Akademie věd České republiky
Ústav teorie informace a automatizace

Academy of Sciences of the Czech Republic
Institute of Information Theory and Automation

RESEARCH REPORT

Nedoma P., Kárný M., Böhm J., Guy T.V. , Tesař L.

Mixtools

Interactive User's Guide

No. 2143 November 2005

Projects GA ČR 102/03/0049
and AV ČR S1075351

ÚTIA AV ČR, P. O. Box 18, 182 08 Prague, Czech Republic
Fax: (+420)286890378, http://www.utia.cz, E-mail: utia@utia.cz

ÚTIA AV ČR, P. O. Box 18, 182 08 Prague, Czech Republic
Fax: (+420)286890378, http://www.utia.cz, E-mail: utia@utia.cz
E-mail: utia@utia.cas.cz

This report constitutes an unrefereed software description. Any opinions and conclusions expressed in this report are those of the author(s) and do not necessarily represent the views of the Institute.

Contents

Chapter 1

Introduction

This guide deals with software aspects of learning with normal probability mixtures. References are made to design of probabilistic advisory system.

Underlying theory is described in [1]. A bridge between theory and software is in Section 2.

This guide is designed as an interactive one but, in can be used in this paper version. The interactive knobs provided are:

name of function lead to help on the function (without invoking MATLAB)

function argument lead to description of arguments

reference to [1] display section number and page in the book

references to .pdf files open them on a page

Run example button invokes .pdf description of a case study
the study allows to run an interactive MATLAB script with other knops

Mixtools Guide opens User's Guide e.g. on a commented case studies, etc

intro.tex, MK, PN November 3, 2005

Chapter 2

Bridge between theory and software

{scenario}}

Here, a bridge between the learning part of the underlying theory and its software image is presented. Theoretical background and toolbox are developed by the team whose communication is much simplified by adopting *common theoretical notation* and *coding agreements.* Often, this guide uses them without repeated explanation.

2.1 Common theoretical notation

The following common symbols used in the theoretical description [1] are useful here:

Symbol	Meaning
x^*	means the set of all values of x
$\mathring{x}$	denotes cardinality of x^*
x_t	is x at discrete time $t \in t^* \equiv \{1, \ldots, \mathring{t}\}$,
$x(t)$	means the sequence $x_1, \ldots, x_t$ and $x_{i;t}$ is i-th entry of x_t,
$\Theta \in \Theta^*$	denotes unknown model parameters,
$f(\cdot\vert\cdot)$	is a common symbol for conditional pdfs: versions are distinguished by identifiers in arguments,
$\propto$	means equality up to a normalizing factor.

2.2 Coding agreements

{coding}

The coding agreements are:

1. The *functions to be converted to MEX-files* are limited to use the following MATLAB entities:
   - (1-by-1) structure, called structure,
   - (1-by-n) cell list, called list,
   - 1 or 2-dimensional numerical array, called vector or matrix, respectively.
2. *Default values* have to be located in the function "defaults".
3. *Identifiers* have to meet the following basic rules:
   - *Global variables* have upper case identifiers. They should be eliminated from processing with the exception of global matrices (DATA , TIME , see Section 4.
   - *Structures and cell lists* begin with an upper case letter, other identifiers are coded by lower case letters;
   - *Identifiers* used should be selected from a common list given in subsections 17.3.
   - The *function names* have maximum length of 8 characters and consist of lower case letters and digits.

2.3 Basic learning scenario

A sequence $d(\mathring{t})$ of data records d_t is observed and mutual relationships are searched for. They are modeled by the joint pdf

$$f(d(\mathring{t})|\Theta) = \prod_{t\in t^*} f(d_t|d(t-1),\Theta)$$

conditioned on unknown parameters Θ. The considered *parameterized mixture* model has the form

{MKmixture}
$$f(d_t|d(t-1),\Theta) = \sum_{c\in c^*} \alpha_c f(d_t|d(t-1),\Theta_c,c). \quad (2.1)$$

The individual pdfs $f(d_t|d(t-1),\Theta_c,c)$ are called *parameterized components*. The unknown parameter Θ consists of probabilistic weights of components $\alpha \equiv (\alpha_1,\ldots,\alpha_{\mathring{c}}) \in \alpha^* \equiv \{\alpha_c \geq 0, \sum_{c\in c^*}\alpha_c = 1\}$ and by individual parameters Θ_c, $c \in c^*$, of components. The components are decomposed by the chain rule

{MKfactors}
$$f(d_t|d(t-1),\Theta_c,c) = \prod_{i\in i^*} f(d_{i;t}|\psi_{ic;t},\Theta_{ic},i,c), \quad (2.2)$$

where $f(d_{i;t}|\psi_{ic;t},\Theta_{ic},i,c)$ are called *parameterized factor*s. They predict scalar entries $d_{i;t}$ of d_t called *factor output*s. They are assumed to depend on regression vectors $\psi_{ic;t}$ that consist of current values of other record entries $d_{j;t}$, $j > i$ and several delayed record values d_{t-k}, $k \geq 1$. The factorization (2.2) allows us to combine entries of logical and continuous nature, to consider factors of different types.

The adopted Bayesian estimation modifies a chosen prior pdf $f(\Theta)$ by applying Bayes rule [2] in order get the posterior pdf $f(\Theta|d(\mathring{t}))$

{MKpostpdf}
$$f(\Theta|d(\mathring{t})) \propto f(d(\mathring{t})|\Theta)f(\Theta). \quad (2.3)$$

This pdf is the most general result of Bayesian estimation. From the software point of view, the estimation transforms data sample $d(\mathring{t})$ into the statistic $\mathcal{S} \equiv \mathcal{S}(d(\mathring{t}))$ that compresses information contained in historical data sample. It may serve for obtaining point estimates of the unknown Θ and information about precision of these estimates. Some of its parts serve for judging quality of the estimate. They are referred to as *states*. The collected statistics also serves for computing predictions

{MKpredict}
$$f(d|\psi,d(\mathring{t})) = \int f(d|\psi,\Theta)f(\Theta|d(\mathring{t}))\,d\Theta. \quad (2.4)$$

They have to be complemented by information that allows to select entries d to be predicted and construct the value of the regression vector ψ.

2.4 Basic scenario for design and advising

Learning provides multiple-mode model $f(d(\mathring{t}))$ of the managed system. Design modifies the elements of the model that are supposed under the operator control. The modification is designed so that the resulting ideal pdf ${}^{\lfloor I}f(d(\mathring{t}))$ is the closest to the user ideal pdf (user target) ${}^{\lfloor U}f(d(\mathring{t}))$ reflecting managing aims. Advising then reduces to presentation of properly selected low-dimensional projections of the designed ideal pdf. Three types of design are developed. *Academic design* optimizes pointers to recommended components, *industrial design* optimizes recommended recognisable actions and *simultaneous design* optimizes both pointers to recommended components and recommended recognisable actions. From here onwards, if not defined more precisely, all types of designs are meant under term *design*.

To evaluate the closeness of pair pdfs, *Kulback-Leibler divergence (KLD)* is employed. It can be expressed as additive loss function summing conditional KLD

{cKLD}
$$\omega(a_t,d(t-1)) \equiv \int {}^{\lfloor I}f(d_t|a_t,d(t-1)) \ln\left(\frac{{}^{\lfloor I}f(d_t|a_t,d(t-1))}{{}^{\lfloor U}f(d_t|a_t,d(t-1))}\right) dd_t, \quad (2.5)$$

where ${}^{\lfloor I}f(d_t|a_t, d(t-1))$ results from design. It is estimated model modified by advises a_t.

Advises, i.e. *actions available to p-system*s

{5pact} $$a_t \equiv (c_t, u_{o;t}, s_t, p_t) \text{ are interpreted as follows.} \tag{2.6}$$

Recommended pointers $\{c_t\}_{t\in t^*}$, $c_t \in c^* \equiv \{1, \ldots, \mathring{c}\}$, are pointers to the components that are recommended to be kept active at respective time moments.

Recommended pointers are *academic advises.*

Recommended recognizable actions $\{u_{o;t}\}_{t\in t^*}$ guide the user in selecting recognizable actions.

These advises result either from the industrial or from simultaneous design.

Priority actions $\{p_t\}_{t\in t^*}$ select entries of $\{d_t\}_{t\in t^*}$ to be shown to the operator.

These advises are called *assigning priorities.*

Signaling actions $\{s_t\}_{t\in t^*}$, $s_t \in s^* \equiv \{0, 1\}$, stimulate the operator to take some measures when behavior of the o-system significantly differs from the desired one.

These advises are called *signaling.*

2.5 Theory and its software images

The software entities inherit names of the underlying parameterized notions. For instance, a (software) factor represents relevant part of statistics describing its estimation together with information about type of the factor, structural information on modeled output, regression vector and possibly state of the estimation. The software entities serving for predictions are distinguished by prefix "p" whenever necessary. They have to contain information necessary for constructing of the current regression vector in addition to the information describing estimation results.

The theoretical entities are implemented as (software) structures or (1-by-n) cell lists referred to as *cell list*s or just *list*s.

The (software) structures contain a field *type*. It contains numerical code of the structure type. The *type=0* means "not specified".

The structures can have a field "states". It contains an auxiliary information needed for convenient processing of different tasks, e.g. statistics computed in mixture estimation. The content of states still varies so that their detailed description is postponed.

The most important relationships of the basic software entities and their theoretical counterparts are summarized in the following table:

Basic software entity	Software name (meaning) & representation	Theoretical counterpart
horizon	ndat, scalar	$\mathring{t}$
number of channels	nchn, scalar	$\mathring{d}$
data sample	DATA, (nchn,ndat) matrix	$d(\mathring{t})$
channel	row number DATA, scalar	index of $d_i(\mathring{t})$
mixture	Structure containing:	$f(\Theta)$ or $f(\Theta\|d(\mathring{t}))$
mixture type	type, scalar	code of the form and use of statistics $\mathcal{S}(d(\mathring{t})$
list of factors	Facs, cell list	labels of parameterized factors available
components	coms, (ncom,nchn) matrix	c-th row lists factors in c-th component $c \in c^* \equiv \{1, \ldots, \mathring{c} \equiv$ }ncom, i.e. $f(d_t\|d(t-1), \Theta_c, c) \equiv \prod_{i \in i^*}$ $f(d_{i;t}\|d_{i+1;t}, \ldots, d_{\mathring{i};t} d(t-1))$ *component weights*
degrees of freedom of components	dfcs, vector	statistics κ estimating component weights α, see Section 2.6
factor	Structure containing:	
factor output	ychn, scalar	index i of the modeled channel (of the factor output $d_{i;t}$
factor type	type, scalar	code distinguishes type of the factor (normal, Markov chain), form of statistics $\mathcal{S}$ (basic, least squares (*LS*) form (estimator or predictor)
factor structure	str, two-row matrix	it describes structure of regression vector; list j^* of channels $d_{j;t-k}$ in regressors; 1st row contains channel numbers j, the 2nd one their time delays $k \in k^*$ optional column [0; value] defines *factor offset* factor offset is θ_{ic} $\times$"value"
factor statistics	fields containing statistics typically "LD" matrix or vector "Eth",matrix "Cth", scalar "cove"	statistics $\mathcal{S}_i$; form is implied by the type: the first option basic, the second one LS
degrees of freedom of factor	dfm, scalar	degrees of freedom $\nu - 2$
regression vector	psi0, vector	description of regressor used in prediction
states	states, structure	it contains initial conditions, statistics used in tests . . . not stabilized yet

2.6 Dirichlet pdf for estimating mixture weights

{Dir}

Mixture weights form the probabilistic vector

$$\alpha \in \alpha^* \equiv \left\{ \alpha_c \geq 0, \sum_{c \in c^*} \alpha_c = 1 \right\}$$

They are universally described by the Dirichlet pdf

$$f(\alpha) \equiv Di_\alpha(\kappa) \propto \prod_{c \in c^*} \alpha_c^{\kappa_c - 1}. \tag{2.7}$$ {Diri}

This pdf is shaped by the $\mathring{c}$-vector statistic κ with positive entries κ_c. This prior form is preserved for all considered approximate estimations.

The vector κ is stored under the name "dfcs".

2.7 Normal parameterized factor and conjugate prior

The considered parameterized normal factors, called ARX factors (auto-regression with exogeneous signals), have the form

$$f(d|\psi, \Theta) = \mathcal{N}_d(\theta'\psi, r) = (2\pi r)^{-0.5} \exp\left\{ -\frac{1}{2r} \left([-1, \theta']\Psi\right)^2 \right\}, \text{ where} \tag{2.8}$$ {MKnor}

$'$ denotes transposition,
$\Theta = [\theta, r] = [\text{regression coefficients, noise variance}]$,
$\Psi = [d, \psi']' =$ [regressand, regression vector].

The factor output d is coded by the channel number "ychn" pointing to row of global data matrix "DATA", see Section 4. Structure of the regression vector is coded by the two-row vector "str".

The conjugate prior pdf $f(\Theta)$ that preserves its functional form during Bayes estimation of the model (2.8) is Gauss-inverse-Wishart (GiW) pdf [2]

$$f(\Theta) = GiW_{[\theta,r]}(L, D, \nu) \propto r^{-\frac{\nu}{2}} \exp\left\{ -\frac{1}{2r}[-1, \theta']L'DL[-1, \theta']' \right\}, \text{ where} \tag{2.9}$$ {MKGiW}

$\nu > 0$ is the number of degrees of freedom of $f(\Theta)$ that can be interpreted as an effective counter of number of data used; it is coded by "dfm"$=\nu - 2$,
$L'DL$ is an extended information matrix in numerically advantageous $L'DL$ decomposition in which L is lower triangular matrix with a unit diagonal,
D is diagonal matrix with positive entries.

Both matrices are stored in the matrix "LD", which coincides with L whose unit diagonal is replaced by the diagonal of D.

The split version of $L'DL$ decomposition

$$L \equiv \begin{bmatrix} 1 & 0 \\ L_{d\psi} & L_\psi \end{bmatrix}, D = \text{diag}[D_d, D_\psi], \ D_d \text{ is scalar} \tag{2.10}$$ {MKsplitLD}

can be unambiguously transformed into well known least squares (LS) quantities

$$\hat{\theta} = L_\psi^{-1} L_{d\psi} \text{ is LS estimate of } \theta, \text{ stored as "Eth"} \tag{2.11}$$ {MKLS}

$$\hat{r} = \frac{D_d}{\nu} \text{ is LS estimate of } r \text{ stored as "cove"} \tag{2.12}$$

$$\hat{r} L_\psi^{-1} D_\psi^{-1} (L'_\psi)^{-1} \text{ is covariance matrix of the LS estimate of } \theta$$

$$L'DL\text{decomposition of } L_\psi^{-1} D_\psi^{-1} (L'_\psi)^{-1} \text{ is stored as "Cth".}$$

Thus, *ARX factor* coincides with the description of the *GiW* pdf with the sufficient statistic $\mathcal{S}_{i;t} = [L_{i;t}, D_{i;t}, \nu_{i;t}]$. The factor is called *ARX LS factor* if the statistic $\mathcal{S}_{i;t} = [\hat{\theta}_{i;t}, \hat{r}_{i;t}, L^{-1}_{\psi i;t}, D^{-1}_{\psi i;t}, \nu_{i;t}]$ represents it.

For communication purposes, factors in single components are described in a common matrix way assuming that structure of their state. Then, matrix version

2.8 Prediction with normal parameterized factor and conjugate prior

{MKpredi}

The predictive (p-) factor – modeling *i*-th channel that corresponds to the normal parameterized factor and *GiW* factor given by the sufficient statistics $S = [L, D, \nu]$ – can be shown to have Student pdf [2] with moments

{MKstudent}
$$\hat{d}_i \quad = \quad \mathcal{E}[d_i|\psi, S] = \hat{\theta}'\psi,\ \hat{r}_d = \mathrm{cov}[d_i|\psi, S] = \hat{r}(1+\zeta),\ \zeta = \psi' L' D L \psi. \tag{2.13}$$

These moments together with degrees of freedom ν determine unambiguously the form of Student distribution.

Note that for a higher ν, Student distribution is well approximated by the normal pdf with above moments. In this case, it is also often possible to neglect the term ζ whose evaluation is computationally expensive.

In addition to statistics obtained in estimation, predictor has to store the value “psi0” of regression vector ψ used in its condition.

