Akademie věd České republiky
Ústav teorie informace a automatizace

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

RESEARCH REPORT

Josef Andrýsek

Initiation Options in Estimation of Dynamic Mixtures

No. 2030 June 2001

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

Acknowledgement: This research has been partially supported by EC, project IST-1999-12058 and by GAČR, project 102/99/1564

Contents

1 Introduction **7**

2 Basic notions and notations **9**

3 Mixture identification **11**
- 3.1 Parameter estimation . . . 11
- 3.2 Estimation of factor structure . . . 11
- 3.3 Constructing of prior estimate . . . 11
  - 3.3.1 Iterative construction, flattening mapping . . . 11
  - 3.3.2 General description of branch and bound techniques . . . 13
  - 3.3.3 Branching by factor splitting . . . 13

4 Model description and software representation **15**

5 Mixinit **17**
- 5.1 Parameter overview . . . 17
- 5.2 Description . . . 18
  - 5.2.1 General scheme . . . 19
  - 5.2.2 decide . . . 19
  - 5.2.3 merging1 . . . 19
  - 5.2.4 canceling1 . . . 20
  - 5.2.5 mixsplit . . . 20
  - 5.2.6 merging . . . 20
  - 5.2.7 canceling . . . 21
  - 5.2.8 last merging . . . 21
  - 5.2.9 final operations . . . 22

6 Example of running mixinit **23**
- 6.1 Processing of mixinit . . . 23

7 Selection of processing options **29**
- 7.1 Number of iterations steps - niter . . . 30
  - 7.1.1 12 static components . . . 30
  - 7.1.2 8 static components . . . 31
  - 7.1.3 3 static components . . . 32
  - 7.1.4 Real data . . . 33
  - 7.1.5 Conclusion . . . 33
- 7.2 Number of iterations in estimation -n . . . 33
  - 7.2.1 Conclusion . . . 35
- 7.3 Dependency of computing time on the dimension of data . . . 35
  - 7.3.1 Sample size `ndat` . . . 35
  - 7.3.2 Dependence on the number of channels - `dim` . . . 36
- 7.4 Boolean options . . . 38

7.4.1 a,g,d 38
7.5 Shrinking of noise covariance estimate `-xcove` 40
7.5.1 7 static components 40
7.5.2 20 static components 41
7.5.3 4 static components 41
7.5.4 5 channels, 17 static components 41
7.5.5 real data 1 41
7.5.6 real data 2 41
7.5.7 7 static components 42

8 Software aspects **43**
8.1 MEX files 43
8.2 Library, calling functions 47
8.3 stand-alone application and memory management 48

9 Conclusions **51**

Chapter 1

Introduction

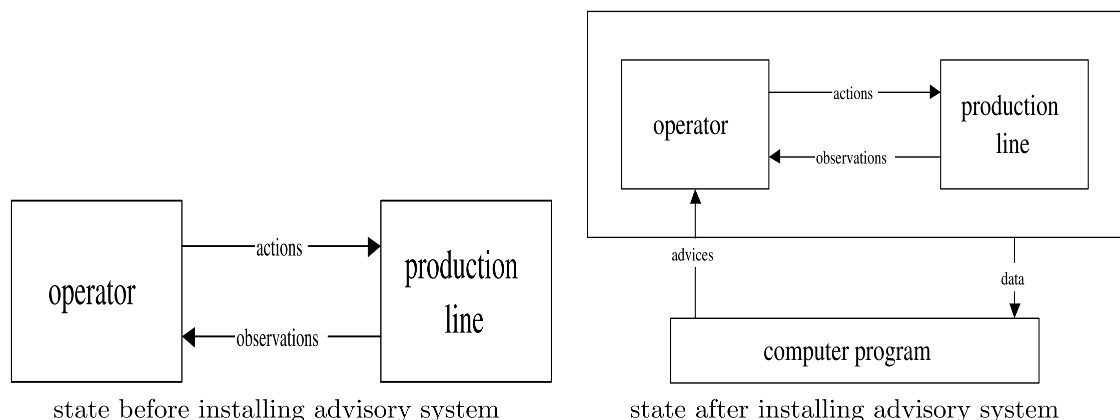
The contemporary computers are used in almost all branches of human activities. The automation in production is especially profitable. It improves the quality of products and it also decreases the cost. But there are a lot of cases, where the automation can not be done. The reason is in complexity of some problems. If we created the adequate physical model close to the reality, neither the fastest supercomputers are able to finish the processing in a reasonable time.

In real manufactories, which are not fully automated, mostly a human(operator) exist, who makes the important decisions and controls the production. It is clear, that the quality of products strongly depends on the experience of the operator and on his mental and physical state. In [3] the methods for elimination of this dependency are discussed. The approach is based on the idea, that the experience of the operator reflects in data measured on the system. Thus it remains "only" to extract the experience from the data. This solution is cheap to install because it does not requires any changes of the current systems. In many cases, the manufactories products large amounts of "outlet" data, which are just stored and not economised. Thought of the advisory system is to be fed just with this information.

The process of extracting information from data is very complex. It can be done in a various manner. Due to the immensity of data, some special presumption about distribution form of the data must be accepted to cope with them.

Project [3], to wich this work contributes, deals with finite mixture models. Details will be explained in chapter 4. This paper inspects partial but important problem of choosing feasible variants of algorithm to initialize mixture estimation.

Changing of the system structure



state before installing advisory system   state after installing advisory system

Chapter 2

Basic notions and notations

x^* *denotes the range of the quantity* x.

$\mathring{x}$ *denotes cardinality of the set* x^*, i.e. x^* is equivalent to $\{1, 2, \cdots, \mathring{x}\}$.

x_t *is the quantity* x *at the discrete time labelled by* $t \in t^* \equiv \{1, \ldots, \mathring{t}\}$.

$\mathring{t} \leq \infty$ *is called time horizon.*

$x_{i;t}$ *is an* ith *entry of the array* x *at time* t. The semicolon in the subscript indicates that the symbol following it has the meaning of a time index.

$x(k_l)$ *denotes the sequence between time moments* $k \leq l$, i.e. $x(k_l) \equiv x_k, \ldots, x_l$.

$x(t) \equiv x(1_t)$.

d *means array of data records measured on the considered system.*

Θ *the unknown parameter, a finite dimensional vector*

f *probability density function (pdf).* The meaning of the pdf is given through its argument:

- $f(\Theta)$- prior pdf. This pdf includes information about the unknown parameter Θ, which is known to the statistician, before he starts the measuring.
- other pdfs will be defined directly in the text.

$\langle x, y \rangle$ - scalar product of vectors x, y.

we - it means we statisticians.

We need the notion of *Kullback-Leibler distance* that measures well proximity of a pair of pdfs.

Agreement 2.0.1 (Kullback-Leibler distance) *Let* f, g *be a pair of pdfs acting on a common set* x^*. *Then,*

Kullback-Leibler distance $\mathcal{D}(f||g)$ *is defined by the formula*

$$\mathcal{D}(f||g) \equiv \int_{x^*} f(x) \ln \left(\frac{f(x)}{g(x)} \right) dx. \tag{2.1}$$

For conciseness, the Kullback-Leibler distance is referred to as KL distance.

Chapter 3

Mixture identification

Identification is the most important subtask in construction of an advisory system. Here, we discuss it in the extent needed for this work. For details, we refer to [3].

3.1 Parameter estimation

The parameter estimation is an important subtask in mixture identification. Unfortunately the mixture is a sum of functions. The standard Bayesian approach can be used theoretically only. No existing method for computing integrals works in dimensions we are considering. Only solution is using aproximate estimation. Project [3] deals with *quasi-Bayes estimation* algorithm and *EM estimation* algorithm. The results of quasi-Bayes estimation depend on the order in which data are processed. Experiments have shown that this dependence may be significant. EM algorithm avoids this drawback but its orientation on point estimation prevents us to combine it with advantageous Bayesian framework. For instance, the construction of the prior pdf by splitting, (see Section 3.3.3) cannot be exploited. In project [3] a novel batch quasi-Bayes estimation algorithm was created, that is processing-order independent. All these algorithm are iterative ones. The number of iterations can be set as a parameter of the algorithm. These methods gives as by-product value of the pdf $f(d(\mathring{t}))$

The mentioned algorithms are relatively strong. But they estimates only the parameters. Before we can use them, we need the prior pdf, which catches up the mixture structure i.e. the number of components, structure of components and structure of factors.

3.2 Estimation of factor structure

The next significant subtask is estimation of factor structure. The well determinated structure is a headstone for parameter estimation. In [3] the iterative algorithm for structure estimation is designed. Against, the number of iterations can be set as a parameter.

3.3 Constructing of prior estimate

The most important subtask in mixture identification is constructing of a prior estimate. Its quality influences the overall result in extreme way. In [3], a systematic way of exploiting physical knowledge is outlined. Here, the data based constructions are sketched.

3.3.1 Iterative construction, flattening mapping

We start from basic bayes rule:$f(\Theta|d(\mathring{t})) \propto f(d(\mathring{t})|\Theta)f(\Theta)$ with a very flat prior $f(\Theta)$. We can use the resulting posterior pdf $f(\Theta|d(\mathring{t}))$ as a new prior guess and use the bayes rule again. After n th repetition we get:

$$f_n(\Theta|d(\mathring{t})) \propto f(d(\mathring{t})|\Theta)f_{n-1}(\Theta|d(\mathring{t}))$$

with $f_0(\Theta|d(\mathring{t})) \equiv f(\Theta)$.

But the posterior estimate of parameters cannot be directly used as a new guess of the prior pdf as it is much more concentrated than a reasonable prior pdf. It is dangerous in the discussed context as the original prior has been assumed unreliable and thus the posterior pdf may be concentrated at false subset of Θ^*. We can avoid this danger by applying a special *flattening mapping* G

$$\mathcal{G} : f(\Theta|d(\mathring{t})) \rightarrow \hat{f}(\Theta) \tag{3.1}$$

By applying this mapping on posterior pdf, we get a new guess $\hat{f}(\Theta)$ of the prior pdf $f(\Theta)$.

The selection of feasible mapping consists of finding following compromise:

- $\hat{f}(\Theta)$ should ressemble $f(\Theta|d(t))$,
- $\hat{f}(\Theta)$ should be flat enough.

We can design G as follows. For the fixed $d(\mathring{t})$, we denote $\tilde{f}(\Theta) \equiv f(\Theta|d(\mathring{t}))$ Moreover, we need any flat distribution. Uniform pdf (even improper one) suits often to this purpose. Let denote the flat distribution $\bar{f}(\Theta)$. The result of flattening will be that $\hat{f}$, which minimize the following expression

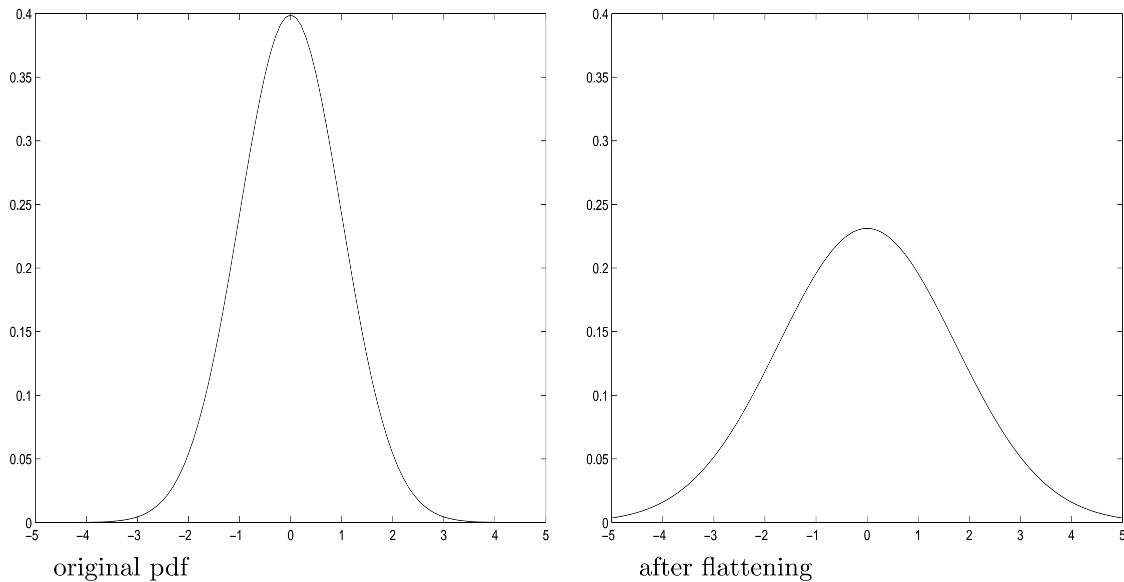
$$\mathcal{D}(\hat{f}||\tilde{f}) + q\mathcal{D}(\hat{f}||\bar{f}). \tag{3.2}$$

The KL distance (see 2.1) $\mathcal{D}(\hat{f}||\tilde{f})$ reflects the first requirement and $\mathcal{D}(\hat{f}||\bar{f})$ the second one. The positive weight $q > 0$ is a design knob that controls the compromise.

Proposition 3.3.1 (Optimal flattening mapping) *Let $\tilde{f}$ and $\bar{f}$ be a pdfs from previous discussion. Then, the pdf $\hat{f}$, defined on Θ^* minimising the functional (3.2) has the form*

$$\hat{f} \propto \tilde{f}^{\Lambda}\bar{f}^{1-\Lambda} \quad with \quad \Lambda = 1/(1+q) \in (0,1). \tag{3.3}$$

Example of using flattening mapping



original pdf after flattening

Now we are able to construct iterative estimates.

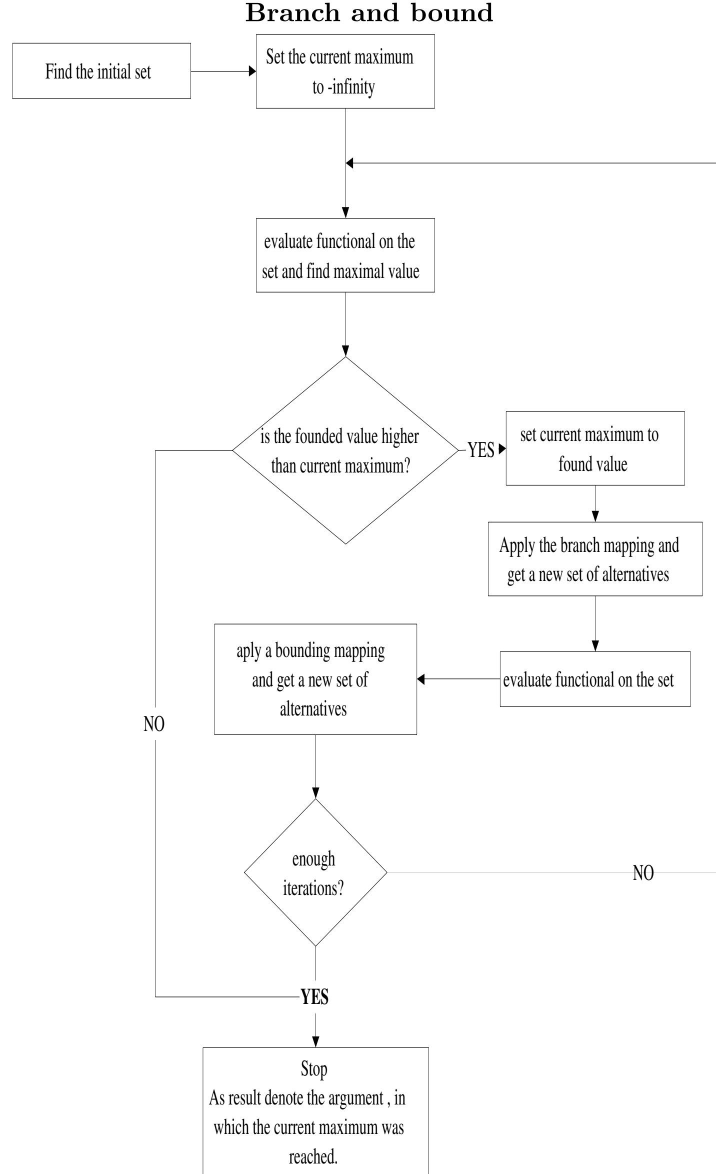
Generally by construction of prior estimate we search for such a variant of $\hat{f}(\Theta)$ for which the joint pdf $\hat{f}(d(\mathring{t}), \Theta)$, evaluated by an approximate algorithm, exhibits the highest posterior-likelihood $\hat{f}(d(\mathring{t}))$.

The dimensionality of the problem orients us towards the maximization by branch-and-bound techniques.

3.3.2 General description of branch and bound techniques

These techniques are used by searching of arguments of maximum of some function or functional. The main idea is very simple and works as follows.

Branch and bound



The most important thing is to choose the right branching mapping.

