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., Böhm, J., Guy, T. V., Jirsa, L., Kárný, M., Nagy, I., Tesař, L., Andrýsek, J.

Mixtools:User's Guide

No. 2060 November 2002

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

ÚTIA AV ČR, P. O. Box 18, 182 08 Prague, Czech Republic
Telex: 122018 atom c, Fax: (+420)(2)6884 903
E-mail: utia@utia.cas.cz

MixTools

Toolbox for Mixtures

User's Guide

EU project IST ProDaCTools No. IST-1999-12058

Contents

Acknowledgment

The work described in this report was done with support of EU project IST ProDaCTools No. IST-1999-12058

1 Introduction

This guide deals with software aspects of the probabilistic advisory system relying on Bayesian estimation and prediction of a finite mixture of probability density functions (pdf) based on a sample of multivariate data. At present, it serves for developers of the system and their nearest co-workers. Guides for a wider set of users will be created after stabilizing the whole system. The underlying theory is described in [?]. Knowledge of Bayesian learning related to auto-regression models with external inputs (ARX) and basic knowledge of normal mixtures are necessary pre-requisite for understanding various parts of the text.

Section 2 serves as a bridge between theoretical notions and their software images. Then, the Section 3 provides general software representation and its specification for basic normal mixtures. Other representations, like Markov chains will be gradually added.

The adopted data management is characterized, Section 5. Important data preprocessing is dealt with in Section 6. The Mixtools functions are summarized in the Section 7.

Knowledge of the introduced notions allows to follow tutorial on mixture handling, Section 8.

The rest of sections include detailed discussion of selected topics – initialization of mixture estimation, Section 9, approximate mixture parameter estimation, Section 10, mixture prediction, Section 11, visualization, Section 12, mixture simulation, Section 17 and estimation of mixture factors, Section 18.

The use of the toolbox functions is documented in the form of MATLAB diaries. The identifiers used follow coding conventions summarized in subsection 20.3.

intro.tex, MK, PN June 25, 2004

2 Bridge between theory and software

{scenarion}

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 [?] 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,
$f(\cdot\|\cdot)$	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 5.

- *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 20.3, 20.4.
- 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

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

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 $\Theta_c,\ c \in c^*$, of components. The components are decomposed by the chain rule

$$f(d_t|d(t-1), \Theta_c, c) = \prod_{i \in i^*} f(d_{i;t}|\psi_{ic;t}, \Theta_{ic}, i, c), \quad (2) \quad \{\texttt{MKfactors}\}$$

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) 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 [**?**] in order get the posterior pdf $f(\Theta|d(\mathring{t}))$

$$f(\Theta|d(\mathring{t})) \propto f(d(\mathring{t})|\Theta) f(\Theta). \quad (3) \quad \{\texttt{MKpostpdf}\}$$

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

$$f(d|\psi, d(\mathring{t})) = \int f(d|\psi, \Theta) f(\Theta|d(\mathring{t}))\, d\Theta. \quad (4) \quad \{\texttt{MKpredict}\}$$

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 ${}^{[I]}f(d(\mathring{t}))$ is the closest to the user ideal pdf (user target) ${}^{[U]}f(d(\mathring{t}))$ 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 distance (KLD)* is employed. It can be expressed as additive loss function summing conditional KLD

$$\omega(a_t, d(t-1)) \equiv \int^{[I]} f(d_t|a_t, d(t-1)) \ln\left(\frac{{}^{[I]}f(d_t|a_t, d(t-1))}{{}^{[U]}f(d_t|a_t, d(t-1))}\right) dd_t, \quad (5) \quad \{\texttt{cKLD}\}$$

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

Advises, i.e. *actions available to p-systems*

$$a_t \equiv (c_t, u_{o;t}, s_t, p_t) \text{ are interpreted as follows.} \quad (6) \quad \{\texttt{5pact}\}$$

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 lists* or just *lists*.

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	labls 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

$$f(\alpha) \equiv Di_{\alpha}(\kappa) \propto \prod_{c \in c^*} \alpha_c^{\kappa_c - 1}. \qquad (7)$$ {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

$$f(d|\psi, \Theta) = \mathcal{N}_d(\theta'\psi, r) = \ (2\pi r)^{-0.5} \exp\left\{-\frac{1}{2r}\left([-1, \theta']\Psi\right)^2\right\}, \text{ where} \qquad (8)$$ {MKnor}

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

The factor output d is coded by the channel number "ychn" pointing to row of global data matrix "DATA", see Section 5. 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 (8) is Gauss-inverse-Wishart (GiW) pdf [?]

$$f(\Theta) = GiW_{[\theta,r]}(L, D, \nu) \propto r^{-\frac{\nu}{2}} \exp\left\{-\frac{1}{2r}[-1, \theta']L'DL[-1, \theta']'\right\}, \text{ where} \qquad (9)$$ {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

$$L \ \equiv \ \left[\begin{array}{cc} 1 & 0 \\ L_{d\psi} & L_{\psi} \end{array}\right], D = \text{diag}[D_d, D_{\psi}], \ D_d \text{ is scalar} \qquad (10)$$ {MKsplitLD}

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

$$\begin{aligned} \hat{\theta} \ &= \ L_{\psi}^{-1}L_{d\psi} \text{ is LS estimate of } \theta, \ \ \text{stored as "Eth"} && (11) \\ \hat{r} \ &= \ \frac{D_d}{\nu} \text{ is LS estimate of } r \text{ stored as "cove"} && (12) \\ & \hat{r}L_{\psi}^{-1}D_{\psi}^{-1}(L_{\psi}')^{-1} \text{ is covariance matrix of the LS estimate of } \theta \\ & L'DL\text{decomposition of } L_{\psi}^{-1}D_{\psi}^{-1}(L_{\psi}')^{-1} \text{ is stored as "Cth".} \end{aligned}$$ {MKLS}

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 $\mathcal{S}_{i;t} = [\hat{\theta}_{i;t}, \hat{r}_{i;t}, L_{\psi i;t}^{-1}, D_{\psi i;t}^{-1}, \nu_{i;t}]$ 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 [?] with moments

$$\hat{d}_i \ = \ \mathcal{E}[d_i|\psi, S] = \hat{\theta}'\psi, \ \hat{r}_d = \text{cov}[d_i|\psi, S] = \hat{r}(1 + \zeta), \ \zeta = \psi'L'DL\psi. \qquad (13)$$ {MKstudent}

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 distances

{coKLD}

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

$$k + \psi' LDL' \psi, \quad \text{where} \tag{14}$$ {cKLDN}

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 (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

$$^{[U]}f(c_t|d(t-1)) \propto ^{[U]} f(c_t) \exp\left[-0.5(^{[U]}k_{c_t;t-1} + \phi'_{t-1}{}^{[U]}L_{c_t;t}\,{}^{[U]}D_{c_t;t}\,{}^{[U]}L'_{c_t;t}\phi'_{t-1})\right], \tag{15}$$ {ufc}

where $^{[U]}f(c_t)$ 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 (15) defines preferences among pointers to components.

scenario.tex, written by MK June 25, 2004

3 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 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.

3.1.1 Coding of estimation results

Estimation describes distribution of parameters, formally *GiW* pdf (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 (9)
2 ARX LS factor – corresponding to the LS (least-squares) form (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

3.1.2 Coding of prediction results

Prediction describes distribution of data, formally Student distribution with moments (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 11. In the projection, the original states are changed.

3.2 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 11;
- *conversions* from other existing p-forms.

3.3 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 *factor 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.

3.3.1 ARX factor

The ARX factor is described by (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 (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 $V = L'DL$.

3.3.2 ARX LS factor

The least squares representation (LS) of an ARX factor, *ARX LS factor*, deals with the LS form of the sufficient statistics (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".

3.4 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.

3.4.1 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.

3.4.2 ARX LS components

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

3.4.3 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* $d_{i+1;t}, \ldots, d_{\mathring{i};t}$
str	(2-by-npsi)	% *regression-vector structure common for all factors*
dfm	(1-by-1)	% *degrees of freedom* $\nu - 2$
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
```

3.4.4 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 (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.

3.5 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 (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".

3.5.1 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
```

3.5.2 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.

3.5.3 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
```

3.5.4 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.

3.5.5 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

3.6 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 11, 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]
```

mixrepr.tex by MK, PN June 25, 2004

4 Software representation of advisory mixtures

{representa}

Mixtools implements algorithms that transform user's aims ${}^{[U]}f(d(\mathring{t}))$ and data d into the ideal mixtures ${}^{[I]}f(d(\mathring{t}))$ that are presented to the user. Mixtures are learned from the data [?]. 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 ${}^{[U]}f(d)$, see Section 15).

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 11) 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 (15). 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 recognisable 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 (see Section 14)
pochn	list of channels with o-innovations
uchn	list of channels with recognisable actions (can be omitted for academic design).

Example of use "inisyn" can be found Appendix C.

mixrepra.tex by TG June 25, 2004

5 Data management

{dataman}

Management of data samples is discussed.

5.1 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:

```
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 modeled channel.

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

An example of use of the function:

```
global TIME DATA                        % data management global matrices
DATA = [1:6;11:16]                      % data sample
DATA =
     1     2     3     4     5     6
    11    12    13    14    15    16


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


for TIME = 3:length(DATA)               % time loop starts for TIME>maxtd
    dvect = getdvect(Fac);              % get data vector
     disp(dvect);                       % display data vector
end
     3     2     1    13    12
     4     3     2    14    13
     5     4     3    15    14
     6     5     4    16    15
```

The second possibility is e.g.:

```
for TIME = 3:length(DATA)               % time loop starts for TIME>maxtd
    dvect = getdvect(str);              % get regression vector
     disp(dvect);                       % display regression vector
end
     2     1    13    12
     3     2    14    13
     4     3    15    14
     5     4    16    15
```

5.2 Huge data sample processing

{huge}

When we process an extremaly *huge data sample*, we are forced to use *buffered estimation* because such data sample can hardly be managed in the MATLAB workspace. This possibility is available with any estimation function (implemented as MEX function) and "mixinit".

The data sample is supposed to be written on disk e.g. by:

```
file  = fopen('filename','wb');         % open a file
fwrite(file, DATA, 'double');           % write DATA
fclose(file);                           % close the file
mdat  = size(DATA,1);                   % number of data rows
```

Instead of the usual argument "ndat" (length of data sample), 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 e.g.

```
DATA = zeros(mdat, e14);                 % buffer allocation
```

data.tex by PN June 25, 2004

6 Data preprocessing

{preprocPN}

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 correspondingly.

6.1 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.

Both input data sample and preprocessed data sample are located in the global matrix "DATA".

The identifier used for the preprocessing list is "`pre`".

6.2 Preprocessing run

The batch preprocessing is done by:

```
pre = preproc(pre);                      % batch preprocessing
```

where the output cell list "pre" is the run-time preprocessing list (used for backwards transformation of processing results).

The recursive preprocessing consists of two steps. The first one is *preprocessing initialization* done by:

```
pre = preinit(pre);                      % initialization of preprocessing
```

The initialization checks integrity of "pre", adds defaults, creates states and returns run-time preprocessing list.

One step of recursive preprocessing is done by the function "prestep". The step where the preprocessing is done is controlled by the global TIME:

```
TIME = ...                               % set preprocessing "time"
pre = prestep(pre);                      % one preprocessing step
```

The the run-time preprocessing list can be modified at each processing step.

6.3 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`

The operation is described in the Section 11.4.4. An example:

```
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.

6.4 Filters

Preprocessing algorithms are described in [**?**].
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

6.4.1 Remark to filter usage of

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.

6.5 Examples

We illustrate usage of data preprocessing by examples.

6.5.1 Example 1

The most common task of preprocessing is demonstrated - removal of outliers and normalization.

Data sample is generated:

```
DATA = (randn(1, 300)+1)*2;             % simulated data sample
```

Simulated outliers are made:

```
val = 10;
for n=50:50:ndat
    DATA(1, n) =  val;
    val = val+1;
end
plot(DATA);
```

The data are plotted in Fig. 1, left side.

To remove the outlier, `olymedianf` is used. After it, data normalization is required using `scale`:

```
pre = {'olymedianf',{'c',1,'level',5 ,'scale',[]};  % preprocessing requirements
pre = preproc(pre);                     % batch preprocessing
plot(DATA);                             % plot data
```

The data are plotted in Fig. 1, right side.

6.5.2 Example 2

Removal of a high frequency noise is illustrated. The signal is sinus based with a high frequency noise:

```
DATA = sin(0:0.1:35);                   % deterministic signal
DATA = DATA + 0.5 *randn(size(DATA));   % high frequency noise is added
plot(DATA);                             % plot of signal
```

The data sample is plotted in Fig. 2, left side.

The noise is removed by the "medianf" filter:

```
pre = {'medianf', {'c', 1, }, 'forgetting_rate', exp(-1/10) } };
pre = preproc(pre);                     % batch preprocessing
plot(DATA);                             % plot data
```

The data sample is plotted in Fig. 2, left side.

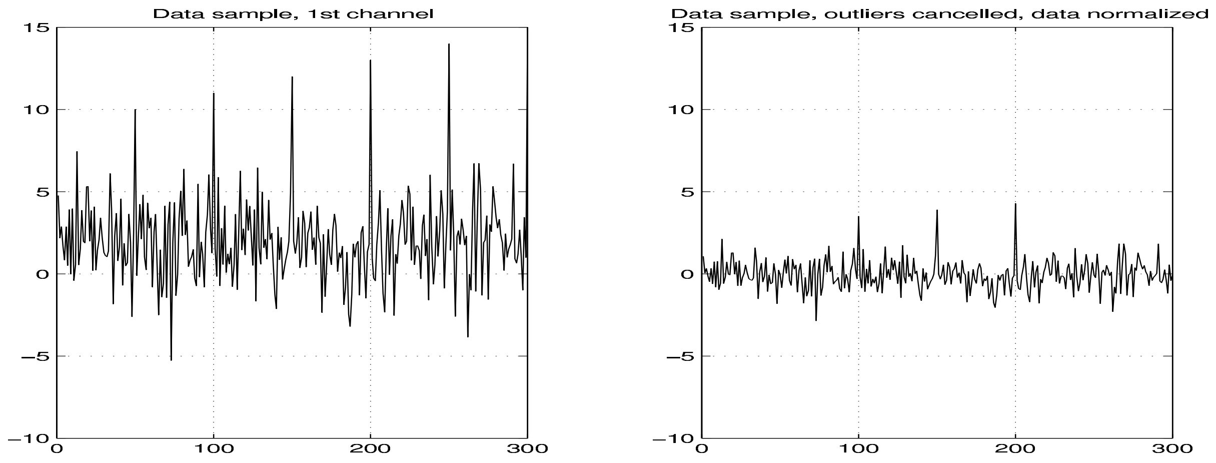

Figure 1: Data sample, channel 1 {zpre1}

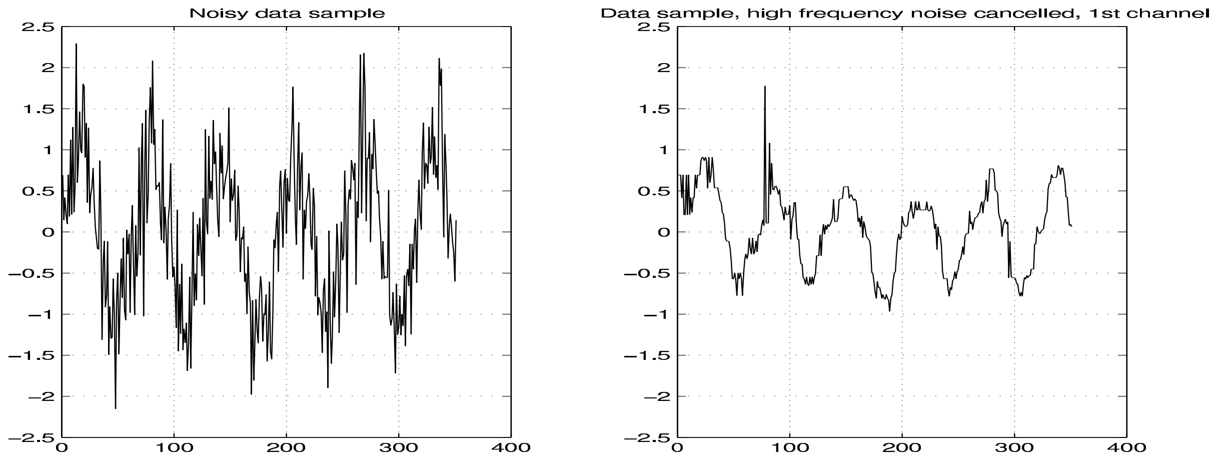

Figure 2: High frequency noise removal {zpre2}

7 Mixtools functions

{functions}

Mixtools functions can be roughly divided in the group of *Mixtools user's functions* that are used by ordinary users for mixture analysis. The rest of the Mixtools functions form a *Mixtools design base*. It is listed in Apendix.

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".

Majority of 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.

7.1 Function arguments

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`	normalised 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 recognisable 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

7.2 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[a]
`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

[a]estimation options: 'q', 'b', 'f', 'm' 'n'+ number of iteration steps; belief expresses user's belief into the regressor specified

Estimation operations

`Mix  = ...` `mixest(Mix0, frg, niter, opt)`	iterative mixture estimation[b]
`Mix  = mixestim(Mix0, frg, ndat)`	quasi-Bayes mixture estimation
`Mix  = mixestim(Mix0, frg)`	recursive quasi-Bayes mixture 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

[b]opt - options: 'q', 'b', 'f','m' for quasi-Bayes, batch Bayes, forgetting branching and estimation with fixed covariances

Prediction operations

`pMix = mix2mixm(Mix, pchns)`	build marginal predictor
`pMix = mix2pro(Mix, pchns, cchns)`	build/re-build mixture projector
`pMix = profix(pMix, psi0, pre)`	build mixture prediction from projector
`pMix = ...` `mixpro(Mix0,pchns,cchns,psi0,pre)`	build mixture projection[a]
`[pMix, weights] = ...` `profixn(pMix, psi0, pre, nstep)`	prediction n-steps ahead [b]

[a]defaults: pchns - [1,2], cchns - no, psi0 - substituted from DATA, no data scaling
[b]the weights are data dependent even for static mixtures

Visualization [a]

[a]defaults: see Prediction operations. The functions allows definition of grid densities and ranges

`mixplot (Mix,pchns,cchns,psi0,pre)`	mixture plot (shaded)
`mixplotc(Mix,pchns,cchns,psi0,pre)`	mixture plot (contours, components)
`[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

[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

Structure estimation

`Mix = ...` `mixstrid(Mix,Mix0,belief,nruns)`	estimate mixture structure
`MAPstr = ...` `facstrid(Fac,Fac0,belief,nbest,nruns)`	estimate structure of a factor

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

Basic conversion functions

`LD   = ltdl(V)`	decompose positive definite matrix to L'DL
`Mix  = mix2mix(Mix, form)`	convert mixture to a specified form[a]
`Com  = com2com(Com, form)`	convert component into a specified form
`X    = arx2arx(X)`	convert between ARX and ARX LS representations

[a]"form" is a coding summarized in "Codes"

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)`	generate normalized vector qualifying unstable components
`aMix  = ...`	
`soptim(aMix,aMixu,ufc,nstep,chis)`	perform simulaneous advisory desing for normal mixtureperform
+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
`Mix   = mixcopy(Mix1, Mix2)`	copy of ARX or ARX LS statistics

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
`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
`mversion`	display current Mixtools version

funlistu.tex by PN June 25, 2004

fnc.tex by PN June 25, 2004

8 Tutorial on mixture simulation, initialization and estimation

{tutorial}

Two case studies of mixture simulation, initialization and estimation are discussed.

8.1 Case study: static mixture

{tutors}

A mixture of four static components is considered.

8.1.1 Simulation

{tests}

A length of the data sample, number of components and common noise variance of the components are selected.

```
randn('seed', 135531);              % seed of random number generator
ndat = 1000;                        % size of data sample
ncom = 4;                           % number of components
cove = [0.1 0.01; 0.01 0.1];        % common component noise variance
```

The static matrix ARX LS component for channels 1 and 2 is build. The covariance is held in the form of L'DL decomposition.

```
ychns    = [1 2];                   % modelled channels
str      = [0;1];                   % common static structure
Com      = comarxls(ychns, str);    % matrix ARX LS component
Com.cove = ltdl(cove);              % L'DL of noise covariance
```

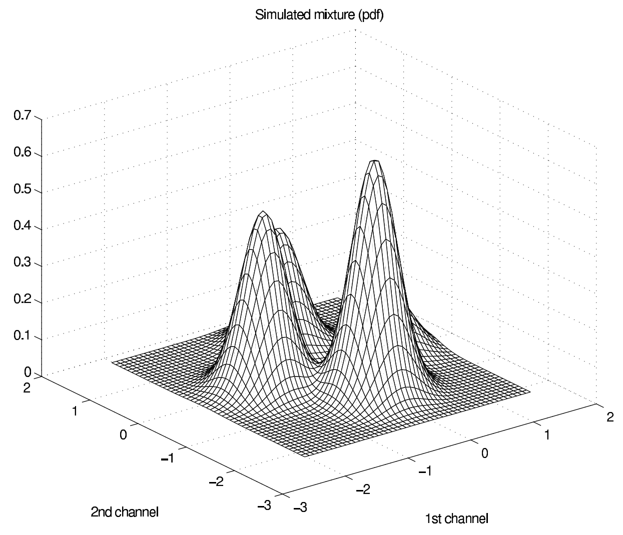

Figure 3: Mesh plot of simulator {ztutors1}

Array of four components with means on unit circle is build.

```
Eths     = [1 0 -1 0; 0 1 0 -1];          % mean of components (column-wise)
for i=1:ncom, Com.Eth = Eths(:,i);
    Coms{i} = Com;                        % array of components
end
```

The component degrees of freedom are defined and the mixture simulator is build:

```
dfcs = 1:ncom;                            % degrees of freedom of components
Sim  = mixconst(Coms, dfcs);              % build mixture simulator
```

Before the simulation starts, the global matrix DATA must be *pre-allocated.* Then the data sample is generated.

```
DATA    = zeros(2, ndat);                 % pre-allocate data
mixsimul(Sim, ndat);                      % get simulated data sample
```

The simulator is displayed in Fig. 3 in the form of mesh plot. The mixture simulator is internally converted to mixture predictor when displayed, see Section 11.

```
[x,y,z] = mixgrid(Sim);                   % get coordinates
meshc(x,y,z);                             % mesh plot
```

The plot of components and contours of the simulator pdf is presented on Fig. 4. The components (left) are plotted for 75 % probability region.

```
mixplotc(Sim);                            % plot of mixture
```

The way of presentation of figures in this guide is based on Matlab *subplots.* The series of subplots starts.

```
sub = 230;
sub=sub+1; subplot(sub);                  % define subplot
contour(x,y,z,15);                        % contour plot

sub = sub+1; subplot(sub);
datascan([1;2],1);
```

The mixture simulator is displayed in Fig. 5, subplot 1 and the data clusters are in Fig. 5, subplot 2.

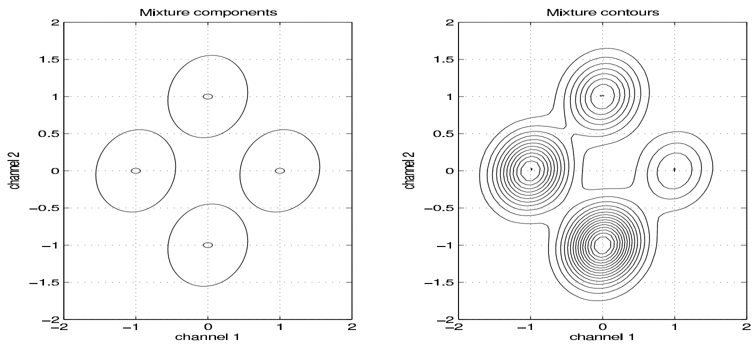

Figure 4: Simulator components and contour plot {ztutors2}

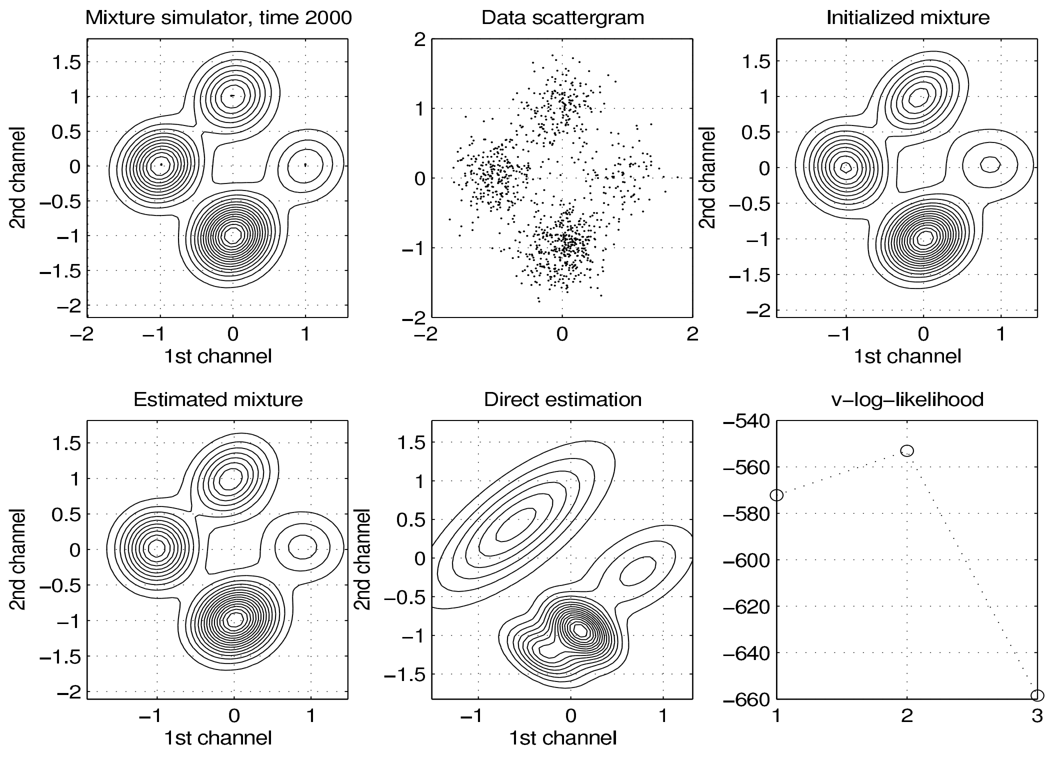

Figure 5: Static mixture simulation, initialization and estimation {ztutors}

subplot	contains
1	mixture simulator
2	data clusters
3	initialized mixture
4	estimated mixture
5	mixture estimated without initialization
6	v-log-likelihood of individual steps

8.1.2 Initialization of mixture estimation

The *initialization of mixture estimation* is the first and crucial step in mixture processing. It consists of search for the mixture structure that maximizes the posterior data likelihood on the learning data sample.

The first step is to build an initial mixture using prior knowledge available. The only prior knowledge used here is that the mixture is a static one.

It is not easy to specify initial mixture correctly. The experience of the authors is encoded in the function "*genmixe*" (discussed in the next subsection. Here, it is used with default values that depend on the data sample. An initial static mixture with one component is build.

```
Mix0 = genmixe;                       % build initial mixture
```

The defaults selected by the function "genmixe" expects *normalized data sample.* Here, the sample normalization step is not discussed as the data have reasonable scaling.

