagonal.

Limitations

A covariance matrix of a stationary process is always positive definite. However, it is very well possible to specify filter coefficients $\vec{A}$, $\overleftarrow{A}$ and covariances $\vec{S}$ and $\overleftarrow{S}$ such that the theoretical ICM calculated per [1] is not positive definite. In such cases, no Cholesky factor can be calculated, and an identity matrix will be returned along with a warning message. The filter should be checked and adjusted in these cases.

Algorithm

The upper-triangular block-band of a sparse banded inverse covariance matrix according to [1] is filled. A direct sparse Cholesky factorization is subsequently performed using MATLAB's internal `chol` function.

Used By

`doptlti`

See Also

`doptlti`, `destmar`

References

[1] B. David, *Parameter Estimation in Nonlinear Dynamical Systems with Correlated Noise*. PhD thesis, Université Catholique de Louvain, Louvain-La-Neuve, Belgium, 2001.

Purpose

Converts a continuous-time LTI state-space model into a parameter vector.

Syntax

```
[theta,params,T] = css2th(A,C,partype)
[theta,params,T] = css2th(A,B,C,partype)
[theta,params,T] = css2th(A,B,C,D,partype)
[theta,params,T] = css2th(A,B,C,D,x0,partype)
[theta,params,T] = css2th(A,B,C,D,x0,K,partype)
```

Description

This function converts a continuous-time LTI state-space model into a parameter vector that describes the model. Model structure:

$$\dot{x}(t) = Ax(t) + Bu(t) + Ke(t),$$
$$y(t) = Cx(t) + Du(t) + e(t).$$

Inputs

`A,B,C,D` — System matrices describing the state space system. The `B` and `D` matrices are optional and can be left out or passed as an empty matrix to indicate it is not part of the model.

`x0` — (optional) Initial state.

`K` — (optional) Kalman gain.

`partype` — This string specifies the type of parameterization that is used to parameterize the state space model. Three types of parameterization are supported:

- `'on'` Output Normal parametrization.
- `'tr'` Tridiagonal parametrization.
- `'fl'` Full parametrization.

Rules for input parameters:

The final parameter should always be the parametrization type. The order for the parameters prior to `partype` is `A,B,C,D,x0,K`. The only exception is `A,C`, when only those are to be parametrized.
All parameters after `A,B,C` and before `partype` are optional. If the last one is not to be parametrized it can be omitted. If any other is not to be parametrized, an empty matrix should be passed.
`(A,B,C,partype)` thus is equivalent to `(A,B,C,[],[],[],partype)` However, `(A,B,C,[],x0,partype)` cannot be abbreviated.

Outputs

`theta`	Parameters vector describing the system.
`params`	A structure that contains the dimension parameters of the system, such as the order, the number of inputs and whether D, x_0 or K is present.
`T`	Transformation matrix between the input state space system and the state space system in the form described by `theta`.

Remarks

This function is based on the SMI Toolbox 2.0 function `css2th`, copyright © 1996 Johan Bruls. Support for the omission of D, x_0 and/or K has been added, as well as support for the full parametrization.

Algorithm

The model parametrization for the output normal form and the tridiagonal parametrization is carried out according to [1]. The full model parametrization is a simple vectorization of the system matrices. In its most general form, the parameter vector is given by

$$\theta = \begin{bmatrix} \operatorname{vec}\left(\begin{bmatrix} A & B \\ C & D \end{bmatrix}\right) \\ \operatorname{vec}(K) \\ x_0 \end{bmatrix}.$$

Used By

`foptlti`

See Also

`cth2ss`, `dss2th`

References

[1] B. Haverkamp, *Subspace Method Identification, Theory and Practice*. PhD thesis, Delft University of Technology, Delft, The Netherlands, 2000.

Purpose

Converts a parameter vector into a continuous-time LTI state-space model.

Syntax

```
[A,C] = cth2ss(theta,params)
[A,B,C] = cth2ss(theta,params)
[A,B,C,D] = cth2ss(theta,params)
[A,B,C,D,x0] = cth2ss(theta,params)
[A,B,C,D,x0,K] = cth2ss(theta,params)
```

Description

This function converts a parameter vector that describes a continuous-time state space model into the state space matrices of that model. Model structure:

$$\dot{x}(t) = Ax(t) + Bu(t) + Ke(t),$$
$$y(t) = Cx(t) + Du(t) + e(t).$$

Inputs

`theta`	Parameter vector describing the system.
`params`	A structure that contains the dimension parameters of the system, such as the order, the number of inputs and whether D, x_0 or K is present in the model.
`T`	Transformation matrix to be applied to the state space system that is constructed from theta. This transformation might come from the function `css2th`.

Outputs

`A,B,C,D`	System matrices describing the state space system. If `theta` does not contain parameters for `D`, this matrix will be returned as an empty matrix.
`x0`	Initial condition. If `theta` does not contain parameters for `x0`, this vector will be returned as an empty matrix.
`K`	Kalman gain. If `theta` does not contain parameters for `K`, this vector will be returned as an empty matrix.

Remarks

This function is based on the SMI Toolbox 2.0 function `cth2ss`, copyright © 1996 Johan Bruls. Support for the omission of D, x_0 and/or K has been added, as well as support for the full parametrization.

Algorithm

See `css2th` on page 94.

Used By
`foptlti`, `ffunlti`

See Also
`css2th`, `dth2ss`

Purpose

Estimates the B matrices in discrete-time LTI state-space models from input-output measurements. D is assumed to be zero.

Syntax

```
B=dac2b(A,C,u,y)
B=dac2b(A,C,u1,y1,...,up,yp)
```

Description

This function estimates the B matrix corresponding to a discrete-time LTI state-space model. The estimate is based on the measured input-output data sequences, and on the A and C matrices, which are possibly estimated using `dmodpo`, `dmodpi` or `dmodrs`. The D matrix is assumed to be zero. Several data batches can be concatenated.

Inputs

`A`	The state-space model's A matrix.
`C`	The state-space model's C matrix.
`u,y`	The measured input-output data from the system to be identified. Multiple data batches can be specified by appending additional `u,y` pairs to the parameter list.

Outputs

`B`	The state-space model's B matrix.

Algorithm

Estimating B and the initial state x_0 from input-output data and A and C is a linear regression [1]:

$$\begin{bmatrix} x_0 \\ \mathrm{vec}(\hat{B}) \end{bmatrix} = \Phi^{\dagger}\theta.$$

The regression matrix Φ and data matrix θ are given by:

$$\Phi = \begin{bmatrix} C & 0 \\ CA & u(1)^T \otimes C \\ \vdots & \vdots \\ CA^{N-1} & \sum_{\tau=0}^{N-2} u(\tau+1)^T \otimes CA^{N-2-\tau} \end{bmatrix},$$

$$\theta = \begin{bmatrix} y(1) \\ y(2) \\ \vdots \\ y(N) \end{bmatrix}.$$

The function `ltiitr` is used to efficiently fill the regression matrix Φ.

Used By

This is a top-level function that is used directly by the user.

Uses Functions

`ltiitr`

See Also

`dac2bd`, `dmodpo`, `dmodpi`, `dmodrs`, `ltiitr`.

References

[1] B. Haverkamp, *Subspace Method Identification, Theory and Practice*. PhD thesis, Delft University of Technology, Delft, The Netherlands, 2000.

Purpose

Estimates the B and D matrices in discrete-time LTI state-space models from input-output measurements.

Syntax

```
[B,D]=dac2bd(A,C,u,y)
[B,D]=dac2bd(A,C,u1,y1,...,up,yp)
```

Description

This function estimates the B and D matrices corresponding to a discrete-time LTI state-space model. The estimate is based on the measured input-output data sequences, and on the A and C matrices, which are possibly estimated using `dmodpo`, `dmodpi` or `dmodrs`. Several data batches can be concatenated.

Inputs

`A`	The state-space model's A matrix.
`C`	The state-space model's C matrix.
`u,y`	The measured input-output data from the system to be identified. Multiple data batches can be specified by appending additional `u,y` pairs to the parameter list.

Outputs

`B`	The state-space model's B matrix.
`D`	The state-space model's D matrix.

Algorithm

Estimating B, D and the initial state x_0 from input-output data and A and C is a linear regression [1]:

$$\begin{bmatrix} x_0 \\ \mathrm{vec}(\hat{B}) \\ \mathrm{vec}(\hat{D}) \end{bmatrix} = \Phi^{\dagger}\theta.$$

The regression matrix Φ and data matrix θ are given by:

$$\Phi = \begin{bmatrix} C & 0 & u(1)^T \otimes I_\ell \\ CA & u(1)^T \otimes C & u(2)^T \otimes I_\ell \\ \vdots & \vdots & \vdots \\ CA^{N-1} & \sum_{\tau=0}^{N-2} u(\tau+1)^T \otimes CA^{N-2-\tau} & u(N)^T \otimes I_\ell \end{bmatrix},$$

$$\theta = \begin{bmatrix} y(1) \\ y(2) \\ \vdots \\ y(N) \end{bmatrix}.$$

The function `ltiitr` is used to efficiently fill the regression matrix Φ.

Used By

This is a top-level function that is used directly by the user.

Uses Functions

`ltiitr`

See Also

`dac2b`, `dmodpo`, `dmodpi`, `dmodrs`, `ltiitr`.

References

[1] B. Haverkamp, *Subspace Method Identification, Theory and Practice*. PhD thesis, Delft University of Technology, Delft, The Netherlands, 2000.

Purpose

Fits a multivariable autoregressive model to a time-series.

Syntax

```
[Af,Ab,Sf,Sb]=destmar(v,d)
```

Description

This function fits a multivariable autoregressive model to a time-series $v(k)$. The model-structure is

$$v(k) = \overrightarrow{e}\ (k) - \overrightarrow{A}_1\ v(k-1) - ... - \overrightarrow{A}_d\ v(k-d),$$
$$v(k) = \overleftarrow{e}\ (k) - \overleftarrow{A}_1\ v(k+1) - ... - \overleftarrow{A}_d\ v(k+d),$$

in which $\overrightarrow{e}\ (k)$ and $\overleftarrow{e}\ (k)$ are innovation sequences with covariance matrices $\overrightarrow{S}$ and $\overleftarrow{S}$ respectively. The fitting is performed according to [1].