3.3.3 Branching by factor splitting

In the case, that we have no information about the structure of the mixture, the *factor splitting* is used. The idea is again simple. If a component is covering a large amount of data, there is a chance, that this component is hiding some other modes. We split one factor from this component and gets two new components which are the same up to the splitted factors. The original component is deleted. By that splitting it is very important to watch the shift of splitted factors from the original one. It is covered by the following general proposition.

Proposition 3.3.2 (Shifted factor) *Let pdfs $f, \bar{f}$ have a common support Θ^* and let g be a (vector) function defined on Θ^*. Let the pdf $\hat{f}$ minimise the KL distance $\mathcal{D}(\hat{f}||f)$ and have the prescribed norm*

of the shift in expectation of $g(\Theta)$

$$\left\langle \int g(\Theta)(\hat{f}(\Theta) - f(\Theta))\, d\Theta, \int g(\Theta)(\hat{f}(\Theta) - f(\Theta))\, d\Theta \right\rangle = \omega, \omega > 0. \tag{3.4}$$

Then, it has the form

$$\hat{f}(\Theta) \propto f(\Theta) \exp[- \langle \mu, g(\Theta) \rangle] \tag{3.5}$$

determined by the array μ *that is compatible with the scalar product* $\langle \cdot, \cdot \rangle$*. Thus, the solution of the considered task can be searched within the set of functions (3.5) "indexed" by the finite-dimensional array* μ*.*

The function $g(\Theta)$ was included there, so that we can influence the result. A proper choice of it helps us to stay within a desirable class of pdfs. Typically, we try to stay within the computationally advantageous class of conjugate pdfs.

Now we have done all preparation work. By combining all things from this chapter, we create algorithm `mixinit` for getting prior pdf. as well as structure of the mixture.

Chapter 4

Model description and software representation

This chapter outlines the specific model used. It insists especially upon the software aspects of this model. Under the terms factor, component etc. we will understand both pdf and the specific software representation of it. From the context, the right meaning always will be clear. The main input of all algorithms are measured data. Individual quantities are called *data channels*. For processing the data are organized in a global matrix `DATA`. Each row means time evolution of some channel. In `DATA(i,j)` is value of the channel i at time j. Current processing time is stored in a global scalar variable `TIME`.

The description of the model must logically start from the most elementary objects, i.e. factors. Each factor is a single regression model, which explains one channel `ychn`

$$d_t = \theta * \psi_t + e$$

where

- e is white process noise distributed as $N(0, r)$
- ψ_t is regression vector
- θ is regression coefficients vector

Point estimates of θ is denoted `Eth` and point estimate of r is `cove`. Clearly $d_t \sim N(\theta * \psi_t, r)$. Each factor needs to specify its structure (see section **??**), so the regression vector should be obtained. Here the factor structure is represented as a two-rows matrix. In each column the first value means channel on them it is depending and the second value means time delay of the channel.

For example :

$$\text{str} = \begin{pmatrix} 1 & 3 \\ 2 & 1 \end{pmatrix}$$

means that the regressor is `[DATA(1,TIME-2),DATA(3,TIME-1)]`

The factor structure may also contain a column $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ which is called as *factor offset* and it introduces an additional constant into the model.

Finally, from the software point of view, the factor is a structure with the following fields:

- `str` factor structure
- `ychn` pointer to channel, which is explained by this factor.
- `cove` point estimate of r
- `Eth` point estimate of θ

- `Cth` covariance matrix of parameter estimate

The factor must contain some additional fields for storing various statistic,(for exact description of factor see [5])

As a *static factor* we call the factor which does not depend on historical data. ie. the factor whose structure contains only zero delays and offset. Other factors are called *dynamic factors*

The next constructions need to operate with more factors. We will store all used factors in array `Facs`

A component is a product of factors. We store it as an array of pointers to corresponding factors. As pointer we mean simply the column index of the factor in array `Fac`.

A mixture is a weighted sum of components. We represent it as an array of weights of corresponding components `dfcs` and as a vector of components. Since the component is an array of pointers to factors, we get a matrix of pointers `coms`. Each row represents one component.

Again we include to the mixture some fields for storing useful statistics. The most important statistics are:

mixll posterior likelihood corresponding to $f(d(\mathring{t})))$.

commer statistic for testing a hypothesis that two particular components are the same.

comcan statistic for testing a hypothesis that a particular component is spurious.

At all the mixture is a structure with following fields:

`Facs` array of factors

`coms` components, matrix of pointers to factors

`dfcs` array of statistics for estimating components weights

`states` struct of useful statistics like `commer, mixll,comcan`

Chapter 5

Mixinit

The procedure `mixinit` gets the prior pdf `Mix0` as the input. The entire information about number of channels and about dynamism[1] is included there. The next inputs are parameters (options), which can essentially influence the processing. The best options are step by step setting as default. The options can be divided into three groups.

5.1 Parameter overview

numerical values

These options have numerical values. The value must be specified together with specifying the option.

- `niter`: The `mixinit` is an iterative algorithm, `niter` specifies the maximum number of iterations. Default value is 10.
- h : The number of runs for structure estimation. Default value is 10.
- `xcove` : A portion of `cove` included in the components split. Default value is 0.5.

bool options

There are options, which can have just values true or false. Default for all options is false.

- a : try to split only single component and single factor
- q : the component created has common factors with the original one
- d : use stopping rules
- c : make structure estimation within `mixinit`

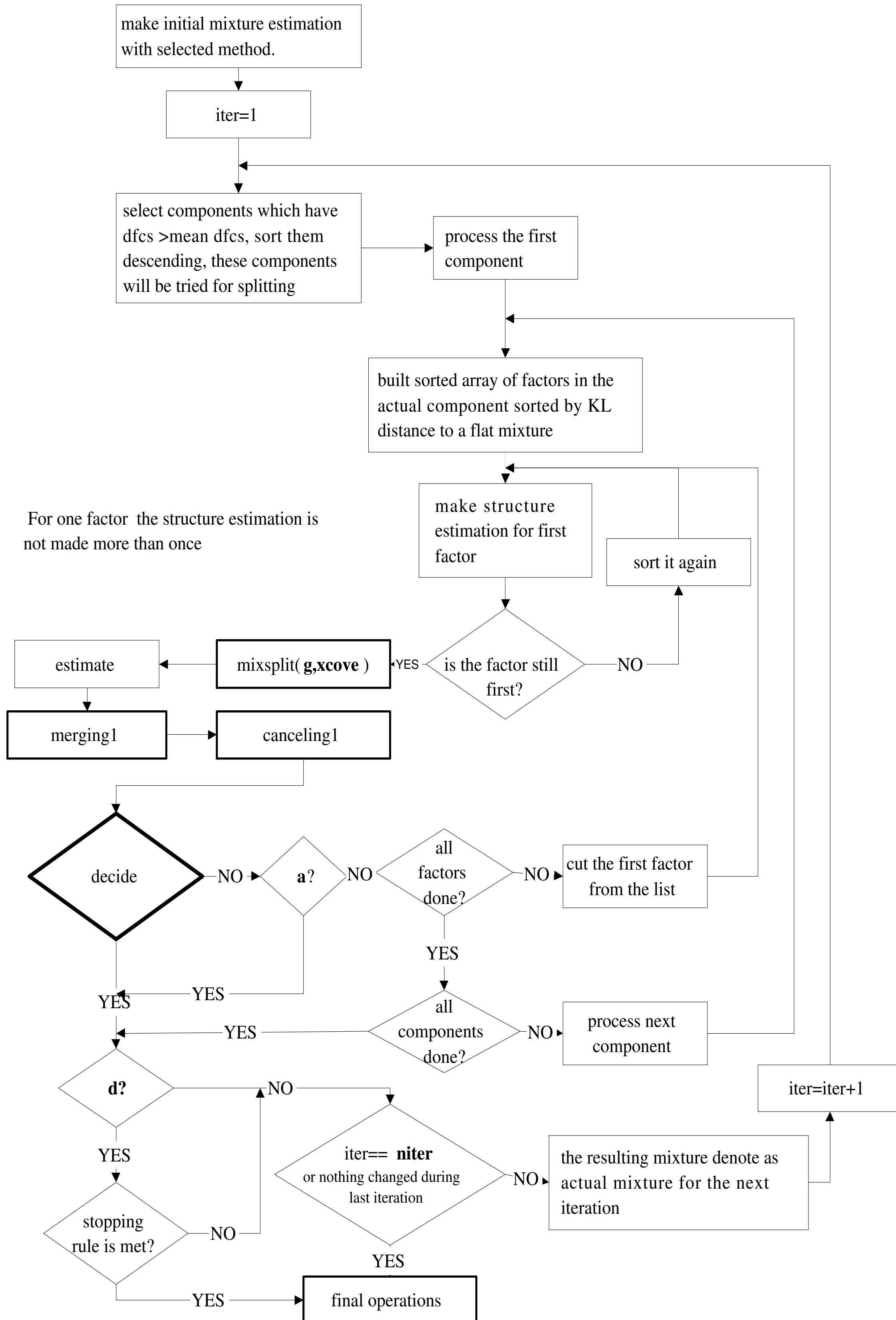
estimation method

These options influences the estimation method used. Single method can be selected. Default value is q.

- q : iterative quasi-Bayes mixture estimation
- b : iterative batch quasi-Bayes mixture estimation
- f : iterative mixture estimation based on forgetting branching
- related numerical values:
  - n : The number of iterations. Default value is 1.
  - s : The number of branching steps for forgetting branching. Default value is 10.

[1]i.e. if the mixture is dynamic or no.

5.2 Description



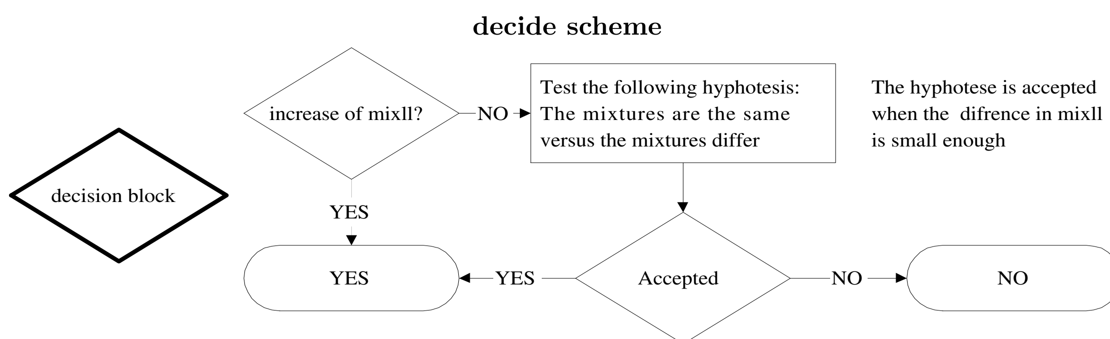
5.2.1 General scheme

A detail scheme of the algorithm, with the selectable options marked, is on the previous page. It is necessary to stress, that the pictures are not flow chart in classical sense. The algorithm acts in directions of the arrows. The bold polygons means a process, which is explained somewhere else. The explanations of the used block follow.

5.2.2 decide

To be able to speak about an increase of `mixll` we would need to specify 2 mixtures. We would not specify it, because it is clear here from the context. For getting the `mixll` of the second mixture we must run the estimation, which was not stressed here.

decide scheme

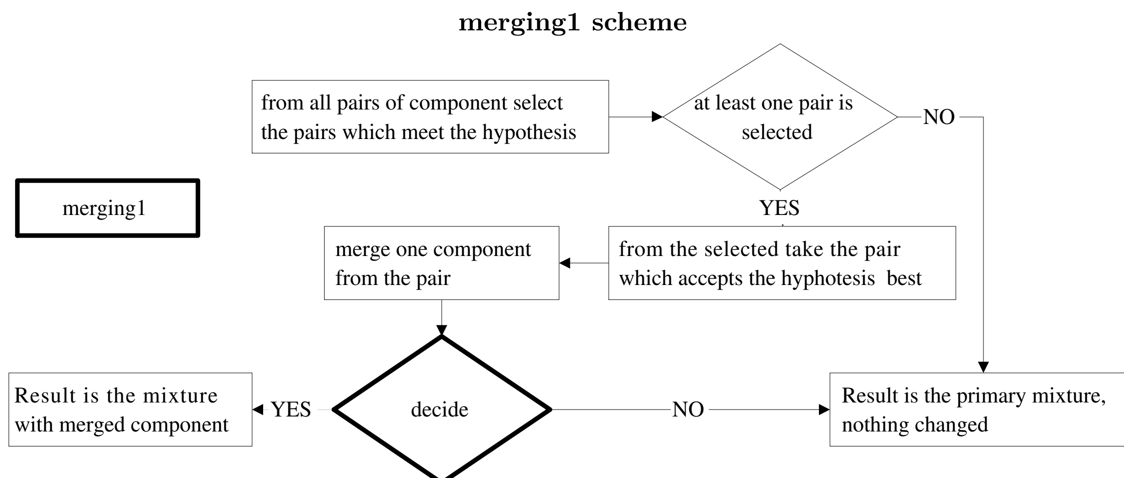


With this, we can decrease the `mixll`, but we help on the next processing[2]. In this case, the original mixture is saved and we can go back to it, if the next processing do not lead to good result.

5.2.3 merging1

The software mixture contains a statistic `commer`, (see chapter 4), which allows to test the hypothesis about equality of two components for each pair of components. All tests accept the hypothesis if value of some statistic `stat` is greater than a particular constant, i.e. `stat` $> \alpha$. When we say, that one component accepts the hypothesis better than other component, we mean that value of `stat` for the better component is higher than the second. We also use the ordering, which is induced by this relation. This relation and the ordering will be used in other diagrams too.

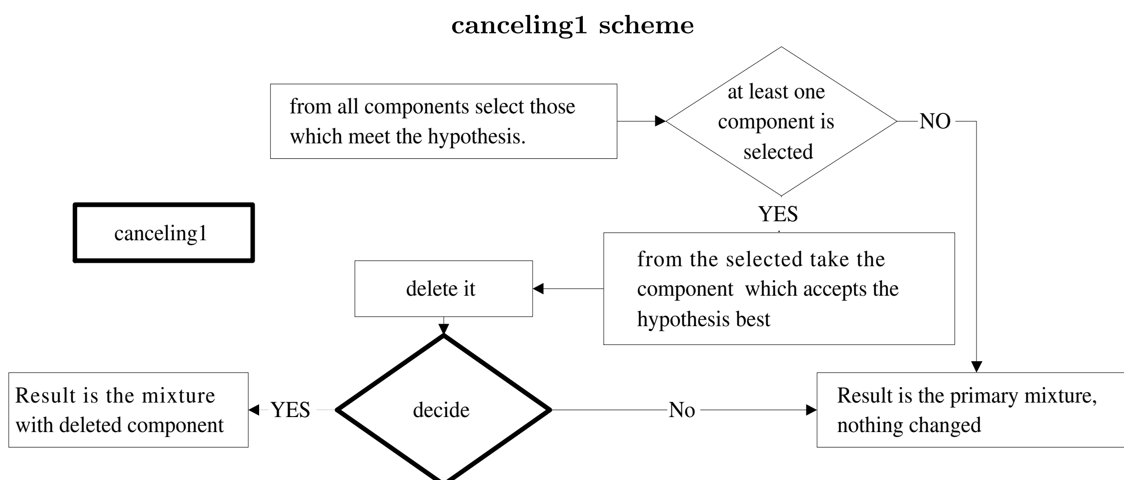
merging1 scheme



[2]the algorithm stops if nothing is changed in one iteration

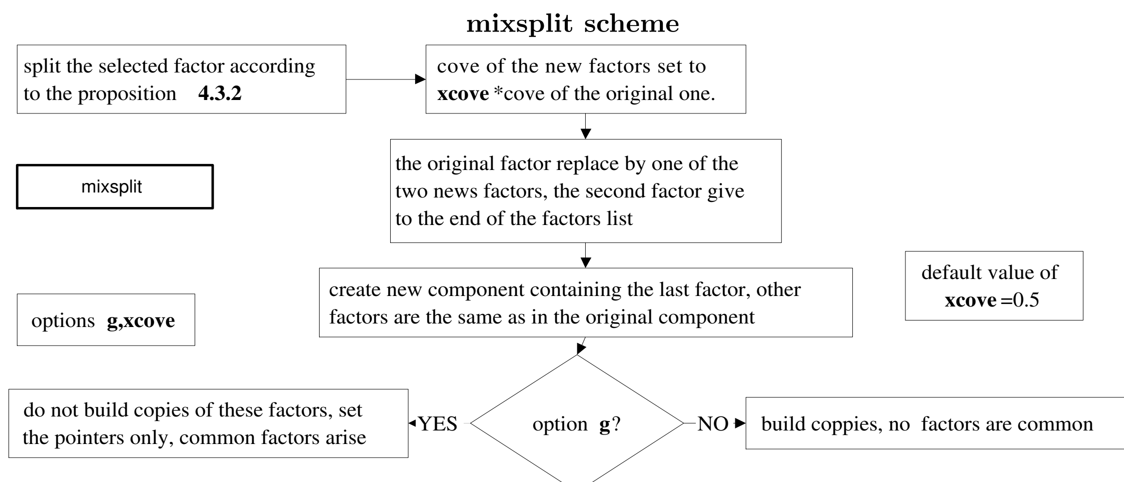
5.2.4 canceling1

The software mixture contains a statistic `comcan` (see chapter 4), which allows for each component to test the hypothesis, that this component is spurious. It means it is too close to a flat component.