The initialization needs a *forgetting rate* for internal mixture estimation. The value recommended is obtained by calling the function "defaults" that summarizes all Mixtools defaults.

The initialization does the function "mixinit". It is an iterative procedure. We use 5 iterations here which is usually enough.

```
niter = 5;                            % maximum number of iterations
frg   = defaults('frg');              % default forgetting rate
Mix   = mixinit(Mix0, frg,ndat,niter); % mixture initialization
mixlls(1) = Mix.states.mixll;   % save value of v-log-likelihood
```

The resulting mixture is displayed in the Fig. 5, subplot 3. The v-log-likelihood is saved for later display.

8.1.3 Mixture estimation

It is recommended to make an iterative mixture estimation after initialization. A better mixture quality is usually reached. The initialized mixture must be *flattened* before the estimation. The number of iteration is set to 20. The resulting mixture is displayed on Fig. 5, subplot 4.

```
niter = 20;                           % number of iterations
Mix0  = mixflat(Mix);                 % mixture flattening
Mix   = mixest(Mix0, frg, ndat, niter);% iterative mixture estimation
mixlls(2) = Mix.states.mixll;
```

The initialization phase is recommended but not obligatory. The mixture estimation can be done without initialization starting from an initial mixture designed using prior knowledge available. The prior knowledge used here is:

- the mixture is static
- it has 4 component (via inspection of data scattergram)

The initial mixture for estimation is build:

```
Mix0  = genmixe(4);                   % build initial mixture
```

The iterative mixture estimation is made:

```
niter = 20;                           % number of iterations
frg = defaults('frg');                % default forgetting rate
Mix = mixest(Mix0, frg, ndat, niter); % iterative mixture estimation
mixlls(3) = Mix.states.mixll;         % record of v-log-likelihood
```

The resulting estimated mixture is displayed in Fig. 5, subplot 5. The values of the v-log-likelihood are in Fig. 5, subplot 6.

8.1.4 Test function "statsim"

{statsim}

In this guide, we use "*statsim*" function that generates static mixture simulator and data sample.

Length of the data sample, number of components and common noise variance of the components are to be specified. The components have means located on unit circle. The component weights linearly increases.

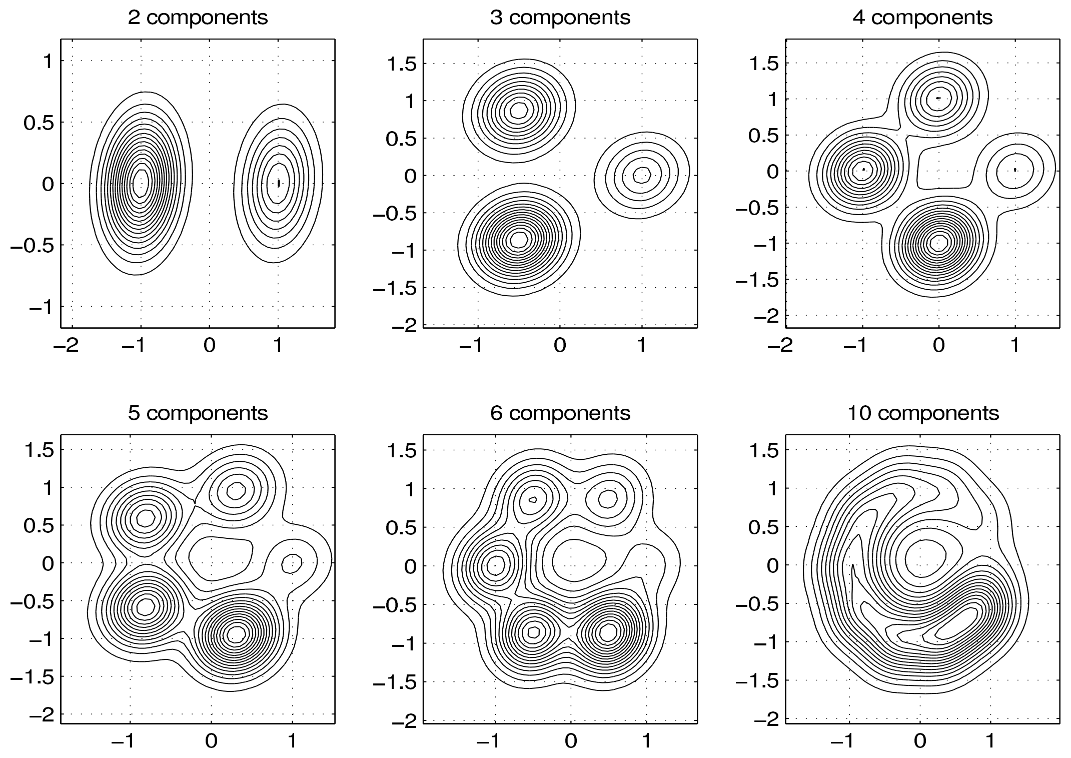

Figure 6: Mixtures made by the function *statsim* {zstatsim}

In example, different simulators are generated and displayed in Fig. 6.

```
cove = ltdl([0.1 0.01; 0.01 0.1]);        % L'DL of component noise variance
ndat = 0;                                 % without generating data sample

sub = 230;
for ncom = 2:7
    Sim  = statsim(ndat, ncom, cove);
    sub  = sub + 1; subplot(sub);
    [x,y,z] = mixgrid(Sim);
    contour(x,y,z,15);
end
```

tutors.tex ztutors.m zstatsim.m by PN June 25, 2004

8.2 Dynamic mixture example

{tutord}

Case study with a *dynamic mixture* is discussed. The mixture consist of two components that have a common factor.

8.2.1 Simulation

{testd}

Simulated data are generated by two components for modelled channels 1 and 2 (referred to here as "output" and "input" channel). The components switch with the mixing probability "alpha".
The dynamic factors in the components have a common structure.

```
ychn = 1;                                 % output channel
uchn = 2;                                 % input channel

str = [1 1 1 1 2 2;  1 2 3 4 3 4];        % common structure
Fac = facarxls(ychn, str);                % build ARX LS factor
```

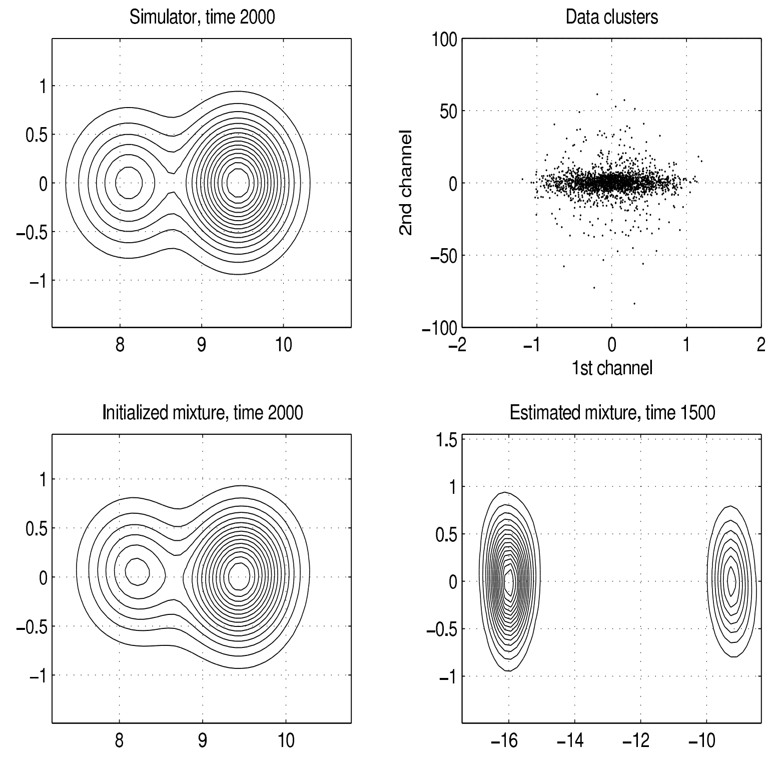

Figure 7: Case study: dynamic mixture {**ztutord**}

subplot	contains
1	mixture simulator at time 2000
2	data clusters
3	initialized mixture at time 2000
4	estimated mixture at time 1500

The array of dynamic factors is build:

```
Fac.Eth  = [1.41833  -1.5894  1.3161  -0.88642  0.2826  0.50666];
Fac.cove = 0.39463^2;                   % variance of output noise
Facs{1}  = Fac;

Fac.Eth  = [2.0968  -2.3196  1.9335  -0.8713  0.641  0.1041];
Fac.cove = 0.37255^2;                   % variance of output noise
Facs{2}  = Fac;
```

The input is modeled as a noise:

```
Fac = facarxls(uchn,[]);                % model of input channel for simulation
Fac.cove = 0.16;                        % variance of input noise
Facs{3}  = Fac;
```

Mixture simulator is build and data sample generated.

```
alpha = 0.3;                            % switching probability
dfcs0 = [alpha, (1-alpha)];             % degree of freedom of components
Sim   = mixconst(Facs,[1 3; 2 3],dfcs0); % mixture simulator

ndat = 2000;                            % sample size
DATA = zeros(2, ndat);                  % pre-allocated DATA
mixsimul(Sim, ndat);                    % generate data sample
TIME = ndat;                            % fix processing time
datascan([2;1], 1);                     % scattergram of data
```

The simulator is displayed in Fig. 7, subplot 1. The data scattergram is displayed in Fig. 7, subplot 2. The "TIME" must be specified as the mixture is a dynamic one. The matrix "DATA" must be pre-allocated before the sample is generated.

8.2.2 Initialization of mixture estimation

{Ggenmixe}

The *initialization of mixture estimation* starts here from an initial mixture consisting of one component. The prior knowledge used is that the mixture is dynamic one and the dynamic can be expressed by a *richest* mixture below.

```
maxstr   = [1 1 1 1 1 1  2 2 2 2 0; 1 2 3 4 5 6  3 4 5 6  1];
```

The initial mixture consisting of single component is build.

```
Mix0     = genmixe(1,[ychn uchn], maxstr);
```

The function "*genmixe*" is called as

`Mix0 = genmixe(ncom, ychns, str, ndat, ...)`	build initial mixture

The arguments together with defaults are:

argument	*meaning*	*defaults*
`ncom`	number of components	1
`ychns`	modeled channels	1 : `size(DATA,1)`
`str`	component structure	[0;1], ie. the static one
`ndat`	size of data sample	`size(DATA,2)`

The output mixture is of the matrix ARX LS type. The function have additional arguments that refer to setting the initial components fields, see help on the function.
Number of iterations and default forgetting rate are specified and initialization made.

```
niter = 5;                              % number of iterations
frg   = defaults('frg');                % default forgetting rate
Mix   = mixinit(Mix0, frg, ndat, niter); % mixture initialization
```

The result of initialization is:

```
  Mix.states.mixll
  ans =
   -454.3336
  ncom = length(Mix.dfcs)                % number of components
  ncom =
       2
  Mix.dfcs/sum(Mix.dfcs)                 % component weights
  ans =
      0.3021    0.6979
  M=mix2mix(Mix, 22);
  M.Facs{1}.str
  ans =
       2     1     1     1     1     1     1     2     2
       0     1     2     3     4     5     6     3     4
  M.Facs{1}.Eth
  ans =
     -0.0898    1.3883   -1.5018    1.1969   -0.7791   -0.0647    0.0197    0.2928    0.5417
  M.Facs{3}.str
  ans =
       1     1     1     1     2     2
       1     2     3     4     3     4
  M.Facs{3}.Eth
  ans =
      2.0966   -2.3211    1.9342   -0.8708    0.6357    0.0982
```

The estimated weights are almost equal to the simulated ones, compare estimated structure and system parameters with the known simulation model.
The plot of the initialized mixture is displayed in Fig. 7, subplot 3.

The common factor is not discovered by "mixinit" – it is limitation of the current version. The initialized mixture is close to the simulated one.

8.2.3 Mixture estimation

The mixture quality is raised by the iterative mixture estimation done after initialization. The initialized mixture must be flattened before the estimation. Ten iterations of quasi-Bayes estimation is chosen.

```
  Mix0  = mixflat(Mix);                  % mixture flattening
  niter = 10;                            % number of iterations
  Mix   = mixest(Mix0, frg, ndat, niter);% iterative mixture estimation
  Mix.states.mixll                       % posterior data likelihood
  ans =
   -431.0828
  TIME = ndat;                           % fix processing time
  ...  plot of estimated mixture...
```

The estimated mixture at terminal TIME = 1500 is displayed in Fig. 7, subplot 4.

tutord.tex ztutord.m by PN June 25, 2004

9 Initialization of mixture estimation

{mixinit}

The initialization searches for the mixture structure (i.e.number of mixture components, the structure of mixture factors etc.) that maximize the posterior data likelihood on a learning data sample. The initialization is made by the function "*mixinit*".

The relevant theory is in [**?**], simple examples of use in the Section 8. This section contains summary of possibilities and recommendations on use of "mixinit".

The function "mixinit" is called as

`Mix = mixinit(Mix0, frg, ndat, niter, options, belief)`	initialization

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 initial (or flattened) mixture. It should be created using all prior information available (e.g. static or dynamic mixture etc.). It is recommended to use the function "genmixe" for the purpose - see the section 8.2.1.

`frg` is a forgetting rate. It should be selected as the default forgetting rate. The only exception is in 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 belief is expressed channel-wise as a cell vector. Each cell can be empty or contain a matrix of three row. The first and second rows specify regressor item (each column of the structure). The third line specifies:

1	regressor item is present	3	regressor item is probably not preset
2	regressor item is probably present	4	regressor item is not present

An example follows. We have a dynamic mixture with 2 channels to be processed by "mixinit". In the process of initialization, a new component is generated and relevant structure estimation is done using the belief on individual channels.

The belief expressed as

```
belief1 = [1 1 1  2 2               % belief for the 1st channel
           1 2 3  0 1];
           2 2 3  1 2];
belief2 = [2 ; 1; 2];               % belief for the 2nd channel
belief  = {belief1, belief2};       % overall belief
```

can be interpreted in the following way. The factors that model the 1st channel are believed to contain surely the structure item $[1; 1]$, probably (with different beliefs) the items $[1\ 1; 2\ 3]$, surely the structure item $[2; 0]$ and probably the item $[2; 1]$.

The recommended practice is to estimate the initialized mixture by an iterative mixture estimation. The initialized mixture must be flattened before, e.g.

```
niter = 10;                            % number of iterations
Mix0  = mixflat(Mix);                  % mixture flattening
Mix   = mixest(Mix0, frg, ndat, niter);% iterative mixture estimation
```

9.1 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.

9.2 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
  'q': by iterative quasi-Bayes mixture estimation
  'b': by iterative batch quasi-Bayes mixture estimation
  'f': by iterative mixture estimation based on forgetting branching
  'm': by iterative mixture estimation with fixed covariances

  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 2 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

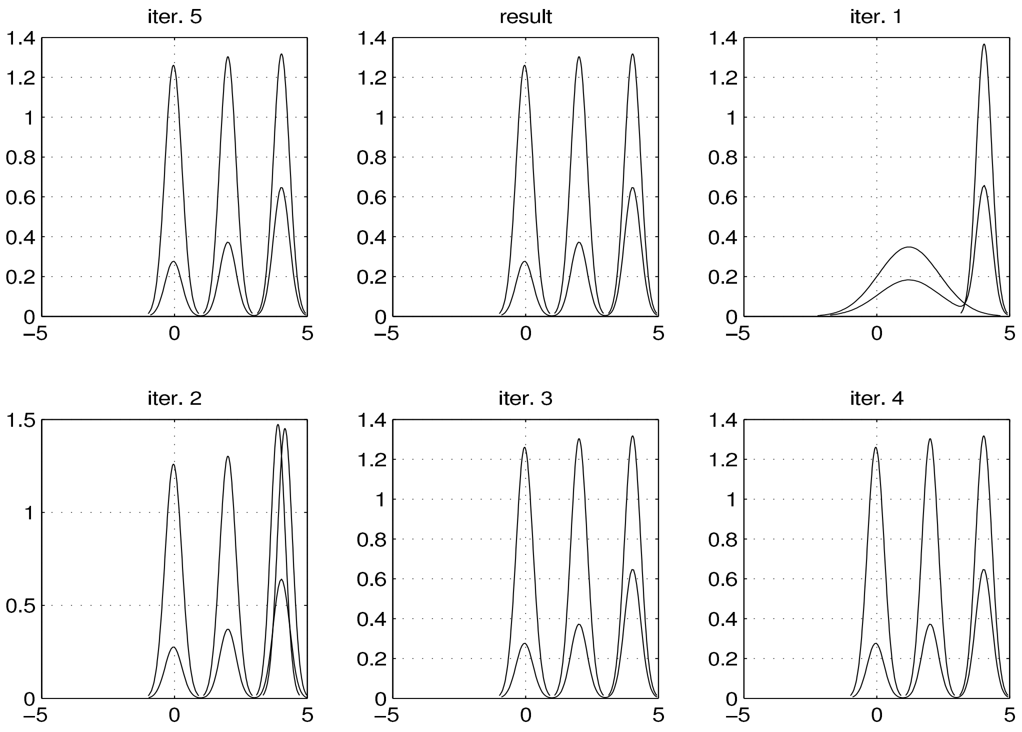

Figure 8: Progress of mixture initialization {zinit}

'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
m	iterative quasi-Bayes with fixed variances
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)

9.3 Processing example

A simple example demonstrates some techniques in mixture initialization. Simulated data sample is generated. The data sample has 1 channel and the simulator has 3 components.

```
ndat  = 1000;                        % length of data
DATA  = zeros (1, ndat);             % pre-allocated data sample
ncom  = 3;                           % number of components
ychns = 1;                           % modelled channel
str   = [0;1];                       % static structure
Com   = comarxls(ychns, str);        % build matrix ARX LS component

Com.cove = 0.1;                      % point estimate of noise variance
Com.Eth  = 0; Coms{1} = Com;         % 1st component
Com.Eth  = 2; Coms{2} = Com;         % 2nd component
Com.Eth  = 4; Coms{3} = Com;         % 3rd component
dfcs     = [0.2 0.3 0.5];            % vector of degrees of freedom of the components
Sim      = mixconst(Coms, dfcs);     % mixture simulator
mixsimul(Sim, ndat);                 % get data sample
... plot of simulator ...
```

The mixture initialization is done with debugging options.

```
PLOTSUB = 230;                       % window where iterations are maintained
PLOTNO  = 2;                         % starting subplot number
DEBUG   = 1;                         % debugging flag
```

The matrices are *global matrices* defined in the function "*prodini*". The PLOTNO is set to 2 because there is already one plot on screen - the simulator in Fig. 8, subplot 1. The PLOTSUB specifies the subplots – here two lines each containing three subplots.
Initial mixture is build, forgetting rate and number of iterations is specified and the initialization is done.

```
Mix0  = genmixe;                     % build initial mixture
frg   = defaults('frg');             % default forgetting rate
niter = 5;                           % number of iterations
Mix   = mixinit(Mix0, frg, ndat, niter); Mix.states.mixll
ans =
 -715.6095
```

In each iteration, the results are plotted in Fig. 8, subplots 2 - 5. The initialization result (the "best" mixture) is displayed in the subplot 6.

In continuation of the example, special processing options are shown in subsections.

9.3.1 Long processing

{longinit}

Initialization may require extremely long processing time. Thus, it is reasonable to exploit the possibility to save initialization results after each iteration and to continue later on with initialization.

This is specified by a global matrix "BREAKPOINT" that specify the filename where the intermediate results are saved:

```
global BREAKPOINT                    % define breakpoint option
BREAKPOINT = 'break';                % name of the file
Mix = mixinit(...);                  % initialization of mixture estimation
```

There are two ways how to continue:

```
mixinit('break');                    % finish an interrupted initialization
```

or select the number of additional iterations, say niter = 10, and

```
mixinit('break', niter);             % make additional "niter" iterations
```

Five iterations of "mixinit" are done for comparison.

```
niter = 5;                           % number of iterations
Mix   = mixinit(Mix0, frg, ndat, niter);
Mix.states.mixll                     % reference v-log-likelihood
ans =
 -715.6095
```

Now, the initialization is done in three iterations using the breakpoint option. and the iterations four and five are done in separately.

```
global BREAKPOINT                      % define breakpoint option
BREAKPOINT = 'break';                  % name of the file
Mix     = mixinit(Mix0, frg, ndat, 3); % 3 iterations
for iter = 4:5
    Mix1  = mixinit('break',1);           % run next 1 iteration
end
Mix1.states.mixll
ans =
 -715.6095
```

9.3.2 Huge data sample processing

Buffered estimation, see paragraph 5.2, should be used when dealing with huge data sample.

The data sample is supposed to be written on disk e.g. by:

```
file  = fopen('filename','wb');        % open a file
fwrite(file, DATA, 'double');          % write DATA
fclose(file);                          % close the file
```

The cell vector

```
Ndat = {'filename',mdat}
```

controls processing instead of the usual matrix "ndat". The 'filename' is the name where data are stored and "mdat" specifies the number of rows of the global matrix DATA used as buffer for buffered estimation. The use looks as follows:

```
file  = fopen('data','wb');            % open a file
fwrite(file, DATA, 'double');          % write DATA
fclose(file);                          % close the file

niter    = 5;                          % maximum number of iterations
filename = 'data';                     % full filename
mdat     = 1;                          % number of sample rows
Ndat     = {filename, mdat};           % this replaces usual "ndat"
DATA     = zeros(mdat,50);             % allocate short buffer

Mix   = mixinit(Mix0, frg, Ndat, niter); % mixture initialization
Mix.states.mixll
ans =
 -715.6095
```

Note that the same data as in previous subsection were processed.

9.4 Test case studies

Several case studies illustrate the mixture initialization. The data samples are simulated, the simulator is displayed as subplot 1 in the plots. The default initial mixture is used. The scripts of the studies is available in Mixtools.

Each study is done for four data samples. Results are displayed in subplots 2–5. The subplot 6 shows values of v-log-likelihood of each study.

9.4.1 Static mixture, 2 dimensions, 4 components

{data4c}

9.4.2 Static mixture, 2 dimensions, 10 components

{data10c}

9.4.3 Static mixture, 4 dimensions, 4 components

{data4d}

9.4.4 Dynamic mixture, 2 dimensions, 2 components

{datadyn}

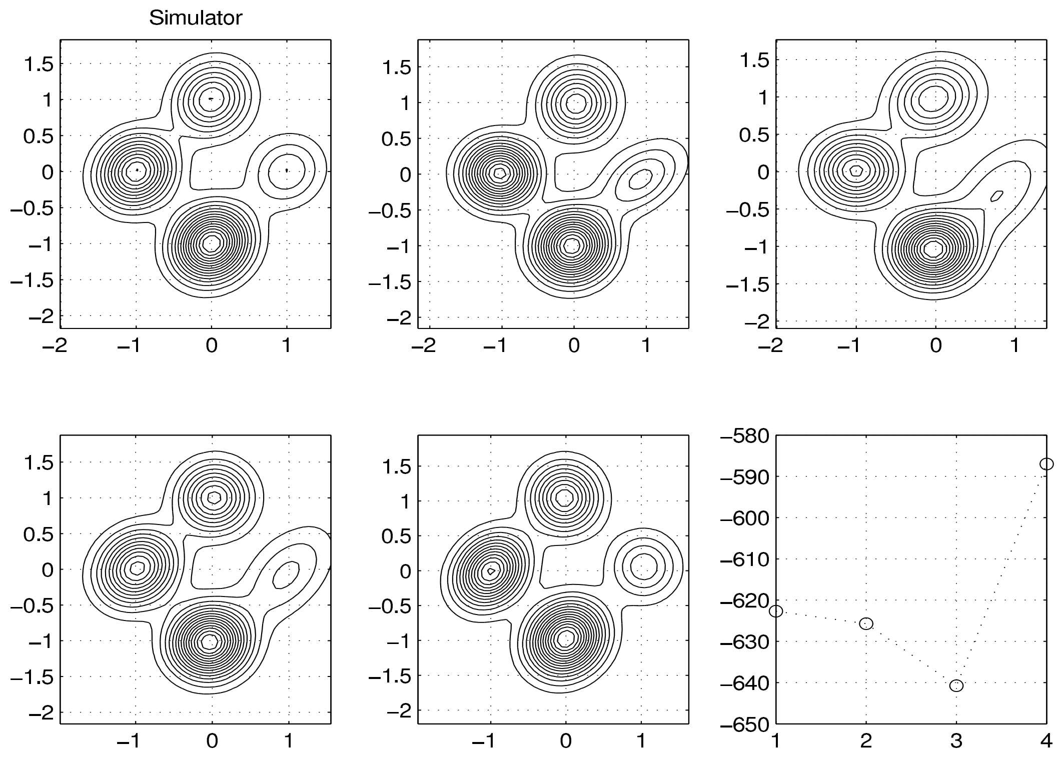

Figure 9: Mixture initialization - 4 static components zini4c {zini4c}

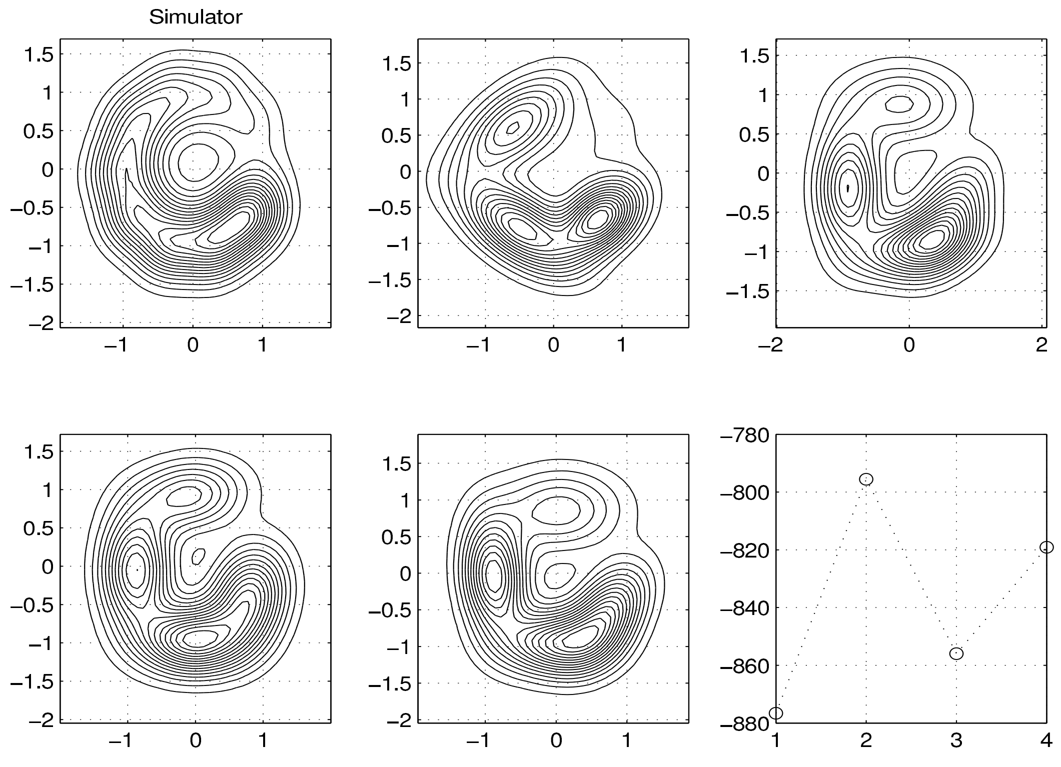

Figure 10: Mixture initialization - 10 static components zini10c {zinit10c}

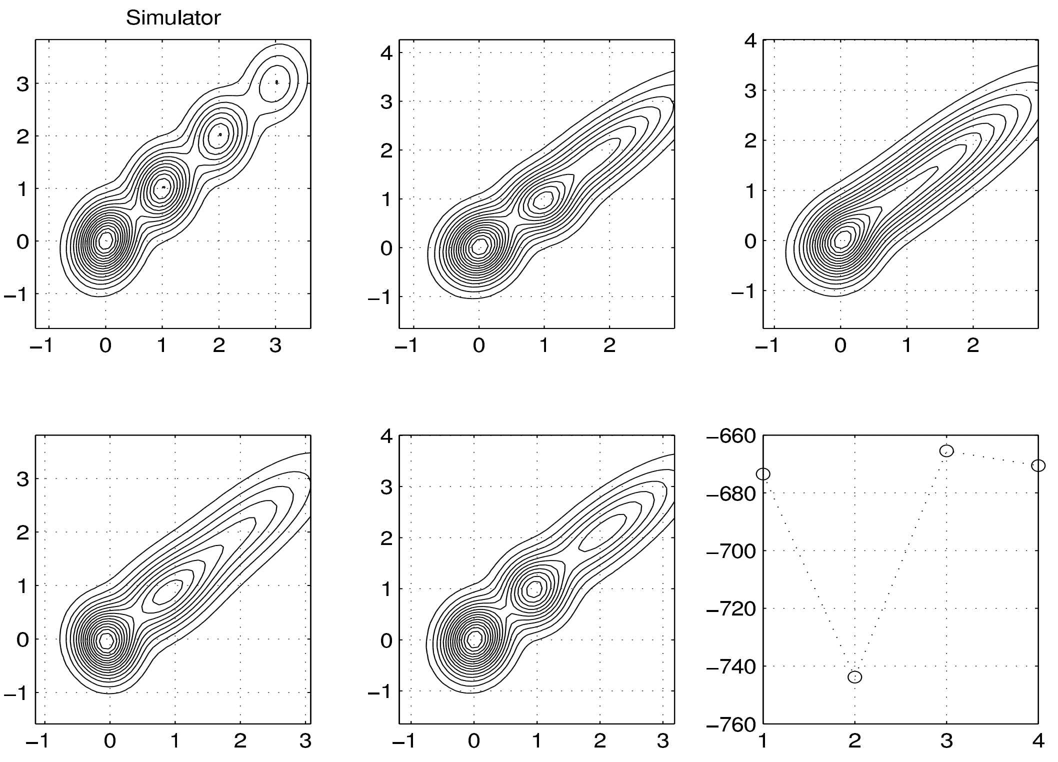

Figure 11: Mixture initialization - 4 dimension, 4 static components zini4d {zini4d}