Inputs

`v`	The time-series, a $N \times \ell$ matrix for a signal having N samples and which is ℓ-dimensional.
`d`	The desired order d of the AR model.

Outputs

`Af,Ab`	The coefficient matrices $\overrightarrow{A}$ and $\overleftarrow{A}$ of the causal and anticausal model.
`Sf,Sb`	The covariance matrices $\overrightarrow{S}$ and $\overleftarrow{S}$ of the causal and anticausal innovations.

Algorithm

A direct Hankel-matrix based estimation of the AR model is performed according to [1].

Used By

This is a top-level function that is used directly by the user.

See Also

`cholicm`, `doptlti`

References

[1] B. David, *Parameter Estimation in Nonlinear Dynamical Systems with Correlated Noise*. PhD thesis, Université Catholique de Louvain, Louvain-La-Neuve, Belgium, 2001.

Purpose

Calculates the cost-function information for `doptlti`.

Syntax

```
[epsilon]=dfunlti(th,u,y,params)
[epsilon,psi]=dfunlti(th,u,y,params)
[epsilon,psi,U2]=dfunlti(th,u,y,params)

[epsilon]=dfunlti(th,u,y,params,options,OptType,...
                  sigman,filtera,CorrD)
[epsilon,psi]=dfunlti(th,u,y,params,options,...
                      OptType,sigman,filtera,CorrD)
[epsilon,psi,U2]=dfunlti(th,u,y,params,options,...
                      OptType,sigman,filtera,CorrD)
```

Description

This function implements the cost-fuction for `doptlti`. It is not meant for standalone use.

Inputs

`th` Parameter vector describing the system

`u,y` The input and output data of the system to be optimized.

`params` A structure that contains the dimension parameters of the system, such as the order, the number of inputs, whether D, x_0 or K is present in the model.

`options` (optional) An `optimset` compatible options-structure. The fields `options.RFactor`, `options.LargeScale`, `options.Manifold` and `options.BlockSize` should have been added by `doptlti`.

`OptType` (optional) Indicates what kind of weighted least squares or maximum likelihood optimization is being performed:

- `'no_mle'` implies a nonlinear (weighted) least squares optimization.
- `'uncorr'` implies a maximum likelihood optimization without correlation among the output perturbances [1].
- `'flcorr'` implies a maximum likelihood optimization with correlated output perturbances [2].

`sigman` (optional) If `OptType` is `'no_mle'`, this can be a vector of size $1 \times \ell$ that indicates the standard deviation of the perturbance of each of the outputs.
If `OptType`is `'uncorr'`, this should be a vector of size $1 \times \ell$ that indicates the standard deviation of the white noise innovations for the output perturbance AR model.

If `OptType` is `'flcorr'`, this should be a Cholesky factor of the AR process' inverse covariance matrix, as obtained by `cholicm`.

`filtera` (optional) If `OptType` is `'uncorr'`, this should be the A-polynomial of a dth order AR noise model. The first element should be 1, and the other elements should be d filter coefficients. In the multi-output case `filtera` should be a matrix having $\max(d_i) + 1$ rows and ℓ columns. If a certain output noise model has a lower order, then the coefficient vector should be padded with `NaN`s.

`CorrD` (optional) If `OptType` is `'uncorr'`, this should be a correction matrix for the lower-right part of the ICM's Cholesky-factor. No details will be provided here.

Outputs

`epsilon` Output of the cost function, which is the square of the error between the output and the estimated output.

`psi` Jacobian Ψ_N of epsilon, $\Psi_N U_2$ if the full parametrization is used.

`U2` Left null-space of Manifold matrix for the full parametrization [3].

Algorithm

The formation of the error-vector is done by simple simulation of the current model:

$$\hat{x}(k+1;\theta) = A(\theta)\hat{x}(k;\theta) + B(\theta)u(k),$$
$$\hat{y}(k;\theta) = C(\theta)\hat{x}(k;\theta) + D(\theta)u(k).$$

The error-vector $E_N \in \mathbb{R}^{N\ell}$ is build up such that its ith blockrow consists of $y(k) - \hat{y}(k;\theta)$. Note that this example corresponds to the error-vector of an output error model in which no output weighting is applied. For innovation models and maximum likelihood corrections, the error-vector is different from the one shown above.

The Jacobian is formed by simulation as well [4]. This is a special case of the Jacobian for LPV systems that has been described in [3]. A QR-factorization is used to obtain its left null-space.

Used By

`doptlti` (via `lmmore`)

Uses Functions

`dth2ss`, `ltiitr`, `simlns`

See Also

`ffunlti`

References

[1] B. David and G. Bastin, "An estimator of the inverse covariance matrix and its application to ML parameter estimation in dynamical systems," *Automatica*, vol. 37, no. 1, pp. 99–106, 2001.

[2] B. David, *Parameter Estimation in Nonlinear Dynamical Systems with Correlated Noise*. PhD thesis, Université Catholique de Louvain, Louvain-La-Neuve, Belgium, 2001.

[3] L. H. Lee and K. Poolla, "Identification of linear parameter-varying systems using nonlinear programming," *Journal of Dynamic Systems, Measurement and Control*, vol. 121, pp. 71–78, Mar. 1999.

[4] N. Bergboer, V. Verdult, and M. Verhaegen, "An efficient implementation of maximum likelihood identification of LTI state-space models by local gradient search," in *Proceedings of the 41st IEEE Conference on Decision and Control*, (Las Vegas, Nevada), Dec. 2002.

Purpose

Estimates the initial state, given estimated discrete-time state-space system matrices and a batch of measured input-output data.

Syntax

```
x0=dinit(A,B,C,D,u,y)
```

Description

This function estimates the initial state for a measured input-output batch of a discrete-time LTI state-space model. The estimate is based on the measured input-output data sequences, and on the A, B, C and D matrices, which are possibly estimated using any of the subspace identification functions.

Inputs

`A,B,C,D`	The discrete-time LTI state-space model.
`u,y`	The measured input-output data from the system to be identified.

Outputs

`x0`	The estimated initial state.

Algorithm

Estimating the initial state x_0 from input-output data and the system matrices is a linear regression [1]:

$$x_0 = \Phi^{\dagger}\theta.$$

The regression matrix Φ and data matrix θ are given by

$$\Phi = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{N-1} \end{bmatrix},$$

$$\theta = \begin{bmatrix} y(1) - \hat{y}(1) \\ y(2) - \hat{y}(2) \\ \vdots \\ y(N) - \hat{y}(N) \end{bmatrix},$$

in which $\hat{y}(k)$ is simulated using the estimated system matrices and the measured input $u(k)$. The function `ltiitr` is used to efficiently calculate $\hat{y}(k)$.

Used By

This is a top-level function that is used directly by the user.

Uses Functions
`ltiitr`

See Also
`dac2b`, `dac2bd`, `ltiitr`.

References

[1] B. Haverkamp, *Subspace Method Identification, Theory and Practice*. PhD thesis, Delft University of Technology, Delft, The Netherlands, 2000.

Purpose

Simulates a discrete-time LTI state-space system.

Syntax

```
y=dltisim(A,B,C,D,u)
[y,x]=dltisim(A,B,C,D,u,x0)
```

Description

This function simulates a discrete-time LTI state-space system. The model structure is the following:

$$x(k+1) = Ax(k) + Bu(k),$$
$$y(k) = Cx(k) + Du(k).$$

An optional initial state can be specified.

Inputs

`A,B,C,D`	An LTI state-space system's matrices.
`u`	An $N \times m$ matrix containing N samples of the m inputs.
`x0`	(optional) The initial state, an $n \times 1$ vector.

Outputs

`y`	The computed output sequence, an $N \times \ell$ matrix.
`x`	(optional) The computed state, an $N \times n$ matrix.

Algorithm

A direct iteration of the system's state-transition equation is used to obtain the state-trajectory for all time-instants. The function `ltiitr` is used to this end.

Uses Functions

`ltiitr`

See Also

`ltiitr`, `dsim`

Purpose

Estimates the A and C matrix in a discrete-time state-space model from time-domain data that was preprocessed by `dordpi`.

Syntax

```
[A,C]=dmodpi(R,n)
```

Description

This function estimates the A and C matrices corresponding to an nth order discrete-time LTI state-space model. The compressed data matrix `R` from the preprocessor function `dordpi` is used to this end.

Inputs

`R`	A compressed data matrix containing information about the measured data, as well as information regarding the system dimensions.
`n`	The desired model order n.

Outputs

`A`	The state-space model's A matrix.
`C`	The state-space model's C matrix.

Algorithm

The data matrix obtained with `dordpi` contains the weighted left singular vectors of the R_{32} matrix (see page 117). The first n of these vectors form an estimate $\hat{\mathcal{O}}_s$ of the system's extended observability matrix:

$$\mathcal{O}_s = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{s-1} \end{bmatrix}.$$

The estimates $\hat{A}$ and $\hat{C}$ are obtained by linear regression:

$$\hat{C} = \hat{\mathcal{O}}_s(1:\ell,:),$$
$$\hat{A} = \hat{\mathcal{O}}_s(1:(s-1)\ell,:)^{\dagger}\hat{\mathcal{O}}_s(\ell+1:s\ell,:).$$

Used By

This is a top-level function that is used directly by the user.

See Also

`dordpi`, `dordpo`, `dmodpo`, `dordrs`, `dmodrs`

Purpose

Estimates the A and C matrices and the Kalman gain in a discrete-time state-space model from time-domain data that was preprocessed by `dordpo`.

Syntax

```
[A,C]=dmodpo(R,n)
[A,C,K]=dmodpo(R,n)
```

Description

This function estimates the A and C matrices corresponding to an nth order discrete-time LTI state-space model. A Kalman gain can be estimated as well. The compressed data matrix R from the preprocessor function `dordpo` is used to this end.