2.9 Conditional KL divergence

{coKLD}

Design with normal mixtures reduces to manipulations with conditional KL divergences that have common form of so-called *lifted quadratic forms*

{cKLDN}
$$k + \psi' L D L' \psi, \quad \text{where} \tag{2.14}$$

L is lower triangular matrix with unit diagonal and D is positive diagonal matrix. ψ_t is regression vector that reduces to the *state vector* $\phi'_{t-1} = [d'_{t-1}, \ldots, d'_{t-\partial}, 1], \partial \geq 0$ if there is no recognizable action in the problem. The lifted quadratic forms (2.14), used in the description of individual factors, components and its average counterpart, also describe approximate Bellman function.

User pf for recommended pointers is determined also in terms of a lifted quadratic form. For instance, in the academic design

{ufc}
$$^{\lfloor U}f(c_t|d(t-1)) \propto\ ^{\lfloor U}f(c_t) \exp\left[-0.5(\ ^{\lfloor U}k_{c_t;t-1} + \phi'_{t-1}\ ^{\lfloor U}L_{c_t;t}\ ^{\lfloor U}D_{c_t;t}\ ^{\lfloor U}L'_{c_t;t}\phi'_{t-1})\right], \tag{2.15}$$

where $^{\lfloor U}f(c_t)$ eliminates pointers to the components, operation on that may lead to wrong behaviour of the system (so-called *dangerous components*) while the used *KLD* in exponent of (2.15) defines preferences among pointers to components.

Chapter 3

Software representations

3.1 Software representation of mixtures

{represent}

The *basic software entities* are listed in Section 2 with relation to their mathematical counterpart. This Section partially repeats their description and extend them to (derived) software entities like matrix components or matrix mixtures.

The software entities are realized as structures and cell lists. The structures may have a field *states* that contains an auxiliary fields explained in relevant sections. The software entities are summarized and related to different types of normal ARX factors, components and mixtures. These forms are distinguished by field " *type*".

We are oriented on *dynamic mixtures* containing *dynamic factors*. Regression vector of dynamic factor contains some delayed values of the factor output or other channels. The *structure of regression vector* is coded by (factor) *structure*, i.e. by 2-rows matrix. The 1st row lists the involved channels and the 2nd one the corresponding time delays. For instance,

```
str = [1 1  2 2
       1 2  0 1]
```

means that the regression vector at a time t is composed of the data value on the channel 1 with delays 1 and 2 (it means DATA(1, t-1) and DATA(1, t-2)) together with the data value on the channel 2 with delays 0 and 1 (DATA(2, t) and DATA(1, t-1)).

Optionally, `str` may contain the column

```
[0; value]
```

that introduces *factor offset* and the *scaling* "value" (often 1).

The special case of *static mixtures* consisting of *static factors*. Their regression vectors contain at most zero-delayed values of other channels and the value multiplying the offset. Thus, no delayed data are considered.

3.1.1 Types related to normal ARX mixtures

Here, various types ARX mixtures are characterized. It has to be stressed that also mode of the use of software entities has to be respected, i.e. the entities related to estimation or prediction are distinguished by the "type" also.

Coding of estimation results

Estimation describes distribution of parameters, formally *GiW* pdf (2.9). Numerical values of various statistics are updated by data sample.

The *factors* are structures. They are coded according to their software representation (e.g. basic or LS ones):

1 ARX factor – corresponding to the form (2.9)
2 ARX LS factor – corresponding to the LS (least-squares) form (2.11)

Note that the value of "type" begins each line above.

A *component* is a list of factors. The factors listed can be of different forms. Then the component type code is 0. Special cases are supported if all the factors are of the same type:

11 ARX component – all factors are ARX factors, the form 1
12 ARX LS component – all factors are ARX factors, the form 2

If moreover all ARX factors have a common regression vector, the *matrix type* of components are also considered:

13 matrix ARX component – matrix version similar to ARX factor
14 matrix ARX LS component – matrix LS version similar to ARX LS factor but regression coefficients and noise covariance estimates are matrices.

A *mixture* is a structure. It is realized as a list of components together with *degrees of freedom of components*.

Mixture can contain components of different type – then the type code is 0. Special cases are supported if all components are of the same type:

21 ARX mixture
22 ARX LS mixture
23 matrix ARX mixture
24 matrix ARX LS mixture

Coding of prediction results

Prediction describes distribution of data, formally Student distribution with moments (2.13). It does not modify numerical values of the estimation statistics but exploits them for the current value of regression vector.

Prediction counter-parts of estimation results are given the same names. In text, if there is a danger of misunderstanding they are given prefix p-. So we have *p-factors*, *p-components* and *p-mixtures*. Codes of p-elements are obtained by adding 100 to codes of estimation counterparts. Thus, the following p-elements are considered:

101 ARX factor
102 ARX LS factor
111 ARX component
112 ARX LS component
113 matrix ARX component
114 matrix ARX LS component
121 ARX mixture
122 ARX LS mixture
123 matrix ARX mixture
124 matrix ARX LS mixture

The p-elements are obtained from corresponding estimation elements by mixture *projection* (marginalization, conditioning, regressor substitution), see Section 9. In the projection, the original states are changed.

3.1.2 Creating of mixture elements

Estimation elements (factors, components, mixture) are created by:

- *constructors* with fields filled by defaults (*default factor*,...) and overridden by user so that initial element (*initial factor*,...) arises;
- *conversions* from other existing form;
- *operations* from initial values through initialization, estimation etc. while processing data.

Prediction elements are created by:

- *projection* - transformation of estimation results while supplying information on predicted channels, channels in condition and their values, see Section 9;

- *conversions* from other existing p-forms.

3.1.3 Factors

The factors used in estimation are discussed. The corresponding p-factors are obtained from estimation factors by projection.

The factors are elaborated for a specific *modeled channel*. Their regression vectors are described by the *factor structure*. As *static factors* we refer to factors with modeled channel independent of delayed data. Its structure either contains offset or is empty.

The factors are structures built by constructors. A constructor creates factor with default values referred to as an *default factor*. The factor fields are filled later on by the user so that *initial factor* is obtained.

ARX factor

The ARX factor is described by (2.9). It is created by the constructor "facarx", e.g.

```
ychn   = 1;                            % modeled channel
str    = [1 1  2 2  0; 1 2  0 1  1];   % dynamic factor structure
Fac    = facarx(ychn, str)             % build ARX factor

Fac =
    ychn: 1                            − > modeled channel
     str: [2x5 double]                 − > factor structure
     dfm: 1                            − > degrees of freedom ν − 2
    type: 1                            − > type: ARX factor
      LD: [6x6 double]                 − > L'DL decomposition of extended inf. matrix
```

The "LD" field is the $L'DL$ decomposition of the *extended information matrix* introduced in (2.10), "dfm" is the field used for degrees of freedom $\nu - 2$. It represents the effective number of data items processed.

The "L" is a lower triangular matrix with units on diagonal. The diagonal matrix "D" is held on the "L" diagonal. The extended information matrix is $V = L'DL$.

ARX LS factor

The least squares representation (LS) of an ARX factor, *ARX LS factor*, deals with the LS form of the sufficient statistics (2.11). The factor is built by the constructor "facarxls":

```
ychn   = 1;                            % modeled channel
str    = [1 1  2 2  0; 1 2  0 1  1];   % dynamic factor structure
Fac    = facarxls(ychn, str)           % build ARX LS factor
Fac =
    ychn: 1                            − > modeled channel
     str: [2x5 double]                 − > dynamic factor structure
     dfm: 1                            − > degrees of freedom ν − 2
    type: 2                            − > type: ARX LS factor
    cove: 1.0000e-010                  − > LS estimate r̂ of noise variance
     Eth: [0 0 0 0 0]                  − > LS estimate θ̂ of regression coefficients
     Cth: [5x5 double]                 −   > LD  decomposition  of  LS  covariance
(L'DL)^-1
```

The covariance matrix "Cth" is held in the form of its $L'DL$ decomposition, i.e. the lower triangular "L" with its unit diagonal replaced by the diagonal of the matrix "D".

3.1.4 Components

A component describes parameter estimates related to multivariate pdf of selected channels. We refer to the selection as *modeled channels*. The distribution of modeled channels may be influenced by data measured on channels whose distribution is not modeled. These channels are introduced by the structures involved. We refer to them as *not-modeled channels.*

Components are of different forms described in subsections.

ARX components

As a basic form, the component is expressed as a list of individual factors. This form is used in estimation.

The list of factors should be ordered according to mutual dependencies but the Mixtools functions do not require to specify the correct order of factors – the sorting is done internally if needed be.

ARX LS components

This component consists of ARX LS factors only. This type (converted to predictor) is used in simulation.

Matrix ARX components

The estimated parameterized component is a multivariate normal pdf that predicts the modeled channels by a multivariate ARX model with a common regression vector. It and its estimates can be written in the form similar to ARX factor.

The matrix ARX component has "nchn" modeled channels. The common length of the regression vector is "npsi". The ARX component is then described by the fields:

```
ychns  (1-by-nchn)      % ordered list of modeled channels: d_{i;t} depends on d_{i+1;t}, ..., d_{i̊;t}
str    (2-by-npsi)      % regression-vector structure common for all factors
dfm    (1-by-1)         % degrees of freedom ν − 2
LD     (nLD-by-nLD)     % L'DL decomposition of the extended information matrix, size nchn+npsi
```

The matrix ARX component is built by the constructor "comarx", e.g.

```
ychns  = [3 2 1];                        % modeled channels
str    = [1 1  2 2  0; 1 2  1 2  1];     % common regressor structure
Com    = comarx(ychns, str)              % build matrix ARX component
Com =
    ychns: [3 2 1]                       −> modeled channels
      str: [2x5 double]                  −> component structure
      dfm: 1                             −> component degrees of freedom
     type: 13                            −> component type, matrix ARX
       LD: [8x8 double]                  −> LD decomposition of extended inf. matrix
```

Matrix ARX LS component

The estimated parameterized component is multivariate normal pdf that describes the modeled channels by a multivariate ARX model.

It has a common regression vector and it is written in the form mimic to ARX LS factor. The estimated regression coefficients and noise covariance only become matrices. The component structure does not contain zero delays of the modeled channels - those dependencies are respected by non-diagonal covariance whose estimate is non-diagonal matrix "cove".

This type of components is employed mainly in the problem formulation and interpretation of results.

The matrix ARX LS component has "nchn" modeled channels. The common length of the regression vector is "npsi". The ARX component is then described by the fields:

`ychns`	`(1-by-nchn)`	% list of modeled channels	ordered
`str`	`(2-by-npsi)`	% regression-vector structure	common one
`dfm`	`(1-by-1)`	% degrees of freedom of a factor	
`Eth`	`(nchn-by-npsi)`	% point estimate of regression coefficients	matrix $\mathcal{E}[\theta\|L, D, \nu]$
`Cth`	`(npsi-by-npsi)`	% covariance of regression coefficients	the same as for single modeled chanel $L'DL$ version stored
`cove`	`(nchn-by-nchn)`	point estimate of noise covariance	matrix $\mathcal{E}[r\|L, D, \nu]$ $L'DL$ version stored

The matrix ARX LS component is created by the constructor "comarxls" e.g.

```
ychns  = [3 2 1];                        % modeled channels
str    = [1 1  2 2  0; 1 2  1 2  1];     % common regressor structure
Com    = comarxls(ychns, str)            % build matrix ARX LD component
Com =
    ychns: [3 2 1]                       −> modeled channels
      str: [2x5 double]                  −> component structure
      dfm: 1                             −> component degree of freedom
     type: 14                            −> component type, matrix ARX LS
     cove: [3x3 double]                  −> point estimate of noise covariance
      Eth: [3x5 double]                  −> point estimate of regression coefficients
      Cth: [5x5 double]                  −> covariance of regression coefficients
```

The covariance matrix "Cth" and the point estimate of noise variance "cove" are held in the form of its $L'DL$ decomposition introduced in (2.10), i.e. the lower triangular "L" with its unit diagonal replaced by the diagonal of the matrix "D".

The field "dfm" holds degrees of freedom $\nu - 2$. It represents the effective number of data items processed.

3.1.5 Mixtures

A Mixtools *mixture* is formed by an *array of components* and *degrees of freedom of components.*

The degrees of freedom of components "dfcs", equal to κ in (2.7), are proportional to point estimates of the mixing probabilities defining the mixture weights (α). They also determine uncertainty of these estimates. The attempt to fix these estimate sufficiently in mixture estimation led us to the recommended initial values of "dfcs" to be close to 10 % of the data sample length.

The mixture is build by *mixture constructor* in the form:

```
Mix = mixconst(Facs, coms, dfcs)        % forms 21 22
Mix = mixconst(Coms, dfcs)              % forms 23 24
```

The first possibility is explained in the next subsection. The second one is equivalent.

The list of components "Coms" can have different forms. The components must have the same selection of the modeled channels.

The constructor analyzes the components, specifies the mixture "type" and writes a descriptive information into the field "states".

ARX mixture – basic estimation form

The ARX mixture is based on an *array of factors* "Facs". Each factor is represented by its position in the array – by an integer index. A component lists its factors as integers pointing to "Facs". The array of components is then a matrix where each row represents a component. It has the dimension ncom-by-nchn where "ncom" is the number of components and "nchn" is the number of modeled channels.

In texts and examples, we use the term *estimator* for this special mixture form in order to stress its dominant use.
Notes:

- a factor can be used by more than one component – in this case we speak about the *common factor*
- the field of factors ”Facs” may contain factors that are not included in any considered component
- the factors define the modeled and not-modeled channels of the mixture.

  The non-modeled channels are used factor structures but they are not listed among the modeled channels.

The ARX mixture is build by the constructor ”mixconst” with 3 arguments:

```
Mix = mixconst(Facs, coms, dfcs)
```

The ARX mixture estimator was designed with respect to easy estimation. It represents the only mean how to specify and support common factors.

An example of a mixture estimator building follows. The mixture has two components. The components contain the dynamic factors Fac1 and Fac2 for the 1st channel and a common static factor Fac4. The Fac1 and Fac2 depend on 1st and 2nd modeled channels and on the not-modeled channel 4. The factor Fac3 is not used in processing.
The diary of building the mixture:

```
Facs{1} = facarx(1,[1 1 2 2 4; 1 2 0 1 0]); % build 1st ARX factor
Facs{2} = facarx(1,[1   2  ; 1   0]);       % build 2nd ARX factor
Facs{3} = facarx(2,[1 0; 1 1]);             % build 3rd ARX factor (not used)
Facs{4} = facarx(2, []);                    % build 4th ARX factor
coms    = [1 4; 2 4];                       % build components
dfcs    = [10 40];                          % degrees of freedom of components
[Mix, maxtd] = mixconst(Facs, coms, dfcs);  % build mixture
maxtd
maxtd =
     2                                      -> maximum time delay in the mixture
```

The mixture consists of the following fields:

```
Mix
Mix =
      Facs: {[1x1 struct]  [1x1 struct]  [1x1 struct]  [1x1 struct]}
      coms: [2x2 double]               -> description of components
      dfcs: [10 40]                    -> degrees of freedom of components
      type: 21                         -> mixture type: ARX mixture
    states: [1x1 struct]               -> states for estimation
```

ARX LS mixture

In the same way, ARX LS mixture is build. The only difference is that the factors used are ARX LS factors. This form of mixture is used for simulation.

Matrix ARX mixture

The components are specified as a list of matrix ARX components.

We use this forms when we gain no advantages from use of the corresponding factorized form.

Example: 3 matrix ARX LS components ”Com1, Com2, Com3” and a ”dfcs” are supposed to be available. The mixture is build as:

```
dfcs = [10 40 20];
Mixc = mixconst({Com1 Com2 Com3}, dfcs)
Mixc =
      Coms: {[1x1 struct]  [1x1 struct]  [1x1 struct]}
      dfcs: [10 40 20]                   -> degrees of freedom of components
      type: 24                           -> mixture type: matrix ARX mixture
    states: [1x1 struct]                 -> states
```

Matrix ARX LS mixture

This form is similar to matrix ARX mixture but the components are specified as a list of matrix ARX LS components.

Summary of coding

{codingsum}

1	ARX factor
2	ARX LS factor
11	ARX component
12	ARX LS component
13	matrix ARX component
14	matrix ARX LS component
21	ARX mixture
22	ARX LS mixture
23	matrix ARX mixture
24	matrix ARX LS mixture
+100	predictor types

3.1.6 Conversions

There are 2 functions for conversion into any specified form:

```
Com = com2com(Com, type) % convert component to the type specified
Mix = mix2mix(Mix, type) % convert mixture to the type specified
```

where "type" is coded element type.