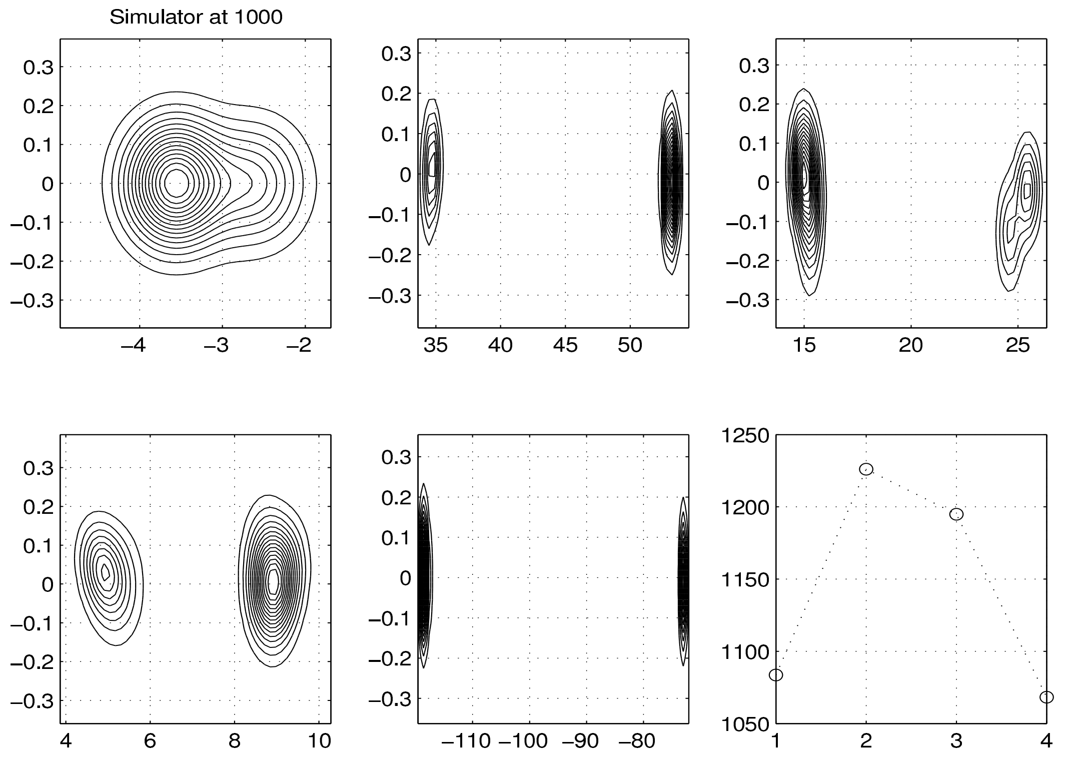

Figure 12: Mixture initialization - dynamic mixture, 2 components zinitdyn {zinidyn}

10 Aproximate parameter estimation of ARX mixtures

{mixestim}

The *approximate parameter estimation* of ARX mixtures (shortly *mixture estimation*) is the topic discussed in this section, refer to [?], Sections ??, ??.

10.1 Implementation notes

In this section, implementation is over-viewed with references to the theory.

estimation methods implemented are:

- quasi-Bayes algorithm, (functions "mixestim"and "mixestqb")
- batch quasi-Bayes algorithm ("mixestbq")
- EM algorithm ("mixestem")
- iterative estimation with fixed covariances ("mixestmt")

recursive data processing is implemented for the quasi-Bayes estimation algorithm ("mixestim").

iterative estimation is available for all algorithms. The prior-posterior branching is the basic technique, [?], Sections ??, ??.

Branching by forgetting algorithm, [?], Sections 10.7 ("mixestbb") is implemented for quasi-Bayes algorithm.

generic estimation function "mixest" calls internally the iterative algorithms defined by an argument "method".

The summary of the estimation functions implemented:

`mixestim`	recursive quasi-Bayes mixture estimation
`mixestqb`	iterative quasi-Bayes mixture estimation
`mixestbq`	batch quasi-Bayes mixture estimation
`mixestbb`	mixture estimation based on forgetting branching, quasi-Bayes
`mixestmt`	mixture estimation by fixed variance algorithm
`mixest`	generic mixture estimation function

Notes:

1. EM algorithm is implemented ("mixestem") but not discussed here due to long processing time;
2. The "stopping rules" are mostly not included in the functions yet and they will be gradually added.
3. The branching by geometric mean, [?], Sections ?? is implemented. Essentially, a new mixture Mix is created as weighted geometric mean of several mixtures Mix1, Mix2,... with probabilistic weights stored in vector "lambdas". It is done by calling the function "mixgmean":
   `Mix=mixgmean(lambdas, Mix1, Mix2,...).`
4. The forgetting with variable forgetting rate is used inside all iterative estimation functions.

10.2 Summary of mixture estimation functions

The basic estimation function is "mixestim":

`Mix = mixestim(Mix0,frg,ndat,Mixa)`	quasi-Bayes mixture estimation with alternative forgetting
`Mix = mixestim(Mix0,frg,ndat)`	quasi-Bayes mixture estimation
`Mix = mixestim(Mix0,frg)`	recursive quasi-Bayes mixture estimation

The iterative estimation functions are:

`Mix = mixestqb(Mix0,frg,ndat,niter)`	quasi-Bayes mixture estimation
`Mix = mixestbq(Mix0,frg,ndat,niter)`	batch quasi-Bayes mixture estimation
`Mix = mixestbb(Mix0,frg,ndat,niter)`	forgetting branching
`Mix = mixestmt(Mix0,frg,ndat,niter)`	estimation with fixed variances
`Mix = mixest(Mix0,frg,niter,options)`	iterative mixture estimation

The auxiliary functions needed in mixture estimation are:

`Mix  = mixstats(Mix, ndat)`	compute estimation statistics
`Mix  = mixstats(Mix)`	compute statistics recursively
`Mix0 = mixflat(Mix)`	mixture flattening

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

10.3 Initial mixtures for estimation

The initial mixtures for estimation (and initialization) are made by a sequence of functions and by setting of various structure fields. The recommendations should be followed.

It is a demanding task to build the initial mixture well. The recommended setting is encoded in the functions ”genmixe” and ”genmixe1”. The functions differ in expected level of output noise, ”genmixe” for high level, ”genmixe1” for low level:

`Mix0 = genmixe(ncom, ychns, str, ndat, ...)`	build initial mixture for estimation

The arguments together with defaults are:

argument	*meaning*	*defaults*
`ncom`	number of components	1
`ychns`	modelled channels	`1 :  size(DATA,1)`
`str`	component structure	[`0;1`], ie. the static one
`ndat`	size of data sample	`size(DATA,2)`

The output mixture is of the matrix ARX LS type. The function have additional arguments that refers to setting the initial components fields. The example of its use is in the sections 8.2.2.

10.4 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, see [?], Section **??**:

`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*

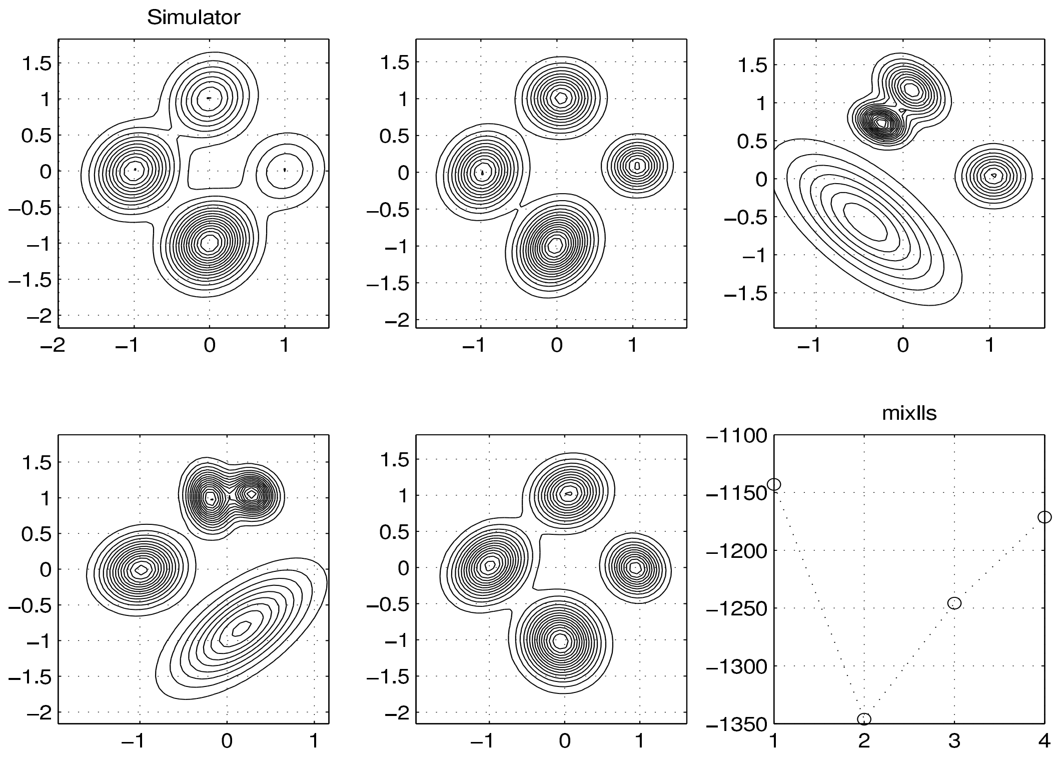

Figure 13: Mixture estimation by "mixestim" zestonly {zeston1}

subplot	contains
1	mixture simulator
2-5	results of individual estimations
6	v-data-likelihood of the estimations

- *summed values in batch data processing.*

The statistics are computed in estimation. They can be computed by the function "mixstats", too.

`Mix = mixstats(Mix)`	*compute statistics iteratively*
`Mix = mixstats(Mix, ndat)`	*compute statistics in batch*

10.5 Quasi-Bayes mixture estimation

{Gquasi-Bayes

The quasi-Bayes estimation is done by the function "mixestim". Both batch and recursive data processing are available.
The function is called as:

`Mix = mixestim(Mix0,frg,ndat,Mixa)`	quasi-Bayes estimation with stabilized forgetting
`Mix = mixestim(Mix0,frg,1,Mixa)`	recursive quasi-Bayes estimation with stabilized forgetting
`Mix = mixestim(Mix0,frg,ndat)`	quasi-Bayes mixture estimation
`Mix = mixestim(Mix0,frg)`	recursive quasi-Bayes mixture estimation

The "Mixa" is the mixture used for stabilized forgetting. It describes guaranteed prior information about mixture that should not be forgotten. For the underlying theory, see [?], Section ??.

Example of mixture estimation by "mixestim" follows. The dependency is discussed of the estimation on different data samples.
Data sample is generated, the simulator is displayed in Fig. 13, subplot 1.

```
ndat = 1000;                        % size of data sample
ncom = 4;                           % number of components
cove = ltdl([0.1 0.01; 0.01 0.1]);  % component noise covariance
Sim  = statsim(ndat, ncom, cove);   % get data sample
... plot simulator ...
```

The initial mixture is build, the forgetting rate is specified as the default one. The prior knowledge employed is that the mixture is static.

```
Mix0 = genmixe(ncom);               % build initial mixture
frg  = defaults('frg');             % default forgetting rate
```

Four estimations are made using the simulator with different random trajectory. The results are plotted in Fig. 13 subplots 2 - 5. The subplot 6 contains plot of v-log-likelihood of individual estimations.

```
for i=1:4
    randn('seed',20*i);             % new random sample
    mixsimul(Sim, ndat);            % get data sample
    Mix  = mixestim(Mix0, frg, ndat); % mixture estimation
    mixlls(i) = Mix.states.mixll;   % record posterior data likelihood
    ... plot of estimated mixture ...
end
... plot of mixlls ...
```

This example shows that for simple cases, the mixture can be estimated even by one pass of mixestim, but the results can differ.

10.6 Batch quasi-Bayes mixture estimation

The batch quasi-Bayes estimation is described in [?], Sections ??. Generally, it is used in the iterative form with a very slow convergence. Its use is necessary when data sample is not homogeneous and the estimation results depend too strongly on the processing ordering.

`Mix = mixestqb(Mix0,frg,ndat,niter)`	batch quasi-Bayes estimation

10.7 Quasi-Bayes mixture estimation based on forgetting branching

{brafor}

The forgetting branching is described in [?], Sections 10.7.

The algorithm starts from a mixture at a time t. The estimation branches. Two estimations run in parallel: with the default forgetting rate and with a very low *forgetting rate* (recommended value is 0.6). Increments of posterior data likelihood are measured for both mixtures estimated. When the difference of log-likelihood values exceeds a level, the poorer mixture is abandoned. The better mixture is used as the starting one for new branching.

Counts of "success" are measured for both forgetting rates. When the counts of success sufficiently differs, the rest of data sample is processed with the default forgetting rate without the branching.

The algorithm prevents premature convergence of the estimation statistics to wrong values.

This processing logic applies in the first iteration only. The rest of iterations is processed by quasi-Bayes estimation.

The selection of alternative forgetting rate does not seem to be critical and the recommended value of 0.6 is acceptable.

The function is called as:

`Mix = mixestbb(Mix0,frg,ndat,niter)`	batch quasi-Bayes estimation

10.8 Mixture estimation with fixed covariances

The algorithm fixes covariances of modelled channels. It is used in initialization of mixture estimation "mixinit". The processing with fixed variances is used in the first iteration and the rest of iterations is done by quasi-Bayes estimation.

The algorithm prevents convergence of the estimation statistics to wrong values in the most important first iteration.
The function is called as:

`Mix = mixestbb(Mix0,frg,ndat,niter)`	estimation with fixed variances

10.9 Iterative estimation and mixture flattening

A single processing of the data sample rarely provides estimation good results. The iterative algorithms estimate mixture on it and take the result as a better guess than the prior one. Its distribution is flattened in order to avoid over-confidence to bad results. Estimation and flattening form a step in the estimation cycle that repeats "niter" times.

This algorithm is in [?], Sections ??, ?? called *prior-posterior branching*. We use the term "*iterative estimation*" in this Guide.
The set of functions solves the iterative mixture estimation:

`Mix = mixestqb(Mix0,frg,ndat,niter)`	quasi-Bayes mixture estimation
`Mix = mixestbq(Mix0,frg,ndat,niter)`	batch quasi-Bayes mixture estimation
`Mix = mixestbb(Mix0,frg,ndat,niter)`	forgetting branching
`Mix = mixestmt(Mix0,frg,ndat,niter)`	estimation with fixed variances
`Mix = mixest(Mix0,frg,niter,options)`	iterative mixture estimation

The function "mixest" calls individual iterative funtions according to the argument "options":

`options` is a character string, the 1st character is decisive. The following options are available:

'q': iterative quasi-Bayes estimation (default);

'b': iterative batch quasi-Bayes estimation

'f': iterative estimation based on quasi-Bayes algorithm

'm': iterative estimation with fixed variances and forgetting branching.

`niter` has the default value of 10 for each algorithm. The higher values are better but the selection depends on amount of computing time available.

Intuitively, a more significant flattening should be made at the beginning of the iterative mixture estimation than at its end. Analysis [?], Section ?? shows necessary properties of flattening that do not prevent the optimal estimation. It serves for the design of flattening algorithm with linearly growing confidence into likelihood. It is implemented as the function "*mixflatv*".

Examples of iterative mixture estimation follow. They are intended as illustrative ones, no conclusions are derived.

The first example is continuation of the example from the subsection 10.5. The same processing with the same data samples is the done but, the iterative quasi-Bayes estimation with 20 iteration replaces the initial processing. The results are displayed in Fig. 14.
The comparison with the example in the subsection 10.5 shows that the first estimation is decisive for estimation success.
Data sample is generated, the simulator is displayed in Fig. 15, subplot 1.

```
ndat = 1000;                           % sample size
ncom = 5;                              % number of components
cove = ltdl([0.1 0.01; 0.01 0.1]);     % common noise variance
Mix  = statsim(ndat, ncom, cove);      % create static mixture
... plot simulator ...
```

Initial mixture is build, forgetting rate and number of iterations is selected. The prior knowledge used here is that the mixture is a static one with 3 components.

```
Mix0    = genmixe(3);                  % initial mixture
frg     = defaults('frg');             % forgetting rate
niter   = 10;                          % number of iterations
```

Individual estimations are done, the results are displayed in Fig. 15, subplots 2-5.

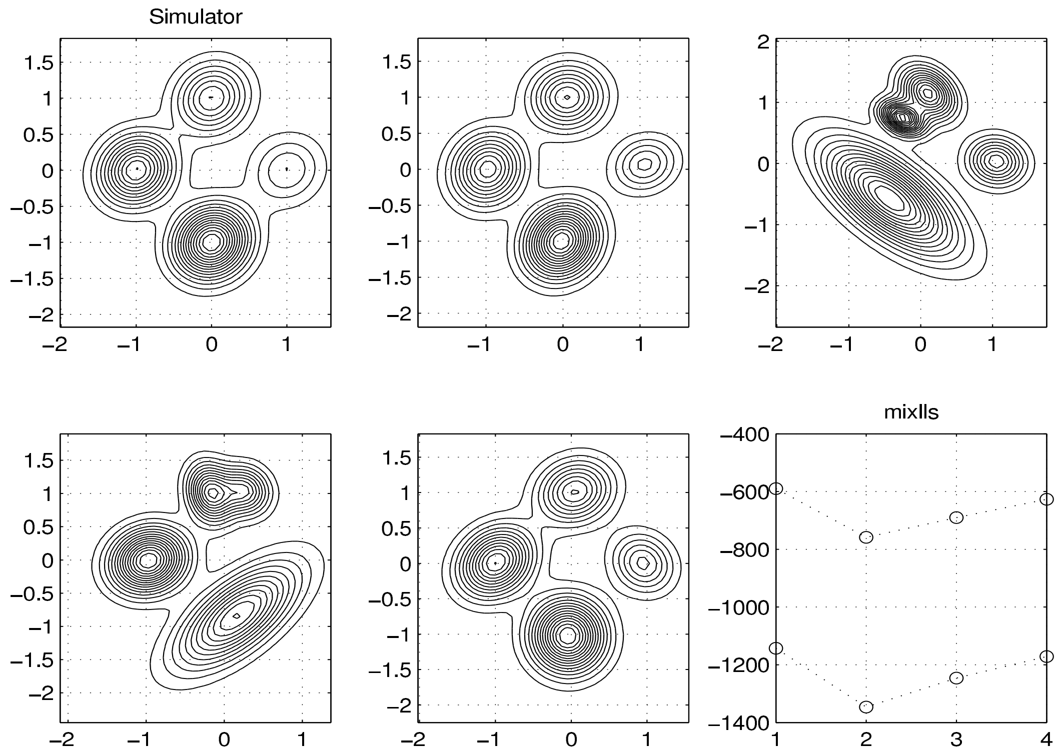

Figure 14: Mixture estimation by iterative quasi-Bayes algorithm {zeston2}

subplot	contains
1	mixture simulator
2-5	results of individual estimations
6	v-data-likelihood of the estimations (comparison)

zestonly.m

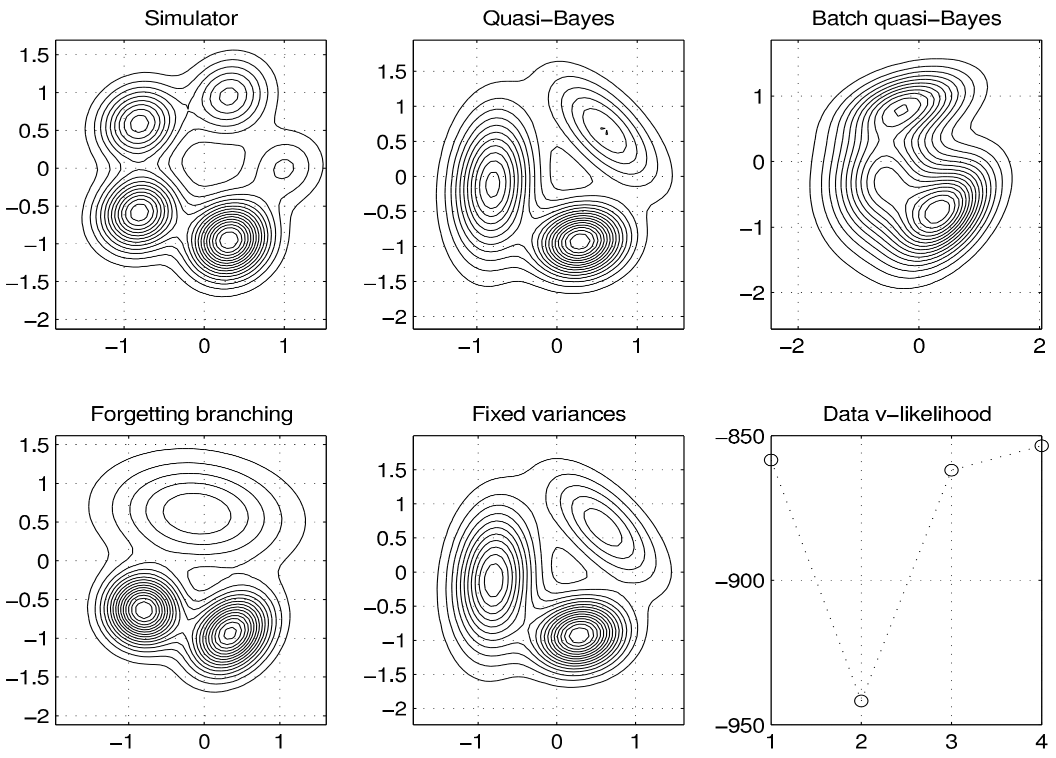

Figure 15: Mixture prediction with iterative estimation zestiter.m {zestiter}

subplot	contains
1	mixture simulator
2	iterative estimation by quasi-Bayes algorithm
3	iterative estimation by batch quasi-Bayes algorithm
4	iterative estimation by forgetting branching
5	iterative estimation by fixed variances
6	comparison of data v-likelihood

Iterative quasi-Bayes estimation

```
method = 'q';                                % quasi-Bayes estimation
Mix = mixest(Mix0, frg, ndat, niter, method); % estimation
mixlls(1) = Mix.states.mixll;
... plot result of quasi-Bayes estimation...
```

Iterative batch quasi-Bayes estimation

```
method = 'b';                                % batch quasi-Bayes
Mix = mixest(Mix0, frg, ndat, niter, method); % estimation
Mix = mixstats(Mix, ndat);                   % compute posterior data likelihood
mixlls(2) = Mix.states.mixll;                % posterior data likelihood
... plot result of batch quasi-Bayes algorithm ...
```

Iterative estimation by forgetting branching

```
method = 'f';                                % forgetting branching
frg    = 0.6;                                % high forgetting
Mix = mixest(Mix0, frg, ndat, niter, method); % estimation
mixlls(3) = Mix.states.mixll;                % posterior data likelihood
... plot result of estimation by forgetting branching ...
```

Iterative estimation with fixed variances

Only one step of the estimation is done. The rest of iterations is done with quasi-Bayes estimation. The 1st steps is used for better initial condition for estimation. In this way, this estimation method is used in "mixinit".

```
method = 'm';                                % fixed variances
frg    = defaults('frg');                    % default forgetting
Mix    = mixest(Mix0, frg, ndat, 1, method); % estimation
Mix0   = mixflat(Mix);
method = 'q';
Mix  = mixest(Mix0, frg, ndat, niter-1, method); % estimation
Mix  = mixstats(Mix, ndat);
mixlls(4) = Mix.states.mixll;                % posterior data likelihood
... plot result of estimation by fixed variances ...
```

The resulting v-data-likelihood is displayed in Fig. 15, subplot 6.

10.10 Processing of huge data samples

Recall that all estimation functions realized as MEXes have the possibility of *buffered estimation*, see Section 5.2. Its use is illustrated on estimation with iterative quasi-Bayes estimation.

The data sample is written to disc e.g. by:

```
file  = fopen('data','wb');              % open a file
fwrite(file, DATA, 'double');            % write DATA
fclose(file);                            % close the file
```

Among usual parameters of the function "mixestim", the argument "ndat" is changed to Ndat={'filename', number of rows in the global buffer matrix DATA}:
Example follows. The data from previous case study will be processed. The ordinary processing gives:

```
Mix = mixest(Mix0, frg, ndat, niter, method); % estimation
Mix.states.mixll
ans =
 -857.3862
```

The buffered processing follows.

```
file  = fopen('data','wb');            % open a file
fwrite(file, DATA, 'double');          % write DATA
fclose(file);                          % close the file

filename = 'data';                     % full filename
mdat     = 2;                          % number of sample rows
Ndat     = {filename, mdat};           % this replaces usual "ndat"
DATA     = zeros(mdat,50);             % allocate short buffer
size(DATA)
ans =
     2    50

Mix = mixest(Mix0, frg, Ndat, niter, method); % buffered estimation
Mix.states.mixll
ans =
 -857.3862
```

mixest.tex zestonly.m zestiter.m zestbuf.m by PN June 25, 2004

11 Mixture prediction

{prediction}

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

- *mixture projection*
  means marginalization and conditioning, see [**?**]. 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*.