Inputs

`R`	A compressed data matrix containing information about the measured data, as well as information regarding the system dimensions.
`n`	The desired model order n.

Outputs

`A`	The state-space model's A matrix.
`C`	The state-space model's C matrix.
`K`	Kalman gain matrix.

Remarks

The data matrix `R` generated by the M-file implementation of `dordpo` is *incompatible* with the `R` matrix generated by the MEX-implementation of `dordpo`. Therefore, either the M-files should be used for both `dordpo` and `dmodpo`, or the MEX-files should be used for both functions.

The MEX-implementation of `dmodpo` uses the `IB01BD` function from the SLICOT library.

The MEX-implementation may generate the following warning:

```
Warning: Covariance matrices are too small:
returning K=0
```

This implies that there is not enough information available to reliably estimate a Kalman gain. $K = 0$ is returned for stability reasons in this case.

Algorithm

The data matrix obtained with `dordpo` contains the weighted left singular vectors of the R_{32} matrix (see page 119). The first n of these

vectors form an estimate $\hat{\mathcal{O}}_s$ of the system's extended observability matrix:

$$\mathcal{O}_s = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{s-1} \end{bmatrix}.$$

The estimates $\hat{A}$ and $\hat{C}$ are obtained by linear regression:

$$\hat{C} = \hat{\mathcal{O}}_s(1:\ell,:),$$
$$\hat{A} = \hat{\mathcal{O}}_s(1:(s-1)\ell,:)^{\dagger}\hat{\mathcal{O}}_s(\ell+1:s\ell,:).$$

The optional Kalman gain is calculated based on estimated noise covariance matrices [1].

Used By

This is a top-level function that is used directly by the user.

See Also

`dordpo`, `dordpi`, `dmodpi`, `dordrs`, `dmodrs`

References

[1] M. Verhaegen, "Identification of the deterministic part of MIMO state space models given in innovations form from input-output data," *Automatica*, vol. 30, no. 1, pp. 61–74, 1994.

Purpose

Estimates the A and C matrix in a discrete-time state-space model from time-domain data that was preprocessed by `dordrs`.

Syntax

```
[A,C]=dmodrs(R)
```

Description

This function estimates the A and C matrices corresponding to an nth order discrete-time LTI state-space model. The compressed data matrix `R` from the preprocessor function `dordrs` is used to this end. As n is determined from the x matrix that was passed to `dordrs`, it does not have to be specified here.

Inputs

`R`	A compressed data matrix containing information about the measured data, as well as information regarding the system dimensions.

Outputs

`A`	The state-space model's A matrix.
`C`	The state-space model's C matrix.

Algorithm

The data matrix obtained with `dordrs` contains the weighted left singular vectors of the R_{32} matrix (see page 122). The first n of these vectors form an estimate $\hat{\mathcal{O}}_s$ of the system's extended observability matrix:

$$\mathcal{O}_s = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{s-1} \end{bmatrix}.$$

The estimates $\hat{A}$ and $\hat{C}$ are obtained by linear regression:

$$\hat{C} = \hat{\mathcal{O}}_s(1:\ell,:),$$
$$\hat{A} = \hat{\mathcal{O}}_s(1:(s-1)\ell,:)^{\dagger}\hat{\mathcal{O}}_s(\ell+1:s\ell,:).$$

Used By

This is a top-level function that is used directly by the user.

See Also

`dordrs`, `dordpo`, `dmodpo`, `dordpi`, `dmodpi`

Purpose

Performs a nonlinear least squares or maximum likelihood optimization of a discrete time LTI state space model.

Syntax

```
[A,B,C,D]=doptlti(u,y,A,B,C,D)
[A,B,C,D,x0,K,options]=doptlti(u,y,A,B,C,D,x0,K,...
            partype,options,sigman,nmodel)
```

Description

This function performs a nonlinear least squares optimization of a discrete time linear state space system model with structure

$$x(k+1) = Ax(k) + Bu(k),$$
$$y(k) = Cx(k) + Du(k).$$

The function also supports innovation models:

$$x(k+1) = Ax(k) + Bu(k) + Ke(k),$$
$$y(k) = Cx(k) + Du(k) + e(k).$$

First, the state space matrices are parameterized. The output normal parametrization, the tridiagonal parametrization and the full parametrization can be used.

The parameterized model is optimized using the supplied `lmmore` Levenberg-Marquardt function. The matrices A, B, and C are always returned. If needed, D, the initial state and a Kalman gain can also be optimized.

Inputs

`u,y`	The input and output data of the system to be optimized.
`A,B,C,D`	Initial estimates of the system matrices A, B, C and D.
`x0`	(optional) The initial state
`K`	(optional) Kalman gain
`partype`	(optional) This parameter specifies the type of parameterization that is used to parameterize the state space model. Three types of parameterization are supported: `'on'` Output Normal parametrization. `'tr'` Tridiagonal parametrization. `'fl'` Full parametrization.
`options`	(optional) Input parameters that are passed on directy to the optimization function. These options may be compatible with the `optimset` options from the MATLAB 6 Optimization Toolbox[1]. Alternatively, a MATLAB 5 Optimization Toolbox compatible `foptions` vector may be specified.

There are a number of fields in addition to the normal fields in the `options` structure. These are described in detail in the remarks section below.

`sigman` (optional) The function of this parameters depends on its format:

1. If `sigman` is a $1 \times \ell$ vector, the output errors will be weighted by the inverse of these factors. In a weighted least squares estimation, `sigman` should contain the standard deviations of the noise on each of the outputs.
   In a maximum likelihood estimation which assumes no correlation between the noise on different outputs [2], `sigman` should contain the standard deviations of the white-noise signals which, when put through the AR filter specified by `nmodel`, generates the output measure- ment noise.
2. If `sigman` is an $\ell \times 2\ell$ matrix, a maximum likelihood estimation which does support correlation between the output noises will be carried out [3]. The `nmodel` parameter *must* be specified in this case.
   sigman should be $[\vec{S}\ \overleftarrow{S}]$, in which $\vec{S}$ is the covariance matrix of the multivariable white noise sequence that is put through the causal filter $\vec{A}$ (see `nmodel`). $\overleftarrow{S}$ is the covariance matrix of the white noise sequence that will be put through the anti-causal filter $\overleftarrow{A}$.

`nmodel` (optional) The specification of the AR noise model. This should be either a matrix of size $d \times \ell$, or a matrix of size $2\ell \times \ell d$, for an AR model of order d.
In the first output case `nmodel` should be a matrix having a number of rows equal to the highest noise-model order on any of the outputs. The matrix should have ℓ columns. If a certain output noise model has a lower order, then pad the coefficient vector with `NaN`s.
In the second case, filtera should be $[\vec{A}; \overleftarrow{A}]$ in which $\vec{A}$ specifies the causal AR filter, and $\overleftarrow{A}$ specifies the anti-causal AR filter, as obtained using `cholicm`.

Outputs

`A,B,C,D` System matrices of the optimized linear model. If the `D` matrix is not estimated, it will be empty.

`x0` Estimate of the initial state. If the `x0` matrix is not estimated, it will be returned empty.

`K` Estimate of the Kalman gain. If the `K` matrix is not estimated, it will be returned empty.

`options` Output parameters from the Optimization Toolbox. See `foptions` or `optimset`.

Remarks

An extra field `options.Manifold` may be set to `'on'` if the full parametrization is used. The `Manifold` field indicates whether the search direction should be confined to directions in which the cost-function changes. If `options.Manifold` is not set, `doptlti` will set it to `'off'` for the output normal and tridiagonal parametrizations, and to `'on'` for the full parametrization.

Another new field that can be set is the `options.BlockSize` field. The value N_b of the `BlockSize` field indicates that the Jacobian in the cost-function is build up N_b block-rows at a time rather than all at once [4]. This option is mainly interesting in tight-memory situations or for problems with a very large number of samples. If `options.BlockSize` is set to N_b, the fields `options.RFactor` and `options.LargeScale` are set to `'on'` automatically. A rule of thumb is that the Jacobian-calculation requires about $24(p + 1 + N_b\ell)(p + 1)$ bytes of computer memory, in which p is the number of free parameters. For the full parametrization, this is the number of parameters *after* a manifold-projection.

If the model is unstable one can use the innovation description. This implies choosing a K such that $(A - KC)$ is stable. The first option is to just specify K in the parameter list. This starts a prediction error optimization in which K is optimized as well. Faster convergence can be obtained by restricting K to a fixed value. To this end, the field `options.OEMStable` should be set to `'on'`, in addition to specifying K in the parameter list.

This optimization function has been targeted at MATLAB version 6 or higher. However, the function will run on MATLAB version 5 using a compatibility kludge. This kludge implies that the options input parameter can either be a MATLAB 6 `optimset`-structure, or a MATLAB 5 compatible `foptions`-vector. However, the latter is discouraged since it does not allow the `Manifold`, `LargeScale`, `RFactor`, `BlockSize` and `OEMStable` fields to be set.

Limitations

The `doptlti`-function is a *non-linear* optimization. This implies that there is the inherent risk of ending up in a local minimum, rather than in the cost-function's global minimum. Therefore, a well-chosen initial model should be used. If the optimization gets stuck in a local minimum nontheless, a different initial model should be tried.

An initial estimate can be obtained by using the time-domain subspace identification functions in this toolbox. The relevant functions

are `dordpo`, `dmodpo`, `dordpi`, `dmodpi`, `dordrs`, `dmodrs`, `dac2b` and `dac2bd`.

Used By

This is a top-level function that is used directly by the user.

Uses Functions

`lmmore`, `dss2th`, `dth2ss`, `dfunlti`, `cholicm`

See Also

`foptlti`, `optimset`, `foptions`, `mkoptstruc`

References

[1] The MathWorks Inc., Natick, Massachusetts, *Optimization Toolbox User's Guide*, version 2.1 (release 12) ed., Sept. 2000.