Use of "mix2mix" is documented on an example. Let us have a ARX mixture estimator "Mix" . First, marginalization by the function "mix2mixm", see Section 9, is performed. By this, mixture is converted to mixture predictor:

```
pMix    = mix2mixm(Mix)                  % build p-ARX LS mixture (predictor)
      Facs: {[1x1 struct]  [1x1 struct]  [1x1 struct]  [1x1 struct]}
      coms: [2x2 double]                 -> description of components
      dfcs: [0.3000 0.7000]              -> degrees of freedom of components
      type: 122                          -> mixture type: ARX LS predictor
    states: [1x1 struct]                 -> states for prediction
```

Then, the p-mixture "pMix" is converted to p-ARX LS mixture:

```
pMix    = mix2mix(pMix, 124)             % p-matrix ARX LS mixture
pMix =
        Coms: {[1x1 struct]  [1x1 struct]} -> mixture components
        dfcs: [0.3000 0.7000]              -> degrees of freedom of components
    reserved: 0                            -> for later use
        type: 124                          -> mixture type: matrix ARX LS prediction
      states: [1x1 struct]
```

3.2 Software representation of advisory mixtures

{representa}

Mixtools implements algorithms that transform user's aims ${}^{\lfloor U}f(d(\mathring{t}))$ and data d into the ideal mixtures ${}^{\lfloor I}f(d(\mathring{t}))$ that are presented to the user. Mixtures are learned from the data [3]. The estimated mixtures are converted into predictors, namely, p-mixtures. They contain information necessary for constructing of the current regression vector in addition to the information from estimation. Design converts predictors (p-mixtures) and management aims (expressed by one component mixture) into advisory type mixture (a-mixtures). In addition to the information describing estimation results, a-mixtures store description of user's aims and states related to the design. Thus, a-mixtures represent a slight extension of p-mixtures, so that majority of notions related to p-mixtures is preserved.

Advise describes the ideal pdf of data, which is constructed such that, if followed, the system behaviour be close to the user target (given by the user ideal pdf ${}^{\lfloor U}f(d)$).

The advising results are represented by *a-mixture* which is similar to p-mixture, but have additional states (*advisory states*) used in design of advises. Mixture *projection* (marginalization, conditioning, regression vector substitution, see Section prediction) operations can be applied to a-mixture as well.

Design converts result of learning (e-mixture) into predictor (p-mixture) and then into advisory mixture *a-mixture*. The last is used for advising and represents the ideal pdf. Basic information stored in individual factors and components are identical with those of corresponding p-mixture, except of: i) field dfcs, which contains probabilistic weights of components α gained from advisory design and ii) *advisory states* used in advises design. Thus the state of a-mixtures contains the following fields:

- `strc` - common structure of data vectors used in design
- `ufc` - vector qualifying components: dangerous component (0), not dangerous (positive number)
- `kc` - user lifts of quadratic forms
- `UDc` - cell vector of $U'DU$ decompositions of the user KLD kernels
- `udca` - $U'DU$ decomposition of the average KLD kernel made of UDc
- `kca` - average lift of quadratic forms made of kc
- `outs` - list of channels with innovations
- `uchn` - list of channels with recognisably actions
- `pochn` - list of channels with o-innovations

Beside that, a new factor state Mixc.Facs$\{\cdot\}$.states.pEth is defined. This state is a pointer table enabling expanding of Facs$\{\cdot\}$.Eth to a common structure strc used by a-mixture.

The only way to get a-mixture is to construct it from the estimated mixture and user target by using *mixture constructor* "inisyn". The function "inisyn" is called in the following way:

```
[aMix,aMixu] = inisyn(Mix,Mixu,Chns)    % converts Mix and Mixu to advisory type
```

or

```
[aMix,aMixu] = inisyn(Mix,Mixu,pochn,uchn)
```

The arguments of the function are:

aMix	constructed a-mixture
aMixu	user target Mixu, converted to advisory type
Mix	learnt ARX mixture
Mixu	user target (one component ARX mixture)
Chns	cell vector with channels descriptions
pochn	list of channels with o-innovations
uchn	list of channels with recognisably actions (can be omitted for academic design).

Chapter 4

Data management

{dataman}

Management of data samples is discussed.

4.1 Access to data sample

The Mixtools uses simple data management based on two global matrices:

- DATA – data sample
- TIME - processing "time"

The DATA matrix must be allocated before processing (mixture simulation and/or estimation) starts. Data vectors should not be accessed directly, but via an interface function "getdvect". In such a way, nothing needs to be changed when the file management changes (e.g. the processing is done outside MATLAB).

The function "getdvect" returns data vector or regression vector according to the form how it is called:

Get data or regression vector	
`psi = getdvect(str)`	get regression vector
`Psi = getdvect(Fac)`	get data vector

Note 1: data vector is the regression vector preceded by the current data value on the modelled channel.

Note 2: the "TIME" must be specified greater then the maximum time delay "maxtd" in the relevant structure.

4.2 Huge data sample processing

{huge}

When we process an extremely *huge data sample*, we are forced to use *buffered estimation* because such data sample is hardly manageable in the MATLAB workspace. This possibility is available with any estimation function.
The data matrix is held on disk in similar form as in MATALB workspace. Instead of the usual argument `ndat`, the cell list

```
Ndat = {'filename',mdat}                  % argument "ndat" replacement
```

is used. Here, 'filename' is the name of file where data are stored and "mdat" specifies the number of rows. The global matrix DATA is then used as a buffer for the buffered estimation. The matrix DATA can have any reasonable number of columns.

data.tex by PN November 3, 2005

Chapter 5

Data preprocessing

{preprocessing}

The data sample should be preprocessed for subsequent data analysis. It is done either as *batch data preprocessing* or *recursive data preprocessing*.

Data preprocessing implies the necessity of *backwards transformation* of processing results to the user's levels.

5.1 Preprocessing requirements

The *preprocessing requirements* are encoded as a *cell list* - a cell vector consisting of pairs of cells. The cells carry the information:

- the first one is a character string that identifies the preprocessing *operation* to be done;
- the second one is a matrix that contains the quantitative values (often again a cell list) needed for performing this operation. This matrix is referred to as operation *"parameters"*.

The preprocessing operations are carried out in the order of operations defined by the preprocessing requirements.

The processing requirements are modified during preprocessing - they usually contain "states". The modified preprocessing cell list is referred to as *run-time preprocessing requirements*. This list is obtained by *initialization of preprocessing*. In the batch data preprocessing, it is modified internally and a final state is returned.

The preprocessed data sampleare located in the global matrix "DATA".

The identifier used for the preprocessing list is pre.

The preprocessing operations are:

Data preprocessing	
`pre = preproc(pre)`	batch preprocess data
`pre = preinit(pre)`	initialize preprocessing
`pre = prestep(pre)`	preprocessing step
`out = filoutm(ord)`	outlier filtration by mixture estimation

The recursive preprocessing consists of two steps. The first one is calling *preiniti* and then the *prestep* is to be called in the cycle of "time". Note that the the run-time preprocessing list can be modified at each processing step.

5.2 Preprocessing algorithms

Algorithms available and corresponding list of (preprocessing) requirements are discussed.

The channels that are accessed by an algorithm are introduced by the construct

```
{ 'c', [channels]}
```

This construct can appear among requirements or among specifications. If it appears among requirements, they are interpreted as default set of channels that is valid till a next specification. At the beginning of processing, the default channels are all channels.

If the channels are defined in the specification, the set of channels is used only for the current operation.

The frequently used preprocessing fast algorithms are

option	meaning	parameters
limit	limit signal	[minimum, maximum] or {'limits', [minimum; maximum]}
scale	scale signal	[add; mult] or {'scaling', [add; mult]}
	Re - sampling	
group	re-sample by group data	extent of data grouping
lsfi0	re-sample by constant fit over a window	[window_size]
lsfit	re-sample by least squares fited line	[window_size]

Description:

`limit`

If the data value is outside limits, the value of the limit violated is substituted. Use of inf or $-inf$ is possible in the parameters. The possible forms of the requirement:

```
1) pre = {'limit', [-1;+1] }
2) pre = {'limit', {'c', [1 3], 'limits', [-1; 1] } }
3) pre = {{'c', [1 3], 'limit', [-1; 1] } }
4) pre = {'limit', {'c', [1 3], 'limits', [-1 -2; 1 2] } }
5) pre = {'limit', {'c', 1, 'limits', [-inf; 5] } };
```

The meaning is:
1) the same limit for all channels;
2) the same limit for the channels specified;
3) change of default channels, the limits as in 2);
4) different limits for the channels specified;
5) only upper limit for the 1st channel.

`scale`

The specification consists of a column vector of 2 elements. The 1st line is added to signal, the result is multiplied by the 2nd row of the vector. The specification can be empty see below - then the data are *normalized* through the sample moments. If more channels are defined, the specification contains corresponding number of columns.

The possible forms of the requirement:

```
1) pre1  ={'c', 2, 'scale', [] }
2) pre2  ={'c', 2, 'scale', [-mean(DATA(2,:)); 1/std(DATA(2,:))]}
3) pre = {'scale', [] }
4) pre = { 'scale', {'c', [1, 3] } }
5) pre = { 'scale', {'c', 1, 'scaling', [10; 2] } }
6) pre = { 'scale', {'scaling', [0.1,0.2 0.3; 1.1 1.2 1.3] } }
```

The meaning is:
1, 2) the requirements are identical;
3) all channels are normalized;
4) channels 1,3 are normalized;
5) general form - 10 is added to the channel 1 data and then multiplied by 2;
6) for more channels, the scalling is matrix.

`group`
An example of data grouping:

```
DATA = [1:10; 11:20]
DATA =
     1     2     3     4     5     6     7     8     9    10
    11    12    13    14    15    16    17    18    19    20
nsk  = 2;                           % extent of data grouping
pre  = { 'group', nsk};             % preprocessing requirement
pre  = preproc(pre);                % preprocessing
DATA
DATA =
     2     4     6     8    10
    12    14    16    18    20
     1     3     5     7     9
    11    13    15    17    19
```

`lsfit0`
Data are re-sampled by a constant fitting. The original samples within the window of the specified length are replaced by a single value equal to the average of the processed samples.

`lsfit`
Data are re-sampled by fitting least-squares straight line. The original samples within the window of the specified length are replaced by a single value equal to the end point of the fitted line.

5.3 Filters

Preprocessing algorithms are described in [4].
The following selected filters are discussed here.

`olymean, olymedian, olymeanf, olymedianf`
serve as outlier removal filters. They preserve majority of data and substitute a new value (mean or median based) the outlier is detected.

`'mean' 'median' 'meanf' 'medianf'`
are mean, median, and forgetting based filters itself. All data are influenced by these filters, no re-sampling is done.

The algorithms use the arguments:

`‘c’` describes channels to be preprocessed by this algorithm (vector);

`startup_period` is scalar to adjust the initial period of algorithm, where the `DATA` matrix is not modified at all, to allow clean startup of the algorithm;

`window_size` determines size of the window for `olymean, olymedian, mean` and `median` algorithms;

`forgetting_rate` is proportion of information from current data item and filtered data, for `olymeanf, olymedianf, meanf` and `medianf` algorithms;

`level` is threshold for the amplitude of outlier and the standard deviation of underlying signal noise;

`m0` is the initial value of mean/median for forgetting-based algorithms;

`s0` is the initial value of standard deviation for forgetting-based algorithms.

Short description of algorithms:

option	meaning
olymean	mean outlier removal filter (window based)
olymedian	median outlier removal filter (window based)
olymeanf	mean outlier removal filter (forgetting based)
olymedianf	median outlier removal fiter (forgetting based)
mean	simple mean filter (window based)
median	simple median filter (window based)
meanf	simple mean filter (forgetting based)
medianf	simple median filter (forgetting based)

Input parameters c and startup_period are used by all filters, default values are substituted if they are not specified. Other parameters needed for respective algorithms are:

option	input parameters
olymean	window_size, level, m0, s0
olymedian	window_size level m0 s0
olymeanf	forgetting_rate level m0 s0
olymedianf	forgetting_rate level m0 s0 importance_threshold
mean	window_size m0 s0
median	window_size m0 s0
meanf	forgetting_rate m0 s0
medianf	forgetting_rate m0 s0 importance_threshold

Remark to filter usage The temporal correlation is introduced into the data if filters `mean`, `median`, `meanf`, and `medianf` are used. It may cause troubles in the closed control or advising loop.

The outlier removal algorithms influence only data detected as outliers, thus correlate only a very few data items. It should not make any damage compared to the negative influence of outliers itself, thus benefiting in the closed control loop enormously.

5.4 Case studies

Diaries of case studies are in Mixtools Guide

preproc.tex, zpre2.m zpre3.m PN, November 3, 2005

Chapter 6

Mixtools functions

{functions}

Mixtools functions can be roughly divided in the group of *Mixtools user's functions* that are used by an ordinary designer user). The rest of the Mixtools auxiliary functions form a *Mixtools design base.*

The user's functions can be used for *batch and recursive processing.* When applicable, the versions are distinguished by presence or absence of the argument "ndat".

The data sample must be located in the global matrix "DATA", the processing time is controlled by the global scalar "TIME".

Some functions can be run in debugging mode. It is controlled by the global matrix DEBUG used for debugging prints, plots etc. If it is set to zero, no information is displayed. A positive value leads to the information display according to the function design.

6.1 Function arguments

The list of all arguments can be found in Appendix A.

The following arguments are used in learning and prediction.

`Com`	component
`Coms`	array of components
`Fac`	factor
`Mix`	mixture
`Mix0`	initial mixture
`Sim`	mixture simulator
`cchns`	channels in condition
`coms`	array of components
`dfcs`	vector of degrees of freedom of components
`frg`	forgetting rate
`ndat`	length of data
`niter`	number of iterations
`opt`	processing options
`pMix`	mixture predictor
`pchns`	predicted channels
`pre`	preprocessing requirements
`psi0`	value of zero-delayed regressor
`str`	structure of regression vector
`ychn`	modeled channel
`ychns`	modeled channels in component

The following arguments are used in design of advisory system.

`aMix`	advised mixture of the type ARX LS + control states
`aMixu`	desired mixture of the type ARX LS + control states
`strc`	common control structure
`ufc`	normalized vector qualifying components
`kc`	lift of quadratic forms
`UDc`	cell vector of u'du decompositions of KLD kernels
`udca`	u'du decomposition of average KLD kernel in UDc
`kca`	average lift of quadratic forms kc
`strc`	common control structure
`uchn`	list of channels with recognisably actions
`pochn`	list of channels with o-innovations
`outs`	list of channels with innovations
`npochn`	number of channels with o-innovations
`chis`	strategy of control design

6.2 Mixtools user's functions

This subsection summarizes the Mixtools user's functions in the form of the functions prototypes.