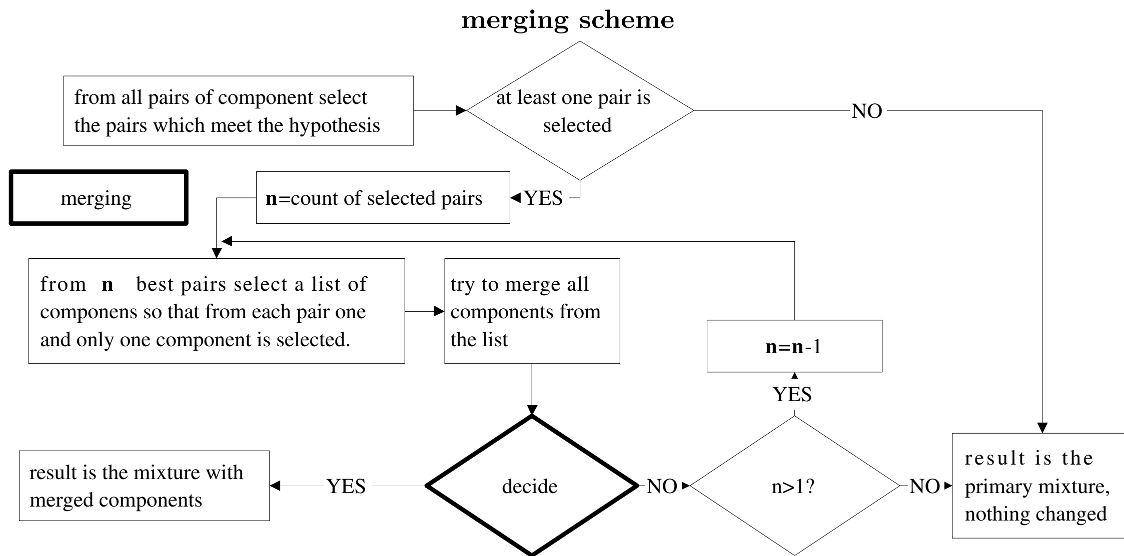
5.2.5 mixsplit

The subroutine `mixsplit` has two parameters.One is **g** and the second is **xcove**. Default value for **xcove** is 0.5 .



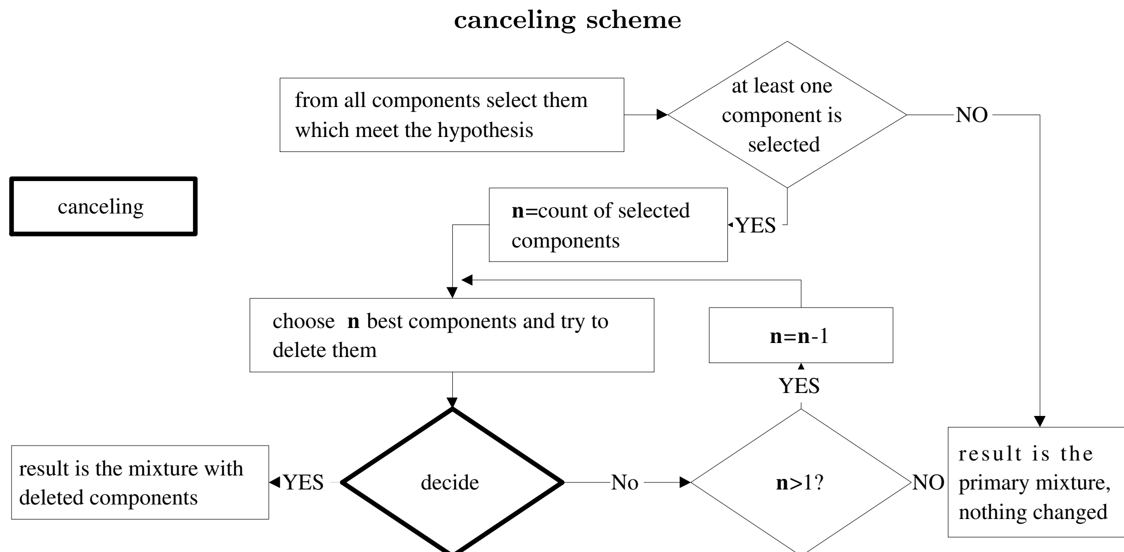
5.2.6 merging

The subroutine `merging` is generalization of the algorithm `merging1`. It tries to delete more than one component at one shot.



5.2.7 canceling

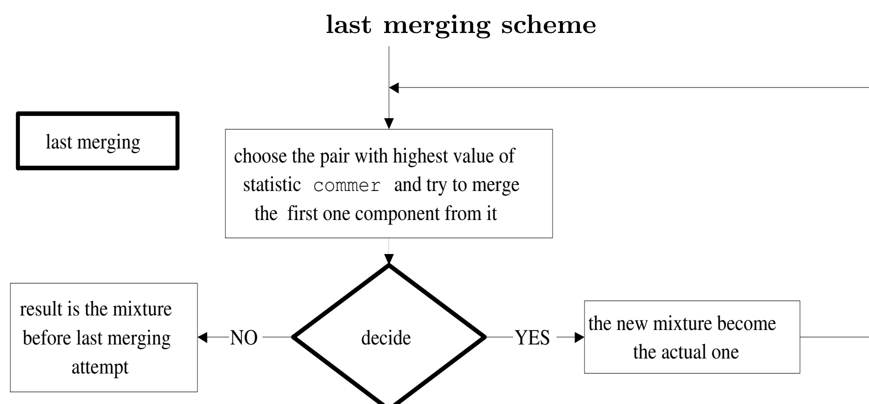
The subroutine `canceling` is generalization of `canceling1`. Again, we are trying to delete more components at once.



5.2.8 last merging

It was found, that is it reasonable to try delete components , for which the hypothesis has been rejected during the merging. It is done by the following diagram.

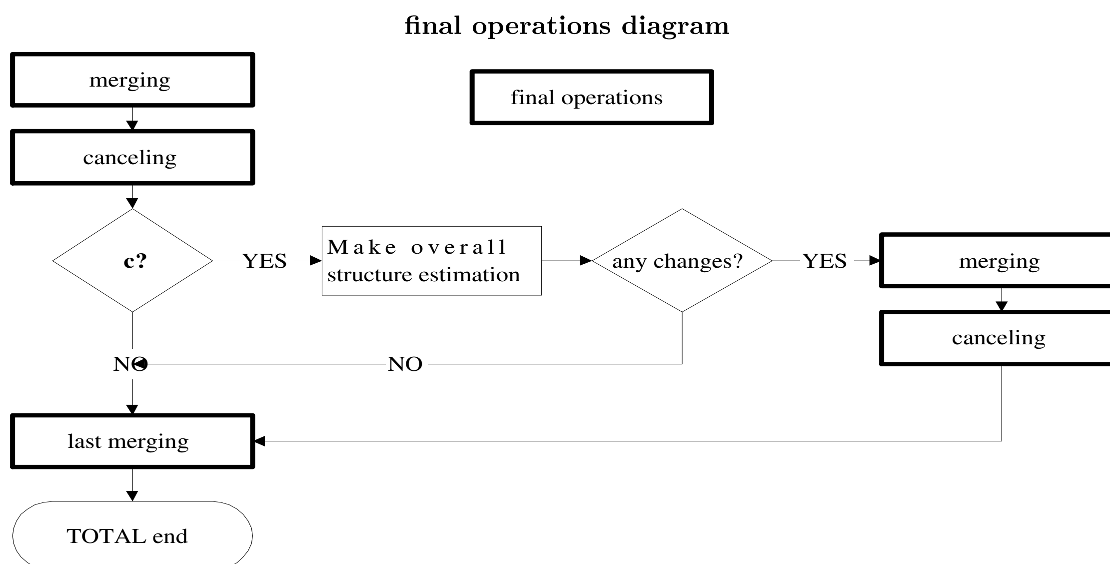
last merging scheme



5.2.9 final operations

The last, what needs to be specified is the final operations block from the main schema. It is composed from the previous blocks.

final operations diagram

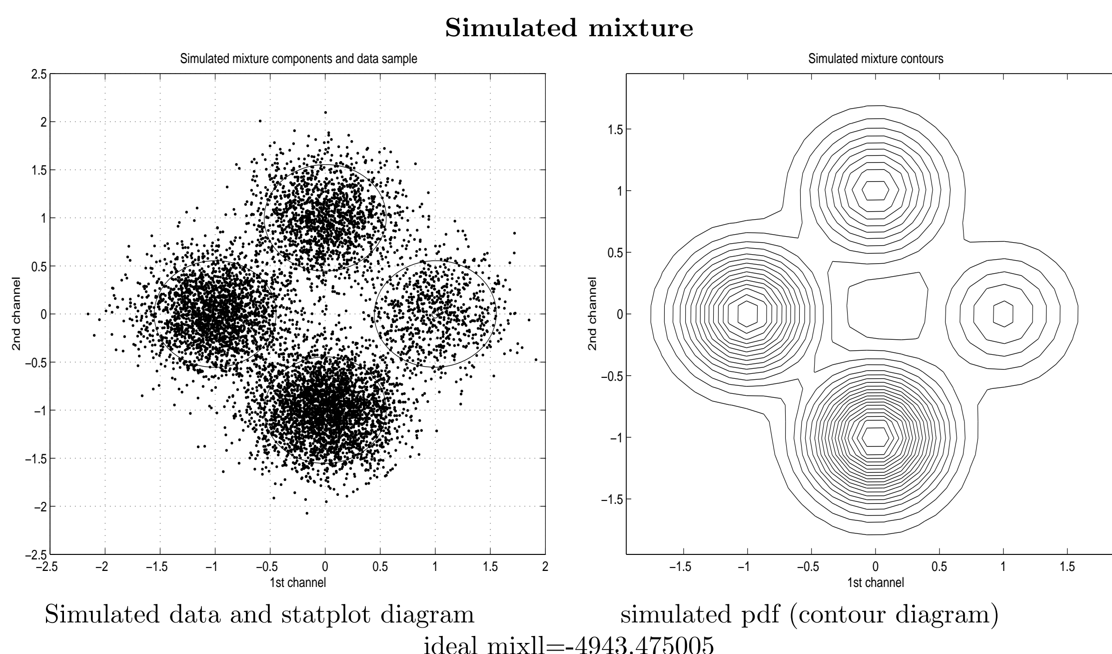


Chapter 6

Example of running mixinit

A two-dimensional static mixture with 4 component was created. A data sample of the length of 8 000 data items was generated. Thus, the global matrix DATA is 2× 8000. The simulated pdf and simulated data are are shown at the picture. We will use two different ways to visualize the two-dimensional mixture. The first is statplot diagram. It plots each component as an ellipsis, which borders the 75% probability region. The second is contour diagram. It plots the equi-probability level curves of the mixture pdf. In this example, all options were set to default values.

Simulated mixture



Simulated data and statplot diagram simulated pdf (contour diagram)

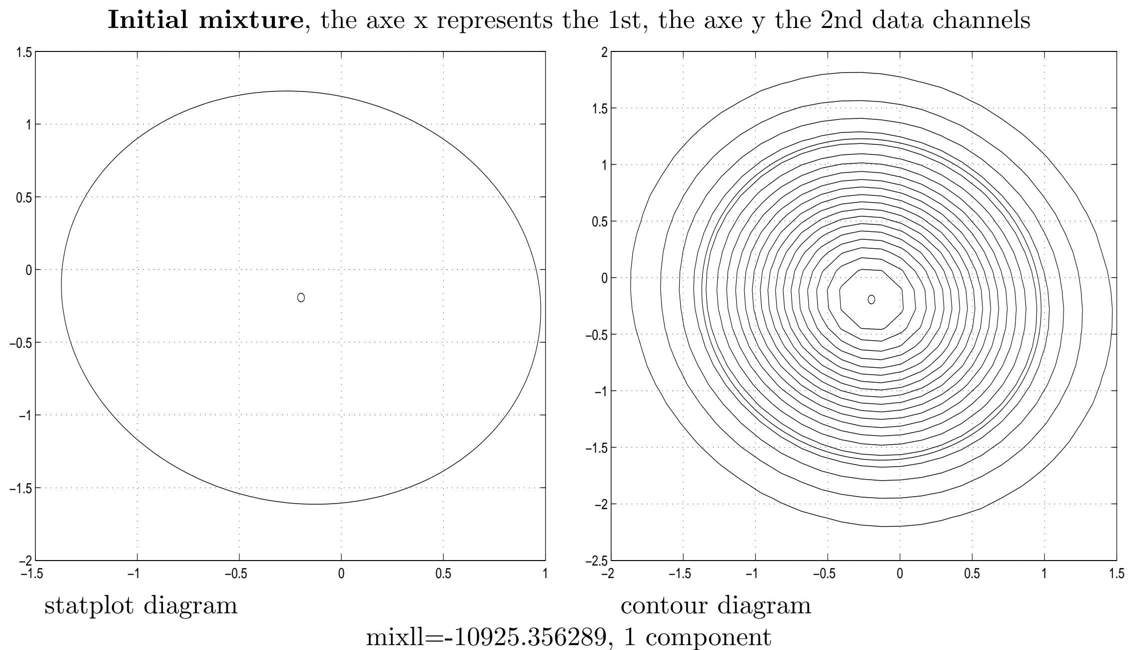
ideal mixll=-4943.475005

When simulated data and known model were used for estimation, the value of the posterior likelihood `mixll` was -4943.47005.

6.1 Processing of mixinit

At first, the algorithm makes the initial estimation. It uses one-component mixture as prior information.

Initial mixture, the axe x represents the 1st, the axe y the 2nd data channels



statplot diagram contour diagram

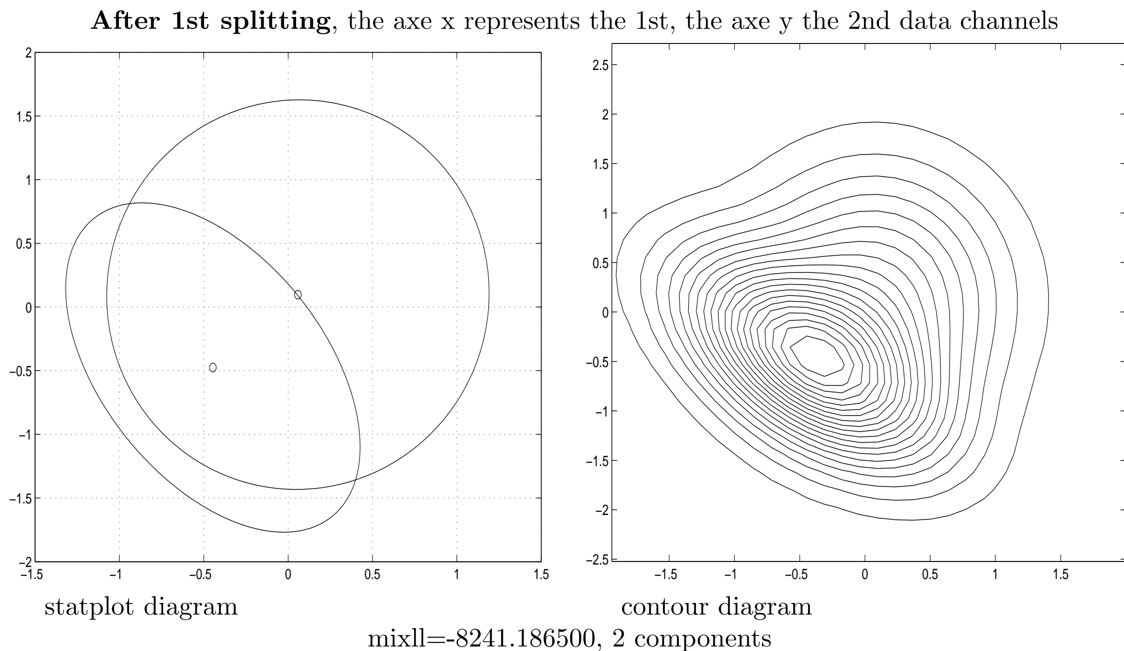
mixll=-10925.356289, 1 component

Then, the status of the estimated mixture in several iteration steps is displayed.