[2] B. David and G. Bastin, "An estimator of the inverse covariance matrix and its application to ML parameter estimation in dynamical systems," *Automatica*, vol. 37, no. 1, pp. 99–106, 2001.

[3] B. David, *Parameter Estimation in Nonlinear Dynamical Systems with Correlated Noise*. PhD thesis, Université Catholique de Louvain, Louvain-La-Neuve, Belgium, 2001.

[4] N. Bergboer, V. Verdult, and M. Verhaegen, "An efficient implementation of maximum likelihood identification of LTI state-space models by local gradient search," in *Proceedings of the 41st IEEE Conference on Decision and Control*, (Las Vegas, Nevada), Dec. 2002.

Purpose

Preprocesses time-domain data for PI-MOESP subspace identification of discrete-time LTI state-space models. Delivers an order-estimate.

Syntax

```
[S,R]=dordpi(u,y,s)
[S,R]=dordpi(u,y,s,Rold)
```

Description

This function performs the initial data compression for PI-MOESP subspace identification based on measured input-output data [1]. In addition, it delivers information usable for determining the required model order. The model structure is the following

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k), \\ y(k) &= Cx(k) + Du(k) + v(k). \end{aligned}$$

Here, $v(k)$ is zero-mean noise of arbitary color, independent of the noise-free input $u(k)$. Several data batches can be concatenated, as shown below. This function acts as a preprocessor to `dmodpi`.

Inputs

`u,y`	The measured input and output data of the system to be identified.
`s`	The block-size parameter. This scalar should be $> n$.
`Rold`	(optional) The data-matrix resulting from a previous call to `dordpi`.

Outputs

`S`	The first s singular values of the rank-deficient R_{32} matrix (see below).
`R`	A compressed data matrix containing information about the measured data, as well as information regarding the system dimensions.

Algorithm

The discrete-time data compression algorithm in [1] is used. The following RQ-factorization is made:

$$\begin{bmatrix} U_{s+1,s,N-2s+1} \\ U_{1,s,N-2s+1} \\ Y_{s+1,s,N-2s+1} \end{bmatrix} = \begin{bmatrix} R_{11} & 0 & 0 \\ R_{21} & R_{22} & 0 \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \begin{bmatrix} Q_1 \\ Q_2 \\ Q_3 \end{bmatrix}.$$

The meaning of the various matrices can be found in the cited article. A weighted SVD of the R_{32} matrix is made, and its left singular

vectors are appended to the R-matrix. Its first s singular values are returned in S.

Used By

This is a top-level function that is used directly by the user.

See Also

dmodpi, dordpo, dmodpo, dordrs, dmodrs

References

[1] M. Verhaegen, "Identification of the deterministic part of MIMO state space models given in innovations form from input-output data," *Automatica*, vol. 30, no. 1, pp. 61–74, 1994.

Purpose

Preprocesses time-domain data for PO-MOESP subspace identification of discrete-time state-space models. Delivers an order-estimate.

Syntax

```
[S,R]=dordpo(u,y,s)
[S,R]=dordpo(u,y,s,Rold)
```

Description

This function performs the initial data compression for PO-MOESP subspace identification based on measured input-output data [1]. In addition, it delivers information usable for determining the required model order. The model structure is the following:

$$x(k+1) = Ax(k) + Bu(k) + w(k),$$
$$y(k) = Cx(k) + Du(k) + v(k).$$

Here, $w(k)$ and $v(k)$ are zero-mean white noise sequences, independent of the noise-free input $u(k)$. Several data batches can be concatenated, as shown below. This function acts as a preprocessor to `dmodpo`.

Inputs

`u,y`	The measured input and output data of the system to be identified.
`s`	The block-size parameter. This scalar should be $> n$.
`Rold`	(optional) The data-matrix resulting from a previous call to `dordpo`.

Outputs

`S`	The first s singular values of the rank-deficient R_{32} matrix (see below).
`R`	A compressed data matrix containing information about the measured data, as well as information regarding the system dimensions.

Remarks

The data matrix `R` generated by the M-file implementation of `dordpo` is *incompatible* with the `R` matrix generated by the MEX-implementation of `dordpo`. Therefore, either the M-files should be used for both `dordpo` and `dmodpo`, or the MEX-files should be used for both functions.

The MEX-implementation of `dordpo` uses the `IB01MD` and `IB01ND` functions from the SLICOT library.

The MEX-implementation may return the warning:

```
Warning: Cholesky failed: using QR for this and any
subsequent batches
```

This implies that a fast Cholesky algorithm failed and that the function has fallen back onto a slower QR algorithm. This warning does not imply that results are invalid; the results can be used without problems.

Algorithm

The discrete-time data compression algorithm in [1] is used. In the M-file implementation, the following RQ-factorization is made:

$$\begin{bmatrix} U_{s+1,s,N-2s+1} \\ \begin{bmatrix} U_{1,s,N-2s+1} \\ Y_{1,s,N-2s+1} \end{bmatrix} \\ Y_{s+1,s,N-2s+1} \end{bmatrix} = \begin{bmatrix} R_{11} & 0 & 0 \\ R_{21} & R_{22} & 0 \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \begin{bmatrix} Q_1 \\ Q_2 \\ Q_3 \end{bmatrix}.$$

The meaning of the various matrices can be found in the cited article. In the MEX-implementation, the following Cholesky-factorization is attempted first:

$$\begin{bmatrix} U_{s+1,s,N-2s+1} \\ \begin{bmatrix} U_{1,s,N-2s+1} \\ Y_{1,s,N-2s+1} \end{bmatrix} \\ Y_{s+1,s,N-2s+1} \end{bmatrix} \begin{bmatrix} U_{s+1,s,N-2s+1} \\ \begin{bmatrix} U_{1,s,N-2s+1} \\ Y_{1,s,N-2s+1} \end{bmatrix} \\ Y_{s+1,s,N-2s+1} \end{bmatrix}^T$$
$$= \begin{bmatrix} R_{11} & 0 & 0 \\ R_{21} & R_{22} & 0 \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \begin{bmatrix} R_{11} & 0 & 0 \\ R_{21} & R_{22} & 0 \\ R_{31} & R_{32} & R_{33} \end{bmatrix}^T.$$

If this factorization fails, the algorithm falls back on the above RQ-factorization. In all cases, a weighted SVD of the R_{32} matrix is made, and its left singular vectors are appended to the R-matrix. Its first s singular values are returned in S.

Used By

This is a top-level function that is used directly by the user.

Uses Functions

SLICOT-functions IB01MD and IB01ND.

LAPACK-function DPOTRF.

(All built into the executable)

See Also

`dmodpo`, `dordpi`, `dmodpi`, `dordrs`, `dmodrs`

References

[1] M. Verhaegen, "Identification of the deterministic part of MIMO state space models given in innovations form from input-output data," *Automatica*, vol. 30, no. 1, pp. 61–74, 1994.

Purpose

Preprocesses time-domain data for the iterative Reconstructed State RS-MOESP algorithm to identify discrete-time state-space models. Delivers an order-estimate.

Syntax

```
[S,R]=dordpo(u,y,x,s)
[S,R]=dordpo(u,y,x,s,Rold)
```

Description

This function performs the initial data compression for RS-MOESP subspace identification based on measured input-output data and a reconstructed state from a previous model estimate [1]. In addition, it delivers information usable for determining the required model order. The model structure is the following:

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k), \\ y(k) &= Cx(k) + Du(k) + v(k). \end{aligned}$$

Here, $v(k)$ is zero-mean noise of arbitary color, independent of the noise-free input $u(k)$. Several data batches can be concatenated, as shown below. This function acts as a preprocessor to `dmodrs`.

Inputs

`u,y`	The measured input and output data of the system to be identified.
`x`	The reconstructed state. This state can be obtained by simulating the state-equation belonging the the the previous model estimate's $\hat{A}$ and $\hat{B}$ matrices: $$x(k+1) = \hat{A}x(k) + \hat{B}u(k)$$ This initial model can be obtained by e.g. PI-MOESP.
`s`	The block-size parameter. This scalar should be $> n$.
`Rold`	(optional) The data-matrix resulting from a previous call to `dordrs`.

Outputs

`S`	The first s singular values of the rank-deficient R_{32} matrix (see below).
`R`	A compressed data matrix containing information about the measured data, as well as information regarding the system dimensions.

Algorithm

The discrete-time data compression algorithm in [1] is used. The following RQ-factorization is made:

$$\begin{bmatrix} U_{s+1,s,N-2s+1} \\ \begin{bmatrix} U_{1,s,N-2s+1} \\ X_{1,s,N-2s+1} \end{bmatrix} \\ Y_{s+1,s,N-2s+1} \end{bmatrix} = \begin{bmatrix} R_{11} & 0 & 0 \\ R_{21} & R_{22} & 0 \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \begin{bmatrix} Q_1 \\ Q_2 \\ Q_3 \end{bmatrix}.$$

The meaning of the various matrices can be found in the cited article. A weighted SVD of the R_{32} matrix is made, and its left singular vectors are appended to the R-matrix. Its first s singular values are returned in S.

Used By

This is a top-level function that is used directly by the user.

See Also

`dmodrs`, `dordpo`, `dmodpo`, `dordpi`, `dmodpi`

References

[1] M. Verhaegen, "Subspace model identification part 3. Analysis of the ordinary output-error state-space model identification algorithm," *International Journal of Control*, vol. 56, no. 3, pp. 555–586, 1993.

Purpose

Converts a discrete-time LTI state-space model into a parameter vector.

Syntax

```
[theta,params,T] = dss2th(A,C,partype)
[theta,params,T] = dss2th(A,B,C,partype)
[theta,params,T] = dss2th(A,B,C,D,partype)
[theta,params,T] = dss2th(A,B,C,D,x0,partype)
[theta,params,T] = dss2th(A,B,C,D,x0,K,partype)
```

Description

This function converts a discrete-time LTI state-space model into a parameter vector that describes the model. Model structure:

$$x(k+1) = Ax(k) + Bu(k) + Ke(k),$$
$$y(k) = Cx(k) + Du(k) + e(k).$$

Inputs

`A,B,C,D` System matrices describing the state space system. The `B` and `D` matrices are optional and can be left out or passed as an empty matrix to indicate it is not part of the model.