Constructors

`Fac = facarx(ychn, str)`	build ARX factor
`Fac = facarxls(ychn, str)`	build ARX LS factor
`Com = comarxls(ychns, str)`	build matrix ARX LS component
`Com = comarx(ychns, str)`	build matrix ARX component
`Mix = mixconst(Facs, coms, dfcs)`	build ARX or ARX LS mixture
`Mix = mixconst(Coms, dfcs)`	build mixture of any type

Initialization of estimation

`Mix = ... mixinit(Mix0,frg,ndat,niter,opt,belief)`	initialization of mixture estimation
`Mix = comdel(Mix, com)`	cancel specified component
`Mix = commerge(Mix, Mix0, com)`	merge mixture components
`Mix = mixcut(Mix)`	cancel components that explain low amount of data
`Mix0 = ... genmixe(ncom,ychns,str,ndat)`	generate initial mixture

Estimation operations

Mix = ... mixest(Mix0, frg, niter, opt)	iterative mixture estimation
Mix = mixestim(Mix0, frg, ndat)	quasi-Bayes mixture estimation
Mix = mixestim(Mix0, frg)	recursive quasi-Bayes mixture estimation
Mix = mixestimp(Mix0, frg, ndat)	projection based quasi-Bayes estimation
Mix = mixestimp(Mix0, frg)	projection based recursive quasi-Bayes estimation
Mix0 = mixflat(Mix)	mixture flattening
Mix = mixstats(Mix, ndat)	compute estimation statistics
Mix = mixstats(Mix)	compute statistics recursively
Mix0 = genmixe(ncom, ychns, str)	generate initial mixture for estimation
[Mix,tstop,Qs] = ... mixestims(Mix, frg, ndat, Mixa,thr)	quasi-Bayes estimation of ARX mixture with stopping
[Mix,tstop, Qs] = ... mixestimps(Mix, frg, ndat, Mixa,thr)	projection based estimation with stopping
[Mix, tstops] =... mixestpbs(Mix0,frg,ndat,niter,thr)	projection based estimation with stopping
[Mix, tstops] = mixestqbs(Mix0,frg,ndat,niter,thr)	repetitive quasi-Bayes estimation with stopping
[Mix, tstops] =... mixestpbs(Mix0,frg,ndat,niter,thr)	repetitive projection based estimation with stopping
[Mix, tstops] =... mixestqbs(Mix0,frg,ndat,niter,thr)	repetitive quasi-Bayes estimation with stopping
[Mix,frg,mixlls,frgs] = estfrg(Mix0,frgs,ndat,niter,method,Mixa)	estimate forgetting rate

Prediction operations

pMix = mix2mixm(Mix, pchns)	mixture flattening
pMix = mix2pro(Mix, pchns, cchns)	mixture flattening
pMix = profix(pMix, psi0, pre)	build mixture prediction from projector
pMix = ... mixpro(Mix0,pchns,cchns,psi0,pre)	build mixture projection
[pMix, weights] = ... profixn(pMix, psi0, pre, nstep)	build mixture projection
[Eths, coves, scales] = ... profixna(pMix, psi0, pre, nstep)	alternative prediction n-steps ahead (by fixing data)
Eth = getth(Eth)	parameter estimates from profix outputs

Visualization

`mixplot (Mix,pchns,cchns,psi0,pre)`	mixture plot (shaded)
`mixplotc(Mix,pchns,cchns,psi0,pre)`	mixture plot (contours, components)
`mixplots(Mix)`	plot of mixture colormap(summer)
`mixplotsl(Mix)`	plot of mixture, displays mixlls, ncomp
`[x,y,z] =... mixgrid(Mix,pchns,cchns,psi0,pre)`	coordinates for mixture plot
`[x,y,z] = datagrid(Mix)`	coordinates for data plot
`datascan(chns)`	scan data for 2 dim clusters
`mixmesh(Mix,pchns,cchns,psi0,pre)`	mixture mesh plot
`mixscan(Mix,chns,pre)`	scan mixture for 2 dim. clusters
`setaxis(list, ax)`	set global axis in subplots [a]
`sigscan(chns)`	scan signal
`fullscreen`	set full screen for current plot
`resizefig()`	set plot position

[a] list is list of subplots, ax a scaling see axis function

Interactive visualization

`mixshow(Mix)`	interactive plot of mixture
`mixbrow(Mix)`	interactive display of mixture attributes
`setdbg('function')`	interactive setting of ”dbstop”

Data preprocessing

`pre = preproc(pre)`	preprocess data
`pre = preinit(pre)`	initialize preprocessing
`pre = prestep(pre)`	preprocessing step
`out = filoutm(ord)`	outlier filtration by mixture estimation

Structure estimation

`Mix = ... mixstrid(Mix,Mix0,belief,nruns)`	estimate mixture structure
`MAPstr = ... facstr(Fac,Fac0,belief,nbest,nruns)`	estimate structure of a factor
`[...] = straux1(...)`	structure estimation based on LD decomposition
`[...] = strmax(...)`	structure estimation based on UD decomposition

Mixture simulation

`mixsimul(Sim, ndat)`	batch mixture simulation
`mixsimul(Sim)`	recursive mixture simulation
`Sim = statsim((ndat, ncom, cove)`	create static mixture with components on unit circle
`Sim = setsim(Sim, list)`	set simulation options
`[res, tstop] = simeval(Sim, chns, nrep, ndat, thr)`	repeated simulation characteristics
`tab = gentab(dim, dia)`	Markov transition table for Metropolis algorithm

Model verification

`[Fac, Res] = bisect(Fac0, nseg)`	estimate length of learning segment
`[Mix, Res ,pH0s] = ... valseg(Mix0, n, flag)`	validation test by temporal segmentation
`pen = relep(Mix, ndat)`	prediction errors norm
`pen = relepn(Mix, nstep)`	multi-step prediction error norm

Support of stopping rules

`[Fac, Q] = stopstac(Fac,dvect)`	stopping of a time series at stationary mode
`[Cl,Cu,Chat] = credit(C,beta)`	evaluation of an credibility interval
`[Cl,Cu,Chat,flag,st] = ... credits(C, beta, epsi)`	evaluation of an credibility interval with stopping

Basic conversion functions

`LD = ltdl(V)`	decompose positive definite matrix to L'DL
`Mix = mix2mix(Mix, form)`	decompose positive definite matrix to L'DL
`Com = com2com(Com, form)`	decompose positive definite matrix to L'DL
`X = arx2arx(X)`	decompose positive definite matrix to L'DL

Design of advisory system

`[aMix, aMixu] = ... inisyn(Mix,Mixu,pochn,uchn)`	initialize advisory design for normal mixture
`[aMix, aMixu] = ... inisyn(Mix,Mixu,Chns)`	call with channel descriptions
`aMix = ... aloptim(aMix,aMixu,ufc,nstep,chis)`	make academic advisory design for normal mixture
`ufc = ufcgen(Mixc, Mixc0)`	vector of unstable components
`aMix = ... soptim(aMix,aMixu,ufc,nstep,chis)`	perform simulaneous advisory design
`aMix = ... uoptim(aMix, aMixu, ufc, nstep, chis)`	simultaneous advisory design for normal mixture
`aMix = algen(aMix,aMixu,ufc)`	compute of probabilistic weights for advisory design
`[Mixu, ychns] = target(Chns)`	create user's target mixture
`[Mixu,Chns,ychns] = targeti(Chns)`	create target mixture according to user's wishes
`Mix = mixcopy(Mix1, Mix2)`	copy of ARX or ARX LS statistics
`[chn,mean,std] = meandisp(Chns)`	momentsof the user-given signal ranges

Channel descriptions

`Chns = chnconst(chns)`	build channel descriptions
`Chns = chnset(Chns,chns,fld, val)`	set channel descriptions field
`val = chnget(Chns,chns,fld)`	get values of channel descriptions fields

General purpose functions

`prodini`	standard Mixtools session beginning
`ashelp(funname, helpfile)`	get help
`prt(X)`	debugging prints
`is = equal(X1,X2, eps)`	test of equivalence up to a small difference
`str = genstr(order, nchn, td)`	generation of model structure of given order
`is = streq(str1, str2)`	compare two structures
`is = isstatic(Mix)`	test whether mixture is static
`is = isdimeq(X1,X2)`	test of equality of dimensions
`is = streq(str1,str2)`	test of equality of dimensions
`Mix = mixreorg(Mix, chns)`	reorganize mixture estimator
`nu = getnu(R,prec)`	solve equation log(0.5*nu)-psi(0.5*nu)=R
`[Mix, Mixs, Mixs1] = ...`	auxiliary estimation for factor splitting
`maxtd = mixmaxtd(Mix)`	maximum time delay in mixture
`mixpaths`	sets paths for mixtools toolboxes

Dialog functions

`dial1, dial2, dial3`	dialogue units for case studies
`testassign`	script used in dial1.m

funlistu.tex by PN November 3, 2005

6.3 Design base

Prior knowledge processing

`Facs = prior(Facs0, pri)`	prior knowledge processing
`lhs = ... pristr(Facs0, pri, beliefs, nbest, nrep)`	structure estimation with prior knowledge
`pristrd(Fac, Fac0, vll, sub)`	display results of structure estimation
`pri = scalepri(pri, pre, ychn)`	scale prior knowledge list
`prtstr(lhs, str)`	auxiliary print results of facstr

Estimation related operations

`[Mix, faclls] = ... mixupdt(Mix, flag, weight)`	one step of mixture update
`Mix = mixupdtp(Mix, flag)`	mixture update for projection based estimation
`Mix = mixestpb(Mix,frg,ndat,niter)`	iterative estimation by projection
`Mix = mixestqb(Mix,frg,ndat,niter)`	iterative quasi-Bayes mixture estimation
`Mix = mixestbq(Mix,frg,ndat,niter)`	iterative batch quasi-Bayes mixture estimation
`Mix = ... mixestbb(Mix,frg,ndat,niter,nstep)`	iterative estimation by forgetting branching
`Mix = mixestmt(Mix0,frg,ndat,niter)`	iterative batch quasi-Bayes mixture estimation
`Mix = mixestem(Mix0, ndat, niter)`	estimation by EM algorithm
`Mix = mixfrg(Mix ,frg)`	mixture forgetting
`[Mix0,handle] = ... mixflatv(Mix,niter,ndat,frg)`	mixture flattening with variable rate}
`[Mix, Mixs, Mixs1] = ... mixestfe(Mix0, frg, ndat, Mixs, Mixs1)`	mixture and filtration error estimate
`ndat=tukinit(Ndat)`	auxiliary function for buffered processing

Auxiliary estimation operations

`Mix = mixgmean(weights,Mix1,... )`	geometric means of mixtures
`dvec = getdvect(Fac)`	get data or regression vector
`lls = facdpred(Mix)`	compute trial factor predictions
`[s,s0] = mixdfms(Mix)`	sum degrees of freedom of the mixture
`Mix0 = mix2mix0(Mix)`	sum degrees of freedom of the mixture
`lls = loglik(LD,dfm,LD0, dfm0)`	increment of loglikelihood
`Sim = sim2pdf(Sim, ndat)`	sum degrees of freedom of the mixture

Prediction related operations

`Facs = fac2marg(Facs, pchns)`	sum degrees of freedom of the mixture
`Com = com2pro(Facs, pchns, cchns)`	sum degrees of freedom of the mixture
`[typ, ychns,...] = comunpk(...)`	get information about component
`Com = pro2pre(Facs, comaux, psi0)`	convert predictor to prediction component
`protest(Mix, pchns, cchns)`	check projection arguments

Pre-processing and data scaling

`data = scaledata(data, pre)`	scale data
`data = scalepsi(data, pre)`	scale data vector
`data = rescalepsi(data, pre)`	unscale data
`pre  = invprescal(pre)`	re-scaling information in list
`Chns = scaledescription(Chns,pre)`	scale description
`Pre  = preaux1(method, time, Pre)`	auxiliary function for data pre-processing

Basic square root algorithms

`[....] = dydrs(...)`	transformation of sum of 2 dyads
`[LD, D] =ldform(AD, D0)`	decomposition A 'D0 A -¿ L'DL decomposition
`[Eth, cove] = udform(Eth, cove)`	restore matrix factorized ARX component

Factor oriented operations

`[Fac, ep] = facupdt(Fac, weight)`	update factor statistics
`[Fac, ep] = ...` `facupdt(Fac, w, ep, zeta, dvect)`	update factor statistics using PB estimation
`Fac = facfrg(Fac, rate, Faca)`	factor forgetting
`Fac = facflat(Fac, rate, Faca)`	factor flattening
`Fac = facgmean(Fac1, Fac2,weight)`	geometric mean of factors
`Fac = ...` `facmerge(Fac1, Fac2, weight, weight1)`	merge factors
`[LH,FSC]  = facpred(Fac,w,Psi)`	compute logarithm of factor prediction
`vll = facvll(Fac, Fac0)`	compute factor v-log-likelihood

Design of advisory system

`Com  = arx2ful(Com, str)`	weights needed for advisory system design
`Com  = canarxls(ychns, str)`	build matrix ARX LS component
`pMix = facchng(pMix, com, Fac)`	auxiliary changes of mixture factors
`Mix  = pro2str(Mix, str)`	additional pointers to external structure
`...  = ricexp(....)`	auxiliary function for computing of expectation
`...  = ricpen(....)`	auxiliary function for computing of penalisation
`...  = ricpenu(....)`	computing of penalisation in simultal design
`...  = ricshift(....)`	shift of matrices and vectors
`aMix = synmixi(Mix, uchn, strc)`	convert mixture to control form aMix

Kullback-Leibler divergence

`dist = kldist(fac, Mix, Mix0)`	divergence of a factor in parameter space
`dist = kldist(  0, Mix, Mix0)`	divergence of all factors
`dist = kldist(Mix, Mix0)`	divergence of all components
`[d1,d2,d3,d4] = kldist(Mix1, Mix2)`	divergence of mixtures [a]
`dist = kldiscom(Mix, ndat)`	distance of components in data space
`dist = kldcom(Mix, Mix0)`	KL distance of components from initial ones
`kld  = kldisdir(s, s0)`	Kulback-Leibler distance of Dirichlet pdfs
`kld  = kldistc(Mix)`	KL distance of components in normal mixture

[a] distances: d1 - overall distance, d2 - distances of factors, d3 - distance of components, d4 - distance of component weights

Conversion functions

Conversion of an array of ARX components to the mixture and back

`Sim  = arxc2mix(Coms, dfcs)`	convert ARX components to simulator
`Coms = mix2arxc(Mix)`	KL distance of components in normal mixture
`Facs = arxc2fac(Com)`	KL distance of components in normal mixture
`Com  = fac2arxc(Facs)`	KL distance of components in normal mixture

Conversions of L'DL decompositions

`V    = ld2v(LD)`	KL distance of components in normal mixture
`LD   = ld2ld(L, D)`	replace diagonal unit of L by D
`[L,D]= ld2ld(LD)`	extract D from diagonal LD and replace it by D
`LD1  = ldchng(LD, str, LD1, str1)`	change part of L'DL decomposition

Conversion of L'DL to LS representations and back

`[Eth,Cth,cove,dfm] = fac2ls(Fac)`	change part of L'DL decomposition
`LD = ls2ld(Eth,Cth,cove,dfm)`	change part of L'DL decomposition
`[Eth,Cth,cove]=ld2ls(LD,dfm,nychn)`	change part of L'DL decomposition

Subselection from an L'DL decomposition

`LD   = ld2ld(LD,str1,str2)`	marginal L'DL decomposition [a]
`LD   = ldperm(LD, i)`	permute L'DL decomposition: i-th row to 1st row

[a] str1 is source and str2 target LD structure, str2 has to be contained in str1

Operations over triangular matrices

`UD  = ld2ud(LD)`	permute L'DL decomposition: i-th row to 1st row [a]
`UD  = utdu(X)`	upper triangular U'DU sym. matrix decomposition
`UD  = ld2ud(LD)`	permute L'DL decomposition: i-th row to 1st row
`LD  = ud2ld(UD)`	permute L'DL decomposition: i-th row to 1st row
`LDi = ldinv(LD)`	invert L'DL decomposition
`ut  = udinv(ut)`	invert upper triangular matrix
`LD  = ldupdt(LD , dvect, weight)`	update L'DL decomposition by weighted data vector
`UD  = udupdt(UD , dvect, weight)`	update U'DU decomposition by weighted data vector
`UD  = utinv(UD)`	upper triangle matrix inversion

[a] the decomposition U'DU, U is upper triangular with unit diagonal, V = U'DU. "UD" is the upper triangular matrix with "D" on diagonal

Factorized matrix ARX and matrix LS components

`Can  = arxc2can(Com)`	restore matrix factorized ARX component
`Com  = can2arxc(Can, n)`	[a] convert "Can" into matrix ARX LS component
`Can  = can2marg(Can)`	restore matrix factorized ARX component
`Facs = can2fac(Can, eps)`	convert matrix factorized component to matrix form
`[Can, ok] = com2can(Facs)`	convert component into matrix factorized component
`Facs = can2ls(Can,eps)`	convert matrix factorized component to matrix form

[a] "n" is number of marginal channels

Visualisation

`statmesh(Mix)`	interactive mesh plot of static mixture or data
`statplot(Mix)`	plot components of static mixture components
`[x,y,z] = statgrid(Mix)`	coordinates grid for 3-D display
`complot(Mix, com)`	plot of component of a mixture
`iterplot(Mix0, Mix, iter)`	plot initial and resulting mixture of an iteration step
`setfig(number)`	set figures windows
`fixerr(Mix)`	interactive set TIME for plots

Dump/restore of MATLAB array	
`savearray(X, filename)`	dump MATLAB array X to the disk file
`X = loadarray(filename)`	restore saved MATLAB array

General purpose functions	
`val = defaults('item')`	get values from database of defaults
`val = gauss1(dvect,Eth,cove)`	value of one dimentional Gaus pdf
`val = gaussn(dvect,Eth,cove)`	value of Gaussian pdf
`setfig(n)`	set figures windows
`val = getflds(cell,member)`	set figures windows
`val = betaln(p,q)`	logarithm of Euler's beta function
`fac = facsort(Facs)`	sort factors of a component
`list= cellcat(LIST)`	concatenate complex cell lists

Random trajectory generation	
`rnd = noise(etyp)`	generate a random number with a "etyp" distribution
`rnd = dirrnd(dfcs, n)`	samples from dirichlet distribution
`rnd = gamrnd(a, b)`	returns a matrix of random numbers chosen
`[r,theta] = giwrnd(Fac,n)`	samples from Gauss-inverse-Wishart distribution
`[...] = rndcheck(...)`	checks arguments of the random number generators
`rnd = randun`	sample from uniform distribution
`rnd = randnm`	sample from Normal distribution

6.4 Tutorial examples

{tutorial}

Commented examples can be found in Mixtools case studies

6.4.1 Case study: static mixture learning and prediction

Run example

6.4.2 Case study: dynamic mixture learning and prediction

Run example

Chapter 7

Construction of prior estimate (initialization)

{mixinit}

The initialization searches for mixture model structure that maximizes v-log-likelihood evaluated for the respective structures and data observed on the system considered.