11.1 Mixture projection

The projection converts mixture estimator into mixture *projector*. It provides description of Student pdf (13) mostly approximated by normal pdf (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 predictor can be re-built for a new selection of these channels.

The mixture projection is done by the function "mix2pro":

`pMix = mix2pro(Mix, pchns, cchns)`	*build mixture projector*

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

11.2 Prediction with mixture projection

The mixture prediction with mixture projector is discussed.

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 taken form 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 an additional argument "pre" that contains record about the data scaling, see section "Data preprocessing". The zero-delayed regression vector is to be introduced in the original data scaling.

The mixture prediction is done by:

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

The input arguments with defaults are:

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 output "lhs" arguments are specified as
pMix or [Eths, coves, alphas] or [pMix, weights] or [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

The prediction can be done ahead for a number of processing steps [1]

```
lhs = profixn(Mix, psi0, pre, nsteps)        prediction nsteps ahead
```

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

11.3 Prediction with mixture and related functions

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

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

The meaning of input and output arguments is the same as in previous sections. The same input arguments are used by visualization functions (mixplot, mixgrid etc.). Multistep version does not exist.

11.4 Prediction examples

11.4.1 Tutorial example

The mixture projection is documented on a simulated example. A static ARX LS mixture is built and displayed in Fig. 16, subplot 1.

```
cove = ltdl([0.1 0.01; 0.01 0.1]);       % common component noise covariance
ncom = 4;                                % number of components
Mix  = statsim(0, ncom, cove);           % build ARX LS mixture
Mix.dfcs                                 % vector of degrees of freedom of components
ans =
    0.1000    0.2000    0.3000    0.4000
...  plot ...
```

[1]this function will be covered by "profix" when MEX version will be available

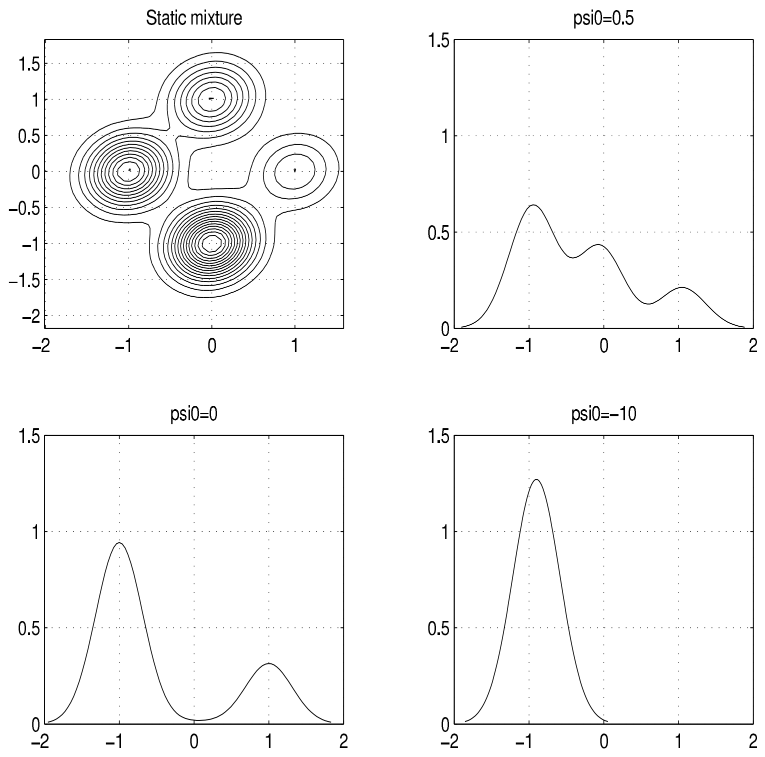

Figure 16: Mixture prediction {zprotut}

subplot	contains
1	mixture simulator
2	prediction: pchns=1, cchns=2, psi0=0.5
3	prediction: pchns=1, cchns=2, psi0=0
4	prediction: pchns=1, cchns=2, psi0=-10

zprotut.m

Then, the mixture projector is build:

```
pchns = 1;                              % predicted channel
cchns = 2;                              % channel in condition
pMix  = mix2pro(Mix, pchns, cchns)      % build mixture projector
pMix =
      Facs: {1x8 cell}                    -> array of factors
      coms: [4x2 double]                  -> description of components
      dfcs: [0.1000 0.2000 0.3000 0.4000] -> degrees of freedom of components
      type: 122                           -> p-mixture type: projector
    states: [1x1 struct]                  -> states, see below
```

The result is p-ARX LS mixture. The mixture states are

```
pMix.states
ans =
       modelled: [1 2]                  -> list of modelled channels
    notmodelled: []                     -> not-modelled channels, none here
      predicted: 1                      -> predicted channels
    incondition: 2                      -> channels in condition
    zerodelayed: []                     -> zero-delayed not modelled channels
         comaux: {{1x7 cell}  {1x7 cell}  {1x7 cell}  {1x7 cell}}
```

The first 2 states refer to the channels of the mixture estimator. The "comaux" is an auxiliary field that enables fast prediction.
The mixture prediction is built and displayed in Fig. 16 subplot 2. The prediction is static p-matrix ARX LS mixture. The degrees of freedom o components "dfcs" are transformed due to conditioning. The resulting p-mixture"pMix1" cannot be re-built or other zero-delayed regression values substituted in it.

```
psi0  = 0.5;                            % zero-delayed regression vector
pMix1 = profix(pMix,psi0)               % psi0 substitution
pMix1 =                               -> prediction
        Coms: {[1x1 struct]  [1x1 struct]  [1x1 struct]  [1x1 struct]}
        dfcs: [0.1667 0.3333 0.5000 3.0266e-005] -> degrees of freedom of components
    reserved: []
        type: 124                           -> p-mixture type: prediction
      states: [1x1 struct]                  -> auxiliary states
```

The substitution of the regression vector into projector can be repeated and it is displayed on Fig. 16 subplot 3.

```
psi0  = 0;                              % zero-delayed regression vector
pMix1 = profix(pMix,psi0);              % regression substitution
```

The mixture projector is re-built into another projector and displayed on Fig. 16 subplot 4. The outputs from "mixpro" can be represented by the cell vector (vector in one-dimensional case) of means, covariance and transformed degrees of freedom of components. Displaying of these values is mostly of interest:

```
psi0 = -10;                             % new prediction arguments
[Eths, coves, alphas] = mixpro(pMix, pchns, cchns, psi0); % prediction
alphas                                  % converted Mix.dfcs
alphas =
    0.0000    0.0000    1.0000    0.0000
```

11.4.2 Prediction with scaled data

In mixture estimation, use of scaled data is recommended. If used, the mixture prediction must be re-scaled to original data scaling. The zero-delayed regression vector is introduced in the original data scale.

In the example, the simulated data sample is generated and displayed in Fig. 17, subplot 1.

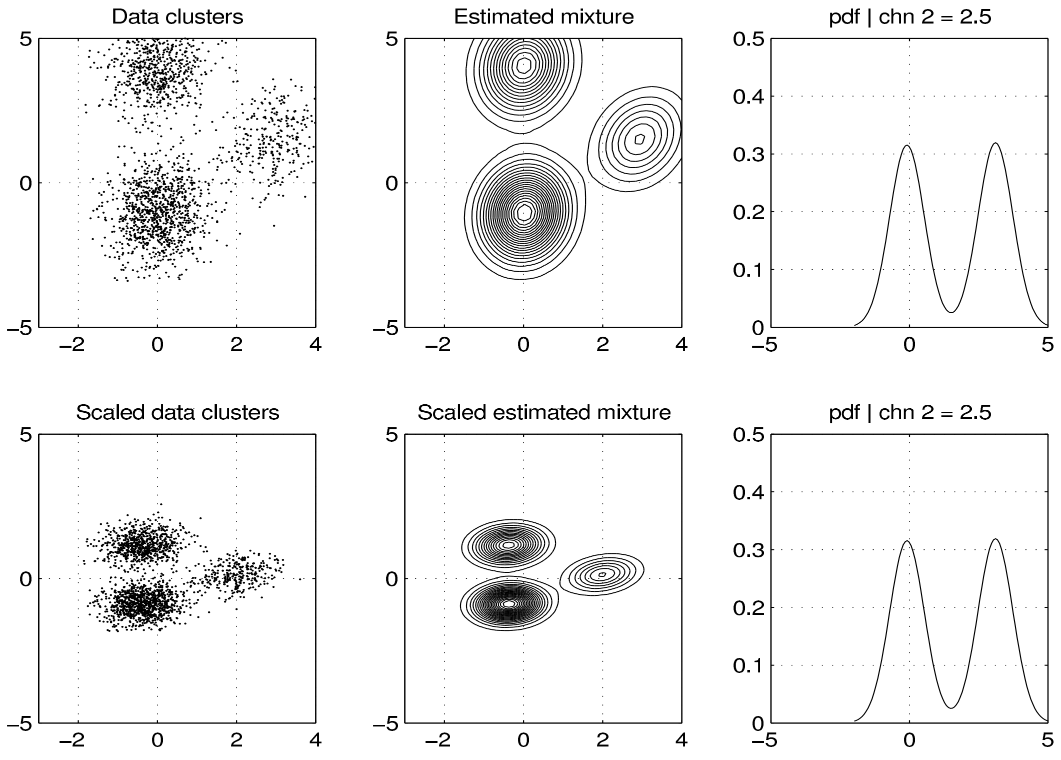

Figure 17: Mixture prediction with scaled data {zproscal}

subplot	*contains*
1, 4	data clusters
2, 5	estimated mixture
3, 6	mixture prediction, psi0=2.5

zproscal.m

```
ndat = 2000;                          % length of data
ncom = 3;                             % number of components
cove = ltdl([0.1 0.01; 0.01 0.1]);    % noise covariance
Mix  = statsim(ndat, ncom, cove);     % get data sample
preproc({'scale', [0.5 0.5; 2 3]});   % transform data
... plot data clusters ...
```

The data transformation means to add 0.5 to both channels and multiply the channel 1 by 2 and the channel 2 by 3.

An initial mixture is build and mixture is estimated. The estimated mixture is diplayed in Fig. 18, subplot 2.

```
Mix0  = genmixe(ncom);                % initial mixture for estimation
frg   = defaults('frg');              % forgetting rate
niter = 10;                           % number of iterations
Mix   = mixest(Mix0,frg,ndat,niter);  % estimate mixture
... mixture plot ...
```

The predictor is build and the zero-delayed regression vector specified, the prediction computed and the prediction is displayed in Fig. 17, subplot 3.

```
pchns = 1;                            % predicted channel
cchns = 2;                            % channel in condition
pMix  = mix2pro(Mix,pchns,cchns);     % mixture projector

psi0  = 2.5;                          % zero-delayed regression vector
pM    = profix(pMix, psi0);           % mixture prediction
... plot of predictor ...
```

The data are scaled and data displayed in Fig. 17, subplot 4.

```
pre = preproc({'scale',[]});          % scale data
... plot of data ...
```

The mixture "Mix0" is estimated and displayed in Fig. 17, subplot 5.

```
Mix   = mixest(Mix0, 1, ndat, niter);  % estimate mixture
... plot of mixture ...
```

Mixture prediction is done. The value of regression vector is specified in the initial data scaling. The prediction is displayed in Fig. 17, subplot 6.

```
pMix = mix2pro(Mix, pchns, cchns);    % mixture projection
psi0 = 2.5;                           % not scaled psi0
pM   = profix(pMix, psi0, pre);       % mixture prediction
... plot of predictor
```

The result is very close to the original prediction, a difference is caused by using diferent estimators. Note that the zero-delayed regression vector is introduced in the original data scaling.

11.4.3 Data-dependent approximate prediction

In the case of prediction with static mixtures without conditioning, a modification of the general algorithm is available that makes the predicted *component weights* data dependent, [?]. They are generated by "profix" as the 2nd or 4th output argument.

Nothing can be gained in prediction if the data sample follows exactly the mixture model, namely if the current component is selected randomly each processing time. This occurs with simulated data but never in practice.

In the example, the data sample is generated by a model with Markov jumps among components. The "CUMTAB" is global matrix describing the transitions among components. The mixture simulation is plotted in Fig. 18, subplot 1.

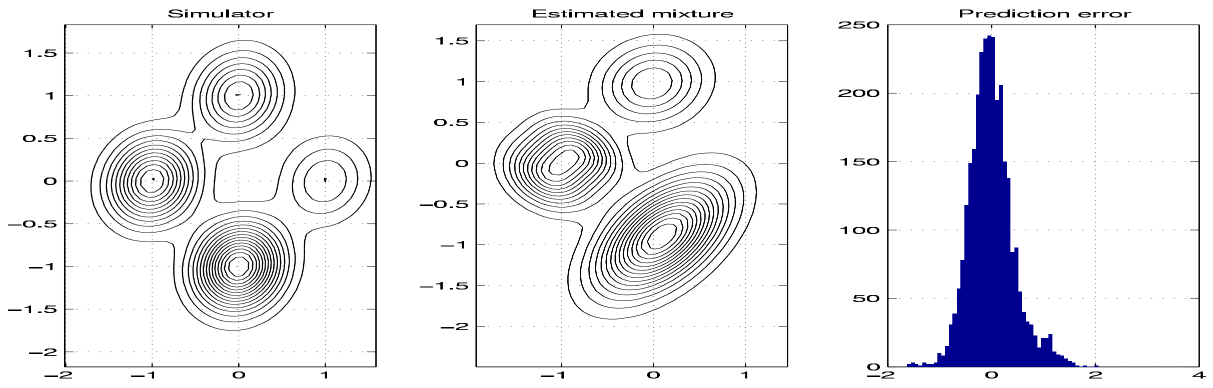

Figure 18: Mixture prediction by data dependent estimator {zprodep}

subplot	contains
1	mixture simulator
2	estimated mixture
3	histogram of prediction error

zprodep.m

```
ndat = 3000;                          % length of data
ncom = 4;                             % number of components
cove = ltdl([0.1 0.01; 0.01 0.1]);    % noise covariance
Sim  = statsim(0, ncom, cove);        % build simulator
global CUMTAB ACTIVE
CUMTAB = [0.97 0.01 0.01 0.01          % Markov transition table
          0.01 0.97 0.01 0.01
          0.01 0.01 0.97 0.01
          0.01 0.01 0.01 0.97 ];
DATA   = zeros(2, ndat);              % pre-allocate data
mixsimul(Sim, ndat);                  % get data sample
... plot simulator ...
```

Mixture estimation without initialization follows, the estimated mixture is plotted in Fig. 18, subplot 2.

```
Mix0 = genmixe(ncom);                 % initial mixture for estimation
niter    = 10;                        % maximum number of iterations
frg      = 1;                         % no forgetting
method   = 'q';                       % method of estimation
Mix = mixest(Mix0, frg, ndat, niter, method); % iterative estimation
... plot estimated mixture ...
```

Marginal distribution of 1st channel (mixture projector) is obtained.

```
pchns = 1;                            % predicted channel
pMix  = mix2pro(Mix,pchns);           % mixture projector
```

Trajectory of prediction is computed.

```
for TIME = 1:ndat                     % getting prediction trajectory
   [Eths,coves,alphas,weights] = profix(pMix); % prediction
   yp(TIME) = Eths * weights';        % data dependent prediction
   y0(TIME) = Eths * alphas';         % constant prediction
end
```

Results are evaluated. The standard deviation of prediction error is much lower than the standard deviation of data, the difference shows amount of data variance explained by the mixture model. The gain is reached due to data dependent weights used in prediction. Histogram of prediction error is displayed in the Fig. 18,

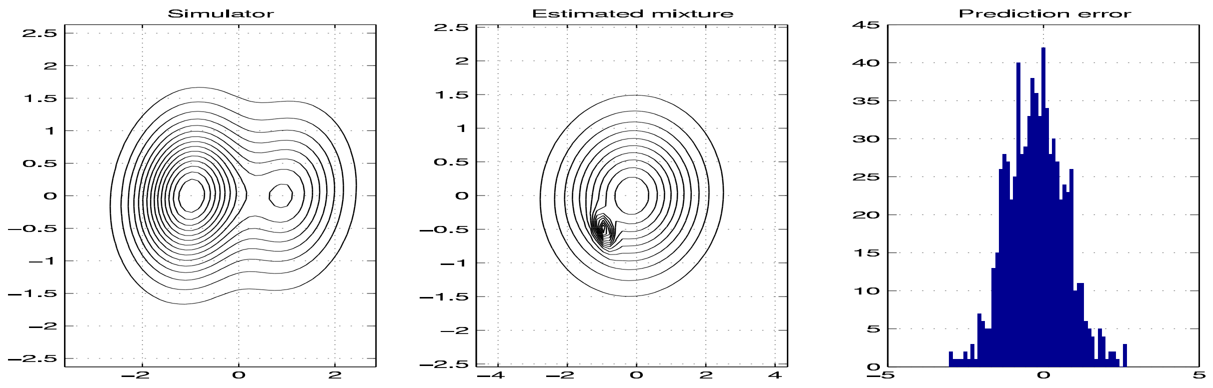

Figure 19: Prediction of groups of data {zpropair}

subplot	contains
1	mixture simulator
2	estimated mixture
3	histogram of prediction error

/

subplot 3. The "normal prediction y0" is really constant.

```
dd  = DATA(pchns,:);                    % data row
ee  = dd-yp;                            % prediction error
s2d = std(dd')                          % standard deviation of data
s2d =
0.6144
s2e = std(ee')
s2e =
0.4316

i   = find(y0~=y0(1))                   % constant prediction
i =
     [ ]
... histogram of prediction error ...
```

If the same work is done without the Markov jumps between components, the results are:

```
s2d =
    0.6822
s2e = std(ee')                          % standard deviation of prediction error
s2e =
    0.8682
```

11.4.4 Prediction with grouped data {pairs}

The algorithm explained here is discussed in [?], Section **??**. It enables to make approximate prediction "naturarly" data dependent even in the case of static mixtures. The algorithm is referred to as *prediction of groups of data*.

Mixture modeling of grouped data forms essence of the trick we are talking about. For simplicity, pair-wise grouping is discussed here. The estimated mixture is then

$$f(d_t, d_{t-1}|d(t-2)) = \sum_{c \in c^*} \alpha_c f(d_t, d_{t-1}|d(t-2), c) = \sum_{c \in c^*} \alpha_c f(d_{t-1}|d(t-2), c) f(d_t|d(t-1), c)$$

and corresponding prediction becomes

$$f(d_t|d(t-1)) \propto \sum_{c \in c^*} \underbrace{\alpha_c f(d_{t-1}|d(t-2),c)}_{\boxed{\text{data-dependent component weight}}} \underbrace{f(d_t|d(t-1),c)}_{\boxed{\text{usual component}}} .$$

The joint distribution $f(d_t, d_{t-1}|d(t-2))$ is simply estimated by the standard tools operating on the reorganized data sample: each even data sample is moved to an added channel. This is solved by data preprocessing:

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

Note that nsk>2 can be used.

The prediction based on this method is demonstrated by simple case of simulated static mixture whose components changed as a Markov chain, see Section 17.

The mixture simulator is built:

```
ndat = 1500;                            % length of data
ncom = 2;                               % number of components
cove = ltdl([0.5 0.05; 0.05 0.5]);      % point estimate of noise covariance
Sim  = statsim(0, ncom, cove);          % build simulator
```

The of Markov transition probabilities defining jumps amomng components is introduced:

```
global CUMTAB
CUMTAB = [0.97 0.03; 0.03 0.97];        % Markov transition probabilities
```

The data sample is pre-allocated and filled by simulated data. The simulator is displayed in Fig. 19, subplot 1.

```
DATA   = zeros(2, 2*ndat);              % pre-allocate data
mixsimul(Sim, 2*ndat);                  % get simulated data sample
...  plot simulator ...
```

The data are split into data for learning and for testing of prediction quality:

```
Data = DATA(:,ndat+1:2*ndat);           % data for prediction
DATA = DATA(:,1:ndat);                  % data for learning
```

The data are grouped:

```
nsk  = 2;                               % extent of data grouping
preproc({'group', nsk});                % preprocess data
[nchn, ndat] = size(DATA)               % dimensions
nchn =
     4
ndat =
   750
```

Initial mixture estimator is built. The mixture is estimated and the result displayed in Fig. 19, subplot 2.

```
Mix0   = genmixe(2);                    % initial mixture for estimation
niter  = 10;                            % maximum number of iterations
frg    = 1;                             % no forgetting
method = 'q';                           % iterative estimation method
Mix = mixest(Mix0, frg, ndat, niter, method); % iterative estimation
... plot mixture ...
```

Data for prediction are reorganized and the prediction trajectory is made.

```
DATA = Data;
preproc({'group', nsk});                % reorganize data
pchns = 1;                              % predicted channel
cchns = [3 4];                          % channels in condition
pMix  = mix2pro(Mix,pchns,cchns);       % predictor

for TIME = 1:ndat                       % getting prediction trajectory
   [Eths, coves, alphas] = profix(pMix);   % prediction
   yp(TIME) = Eths * alphas';
pMix  = mix2pro(Mix,pchns,cchns);       % predictor
end
```

The results are:

```
  dd  = DATA(pchns,:);   ee  = dd-yp;
  s2d = std(dd')
s2d =
    1.0775
  s2e = std(ee')
s2e =
    0.9102
...  plot histogram of prediction error ...
```

The histogram of prediction error is in Fig 19, subplot 3. Because the mixture is a static one, the prediction (marginal on the channel 1) is constant. The approximate prediction makes the prediction data dependent.

When used, the standard deviation of the prediction error "s2e" is less than std of data.

zpropair.m

11.4.5 Prediction n-steps aread

{nstepahead}

The function "profixn" makes the prediction n-steps ahead. Example follows.

Data are generated with Markov jumps of components and displayed in Fig. 20, subplot 1.

```
ndat = 2000;                            % length of data
ncom = 4;                               % number of components
cove = ltdl([0.1 0.01; 0.01 0.1]);      % noise covariance
Sim  = statsim(0, ncom, cove);          % build simulator
global CUMTAB ACTIVE
CUMTAB = [0.97 0.01 0.01 0.01            % Markov transition probabilites
          0.01 0.97 0.01 0.01
          0.01 0.01 0.97 0.01
          0.01 0.01 0.01 0.97 ];
DATA   = zeros(2, ndat);                % pre-allocate data
mixsimul(Sim, ndat);                    % get data sample
... plot of simulator ...
```

An initial mixture is build, estimated and displayed in Fig. 20, subplot 2.

```
Mix0 = genmixe(ncom);                   % initial mixture for estimation
niter    = 10;                          % maximum number of iterations
frg      = 1;                           % no forgetting
method   = 'q';                         % method of estimation
Mix = mixest(Mix0, frg, ndat, niter, method); % iterative estimation
... plot  of estimated mixture ...
```

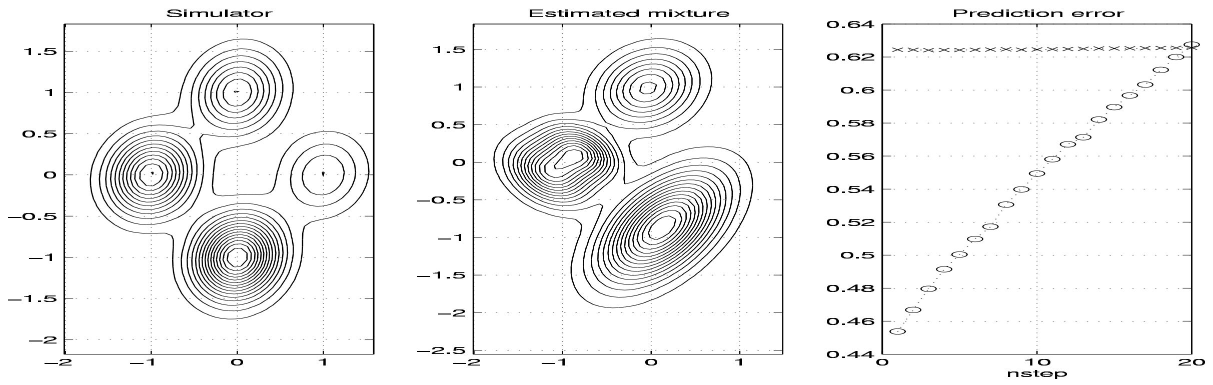

Figure 20: Prediction for several steps ahead {zprofixn}

subplot	contains
1	mixture simulator
2	estimated mixture
3	standard deviation of prediction error by number of steps ahead (marked by '*') corresponding standard defiation of data (marked by 'x');

Computation of the prediction error by number of steps ahead is done and results are displayed in Fig. 20. Note the dependency of the prediction error on the number of steps.

```
pchns = 1;                              % predicted channel
pMix  = mix2pro(Mix,pchns);             % mixture projector

for nstep = 1:20
   maxtime = ndat - nstep;                 % upper bound for time
   for TIME = 1:maxtime                    % getting prediction trajectory
       [Eths, coves, alphas, weights] = profixn(pMix, [], [], nstep);
       yp(TIME) = Eths * weights';
   end
   dd  = DATA(pchns, nstep+1:length(DATA));
   ee  = dd-yp;
   s2d(nstep) = std(dd');
   s2e(nstep) = std(ee');
   yp = [];
end
... plot of s2e, s2d ...
```

zprofixn.m

11.5 Reduction of data space

The marginalization by "mix2pro" preserves in the p-mixture all factors for original channels so that the re-building operation can be done.

The function "mix2mixm" 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.

An example follows. Let us build an matrix ARX LS mixture estimator:

```
ychns = 1:7;                            % modelled channels
str   = [1 2 3 4 5 6 7   8  0; 1 1 1 1 1 1 1  0   1]; % component structure
Com   = comarxls(ychns, str);           % build matrix ARX LS component
Mix   = mixconst({Com, Com}, [11 22]);  % build matrix ARX LS mixture with dfcs=[11 22]
```

The predicted (marginal) channels are:

```
pchns = [5,6];                          % marginal channels
```

The marginalization is done:

```
pMix  = mix2mixm(Mix, pchns);           % marginalization in data space
pMix.states
ans =
       modelled: [5 6]                  -> list of modelled channels
    notmodelled: [1 2 3 4 7 8]          -> list of not-modelled channels
      predicted: [5 6]                  -> predicted channels
    incondition: []                     -> channels in condition, here no
    zerodelayed: 6                      -> zero-delayed not modelled channels
         comaux: {{1x7 cell}  {1x7 cell}}
```

Here, all modelled channels are predicted.

mixpro.tex zprotut.m zpropair.m zprodep.m zproscal.m zprofixn.m zpromarg.m by PN June 25, 2004

12 Visualization

{vizual}

By *mixture visualization* we mean 1D, 2D and 3D plots of mixture projections and(or) its components. This is done with the functions:

`mixplot`	mixture plot
`mixplotc`	contour mixture plot and components
`mixmesh`	mesh plot of mixture

Coordinates for plotting are generated by:

`mixgrid`	coordinates for plot
`datagrid`	get coordinates of data

Scanning mixture predictions and signal:

`mixscan`	scanning mixture
`datascan`	scan data
`sigscan`	scan signal

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 11). Other arguments refer to coordinates, grid densities and ranges. The arguments are listed:

`Mix`	mixture, any mixture form
`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

Important: 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.