`x0` (optional) Initial state.

`K` (optional) Kalman gain.

`partype` This parameter specifies the type of parameterization that is used to parameterize the state space model. Three types of parameterization are supported:

- `'on'` Output Normal parametrization.
- `'tr'` Tridiagonal parametrization.
- `'fl'` Full parametrization.

Rules for input parameters:

The final parameter should always be the parametrization type. The order for the parameters prior to `partype` is `A,B,C,D,x0,K`. The only exception is `A,C`, when only those are to be parametrized.
All parameters after `A,B,C` and before `partype` are optional. If the last one is not to be parametrized it can be omitted. If any other is not to be parametrized, an empty matrix should be passed.
`(A,B,C,partype)` thus is equivalent to `(A,B,C,[],[],[],partype)` However, `(A,B,C,[],x0,partype)` cannot be abbreviated.

Outputs

`theta`	Parameters vector describing the system.
`params`	A structure that contains the dimension parameters of the system, such as the order, the number of inputs and whether D, x_0 or K is present.
`T`	Transformation matrix between the input state space system and the state space system in the form described by `theta`.

Remarks

This function is based on the SMI Toolbox 2.0 function `dss2th`, copyright © 1996 Johan Bruls. Support for the omission of D, x_0 and/or K has been added, as well as support for the full parametrization.

Algorithm

See `css2th` on page 94.

Used By

`doptlti`, `foptlti`

See Also

`dth2ss`, `css2th`

Purpose

Converts a parameters vector into a discrete-time LTI state-space model.

Syntax

```
[A,C] = dth2ss(theta,params)
[A,B,C] = dth2ss(theta,params)
[A,B,C,D] = dth2ss(theta,params)
[A,B,C,D,x0] = dth2ss(theta,params)
[A,B,C,D,x0,K] = dth2ss(theta,params)
```

Description

This function converts a parameter vector that describes a discrete-time state space model into the state space matrices of that model. Model structure:

$$x(k+1) = Ax(k) + Bu(k) + Ke(k),$$
$$y(k) = Cx(k) + Du(k) + e(k).$$

Inputs

`theta`	Parameter vector describing the system.
`params`	A structure that contains the dimension parameters of the system, such as the order, the number of inputs and whether D, x_0 or K is present in the model.
`T`	Transformation matrix to be applied to the state space system that is constructed from theta. This transformation might come from the function `dss2th`.

Outputs

`A,B,C,D`	System matrices describing the state space system. If `theta` does not contain parameters for `D`, this matrix will be returned as an empty matrix.
`x0`	Initial condition. If `theta` does not contain parameters for `x0`, this vector will be returned as an empty matrix.
`K`	Kalman gain. If `theta` does not contain parameters for `K`, this vector will be returned as an empty matrix.

Remarks

This function is based on the SMI Toolbox 2.0 function `dth2ss`, copyright © 1996 Johan Bruls. Support for the omission of D, x_0 and/or K has been added, as well as support for the full parametrization.

Algorithm

See `css2th` on page 94.

Used By

`doptlti`, `foptlti`, `dfunlti`, `ffunlti`

See Also

`dss2th`, `cth2ss`

Purpose

Shows an example on how to use the toolbox functions.

Syntax

```
example
```

Description

This example shows how to use the subspace identifications functions, as well as the nonlinear least squares and maximum likelihood optimization function. The test object is a 4^{th}-order steam engine model that has 2 inputs and 2 outputs.

Inputs

```
None
```

Outputs

```
None
```

Remarks

This script uses the datafile `example.mat`.

Used By

This is a top-level function that is used directly by the user.

Purpose

Estimates the B matrix in discrete-time and continuous-time state-space models from frequency response function (FRF) data, the D matrix is assumed to be zero.

Syntax

```
B=fac2b(A,C,H,w)
B=fac2b(A,C,H1,w1,...,Hp,wp)
```

Description

This function estimates the B matrix corresponding to a discrete-time or continuous-time LTI state-space model. The estimate is based on the measured frequency response function (FRF) data, and on the A and C matrices, which are possibly estimated using `fdmodom` or `fcmodom`. The D matrix is assumed to be zero. Several data batches can be concatenated, though this is possible for discrete-time models only.

Inputs

`A` The state-space model's A matrix.

`C` The state-space model's C matrix.

`H` The measured frequency response function (FRF). This should be a matrix which follows the convention of MATLAB 6; it should be $\ell \times m \times N$ in which `H(:,:,i)` contains the complex FRF at the i^{th} complex frequency.

`w` Vector of complex frequencies at which the FRF is measured. Although the function can operate using arbitrary complex frequencies, the following two choices are rather standard for discrete and continuous time models respectively:

$$\mathtt{w} = e^{j\omega}$$
$$\mathtt{w} = j\omega$$

For discrete-time models, multiple data batches can be concatenated by appending additional `H`,`w` pairs to the parameter list.

Outputs

`B` The state-space model's B matrix.

`R` A compressed data matrix that can be used to concatenate another data batch in a subsequent call to `fac2b` (discrete-time models only).

Algorithm

Estimating B from the frequency response function (FRF) data and

A and C is a linear regression [1]:

$$\hat{B} = \begin{bmatrix} \mathrm{Re}(\Phi) \\ \mathrm{Im}(\Phi) \end{bmatrix}^{\dagger} \begin{bmatrix} \mathrm{Re}(\theta) \\ \mathrm{Im}(\theta) \end{bmatrix}.$$

The complex regression matrix Φ and data matrix θ are given by:

$$\Phi = \begin{bmatrix} \hat{C}(\xi_1 - \hat{A})^{-1} \\ \hat{C}(\xi_2 - \hat{A})^{-1} \\ \vdots \\ \hat{C}(\xi_N - \hat{A})^{-1} \end{bmatrix},$$

$$\theta = \begin{bmatrix} G_1 \\ G_2 \\ \vdots \\ G_N \end{bmatrix}.$$

The function `ltifrf` is used to efficiently fill the regression matrix Φ.

Used By

This is a top-level function that is used directly by the user.

Uses Functions

`ltifrf`

See Also

`fac2bd`, `fdmodom`, `fcmodom`, `ltifrf`.

References

[1] T. McKelvey, H. Akçay, and L. Ljung, "Subspace-based multivariable system identification from frequency response data," *IEEE Transactions on Automatic Control*, vol. 41, pp. 960–979, July 1996.

Purpose

Estimates the B and D matrices in discrete-time and continuous-time state-space models from frequency response function (FRF) data.

Syntax

```
[B,D]=fac2bd(A,C,H,w)
[B,D]=fac2bd(A,C,H1,w1,...,Hp,wp)
```

Description

This function estimates the B and D matrices corresponding to a discrete-time or continuous-time LTI state-space model. The estimate is based on the measured frequency response function (FRF) data, and on the A and C matrices, which are possibly estimated using `fdmodom` or `fcmodom`. Several data batches can be concatenated, though this is possible for discrete-time models only.

Inputs

`A` The state-space model's A matrix.

`C` The state-space model's C matrix.

`H` The measured frequency response function (FRF). This should be a matrix which follows the convention of MATLAB 6; it should be $\ell \times m \times N$ in which `H(:,:,i)` contains the complex FRF at the i^{th} complex frequency.

`w` Vector of complex frequencies at which the FRF is measured. Although the function can operate using arbitrary complex frequencies, the following two choices are rather standard for discrete and continuous time models respectively:

$$\mathtt{w} = e^{j\omega}$$
$$\mathtt{w} = j\omega$$

For discrete-time models, multiple data batches can be concatenated by appending additional `H`,`w` pairs to the parameter list.

Outputs

`B` The state-space model's B matrix.

`D` The state-space model's D matrix.

Algorithm

Estimating B and D from the frequency response function (FRF) data and A and C is a linear regression [1]:

$$\begin{bmatrix}\hat{B}\\ \hat{D}\end{bmatrix} = \begin{bmatrix}\mathrm{Re}(\Phi)\\ \mathrm{Im}(\Phi)\end{bmatrix}^{\dagger} \begin{bmatrix}\mathrm{Re}(\theta)\\ \mathrm{Im}(\theta)\end{bmatrix}.$$

The complex regression matrix Φ and data matrix θ are given by:

$$\Phi = \begin{bmatrix} \hat{C}(\xi_1 - \hat{A})^{-1} & I_\ell \\ \hat{C}(\xi_2 - \hat{A})^{-1} & I_\ell \\ \vdots & \vdots \\ \hat{C}(\xi_N - \hat{A})^{-1} & I_\ell \end{bmatrix},$$

$$\theta = \begin{bmatrix} G_1 \\ G_2 \\ \vdots \\ G_N \end{bmatrix}.$$

The function `ltifrf` is used to efficiently fill the regression matrix Φ.

Used By

This is a top-level function that is used directly by the user.

Uses Functions

`ltifrf`

See Also

`fac2b`, `fdmodom`, `fcmodom`, `ltifrf`.

References

[1] T. McKelvey, H. Akçay, and L. Ljung, "Subspace-based multivariable system identification from frequency response data," *IEEE Transactions on Automatic Control*, vol. 41, pp. 960–979, July 1996.

Purpose

Estimates the A and C matrix in a continuous-time state-space model from frequency response function (FRF) data that was preprocessed by `fcordom`.

Syntax

```
[A,C]=fcmodom(R,n)
```

Description

This function estimates the A and C matrices corresponding to an nth order discrete-time LTI state-space model. The compressed data matrix R from the preprocessor function `fcordom` is used to this end.

Inputs

`R`	A compressed data matrix containing information about the measured data, as well as information regarding the system dimensions.
`n`	The desired model order n.

Outputs

`A`	The state-space model's A matrix.
`C`	The state-space model's C matrix.