The initialization is done by the functions *mixinit*. An article describes the processing logic. Detailed examples are available.

List of references to [1]: 6.4{122}, 8.4{275}, 6.4.4{125}, 6.4.5{125}, 6.4.21{137}, 6.4.24{138}, 6.4.26{139}, 6.4.30{141}, 6.4.34{143}, 6.4.46{151}, 8.4.5{277}, 8.4.7{278}, 8.4.9{279}, 8.4.12{281}, 8.4.15{283}, 6.6{167}, 6.6.3{168}, 6.6.5{169}, 8.6{300}, 8.6.1{300}, 8.6.8{304}.

The function *mixinit* is called as

Mixture initialization	

`Mix = ...`
`mixinit(Mix0,frg,ndat,niter,opt,belief)`
	initialization of mixture estimation

The arguments are (defaults are discussed below):

Mix	initialized estimated mixture
Mix0	initial mixture
frg	forgetting rate
ndat	length of data
niter	number of iterations
options	processing options
belief	belief on a guess of richest structure

Meaning of the input arguments with defaults follow.

`Mix0` is an pre-prior or flattened mixture. It should be created using all prior information available (e.g. static or order of dynamic mixture etc.). It is recommended to use the function genmixe for the purpose.

`frg` is a forgetting rate. Usually it is one or a default forgetting rate. The only exception is the case of the estimation based on forgetting branching where a very low forgetting is recommended (e.g. 0.6).

`niter` is a number of iterations. A low number of iterations is sufficient, the default value is 5.

`options` specifies processing options. They are coded as characters optionally followed by numbers. The options are discussed in the next subsection.

`belief` is an user's guess about the richest structure considered. As default, no belief is used. The detailed description is available.

The recommended practice is to estimate the initialized mixture by an iterative mixture estimation with many iterations.

Long processing and huge data size support is available.

7.1 Processing logic

One iteration of "mixinit" consists of the steps:

1. The mixture from previous step is flattened.
2. The initial mixture is estimated by a single pass of "mixestim". During the estimation, a pair of two-component static mixtures is fitted to prediction errors of the factors. The result is used for recognizing whether each factor consists just of a single "hill" or whether it covers several hills. The factors that result in multiple hills are candidates for splitting.
3. All components containing a candidate for splitting are split. During the split, the structure of factors is estimated.
4. The split mixture is flattened and estimated.
5. Splitting of components may lead to an excessive number of components so that an attempt is made to reduce the number of components by merging and cancelling them.
6. The resulting mixture is split and estimated. The "best mixture" (in the sense of maximum v-likelihood) is maintained during all iterations.

When number of iterations is exhausted or no other factors can be selected for split, the initialization ends. The last step is mixture structure estimation and a reduction of number of components.

7.2 Initialization options

The process of initialization can be modified by *initialization options*. The options are described below. For each of them, the processing without the option is described and marked by bullet. The alternative processing introduced by the options is described below. The comments and suggestions are presented in italic.

The options are:

- Mixture estimation inside "mixinit" is done by non-iterative quasi-Bayes estimation
  'p': by iterative mixture based on projection
  'q': by iterative quasi-Bayes mixture estimation
  'b': by iterative batch quasi-Bayes mixture estimation
  'f': by iterative mixture estimation based on forgetting branching
  Comment: the iterative estimation leads to a better mixture quality. The price paid for quality is a higher requirement on computing time.
- If the iterative estimation is selected, the default number of iterations in estimation is 10
  'n': the number that follows specify number of iterations in estimation
- The structure estimation of the mixture factors is based on 10 searches differing in initial conditions
  'h': number that follows specify the number of searches, e.g. 'h100' specifies that 100 searches differing in initial guesses of the structure are done

  Comment: this option can lead to better structure estimation and higher quality of the result but, the processing time visibly increases.

- There are two tuning knobs that can modify processing substantially - the default value is 3 iterations
  'g': number of initial iterations when all factors are split
  'k': number of iterations when components are not merged or erased

  Comment: the first option is to be specified when the initialization results in excessively small number of components. Note that each iteration increases the number of factors twice.

  The option 'k' is recommended if the merging process is an excessive one, e.g. when number of components during initialization does not increase.

- If no factor can be selected for split or the number of iterations is exhausted, the initialization ends. The last step is mixture structure estimation and reduction of number of components
  'c': this housekeeping is skipped

  This option is used when the user wants to make the structure estimation by different means.

- Other character among options are ignored, e.g. the option '0' implies that defaults are used.

Summary of options	
options for estimation	
q	iterative quasi-Bayes mixture estimation
b	iterative batch quasi-Bayes mixture estimation
f	iterative mixture estimation based on forgetting branching
n	number of iterations for iterative estimation, a number follows
option for structure estimation	
h	number of runs for structure estimation (integer follows)
option that modify processing	
c	do not make the final housekeeping
g	number of initial steps when all factors are split (2)
k	umber of steps when components are not merged or erased (2)

7.3 Case studies in initialization

Commented examples of mixinit are are available.

7.3.1 Computational Efficiency of Static Mixture Initialization

Run example

7.3.2 Dynamic Mixture Initialization

Run example

7.3.3 Static Mixture Initialization: "Banana Shape" Benchmark

Run example

7.3.4 BMTB Algorithm of Mixture Initialization

Run example

7.3.5 Initialization of Static Onedimensional Mixture

Run example

mixinit.tex zinit.m by PN November 3, 2005

Chapter 8

Approximate parameter estimation

{mixestim}

The *approximate parameter estimation* of ARX mixtures (shortly *mixture estimation*) is the topic discussed in this section, refer to 6.5{157} and 8.5{294}.

8.1 Implementation

Common rules:

Estimation methods implemented are:

- projection based algorithm;
- quasi-Bayes algorithm;
- batch quasi-Bayes algorithm.

branching by forgetting algorithm (see 6.4.30{141}) is implemented for quasi-Bayes algorithm;

recursive processing is available for the projection and quasi-Bayes estimation algorithm;

iterative estimation is available for all algorithms 6.4.26{139};

generic estimation function mixest calls the iterative algorithms defined by an argument opt ;

estimation using *stopping rules* is available for projection and quasi-Bayes estimation;

forgetting during estimation can be introduced as:
forgetting rate for all estimation functions;
alternative forgetting rate in projection and quasi-Bayes estimation.

The estimation functions have the following arguments: The functions input arguments with together with defaults are:

argument	*meaning*	*defaults*
Mix	output estimated mixture	
Mix0	initial mixture to be estimated	must be specified
frg	forgetting rate	default forgetting rate
ndat	size of data sample	length of "DATA"
niter	number of iterations	10
Mixa	mixture used for stabilized forgetting	no stabilized forgetting is used
opt	coded method for the generic function	'p'
thr	threshold for stopping rules	0.0025

The *opt* is coded as
'p': iterative projection based estimation (default);
'q': iterative quasi-Bayes estimation (default);
'b': iterative batch quasi-Bayes estimation;
'f': iterative estimation based on branching by forgetting

Mixture estimation

— *Basic functions*	
`Mix = mixestim(Mix0,frg,ndat,Mixa)`	quasi-Bayes mixture estimation
`Mix = mixestim(Mix0, frg)`	recursive quasi-Bayes mixture estimation
`Mix = mixestimp(Mix0, frg, ndat,Mixa)`	projection based quasi-Bayes estimation
`Mix = mixestimp(Mix0, frg)`	projection based recursive quasi-Bayes estimation
— *iterative functions*	
`Mix = mixestpb(Mix,frg,ndat,niter)`	iterative estimation by projection
`Mix = mixestqb(Mix,frg,ndat,niter)`	iterative quasi-Bayes mixture estimation
`Mix = mixestbq(Mix,frg,ndat,niter)`	iterative batch quasi-Bayes mixture estimation
`Mix = mixestbb(Mix,frg,ndat,niter,nstep)`	xxxxx
— *generic function*	
`Mix = ... mixest(Mix0, frg, niter, opt)`	iterative mixture estimation
— *estimation functions with stopping rules*	
`[Mix,tstop,Qs] = ... mixestims(Mix, frg, ndat, Mixa,threshold)`	quasi-Bayes estimation of ARX mixture with stopping
`[Mix,tstop, Qs] = ... mixestimps(Mix, frg, ndat, Mixa,threshold)`	projection based estimation with stopping
`[Mix, tstops] =... mixestpbs(Mix0,frg,ndat,niter,thr)`	projection based estimation with stopping
`[Mix, tstops] = mixestqbs(Mix0,frg,ndat,niter,thr)`	repetitive quasi-Bayes estimation with stopping
`[Mix, tstops] =... mixestpbs(Mix0,frg,ndat,niter,thr)`	repetitive projection based estimation with stopping
`[Mix, tstops] =... mixestqbs(Mix0,frg,ndat,niter,thr)`	repetitive quasi-Bayes estimation with stopping

Mixture flattenning and forgetting

`Mix0 = mixflat(Mix)`	mixture flattening
`Mix = mixfrg(Mix ,frg)`	mixture forgetting

8.1.1 Estimation statistics

The quality of the estimated mixture can be judged from the value of v-log-likelihood. This statistic offers the possibility of comparison of different mixtures estimated with the same data sample.

In the quasi-Bayes mixture estimation, the statistics are computed recursivelly and are held in the mixture states.

facllds	trial factor predictions determining factor weights
comlls	component predictions
mixll	posterior data likelihood (mixture prediction)
comwgs	component weights
facwgs	factor weights

The letters "ll" in the name means that logarithms of the statistics are computed. Details, how the statistics are evaluated can be found in function "mixupdt.m" (m-version of "mixestim").
The computed statistics are:

- *actual values in recursive data processing*
- *summed values in batch data processing.*

The statistics are computed in estimation. They can be computed by the function:

Estimation statistics

`Mix   = mixstats(Mix, ndat)`	compute estimation statistics
`Mix   = mixstats(Mix)`	compute statistics recursively

8.1.2 Forgetting

The value of forgetting rate should be close to 1 but, a small value (default 0.6) is to be selected for quasi-Bayes algorithm and forgetting branching.

The optimum forgetting rate can be computed:

Estimation of forgetting

`[Mix,frg,mixlls,frgs] =`
`estfrg(Mix0,frgs,ndat,niter,method,Mixa)`
	estimate forgetting rate

8.2 Case studies

8.2.1 Quasi-Bayes Mixture Estimation

Run example

8.2.2 Comparison of Mixture Estimation Algorithms: Static Case

Run example

8.2.3 Comparison of Mixture Estimation Algorithms: Dynamic Case

Run example

8.2.4 Computational Efficiency of Mixture Estimation Algorithms

Run example

8.2.5 Mixture Estimation Based on Batch Quasi-Bayes Algorithm (BQB)

Run example

8.2.6 Mixture Estimation Based on Branching by Forgetting (BFRG)

Run example

Chapter 9

Prediction with normal mixture

{prediction}

9.1 Projection and prediction with mixtures

There are two basic operations related to prediction with normal mixture:

- *mixture projection*
  means marginalization and conditioning, see [1]. The result of these operations is referred to as *mixture projector*.
- *mixture prediction*
  arises from the mixture projection by substitution of a specific regression vector into it. The result is referred to as *mixture predictor*.

More detailed text is available in Mixtools.

9.1.1 Mixture projection

The projection converts mixture estimator into mixture *projector*. It provides description of Student pdf (2.13) mostly approximated by normal pdf (2.8). The projection is conditional pdf on a set of modelled channels referred to as *predicted channels*. It is conditioned by another set of modelled channels referred to as *channels in condition*. The projector can be re-built for a new selection of those channels.

The mixture projection is done by the function:

Mixture projection	
`pMix = mix2pro(Mix, pchns, cchns)`	build mixture projection

The argument together with defaults are:

argument	*meaning*	*defaults*
Mix	mixture estimator or projector	must be specified
pchns	predicted channels	all channels
cchns	channels in condition	no channels in condition
pMix	output mixture projector	

9.1.2 Reduction of data space

The marginalization by *mix2mixm* preserves in the p-mixture all factors for original channels so that the re-building operation can be done.

The function *mix2mix* makes the marginalization but it builds a new p-mixture without unused factors. It reduces the data space before prediction and consequently reduces computing time.

Data-marginal projection	
`pMix = mix2mixm(Mix, pchns)`	build data-marginal projector

9.1.3 Prediction with mixture projection

{propred}

The mixture prediction with mixture projector is done as:

Prediction with projector	
`lhs = profix(pMix, psi0, pre)`	mixture prediction

The input arguments with defaults are listed and some explained below:

argument	*meaning*	*defaults*
pMix	mixture projector	must be specified
psi0	zero-delayed regression vector	extracted from DATA
pre	prediction scaling	no prediction scaling is done

The outputs "lhs" arguments are specified as

```
     pMix              [ Eths,  coves,  alphas ]
[ pMix ,  weights]     [ Eths,  coves,  alphas ,  weights].
```

The meaning of the arguments is:

pMix	mixture prediction (static matrix ARX LS p-mixture)
Eths	vector or cell vector of means of individual components
coves	vector or cell vector of noise covariances
alphas	weights of individual components corresponding to normalized dfcs modified due to conditioning
weights	data-dependent approximate component weights

Comments on arguments:

Zero-delayed regression vector

The projector is converted into *predictor* by substituting a data vector at a specific time t. The vector consists of data values up to the time $t-1$ and the current values of channels in condition as well as the values of not-predicted channels with zero delay. The data vector is referred to as *zero-delayed regression vector.*

The historical values are implicitly extracted from the signal database – global matrix DATA. The zero-delayed entries of the regression vector can be specified; if not fully specified, the values are extracted from the signal database, too.

The zero-delayed regression vector has 2 rows, the first row contains values, the second one the corresponding channels; the second row can be omitted if there is only one item in the vector.

Prediction scaling

Use of scaled data is recommended in learning with mixtures. In this case, the mixture prediction must be re-scaled to the original data scaling. This is done by the argument *pre* that contains record about the data scaling, see section "Data preprocessing". The zero-delayed regression vector is returned in the original data scaling.

Diaries available: conditional prediction marginal prediction

9.1.4 Prediction with mixture

Joined mixture projection and prediction done by one function is available:

Mixture prediction	
`lhs = mixpro(Mix,pchns,cchns,psi0,pre)`	mixture prediction

The meaning of input and output arguments is the same as in the section 9.1.3.

9.1.5 Multi-step prediction

The prediction can be done for a number of processing steps ahead:

Prediction n-steps ahead	
`lhs = profixn(Mix, psi0, pre, nsteps)`	prediction nsteps ahead
`lhs = profixna(Mix, psi, pre, nsteps)`	prediction nsteps ahead with data specified

Meaning of the arguments and defaults are the same as above.

The argument *psi* contains specification of data that temporarily preplaces data in the *DATA* matrix in each prediction step. The form specification is:
`psi = [values; channels; time_delays]`

The *time_delays* are relative to the current *TIME* value and can have any sign.

Diaries of case studies are available. The *profixn* function was modified that it show the data internally generated.

Diaries available: prediction details
different projetors
prediction with fixed data

9.1.6 Prediction error

Prediction error norm and multi-step prediction error norm are convenient characteristics that express quality of estimated mixtures.

Prediction error norm	
`[epn,yp,dd]=relep(pMix,ndat)`	prediction error norm
`[epn,yp,dd]=relepn(pMix,ndat,nstep)`	multi-step prediction error norm

The meanings of the arguments are:

Mix	mixture predictor or projector
ndat	length of data to be processed
step	number of steps of multi-step prediction
epn	value of prediction error-norm
yp	trajectory of prediction
dd	corresponding data

Note: both functions contain and additional argument that allow to compute data-dependent prediction error in the case of static mixture. Up to now, the option is not sufficiently tested.