The full synopses of the functions are summarized.

```
mixplot (Mix,pchns,cchns,psi0, n, r)
mixplotc(Mix,pchns,cchns,psi0, n, r)
mixmesh (Mix,pchns,cchns,psi0, n, r)
[x,y,z] = mixgrid(Mix,pchns,cchns,psi0,n,r)
[x,y,z] = datagrid(chns, nx, ny, rx, ry)
```

The functions for scanning are:

```
mixscan(Mix,chns,pre)
datascan(chns)
sigscan(chns)
```

The "chns" argument is a list of channels in the function "sigscan" or a two rows of pairs of channels in the "mixscan" and "datascan".
Auxiliary functions for plotting static mixtures with two channels and its components (or plot coordinates) are summarized. These functions do not make any prediction.

`statmesh(Mix)`	interactive mesh plot
`statplot(Mix, 'options')`	plot of mixture components
`[x,y,z] = statgrid(Mix, n, r)`	coordinates for plot
`[x,y,z] = ...` `statgrid(Eths, coves, alphas, n, r)`	alternative generating of coordinates

Note: the "options" argument is sent to "plot" function. The arguments "Eth", "coves" and "alphas" are the mixture characteristics (means, covariances and weights of its components) that can be generated by "mixpro".
Two interactive functions are provided. The user specifies the arguments above in an interactive window:

`mixshow(Mix)`	interactive mesh plot
`mixbrow(Mix)`	interactive mesh plot

The use of visualization functions is documented on examples. The static mixture of four dimensions and four components is build and data sample is generated. An array of matrix ARX LS components is build:

```
ncom = 4;                                % number of components
ndat = 500;                              % length of data
DATA = zeros(4, ndat);                   % data sample

ychns = [1 2 3 4];                       % modelled channels in component
str   = [0; 1];                          % static factor structure
Com   = comarxls(ychns, str);            % build matrix ARX LS component
cove  =  0.2*eye(4);                     % noise variance
Eth   = zeros(4,1);                      % regression coefficients

for com = 1:ncom                         % generate array of components
    Com.Eth   = Eth  + 2*rand(4,1);
    Com.cove  = cove + 0.2*rand(4,4);
    Coms{com} = Com;
    Eth = Eth + ones(4,1);
end
```

A mixture is build and data sample generated:

```
Mix   = mixconst(Coms,[4 3 2 1]);        % mixture simulator
mixsimul(Mix, ndat);                     % get data sample
```

Data clusters and mixture clusters are scanned, the results are in the Fig. 21 and 22.

```
datascan;                                % data clusters
mixscan(Mix);                            % mixture clusters
```

A prediction will be displayed in the Fig. 22 subplot 1. In the subplot 2, the prediction is done inside "mixgrid" function. Both plots are equal.

```
pchns    = [1 4];                        % predicted channels
cchns    = [2 3];                        % channels in condition
TIME     = 500;                          % TIME specified
psi0     = [1.4  2.6; 2  3];             % zero-delayed data vector
pMix     = mixpro (Mix,pchns,cchns,psi0); % mixture prediction
```

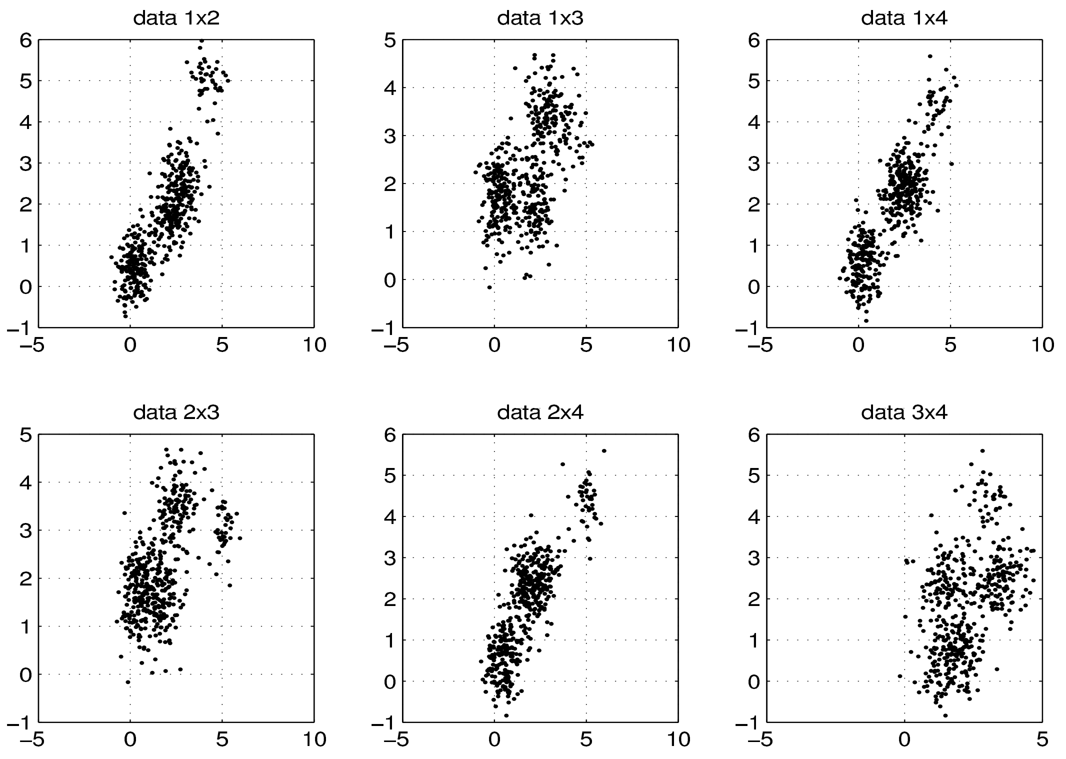

Figure 21: Data clusters {zvisual1}

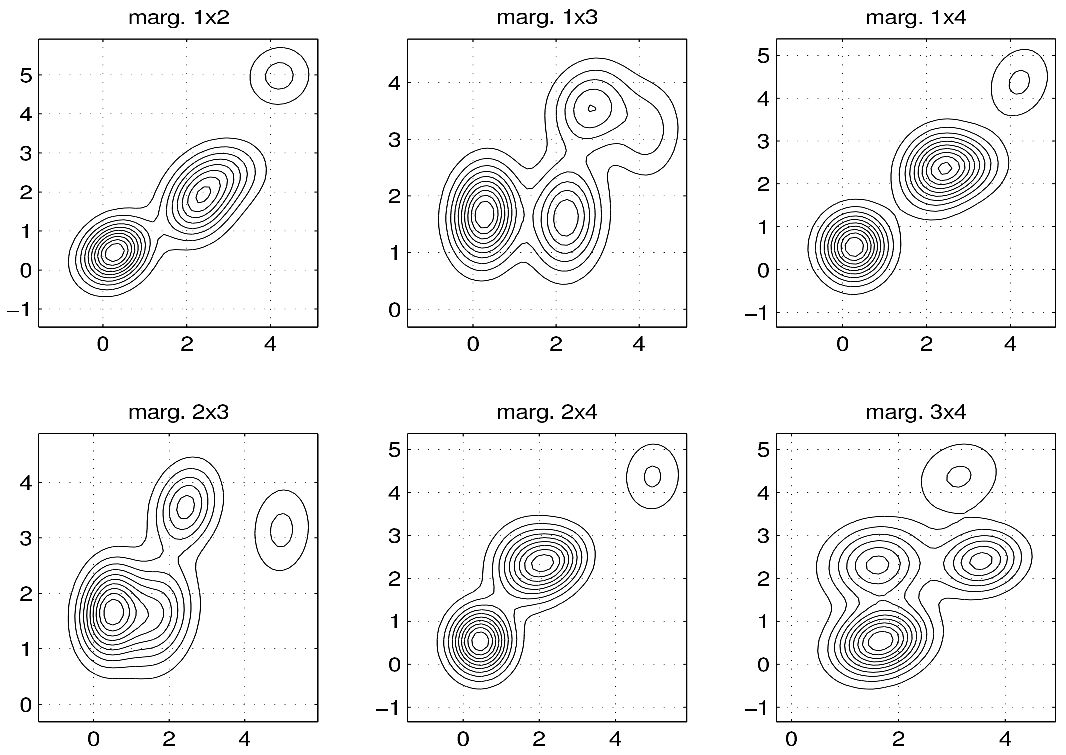

Figure 22: Mixture clusters {zvisual2}

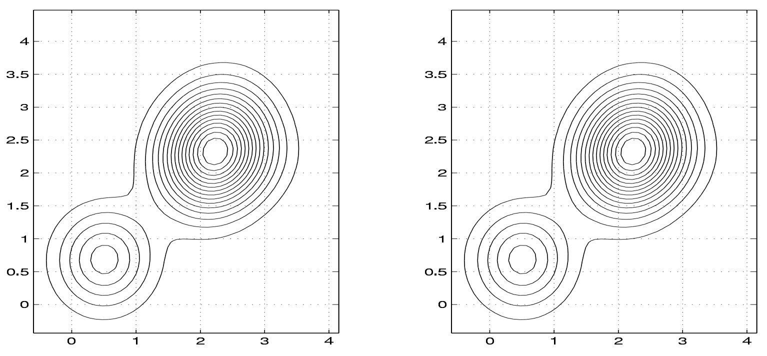

Figure 23: Mixture prediction contours {zvisual3}

```
[x,y,z] = mixgrid(Mix,pchns,cchns,psi0);  % plot coordinates
contour(x,y,z,15);

[x,y,z] = mixgrid(pMix);  % plot coordinates
subplot(122);
contour(x,y,z,15);
```

visual.tex, zvisual.m by PN June 25, 2004

13 Model validation

{validate}

The learning and prediction methods offer a good background for model validation. This section summarizes the procedures that are the most successful in processing of real data samples.

13.1 Test data sample

{data4dm}

Simulated data sample is used with four channels and four components. Changes of components are modeled as a Markov chain, see Section 17.3. This leads to the data sample that is "problematic" from the point of view of model validation.
The array of components for simulation is generated.

```
nchn  = 4;                             % number of data channels
ncom  = 4;                             % number of components
ndat  = 4000;                          % length of data sample
DATA  = zeros(nchn, ndat);             % pre-allocated data sample

str   = [0; 1];                        % static component structure
ychns = 1:4;                           % modeled channels
Com   = comarxls(ychns, str);          % build matrix ARX LS component
cove  = 0.2*eye(nchn);                 % diagonal of noise covariance
Eth   = zeros(nchn,1);                 % regression coefficients

for com = 1:ncom                       % generate array of components
    Com.Eth   = Eth  + 2*rand(4,1);
    Com.cove  = ltdl(cove + 0.2*rand(4,4));
    Coms{com} = Com;
    Eth = Eth + ones(4,1);
end
```

The component weights are selected (not normalized here), mixture simulator is build and Markov transition probability of jumps between components is selected.

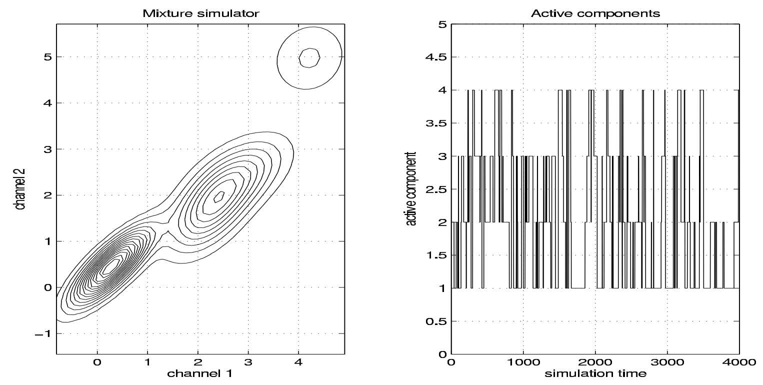

Figure 24: Mesh plot of simulator {zdata4dm}

```
Sim   = mixconst(Coms,[4 3 2 1]);        % mixture simulator
global CUMTAB ACTIVE
   CUMTAB = [0.94 0.02 0.02 0.02          % Markov transition probability
             0.02 0.94 0.02 0.02
             0.02 0.02 0.94 0.02
             0.02 0.02 0.02 0.94];
```

The recursive simulation allows to record which component is active in each simulation time.

```
actives = zeros(1,ndat);                 % trajectory of active component
for TIME=1:ndat                          % simulation cycle
    mixsimul(Sim);                       % one simulation step
    actives(TIME) = ACTIVE;              % get trajectory
end
```

The mixture simulator is displayed in Fig. 24 subplot 1. The plot of active components in time is in Fig. 24 subplot 2.

Note: for comparison, we use another data sample obtained by the same procedure but without Markov jumps between components.

zdata4dm.m

13.2 Model validation via simulation {Gvalidate}

The initial model is build. The prior knowledge is that the mixture is a static one. The initialization follows a standard pattern:

```
load data4dm                             %  load data sample
...  plot mixture simulator ...
Mix0  = genmixe;                         % build initial mixture
frg   = defaults('frg');                 % default forgetting rate
niter = 10;                              % number of iterations
opt   = 'q';                             % option: quasi-Bayes estimation
Mix = mixinit(Mix0,frg,ndat,niter,opt);% mixture initialization
```

The resulting mixture is flattened and estimated using iterative quasi Bayes estimation:

```
niter = 10;                              % number of iterations
Mix0  = mixflat(Mix);                    % mixture flattening
Mix   = mixest(Mix0,frg,ndat,niter);     % iterative quasi Bayess estimation
...  plot estimated mixture ...
```

The mixture simulator is displayed in Fig. 25 subplot 1, the estimated mixture in the subplot 2.

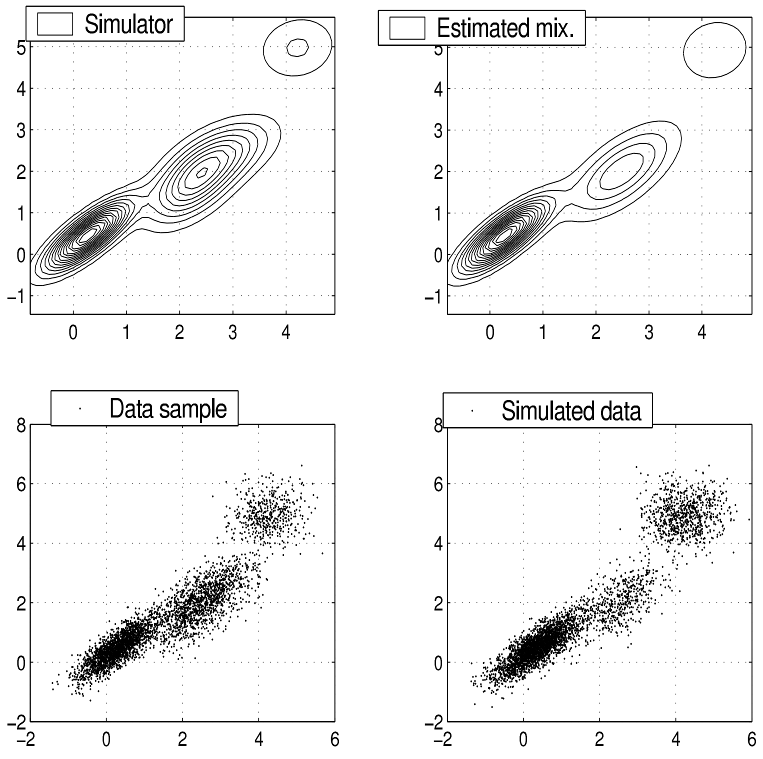

Figure 25: Mixture validation via simulation {zversim}

subplot	contains
1	mixture simulator
2	estimated mixture
3	data clusters
4	simulated data clusters

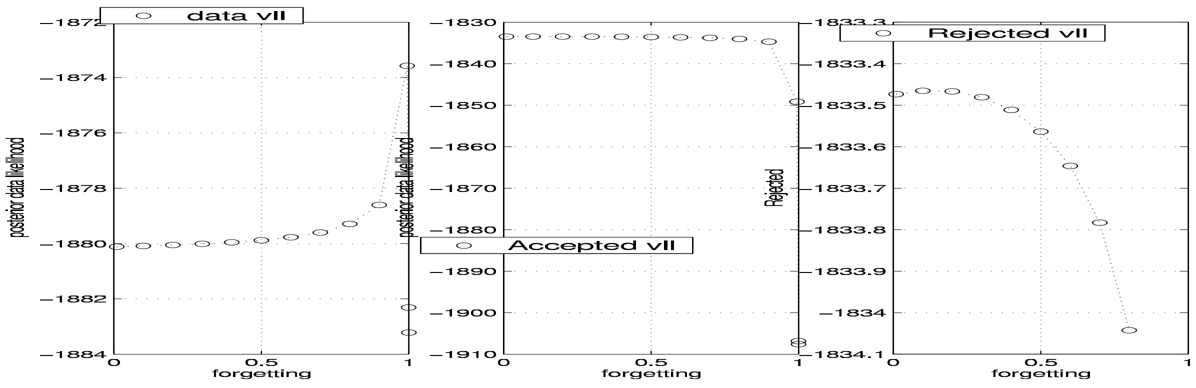

Figure 26: Model validation via stabilized forgetting {zverfrg}

subplot	contains
1	rejected model
2	accepted model
3	accepted model, cut

The estimated mixture can be used for simulation. The simulated and real data are visually compared. This method is a very rough method of model validation. It is suitable for static mixture only.

```
plot(DATA(1,:),DATA(2,:),'.');
mixsimul(Mix,ndat);
plot(DATA(1,:),DATA(2,:),'.');
```

The plots are in Fig. 25, subplot 2 and 3. The correspondence is obvious.

13.3 Stabilized forgetting

The validated model is taken as alternative model reflecting "sure" information in estimation with stabilised forgetting. By comparing posterior data likelihoods of models resulting from estimation with different forgetting rates, the contribution of the additional data processing to the sure model can be judged.

The model is taken as validated if the smallest frogetting rates are the best. In the case, the additional processing brings nothing to the estimated model.

We use the estimated mixture from the previous section. The forgetting rates are selected and estimation is made with the stabilizing forgetting mixture.

```
ndat  = 500;
Mix0  = mixflat(Mix);                   % mixture flattening
frgs  = [0.01 0.1:0.1:0.9 0.995 0.99999 1];% selection of forgetting rates
for i=1:length(frgs)
    Mix       = mixestim(Mix0, frgs(i), ndat, Mix); % mixture estimation
    mixlls(i) = Mix.states.mixll;       % posterior data likelihood is recorded
end
plot(frgs, mixlls);                     % plot of posterior data likelihood
```

The results is in Fig. 26, subplot 1. The conclusion can be made that the estimated mixture is not a good model.

To show the opposite result, we use the data sample without the Markov jumps among component. The processing proceed as in the previous sections.

The result is in Fig. 26 subplot 2 and 3 (separatelly plotted initial part). In this case, the estimated mixture is a good model.

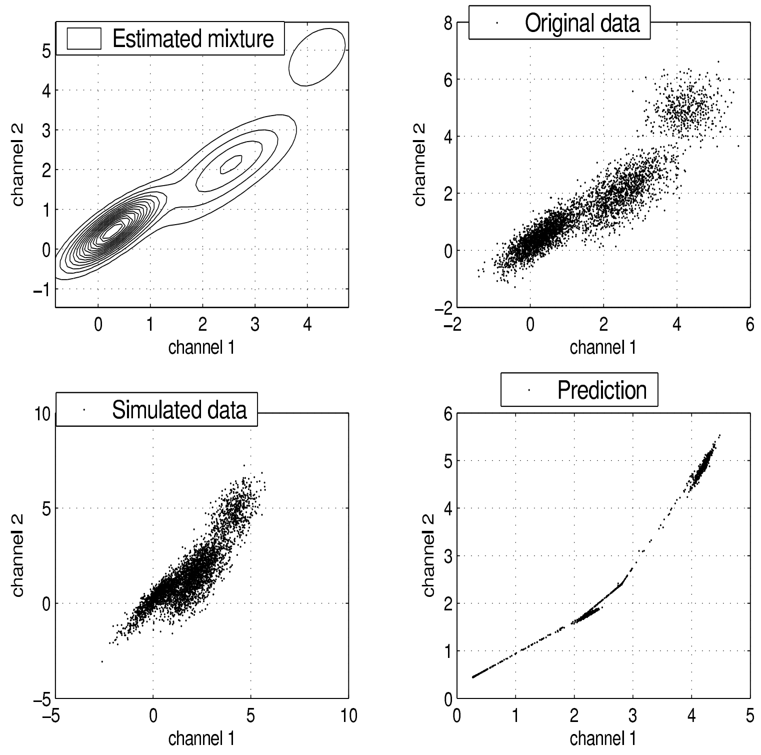

Figure 27: Mixture validation via simulation {zverpre}

subplot	contains
1	estimated mixture
2	data clusters
3	simulated data clusters
4	prediction x data

13.4 Prediction based tests - static mixtures

The mixture prediction is an important mixture characteristics. However, the static mixture marginal projections are static ones. For the case, the method of prediction of groups of data is available. It is discussed in Section 11.4.4. When groups are estimated, the conditional projection depends on data.
The data sample is grouped:

```
nsk  = 2;                              % number of data groupped
preproc({'group', nsk});               % reorganize data
```

Now we have data with eight channels. Initial mixture is build and initialization and estimation done. The estimated mixture is displayed in Fig. 27 subplot 1.

```
Mix0  = genmixe;                       % build initial mixture

frg   = defaults('frg');               % default forgetting rate
niter = 5;                             % number of iterations
opt   = 'q';                           % processing option: quasi-Bayes estimation
Mix   = mixinit(Mix0, frg, ndat, niter, opt); % mixture initialization

niter = 30;
Mix0  = mixflat(Mix);
Mix   = mixest(Mix0, frg, ndat, niter);
... plot of estimated mixture
```

We display data and simulated data clusters in Fig. 27 subplot 2 and 3.

```
plot(DATA(1,:),DATA(2,:),'.');
mixsimul(Mix,ndat);                    % get simulated sample
plot(DATA(1,:),DATA(2,:),'.');
```

Now, the prediction is evaluated over whole data sample. The mixture projection is done for the channels 1 and 2 conditioned by the "new" channels 5 to 8.

```
pchns = [1 2];                         % predicted channel
cchns = 5:8;                           % channels in condition
pMix  = mix2pro(Mix, pchns, cchns);    % predictor
yp    = zeros(2, ndat);
for TIME = 1:ndat                      % getting prediction trajectory
   [Eths, coves, dfcs] = profix(pMix); % prediction
   Eths = [Eths{:}];
   yp(:,TIME) = Eths * dfcs';
end
```

The quality of prediction is evaluated:

```
data = DATA(pchns,:);
ep   = data-yp;
std(data')
ans =
    1.4417    1.5464
std(ep')
ans =
    0.8069    0.8670
```

The standard deviation of data sample is greater then the standard deviation of prediction error. Without data grouping, both deviation are equal.

Data versus prediction appears in the Fig. 27 subplot 4:

```
plot(DATA(1,:),DATA(2,:),'.');
plot(yp(1,:),yp(2,:),'.');
```

13.5 Prediction based tests - dynamic mixtures

We use data sample and estimated mixture from Section 8.2. The prediction test follows.

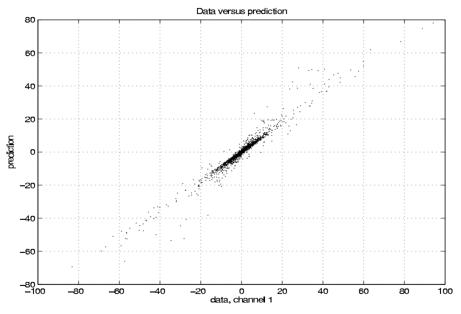

Figure 28: Data x prediction {zverdyn}

```
pchns = 1;                            % predicted channel
cchns = [];                           % channels in condition
pMix  = mix2pro(Mix, pchns);          % predictor
yp    = zeros(1, ndat);
for TIME = 5:ndat                     % getting prediction trajectory
   [Eths, coves, dfcs] = profix(pMix);  % prediction
   yp(TIME) = Eths * dfcs';
end
data = DATA(pchns,:);
ep   = data-yp;
std(data')
ans =
    11.2965
std(ep')
ans =
    2.5220
plot(DATA(1,:),yp,'.');
```

The std. of prediction is again much smaller then std. of data. The plot of data versus prediction is in Fig. 28.

validate.tex, zdata4d.m zdata4dm.m zverdyn.m zverfrg.m zverpre.m zversim.m by PN June 25, 2004

14 Channels descriptions

{channels}

An information about individual data channels is needed in design and use of advisory system. This information is encoded in a cell vector whose cells contain structures describing the individual channels. The cell vector is referred to as *channel descriptions*, the structure describing individual channel is referred to as *channel description*. The relevant cryptonyms are `Chns` and `Chn`.
For a selection of channels, the channels description is build by function "chnconst":

```
chns = 1:4;                           % list of channels
Chns = chnconst(chns)                 % build channels description
Chns =
    [1x1 struct]    [1x1 struct]    [1x1 struct]    [1x1 struct]
```

A (default) channel description is the structure:

```
Chn{3}=
        chn: 3              −> channel number (1,2,...)
       name: 'channel 3'    −> name of the channel
      oitem: []             −> visibility by operator (0 | 1)
    raction: []             −> available for control (0 | 1)
       prty: []             −> presentation priority (< 0,1 >)
       type: 1              −> 1-continuous, 0-discrete (0 | 1)
     drange: []             −> desired range ([min, max])
     prange: []             −> physical range ([min, max])
     irange: []             −> increment range ([min, max])
      scale: []             −> scaling: vector of 2 elements
```

Notes:

- the fields that are empty, must be specified by the user and they are checked for their completeness.
- the `drange` is desired range for advisory system design;
- the `prange` is physical range that is used for comparison with desired scaling; both ranges are set as vector of minimum and maximum values;
- the "scale" field contains additive and multiplicative constants applied to DATA, see "Data preprocessing".

The following rules apply:

the channels description can substitute arguments of a selected functions where appropriate, e.g.

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

can be substituted by:

```
[aMix,aMixu]=inisyn(Mix,Mixu,Chns);
```

the Mixtools processing functions gets information only from arguments but, the superstructure functions (as GUI) can use it as a global definition. The current channels description reside in then the global cell vector `CHANNELS`;

the current channels description is automatically edited by selected processing functions, e.g. the preprocessing `preproc` edits the value of "scale" if the global CHANNELS cell vector exists in workspace;

14.1 Access to description fields

It is easy to get/set the channels description fields using basic Matlab means. The functions "chnset" and "chnget" are available for designers:

`Chns = setchn(Chns, chns, field, values) values = getchn(Chns, chns, field)`

where the arguments are

Chns	channel descriptions
chns	list of channels to be accessed or empty for all channels
field	the field to be set (character string)
values	values of the fields

Simple and self-explanatory examples follow. A channel descriptions object is build, its values set and get:

```
Chns   = chnconst([1 4 17 3]);               % build channels description
Chns   = chnset(Chns, [4 3],  'oitem', 0);   % set fields
Chns   = chnset(Chns, [1 17], 'oitem', 1);   % set fields

oitems = chnget(Chns, [], 'oitem')
oitems =
     1     0     1     0

oitems = chnget(Chns, 17, 'oitem')
oitems =
     1
```

Note: the fields must not be empty when accessed by "chnget".
The names are accessed as cell vector, e.g.

```
names  = chnget(Chns, [], 'name')
names =
    'channel 1'    'channel 4'    'channel 17'    'channel 3'
name   = chnget(Chns, 17, 'name')
name =
    'channel 17'
```

Ranges are get in the form of two rows, the first one contain minimum, the second one maximum, e.g.

```
Chns   = chnset(Chns, [1 17], 'drange', [-1.1 -1.2; 1.1 1.2] );
chnget(Chns, 17, 'drange')
ans =
   -1.2000
    1.2000
chnget(Chns, [1 17], 'drange')
ans =
   -1.1000   -1.2000
    1.1000    1.2000
```

channels.tex zchnset.m zchnget.m by PN June 25, 2004

15 Design and advising

{Design}{advising}

15.1 Academic design

{academic}

Academic design consists of four stages. First, during the *preparatory stage*, user have to define *target mixture* Mixu in a form of one component usually static mixture where the parameters represent the desired values for corresponding channels and noise covariances defining the range how strict these values should be kept. The user's qualification of components should be given in the form of vector "ufc", having the length equal to the number of components. Vector elements assign the priorities of mixture components.