Mixture status in iteration 1

The mixture after component splitting is displayed.

After 1st splitting, the axe x represents the 1st, the axe y the 2nd data channels



statplot diagram contour diagram

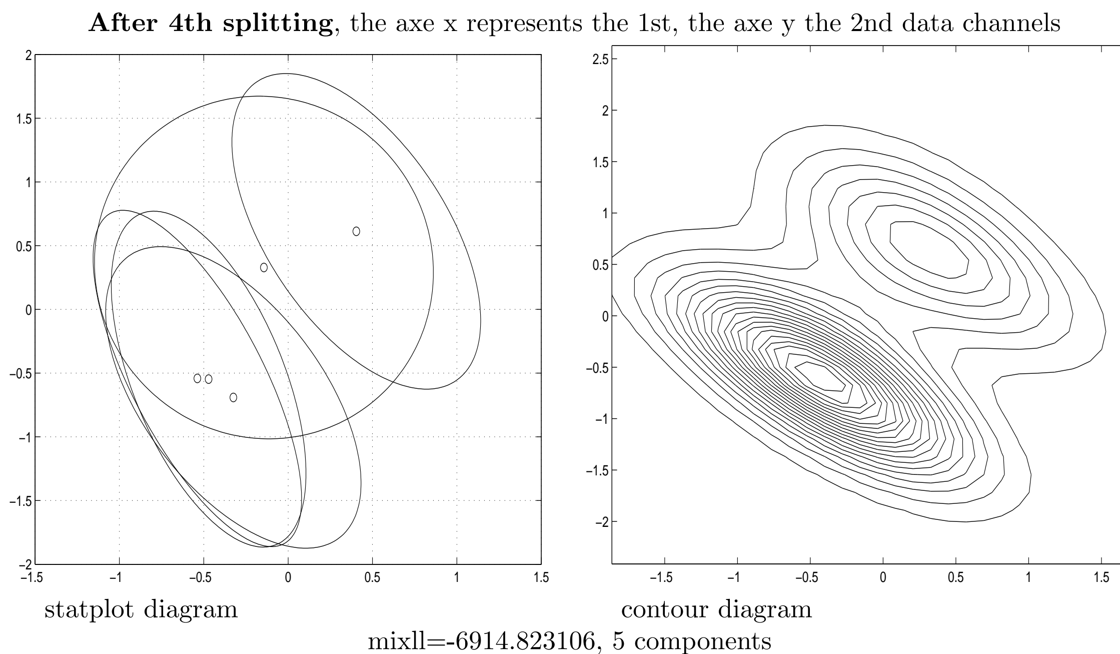
mixll=-8241.186500, 2 components

Of course, during this iteration, no components were merged and no component cancelled. These operations are meaningful only after more iterations.

Mixture status in iteration 4

The contour plot clearly shows that the pdf concentrates on the same half plane as the original mixture.

After 4th splitting, the axe x represents the 1st, the axe y the 2nd data channels



statplot diagram contour diagram

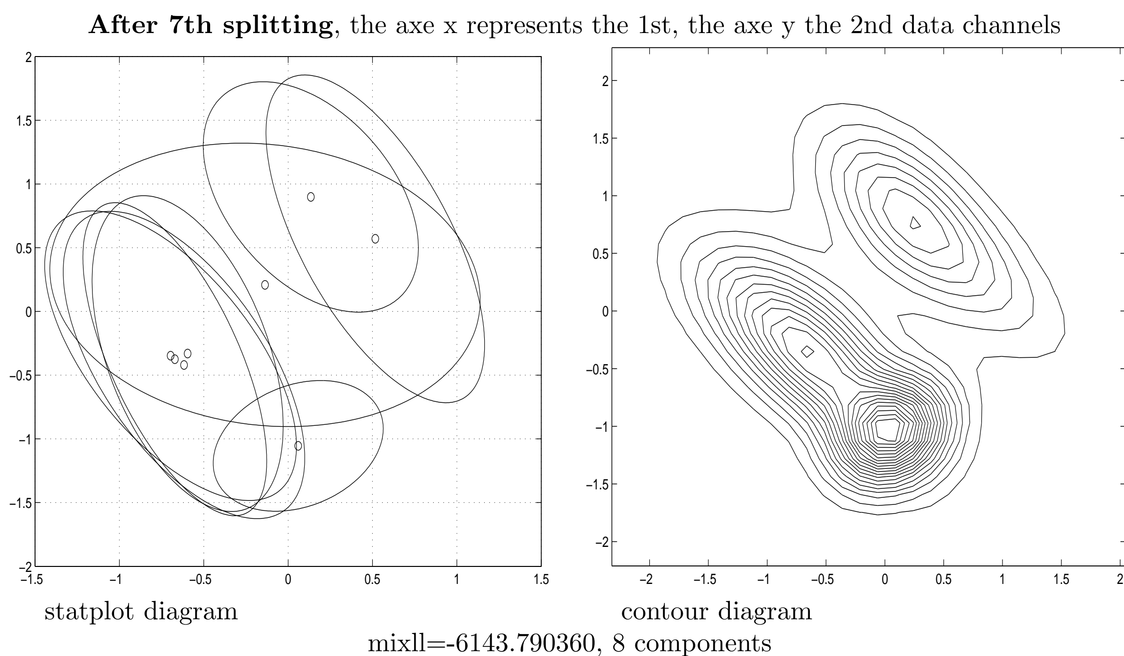
mixll=-6914.823106, 5 components

In this iteration, no components were merged nor cancelled. There were not even an attempt to make these operations. The hypothesis were not accepted.

Mixture status in iteration 7

In this iteration, the next mode arise.

After 7th splitting, the axe x represents the 1st, the axe y the 2nd data channels

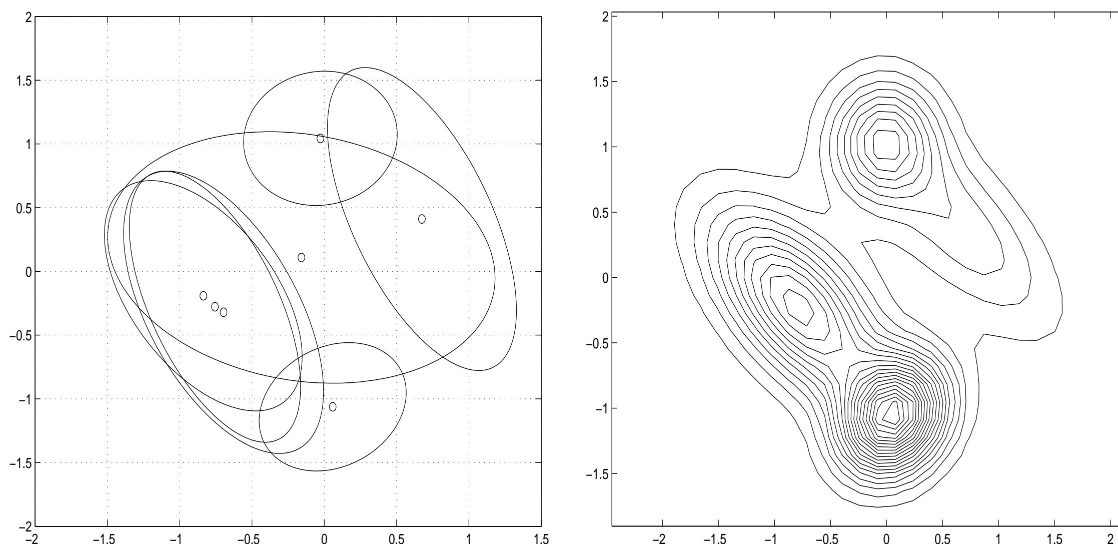


statplot diagram contour diagram

mixll=-6143.790360, 8 components

The merging attempt was successful. The mode is more pronounced now.

After successful merging, the axe x represents the 1st, the axe y the 2nd data channels



statplot diagram contour diagram

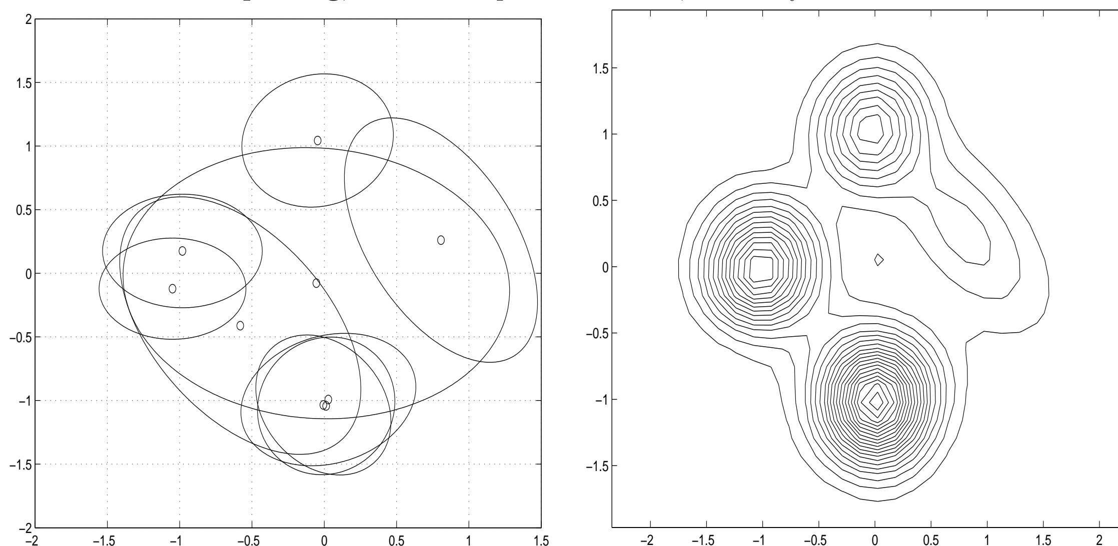
mixll=-5686.965734, 7 components

The hypothesis about containing an outlining component was again rejected.

Mixture status in iteration 10

Now, even 3 modes are precise. No merging nor canceling were done.

After 10th splitting, the axe x represents the 1st, the axe y the 2nd data channels



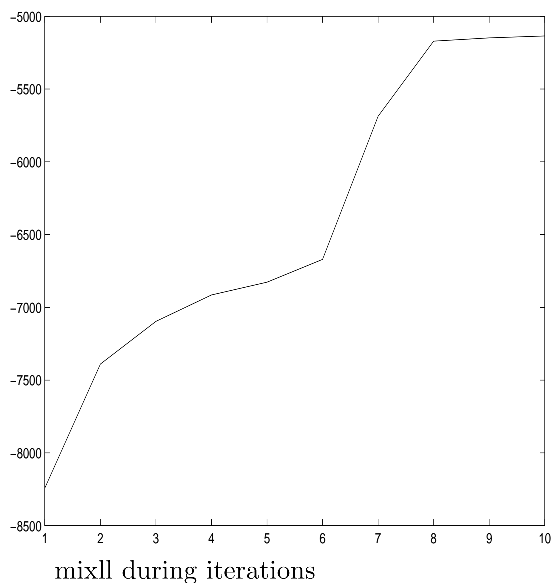
statplot diagram contour diagram

mixll=-5135.641524, 9 components

This was the result after the iterative part of the algorithm.

The gradual increase of mixll is shown on the figure bellow. The right column of the table shows values of statistic, that would be used in stopping rules (if the option d was selected). Algorithm would stopped if value of this statistic were smaller than 0.0376

Evolution of mixll during iterations



mixll during iterations

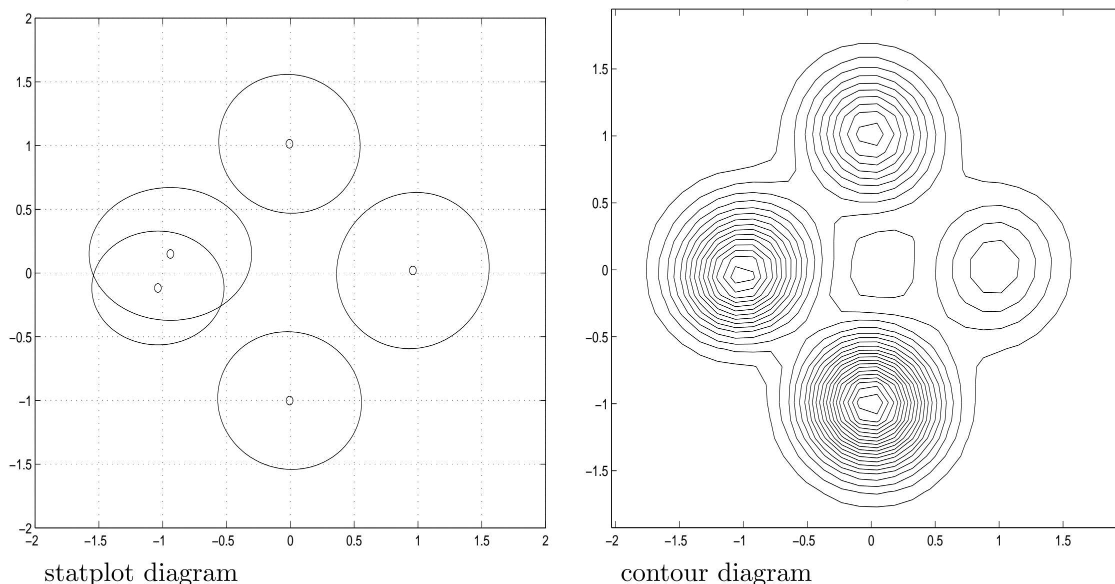
1	23.8299
2	7.8195
3	2.7549
4	1.7076
5	0.8281
6	1.4763
7	9.5763
8	4.9839
9	0.2115
10	0.1293

(relative growth) the statistic offered to the stopping rule

Final operations

The final operations are shown including attempts of merging surplus components. The hypothesis testing allowed neither cancel nor merge anything. It is now the time for last merging. In this case, the last merging is relatively successful. The result follows.

After the last successful merging, the axe x represents the 1st, the axe y the 2nd data channels

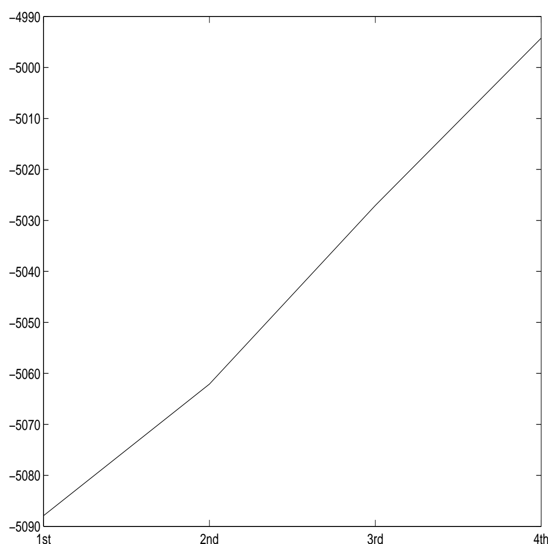


statplot diagram contour diagram

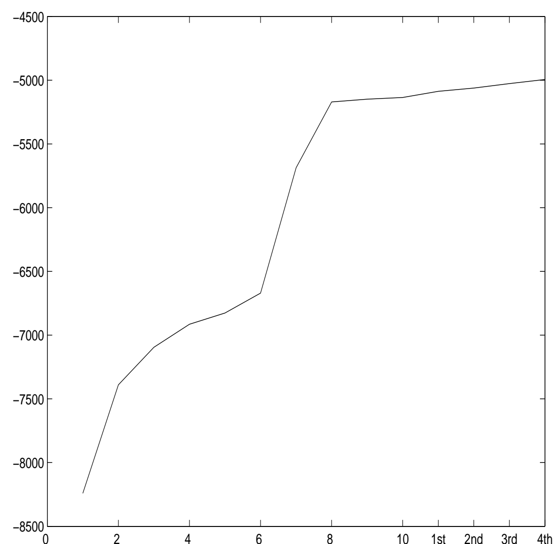
mixll=-4994.240284, 5 components

The following figure shows progress of mixll during final operations and during entire `mixinit` processing.

progression of mixll



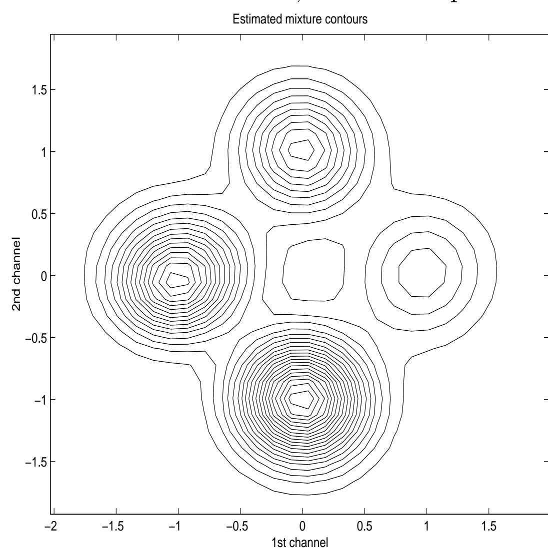
mixll during final merging



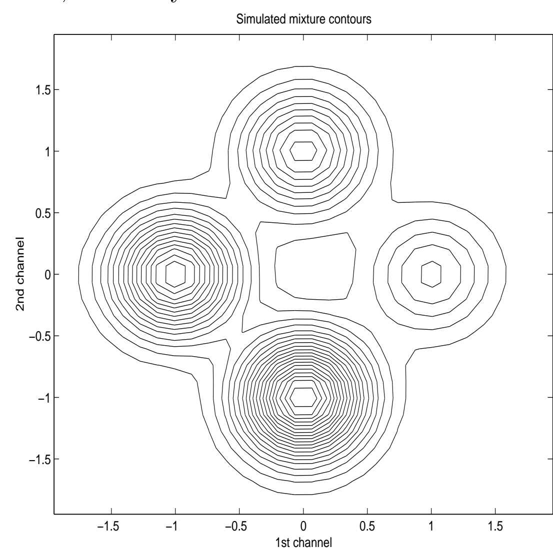
mixll during all process,
terminal merging applied after iteration 10

Now, the resulting mixture is compared by the original mixture.