Algorithm

The data matrix obtained with `fcordom` contains the weighted left singular vectors of a matrix similar to the R_{22} matrix (see page 137). Unlike in the discrete-time case, the first n of these vectors do not form a direct estimate $\hat{\mathcal{O}}_s$ of the extended observability matrix. Rather, a generalized matrix $\hat{\mathcal{O}}_{s,\perp}$ is estimated because of the Forsythe-recursions in the data-compression step. The $\hat{A}$ and $\hat{C}$ estimates are extracted such that this generalized shift-structure is taken into account [1].

Used By

This is a top-level function that is used directly by the user.

See Also

`fcordom`, `fdmodom`

References

[1] R. Pintelon, "Frequency domain subspace system identification using non-parametric noise models," in *Proceedings of the 40th IEEE Conference on Decision and Control*, (Orlando, Florida), pp. 3916–3921, Dec. 2001.

Purpose

Estimates the A and C matrix in a discrete-time state-space model from frequency response function (FRF) data that was preprocessed by `fdordom`.

Syntax

```
[A,C]=fdmodom(R,n)
```

Description

This function estimates the A and C matrices corresponding to an nth order discrete-time LTI state-space model. The compressed data matrix R from the preprocessor function `fdordom` is used to this end.

Inputs

`R`	A compressed data matrix containing information about the measured data, as well as information regarding the system dimensions.
`n`	The desired model order n.

Outputs

`A`	The state-space model's A matrix.
`C`	The state-space model's C matrix.

Algorithm

The data matrix obtained with `fdordom` contains the weighted left singular vectors of the R_{22} matrix (see page 137). The first n of these vectors form an estimate $\hat{\mathcal{O}}_s$ of the system's extended observability matrix:

$$\mathcal{O}_s = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{s-1} \end{bmatrix}.$$

The estimates $\hat{A}$ and $\hat{C}$ are obtained by linear regression:

$$\hat{C} = \hat{\mathcal{O}}_s(1:\ell,:),$$
$$\hat{A} = \hat{\mathcal{O}}_s(1:(s-1)\ell,:)^{\dagger}\hat{\mathcal{O}}_s(\ell+1:s\ell,:).$$

Used By

This is a top-level function that is used directly by the user.

See Also

`fdordom`, `fcmodom`

Purpose

Preprocesses frequency-domain data for frequency-domain subspace identification of continuous-time state-space models.

Syntax

```
[S,R]=fcordom(H,w,s)
```

Description

This function performs the initial data compression for continuous-time subspace identification based on measured frequency reponse function (FRF) data. In addition, it delivers information usuable for determining the required model order. The model structure is the following:

$$\dot{x}(t) = Ax(t) + Bu(t),$$
$$y(t) = Cx(t) + Du(t).$$

This function acts as a preprocessor to `fcmodom`. Unlike in the discrete-time case, concatenating multiple data batches are not supported.

Inputs

`H`	The measured frequency response function (FRF). This should be a matrix which follows the convention of MATLAB 6; it should be $\ell \times m \times N$ in which `H(:,:,i)` contains the complex FRF at the i^{th} complex frequency.
`w`	Vector of complex frequencies at which the FRF is measured: $\mathtt{w} = j\omega.$
`s`	The block-size parameter. This scalar should be $> n$.

Outputs

`S`	The first s singular values of the rank-deficient R_{22} matrix (see below).
`R`	A compressed data matrix containing information about the measured data, as well as information regarding the system dimensions.

Remarks

The MEX-implementation may generate the following warning:

```
Cholesky-factorization failed; falling back on
QR-factorization.
```

This implies that the fast Cholesky-algorithm failed. The function has automatically fallen back onto a slower QR-algorithm. Results

from `fcordom` can be used without problems if this warning appears.

Algorithm

The continuous-time data compression algorithm in [1] is used. The same factorizations as in the discrete-time function `fdordom` on page 137 are used. However, the $\mathcal{W}$ and $\mathcal{G}$ matrices are formed by Forsythe-recursions to prevents ill-conditioning because the complex frequencies are not of unit magnitude [1, 2].

A weighted SVD of the R_{22} matrix is made, and its left singular vectors are appended to the `R`-matrix. Its first s singular values are returned in `S`.

Used By

This is a top-level function that is used directly by the user.

Uses Functions

LAPACK-functions `DPOTRF`, `DGEQRF`, `DGESVD`, `DTRTRS`.

BLAS-functions `DTRMM` and `DGEMM`.

(All built into the executable)

See Also

`fcmodom`, `fdordom`

References

[1] P. van Overschee and B. De Moor, "Continuous-time frequency domain subspace system identification," *Signal Processing*, vol. 52, no. 2, pp. 179–194, 1996.

[2] R. Pintelon, "Frequency domain subspace system identification using non-parametric noise models," in *Proceedings of the 40th IEEE Conference on Decision and Control*, (Orlando, Florida), pp. 3916–3921, Dec. 2001.

Purpose

Preprocesses frequency-domain data for frequency-domain subspace identification of discrete-time state-space models. Delivers an order-estimate.

Syntax

```
[S,R]=fdordom(H,w,s)
[S,R]=fdordom(H,w,s,Rold)
```

Description

This function performs the initial data compression for discrete-time subspace identification based on measured frequency reponse function (FRF) data. In addition, it delivers information usable for determining the required model order. The model structure is the following:

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k), \\ y(k) &= Cx(k) + Du(k). \end{aligned}$$

Several data batches can be concatenated, as shown below. This function acts as a preprocessor to `fdmodom`.

Inputs

`H`	The measured frequency response function (FRF). This should be a matrix which follows the convention of MATLAB 6; it should be $\ell \times m \times N$ in which `H(:,:,i)` contains the complex FRF at the i^{th} complex frequency.
`w`	Vector of complex frequencies at which the FRF is measured. $$\texttt{w} = e^{j\omega}.$$
`s`	The block-size parameter. This scalar should be $> n$.
`Rold`	(optional) The data-matrix resulting from a previous call to `fdordom`.

Outputs

`S`	The first s singular values of the rank-deficient R_{22} matrix (see below).
`R`	A compressed data matrix containing information about the measured data, as well as information regarding the system dimensions.

Remarks

The MEX-implementation may generate the following warning:

```
Cholesky-factorization failed; falling back on
QR-factorization.
```

This implies that the fast Cholesky-algorithm, as described in the algorithm section below, failed. The function has automatically fallen back onto a slower QR-algorithm. Results from `fdordom` can be used without problems if this warning appears.

Algorithm

The discrete-time data compression algorithm in [1] is used. In the M-file implementation, the following RQ-factorization is made:

$$\begin{bmatrix} \mathcal{W} \\ \mathcal{G} \end{bmatrix} = \begin{bmatrix} R_{11} & 0 \\ R_{21} & R_{22} \end{bmatrix} \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix}.$$

The meaning of the various matrices can be found in the cited article. In the MEX-implementation, the following Cholesky-factorization is attempted first:

$$\begin{bmatrix} \mathcal{W} \\ \mathcal{G} \end{bmatrix} \begin{bmatrix} \mathcal{W} \\ \mathcal{G} \end{bmatrix}^T = \begin{bmatrix} R_{11} & 0 \\ R_{21} & R_{22} \end{bmatrix} \begin{bmatrix} R_{11} & 0 \\ R_{21} & R_{22} \end{bmatrix}^T.$$

If this factorization fails, the algorithm falls back on the above RQ-factorization. In all cases, a weighted SVD of the R_{22} matrix is made, and its left singular vectors are appended to the R-matrix. Its first s singular values are returned in S.

Used By

This is a top-level function that is used directly by the user.

Uses Functions

LAPACK-functions `DPOTRF`, `DGEQRF`, `DGESVD`, `DTRTRS`.

BLAS-function `DTRMM`.

(All built into the executable)

See Also

`fdmodom`, `fcordom`

References

[1] T. McKelvey, H. Akçay, and L. Ljung, "Subspace-based multivariable system identification from frequency response data," *IEEE Transactions on Automatic Control*, vol. 41, pp. 960–979, July 1996.

Purpose

Calculates cost-function information for `foptlti`.

Syntax

```
[epsilon]=ffunlti(th,H,params,timing)
[epsilon,psi]=ffunlti(th,H,params,timing)
[epsilon,psi,U2]=ffunlti(th,H,params,timing)
```

Description

This function implements the costfuction for the `foptlti` frequency domain optimization framework. It is not meant for standalone use.

Inputs

`th`	Parameter vector describing the system.
`H`	The frequency response function of the system to be optimized: an array of size $\ell \times m \times N$ in which `H(:,:,i)` contains the complex FRF at the i^{th} complex frequency.
`w`	Complex frequencies at which the FRF is measured.
`params`	A structure that contains the dimension parameters of the system, such as the order, the number of inputs, whether D, x_0 or K is present in the model.
`timing`	Either `'cont'` or `'disc'`, indicating that the supplied model is continuous of discrete time. Note that this influences *only* the way in which the output normal parametrization is built. The user is responsible for supplying suitable frequency data.

Outputs

`epsilon`	Output of the costfunction, which is the square of the error between the actual and predicted vectorized frequency response function.
`psi`	Jacobian of `epsilon`.
`U2`	Left null-space of Manifold matrix for the full parametrization [1].

Algorithm

The formation of the error-vector is done bu calculating the FRF of the current model:

$$\hat{H}(\xi_k;\theta) = C(\theta)(\xi_k I_n - A(\theta))^{-1}B(\theta) + D(\theta).$$

The error-vector $E_N \in \mathbb{R}^{2N\ell m}$ is build up such that its ith blockrow consists of $\text{vec}(\hat{H}(\xi_i,\theta) - H(\xi_i))$, in which the real and imaginary components have been interleaved.

The Jacobian is formed efficiently by calculating FRFs as well. The formation of the Manifold matrix is performed according to [1]. A QR-factorization is used to obtain its left null-space.