The second, *initialisation* stage, prepares main structures of advisory design; converts identified mixture Mix, gained from the learning phase, as well as user target Mixu to the advisory type mixtures. If the user was not able to set the preferences to a components reasonably, function "ufcgen" can be used here to generate vector "ufc" which qualifies unstable components with zero weight. Preliminary analysis of the components can be done by a function "stedopt", which calculates steady state losses of individual components.

The third, *optimisation* stage, performs search of the optimal components that have minimal KL distance from the user defined target Mixu. The last stage computes the probabilistic weights of the components found during optimisation stage. Only this stage directly uses data and can be called an on-line phase.

Preparatory stage

This stage is realised by function "target", which is called:

```
[Mixu,ychns] = target(Chns)
```

The arguments of the function are:

Mixu	constructed user target (one component ARX mixture)
ychns	list of modelled channels in component
Chns	cell vector with channels descriptions (see Section 14).

Using the given channels descriptions, function "target" creates one component ARX mixture that expresses management aims. Besides, it regroup list of modelled channels in component to the following form [channels with o-innovations, channels with surplus p+, channels with recognisable actions]. Channels order inside of each group is set respectively to ascending priority given by the user. Notice, that before using "target", the channels descriptions should be scaled (function "ScaleDescriptions") in accordance with used preprocessing see Section 6 and Section 14.

Initialisation

This stage is realised by function "inisyn", which can be called in two ways:

```
[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 (see Section 14)
pochn	list of channels with o-innovations
uchn	list of channels with recognisable actions (can be omitted for academic design).

The tasks covered by initialisation stage are implemented by employing function "synmixi".

The function "synmixi" converts ARX mixture Mix to advisory type mixtures (a-mixture). The function "synmixi" is called as follows:

```
aMix = synmixi(Mix,uchn,strc)        % converts Mix to a-mixture
aMix = synmixi(Mix,uchn)             % strc=[]
aMix = synmixi(Mix)                  % strc=[], uchn=[]
```

The arguments of the function are:

aMix	constructed a-mixture
Mix	learnt ARX mixture
uchn	list of channels with recognisable actions (can be omitted)
strc	common structure of a-mixture (can be omitted)

Function "synmixi" finds common full structure strc of all components (and factors) in the mixture. If strc is known in advance, it can be used as an input parameter. For academic design, list of channels with recognisable actions uchn is not relevant, so it need not be used or can be set uchn=[].

The "synmixi" also creates a-mixture states used in advises design, so-called *advisory states*, that consist of the following fields:

- `strc` - common structure of data vectors
- `ufc` - vector qualifying components: dangerous component (0), not dangerous (positive number)
- `kc` - lift of quadratic forms
- `UDc` - cell vector of $U'DU$ decompositions of the 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 recognisable actions
- `pochn` - list of channels with o-innovations

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

Another basic function from the initialisation stage is "stedopt". It performs preliminary analysis of components of the mixtures aMix and aMixu. It consists of calculation of KL distances (5) of each component of the mixture aMix to a user defined target aMixu. These distances are represented by a lift "kc" and a kernel "UDc" calculated as a infinite horizon quadratic losses. These vales are saved in corresponding states of a mixture aMixu and can be used at the subsequent optimisation stage. Function "stedopt" is called in the following way:

```
aMixu = stedopt(aMix,aMixu).
```

During the analysis, unstable components are also detected. The function "ufcgen" generates normalised vector "ufc" qualifying components with zero entries for unstable components:

```
ufc= ufcgen(aMix,aMixu).
```

Optimisation

The function "aloptim" performs optimization. It judges behavior of components by evaluating lifts and kernels of KL distance (5) of respective components to a user given target aMixu. Thus, it prepares necessary information for generating optimal pf on recommended pointers to components. The function "aloptim" is called in the following way:

```
aMix = aloptim(aMix,aMixu,ufc,nstep,chis)
aMix = aloptim(aMix,aMixu,ufc,nstep)       % chis=1 is assumed
aMix = aloptim(aMix,aMixu,ufc)             % chis=1, nstep=200 is set
```

The arguments of the function are:

aMix	learnt ARX mixture, converted to advisory type
aMixu	user target, converted to advisory type
nstep	parameters determining horizon for the evaluation of the KL distance
ufc	user defined vector qualifying components
chis	an indicator of receding horizon (chis=1) or iterations-spread-in-time (chis=-1) IST strategy

Argument "nstep" can be either scalar or two element vector, i.e. "nstep=[horizon,period]". Function "aloptim" calculates KL distance so that partial quadratic losses of particular components are properly mixed together. The way of mixing can be tuned and depends on the parameters "horizon" and "period". Parameter "period" determines a number of iterations in which the lifts and kernels are evaluated separately for each component, while "horizon" is a number of these periods. If "nstep" is scalar, the default value of "period" is set to 1.

Examples: assignments nstep = [1000,1] and nstep = 1000 are equivalent and indicate that mixing take place in each iteration, while nstep = [1,1000] means that the kernels and lifts are calculated independently (likewise in "stedopt").

The kernel and lifts of the KLD can be influenced by corresponding values in user defined mixture aMixu, i.e aMix.states.kc and aMix.states.UDc. By default these values are set to zeros. If values representing independent losses of components need to be used, the function "stedopt" should be called before "aloptim".

To solve the optimisation task, three auxiliary functions are developed: "ricexp", "ricshift" and "ricpen". First function computes the expectation part of the of quadratic loss update for the given component and factor; second function performs parameter independent step of optimisation and third function computes contribution of penalisation of o-innovations to kernel and lift of quadratic form.

Probabilistic weights computation

Function “algen” computes probabilistic weights for academic advisory design, i.e. recommended data dependent pf of pointers to components, and overwrites existing aMix.dfcs by the recommended values. The function ”algen” is called in the following way:

```
aMix = algen(aMix,ufc).
```

The arguments of the function are:

aMix	constructed a-mixture
aMixu	user target, converted to advisory type
ufc	user defined vector qualifying components

Global variables TIME and DATA are used inside the function.

academic, MK PN, JB June 25, 2004

15.2 Industrial design

{industrial}

Although the industrial design was developed as well, its practical use is rather limited and detailed description is omitted here. For the reference purposes, the basic steps of industrial design can be found in simultaneous design which is more general type of design, see Section 15.3.

industrial, MK PN, JB TG June 25, 2004

15.3 Simultaneous design

{simultaneous

Similarly to the academic design, simultaneous design consists of four stages. During the *preparatory stage*, user have to define *target mixture* Mixu usually in a form of one component static mixture. Mixture parameters set the desired values for corresponding channels as well as noise covariances defining the range how strict these values should be kept. The user’s qualification of components should be given in a form of a vector ”ufc”, having the length equal to the number of components. Vector elements assign the priorities of mixture components.

The second, *initialisation* stage, prepares main structures of advisory design; converts mixture Mix, gained from the learning phase, as well as user target Mixu to the mixtures of advisory type. Preliminary analysis of the components can be done by a function “stedopt”. If user was not able to set the preferences to a components reasonably, function “ufcgen” can be used here to generate vector “ufc” which will qualify unstable components with zero weight.

The third, *optimisation* stage, performs search of the optimal components that have minimal KL distance from the user defined target and substitutes all factors which model recognisable actions by calculated optimal ones in a mixture aMix.

The last stage computes the probabilistic weights of the components found during optimisation stage. Only this stage directly uses data and is called on-line phase. Unlike academic design, the on-line stage in simultaneous design also contains computation of the most probable optimal recognisable actions which can be applied directly to the system. The recommended recognisable actions are calculated using the prediction.

Preparatory stage

This stage is realised by function ”target”, which is called:

```
[Mixu,ychns] = target(Chns)
```

The arguments of the function are:

Mixu	constructed user target (one component ARX mixture)
ychns	list of modelled channels in component
Chns	cell vector with channels descriptions (see Section 14).

Using the given channels descriptions, function ”target” creates one component ARX mixture that expresses management aims. Besides, it regroup list of modelled channels in component to the following form [channels with o-innovations, channels with surplus p+, channels with recognisable actions]. Channels order inside of each group is set respectively to ascending priority given by the user. Notice, that before using ”target”, the channels descriptions should be scaled (function ”ScaleDescriptions”) in accordance with used preprocessing see Section 6 and Section 14.

Initialisation

This stage is realised by function "inisyn", which can be called in two ways:

```
[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 (see Section 14)
pochn	list of channels with o-innovations
uchn	list of channels with recognisable actions.

The tasks covered by initialisation stage are implemented by employing function "synmixi".

The function "synmixi" converts ARX mixture Mix to advisory type mixtures (a-mixture). The function "synmixi" is called as follows:

```
aMix = synmixi(Mix,uchn,strc)        % converts Mix to a-mixture
aMix = synmixi(Mix,uchn)             % strc=[]
```

The arguments of the function are:

aMix	constructed a-mixture
Mix	learnt ARX mixture
uchn	list of channels with recognisable actions
strc	common full structure of a-mixture (can be omitted)

Function "synmixi" finds common full structure strc of all components (and factors) in the mixture. If strc is known in advance, it can be used as an input parameter.

The "synmixi" also creates a-mixture states used in advises design, so-called *advisory states*, that consist of the following fields:

- `strc` - common structure of data vectors
- `ufc` - vector qualifying components: dangerous component (0), not dangerous (positive number)
- `kc` - lift of quadratic forms
- `UDc` - cell vector of $U'DU$ decompositions of the 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 recognisable actions
- `pochn` - list of channels with o-innovations

Beside that, a factor state Mixc.Facs{·}.states.pEth is set. This state is a pointer table enabling expanding of Facs.Eth to a full structure strc.

Another basic function, which can be used here, is "stedopt". It performs preliminary analysis of components of aMix and aMixu. It consists in calculation of KL distances (5) of each component of aMix to a user defined target aMixu. These distances are represented by a lift "kc" and a kernel "UDc" calculated as a infinite horizon quadratic losses. These values are saved in the corresponding states of a mixture aMixu. During the analysis, unstable components are detected. The function "stedopt" generates normalized vector qualifying components with zero entries for unstable components and saves the vector to the states of mixture aMix.states.ufc. This function "stedopt" is called in the following way:

```
aMixu = stedopt(aMix,aMixu).
```

During the analysis, unstable components are also detected. The function "ufcgen" generates normalized vector "ufc" qualifying components with zero entries for unstable components:

```
ufc= ufcgen(aMix,aMixu).
```

Optimisation

The function "soptim" performs optimization and judges behavior of components. It evaluates lifts and kernels of KLD distance (5) of respective components to a user given target aMixu and thus prepares necessary information for generating optimal pf on recommended pointers to components. The function "soptim" is called in the following way:

```
aMix = soptim(aMix,aMixu,ufc,nstep,chis)   % performs optimisation
aMix = soptim(aMix,aMixu,ufc,nstep)        % chis=1 is assumed
aMix = soptim(aMix,aMixu,ufc)              % chis=1, nstep=200 is set)
```

The arguments of the function are:

aMix	learnt ARX mixture, converted to advisory type
aMixu	user target, converted to advisory type
nstep	horizon for the evaluation of the KL distance
ufc	user defined vector qualifying components
chis	an indicator of receding horizon (chis=1) or iterations-spread-in-time (chis=-1) IST strategy

Argument "nstep" can be either scalar or two element vector, i.e. "nstep=[horizon, period]". Function "soptim" calculates KL distance so that partial quadratic losses of particular components are properly mixed together. The way of mixing can be tuned and depends on the parameters "horizon" and "period". Parameter "period" determines a number of iterations in which the lifts and kernels are evaluated separately for each component, while "horizon" is a number of these periods. If "nstep" is scalar, the default value of "period" is set to 1.

Examples: assignments nstep = [1000,1] and nstep = 1000 are equivalent and indicate that mixing take place in each iteration, while nstep = [1,1000] means that the kernels and lifts are calculated independently (likewise in "stedopts").

The kernel and lifts of the KLD can be influenced by corresponding values in user defined mixture aMixu i.e aMix.states.kc and aMix.states.UDc. By default these values are set to values representing independent losses of components in function "stedopts". If they are to be used "stedopts" is to be called before "soptim".

To solve the optimisation task, four auxiliary functions are developed: "ricexp", "ricshift", "ricpen" and "ricpenu". First function computes the expectation part of the of quadratic loss update for the given component and factor; second function performs parameter independent step of optimisation; third function computes contribution of penalisation of o-innovations to kernel and lift of quadratic form; the last function computes contribution of penalisation of recognisable actions.

Computation of the probabilistic weights

Function "algen" computes probabilistic weights, i.e. recommended data dependent pf of pointers to components, and overwrites existing aMix.dfcs by the recommended values. The function "algen" is called in the following way:

```
aMix = algen(aMix,ufc).
```

The arguments of the function are:

aMix	constructed a-mixture
aMixu	user target, converted to advisory type
ufc	user defined vector qualifying components

Global variables TIME and DATA are used inside the function.

Computation of the recommended recognisable actions

The most probable recognisable action can be obtained by calculating prediction. The component of aMix having the highest computed probabilistic weight, is the closest component to the user defined target Mixu.

To obtain the most probable recognisable action, the prediction should be computed. This can be done on-line as follows:

```
....
    pMix = mix2pro(aMix);                              % build predictor pMix
    for TIME=start_time+1:end_time;                    % time loop
        pMix = mixcopy(aMix,pMix);                     % copy advice to predictor
        aMix = algen(aMix,ufc);                        % generating an advice
        [Eth,coves,alpha] = profix(pMix);              % build prediction
        for com = 1:size(Eth,2)                        % loop over components
            pred(:,TIME) = pred(:,TIME)+Eth{com}*aMix.dfcs(com);   % prediction
        end;
    end;
```

Global variables TIME and DATA are used inside the functions.

simultan, MK PN, JB TG June 25, 2004

15.4 Design validation

{validation}

The validation of design can be done in different ways. At present two of them are used:

Closed-loop simulation

Two different mixtures learned on the same data are employed, say Mix1 and Mix2. Mixture Mix1 is then used to design advisory mixture aMix1 and subsequent advising, while second mixture Mix2 is used for system simulation in the closed loop. The behaviour of the resulting closed-loop helps the user judge about the quality of advising.

Criterion using

For the design validation the value of the following criterion can be used:

$$\frac{1}{T}\sum_{i=1}^{T}\left[Q^y(y_i - Y_i)^2\right], \tag{16}$$

where values Q^y, Y_i are given by system requirements. Function "criter" calculating the value of the criterion is called in the following way:

```
crit = criter(Mixu,endtime,starttime).
```

The arguments of the function are:

crit	the value of criterion
Mixu	constructed user target (one component ARX mixture)
endtime	end time determining the time interval for criterion computation
starttime	start time determining the time interval for criterion computation.

An example of using both validation tests is presented in Appendix C.

validate.tex, by TG June 25, 2004

15.5 Signaling

{signaling}

The signalling serves to stimulate the operator to take some actions when the o-system behaviour is significantly different from the desired one. The design of signalling strategy can be viewed as a special kind of academic design.

To make the advisory system perform the signalling strategy, the user should define pf of signalling actions "ufs" and set thresholds for its values. Thresholds are necessary for mapping the signalling actions.

Off-line initialisation stage

The function "asignal" performs computation and comparison of steady state lifts and kernels of KL distance (5) of original identified mixture Mixor (converted to advisory type) and designed advisory mixture aMix to a user given target aMixu. The function "asignal" is called in the following way:

```
[aMixor,aMix]=asignal(aMixor,aMix,aMixu,ufc,ufs,nstep);
```

The arguments of the function are:

aMixor	original learnt ARX mixture, converted to advisory type
aMix	advisory ARX mixture obtained from the design
aMixu	user target, converted to advisory type
ufc	user defined vector qualifying components
ufs	user defined vector of pf of signalling actions
nstep	parameters determining horizon for the evaluation of the KL distance

Argument "nstep" has the same meaning as in academic or simultaneous design, see Sections 15.1

Evaluation of signalling probabilities

Function "asiggen" computes signalling probabilities, i.e. data dependent pf of signalling actions. The function "asiggen" is called in the following way:

```
fs = asiggen(aMixor,aMix,ufs).
```

The arguments of the function are:

fs	signalling probabilities
aMixor	original learnt ARX mixture in advisory form
aMix	constructed a-mixture
ufs	user defined vector qualifying components

Global variables TIME and DATA are used inside the function.

signaling, TG June 25, 2004

16 Tutorial on design and advising

{tutoriala}

In this section, the examples of design and advising are discussed. The examples are referred in the following sections, too. a-Mixtures are derived objects whose construction is performed indirectly. Construction of a-mixture requires the following steps:

1. Select the basic advising scenario that requires:
   (a) channels with predicted data to be considered,
   (b) needed type of design (academic, industrial, simultaneous) to be chosen,
   (c) *o-innovations*, i.e. data observable by operator to be defined,
   (d) *recognizable actions*, i.e. data that can be chosen by operator to be considered.
2. Express management aims as target one-component mixture Mixu.
3. Convert learnt mixture Mix and user defined target Mixu into a-mixtures, i.e. a common structure of regression vectors for individual components should be created as well as space for specific a-states.

4. Perform preliminary analysis of components, assuming their permanent activity. It distinguishes dangerous and non-dangerous components and allows to specify prior (static) preference among components.

5. Perform advisory design generating the ideal mixture.

6. Validate result of design and re-iterate, if needed.

The above steps are performed in *off-line mode.* Only the last step can be re-computed in *on-line mode* if we deal with *adaptive advisory system* that permanently learns mixture Mix.

In *on-line mode* design, results are presented by introducing adequate projections of the ideal mixture to the operator. Selection of proper projections and indication of a need for action can be optimized in this mode using presentation and signaling design.

tutorial, TG June 25, 2004

16.1 Academic design and advising

{tutora}

Academic design modifies component weights in learnt mixture Mix so that the resulting ideal a-mixture is as close as possible to the *user specified target* Mixu reflecting management aims. At present, Mixu has just single component. In usual static specification, offsets of particular factors corresponding to respective channels represent desired value of the channel and variance specifies the desired width of its variations. The following example shows how the academic advises can be designed.

A simple nonlinear system has been chosen to demonstrate the behavior of the academic design. The system is governed by the following equations.

$$\begin{aligned} y_t &= \frac{306}{300+k_t} y_{t-1} + e_{1;t} \\ u_t &= k_t y_{t-1} + e_{2;t} \\ k_t &= k_{t-1} + e_{3;t}, \end{aligned} \qquad (17) \quad \{\texttt{sys}\}$$

where observed three-dimensional data record d_t consist of three scalar signals y_t, u_t, k_t and the normal white noise $e_t = [e_{1;t}, e_{2;t}, e_{3;t}]'$ has zero mean, mutually independent entries with variances [1 .001 1000].

These relationships were linearized around 8 different values of in the "reasonable" interval of $k \in [0,\ 105]$.

The corresponding mixture of 8-component is generated by the file "model1.m":

```
k    = [3, 12, 24, 37.5, 52.5, 67.5, 82.5, 97.5, 105]  % k-grid
n    = length(k)-1;                    % number of components
str1 = [1 3 0; 1 1 1];                 % 1th channel structure
str2 = [1 3 0; 1 1 1]                  % 2nd channel structure
str3 = [0; 1];                         % 3rd channel static structure
coms = [];                             % list of components
Facs=[];                               % list of factors
```

Model generation:

```
for i=1:n
    Fac1 = facarxls(1,str1);           % model of the 1st channel
    Fac1.cove = 1;                     % factor variance
    pom2 = (k(i+1)+k(i))/2;            % centers of the intervals
    pom1 = 300 + pom2;
    Fac1.Eth  = [306/pom1 -306/(pom1*pom1)...  % 1st channel regression coefficients
               306/(pom1*pom1)*pom2];


    Fac2 = facarxls(2,str2);           % model of the 2nd channel
    Fac2.Eth  = [-pom2 -100/(20+pom2)  100/(20+pom2)*pom2];  % regression coefficient
    Fac2.cove = 10e-6;                 % noise variance

    Fac3 = facarxls(3,str3);           % model of the 3rd channel (k)
    Fac3.Eth  = (k(i+1)+k(i))/2;       % k-offsets
    Fac3.cove = (k(i+1)-k(i))/3;       % k-variances
    Facs{3*i-2}=Fac1; Facs{3*i-1}=Fac2; Facs{3*i}=Fac3; % array of factors
    coms = [coms; 3*i-2,3*i-1, 3*i];    % new component
end

dfcs = ones(1,n)./n;                  % component weights
Sim  = mixconst(Facs,coms,dfcs);      % mixture simulator
Mix  = mix2mix(Sim,21);               % conversion to ARX mixture
save model1
```

User wants to have y_t and u_t close to zero while keeping k near the value 75.

The construct of the user defined mixture representing his requirements is shown in file "penal1.m":

```
coms0 = [ 1  2  3];                   % components
str0  = [0; 1];                       % static structure
Fac01 = facarxls(1,str0);             % build factors
Fac02 = facarxls(2,str0);
Fac03 = facarxls(3,str0);
Fac01.cove = 1;                       % quadratic penalty on y is 1/cove=1
Fac02.cove = 6;                       % quadratic penalty on u is 1/cove=1/6
Fac03.Eth  = 75;                      % set-point of k
Fac03.cove = 20;                      % quadratic penalty on k-75 is 1/cove=1/20
Facs={Fac01 Fac02 Fac03};             % array of factors
dfcs0 = 1;                            % a component weight
[Mixu, maxtd0]= mixconst(Fac0,coms0,dfcs0);
```

Obviously, the variable k_t influences the behavior both of y_t and u_t. The design that aims to make y_t, u_t small and k_t close to 75 has to find a suitable compromise. It lies between high values of k_t, making y_t small but u_t large and small values making u_t smaller but y_t larger.

The result is a vector of probabilities of all components the higher probability indicates the component close to a desired one. Mixture aMix with these probabilities is used to predict desired value of k.

File **acdes.m** perform all steps of academic design. Files **acdesxy.m** for diferent x and y are specified for particular criterion (Mixu) or ufc. Resulting k is shown in picture Fig. 29,

tutora.tex by JB, MK, TG, PN June 25, 2004

16.2 Industrial design and advising

{tutori}

As far this type of design is of limited practical use, the tutorial on it has not been prepared.

tutoritex by PN June 25, 2004

16.3 Simultaneous design and advising

{tutorsim}

Unlike academic design, *simultaneous design* not only modifies component weights in the learnt mixture Mix so that the resulting ideal aMix is close as possible to the *user specified target* Mixu reflecting management aims, but also computes the optimal factors for generating recognisable actions.

At present, Mixu has just single component. In usual static specification, offsets of particular factors corresponding to respective channels represent desired value of the channel and variance specifies the desired width of its variations. The following example shows how the simultaneous advises can be designed.

A simple nonlinear system has been chosen to demonstrate the behavior of the academic design. The system is governed by the following equations.

$$\begin{aligned} y_t &= \frac{306}{300+k_t} y_{t-1} + e_{1;t} \\ u_t &= k_t y_{t-1} + e_{2;t} \\ k_t &= k_{t-1} + e_{3;t}, \end{aligned} \qquad (18)$$

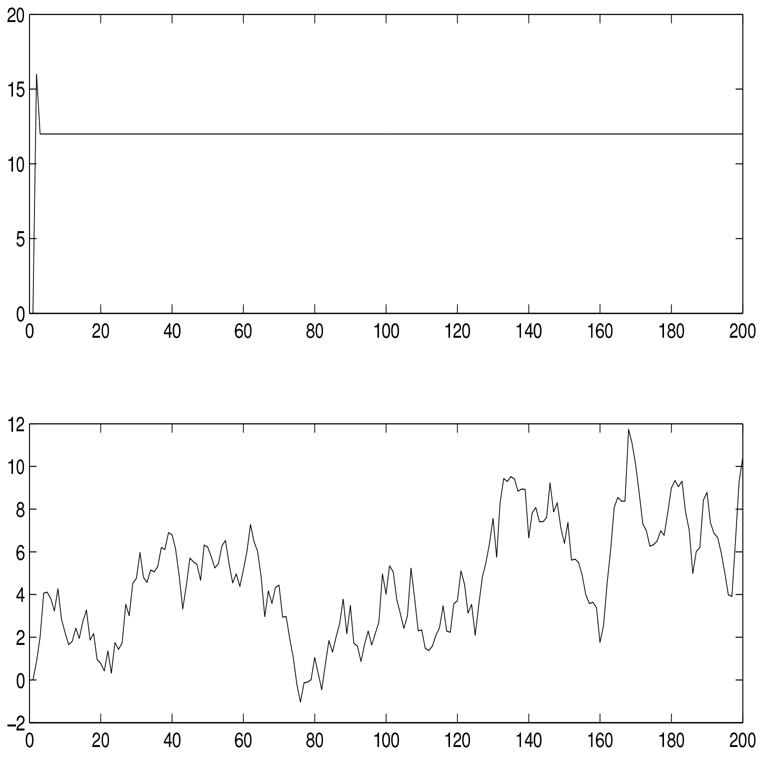

Figure 29: Resulting k and y for the requirements in Mix01 {krit1b1}

where observed three-dimensional data record d_t consist of three scalar signals y_t, u_t, k_t and the normal white noise $e_t = [e_{1;t}, e_{2;t}, e_{3;t}]'$ has zero mean, mutually independent entries with variances [1 .001 1000].