RESULT, the axe x represents the 1st, the axe y the 2nd data channels



statplot diagram
estimated mixture, mixll=-4994.240284



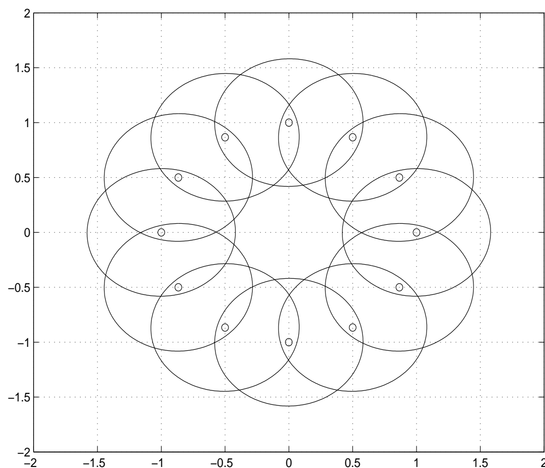
contour diagram
simulated mixture, mixll=-4943.475005

Chapter 7

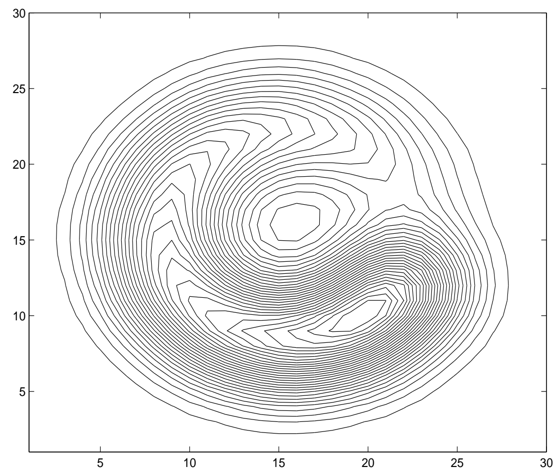
Selection of processing options

In this chapter, the individual options will be discussed in details and some conclusions will be drawn. Near all simulated data are from one general model, with 1000 data samples and optional number of components. The centers of the components are uniformly situated on the unit circle. Their `dfcs` grows linearly clockwise starting from point [1,0]. We demonstrate it on the following examples.

12 static components, the axe x represents the 1st data channel, the axe y represents the 2nd data channel

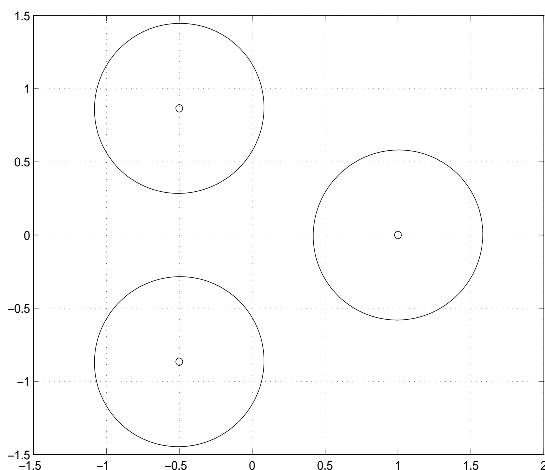


statplot diagram

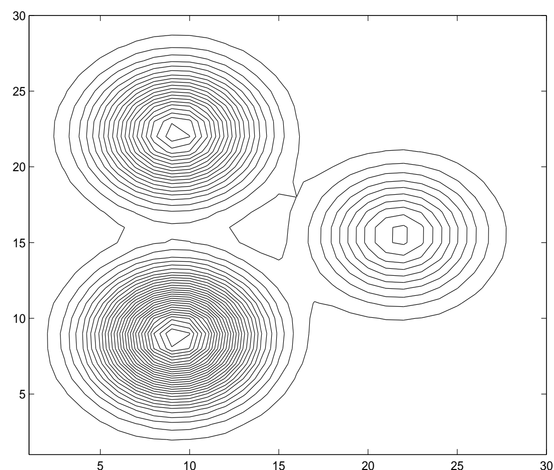


contour diagram

3 static components, the axe x represents the 1st data channel, the axe y represents the 2nd data channel



statplot diagram



contour diagram

7.1 Number of iterations steps - niter

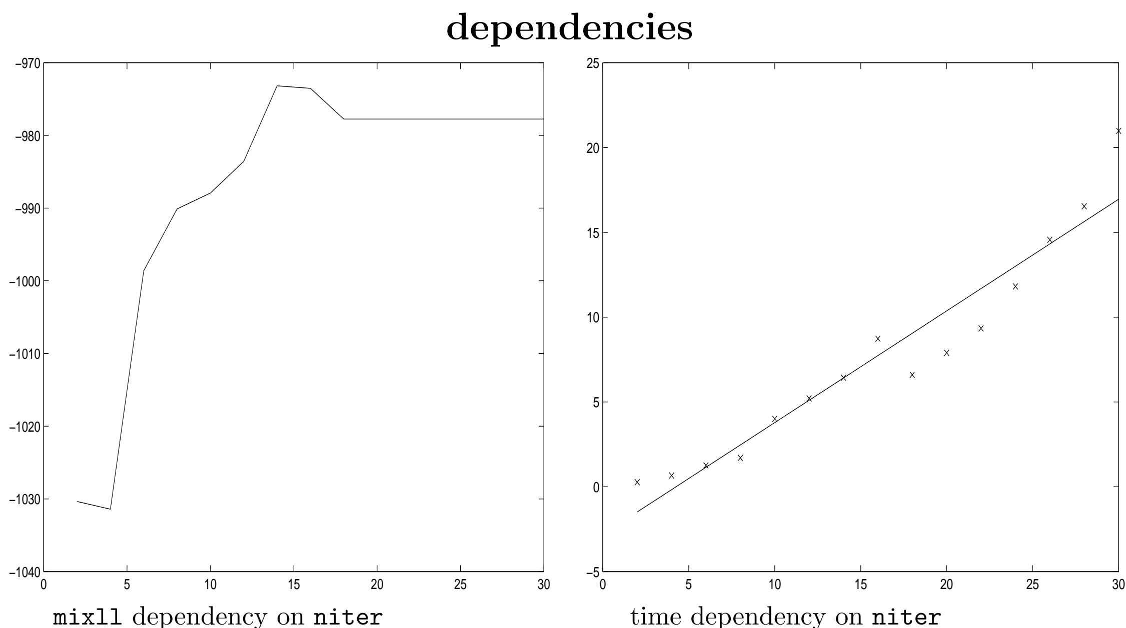
This option sets the maximum number of iterations. The real count of iterations in mixinit can be lower if the stopping rules are used, or if nothing has been changed during an iteration. Default value for this option is 10. Setting of higher value can lead to better results. But the computation time grows.

Conclusion: Setting of `niter` significantly above the default value does not lead to significantly better results.

The conclusion will be documented by some experiments.

7.1.1 12 static components

The left figure shows dependency of the resulting `mixll` on the parameter `niter`. The right figure shows dependency of the elapsed time[1] on the same parameter. The line sketched means estimate with least square method. It says how is it with the linearity of time dependency.



`mixll` dependency on `niter` time dependency on `niter`

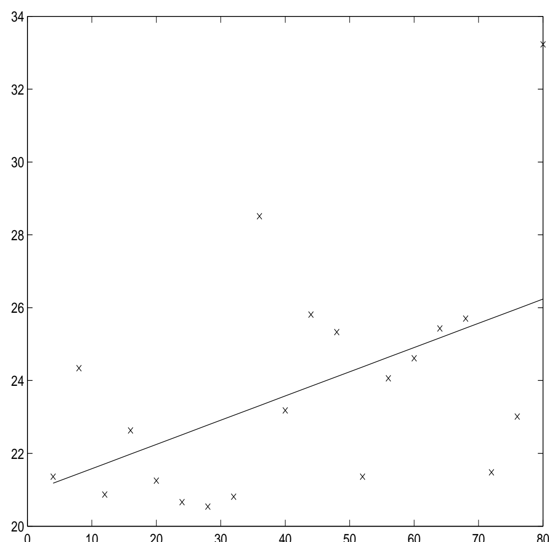
Note that when `niter` gets over a particular boundary, no improvement is obtained, but the computing time still grows. This fact has a simple explanation. In each iteration, the mixture is split even at the expense of getting lower `mixll` (see diagram DECIDE). After finishing the iterative part of `mixinit` algorithm, the mixture that was the best during the processing is selected. In this case, the many splitting operations does not lead to improvement and the mixture obtained for a fewer iteration is selected.

- Theoretically, it is possible that a further increase of `niter` could bring better results, but it would not pay off.

- The dependency of computing time seems to be linear. The measuring of time under Windows is not very precise. The same process could have two different times. Nobody knows what Windows is doing on background. We must deal with this fact and thus we must tolerate a higher variance of the time. To illustrate this fact, the time requirements for running the same estimation is displayed:

[1]It is time in seconds on PC athlon 650MHz

Figure illustrating the variance of measuring time under Windows.
The elapsed time for running the same procedure is displayed.



The next table shows data plotted on the figure `dependencies`.

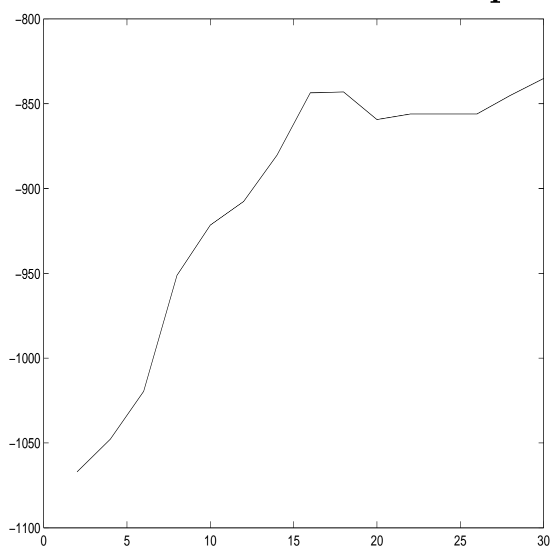
vysledek, part 1

niter	2	4	6	8	10	12	14	16
mixll	-1030.35	-1031.42	-998.61	-990.13	-987.93	-983.58	-973.19	-973.53
time	0.27	0.66	1.26	1.71	4.01	5.21	6.43	8.73

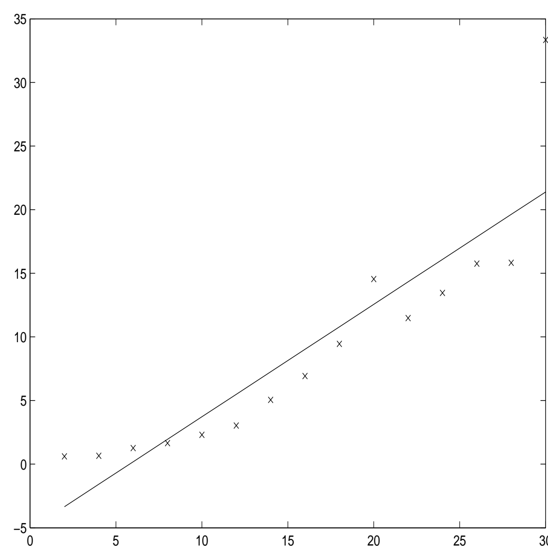
niter	18	20	22	24	26	28	30
mixll	-977.76	-977.76	-977.76	-977.76	-977.76	-977.76	-977.76
time	6.60	7.90	9.34	11.81	14.56	16.53	20.98

7.1.2 8 static components

dependencies



`mixll` dependency on `niter`



time dependency on `niter`

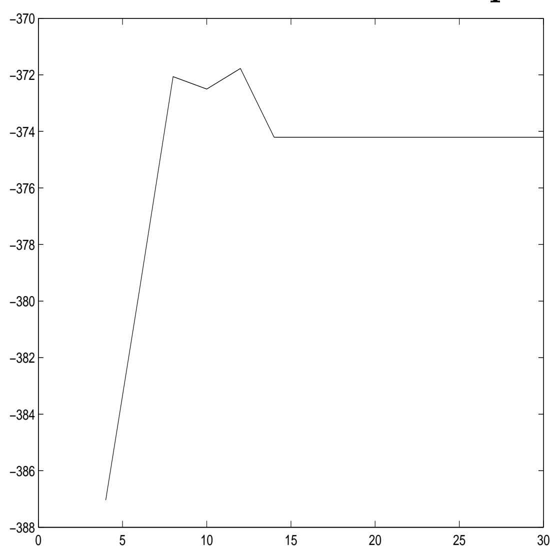
niter	2	4	6	8	10	12	14	16
mixll	-1067.11	-1047.75	-1019.56	-951.23	-921.66	-907.69	-880.43	-843.65
time	0.61	0.66	1.26	1.65	2.30	3.03	5.05	6.92
ncom	2	2	2	3	4	4	5	6

niter	18	20	22	24	26	28	30
mixll	-843.13	-859.39	-856.15	-856.15	-856.15	-845.22	-835.15
time	9.45	14.55	11.48	13.46	15.76	15.82	33.34
ncom	6	5	6	6	6	6	6

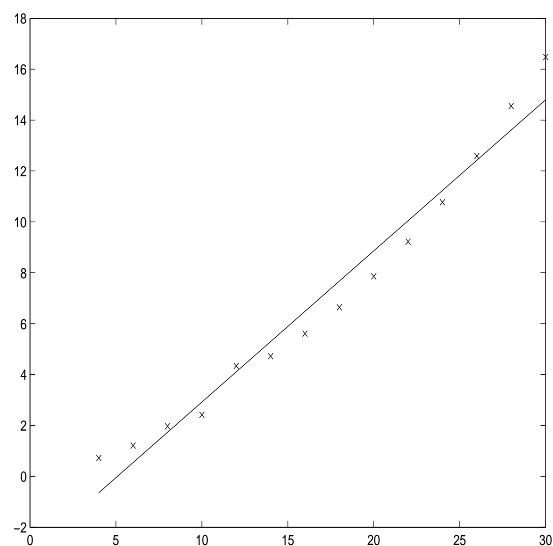
This is a good example of the fact, that even after a decrease of `mixll` with `niter` an improvement may occur. However, this improvement does not seem substantial.