Used By

`foptlti` (via `lmmore`)

Uses Functions

`dth2ss`, `cth2ss`, `ltifrf`

See Also

`dfunlti`

References

[1] L. H. Lee and K. Poolla, "Identification of linear parameter-varying systems using nonlinear programming," *Journal of Dynamic Systems, Measurement and Control*, vol. 121, pp. 71–78, Mar. 1999.

Purpose

Performs a frequency-domain nonlinear least squares optimization of an LTI state-space model.

Syntax

```
[A,B,C,D]=foptlti(H,w,A,B,C,D)
[A,B,C,D,options]=foptlti(H,w,A,B,C,D,model,partype,...
                          options)
```

Description

This function performs a nonlinear least squares optimization of a discrete or continuous time linear state space model based on frequency reponse data. The model structure is the following:

$$x(k+1) = Ax(k) + Bu(k),$$
$$y(k) = Cx(k) + Du(k).$$

First, the state space matrices are parameterized. The output normal parametrization, the tridiagonal parametrization and the full parametrization can be used.

The parameterized model is optimized using the supplied `lmmore` Levenberg-Marquardt function. The matrices A,B, C and D are returned.

Inputs

`H` The measured frequency response function (FRF). This should be a matrix which follows the convention of MATLAB 6; it should be $\ell \times m \times N$ in which `H(:,:,i)` contains the complex FRF at the i^{th} complex frequency.

`w` Vector of complex frequencies at which the FRF is measured. Although the function can operate using arbitrary complex frequencies, the following two choices are rather standard for discrete and continuous time models respectively:

$$\mathtt{w} = e^{j\omega},$$
$$\mathtt{w} = j\omega.$$

`A,B,C,D` Initial estimates of the system matrices A, B, C and D.

`partype` (optional) This parameter specifies the type of parameterization that is used to parameterize the state space model. Three types of parameterization are supported:

- `'on'` Output Normal parametrization.
- `'tr'` Tridiagonal parametrization.
- `'fl'` Full parametrization.

`options` (optional) Input parameters that are passed on directy to the optimization function. These options may be compatible with the `optimset` options from the MATLAB 6 Optimization Toolbox[1]. Alternatively, a MATLAB 5 Optimization Toolbox compatible `foptions` vector may be specified.
There are a number of fields in addition to the normal fields in the `options` structure. These are described in detail in the remarks section below.

`timing` Must be either `'cont'` or `'disc'` to specify that the model is continuous or discrete time. Note that this changes *only* the stability check and the output normal parametrization. It is up to the user to supply suitable frequency data.

Outputs

`A,B,C,D` System matrices of the optimized linear model. If the D matrix is not estimated, it will be returned empty.

`options` Output parameters from the Optimization Toolbox. See `foptions` or `optimset`.

Remarks

An extra field `options.Manifold` may be set to `'on'` if the full parametrization is used. The Manifold field indicates whether the search direction should be confined to directions in which the cost-function changes.

If `options.Manifold` is not set, `foptlti` will set it to `'off'` for the output normal and tridiagonal parametrizations, and to `'on'` for the full parametrization. See `foptions` or `optimset` for more information.

Another new field that can be set is the `options.BlockSize` field. The value N_b of the `BlockSize` field indicates that the Jacobian in the cost-function is build up N_b block-rows at a time rather than all at once [2]. This option is mainly interesting in tight-memory situations or for problems with a very large number of samples. If `options.BlockSize` is set to N_b, the fields `options.RFactor` and `options.LargeScale` are set to `'on'` automatically. A rule of thumb is that the Jacobian-calculation requires about $24(p + 1 + 2N_b\ell m)(p + 1)$ bytes of computer memory, in which p is the number of free parameters. For the full parametrization, this is the number of parameters *after* an optional Manifold-projection.

This optimization function has been targeted at MATLAB version 6 or higher. However, the function will run on MATLAB version 5 using a compatibility kludge. This kludge implies that the options

input parameter can either be a MATLAB 6 `optimset`-structure, or a MATLAB 5 compatible `foptions`-vector. However, the latter is discouraged since it does not allow the `Manifold`, `LargeScale`, `RFactor` and `BlockSize` fields to be set.

Used By

This is a top-level function that is used directly by the user.

Uses Functions

`lmmore`, `dss2th`, `dth2ss`, `css2th`, `cth2ss`, `ffunlti`

See Also

`lsqnonlin`, `lmmore`, `optimset`, `foptions`, `mkoptstruc`

References

[1] The MathWorks Inc., Natick, Massachusetts, *Optimization Toolbox User's Guide*, version 2.1 (release 12) ed., Sept. 2000.

[2] N. Bergboer, V. Verdult, and M. Verhaegen, "An efficient implementation of maximum likelihood identification of LTI state-space models by local gradient search," in *Proceedings of the 41st IEEE Conference on Decision and Control*, (Las Vegas, Nevada), Dec. 2002.

Purpose

Performs a Moré-Hebden Levenberg-Marquardt optimization

Syntax

```
x=lmmore('func',xinit,lb,ub,options,arg2,...)
[x,resnorm,residual,exitflag,output,lambda,
 jacobian]=lmmore('func',xinit,lb,ub,options,...
                  arg2,...)
```

Description

This function is a Moré-Hebden implementation of the Levenberg-Marquardt nonlinear least-squares optimization algorithm. The function is interface-compatible with the `lsqnonlin`-function from the MATLAB 6 Optimization Toolbox.

Inputs

`'func'`	The cost-function that is to be used.
`xinit`	The parameter-vector's starting point in the non-linear optimization.
`lb`	Lower-bound on the parameters. This value is *not* used.
`ub`	Upper-bound on the parameters. This value is *not* used.
`options`	A MATLAB 6 compatible `optimset`-structure that contains options for the optimization algorithm [1]. In addition, a number of extra fields may be present. See the Remarks section below for more information.
`arg2`	This will be passed as second argument to the cost-function `'func'`. Arguments 3 to N may be appended after `arg2`.

Outputs

`x`	Result of the optimization. The solution `x` is guaranteed to have an equal or smaller cost than `xinit`. All other parameters are compatible with the MATLAB 6 `lsqnonlin` function.

Remarks

The interface to lmmore has been made compatible with the `lsqnonlin` optimization function in the MATLAB 6 Optimization Toolbox. Note that although a lower and upper bound are given (consistent with `lsqnonlin`'s interface), they are *not* used internally.

This optimization implementation supports overparametrized cost-functions. If `options.Manifold` (not part of `optimset`'s normal structure) is passed and set to `'on'`, `lmmore` expects the cost- function to be able to return three arguments: an error-vector E_N, a Jaco-

bian $\Psi_N U_2$ and a projection matrix U_2. The columns of this matrix U_2 must form an orthonormal basis of the subspace in which the cost-function does not change because of over-parametrization.

This optimization implementation supports cost-functions that return the R-factor of the (projected) Jacobian Ψ_N and the error-vector E_N:

$$[\Psi_N \; E_N] = QR,$$
$$[\Psi_N U_2 \; E_N] = QR.$$

Cost-functions may use this functionality, e.g. to build up the R-factor in such a way that less memory is required. In order to use this feature with costfunctions that support it, the field `options.RFactor` should be set to `'on'`.

Algorithm

This function implements a Moré-Hebden trust-region based Levenberg-Marquardt optimization according to [2, chap. 10],[3].

In addition, this function supports projected gradients according to [4, 5].

Used By

`doptlti`, `foptlti`

Uses Functions

`dfunlti`, `ffunlti`

See Also

`lsqnonlin`, `optimset`

References

[1] The MathWorks Inc., Natick, Massachusetts, *Optimization Toolbox User's Guide*, version 2.1 (release 12) ed., Sept. 2000.

[2] J. E. Dennis and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. New Jersey: Prentice-Hall, 1983.

[3] J. J. Moré, "The Levenberg-Marquardt algorithm: Implementation and theory," in *Numerical Analysis* (G. A. Watson, ed.), vol. 630 of *Lecture Notes in Mathematics*, pp. 106–116, Springer Verlag, 1978.

[4] N. Bergboer, V. Verdult, and M. Verhaegen, "An efficient implementation of maximum likelihood identification of LTI state-space models by local gradient search," in *Proceedings of the 41st IEEE Conference on Decision and Control*, (Las Vegas, Nevada), Dec. 2002.

[5] L. H. Lee and K. Poolla, "Identification of linear parameter-varying systems using nonlinear programming," *Journal of Dynamic Systems, Measurement and Control*, vol. 121, pp. 71–78, Mar. 1999.

Purpose

Calculates an LTI state-trajectory.

Syntax

```
x=ltiitr(A,B,u,w,x0)
```

Description

In its most general setting, this function iterates the state equation of an linear time-invariant (LTI) system. It computes the state $x(k)$ for $k = 1, 2, \ldots, N$ satisfying the LTI state equation:

$$x(k+1) = Ax(k) + Bu(k) + w(k).$$

This function is used internally by `dfunlti`, `dac2b`, `dac2bd`, `dinit` and `dltisim`. It is not meant for stand-alone use.

Inputs

`A`	An LTI state-transition matrix of size $n \times n$
`B`	An LTI input matrix of size $n \times m$. If $s > 0$ and $B \in \mathbb{R}^{n \times m}$, the system is assumed to be Bilinear.
`p`	(optional) An $N \times s$ matrix containing the time varying parameters.
`u`	A $N \times m$ matrix containing N samples of the m inputs.
`w`	(optional) A $N \times n$ matrix containing the process noise.
`x0`	(optional) The initial state, an $n \times 1$ vector.

Outputs

`x`	The computed state, an $N \times n$ matrix.

Algorithm

A direct iteration of the system's state-transition equation is used to obtain the state-trajectory for all time-instants.