These relationships were linearized around 8 different values of in the "reasonable" interval of $k \in [0,\ 105]$. k=[0,6,18,30,45,60,75,90,105];

The corresponding mixture of 8-component is generated:

```
k    = [0, 6, 18, 30, 45, 60, 75, 90, 105]  % k-grid
n    = length(k)-1;                        % number of components
str1 = [1 3 0; 1 1 1];                     % 1th channel structure
str2 = [1 3 0; 1 1 1]                      % 2nd channel structure
str3 = [0; 1];                             % 3rd channel static structure
coms = [];                                 % list of components
Facs=[];                                   % list of factors


for i=1:n
    pom1 = (k(i+1)+k(i))/2;                % centers of the intervals
    pom2 = 306/(300+pom1);                 %
    pom3 = 1/(1-pom2^2)/5;                 %
    pom4 = 306/(300+pom1)^2;               %

    Fac1 = facarxls(1,str1);               % model of the 1st channel (y)
    Fac1.cove = 1;                         % factor variance
    Fac1.Eth  = [pom2 -pom3*pom4 ...       % 1st channel regression coefficients
                 pom1*pom3*pom4]
    Fac2 = facarxls(2,str2);            % model of the 2nd channel (u)
    Fac2.Eth  = [-pom1 -pom3 pom1*pom3];% regression coefficient
    Fac2.cove = 10e-3;                  % noise variance

    Fac3 = facarxls(3,str3);            % model of the 3rd channel (k)
    Fac3.Eth  = pom1;                   % k-offsets
    Fac3.cove = (k(i+1)-k(i))/3;        % k-variances


    Facs{3*i-2} = Fac1;                 % array of factors
    Facs{3*i-1} = Fac2;
    Facs{3*i}   = Fac3;
    coms = [coms; 3*i-2, 3*i-1, 3*i];   % new component
end

dfcs = ones(1,n)./n;                    % component weights
Sim  = mixconst(Facs,coms,dfcs);        % mixture constructor
Mix  = mix2mix(Sim,21);                 % conversion to ARX type mixture
```

Let us suppose that the user needs to have y_t and u_t close to zero while k be near the value 75. The example of constructing user tagret is shown for the mixture "Mixu":

```
coms0 = [ 1  2  3];                     % components
str0  = [0; 1];                         % static structure
Fac01 = facarxls(1,str0);               % build factors
Fac02 = facarxls(2,str0);
Fac03 = facarxls(3,str0);
Fac01.cove = 1;                         % quadratic penalty on y is 1/cove=1
Fac02.cove = 6;                         % quadratic penalty on u is 1/cove=1/6
Fac03.Eth  = 75;                        % set-point of k
Fac03.cove = 20;                        % quadratic penalty on k-75 is 1/cove=1/20
Facs={Fac01 Fac02 Fac03};               % array of factors
dfcs0 = 1;                              % a component weight
[Mixu, maxtd0]= mixconst(Fac0,coms0,dfcs0);
```

Constructed mixtures expressing user's requirement are realized in file "penal1.m"

Unlike academic design, simultaneous design also provides the most probable recognisable action. For the given example, k_t plays recognisable action role and the user is advised what value of the action should be chosen to operate as close as possible to the specified target.

The main steps of the simultaneous advisory design for the given example are:

```
model1                                 % load system model
penal1                                 % load user specified target
ncom=size(Mix.coms,1);                      % number of components
pochn=[1 2 3];                              % list of channels with o-innovations
niter=200;                                  % number of time iterations
uchn=[3];                                   % channel with recognisable actions
nstep=200;                                  % horizon for evaluation of KLD
ufc=ones(1,ncom);                           % user defined priorities of components
[aMix,aMixu] = inisyn(Mix,Mixu,pochn,uchn);% initialisation
Mixc = soptim(aMix,aMixu,nstep,ufc);            % optimisation
k_new = 16;y = 0;y_old = 0;                % initial values
for TIME=Mix.states.maxtd+1:niter           % main loop on time
    y = (306/(300+k_new))*y+randn;          % new system output
    u = -k_new*y_old;                       % new system input
    DATA(:,TIME) = [y, -k_new*y_old, k_new]'; % new data entry
    yold = y;                               % save value of the output
    aMix = algen(aMix,ufc);         % compute probabilistic weights and
                                            % recommended recognisable actions
end
```

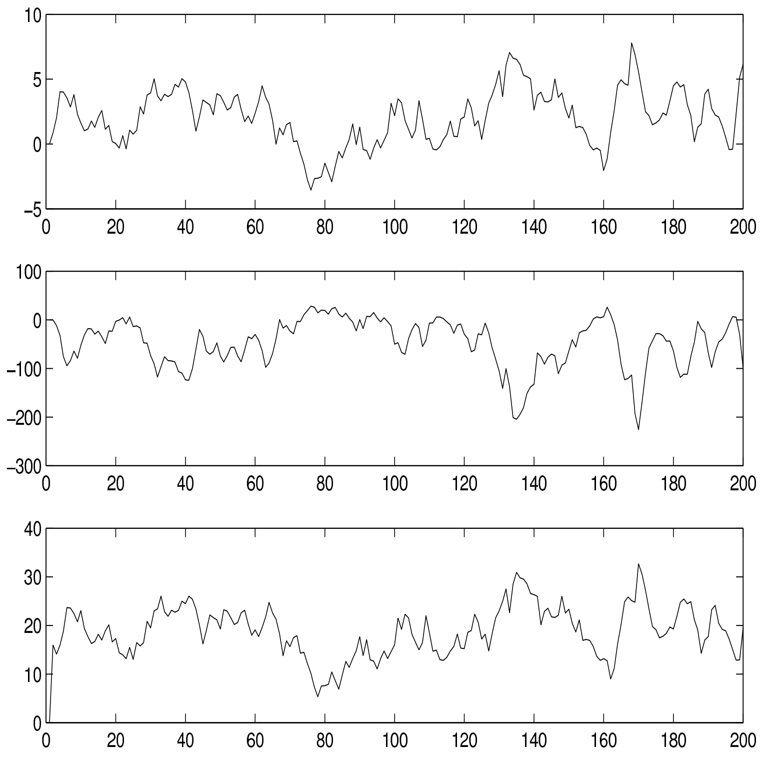

Figure 30: Resulting k and y for the requirements in Mix01 {krit1b}

The element of the vector "aMix.dfcs", having the highest value, points to the closest component to the user defined target "Mixu". Thus, the obtained advisory mixture "aMix" can be used to predict value of k_t, which then be taken as the recommended recognisable action:

```
.....
     [Eth,coves,alpha] = mixpro(aMix,pochn,[],[]); % build prediction on aMix
     pom = Eth1*aMix.dfcs(1);
     for i=2:ncom                                  % loop over components
         pom = pom+Eth{i}*aMix.dfcs(i);
     end;
     k_new = pom(3);                               % new recognisable action
```

In the example (17), the variable k_t influences the behavior both of y_t and u_t. The design, that aims to make y_t and u_t sufficiently small while k_t close to 75, has to find a suitable compromise between these requirements. It lies between high values of k_t, making y_t small but u_t large and small values making u_t smaller but y_t larger. The obtained k_t is shown in picture Fig. 30.

tutorsi.tex by JB, MK, TG, PN June 25, 2004

17 ARX mixture simulation

{simulation}

Mixture simulation serves for development of algorithms, debugging and case studies. Any type of mixture predictor can be used for the simulation. The mixture is internally converted into the form of ARX LS mixture.

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

The simulation fills modelled channels the global matrix "*DATA*" by simulated data. The matrix must be allocated (*pre-allocated*) before the simulation starts.

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

The obligatory pre-allocation makes it possible to fill "DATA" channels by different simulators and/or use specific channels e.g. for control value.

The examples of mixture simulation are in the Section 8.

There are several simulation options that can be selected by setting the fields of "states" of the mixture simulator.

The field contains the cell vector "Facs" that contains settings related to individual simulation factors.

The following options can be set for a factor "fac":

`Sim.states.Facs{fac}.etype`	`= ...`	specify type of process noise
`Sim.states.Facs{fac}.useCth`	`= ...`	specify use of covariance of regression coefficients

17.1 Distribution of process noise

The uniform noise with the type equal to 2 of the factor `fac` is specified by e.g:

```
Sim.states.Facs{fac}.etype    = 2;    % specify type of process noise
```

The noise type is coded as follows:

1	Gaussian
2	uniform
3	lognormal
4	Cauchy

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

17.2 Covariance of regression coefficients in simulation

The uncertainty of regression coefficients expressed by their covariance "Cth" can be respected in simulation. Experts are assumed to exploit this option. It is enabled by setting the field "useCth" to 1, e.g.

```
Sim.states.Facs{fac}.useCth   = 1;      % use of covariance of regression coefficients
```

Setting this field to 0 means that the covariance matrix is not used in simulation.

The use of covariance of regression coefficients increases noise variance due to the uncertainty of regression coefficients projected to the direction of the current regression vector. The increase can be rather high.

The simulator constructor sets "useCth" always to zero.

17.3 Markov switching among components

{markov_jumps

The real processes modelled by mixtures do not change active component at each time instant. In order to check how various algorithms cope with this situation, simulator was prepared changing active component according to a Markov chain. The *Markov transition probabilities table* must be specified and located in the global matrix "*CUMTAB*". An example folows.

Data are pre-allocated and mixture simulator with two components is defined:

```
ndat = 5000;                           % length of data
DATA = zeros(2,ndat);                  % data sample
```

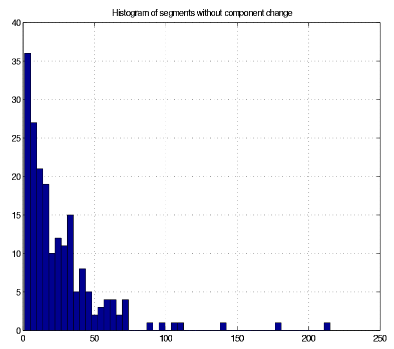

Figure 31: Histogram of segments without component change {zsimmark}

```
Com      = comarxls([1 2], [0;1]);         % static matrix ARX LS component
Com.cove = ltdl([0.1 0.01; 0.01 0.1]);    % point estimate of noise covariance
Com.Eth  = [-1; 0]; Coms{1}  = Com;       % 1st component
Com.Eth  = [1; 0];  Coms{2}  = Com;       % 2nd component
dfcs0    = [1 2];                          % simulator vector of degrees of freedom of components
Sim  = mixconst(Coms, dfcs0);              % build simulator
```

The transition probability table is defined in global matrix CUMTAB. The sum of the rows are equal to 1. *The CUMTAB is modified at the first simulation step.*

```
global CUMTAB                              % Markov change of components
CUMTAB = [0.95 0.05; 0.05 0.95];           % component transition table
```

The global variable "ACTIVE" is assigned to the current component in recursive simulation. It is used for recording trajectory of component changes in the recursive simulation that follows:

```
global ACTIVE                              % active component
for TIME = 1:ndat
    mixsimul(Sim);                         % recursive simulation
    actives(TIME) = ACTIVE;                % trajectory of use of component
end
```

The number of steps of stay in the components is

```
sum(actives==1)/ndat                       % No. of active component 1
ans =
    0.3808
sum(actives==2)/ndat                       % No. of active component 2
ans =
    0.6192
```

The values roughly correspond to the stationary probabilities with which components are active.

The histogram of segments without component change is in Fig. 31.

mixsim.tex, zsimmark.m by PN June 25, 2004

18 Estimation of structure of mixture factors

{structure}

The function "facstrid" 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 [?].

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 "facstrid" is called as:

`[MAPstr,lhs] = facstrid(Fac,Fac0,belief,nbest,nrep)` *factor structure estimation*

The function arguments are:

`MAPstr` is the estimated factor structure

`lhs` contains likelihood and structure of the most successful regression vectors structure, see the example below

`Fac` is the treated factor

`Fac0` is the corresponding initial factor

`belief` specifies user's *belief on a guess of richest structure* of the richest factor. The belief is a vector of the same length as the guess of the richest structure. Its elements specify that the corresponding items (the pairs of channel and delay) of the guess of the richest structureare in the estimated model structure:

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

`nbest` specifies the number of "best" structures maintained during repetitive estimations

`nrep` specifies number of repetitive search with random starts. Two runs are done automatically - start from empty and full regression vector structure. If results of the runs differ, a warning is displayed ("*not fully informative data*").

Example of structure estimation follows. We use simulated data from the Section 8.2 (dynamic mixture, 2 channels, 2 components).
The initial mixture consist of a single component. The richest factor structure is specified as:

```
maxstr   = [ones(1,6), 1+ones(1,7), 0
            1:6,       0:6,         1];
```

The richest mixture is build:

```
Fac      = facarxls(ychn, maxstr);      % initial dynamic factor
Fac.cove = 0.01;                        % point estimate of noise covariance
Fac.Cth  = eye(length(Fac.Cth));        % covariance of regression coefficients
Facs{1}  = arx2arx(Fac);
Fac      = facarxls(uchn, maxstr);      % initial noise factor
Fac.cove = 0.01;                        % setting factor fields
Fac.Cth  = eye(length(Fac.Cth));
Facs{2}  = arx2arx(Fac);

Mix0     = mixconst(Facs, [1 2], 1);    % initial mixture
frg      = defaults('frg');             % default forgetting rate
Mix      = mixestim(Mix0, frg, ndat);   % mixture estimation
```

The first factor structure estimation is done:

```
MAPstr   = facstrid(Mix.Facs{1}, ...    % factor structure estimation
           Mix0.Facs{1})
MAPstr =
       1     1     1     1     2
       1     2     3     4     3
```

The "true" factor structure is (see 8.2):

```
       1     1     1     1     2     2
       1     2     3     4     3     4
```

that is close to the estimated one.
The function "facstrid" offers the possibility of experimentation. For the first factor, we make 100 estimation runs and record 10 "best" (in the MAP sense) regression vectors structures found.

```
nrep  = 100;                          % number of optimization runs
nbest = 10;                           % number of regression vector structures
Fac   = Mix.Facs{1};                  % 1st factor
Fac0  = Mix0.Facs{1};                 % initial 1st factor

[MAPstr, lhs] = facstrid(Fac, Fac0, [], nbest, nrep)
MAPstr =
     1     1     1     1     2
     1     2     3     4     3
lhs =
    [1x10 double]    [14x10 double]
```

The "lh" output argument contains the best structures found and corresponding value of likelihood. It is converted to probabilities:

```
vlh = lhs{1};                        % value of log. likelihood
ilh = lhs{2};                        % the best MAP estimates of the structure
vlh = vlh-max(vlh);
vlh = exp(vlh);
vlh = vlh/sum(vlh);
```

The best regression vector structures are displayed (details of display are hidden):

```
 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  1  1  0  0  0  0  0  1  0  0  0  0    0.589
 1  1  1  1  0  0  0  1  0  1  0  0  0  0    0.131
 1  1  1  1  0  0  0  0  0  1  0  0  1  0   0.0632
 1  1  1  1  0  0  0  0  0  1  1  0  0  0   0.0579
 1  1  1  1  0  0  0  0  1  1  0  0  0  0   0.0385
 1  1  1  1  0  0  1  0  0  1  0  0  0  0   0.0359
 1  1  1  1  0  0  0  0  0  1  0  1  0  0   0.0346
 1  1  1  1  0  0  0  0  0  1  0  0  0  1   0.0198
 1  1  1  1  0  1  0  0  0  1  0  0  0  0    0.017
 1  1  1  1  0  0  0  1  0  1  0  0  1  0   0.0139
```

From the values displayed, it can be judged that the space of competing regression vector structures is relatively flat close to the MAP one. The "true" regression vector structure is the 5th one.

PN. mixstr.tex, zfacstr.m by PN June 25, 2004

19 Selected techniques

{cookbook}

19.1 Mixtools global matrices

{Gglobals}

19.2 Kullback - Leibler distance

The Kullback-Leibler distance is a prominent measure used in a range of Mixtools tasks.

19.2.1 Kullback - Leibler distance in parameter space

The Kulback-Leibler distance of a pair pdf is evaluated by "kldist". The pair must be of the same dimensions. The "kldist" evaluates the distance of

1. two factors

   ```
   kld = kldist(0,   Fac1, Fac2)          % distance of two factors
   ```

2. factor in a pair of mixtures

```
kld = kldist(fac, Mix1, Mix2)            % distance of factors "fac" in Mix1 and Mix2
```

The "fac" is factor number.

3. pair-wise distance of components of a mixture

   ```
   kld = kldist(Mix)                       % components of a mixture
   ```

   The distance is a non-symmetric matrix with zeros on diagonal. If the components differ in structure, the distance is infinite.

4. mixtures

   ```
   [d1, d2, d3, d4] = kldist(Mix1, Mix2)   % pair of mixtures
   ```

   The individual output elements are:

d1	overall distance
d2	distances of factors
d3	distances of components
d4	distances of the component weights

19.2.2 Kullback - Leibler distance in data space

The Kullback - Leibler distance of components in data space is calculated by "kldiscom":

```
[kldcom, ij] = kldiscom(Mix, ndat)
```

The "kldcom" is vector of distances - linearly organized lower triangle of distances. The array "ij" contains component numbers related to an element of "kldcom".

19.3 Setting "dbstop" in dialog

Support of setting of "dbstop" in dialog is to offer to the user a selection of lines in a function where it is effective to set "dbstop" marks. When the funtion is called, MATLAB enters the debugging mode and stops on the mark selected.

Each function can be prepared for setting "dbstop" in dialog. For the purpose, comment(s) of the form

```
%>any text
```

are placed inside the function body.

The interactive setting is done by the function "setdbg":

```
setdbg('name');                          % set dbstop in function 'name'
```

The function "setdbg" opens the function, finds the relevant comments, and displays them. One is selected in dialog by the user. The "dbstop" is set at the place where the comment appears. No action is done when only Enter is pressed, 0 means to clear all "dbstop".

If the function does not contain any relevant comment, the "dbstop" is set to the first function line.

Remember that most of functions are MEX-files and the M-versions must be copied into the working directory.

Example: the form of the interactive setting is displayed

```
setdbg('mixinit');
-1- beginning of each mixinit iteration
-2- mixsplit operation
-0- reset all debug stops
-------------------------
 Enter for no action

select >
```

20 Mixtools design base

20.1 Design base functions

Estimation related operations	
`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)`	quasi-Bayes iterative estimation with fixed variances
`Mix = mixestem(Mix0, ndat, niter)`	mixture estimation by EM-algorithm
`Mix = mixfrg(Mix ,frg)`	mixture forgetting
`[Mix0,handle] = ...`	
`mixflatv(Mix,niter,ndat,frg)`	mixture flattening with variable rate
`[Mix0,handle] = mixflatv(Mix,handle)` [a]	

[a] the first call in initialization, the second one in iterations

Auxiliary estimation operations	
`Mix = mixgmean(weights, Mix1, Mix2,...)`	geometric means of mixtures
`dvec = getdvect(Fac)`	get data or regression vector
`Mix = facupdt(Mix, facs, weights)`	update factors of a mixture
`lls = facdpred(Mix)`	compute trial factor predictions
`[s,s0] = mixdfms(Mix)`	sum degrees of freedom of the mixture
`Mix0 = mix2mix0(Mix)`	create initial mixture mimic to a mixture
`lls = loglik(LD,dfm,LD0, dfm0)`	compute increment of log-likelihood for an ARX factor
`Sim = sim2pdf(Sim, ndat)`	convert simulator to estimator

Prediction related operations	
`Facs = fac2marg(Facs, pchns) )`	convert factor into data-marginal factor
`Com = com2pro(Facs, pchns, cchns)`	convert ARX LS component to predictor
`[typ, ychns,...] = comunpk(...)`	get information about component

Preprocessing	
`Pre = preaux1(method, time, Pre)`	auxiliary function for data pre-processing

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)`	transforms mixture estimate Mix to the control form aMix

Kullback-Leibler distance

`dist = kldist(fac, Mix, Mix0)`	distance of a factor in parameter space
`dist = kldist(  0, Mix, Mix0)`	distance of all factors
`dist = kldist(Mix, Mix0)`	distance of all components
`[d1,d2,d3,d4] = kldist(Mix1, Mix2)`	distance 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)`	convert normal mixture into array of ARX components
`Facs = arxc2fac(Com)`	convert ARX component to ARX LS factors
`Com  = fac2arxc(Facs)`	convert ARX LS factors to ARX component

Conversions of L'DL decompositions

`V    = ld2v(LD)`	convert L'DL decomposition to original matrix
`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)`	convert ARX factor to least square representation
`LD = ls2ld(Eth,Cth,cove,dfm)`	convert Eth, Cth, cove, dfm into LD
`[Eth,Cth,cove]=ld2ls(LD,dfm,nychn)`	convert L'DL into Eth, Cth, cove

Subselection from an L'DL decomposition

`LD   = ld2ld(LD,str1,str2)`	reduce L'DL decomposition to get marginal parametric pdf [a]

Permutation of entries in L'DL decomposition – auxiliary function

`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)`	convert decomposition L'DL to U'DU[a]
`UD  = utdu(X)`	upper triangular decomposition U'DU of a symmetric matrix
`UD  = ld2ud(LD)`	convert decomposition L'DL to U'DU
`LD  = ud2ld(UD)`	convert decomposition U'DU to L'DL
`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
`[Eth, cove] = udform(Eth, cove)`	restore matrix factorized ARX component

[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)`	convert ARX LS to matrix factorized ARX component
`Com  = can2arxc(Can, n)`	[a] convert "Can" into matrix ARX LS component
`Can  = can2marg(Can)`	permute matrix factorized ARX component

[a]"n" is number of marginal channels

`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 a MATLAB array	
`mixdump(Mix, filename,...)`	dump MATLAB object to disk
`Mix = restore(filename,...)`	restore dumped MATLAB object

General purpose functions	
`val = defaults('item')`	get values from database of defaults
`e   = noise(etyp)`	generate a random number with a "etyp" distribution[a]
`val = gauss1(dvect,Eth,cove)`	value of one-dimensional Gaussian pdf
`val = gaussn(dvect,Eth,cove)`	value of Gaussian pdf
`setfig(n)`	set figures windows
`val = getflds(cell_vect, 'field')`	get fields from a cell-vector of structures
`val = betaln(p,q)`	logarithm of Euler's beta function
`fac = facsort(Facs)`	sort factors of a component

[a]generators have mean 0 and covariance 1 (with exception of Cauchy); the etyp is: 1 - Gaussian, 2 - uniform, 3 - lognormal, 4 - Cauchy

funlistb.tex by PN June 25, 2004

20.2 Alphabetic list of Mixtools functions

Alphabetic list of functions

`ACTIVE`	active component
`CUMTAB`	transition table of components
`Can`	component in matrix factorized ARX LS form
`Cans`	array of components in matrix factorized ARX LS form

`Com`	matrix ARX or ARX LS component
`Coms`	array of matrix ARX or ARX LS components
`Cth`	covariance of regression coefficients
`D`	diagonal part of L'DL decomposition of extended information matrix
`DATA`	data sample
`DEBUG`	global debugging flag
`Eth`	point estimate of regression coefficients
`Fac`	factor
`Facs`	array of factors
`L`	triangular part of L'DL decomposition
`LD`	L'DL decomposition of the extended information matrix
`MAPstr`	MAP estimate of the factor structure
`Mix`	mixture estimate
`Ndat`	specification for buffered processing
`Psi`	data vector
`Sim`	mixture simulator
`TIME`	processing time
`UDc`	cell vector of u'du decompositions of KLD kernels
`aMixc`	advised mixture of the type ARX LS + control states
`aMixu`	desired mixture of the type ARX LS + control states
`alphas`	normalized vector of degrees of freedom of components
`belief`	belief on a guess of richest structure
`cchns`	channels in condition
`chbelief`	belief on factors of a channel
`chis`	strategy of control design
`chn`	channel (data row)
`com`	component
`comlls`	component predictions
`coms`	array of components
`comwgs`	component weights
`cove`	point estimate of noise covariance
`dfcs`	vector of degrees of freedom of components
`dfcs0`	initial degrees of freedom of components
`dfm`	degrees of freedom of a factor
`fac`	position of a factor in an array of factors
`faclls`	virtual factor predictions
`facs`	list of factors
`facwgs`	factor weights
`frg`	forgetting rate
`frgd`	default forgetting rate
`irep`	iteration
`kc`	lift of quadratic forms
`kca`	average lift of quadratic forms kc
`kld`	Kullback-Leibler distance
`ll`	log of posterior likelihood on data: v-log-likelihood
`maxFac`	richest factor
`maxMix`	richest mixture
`maxerr`	maximum possible error
`maxstr`	guess of the richest structure
`maxtd`	maximum time delay of factors in a mixture
`mixll`	posterior data likelihood (mixture prediction)
`nPsi`	length of data vector
`nbest`	number of "best" MAP structures stored
`nchn`	number of modeled channels
`ncom`	number of components
`ndat`	length of data
`nfac`	number of active factors

`niter`	number of iterations
`npochn`	number of channels with o-innovations
`npsi`	length of regression vector
`nrep`	number of random starts
`nruns`	number of runs in iterative mixture estimation
`nsk`	extent of data grouping
`nychn`	number of modeled channels
`options`	computational options
`outs`	list of channels with innovations
`pMix`	mixture predictor
`pMixfix`	mixture prediction
`pchns`	predicted channels
`pdf`	probability density function
`pochn`	list of channels with o-innovations
`pre`	preprocessing requirements
`psi`	regressor vector
`psi0`	value of zero-delayed regressor
`rate`	mixture flattening rate
`relerr`	relative error
`seed`	seed of random generator
`sig`	standard deviation of output noise
`std`	standard deviation
`str`	structure of regression vector
`strc`	common control structure
`uchn`	list of channels with recognisable actions
`udca`	u'du decomposition of average KLD kernel in UDc
`ufc`	normalised vector qualifying components
`ychn`	modeled channel
`ychns`	modeled channels in component

20.3 List of recommended identifiers

{cryptony}

Cryptonyms

Data management	
`TIME`	processing time
`DATA`	data sample
`ndat`	length of data
`psi`	create regression vector
`Psi`	data vector
`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`	L'DL decomposition of the extended information matrix
`L`	triangular part of L'DL decomposition
`D`	diagonal part of L'DL decomposition of extended information matrix
`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`	initial 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 modeled 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`	value of zero-delayed regressor

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 recognisable 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`	log of posterior likelihood on data: v-log-likelihood
`niter`	number of iterations
`opt`	option
`options`	computational options
`seed`	seed of random generator
`uchn`	list of channels with recognisable actions
`sig`	standard deviation of output noise
`CUMTAB`	transition table of components
`ACTIVE`	active component

cryptony.tex by PN June 25, 2004

20.4 Alphabetic list of recommended identifiers

{cryptona}

Alphabetic list of cryptonyms

`ACTIVE`	active component
`CUMTAB`	transition table of components
`Can`	component in matrix factorized ARX LS form
`Cans`	array of components in matrix factorized ARX LS form
`Com`	matrix ARX or ARX LS component
`Coms`	array of matrix ARX or ARX LS components
`Cth`	covariance of regression coefficients
`D`	diagonal part of L'DL decomposition of extended information matrix
`DATA`	data sample
`DEBUG`	global debugging flag
`Eth`	point estimate of regression coefficients
`Fac`	factor
`Facs`	array of factors
`L`	triangular part of L'DL decomposition
`LD`	L'DL decomposition of the extended information matrix
`MAPstr`	MAP estimate of the factor structure
`Mix`	mixture estimate
`Ndat`	specification for buffered processing
`Psi`	data vector
`Sim`	mixture simulator
`TIME`	processing time
`UDc`	cell vector of u'du decompositions of KLD kernels
`V`	information matrix
`aMixc`	advised mixture of the type ARX LS + control states
`aMixu`	desired mixture of the type ARX LS + control states
`alphas`	normalized vector of degrees of freedom of components
`belief`	belief on a guess of richest structure
`cchns`	channels in condition
`chbelief`	belief on factors of a channel
`chn`	channel (data row)
`com`	component
`comlls`	component predictions
`coms`	array of components
`comwgs`	component weights
`cove`	point estimate of noise covariance
`dfcs`	vector of degrees of freedom of components

`dfcs0`	initial degrees of freedom of components
`dfm`	degrees of freedom of a factor
`fac`	position of a factor in an array of factors
`faclls`	virtual factor predictions
`facs`	list of factors
`facwgs`	factor weights
`frg`	forgetting rate
`frgd`	default forgetting rate
`irep`	iteration
`kc`	lift of quadratic forms
`kca`	average lift of quadratic forms kc
`kld`	Kullback-Leibler distance
`ll`	log of posterior likelihood on data: v-log-likelihood
`maxFac`	richest factor
`maxMix`	richest mixture
`maxerr`	maximum possible error
`maxstr`	guess of the richest structure
`maxtd`	maximum time delay of factors in a mixture
`mixll`	posterior data likelihood (mixture prediction)
`nPsi`	length of data vector
`nbest`	number of "best" MAP structures stored
`nchn`	number of modeled channels
`ncom`	number of components
`ndat`	length of data
`nfac`	number of active factors
`niter`	number of iterations
`npochn`	number of channels with o-innovations
`npsi`	length of regression vector
`nrep`	number of random starts
`nruns`	number of runs in iterative mixture estimation
`nsk`	extent of data grouping
`nychn`	number of modeled channels
`options`	computational options
`outs`	list of channels with innovations
`pMix`	mixture predictor
`pMixfix`	mixture prediction
`pchns`	predicted channels
`pdf`	probability density function
`pochn`	list of channels with o-innovations
`pre`	preprocessing requirements
`psi`	regressor vector
`psi0`	value of zero-delayed regressor
`rate`	mixture flattening rate
`relerr`	relative error
`seed`	seed of random generator
`sig`	standard deviation of output noise
`std`	standard deviation
`str`	structure of regression vector
`strc`	common control structure
`uchn`	list of channels with recognisable actions
`udca`	u'du decomposition of average KLD kernel in UDc
`ufc`	normalised vector qualifying components
`ychn`	modeled channel
`ychns`	modeled channels in component

21 Mixtools MEX and API functions

21.1 Mixtools MEX functions

The Mixtools toolbox contains more than hundred M-functions. Mostly, they are converted into MEX-functions. The MEX-functions can be called from any other MEX-function and from any stand-alone application program.