7.1.3 3 static components

dependencies



`mixll` dependency on `niter`



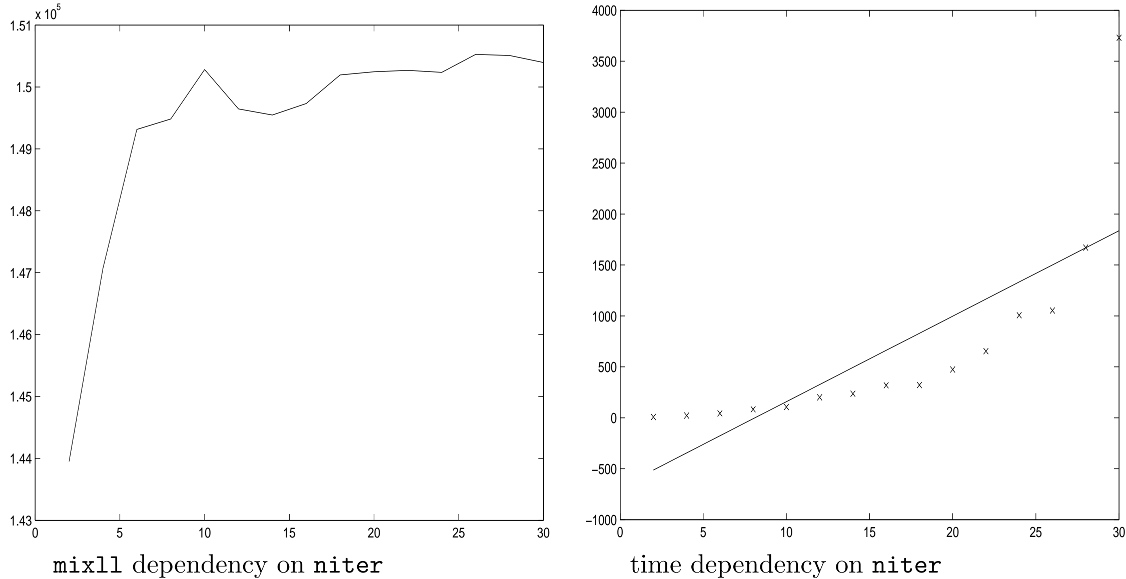
time dependency on `niter`

niter	4	6	8	10	12	14	16
mixll	-387.04	-379.63	-372.07	-372.50	-371.77	-374.21	-374.21
time	0.72	1.21	1.97	2.42	4.34	4.72	5.61
ncom	3	4	5	6	5	7	7

niter	18	20	22	24	26	28	30
mixll	-374.21	-374.21	-374.21	-374.21	-374.21	-374.21	-374.21
time	6.64	7.86	9.22	10.77	12.58	14.55	16.48
ncom	7	7	7	7	7	7	7

Behaviour of `mixinit` in this case is similar to that with 10 components.

7.1.4 Real data



`mixll` dependency on `niter` — time dependency on `niter`

niter	2	4	6	8	10	12	14	16
mixll	143950.01	147069.90	149314.33	149481.58	150281.72	149645.64	149546.54	149731.74
time	7.80	21.31	42.62	83.38	105.90	200.04	235.63	319.11

niter	18	20	22	24	26	28	30
mixll	150194.64	150244.99	150268.37	150236.46	150526.23	150509.51	150394.69
time	321.81	473.90	653.67	1006.23	1052.98	1670.39	3729.94

This data set rejects the hypothesis (based on simulated data) about linear time dependency on niter. It is logical. By the big number of iterations, the stucture of the mixture is richer and thus all operating with it needs more time. But from this cases is clear, that there is no argument, for setting a high value of `niter`. The value about 10 seems to be sufficient.

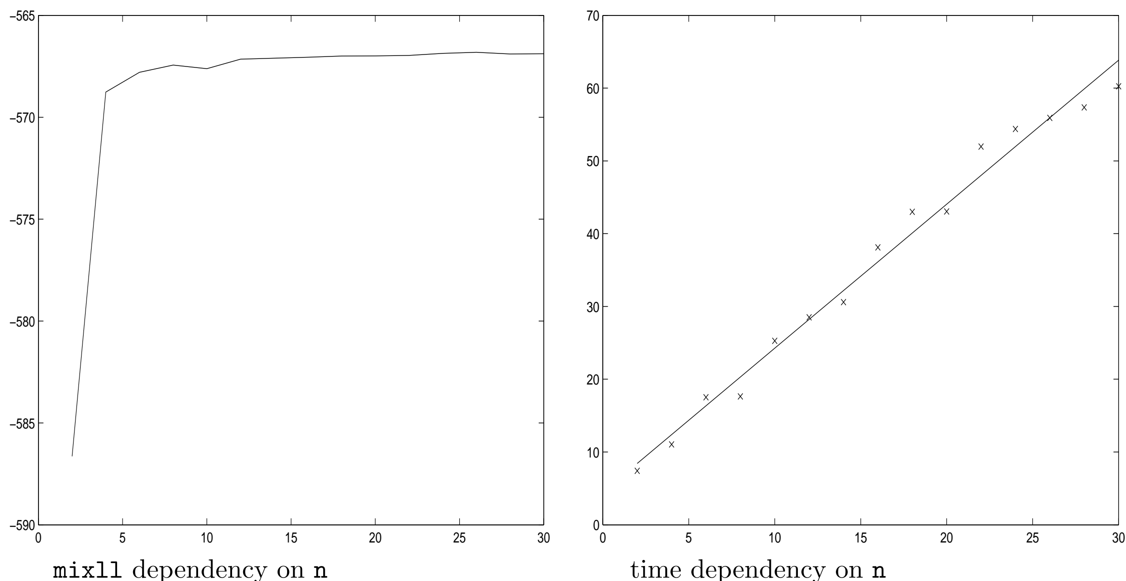
7.1.5 Conclusion

The default value were approved as a convenient one. If more computing time is at disposal, `niter` could be slightly increased. To invest the computing time into other options seems to be better.

7.2 Number of iterations in estimation -n

This option influences approximate parameter estimation. It determines the number of iterations for the estimation. The estimation takes approximately 96% of entire computing time. Single estimation step for a given data width needs a fixed computation time. So the time needed for entire procedure should depend linearly on `n`.

3 static components



`mixll` dependency on `n` time dependency on `n`

niter	2	4	6	8	10	12	14	16
mixll	-586.64	-568.76	-567.79	-567.44	-567.62	-567.15	-567.10	-567.06
time	7.41	11.04	17.52	17.63	25.27	28.50	30.60	38.12
ncom	4	4	4	4	4	4	4	4

niter	18	20	22	24	26	28	30
mixll	-566.99	-566.99	-566.97	-566.87	-566.81	-566.90	-566.88
time	43.00	43.06	51.96	54.38	55.91	57.35	60.25
ncom	4	4	4	4	4	4	4

The hypothesis about linearity of time dependency is confirmed. The quality of result grows up to some level. The values `n` close to 10 seems to be the best selection. It is interesting, that all variants have lead to the same number of components. It indicates that the resulting structure was equal. Thus it is a good idea to try to use small value of `n` and after finishing of `mixinit` make one estimation with big `n`. With the same data we have tried run `mixinit` with `n`=1 and after that we estimate result with `n`=14. The result is as follows.

	n=14	n=30	n=1 last n=14
mixll	-567.10	-566.8845	-566.5325
time	30.60	60.25	4.4000
ncom	4	4	5

The column n=1 last n=14 means that we run `mixinit` with `n`=1 and after that we make the estimation with `n=14` We obtain result with the same value of `mixll`, but the time needed is here ten times smaller than before. Now, we try this trick with other data.

8 static components

	n=10	n=2 last n=14
mixll	-814.5167	-815.6284
time	20.5400	7.5800
ncom	5	8

12 static components

	n=10	n=2 last n=14
mixll	-952.7136	-965.0735
time	17.08	6.15
ncom	6	6

real data

	n=10	n=2 last n=14
mixll	151117	149917
time	972.67	308.80
ncom	9	8

7.2.1 Conclusion

These examples show, that if the computing time is bounded, it is better to invest time to the last estimate than to make all estimates within iterations with a big `n`.

7.3 Dependency of computing time on the dimension of data

Now, the dependency of the computing time on the dimensions of `DATA` will be discussed. At first we will investigate the number of columns. i.e. size of data samples `ndat`.

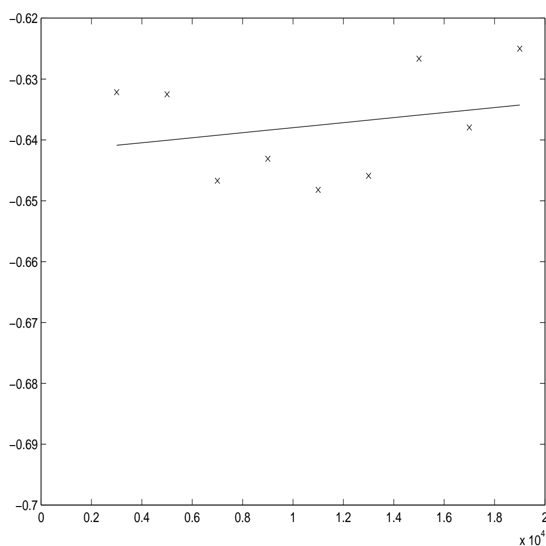
7.3.1 Sample size `ndat`

We generate one realization from each mixture with 20 000 data samples. then we use parts of this sample with different sizes. Very interesting is to observe the fraction `mixll/ndat` as it indicate value of the information brought by new data from a fixed system.

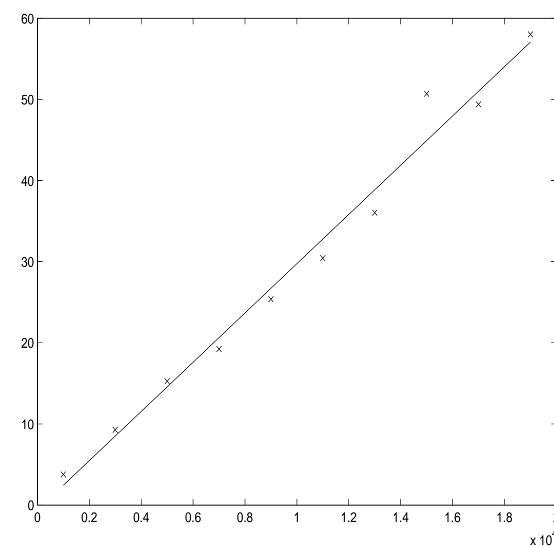
3static components

The left hand figure shows values of statistic `mixll/ndat`, the line means least square estimate. The second figure plots dependency of computing time, as in the previous pictures.

RESULT



evolution of `mixll/ndat` in dependence on `ndat`



evolution of the consumed time in dependence on `ndat`

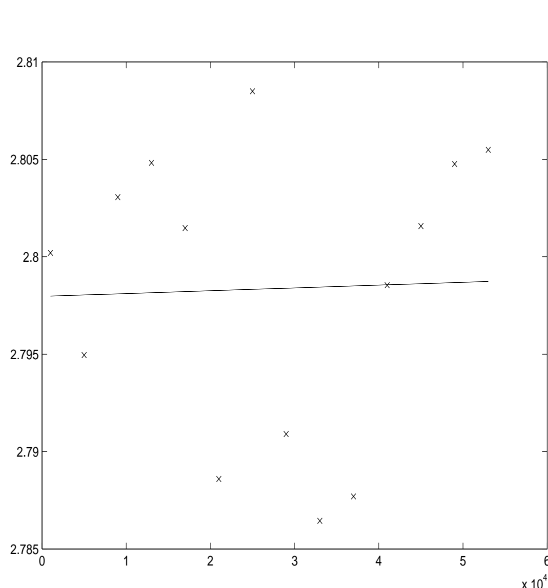
As the slope of the line fitted to `mixll/ndat` is $4.1 * 10^{-7}$, we can accept, that the ratio `mixll/ndat` is close to a constant. The linear dependency of time on `ndat` is clear from the picture.

The data shown in figures are:

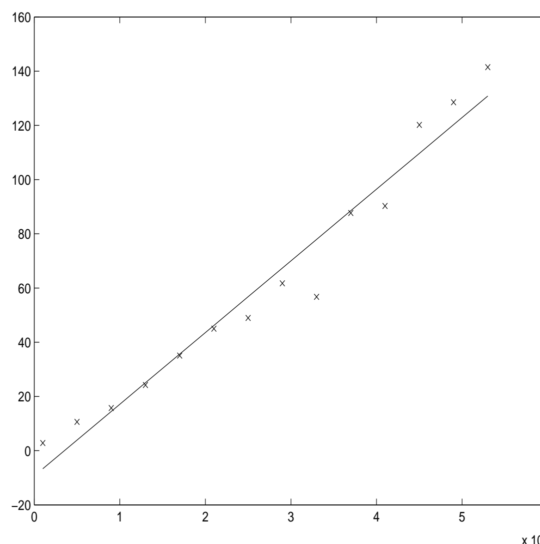
ndat	1000	3000	5000	7000	9000	11000	13000	15000	17000	19000
mixll/ndat	-0.72	-0.63	-0.63	-0.65	-0.64	-0.65	-0.65	-0.63	-0.64	-0.63
time	3.79	9.28	15.27	19.23	25.37	30.43	36.03	50.69	49.38	58.00
ncom	4	5	5	5	4	8	8	6	7	6

real data

RESULT



evolution of mixll/ndat in dependence on `ndat`



evolution of the consumed time in dependence on `ndat`

The slope of the fitted line is $1.4 * 10^{-8}$, thus `mixll/ndat` is close to a constant.

`ndat`	1000	5000	9000	13000	21000	25000	29000	37000	41000	45000	49000	53000
`mixll`/`ndat`	2.80	2.80	2.80	2.80	2.79	2.81	2.79	2.79	2.80	2.80	2.80	2.81
time	2.80	10.60	15.71	24.22	44.98	48.94	61.74	87.67	90.29	120.17	128.53	141.49
`ncom`	7	6	6	6	9	6	6	10	7	7	7	7

Conclusion

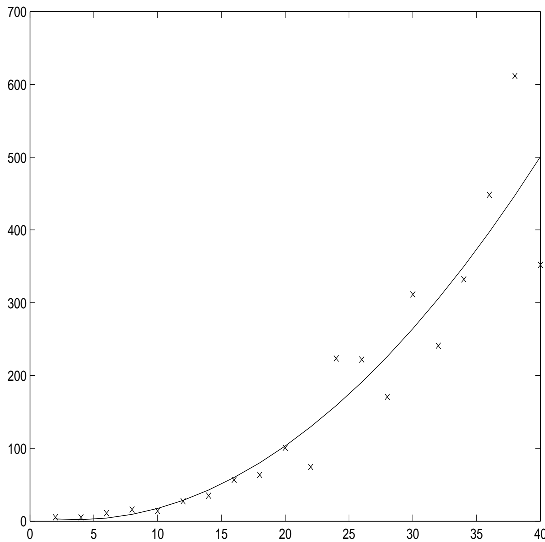
Previous data comfirmed the thesis about linear dependence of the time on `ndat`. The constant character of the fraction `mixll/ndat` can be seen,too.

7.3.2 Dependence on the number of channels - `dim`

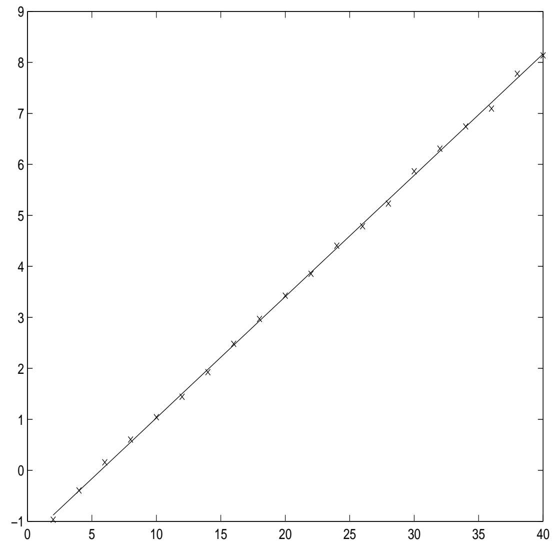
Now, we will investigate the number of rows of `DATA`. i.e. number of data channels `dim`.

By construction , the quadratic dependence of the computing time on the `dim` is expected. The time for processing of real data will be very high iac.

RESULT



evolution of the consumed time in dependence on `dim`



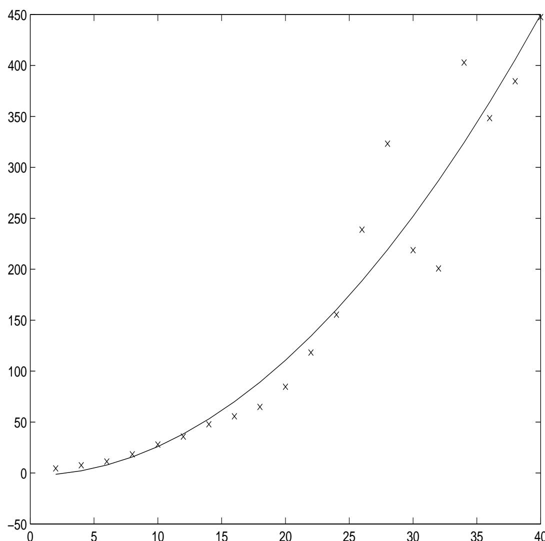
evolution of `mixll/ndat` in dependence on `dim`

dim	2	4	6	8	10	12	14	16	18	20
mixll/ndat	-0.9689	-0.3962	0.1571	0.6035	1.0405	1.4426	1.9240	2.4803	2.9677	3.4251
time	5.21	5.11	10.93	15.87	13.79	27.29	34.83	56.63	63.33	100.62

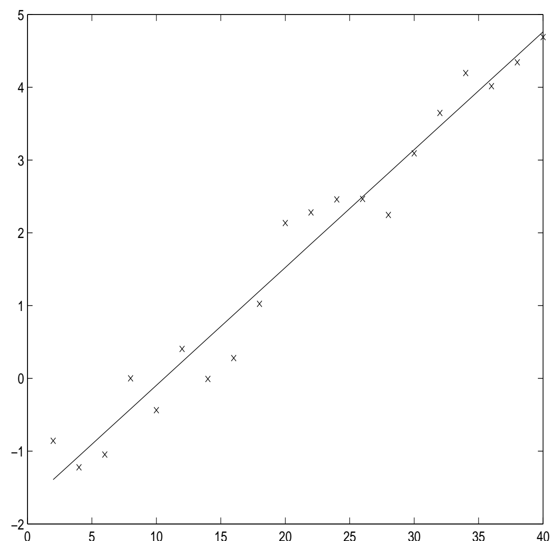
dim	22	24	26	28	30	32	34	36	38	40
mixll/ndat	3.8539	4.4037	4.7857	5.2301	5.8643	6.3085	6.7430	7.0947	7.7769	8.1362
time	74.37	223.43	222.01	170.55	311.37	240.80	332.14	448.24	611.59	352.01

The time dependency seems to be really quadratic. We must deal with greater time requirements. The time needed for make the case with dimension 40 is more than 100* bigger, than the time for dimension 2. Very interesting is the evolution of the statistic `mixll/ndat`. The column count `ndat` is constant here. Hence, the `mixll` seems to depend linearly on the dimension of the problem. The next experiment could reject it.