Used By

`dfunlti`, `dac2b`, `dac2bd`, `dinit`, `dltisim`

See Also

`dfunlti`, `dac2b`, `dac2bd`, `dinit`, `dltisim`

Purpose

Calculates an LTI Frequency Response Function

Syntax

```
H = ltifrf(A,B,C,[],[],w,outopt)
H = ltifrf(A,B,C,D,[],w,outopt)
H = ltifrf([],[],[],D,[],w,outopt)
H = ltifrf(A,B,C,[],dA,w,outopt)
```

Description

ltifrf will return the Frequency Response Function (FRF) of a linear time-invariant state-space model, evaluated at the complex frequencies provided in `w`:

$$H = C(\mathtt{w}I_n - A)^{-1}B + D.$$

This function is used internally by `ffunlti`, `fac2b` and `fac2bd`. It is not meant for stand-alone use.

Inputs

`A`	State-space model matrix A.
`B`	State-space model matrix B.
`C`	State-space model matrix C.
`D`	(optional) State-space model matrix D.
`dA`	(optional) Calculates the change in FRF given the deviation $\mathrm{d}A$ in A. D and $\mathrm{d}A$ are mutually exclusive.
`w`	Vector of complex frequencies. $e^{j\omega}$ for discrete-time systems and $j\omega$ for continuous-time systems.
`outopt`	Controls how H will be returned (see below).

Outputs

`H`	The FRF. Usually a 3D-array of size $\ell \times m \times N$. However, if `outopt` is non-empty and 1, H will be a vector of size $\ell m N \times 1$. If `outopt` is non-empty and 2, H will be a matrix of size $\ell \times mN$.

Algorithm

The state-space model is first transformed such that its state-transistion matrix A is in upper-Hessenberg form. The matrix $(\mathtt{w}I_n - A)^{-1}B$ is subsequently solved by an efficient upper-Hessenberg solver in SLICOT, after which premultiplication by C and addition of D yields the FRF. This approach follows [1].

If a deviation δA in A is given, the FRF deviation is given by:

$$\delta H = C(\mathtt{w}I_n - A)^{-1}\delta A(\mathtt{w}I_n - A)^{-1}B.$$

Again, the model is transformed so that A has upper-Hessenberg form, after which the SLICOT Hessenberg solver is used to obtain $(\omega I_n - A)^{-1}B$ and $(\omega I_n - A)^{-1}\delta A$. Multiplication then yeilds the FRF deviation.

Used By

`ffunlti`, `fac2b`, `fac2bd`.

Uses Functions

SLICOT-functions `MB02RZ`, `MB02SZ`, `TB05AD`.

LAPACK-functions `DGEHRD` and `DORMHR`.

(All built into the executable)

See Also

`ffunlti`, `fac2b`, `fac2bd`.

References

[1] A. J. Laub, "Efficient multivariable frequency response calculations," *IEEE Transactions on Automatic Control*, vol. 26, pp. 407–408, Apr. 1981.

Purpose

Creates a MATLAB 6-compatible `optimset`-structure

Syntax

```
optstruc=mkoptstruc
```

Description

This function provides a MATLAB 6 `optimset` work-alike. It generates an empty `optimset`-structure that can be passed to the high-level `doptlti` or `foptlti` function, or to the lower-level `lmmore` function.

Note that this function only generates a default `optimset`-structure. It is not capable to option-merging like the MATLAB 6 `optimset` function. It is recommended to use `optimset` if running MATLAB 6.

Inputs

`None`

Outputs

`optstruc`	A default `optimset`-structure

Used By

`optim5to6`

See Also

`optimset`

Purpose

Translates a `foptions`-vector into an `optimset`-structure.

Syntax

```
options=optim5to6(fopts)
```

Description

This function translates a MATLAB 5 optimization options vector — as generated using `foptions`— into a MATLAB 6 compatible `optimset`-structure. Translated fields are:

1 `Display`
2 `TolX`
3 `TolFun`
9 `Jacobian`
14 `MaxFunEval`

Inputs

`fopts`	A MATLAB 5 compatible `foptions`-vector

Outputs

`options`	A MATLAB 6 compatible `optimset`-structure

Remarks

MATLAB 5 uses a default parameter and function tolerance of 10^{-4}. This is indicated by the second and third element of `fopts`, that are 10^{-4} in the default case.

MATLAB 6 uses a default value of 10^{-6} for both tolerances, but setting the tolerances to 10^{-6} if the `fopts` vector contains the default values is impossible: there is no way of telling whether the user used the default values or that he actually specified 10^{-4} as tolerance.

Consequently, the tolerances are copied verbatim, *and there will thus be different results in an optimization when using a default* `foptions`*-vector or a default* `optimset`*-structure.*

Uses Functions

`mkoptstruc`

See Also

`foptions`, `optimset`, `mkoptstruc`

Purpose

Produces a pseudo-random binary sequency suitable as identification input.

Syntax

```
y=prbn(N,rate)
```

Description

This function produces a binary sequence, with values 0 and 1. The chance of switching from level is given by the parameter `rate`. `rate=0` yields a constant value 0. `rate=1` gives an signal that changes between 0 and 1 at every time-instant. Any value in between results in a random binary sequence. This kind of testsignal has been described in [1].

Inputs

`N`	The number of samples.
`rate`	(optional) Probability of the signal changing level at each time-instant. The default value is 0.5.

Outputs

`y`	Pseudo-random binary noise.

Used By

This is a top-level function that is used directly by the user.

References

[1] H. J. A. F. Tulleken, "Generalized binary noise test-signal concept for improved identification-experiment design," *Automatica*, vol. 26, no. 1, pp. 37–49, 1990.

Purpose

Calculates the left null-space of the basis of similarity transformations.

Syntax

```
U2=simlns(A,B,C,[],[],[])
U2=simlns(A,B,C,K,fD,fx)
```

Description

The function `simlns` calculates the left null-space of an LTI system's similarity map M_θ. In the most general case, when A, B, C, D, K and x_0 are part of the parameter vector, this matrix is given by [1]:

$$M_\theta = \begin{bmatrix} I_n \\ 0_{m\times n} \\ 0_{\ell\times n} \end{bmatrix} \otimes \begin{bmatrix} A \\ C \end{bmatrix} - \begin{bmatrix} A^T \\ B^T \\ K^T \end{bmatrix} \otimes \begin{bmatrix} I_n \\ 0_{\ell\times n} \end{bmatrix}. \tag{5.1}$$

A QR-factorization is used to obtain the left null-space.

This function is used internally by `dfunlti` and `ffunlti` and is not meant for stand-alone use.

Inputs

`A,B,C`	System matrices describing the LTI State Space system.
`K`	(optional) Kalman gain, specify as empty matrix when not present.
`fD`	(optional) specifies whether D is part of the parameter vector, specify as empty, 0 or 1.
`fx`	(optional) specifies whether x_0 is part of the parameter vector, specify as empty, 0 or 1.

Outputs

`U2`	The left null-space of the similarity map.

Remarks

Specifying `fx=1` only causes an $n \times n$ identify-matrix to be appended to the lower right of the left null-space matrix; in a non-linear optimization, applying the left null-space ensures that the state-basis does not change. It thus does not have to be projected.

Algorithm

The manifold matrix M_θ is calculated according to [1]. A QR-factorization is used subsequently to obtain the left null-space.

Used By

`dfunlti`, `ffunlti`

References

[1] L. H. Lee and K. Poolla, "Identification of linear parameter-varying systems using nonlinear programming," *Journal of Dynamic Systems, Measurement and Control*, vol. 121, pp. 71–78, Mar. 1999.

Purpose

Reduces spikes in measured signals.

Syntax

```
shave(x)
y=shave(x)
y=shave(x,factor,Wn,lolim,uplim)
```

Description

This function is used for reducing spikes in a measured signal. The spikes are shaved using the method in [1].

If no output argument is specified, a figure containing the original signal and shaved signal is drawn. The figure also contains the band (see "Algorithm" below). Detected spikes are indicated with crosses.

Inputs

`x`	The signal to be shaved.
`factor`	(optional) Multiplication factor which determines the width of the detection band. When the detection is poor, this factor should be changed. The default value is 2.
`Wn`	(optional) Cut-off frequency of the low-pass filter used for trend determination. It must be in the range $0.0 <$ `Wn` < 1.0, with 1.0 corresponding to half the sample rate. Its default value is 0.01.
`lolim,uplim`	(optional) The signal `x` will be clipped to the band $[$`lo`$_1$`im`$, $`up`$_1$`im`$]$ before the shaving starts.

Outputs

`y`	The shaved signal.

Algorithm

The spike removal algorithm developed in [1] is used. This algorithm can be summarized as follows:

- The trend in the signal `x` is calculated using a fourth-order Butterworth filter.
- The standard deviation of the trend-corrected, clipped signal is calculated.
- The detection band is defined by the signal trend plus and minus a certain factor times the standard deviation. All samples outside this band are regarded as spikes, and are replaced using linear interpolation.

Used By

This is a top-level function that is used directly by the user.

References

[1] A. Backx, *Identification of an Industrial Process: A Markov Parameter Approach*. PhD thesis, University of Eindhoven, Eindhoven, The Netherlands, 1987.

Purpose

Calculates the Variance Accounted For between two signals.

Syntax

```
v=vaf(y,ye)
```

Description

The function `vaf` calculates the Variance Accounted For between two signals. The VAF between y and $\hat{y}$ for the i^{th} component is defined as

$$\text{VAF}_i = \left(1 - \frac{\text{var}(y_i - \hat{y}_i)}{\text{var}(y_i)}\right) \cdot 100\%. \tag{5.2}$$

The VAF of two signals that are the same is 100%. If they differ, the VAF will be lower. If y and $\hat{y}$ have multiple columns, the VAF is calculated for every column in y and $\hat{y}$ seperately.

The VAF is often used to verify the correctness of a model, by comparing the real output with the estimated output of the model.

Inputs

`y`	The measured output $y(k)$.
`ye`	The estimated output $\hat{y}(k)$.

Outputs

`v`	The Variance Accounted For as defined above.

Used By

This is a top-level function that is used directly by the user.

0.4pt0.4pt

Index

MATLAB functions that are part of the toolbox software have both normal and bold page numbers. The bold page number refers to the function manual page. Normal page numbers refer to locations in the text where the function is used.