The MEX-functions are held in a library ”prodact.lib” Each MEX-function is stored in it using the pre-processor directive -DLIBRARY. The header file ”mexlib.h” contains functions prototypes and descriptive information.

The structure of MEX-functions is docummented by an example of a (fictive) function ”mexfun”:

```
// Sample function mexfun.c

#include <math.h>
#include "mex.h"              // MATLAB definitions
#include "mexlib.h"           // Mixtools definitions

#ifndef LIBRARY
void mexFunction( ... )       // translated to "mexfun.dll" under MATLAB
{
   void mexfun( ... );        // call MEX-function in library
}
#else                         // compiled if LIBRARY is defined
void mexfun( ... )            // maintained in library as "mexfun"
{
   // MEX - file interface
   ...
}
mxArray * mexfun1(...)        // stored in prodact.lib
{                             // prototypes are transferred to mexlib.h
}
#endif                        // end processing of library code
```

Conversion of M-functions to MEX-functions and their debugging is a routine work based on MATLAB API (see [?]). An alternative way is use of MATLAB Compiler, product of MathWorks. This possibility has several disadvantage and has not been selected for ProDaCTools project.

21.1.1 Getting data vector

The following global variables are defined in ”mexlib.h”:

```
double *DATA;                 // pointer to data sample
int    M_DATA;                // number of data rows (channels)
int    N_DATA;                // length of data sample
double *TIME;                 // pointer to current time
double PSI[1];                // auxiliary array used to store data vector
```

To get the pointers and dimensions, ”bldlmex.c” is included in any function that makes recursive processing. To get value of data vector use the function:

```
  void getrgr(int ychn, double *str, int len);
```

The meaning of the arguments is obvious, the data vector is build in the global array ”PSI”.

written by PN June 25, 2004

22 Mixtools Application Program Interface

Stand-alone application programs can

- load previously dumped MATLAB arrays
- call Mixtools functions
- dump results of computation for processing under MATLAB.

22.0.2 Communication with MATLAB

The communication between the stand-alone application program and MATLAB is done by binary files. They contain unloaded MATLAB double arrays as well as some descriptive information. The MATLAB "iofun" functions ("fopen", "fread", "fwrite") are use to write/read the MATLAB arrays. Those functions offer a broad selection of formats of the dump files that cover many relevant platforms. The example of dump and restore under MATLAB:

```
...
filename      = 'mixdumped';
machineFormat = 'native';
precision     = 'double';
mixdump(Mix, filename, machineFormat, precision);
...
Mix = mixrest(filename, machineFormat, precision);
Mix = rename(Mix,Mix0);
```

See help on "fopen" function for meaning of the arguments. The selection shown is the functions default. The function "rename" is called to assign names to the structure fields - names of fields are not dumped. The "Mix0" must be structure build under MATLAB.

The same functions are available in MAPI (dump.c):

```
mxArray *mixload (const char *filename);
void mixsave (const mxArray *mix, const char *filename);
```

22.0.3 Programming

The MATLAB MEX-functions are written using functions of MATLAB API (Application Program Interface) [?].

The API functions are substituted by independent API encoded in ProDAcTools. The definitions are held in "mex.h".

The library of MEX and API functions is "api.lib".

Not all MATLAB API functions are implemented. The list of functions is held in "mex.h", The MATLAB API equivalents implemented (see [?] for meaning and prototypes):

mxCalloc	mxGetPr
mxCreateCell	mxGetScalar
mxCreateCellMatrix	mxGetString
mxCreateDoubleMatrix	mxIsCell
mxCreateStruct	mxIsChar
mxCreateStructMatrix	mxIsComplex
mxCreateString	mxIsDouble
mxDuplicateArray	mxIsEmpty
mxGetCell	mxIsNumeric
mxGetFieldPr	mxIsStruct
mxGetM	mxSetCell
mxGetN	mxSetM
mxGetNumberOfFields	mxSetN
mxGetNumberOfElements	mxSetPr
mxGetPi(X)	

22.0.4 Memory management

The standard memory management - "malloc" (connected with "mxAlloc", "mxCalloc") is possible but not recommended. MEX-functions are written by different persons and quality of memory freeing can hardly be traced.

Instead, a huge global array is used for allocations:

```
// in mex.h
#define WORKSPACE_LENGTH (1000000)// heap allocation
char   workspace[WORKSPACE_LENGTH];
int    wsp = 0;                    // pointer to free workspace
```

The function that makes allocations in the workspace is

```
  char* aloc(int n); // allocates n Bytes on double word boundary
```

The freeing of memory is simple - at the beginning of a function, the "wsp" is recorded and after all allocations it is returned to the initial value (permanent definition of course remain). No other of freeing of memory is required in the function body.

22.0.5 Getting data vector

The mechanism of getting data vector is similar as the one used for MEXes. The global variables used are:

```
mxArray*XDATA;                // pointer to data as mxArray *
mxArray*XTIME;                // pointer to time as mxArray *
```

The variables should be initialized by user's code. The code "bldlmex.c" differs from MEXes.

22.0.6 In-place processing

Unlike in MEXes, API functions can be used for in-place processing (changing input arrays). The following example shows in-place processing of a mixture:

```
#ifndef MATLAB_MEX_FILE       // this construct makes possible
if (no)
   Mix = in[0];               // to use "in-place" processing
else
   Mix = mxDuplicateArray(in[0]);  // standard MEX processing
#else                         // outside MATLAB (example)
   Mix = mxDuplicateArray(in[0]);  // standard MEX processing
#endif
```

22.0.7 Example of stand-alone program

MATLAB processing and equivalent stand-alone program are discussed. We have the MATLAB script function:

```
ychns    = [1 2];                     % output channels
str      = [0;1];                     % common static structure

Com      = comarxls(ychns, str);      % initial ARX LS component
Com.Eth  = [3;0];  Coms{1} = Com;     % 1st component
Com.Eth  = [-3;0]; Coms{2} = Com;     % 2nd component

dfcs = [100, 150];                    % initial dfcs
Mix0 = mixconst(Coms, dfcs);          % build initial mixture
Mix0 = mix2mix(Mix0, 21);             % convert it into convenient form

frg  = 0.99999;                       % default forgetting rate
ndat = 1000;                          % size of data sample
```

```
niter= 30;                            % number of iterations

Mix = mixestqb(Mix0,frg,ndat,niter);   % quasi-Bayes repeated estimation
```

The equivalent stand-alone program:

```
#include <math.h>
#include "mex.h"
#include "mexlib.h"
#include <stdio.h>

void main(void)
{  mxArray *ychns, *str, *ndat, *frg, *niter,
           *Com, *Coms, *Eth, *dfcs, *Mix0;
   double  *p;
   mxArray *out[1];
   const mxArray *in[4];

   XDATA = mixload("data");              // load data
   XTIME = mxCreateDoubleMatrix(1,1,0);  // create TIME variable
#include "bldlmex.c"                     // required for data processing

// ychns    = [1 2];                      % output channels
// str      = [0;1];                      % common static structure

   ychns = mxCreateDoubleMatrix(1,2,0);
   p     = mxGetPr(ychns);
   p[0]  = 1; p[1] = 2;
   str   = mxCreateDoubleMatrix(2,1,0);
   p     = mxGetPr(str);
   p[0]  = 0; p[1] = 1;

// Com      = comarxls(ychns, str);       % initial ARX LS component
   in[0] = ychns; in[1] = str;
   comarxls(1,out,2,in);
   Com   = out[0];

// Com.Eth  = [3;0];  Coms{1} = Com;      % 1st component
   Eth   = mxCreateDoubleMatrix(1,2,0);
   p     = mxGetPr(Eth);
   p[0]  = 3;  p[1] = 0;
   copyfield(Com, LS_Eth, Eth);
   Coms  = mxCreateCellMatrix(1, 2);
   mxSetCell(Coms,0,Com);

// Com.Eth  = [-3;0]; Coms{2} = Com;      % 2nd component
   p[0]  = -3;
   Com   = mxDuplicateArray(Com);
   copyfield(Com, LS_Eth, Eth);
   mxSetCell(Coms,1,Com);

// dfcs = [100, 150];                     % initial dfcs
   dfcs = mxCreateDoubleMatrix(1,2,0);
   p    = mxGetPr(dfcs);
   p[0] = 100; p[1] = 150;

// Mix0 = mixconst(Coms, dfcs);           % build initial mixture
   in[0] = Coms; in[1] = dfcs;
   mixconst(1,out,2,in);
   Mix0  = out[0];

// Mix0 = mix2mix(Mix0, 21);              % convert into convenient form
   in[0] = Mix0; in[1] = mxCreateScalarDouble((double)21);
   mix2mix(1, out, 2, in);
   Mix0 = out[0];

// frg  = 0.99999;                        % default forgetting rate
// ndat = 1000;                           % size of data sample
// niter= 30;                             % number of iterations
   frg   = mxCreateScalarDouble(0.99999);
   ndat  = mxCreateScalarDouble(1000.0);
   niter = mxCreateScalarDouble(30);

// Mix = mixestqb(Mix0,frg,ndat,niter);   % quasi-Bayes repeated estimation
   in[0] = Mix0; in[1] = frg; in[2] = ndat; in[3] = niter;
   mixestqb(1, out, 4, in);

// results are saved in "mixture"
   mixsave(out[0], "mixture");

}
```

api.tex by PN June 25, 2004

23 Termbase

Mixtools Guide, case studies and any related texts must use unique technical terms. It is impossible without a computer aided support.

This support is based on a database of technical terms and functions that work with it. The database is held in an (ascii) file "termbase" in the Mixtools "termbase" directory.

The "termbase" contains information about all Mixtools functions and about the recommended identifiers referred to as cryptonyms here.

The "termbase" consists of "records". The records are distinguished by "keywords" that are names of Mixtools functions and cryptonyms.

Each record begins with the keyword (from 1 to 8 columns) and a "keyword description" begins from 10th column.

An example of two termbase records:

```
mix2mix  convert mixture to a specified form
Eth      point estimate of regression coefficients
```

Note: with the records, lines that begin with the character ' ' or the character '#' may freely be placed. Those lines are treated as termbase comments.

The "termbase" file is converted to a "termbase.mat" that contains information needed for direct access to individual records. It is done by the function "tbbldl":

`tbbldl` % *convert termbase into internal form*

This function must be called whenever the "termbase" file changes.

23.1 Use of termbase in ascii texts

The keyword descriptions can be copied into any ascii text by specifying the keyword only.

This solves the function "tbedit". It converts the input file and displayes it on screen. the input file is not changed and the output file that contains the edited text:

`tbedit('input file');` % *edit ascii text*

The input file can contain the termbase keyword proceeded by the character '\'. This is substituted by the keyword description in the output file.
Example follows.

```
infile  = 'infile';

% create text file
in = fopen(infile, 'wt');
line1 = 'function "mixestim" makes \mixestim . The argents are:';
line2 = 'Mix (\Mix), frg (\frg) and ndat (\ndat)';

fprintf(in, '%s\n', line1);
fprintf(in, '%s\n', line2);
fclose(in);

tbedit(infile);                       % edit the text file

function "mixestim" makes quasi-Bayes mixture estimation . The argents are:
Mix (mixture estimate), frg (forgetting rate) and ndat (length of data)
```

23.2 Use of termbase in Latex texts

The "termbase" is converted to "termbase.tex" by the function

```
  tb2tex                              % build termbase.tex
```

The "termbase.tex" contains Latex "newcommands" for all termbase records. Example of the "newcommand" corresponding to the keyword "Eth":

```
\newcommand{\Eth}[0]{point estimate of regression coefficients}
```

The "newcommands" correspond to the termbase keywords with two exceptions listed below.

1. the keywords that contain a digit (impossible in Latex) are changed - the digit is converted into upper case letter:

keyword	newcommand
2	T
0	I
1	J

   For example, the "newcommand" corresponding to the keywords "mix2mix" is:

```
\newcommand{\mixTmix}[0]{convert mixture to a specified form}
```

2. Several keywords interfere with existing Latex commands and are changed:

keyword	newcommand
psi	psix
Psi	Psix
LD	LDt
L	Lt
D	Dt
ll	llx

A Example of channel descriptions

{desc}

The example contains example of channel descriptions.

```
% ChanDesKOR channels description of KOR data
%
% Design     : T.V. Guy
% Updated    : September, 2002
% Project    : ProDaCTools, IST-1999-12058

% References :

  prodini
  echo off

% file name
  filename = 'ChnDes';                            % filename where channel descriptions be saved

% channels description
  chns = 1:10;                                    % number of channels
  Chns = chnconst(chns);                          % build channels description
  Chns = chnset(Chns, [], 'oitem', 0);            %
  Chns = chnset(Chns, [], 'raction', 0);          %

%%%%%%%%%%%%%%% set channels with o-innovations %%%%%%%%%%%%%%%%%
  field = 'oitem';
  chns = [8 10];                                      % channel numbers
  Chns = chnset(Chns,chns,field,1);

%%%%%%%%%%%%%%% set channels with recognisable actions %%%%%%%%%%
  field = 'raction';
  chns = [1 2 4 5];                                 % channel numbers
  Chns = chnset(Chns,chns,field,1);

%%%%%%%%%%%%%%%% set names of the channels %%%%%%%%%%%%%%%%%%%%%
  field = 'name';

  chn = 1;
  name = 'InStripTension';
  Chns = chnset(Chns,chn,field,num2cell(name,2));  % set channel description

  chn = 2;
  name = 'OutStripTension';
  Chns = chnset(Chns,chn,field,num2cell(name,2));  % set channel description

  chn = 3;
  name = 'StripSpeedRatio';
  Chns = chnset(Chns,chn,field,num2cell(name,2));  % set channel description

  chn = 4;
  name = 'InStripSpeed';
  Chns = chnset(Chns,chn,field,num2cell(name,2));  % set channel description

  chn = 5;
  name = 'OutStripSpeed';
  Chns = chnset(Chns,chn,field,num2cell(name,2));  % set channel description
  chn = 6;
  name = 'InCoilerCurrent';
  Chns = chnset(Chns,chn,field,num2cell(name,2));  % set channel description

  chn = 7;
  name = 'OutCoilerCurrent';
  Chns = chnset(Chns,chn,field,num2cell(name,2));  % set channel description

  chn = 8;
  name = 'MillDriveCurrent';
  Chns = chnset(Chns,chn,field,num2cell(name,2));  % set channel description

  chn = 9;
  name = 'InThicknessDeviation';
  Chns = chnset(Chns,chn,field,num2cell(name,2));  % set channel description

  chn = 10;
  name = 'OutThicknessDeviation';
  Chns = chnset(Chns,chn,field,num2cell(name,2));  % set channel description

%%%%%%%%%%%%%%%%%%%%% set physical range <min;max> %%%%%%%%%%%%%%%%%%%%%%%
  field = 'prange';

  chn = 1;
  value = [0;50];                        %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 2;
  value = [0;50];                        %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 3;
  value = [0.5;0.7];                     %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 4;
  value =[0.1;0.3];                      %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 5;
  value = [0.2;0.5];                     %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 6;
  value = [30;180];                      %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 7;
  value = [30;180];                      %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 8;
  value = [0;200];                       %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 9;
  value = [-50;50];                      %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 10;
  value = [-5;5];                       %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  %%%%%%%%%%%%%%%%%%%%% set desired range <min;max> %%%%%%%%%%%%%%%%%%%%%%%
  field = 'drange';

  chn = 8;
  value = [30;180];                     %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 10;
  value = [-2;2];                       %
  Chns = chnset(Chns,chn,field,value);  % set channel description

%%%%%%%%%%%%%%%%%%%%% set priorities <0;1> %%%%%%%%%%%%%%%%%%%%%
  field = 'prty';

  chn = 1;
  value = 1;                            %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 2;
  value =1;                             %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 3;
  value = 0.5;                          %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 4;
  value = 1;                            %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 5;
  value = 1;                            %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 6;
  value = 0.5;                          %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 7;
  value = 0.5;                          %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 8;
  value = 0.9;                          %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 9;
  value = 0.5;                          %
  Chns = chnset(Chns,chn,field,value);  % set channel description

  chn = 10;
  value = 1;                            %
  Chns = chnset(Chns,chn,field,value);  % set channel description


% saving the channel description in the file filename
  eval(['save ',filename,' Chns'])
```

chandes, TG June 25, 2004

B Example of identification

{ident}

The example of initialization, estimation and prediction of KOR data.

```
% Design     : I.Nagy
% Updated    : August, 2002
% Project    : ProDaCTools, IST-1999-12058
% Calls      :

% References :

  prodini
  echo off
  DEBUG=2;
  global BREAKPOINT;

% file with raw data
  file_dat = 'data';

% file with channel descriptions
  file_des = 'ChnDes';
  eval(['load ', file_des])          % load the data file with channel descriptions

% basic file name for the current experiment
  filN = 'exp000';  % basic name of the experiment

% basic filename for Tex documentation (can be used for parameter setting)
  babble = 'e000';

% options controlling the tasks performed
  iest = 1;         % run with estimation? ( 0-no, 1-yes )
  iplo = 0;         % plot clusters?   ( 0-no, 1-yes )
  ival = 0;         % validation test?  ( 0-no, 1-yes )
  itex = 0;         % conver results to LaTeX form?  ( 0-no, 1-yes )

  istack =0;        % use stacking?  ( 0-no, 1-yes )
  nsk   = 2;        % number of data for stacking

  ibrk = 0;         % run after breakpoint?  ( 0-no, 1-yes )
  nbrk = 1;         % number of steps of init after breaking
  BREAKPOINT = [filN,'brk'];  % breakpoint name

% channels
  chns = getflds(Chns, 'chn');  % list of channels from channel description
  nchn = length(Chns);          % total number of channels
  nchn = 1:10;                  % list of channels that will be proceeded

% data
```

```
  nd   = 5000;     % length of the data be used
  ndat = 4000;     % length of data be used for estimation
  nval = 1000;     % length of data be used for validation (after ndat)

% filtration
  nfi   = 3;        % number of data for averaging
  if istack
     pre = {'olymedian', {'level', 1}, 'scale', [], 'lsfit0', nfi, 'group', nsk};
                    % options for pre-filtration
  else
      nsk = 1;
      pre = {'olymedian', {'level', 1}, 'scale', [], 'lsfit0', nfi};
                    % options for pre-filtration
  end;
% initialization + estimation options
  ord   = 1;        % model order
  ncom  = 1;        % numb. of components (0 = automatic init)
  frg   = 1;        % forgetting for the first part
  niter = 5;        % number of iterations for initialization
  nide  = 15;       % number of iterations for identification
  opt   = ' ';      % options for init
  Cth   = 1e3;      % initial Cth
  cove  = 1e-2;     % initial cove
  pisch = .1;       % pischvortz constant for prior df(m,cs)

  dfcs  = pisch*ndat/(ncom+1);      % initial dfcs
  dfm   = dfcs/length(chns)/nsk;    % initial dfm

% validation (prediction) options
  pchn  = 1:10;     % channels be used for validation
  pchns = 1:length(chns)*nsk; % predicted channels
  cchns = [];       % channels in condition
  npred = 15;       % steps of prediction (1 - default)

  eval('babble');                   % currently changed parametr

% changes due to preprocessing
 nchn = length(nchn)*nsk;        % ..in the number of channels
 ychns = 1:nchn;
 ndat = fix(ndat/(nfi*nsk));     % ..in the number of data used for identification
 nval = fix(nval/(nfi*nsk));     % ..in the number of data used for validation

% ****************** DATA LOAD ********************
eval(['load ',name]);
DATA = Dat(chns, 1:nd);     % Dat - common name of strored data
clear Dat;

% **************** PREPROCESSING ******************
pre  = preproc(pre);                 % preprocessing
Data=DATA;                           % store original data for prediction

% ************** INITIAL COMPONENT ****************
if iest
   if ncom<=0,
      ini=1; ncom=1;
   else ini=0;end
   str = genstr(ord, nchn);
   dfcs = dfcs*ones(1,ncom);
   Mix0 = genmixe(ncom, ychns, str, ndat, Cth, cove, dfm, dfcs);% construct initial mixture Mix0

% *************** IDENTIFICATION ******************
% batch processing
   file = fopen([filN,'.dat'], 'wb');
   fwrite(file, DATA(:,1:ndat), 'double'); % data for batch processing
   fclose(file); Ndat = {[filN,'.dat'], nchn};
   DATA = zeros(nchn, 1000);     % length of the batch

   if ini
      if ~ibrk,  MixIn = mixinit(Mix0, frg, Ndat, niter, opt);
      else       MixIn = mixinit(BREAKPOINT, nbrk);       end
      MixFl = mixflat(MixIn);
   else
      MixFl = Mix0;
   end;
   Mix  = mixest(MixFl, frg, Ndat, nide);
else
eval(['load ',filN,' Mix']); end

% **************** PREDICTION *********************
DATA = Data;                        % all data without batch proc.
pMix = mix2pro(Mix, pchns, cchns); % conditional mixture
time = (ndat+npred-1):(ndat+nval-1);   % time instants of predictions
yp=[];
for tt=ndat:(ndat+nval-npred)
    if fix(TIME/100)*100==TIME, fprintf('.'); end
    ttp = tt+(0:npred-1);
    for TIME=ttp
        [Et,co,df,wt] = profix(pMix); % factor prediction
        th = GetTh(Et);
        if 1 ypt = th*df'; else ypt = th*wt'; end
        DATA(pchns,TIME) = ypt;       % replace unavailable data by prediction
    end
    yp=[yp ypt];
    DATA=Data;                        % put original data back to DATA matrix
end;
clear Data; fprintf('\n');

% **************** VALIDATION *********************
dd   = DATA(pchn,time);        % predicted data
yp   = yp(pchn,:);
ep   = dd-yp;          % prediction errors
dfcs = Mix.dfcs/sum(Mix.dfcs)  % dfcs of Mix
if ival==1
  [histEp, binsEp] = hist(ep,20);     % histogram of ep
  hist_bin_Ep = [histEp; binsEp]
  testAlt   = altval(Mix)             % mixll + alt. forgetting
  testStab  = stabmix(Mix)            % stability test (0 stable)
  testNois  = facnois(Mix);           % factor noises
  testNois_dfcs = [testNois ones(length(dfcs),1)*NaN dfcs']
  [testCep, sumCep] = wtest(dd,yp)    % correlations od delayed ep
  disp('<-:-:=::=:-:->')
end
mixll     = Mix.states.mixll        % v-likelihood of Mix
testSE    = sterr(dd,yp)            % var(ep)/std(data)
```

```
% ************** STORING RESULTS *******************
save filN filN; eval(['save ',filN]);
```

ident, TG June 25, 2004

C Example of design

{desi}

The example of simultaneous design for the KOR data is presented.

```
% Example of simultaneous design
% Channels descriptions Chns are defined in ChanDesKOR.m
%       Mix  : identified mixture loaded from the file 'file_ide'
%       Sim  : mixture used for simulation in the closed loop;
%              loaded from  the file 'file_sim'
%       Mixu : target mixture created using channels descriptions Chns by target.m
%
% Design     : T. V. Guy
% Updated    : October, 2002
% Project    : ProDaCTools, IST-1999-12058
% Calls      : inisyn, ufcgen, aloptim, mix2pro, algen, mixcopy, profix

% References :

  prodini
  echo off

% general options
  ChanDesKOR                % creates channel description of KOR data

  file_dat = 'MM_PallN';    % file with data
  file_ide = 'kor2_21';     % file with identified mixture
  file_sim = 'kor2_22';     % file with mixture for simulation

  niter = 1000;             % number of time iterations
  nstep = [100;1];          % horizon for evaluation of KLD


  %***************** DATA LOAD *****************************
  load(file_sim,'Mix');     % load mixture used for simulation
  Sim = Mix;                % Sim - mixture used for system simulation

  load(file_ide,'Mix,pre'); % load identified mixture Mix and
                            % used preprocessing options pre
  load(file_dat,'Dat');     % load data
  DATA = Dat;               % global DATA

  %************************** PREPARATORY STAGE ***********************************
  Chns = scaleDescription(Chns, pre);      % scaling according to preprocessing options
  [Mixu,Chns,ychns] = target(Chns);        % build target mixture  Mixu
                                           % (in accord. with channel description Chns)

  chns = length(Chns);                     % number of channels
  ncom = size(Mix.coms,1);                 % number of components in identified mixture

  % ************************** INITIALISATION ************************************
  [aMix, aMixu] = inisyn(Mix, Mixu, Chns); % initialisation for advisory design
```

```
 ufc = ufcgen(aMix, aMixu);              % automatic generation of user preferences
                                         % according to stability test

 % ************************** OPTIMISATION ***************************************
 aMix = soptim(aMix, aMixu, ufc, nstep);  % perform simultaneous  design (optimisation)
 pMix  = mix2pro(aMix);                  % build predictor pMix
 pred = zeros(chns, niter);              % prediction

 % get indexes of channels with recognisable actions
 [uchn,indx] = intersect(getflds(Chns,'chn'),aMix.states.uchn);

 start_time = max(Sim.states.maxtd, Mix.states.maxtd); % start time for the simulation

 for TIME = start_time+1:niter               % time loop
     mixsimul(Sim);                          % simulation of new DATA item
     aMix = algen(aMix,ufc);                 % generating an advice
     pMix  = mixcopy(aMix, pMix);            % copy advice to predictor
     [Eth, coves, alpha] = profix(pMix);     % build prediction
     [Eth1, coves1,alpha1] = profix(pMix,[],pre);% build non-scaled prediction
     pred1 = zeros(chns,1);                      % array to save non-scaled prediction
     al = aMix.dfcs;                             % probabilistic weights
     for com = 1:size(Eth,2)                     % computation of the prediction
         pred(:,TIME) = pred(:,TIME)+Eth{com}*al(com); % scaled
         pred1 = pred1+Eth1{com}*al(com);              % non-scaled
     end;
     DATA(indx,TIME) = pred(indx,TIME);     % save the predicted recognisable actions
                                            % to use them in the new simulation step
     crit(TIME) = criter(Mixu,TIME,TIME-10);% criterion computation
end
```

Index