RESULT



evolution of the consumed time in dependence on `dim`



evolution of `mixll/ndat` in dependence on `dim`

dim	2	4	6	8	10	12	14	16	18	20
mixll/ndat	-0.85	-1.22	-1.04	0.00	-0.43	0.40	0.00	0.27	1.02	2.13
time	4.51	7.53	11.26	18.40	27.90	35.76	47.84	55.64	64.81	84.64
ncom	4	7	6	7	4	6	5	6	5	6

dim	22	24	26	28	30	32	34	36	38	40
mixll/ndat	2.28	2.45	2.46	2.24	3.09	3.64	4.19	4.01	4.34	4.68
time	118.20	155.39	238.81	323.24	218.71	200.70	402.82	348.23	384.47	447.43
ncom	6	6	5	8	6	9	5	8	5	4

The hypothesis about linear dependency of `mixll` on `dim` could not be rejected.

conclusion

The thesis about quadratic time dependency is accepted. We can consider, that the time needed for dimension 40 is approximately 100 times greater, than the time for dimension 2.

7.4 Boolean options

The option 'c' is not studied here. The overall structure estimation brings some improvement, but it is an independent problem weakly related to `mixinit`. It can be run after finishing the `mixinit` algorithm.

7.4.1 a,g,d

We will inspect the options 'a','g','d' and default, which is denoted as '0'. Their meaning is described in section 5.1. We will consider the combination of them, too.

For each mixture tested we made 10 independent realizations. Each option was tried on all these realizations. Then the results was sorted and displayed in the table. Then the average values are computed. The meaning of used symbols is as follows.

symbol	meaning
$\overline{\texttt{mixll}}$	average of `mixll`s on all realizations
std	standard deviation
`nmixll`	values of `mixll/ndat`
$\overline{\texttt{ncom}}$	average component count
$\overline{\texttt{time}}$	average value of time needed for running `mixinit`.

7 static componens

Table of results

Option	a	0	ad	d	ag	g	dg	agd
$\overline{\texttt{mixll}}$	-913.83	-913.83	-933.96	-933.96	-1085.71	-1085.71	-1120.95	-1120.95
std	37.87	37.87	50.60	50.60	31.89	31.89	29.35	29.35
`nmixll`	-0.91	-0.91	-0.93	-0.93	-1.08	-1.08	-1.12	-1.12
$\overline{\texttt{ncom}}$	4.0	4.0	3.6	3.6	8.2	8.2	2.0	2.0
$\overline{\texttt{time}}$	3.12	3.16	2.71	2.70	3.32	3.36	0.31	0.31

All options, which contained 'g' failed. It can be concluded that, in this case, the influence of 'a' was not observed. Thus the results are the same in each following selection (in parenthesis) ('a','0')('ad','d')('ag','g') ('adg','dg'). It just means that always the first selected factor in the first selected component was split.

4 static components

Table of results

Option	a	0	ad	d	ag	g	agd	dg
$\overline{\mathtt{mixll}}$	-780.07	-780.07	-803.10	-803.10	-1074.48	-1074.48	-1094.98	-1094.98
std	71.01	71.01	103.28	103.28	48.98	48.98	64.68	64.68
`nmixll`	-0.78	-0.78	-0.80	-0.80	-1.07	-1.07	-1.09	-1.09
$\overline{\mathtt{ncom}}$	3.6	3.6	3.6	3.6	5.4	5.4	3.4	3.4
$\overline{\mathtt{time}}$	5.73	4.94	4.01	3.97	5.14	5.54	1.15	1.22

With these data, the results are practically the same results as the previous ones. The option 'g' does not led to good results.

Creating of common factors has its sense only if the real mixture really had the common factors. But this is not known to the system. It must be given as a prior information or we have to try both variants.

For the next testing, we will throw out all options containing 'g'.

12 static components,5 channels

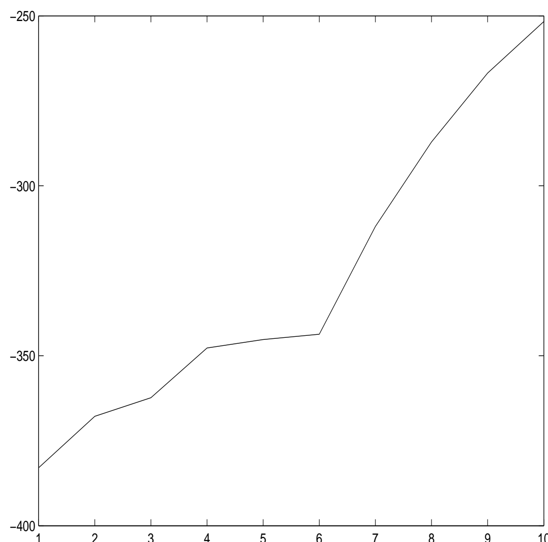
Table of results

Option	0	a	ad	d
$\overline{\mathtt{mixll}}$	-192.86	-233.17	-305.11	-306.01
std	51.44	80.98	73.78	74.53
`nmixll`	-0.19	-0.23	-0.30	-0.31
$\overline{\mathtt{ncom}}$	3.0	2.6	2.8	3.0
$\overline{\mathtt{time}}$	9.40	7.21	2.66	2.77

This case clearly speaks against the use of options 'a' and 'd'. These options can save the time, but the cost we are paying for it is too high.

Danger by using stopping rules

The following picture shows the situation, when the stopping rule would be satisfied, but the `mixll` grows then again.



mixll during iterations

1	122.9168
2	0.3529
3	0.1269
4	0.3318
5	0.0561
6	0.0356
7	0.7189
8	0.5566
9	0.4554
10	0.3378

(relative growth) the statistic offered to the stopping rule

The iteration would stopped if the value of the statistic would be under 0.09575. So if we used stopping rules in this case, the algorithm would stopped in fifth iteration.

The conclusions are clear, but real data will be examined.

real data 1

Table of results

Option	ad	d	a	0
mixll	150281.72	150281.72	150281.72	150281.72
`nmixll`	2.81	2.81	2.81	2.81
ncom	6	6	6	6
time	116.44	117.65	126.99	118.91

Here, all options produce the same result. The algorithms do the same work, so that the elapsed time should be same for all uses.[2] So the unreliability of time measuring is seen again.

real data 2

Table of results

Option	d	0	ad	a
mixll	147373.78	147373.78	131482.76	131482.76
`nmixll`	2.75	2.75	2.46	2.46
ncom	4.0	4.0	3.0	3.0
time	165.87	166.97	11.70	11.70

Here, the saving of time is clear. But we can see that contribution is rather small.

Conclusions

Default values seem to be properly chosen. The options 'a','d' should be used just in the case that there is a short computing time available. We must accept the fact, that it brings worse results.

7.5 Shrinking of noise covariance estimate -`xcove`

This options determines how many percent of `cove` of the original factor will get the new factors during splitting. Default value is 50% The idea to be tested is to use lower value of it, so that the new components will not overlap. We generate 10 realizations of each mixture again and evaluate them. The results is sorted.

7.5.1 7 static components

Dependence of results on xcove

`xcove`	k0.2	k0.4	k0.1	k0.3	k0.7	k0.8	k0.5	k0.6
$\overline{\texttt{mixll}}$	-902.33	-906.07	-906.79	-907.43	-911.41	-914.41	-918.37	-918.68
std	41.23	35.86	38.40	28.81	35.71	33.40	38.60	27.31
`nmixll`	-0.9023	-0.9061	-0.9068	-0.9074	-0.9114	-0.9144	-0.9184	-0.9187
$\overline{\texttt{ncom}}$	4.8	4.8	4.8	5.0	4.0	3.8	4.0	4.0
$\overline{\texttt{time}}$	3.22	3.04	2.48	2.49	2.80	2.55	2.56	2.57

In this case, the option (0.2) is the best, than following options are (0.4;0.1;0.3). The values below 0.5 seems to be better than the others.

[2]By stopping rules, there is an additional time for testing of the rule, but it is too small to consider it.

7.5.2 20 static components

Dependence of results on xcove

`xcove`	k0.2	k0.3	k0.1	k0.4	k0.8	k0.7	k0.6	k0.5
$\overline{\mathtt{mixll}}$	-997.73	-998.76	-1004.66	-1004.89	-1012.86	-1013.78	-1014.45	-1014.82
std	49.62	52.42	45.87	39.63	47.57	48.28	46.29	33.16
`nmixll`	-0.9977	-0.9988	-1.0047	-1.0049	-1.0129	-1.0138	-1.0145	-1.0148
$\overline{\mathtt{ncom}}$	4.9	4.8	4.4	4.4	3.5	3.5	3.5	3.8
$\overline{\mathtt{time}}$	4.73	4.11	3.04	3.25	3.29	3.13	3.12	3.19

Here, the best options are (0.2;0.3)(0.1;0.4). Again, generally values below 0.5 are better.

7.5.3 4 static components

Dependence of results on xcove

`xcove`	k0.4	k0.1	k0.2	k0.7	k0.3	k0.5	k0.6	k0.8
$\overline{\mathtt{mixll}}$	-493.18	-493.19	-493.26	-493.30	-493.72	-493.93	-493.97	-494.18
std	26.67	27.40	26.71	26.49	26.33	26.26	26.21	25.31
`nmixll`	-0.4932	-0.4932	-0.4933	-0.4933	-0.4937	-0.4939	-0.4940	-0.4942
$\overline{\mathtt{ncom}}$	5.0	5.8	5.3	4.5	4.9	4.5	4.3	4.0
$\overline{\mathtt{time}}$	3.36	3.00	3.12	3.02	2.98	3.11	2.95	2.93

In this case, all options gives approximately the same results.

7.5.4 5 channels, 17 static components

Dependence of results on xcove

`xcove`	k0.2	k0.8	k0.1	k0.4	k0.3	k0.7	k0.6	k0.5
$\overline{\mathtt{mixll}}$	-204.03	-207.27	-207.38	-209.93	-215.88	-217.85	-217.92	-240.74
std	53.01	28.33	47.32	42.15	54.07	51.50	45.51	39.92
`nmixll`	-0.2040	-0.2073	-0.2074	-0.2099	-0.2159	-0.2179	-0.2179	-0.2407
$\overline{\mathtt{ncom}}$	3.7	3.2	4.4	3.2	3.2	3.0	3.2	3.3
$\overline{\mathtt{time}}$	8.40	7.00	7.80	7.70	7.63	8.09	6.58	6.77

From the above data samples, the value 0.2 seems to be the best. The other values gives sometimes good results and sometimes not. Generally , we can say, just that the values below 0.5 are better.

7.5.5 real data 1

Dependence of results on xcove

`xcove`	k0.2	k0.3	k0.5	k0.6	k0.1	k0.4	k0.8	k0.7
mixll	150732.8	150300.3	150281.7	150270.6	150142.6	150110.9	149647.1	149416.3
`nmixll`	2.8224	2.8143	2.8139	2.8137	2.8113	2.8108	2.8021	2.7977
ncom	8.0	7.0	6.0	6.0	7.0	6.0	9.0	11.0
time	138.47	103.86	126.06	107.43	107.44	107.92	102.50	121.05

Again, we can state just that the value 0.2 seems to be good.

7.5.6 real data 2

Dependence of results on xcove

`xcove`	k0.4	k0.5	k0.3	k0.6	k0.7	k0.8	k0.2	k0.1
$\overline{\mathtt{mixll}}$	147749.6	147373.7	146598.8	145311.9	145243.8	145081.4	143719.7	143118.8
`nmixll`	2.7665	2.7595	2.7450	2.7209	2.7196	2.7166	2.6911	2.6798
$\overline{\mathtt{ncom}}$	4.0	4.0	4.0	10.0	4.0	4.0	6.0	8.0
$\overline{\mathtt{time}}$	146.32	77.50	112.05	165.99	182.40	191.58	89.15	110.78

Now, the result for option 0.2 is very bad. It rejects the hypothesis about existing a good general value of `xcove`. The following tables just validates this fact.

7.5.7 7 static components

This mixture was studied here before. Now we generate two times 10 independent realizations. The results validate the fact, that no god general value of `xcove` can be found.

Dependence of results on xcove

`xcove`	k0.3	k0.5	k0.4	k0.8	k0.6	k0.7	k0.2	k0.1
$\overline{\texttt{mixll}}$	-899.68	-899.85	-901.27	-903.93	-905.74	-908.88	-912.24	-931.78
std	6.24	14.90	21.93	17.06	9.77	11.07	29.78	42.31
`nmixll`	-0.8997	-0.8999	-0.9013	-0.9039	-0.9057	-0.9089	-0.9122	-0.9318
$\overline{\texttt{ncom}}$	5.2	4.4	5.0	4.4	4.6	4.2	4.8	4.0
$\overline{\texttt{time}}$	2.66	2.88	2.68	2.66	2.71	2.64	2.58	2.52

Dependence of results on xcove

`xcove`	k0.5	k0.2	k0.4	k0.1	k0.6	k0.3	k0.8	k0.7
$\overline{\texttt{mixll}}$	-909.66	-910.17	-910.94	-916.69	-919.15	-923.14	-926.35	-926.62
std	55.45	51.69	51.04	46.47	49.12	54.06	42.31	40.96
`nmixll`	-0.9097	-0.9102	-0.9109	-0.9167	-0.9192	-0.9231	-0.9264	-0.9266
$\overline{\texttt{ncom}}$	5.6	5.0	5.0	5.2	4.6	5.0	4.6	4.2
$\overline{\texttt{time}}$	3.69	3.00	2.68	3.10	2.79	2.75	2.86	2.62

The behaving of the algorithm for 2 sets of 10 realizations of the same mixture shows, that there is no chance to find a generally optimal value.

Conclusions

The general optimal value of `xcove` could not be found. Only one general conclusion can be that the values below 0.5 are better.

Chapter 8

Software aspects

In this chapter, selected software aspects of the project [3] are discussed. The development environment is MATLAB (product of MathWorks) that offers powerfull language, functions and graphical means. Design and debugging of the algorithms is relatively easy.

MATLAB is an interpreted language and it predestinates it to be slow. The aims of the project are to create toolbox for MATLAB and for stand alone applications, too, which will be able to run on all platforms.

The current state is a compromise between these two variants. MATLAB allows that a part of a programme can be written in another programming language and pre-compiled to the form of a dynamic library. By calling this function, MATLAB links the dynamic library into his address space and allows to use the function as other built-in function. This fact bring us two great advantages. The first one is, that algorithms written in MATLAB can be step by step re-implemented in another language. The second one is, that it brings a great speed up. The producers assure double speed-up, but the usual program accelerates on order of ten times. This fact is used by computing time intensive functions, which are compiled and included in the resulting toolbox. However, the compatibility with other operation systems is of course lost and each operating system requires different version of the toolbox.

8.1 MEX files

The compiled files are referred to as MEX-files. They can be written in several programming languages. One of our aims is to create a stand alone application, which will run everywhere. The programming language C seems to be best for this purpose. The C++ has not been selected because its compilers do not exists on some platforms of real time control computer. C compiler si supposed to be present on any platform.

The basic topic in building MEX-Files are inputs and outputs to the file. The programmer has to understand how MATLAB represents data types that supports. It is a single object type referred to as "mxArray". All variables are stored in objects of this type. The variables are matrices (numerical, sparse, multi dimensional), structures and cell matrices. Object oriented programming is supported by different means. In our project, just three types are used, matrix, one-dimensional structure and cell array. The object oriented processing is not used.

Each MEX-file has one function with the name `mexfunction` which is automatically exported from the DLL[1]. It is clear that this function must have a standard interface in the form:

```
void mexFunction( int no, mxArray **out, int ni, const mxArray **in)
```

where `no` is the number of outputs, `out` is the array of pointers to outputs, `ni` is the number of input arguments, `in` is the array of pointers to input arguments.