The normalization is done inside the functions:

```
ep  = dd-yp;                          prediction error
sy  = std(dd');
er  = std(ep') + abs(mean(ep'));
epn = (er./sy)';
```

The prediction error norm expresses the portion of standard deviation of data that can be explained by the linear model behind.

9.2 Case studies in projection and prediction

Commented examples of mixinit are are available.

9.2.1 Prediction with Static Mixtures - scaled data

Run example

9.2.2 Prediction with SISO dynamic component

Run example

9.2.3 Prediction with Mixture of Two Dynamic SISO Components

Run example

9.2.4 Multi-Step Prediction with Static Mixture

Run example

9.2.5 Multi-step Prediction with Mixture of Two Dynamic SISO Components

Run example

9.2.6 Multi-step Prediction with SISO Dynamic Model

Run example

9.2.7 Prediction with Mixture on Grouped Data

Run example

9.2.8 Prediction with Static mixture

Run example

Chapter 10

Simulation

{mixsimul}

10.1 ARX mixture simulation

Mixture simulation serves for development of algorithms, debugging and case studies. Mixture of any type *Sim* can be used for the simulation. Two simulation modes are available:

Simulation	
`Sim = mixsimul(Sim,ndat)`	batch simulation
`Sim = mixsimul(Sim)`	recursive simulation

The simulation fills modeled channels the global matrix *DATA* by simulated data. The matrix must be allocated (pre-allocated) before the simulation starts. The obligatory pre-allocation makes it possible to fill DATA channels by different simulators and/or use specific channels e.g. for control values.
There are several simulation options:
- Markov jumps among components
- data scaling
- changes of internal simulation states
- use of covariance of regression coefficients
- different type of noise

The simulation options can be set by the function *setsim*:

Setting simulation options	
`Sim = setsim(Sim, list)`	set simulation option

Examples of setting simulation options follow.

Markov jumps among components
the probability transition table is specified:

```
table = ...                          % probability of transitions among components
Sim  = setsim(Sim, table, table) }   % set the option
```

This option has a global character, does not refer to individual components. The table can be generated by:

```
tab = gentab(dim, dia)                % generate the table
```

The table has diagonal filled by *dia*, other entries are calculated.

data scaling
The mixture learning is done using scaled data. The preprocessing list is Pre. The simulator

generates data in original data ranges by

```
Pre = preproc({scale, []});            % data scaling
Sim = setsim(Sim, pre, Pre);           % set scaling option
```

changes of internal simulation states during recursive simulation
The internal simulation states are held in states that are generated at the first simulation step. If the mixture changes, the states must be reset.

use of covariance of regression coefficients Cth in simulation

```
Sim = setsim(Sim, useCth 1);           % use of covariance of coefficients
```

Different noise type

```
Sim = setsim(Sim, noise, 3);           % use log-normal noise
```

The noise type is coded:

1	Gaussian
2	uniform
3	lognormal
4	Cauchy

The process noise is normalized to zero mean and standard deviation one (with exception of the Cauchy distribution that have no mean and standard deviation).

selection of factors
The options above are set for a selection of factors. The selection is marked by 0 or 1 in a matrix coms that has the same dimension as Sim.coms. The factor numbers are extracted from Sim.coms. The selection is valid to a next change, the default selection is $1 + 0 * Sim.coms$. For example, the setting of factors that model the 3rd channel:

```
coms = 0*Sim.coms;
coms(:,3) = coms(:,3) + 1
coms =
     0 0 1 0
     0 0 1 0
     0 0 1 0
     0 0 1 0
     0 0 1 0
Sim = simset(Sim, {coms, coms});     % set selection of factors
```

Note: The global matrix DATA is processed by MEX functions. In this case, the matrix must not be defined by reference:

```
DATA = zeros(nchn, ndat);              % correct definition
DATA = data;                           % incorrect definition by reference
DATA = 1*data;                         % correct definition
```

10.2 Simulation case studies

10.2.1 Case study: Markov jumps, scalling and noise type in simulation

Run example

10.2.2 Case study: use of covariance of regression coefficients

Run example

10.2.3 Case study: simulation with projection

Run example

Chapter 11

Visualization

{visual}

The visualization functions implemented are

Visualization	
`mixplot (Mix,pchns,cchns,psi0,pre)`	mixture plot (shaded)
`mixplotc(Mix,pchns,cchns,psi0,pre)`	mixture plot (contours, components)
`mixplots(pMix)`	plot of mixture colormap(summer)
`mixplotsl(pMix)`	plot of mixture, displays mixlls, ncomp
`[x,y,z] =... mixgrid(Mix,pchns,cchns,psi0,pre)`	coordinates for mixture plot
`[x,y,z] = datagrid(Mix)`	coordinates for data plot
`datascan(chns)`	scan data for 2 dim clusters
`mixmesh(Mix,pchns,cchns,psi0,pre)`	mixture mesh plot
`mixscan(Mix,chns,pre)`	scan mixture for 2 dim. clusters
`setaxis(list, ax)`	set global axis in subplots [a]
`sigscan(chns)`	scan signal
`fullscreen`	set full screen for current plot
`resizefig()`	set plot position

[a]list is list of subplots, ax a scaling see axis function

Mixture predictions are displayed so that the mixture visualization is closely related to mixture prediction. The visualization functions uses the same arguments as the function "mixpro" (that is called internally, see Section 9). Other arguments refer to coordinates, grid densities and ranges.

The arguments are:

Mix	mixture, any mixture form
pMix	mixture projector or prediction
pchns	predicted channels, default is 1st and 2nd channel
cchns	channels in condition, default is no channels
psi0	zero-delayed data vector, by default taken from DATA
x, y, z	plot coordinates
n	grid density or vector of densities
	default is 100 for 1 dimension and 50 for 2 dimensions
r	is range of x,y coordinates or vector of 4 elements
	the default is a convenient range from components ranges

For dynamic mixture projection, the user must specify TIME and supply the global matrix DATA. The same is valid for the case of conditional projection of a static mixture. But, in the later case the zero-delayed data vector can be specified by psi0.

It is recommended to use the mixture argument in the form of mixture projection or prediction.

Examples of the function usage are in are in Mixtools guidel

Chapter 12

Stopping rules

{stopping}

12.1 Introduction

{stac-intro}

Each learning process contains a transient period followed by a stationary part. The stopping rules determine when the stationary part begins. It means that a *stopping statistic* Q is computed and compared with a *threshold value* view a summary paper.

12.2 Learning with normal ARX factors

{stac-factor}

The stopping rules based on recursive estimation of a factor are presented. In recursive estimation, the factor is feeded by a relevant statistics, in the case of mixture estimation by v-log-likelihood.
The function *stopstac* supports application of stopping rules:

Stopping of a time series at stationary mode	
`Fac, Q] = stopstac(Fac , dvect)`	update and compute statistics

where is

xFacx	the recursively updated factor;
xQ	the stopping statistics;
xdvect	data vector used for updating of Fac

When *dvect* is missing, the data vector is extracted from signal database using $Fac.str$.
When in recursive estimation Q falls below the treshold value, the stationary behavior begins.
The statistics Q has an exponential shape overlayed by noise. It can lead to false determination of the beginning of stationary behavior. It is recommended to estimate it using the model $log(Q) = k * log(t) + c$.

Examples follows.

12.2.1 Case study: Stopping rules in factor estimation

In this case study:
- observed data are generated by a SISO ARX dynamic component in open loop;
- in recursive estimation, the stopping statistics is computed and displayed together with the threshold value and trajectory of parameters estimate.

Run example

12.2.2 Case study: Stopping based on a statistics

{mixll}

In this case study:

- static mixture with several components generates data sample;
- static factor is estimated using v-log-likelihood;
- the stationarity of the factor estimate implies the stationarity of the mixture estimate.

Run example

12.3 Estimation of credibility intervals

{stac-credit}

The function *credits* makes the estimation of credibility intervals. It computes the stopping statistics and stops processing when the stationary behavior is reached:

Estimation of credibility intervals and stopping	

`[Cl,Cu,Chat,flag,Q] = ...`
`credits(C, beta, threshold)`
	credibility intervals

where is

Cl	lower credibility bound
Cu	upper credibility bound
Chat	center of the credibility interval
flag	stopping flag 1 - stop, 0 - do not stop data acqusition
Q	value of the stopping statistics
C	vector of independent realizations
beta	credibility level in (0,1)
threshold	upper bound on relative error

Note: the useful function *credit* evaluates a credibility interval without stopping:

Credibility interval	
`[Cl,Cu,Chat] = credit(C,beta)`	evaluate credibility interval

where is

Cl	lower credibility bound
Cu	upper credibility bound
Chat	center of the credibility interval
C	no-vector of independent realizations
beta	credibility level in (0,1)

12.3.1 Case study: Stopping and credibility intervals

In this case study:

- static mixture of several components generates data;
- simulation and estimation tasks are carried out in a repetitions; stopping time is recorded;
- the repetitions are stopped when the stopping times trajectory shows stationarity

Run example

12.4 Mixture estimation with stopping rules

{stac-estim}

An argument of the relevant functions is added - a threshold *thr*. Its presence forces the function to use the stopping algorithms. The argument may be empty - then a default is used (see defaults('E')).

Basic estimation methods with stopping rules	
`[Mix, tstops] = mixestpbs(Mix0,frg,ndat,niter,thr)`	projection based estimation with stopping
`[Mix, tstops] = mixestqbs(Mix0,frg,ndat,niter,thr)`	repetitive quasi-Bayes estimation with stopping

Note: the argument *tstop* contain the stopping time. The *Mixa* is usually empty.
The repetitive estimation is done by functions

Repetitive estimation with stopping rules	
`[Mix, tstops] = mixestpbs(Mix0,frg,ndat,niter,thr)`	projection based estimation with stopping
`[Mix, tstops] = mixestqbs(Mix0,frg,ndat,niter,thr)`	repetitive quasi-Bayes estimation with stopping

Notes:
- the function *mixest* options are extended by 'P' and 'Q' for usage of stopping rules;
- the function *mixinit* can used the options
- the argument *tstops* contains stoping times of individual iterations.
- individual iterations are stopped by the functions *credits*.

12.4.1 Case study: use of basic estimation functions with stopping

In this case study:
- static mixture of several components generates data;
- estimation is done either by *mixestims* or *mixestimps*;
- trajectory of log(Q) is recorded and displayed.

Run example

12.4.2 Case study: repetitive estimation with stopping

In this case study:
- static mixture of several components generates data;
- estimation is done either by *mixestims* or *mixestimps*;
- trajectory of stoping times is recorded and displayed.

Run example

12.4.3 Case study: Comparison of estimation functions with and without stopping

{zeststop}

In this case study:
- static mixture of several components generates data;
- estimation is done either by *mixestims* or *mixestimps*;
- trajectory of stoping times is recorded and displayed.

Run example

12.4.4 Case study: mixture initialization using stopping rules

The stopping rules are applied as *mixest* options. In this case study:
- static mixture of several components generates data;
- initialization is done;
- result of each iteration is displayed.

Run example

12.5 Model characteristics based on simulations with stopping rules

{stacsimul}

Estimated model characteristics are often obtained by repeated simulations. This is usually demanding and time-consuming task. The function *simeval* is designed to collect basic confidence intervals and to record repeated trajectories effectively with the use of stopping rules.
Two benefits of the using *simeval* are:

- individual simulation runs are stopped when stationary state is reached (the function *stopstac* is used);
- the repetitive simulation runs are finished when stationarity is reached (the function *credits* is employed);
- MEX function solution makes experiments in acceptable computing time.

The function *simeval* makes the simulations:

Model characteristics via simulation	
`[res,tstop] = simeval(Sim,chns,nrep,ndat,thr)`	get model characteristics and trajectories

where

res	cell vector containing results
tstop	stop time of individual trajectories
Sim	simulator or a task, see below
chns	list of relevant channels or Facs - cell array of factors
nrep	maximum number simulation runs
ndat	maximum length of trajectories
thr	threshold value for stopping

The processing results are held in a cell vector. Each cell contains results related to a individual channels. The result fields are:

stats	confidence interval for range and increments
tra	trajectories of individual simulation runs

Notes:

- the confidence intervals are presented as (low border - mean - high border) – the evaluation is done by the function *credit*;
- the trajectories recorded can be used for any type of analysis;
- instead of the argument *Sim* in *simeval* call, a *task* can be used. It consists of a function handle *h* and an argument *states*. The function is called in the repetitive simulation in *simeval* as `task{2} = feval(h, task{2});` In this case, the *ndat* argument of must be a vector of initial and terminal processing time.

12.5.1 Case sstudy: SISO model

In this case study:

- dynamic mixture of one component generates data;
- model characteristic are collected and displayed.

Run example

Chapter 13

Stabilization of mixture estimate

{stabil}

13.1 Supporting functions

{stabil}

The estimated mixture can be unstable. With respect to control design, the mixture should be stabilized before used. The stabilization is an iterative process with stopping rules employed.

The stabilized mixture should be close to the unstable one in respect of prediction error.

The function *stabmix* makes the stabilization:

Mixture stabilization	
`[Mix,m,Qs]= stabmix(Mix,g,no,thr)`	mixture stabilization

where the arguments are:

Mix	output mixture of original type with stabilized components
m	number of stable iterations
Qs	stopping statistics
Mix	input mixture of any type
g	radius of the circle of stable eigenvalues, g in (0,1]
no	upper bound on the number of Monte-Carlo samples
thr	threshold for stopping

The test of stability is solved by:

Test of mixture stability	
`[is, eigs] = isstable(Mix)`	test of mixture stability

where

is	1 if stable, 0 if not
eigs	absolute value of eigenvectors of components
Mix	mixture of any type

13.1.1 Case study: Mixture stabilization

Run example

Chapter 14

Structure estimation and prior knowledge

{prior}

14.1 Estimation of structure of mixture factors

During initialization, estimation of structure of factors, i.e., the selection of significant entries within a richest regression vector containing all potential regressors, must be estimated,

With presence of a prior knowledge about the structure, this task can be advantageously solved by the structure pristr, see section ”Prior knowledge incorporation”.

The function facstr is designed for estimation of structure of mixture factors. It searches for the factor structure that has the highest posterior probability in a space of competitive factor structures [5].

The user specifies the space of competitors in the form of the richest (maximum possible) factor structure.

The structures of factors are estimated inside the mixinit function so that no explicit estimation of factors is necessary after the mixture initialization. Nevertheless, the structure estimation is used for detailed analysis, experiments or correction of mixinit results.

The function structure estimation is done by:

Structure estimation of factors	

`[MAPstr,lhs] = ...`
`   facstr(Fac,Fac0,belief,nbest,nruns)`
	estimate structure of a factor

`Mix = ...`
`   mixstrid(Mix,Mix0,belief,nruns)`
	estimate mixture structure

The function arguments are:

MAPstr	the MAP estimated factor structure
lhs	likelihoods and best structures found
Fac	the estimated factor
Fac0	the corresponding initial factor
belief	specifies user’s belief on a guess of richest structure about the MAP structure
nbest	specifies the number of ”best” structures held in estimation
nrep	specifies number of repetitive search with random starts

THe function mixstrid estimate the initial mixture and calls facstr for all factors.
The belief is a vector of the same length as the iprichest factor. Its elements specify that the corresponding items (the pairs of channel and delay) of the richest factor:

1	surely present	2	probably present
3	probably not present	4	surely not present

Commented example can be found in Mixtools Guide.

14.1.1 Case study: Factor structure estimation

The case study demonstrates dependence of factor structure estimation on initial model settings and to inspects time evolution of point estimates.

Run example

14.2 Prior knowledge in ARX models

The prior knowledge is a useful tool in demanding learning tasks, e.g. in the task of ARX model structure estimation. As a rule, data available are usually poorly informative in this case and use of prior knowledge is the only possibility to derive a usable model. Underlying philosophy and processing algorithms are described in [6, 7, 8, 9, 10, 11].

14.2.1 Prior knowledge coding

The prior knowledge is coded in the form of a *cell list* that means as a cell vector of pairs of cells. The first one of each pair specifies the type of the prior knowledge, the second one (often again cell vector) contains numerical characteristics.

Individual prior knowledge types are exposed. The first two types refers to all outputs.

Data sample

can contain historical data not fully consistent with the current data sample, e.g. not sufficiently excited or recorded in another working point. The data collected on simulated system has this character, too.

The data are processed with a forgetting rate or with a grid of forgetting rates. If the forgetting rates are not specified, the `prior` function adds a default grid of rates (from "defaults.m"). For the data sample recorded in the matrix `data`, the respective coding is:

```
{'data' {data frgs} }  or
     {'data' data}
```

Data envelope

are two data matrices representing data ranges *datalow* and *datahigh*. The coding is:

```
{'fdata' {datalow datahigh}}
```

14.2.2 Channel specific prior knowledge

The prior knowledge pieces that follow are responses observed on an *output channel* caused by a special signal applied to an *input channel*. All frequencies specified are in Hertz, the phases are in degrees. With the exception of system static gain, it is necessary to specify also the *sampling time* (in seconds) in any position in the prior knowledge list:

```
{'stime' sampling_time }
```

The identifiers used are:

xychn	output channel
xuchn	input channel
xst	sampling time

The relevant prior knowledge pieces are discussed.

System static gain
is change of an output in steady state due to unit change of an input. It is specified in range of values:

```
{'gain' [uchn gainlow gainhigh] }
```

Frequency response
means amplitude and phase of output when a sinusoidal signal of a frequency and amplitude one is applied to an input. The amplitude is expected in a range of values (`alow, ahigh`). The coding is:

```
{'ampl' [uchn frequency alow ahigh phase]}
```

The phase can be specified as a vector of values. If it is empty, the `prior` function adds a default grid of phases.

Cut–off frequency
is the frequency of the signal applied to an input that is not reflected by the output:

```
{'cut'  [uchn frequency]}
```

The frequencies allow multiple specification. If not specified, the function `prior` adds a default grid of values (multiples of the frequency specified).

Dominant time constants
are implemented by modelling lower and upper envelope of the impulse response generated by the first or second order model with time constants equal to the specified bounds ($tclow, tchigh$) on the time constant. It is specified in range of values:

```
{'tc' [uchn tclow tchigh]}
```

The prior knowledge list can contain specification related to several output channels. In the case, the sub-lists must be separated by output specification that is valid till a new specification:

```
{'ychn' ychn }
```

14.2.3 Conversion of the prior knowledge into fictitious factors

Generally, it is strongly recommended to scale data sample. At the case, the prior knowledge must be scaled, too. The scaling of data supplies scale of individual channels. It is used in scaling of prior knowledge pieces:

```
pre  = preproc( {'scale' [ ]} );          % scale data
pri  = scalepri( pri, pre);               % scale prior knowledge
```

Scaling of prior knowledge	
`pri  = scalepri( pri, pre);`	scale prior knowledge

14.2.4 Prior knowledge processing

The prior knowledge is converted into fictitious factor(s) by:

prior

Prior knowledge processing	
`Facs = prior(Facs0, pri)`	prior knowledge processing

The arguments are:

Facs	fictitious factor or array of factors for individual outputs
Facs0	initial ARX factor or array of factors for individual outputs
pri	prior knowledge list

The results of prior data processing must be weighted and merged with data sample. The merger is used in structure estimation and as the alternative model for stabilized forgetting. The merging is done by:

primerge

Merging of fictitious factors with data	

`[Fac, vll, Fac0] = ...`
`primerge(FacD, Fac0, Facs)`
	merging of ficticious factors

The arguments are:

Fac	merged posterior factor with prior knowledge
vll	v-log-likelihood
Fac0	merge prior factors
FacD	estimated factor with very flat initial factor
Fac0	initial fictitious factor
Facs	fictitious factor or cell vector of the factors reflection prior knowledge

pristr

The function `pristr` is designed for estimation of structure of factors. It searches for the factor structure that has the highest posterior probability in a space of competitive factor structures.

Structure estimation with use of prior knowledge is done by:

Factor structure estimation with use of prior knowledge	

`[Facs,vlls]=...`
`pristr(Facs0,pri,belifs,nbest,nrep)`
	structure estimation with prior knowledge

The arguments are:

Facs	resulting estimated factor or cell vector of factors
vlls	cell vector of information or cell vector of the information
Facs0	richest initial factor or cell vector of factors
pri	prior knowledge list (can be empty)
beliefs	belief of cell vector of belifs or empty matrix
nbest	number of "best" regressor maintained
nrep	number of repetitions of search in space of regressors

The argument vlls contains information about v-log-likelihood:

vlls1	v-log-likelihood, nested prior
vlls2	v-log-likelihood with prior
vlls3	"best" structures in cell array
vlls4	structures in a block

{Xoldprior}

Commented diaries are available.

14.3 Case studies with prior knowledge

14.3.1 Case study: Prior knowledge in structure estimation

The case study shows influence of prior knowledge model in factor structure estimation. The prior knowledge is static gain.

Run example

14.3.2 Case study: Prior knowledge sources

The case study shows influence of prior knowledge of different types in estimation and structure estimation

Run example

Chapter 15

Model validation

{validate}

Complexity of mixture estimation makes a model validation a necessary part of the model design. Several validation case studies show the basic algorithms described in [1], sections 6.7{185}, 8.7{306}, 6.7.3{191}, 8.7.3{309}.

A selection of methods is exposed in the form of case studies. Commented diaries of the methods discussed are available.

15.1 Case studies

15.1.1 Static Mixture Checking via Simulation

{zversim}

Run example

15.1.2 Model validation by learning results

{zverres}

Run example

15.1.3 Forgetting based model validation

{zverfrg}

Run example

15.1.4 Model validation by prediction error

{zverep}

Run example

15.1.5 Model Validation by Cross-Validation of Learning Results

{zverseg}

Run example

Chapter 16

References to Mixtools Guide

This chapter summarises the sections of Mixtools Guide that remains unchanged.

16.1 Channels description

Reference to Mixtools Guide.

16.2 Design and advising

Reference to Mixtools Guide.

16.3 Tutorial on design and advising

Reference to Mixtools Guide.

16.4 Mex and API functions

Reference to Mixtools Guide.

Chapter 17

Appendices

17.1 References

Bibliography

[1] M. Kárný, J. Böhm, T.V. Guy, L. Jirsa, I. Nagy, P. Nedoma, and L. Tesař, *Optimized Bayesian Dynamic Advising: Theory and Algorithms*, Springer, London, 2005, ISBN 1-85233-928-4, pp. 552.

[2] V. Peterka, "Bayesian system identification", in *Trends and Progress in System Identification*, P. Eykhoff, Ed., pp. 239–304. Pergamon Press, Oxford, 1981.

[3] P. Nedoma, M. Kárný, I. Nagy, and M. Valečková, "Mixtools. MATLAB Toolbox for Mixtures", Tech. Rep. 1995, ÚTIA AV ČR, Prague, 2000.

[4] L. Tesař, "Data preprocessing manual", Tech. Rep.

[5] L. Berec, *Model Structure Identification: Global and Local Views. Bayesian Solution*, Ph.D. Thesis, Czech Technical University, Faculty of Nuclear Sciences and Physical Engineering, Prague, Czech Republic, 1998.

[6] P. Nedoma, M. Kárný, and J. Böhm, "Software tools for use of prior knowledge in design of LQG adaptive controllers", in *The preprints of IFAC workshop ACASP'98*, pp. 425–429. Glasgow, 1998.

[7] Kárný M. and Nedoma P., "Automatic processing of prior information with application to identification of regression model", *Kybernetika*, 1999, submitted.

[8] Kárný M., Khailova N., Nedoma P., and Bohm J., "Quantificaton of prior information revised", *Adaptive Control and Signal Processing*, vol. 15, no. 1, pp. 67–84, 1999.

[9] M. Kárný, N. Khailova, P. Nedoma, and J. Böhm, "Quantification of prior information revised", *International Journal of Adaptive Control and Signal Processing*, vol. 15, no. 1, pp. 65–84, 2001.

[10] N. Khailova, M. Kárný, P. Nedoma, and J. Bůcha, "Apriorní znalost pro počítačový návrh adaptivního řízení", *Automa*, vol. 8, no. 10, pp. 45–49, 2002.

[11] N. Khailova, *Exploitation of Prior Knowledge in Adaptive Control Design. Ph.D. Thesis*, PhD thesis, 2002.

17.2 Index

Index

17.3 Recommended identifiers

{cryptony}

Data management	
`TIME`	processing time
`DATA`	data sample
`ndat`	length of data
`psi`	create regression vector
`Psi`	length of data
`npsi`	length of regression vector
`nPsi`	length of data vector
`str`	structure of regression vector

Factors	
`Fac`	factor
`Facs`	array of factors
`fac`	position of a factor in an array of factors
`ychn`	modeled channel
`str`	structure of regression vector
`dfm`	degrees of freedom of a factor
	standard ARX factors
`LD`	degrees of freedom of a factor
`L`	degrees of freedom of a factor
`D`	degrees of freedom of a factor
`V`	information matrix
	ARX factors in least squares representation
`Eth`	point estimate of regression coefficients
`Cth`	covariance of regression coefficients
`cove`	point estimate of noise covariance

Components	
`com`	component
`coms`	array of components
`dfcs`	vector of degrees of freedom of components
`dfcs0`	vector of degrees of freedom of components
`alphas`	normalized vector of degrees of freedom of components
`Com`	matrix ARX or ARX LS component
`Coms`	array of matrix ARX or ARX LS components
`Can`	component in matrix factorized ARX LS form
`Cans`	array of components in matrix factorized ARX LS form
`ychns`	modeled channels in component
`nychn`	number of modelled channels

Mixtures

`Mix`	mixture estimate
`Sim`	mixture simulator
`pMix`	mixture predictor
`pMixfix`	mixture prediction
`facs`	list of factors
`nfac`	number of active factors [a]
`ncom`	number of components
`nchn`	number of modeled channels

[a]dimensions are computed as :
`[ncom, nchn] = size(Mix.coms); nFacs = length(Mix.Facs); nfac = length(Mix.states.facs);`

Mixture estimation

`frg`	forgetting rate
`frgd`	default forgetting rate
`rate`	mixture flattening rate
`maxtd`	maximum time delay of factors in a mixture
`nruns`	number of runs in iterative mixture estimation
`relerr`	relative error
`maxerr`	maximum possible error
	states in mixture estimation [a]
`faclls`	trial factor predictions $log(f(d_{t+1}\|fac, t+1))$
`comlls`	component predictions $log(f(d_t\|com))$
`mixll`	mixture prediction $log(f(d_t\|mix))$
`comwgs`	component weights
`facwgs`	factor weights

[a]refer to mixupdt.m for meaning of the statistics

Mixture projection

`pchns`	predicted channels
`cchns`	channels in condition
`psi0`	channels in condition

Advisory system design

`aMixc`	advised mixture of the type ARX LS + control states
`aMixu`	desired mixture of the type ARX LS + control states
`strc`	common control structure
`kc`	lift of quadratic forms
`UDc`	cell vector of u'du decompositions of KLD kernels
`udca`	u'du decomposition of average KLD kernel in UDc
`kca`	average lift of quadratic forms kc
`uchn`	list of channels with recognisably actions
`pochn`	list of channels with o-innovations
`outs`	list of channels with innovations
`npochn`	number of channels with o-innovations
`udca`	u'du decomposition of average KLD kernel in UDc
`ufc`	normalised vector qualifying components

Structure estimation

`maxstr`	guess of the richest structure
`maxFac`	richest factor
`maxMix`	richest mixture
`belief`	belief on a guess of richest structure
`chbelief`	belief on factors of a channel
`nrep`	number of random starts
`MAPstr`	MAP estimate of the factor structure

General cryptonyms

`DEBUG`	global debugging flag
`chn`	channel (data row)
`std`	standard deviation
`pdf`	probability density function
`kld`	Kullback-Leibler distance
`ll`	Kullback-Leibler distance
`niter`	number of iterations
`opt`	option
`options`	computational options
`seed`	seed of random generator
`uchn`	list of channels with recognisably actions
`sig`	standard deviation of output noise
`CUMTAB`	transition table of components
`ACTIVE`	active component

back

```
zpredict

prodini

echo off
ndat = 3;
DATA = [1:ndat; (1:ndat)/1000]
DATA =
    1.0000    2.0000    3.0000
    0.0010    0.0020    0.0030

% building mixture of one component
str = [1 2 2; 1 0 1];

Fac = facarxls(1, str);
Fac.cove = 1;
Fac.Eth = [10 10 10];
Facs{1} = Fac;

Fac = facarxls(2,[]);
Fac.cove = 1;
Facs{2} = Fac;

Mix = mixconst(Facs, 1:2, 1);
pMix = mix2pro(Mix, 1, 2);

% =========================
TIME = ndat;
psi0 = 100;

% prediction
[Eth, cove, dfcs] = profix(pMix);
Eth
Eth =
   20.0500
10*DATA(1,TIME-1) + 10*DATA(2,TIME) + 10*DATA(2,TIME-1)
ans =
   20.0500

% prediction, psi0 supplied
[Eth, cove, dfcs] = profix(pMix, psi0);
Eth
Eth =
  1.0200e+003
10*DATA(1,TIME-1) + 10*psi0(1) + 10*DATA(2,TIME-1)
ans =
  1.0200e+003

% preprocess data
pre = preproc({'scale', [1 1; 2 2]});

[Eth, cove, dfcs] = profix(pMix);
Eth
```

```
Eth =
  100.1000
10*DATA(1,TIME-1) + 10*DATA(2,TIME) + 10*DATA(2,TIME-1)
ans =
  100.1000

% psi0 is in user's level
% transform it into current scale
psi1 = (psi0+1)*2
psi1 =
   202

% make prediction
[Eth, cove, dfcs] = profix(pMix, psi1);
Eth
Eth =
  2.1000e+003
10*DATA(1,TIME-1) + 10*psi1(1) + 10*DATA(2,TIME-1)
ans =
  2.1000e+003

% transform to the user's level
(Eth-1)/2
ans =
  1.0495e+003

% equivalent
[Eth, cove, dfcs] = profix(pMix, psi0, pre);
Eth
Eth =
  1.0490e+003
```

back

```
zpredict1
prodini

echo off
ndat = 3;
DATA = [1:ndat; (1:ndat)/1000];

prt('DATA            ' , DATA);
DATA
    1  2  3
    0.001  0.002  0.003

% building mixture of one component for prediction
str = [1 2 2; 1 0 1];

Fac = facarxls(1, str);
Fac.cove = 1;
Fac.Eth = [10 10 10];
Facs{1} = Fac;

Fac = facarxls(2,[]);
Fac.cove = 1;
Facs{2} = Fac;

Mix = mixconst(Facs, 1:2, 1);
pMix = mix2pro(Mix, 1);

% ====================
TIME = ndat;

% prediction
[Eth, cove, dfcs] = profix(pMix);
Eth
Eth =
   20.0200
10*DATA(1,TIME-1) + 10*DATA(2,TIME-1)
ans =
   20.0200

% it has no sence to supply psi0
% preprocess data
pre = preproc({'scale', [1 1; 2 2]});

[Eth, cove, dfcs] = profix(pMix);
Eth
Eth =
   80.0400
10*DATA(1,TIME-1) + 10*DATA(2,TIME-1)
ans =
   80.0400

% transform to the user's level
Eth/2 - 1
```

```
ans =
   39.0200

% equivalent
[Eth, cove, dfcs] = profix(pMix, [], pre);
Eth
Eth =
   39.0200
```

back

```
zpredictn
echo on
% === DATA ============================
ndat = 6;
DATA = [1:ndat; 0.1:0.1:0.1*ndat]
DATA =
    1.0000    2.0000    3.0000    4.0000    5.0000    6.0000
    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000
data = DATA;

% === model ===========================
str = [1 2; 1 1];
Fac = facarxls(1, str);
Fac.cove = 1;
Fac.Eth = [10 10];
Facs{1} = Fac;

Fac = facarxls(2,[]);
Fac.cove = 1;
Facs{2} = Fac;

Mix = mixconst(Facs, 1:2, 1);

% =======
TIME = 3;
% =======
% === marginal pdf 1st channel  =======
pMix = mix2pro(Mix,1);

[Eth, coves, dfcs] = profixn(pMix,[], [], 2);
generated data    22  223
Eth
Eth =
   223

% --- internal processing -------------
[Eth,coves,dfcs] = profix(pMix);
Eth
Eth =
    22
DATA(1,TIME) = Eth;

TIME = TIME+1;
[Eth,coves,dfcs] = profix(pMix);
Eth
Eth =
   223

% --- more steps ----------------------
DATA=data; TIME=TIME-1;

[Eth, coves, dfcs] = profixn(pMix,[], [], 3);
generated data    22  223  2234
```

```
Eth
Eth =
        2234

% === marginal distribution ===========
pMix = mix2pro(Mix);

[Eth, coves, dfcs] = profixn(pMix,[], [], 2);
generated data    22  220
Eth{1}
ans =
   220
     0

% internal processing
[Eth,coves,dfcs] = profix(pMix);
Eth{1}
ans =
    22
     0
DATA(:,TIME) = Eth{1};

TIME = TIME+1;
[Eth,coves,dfcs] = profix(pMix);
Eth{1}
ans =
   220
     0

% --- more steps ---------------------
DATA=data; TIME=TIME-1;

[Eth, coves, dfcs] = profixn(pMix,[], [], 3);
generated data    22  220  2200
Eth
Eth =
    [2x1 double]

% ===  pdf 1st channel conditioned by 2nd channel
pMix = mix2pro(Mix,1,2);

[Eth, coves, dfcs] = profixn(pMix,[], [], 2);
generated data    22  223
Eth
Eth =
   223

% internal processing
[Eth,coves,dfcs] = profix(pMix);
Eth
Eth =
    22
DATA(1,TIME) = Eth;
```

```
TIME = TIME+1;
[Eth,coves,dfcs] = profix(pMix);
Eth
Eth =
   223
DATA(1,TIME) = Eth;

[Eth,coves,dfcs] = profix(pMix);
Eth
Eth =
   223

% --- more steps ----------------------
DATA=data; TIME=TIME-1;

[Eth, coves, dfcs] = profixn(pMix,[], [], 3);
generated data    22  223  2234
Eth
Eth =
        2234
```

zpredictn1

back

```
zpredictn1

prodini

echo off

% === DATA ============================
ndat = 6;
DATA = [1:ndat; 0.1:0.1:0.1*ndat]
DATA =
    1.0000    2.0000    3.0000    4.0000    5.0000    6.0000
    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000
data = DATA;

% === model ===========================
str = [1 2; 1 1];
Fac = facarxls(1, str);
Fac.cove = 1;
Fac.Eth = [10 10];
Facs{1} = Fac;

Fac = facarxls(2,[]);
Fac.cove = 1;
Facs{2} = Fac;

Mix = mixconst(Facs, 1:2, 1);

TIME = 3;

pMix = mix2pro(Mix);
[Eth, coves, dfcs] = profixn(pMix,[], [], 3);
generated data    22  220  2200
Eth{1}
ans =
        2200
           0

pMix = mix2pro(Mix,1);
[Eth, coves, dfcs] = profixn(pMix,[], [], 3);
generated data    22  223  2234
Eth
Eth =
        2234

pMix = mix2pro(Mix,1, 2);
[Eth, coves, dfcs] = profixn(pMix,[], [], 3);
generated data    22  223  2234
Eth
Eth =
        2234
```

back

```
zprofixna

prodini

% === DATA ============================
ndat = 6;
DATA = [1:ndat; 0.1:0.1:0.1*ndat]
DATA =
    1.0000    2.0000    3.0000    4.0000    5.0000    6.0000
    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000
data = DATA;

% === model ===========================
str = [1 2; 1 1];
Fac = facarxls(1, str);
Fac.cove = 1;
Fac.Eth = [10 10];
Facs{1} = Fac;

Fac = facarxls(2,[]);
Fac.cove = 1;
Facs{2} = Fac;

Mix = mixconst(Facs, 1:2, 1);
% ===============
TIME = 3;
% ===============
pMix = mix2pro(Mix,1, 2);
[Eth, coves, dfcs] = profixn(pMix, [], [], 2);
generated data    22  223
Eth
Eth =
   223

psi0 = [100; 2; 0]
psi0 =
   100
     2
     0
[Eth, coves, dfcs] = profixna(pMix, psi0, [], 2);
generated data    22  1220
Eth
Eth =
        1220

% --- internal processing -------------
DATA(2, TIME) = 100;
[Eth, coves, weights] =   profix(pMix);
Eth
Eth =
    22
DATA(1, TIME) = Eth;
```

```
TIME = TIME+1;
[Eth, coves, weights] = profix(pMix);
Eth
Eth =
        1220
```

back

Example of structure estimation, SISO case

Example of structure estimation follows. Dynamic SISO model is considered. Data are simulated for insight into processing. The dynamic model is: {oldprior}

```
ychn = 1;                                   % output channel
uchn = 2;                                   % input channel
str  = [ychn ychn uchn uchn; 1 2 0 1];      % factor structure
Eth  = [1.81 -0.8187 0.00468 0.00438];      % regression coefficients
Fac  = facarxls(ychn, str);                 % ARX LS factor
Fac.Eth  = Eth;
Fac.cove = 0.0001;                          % noise covariance
```

The data sample is generated:

```
ndat = 300;                                 % sample size
randn('seed', 7);                           % fix seed
DATA = [zeros(1, ndat)                       % pre-allocated data sample
        0.4*randn(1, ndat)];                 % output std = 0.4
Sim  = mixconst(Fac, 1, 1);                 % build simulator
mixsimul(Sim, ndat);                        % get data sample
```

The task of structure estimation follows. The richest structure is expressed as the model of 6th order and the initial factor is build:

```
maxstr = [ones(1,6), 1+ones(1,7), 0      % richest structure
            1:6,        0:6,        1];
Fac0   = facarx(ychn, maxstr);          % richest initial factor
```

The prior knowledge items considered are:

```
gain = [uchn 0.99 1.01];                    % static gain
tc   = [uchn 0.82 0.84 ];                   % dominant time constant
load ... dlow dhigh                         % load data envelope
load ... data                               % load data sample
```

The data sample of the length 100 is generated by the model above with the regression coefficient $b_0 = 0$. The data envelope is average of 20 realizations of step response specified in two matrices `dlow, dhigh`. The length considered was 150.
The basic prior knowledge items are:

```
pri  = {'gain' gain};
pri  = {'data' data};
pri  = {'fdata' {dlow dhigh} };
pri  = {'st' 0.1 'tcons' tc};
```

Data to be processed by the structure estimation algorithm should be scaled. Accordingly, the prior knowledge must be scaled. This is done:

```
pre  = preproc( {'scale' []} );            % scale data
pri  = scalepri( pri, pre, ychn);          % scale prior knowledge
```

Now, the number of "best" regressors considered and number of repetitive random starts is to be specified:

```
nbest = 10;                                 % number of "best" regressors
nruns = 50;                                 % number of estimation runs
```

Note: the values of 30 for nbest and 50 for nruns are current default values.
The structure estimation is done (the belief is omitted):

```
[Fac, vll] = pristr(Fac0, pri, [ ], nbest, nruns);
```

The optional second output argument is designed for detailed analysis. The possible processing is as follows:

```
wgs  = vll{1};                            % v-log-likelihood with nested prior
wgs1 = wgs/sum(wgs);                      % normalization

wgs  = vll{2};                            % v-log-likelihood with prior
wgs2 = wgs/sum(wgs);

plot(wgs1,'o'); hold on;                  % plot of v-log-likelihood
plot(wgs2,'*');

ilh = vll{3};                             % simplified plot of best structures
for i=1:nbest
    fprintf('%2i',ilh(i,:)); disp(' ');
end
```

The display of "best" regressors shows that the "true" regressor is the 4th one:

```
 1  1  1  1  1  1  2  2  2  2  2  2  2  0 structure
 1  2  3  4  5  6  0  1  2  3  4  5  6  1 probability
------------------------------------------------------
 1  1  0  0  0  0  0  0  0  0  0  0  0  0   0.82800
 1  1  0  0  0  0  0  1  0  0  0  0  0  0   0.15400
 1  1  0  0  0  0  1  0  0  0  0  0  0  0   0.00685
 1  1  0  0  0  0  1  1  0  0  0  0  0  0   0.00656
 1  1  1  0  0  0  0  0  0  0  0  0  0  0   0.00184
 1  1  0  0  0  0  0  0  0  0  0  0  1  0   0.00171
 1  1  0  1  0  0  0  0  0  0  0  0  0  0   0.00042
 1  1  0  0  1  0  0  0  0  0  0  0  0  0   0.00029
 1  1  0  0  0  1  0  0  0  0  0  0  0  0   0.00022
 1  1  1  0  0  0  0  1  0  0  0  0  0  0   0.00020
```

The plots of v-log-likelihood is in Fig. 17.1 and 17.2. The numbers on x-axis coincide with rows of the table above. The normalized v-log-likelihood is displayed on y-axis marked by character 'o' for nested prior knowledge. The probabilities marked by '*' resulted from the use of prior knowledge described in heading.

The simple conclusions can be drawn from the plots:

- static gain, data envelope and data increase probability of the true structure significantly
- time constant brings nothing in this case
- combination of knowledge items does not guarantee significant improvement
- combination of bad and good knowledge do not spoil the result.

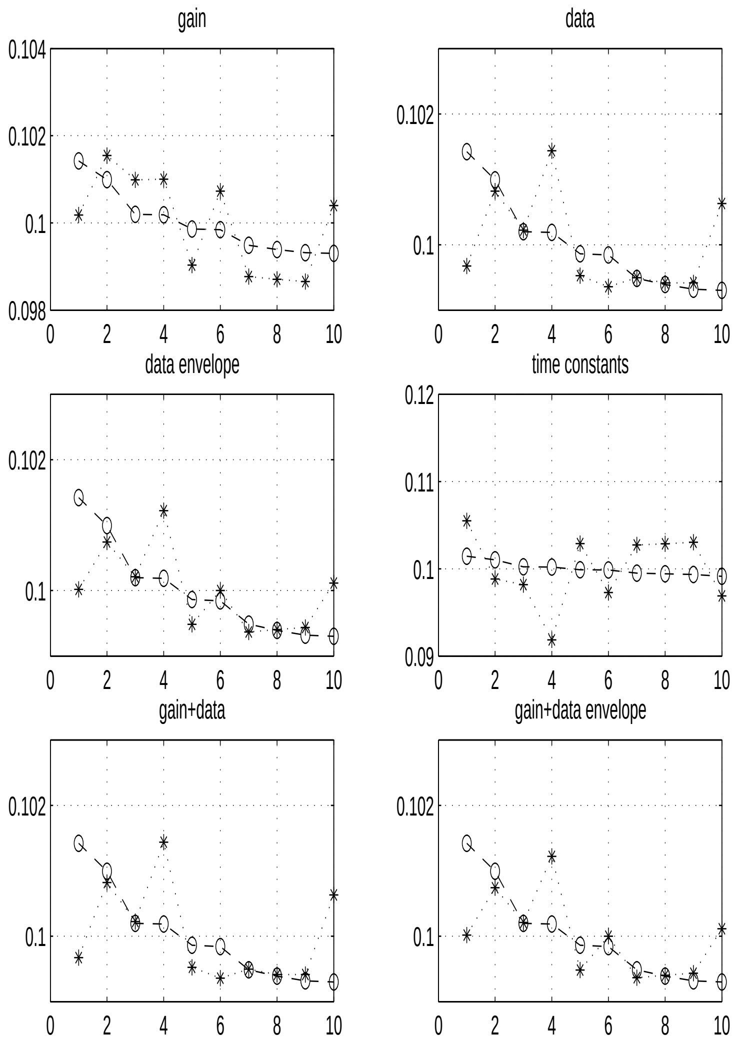
Example of structure estimation, MIMO case

The example is extension of the example of previous subsection - two independent systems are build on four channels. For simplicity, the data are: DATA = [DATA; DATA];

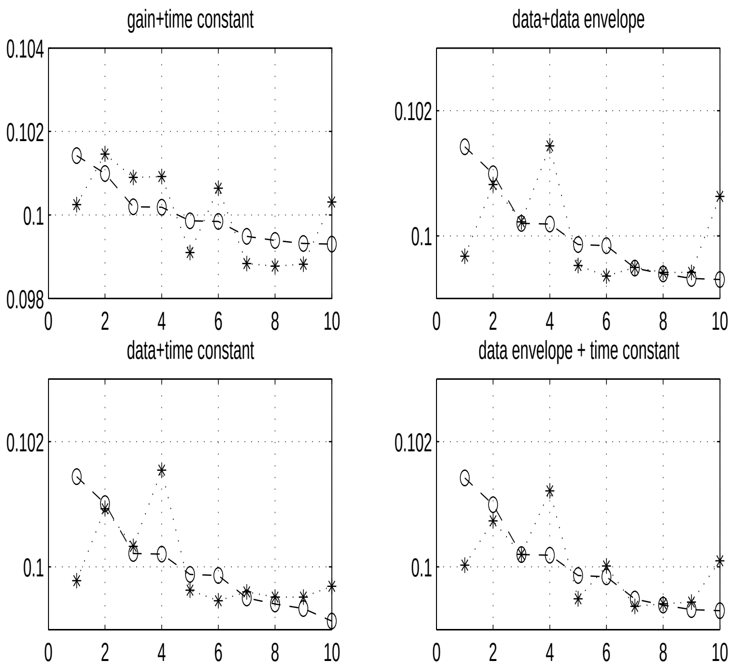
```
DATA = [DATA; DATA];                      % data sample, 4 dimensions
```

The richest initial factors are build:

```
ychn1 = 1;                                % output channel
uchn1 = 2;                                % input channel
maxstr1 = [ones(1,6), 1+ones(1,7), 0       % richest structure, channel1
                 1:6,       0:6,       1];
Fac01    = facarx(ychn, maxstr1);         % richest initial factor channel1
```



{prior1} Figure 17.1: Stucture estimation with prior knowledge



{prior2} Figure 17.2: Stucture estimation with prior knowledge

```
ychn2   = 3;
uchn2   = 4;
maxstr2 = [2+ones(1,6), 3+ones(1,7), 0  % richest structure, channel 3
           1:6,       0:6,         1];
Fac02    = facarx(ychn2, maxstr2);     % richest initial factor, channel 3
Facs0 = {Fac01 Fac02};                 % initial factors
```

The data sample is pre-processed:

```
pre  = preproc( {'scale' []} );        % data are preprocessed
```

The prior knowledge used is:

```
pri  = {'ychn' ychn1 'gain' [uchn1 gain] ...
        'ychn' ychn2 'gain' [uchn2 gain] ...
        'fdat' {dlow dhigh} };
```

The prior knowledge must be scaled. It is done simply by:

```
pri  = scalepri(pri, pre);             % preprocess prior knowledge
```

The structure is estimated and results displayed:

```
nbest = 10;                            % number of "best" regressors, default is 30
nruns = 50;                            % number of estimation runs
[Facs, vlls] = pristr(Facs0, pri, [], nbest, nruns);
```

```
Facs = arx2arx(Facs);
Fac1 = Facs{1};
Fac1.str, Fac1.Eth
ans =
     1     1     2     2
     1     2     0     1
ans =
    1.7624   -0.7731    0.0119    0.0146
```

```
Fac2 = Facs{2};
Fac2.str, Fac2.Eth
ans =
     3     3     4     4
     1     2     0     1
ans =
    1.7624   -0.7731    0.0119    0.0146
```

Note: in simple cases, the structure estimation can be done channel-wise. The advantage of processing outlined is that the data increment of initial factors is computed only ones.

Prior knowledge and channel description

A brief excursion into the channel description approach. The example of the previous section is continued.

Before the structure estimation starts, the channel description is build

```
Chns = chnconst(1:4);                   % channel description
```

The scaling of individual channels is done and the scaling of individual channels is recorded:

```
pre  = preproc( {'scale' []} );         % data are preprocessed
Chns = chnset(Chns, 'scale', pre);      % set channel scaling
```

Prior knowledge is scaled and recorded in the channels structure:

```
pri  = scalepri( pri, pre);             % scale prior knowledge
Chns = chnset(Chns, 'prior', pri);
Chns{1}.prior
ans =
    'ychn'    [1]    'gain'    [1x3 double]
Chns{3}.prior
ans =
    'ychn'    [3]    'gain'    [1x3 double]    'fdat'    {1x2 cell}
```

The structure estimation is done. The `pri` argument can be replaced by the channel description.

```
nbest = 10;                             % number of "best" regressors
nruns = 15;                             % number of estimation runs
[Facs, vlls] = pristr(Facs0, Chns, [], nbest, nruns);
Facs = arx2arx(Facs);
Facs{1}.str
ans =
     1     1     2     2
     1     2     0     1
Facs{1}.Eth
ans =
    1.7624   -0.7731    0.0119    0.0146
```

Note: in future design, the scaling of prior knowledge can be done automatically inside the function `chnset`*. It the* `pri` *argument is empty, it will be substituted by a global channel description.*

back