The type `mxArray` is the only one for all MATLAB objects. The instances of this type cannot be create directly, only with help of special constructors, which returns pointer to the place in memory,

[1]We mean the dynamical linked library, these extension has the library just under Windows.

where the `mxArray` was created. So that the type `mxArray` is never used single but as the pointer to the structure only. Structure of an MEX-file can be generally described as follows:

1. parsing inputs i.e. converting the MATLAB data structures to programming structures;
2. algorithm implementation;
3. outputs building

1. Parsing inputs

Processing of the 3 basic types is described. When we have an `mxArray *` we can recognize whether this object[2] is a matrix or cell array or a struct using the following functions.

`bool mxIsNumeric(mxArray *a)`	returns true if the object is a matrix
`bool mxIsCell(mxArray *a)`	returns true if the object is a cell array
`bool mxIsStruct(mxArray *a)`	returns true if the object is a struct

matrix

By this basic type, we can use functions

- `int mxGetN(mxArray *matr)` returns the number of columns
- `int mxGetM(mxArray *matr)` returns the number of rows
- `double * mxGetPr(mxArray *matr)` returns the pointer to the first element of the matrix. The elements are folded up in one memory block along columns and they are of the type double.

By using these functions, we can acces the matrix element by element. Each element can be changed, of course. The example shows using these functions, in which each element of matrix `matr` is multiplied by 2.

```
mxArray *matr;
double *p;
.....

n=mxGetN(matr);
m=mxGetM(matr);
p=mxGetPr(matr)
for(i=0;i<m*n;i++) p[i]*=2;
```

struct

Struct is handled by

- `int mxGetNumberOfFields(mxArray *str)` returns number of fields
- `double * mxGetFieldN(mxArray *str,int fieldnumber)`[3] returns the `mxArray *`pointer to the field given by the `fieldnumber`. First field has index 0. In header file `mexlib.h` there are defined useful macros for accessing specific fields of the used structures. It should be noted that the returning pointer points to the same place as the pointer inside the struct. No data are copied.

[2]It is not object as in Object Oriented Programming

[3]This function is not standard matlab function it is defined in mexlib.h as follows
`#define mxGetFieldN(X,i) mxGetFieldByNumber(X,0,i)`

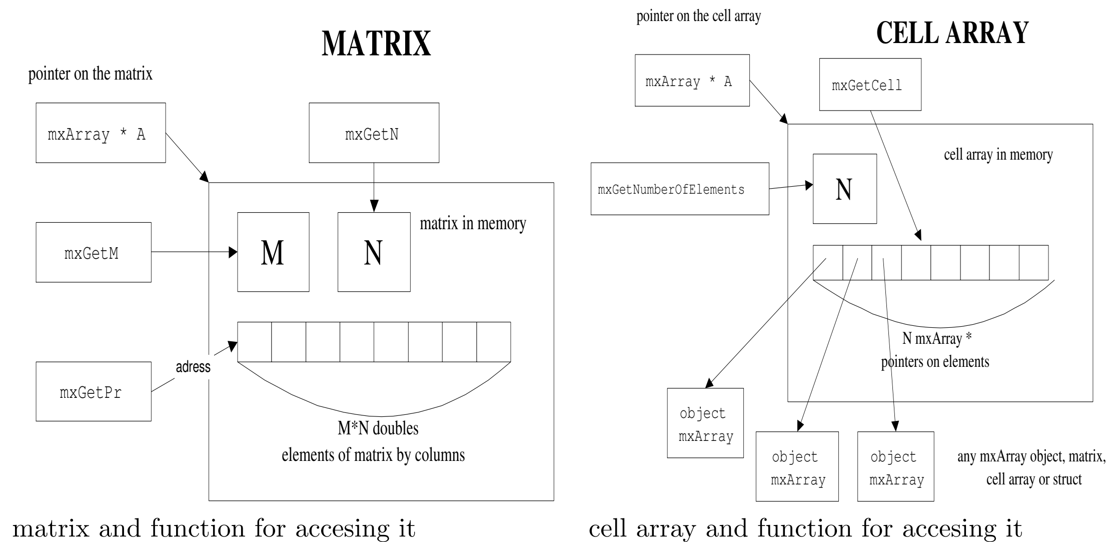
cell array

Working with cell array is almost the same as with the struct. The corresponding functions are

- `int mxGetNumberOfElements(mxArray *cell)` returns number of elements
- `double * mxGetCell(mxArray *cell,int index)` returns the `mxArray *`pointer to the element given by the `index`. First element has index 0. The other behavior is the same as shown for struct.

The following figures show accessing methods and simplified placing of matrix and cell array to memory. The struct is not shown because of its similarity to cell array.

accesing methods



matrix and function for accesing it   cell array and function for accesing it

2. algorithm implementation

We can either convert all data to classical data structures (described in the previous paragraf), or we can directly use the mxArray structure. Using the second variant, the third step need not be partially done. But if we have some earlier implemented algorithms, it is better to convert data and use these algorithms. The aims to create the stand alone application will be easy in all variants. There is no reason not to use mxArray structs in it. Without it, we wont be able to write universals codes for both aims and this requirement is very important. Two versions of each procedure would lead to confusion.

3. output create

Using the constructors, we will create the outputs structures, fill them with data and assign pointers on them in array `out`. The `mxArray` can be created by constructor or by copying another `mxArray`.
`mxArray *mxDuplicateArray(mxArray *array)` will copy entire `mxArray` with all fields and returns pointer on new structure. The description of the constructors for used `mxArrays` is:

matrix

`mxArray *mxCreateDoubleMatrix(int m,int n,0)` creates matrix $m \times n$ with zero elements. The zero in definition does not introduce the zero values. It just means that we want to create **double** matrix. Accesing of elements is described above.

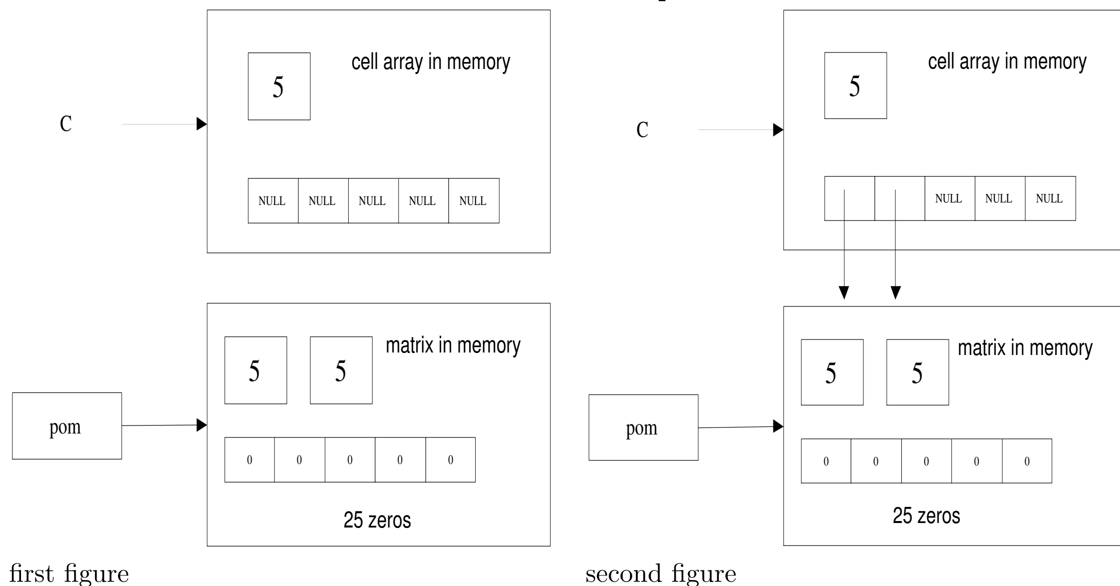
cell array

`mxArray *mxCreateCellMatrix(1,int n)` creates cell array, with n pointers on elements. After creation, the pointers have value NULL. The elements must be specified by calling `mxSetCell(mxArray *array,int i,mxArray *othermxArray)`, where `array` means pointer to our Cell array, `i` means index of the element we are setting, and `othermxArray` is pointer to structure we want to assign to our cell array. This structure will not be copied. We must avoid the situation with assigned pointers of one `mxArray` into two other `mxArrays`.

Example 1:

```
mxArray *C=mxCreateCellMatrix(1,5);
mxArray *pom;
pom=mxCreateDoubleMatrix(5,5,0); {first figure}
mxSetCell(C,0,pom);
mxSetCell(C,1,pom); {second figure}
```

ilustrative example



first figure second figure

The example contains serious error. Two pointers in one cell array points to one data structure. If we want all cells to be the same we must create copies of assigned data structures.

Example 2:

```
mxArray *C=mxCreateCellMatrix(1,5);
mxArray *pom,*pom2;
pom=mxCreateDoubleMatrix(5,5,0);
for(i=0;i<5;i++)
{
 pom2=mxDuplicateArray(pom);
 mxSetCell(C,i,pom2);
}
```

When we have all elements pointers assigned to a value, and we against call mxSetCell, we can loose pointer to some data structures. Correctly, we must destroy the structure before loosing its pointer.

Example 3; We have the cell array from previous example and want to assign to the third cell bigger matrix.

```
mxArray *p=mxGetCell(C,2);
mxArray *bigger=mxCreateDoubleMatrix(10,10,0);
```

```
mxSetCell(C,2,bigger);
mxDestroyArray(p);
```

Struct

The function `mxArray *mxCreateStructMatrix(1,1,int n, const char **names)` creates a struct with n fields. The names of the fields must be specified in `names` , which is array of `char*`. The simplest way to create the names is:

```
const  char r[n][maxlen+1]={"name1","name2", ....,"namen"};
const char **names;
names=mxCalloc(n,sizeof(*names));
for(i=0;i<n;i++) names[i]=r[i];
```

`maxlen` is length of the longest name.

After creating the struct, we must assign the fields pointers. It is functional in the same way as by the cell array. the function here is `mxSetFieldN(mxArray *array,int field,mxArray *othermxarray)`[4] It reads also all notes from cell arrays.

WARNING!!!!

The mex function must not change its inputs!!!!!! It means, that the array `in[]` can be used just for reading. When we change something of this, it can leads to unpredictable results.

The producers now says: Note Inputs to a MEX-file are constant read-only mxArrays and should not be modified. Using mxSetCell* or mxSetField* to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

But, in Matlab versions 3 and 4, the producers adviced changing the inputs. It brings more computting speed. Let us have a very large structure, and we want to change one element. By changing the inputs, we change just this element. Otherwise, the whole structure must be copied. In Matlab 5, the producers said nothing about changing inputs. But it worked. The most part of this project was created on MATLAB 5. By releasing MATLAB 6, the toolbox stopped to work. The results were really unpredictable. Standard MATLAB function `any` did not work. The way to find the error was very hard. We discovered a small mex-file, which changed its inputs. The producers notice did we find out after that on their web site.

compiling

When we have written a MEX - file , we must compile it so that the matlab can use it. It is simply done by calling `mex func.c` on the matlab command prompt. Before this, we must specify to the matlab, which compiler should it use. It is done by `mex -setup`

8.2 Library, calling functions

We have a lot of m-files disposable, which calls each other. For each file `func.m` we create file `func.c`, which has to allow compiling of the considered mex firstly, and calling this function from other c-files secondly.

Writing a mex file is described in previous section. Solving of the second task is done by creating a library of functions `prodact.lib` All used functions must be compiled and linked into this library. The format of functions, which makes the same computing as the m-file, must be uniform. When another programer want to use my c-file, he must not be forced to find out the declaration of the function. Easy and usefull solution is to to declare these functions formally in the same form as the mexFunction, just the name of the function corresponds to name of the m-file. It means

`void func( int no, mxArray **out, int ni, const mxArray **in)`. The format of auxiliary functions depends just on the programer. The programmers is not forces to use these functions.

[4] This function is not standard matlab function it is defined in mexlib.h as follows
`#define mxSetFieldN(X,i,X1) (mxSetFieldByNumber(X,0,i,X1))`

To allow to a file to solve two different task, we must to specify the task by compiling. If we want to compile the c-file into the library, we define macro `LIBRARY`. In the file, of course we must use this information. Concretely, each c-file should contain:

```
#include <math.h>
#include "mex.h"     /*header file, which must contained all mex*/
#include "mexlib.h"  /*header file with defined macros, which are often used*/

/*=====================================================
function r = func(params)   header of original m-file
%
%help of the m-file
  ----------------------------------
  programmed: name of the programmer
=====================================================*/


declarations of auxiliary functions
declarations of extern functions

#ifndef LIBRARY

void mexFunction( int no, mxArray **out, int ni, const mxArray **in)

#else

void func( int no, mxArray **out, int ni, const mxArray **in)

#endif
 {
   body of the main function
 }


#ifdef LIBRARY
  definition of  auxiliary functions
#endif
```

When a dll is compiled and linked for MATLAB, the programmer must tell to the compiler to use the library `product.lib` and, of course, all auxiliary functions used must be included in the library. The compile command than looks as follows. `mex func.c prodact.lib`

8.3 stand-alone application and memory management

When all m-files are converted to c-files like in prev. section, there should be no problem to link a stand-alone application from the library. But in the history, there were problems with classical memory management realized by malloc. It is slow and programmers errors occur frequently by wrong deallocation of the memory.

Because of this a stack based model was selected. The application allocates a large block of memory called WORKSPACE and sets a workspace pointer WP to point to the begining of this block. All memory allocations are stack-wise defined, so that they returns WP and then move WP forwards in the corresponding length. A next allocation becomes another piece of the WORKSPACE. All destructors are overdefined to do nothing. A deallocation can free just the memory, which was allocated as last one.

It is done simply by moving WP back to the value, which he has before the allocation. The mechanism of allocation and deallocation must be considered when the algorithm is designed. Some functions are called iteratively and could soon use all WORKSPACE, if the memory was not cleared. There is a simple solution, which was designed and implemented in this work. An extra auxiliary workspace is used. Before calling the dangerous function, the application switch all allocations to be from the auxiliary WORKSPACE. Then, after finishing the function, the result is coppied to WORKSPACE and the auxiliary one is deallocated (ie. WP2 is set to the beginning.) Disadvantage of this approach is, that the nesting of this memory switching has no equivalent in normal memory management.

Chapter 9

Conclusions

In this chapter, results achieved are summarized. Generally, it contributed to a conversion of an experimental version of `mixinit` into real working tool. Moreover, reasonable set of default values of the optional parameters was modified according to results achieved. A critical one (xcove) was also singled out. Specific achievements worth of naming are.

- A conversion of the Mixtools functions for use with MATLAB ver. 6 (see section 8.1).
- The detailed description of algorithm for initialization of mixture estimation has been elaborated (see section 5.2) and the particular processing options were discussed (see section 5.1).
- In chapter 7 the behaviour of the algorithm in dependency of these parameters was studied. A particular attention was paid to the time requirements of the individual options. The best selection of options was outlined.
- During experiments, some errors were found in the current version of the mixinit algorithm. These were corrected.
- Many Mixtools m-function, has been implemented in language C as MEX-functions. It has led to a higher efficiency of the toolbox. Particularly, all functions, which are used in function mixinit, were transferred to MEX files.
- The first version of mixinit in C was created and tested.
- The chapter 8 can serve as a manual in programming of Mixtools functions in C. It can be used as a reference manual for the toolbox designers, too. The programmers conventions for working in Mixtols were described.
- In the section 8.3 the new memory model for stand-alone application was designed and implemented.

What remains to be done!

The algorithm `mixinit` is very time-consuming. The computing should be parallelized, so that the time for running it on a more-processors-computer will decrease. The toolbox should be tried on multi dimensional real data. One run of `mixinit` with great data last more than one day. It is reasonable to develop a mean, so that we can interrupt the computing, if we want to do with the computer something else. The mysterious parameter `xcove` should be studied more in detail than in this paper.

Index

batch quasi-Bayes estimation, 11

constructor, 45

data channels, 15
DATA., 15
discrete time, 9
dynamic factors, 16

EM estimation , 11

factor offset, 15
factor splitting, 13
flattening mapping, 12

horizon, 9

KL distance, 9
Kullback-Leibler distance, 9

mxArray, 43
mxCreateCellMatrix, 46
mxCreateDoubleMatrix, 45
mxCreateStructMatrix, 47
mxDuplicateArray, 45
mxGetCell, 45
mxGetFieldN, 44
mxGetM, 44
mxGetN, 44
mxGetNumberOfElements, 45
mxGetNumberOfFields, 44
mxGetPr, 44
mxIsCell, 44
mxIsNumeric, 44
mxIsStruct, 44
mxSetCell, 46
mxSetFieldN, 47

niter, 17

posterior estimate, 12
prior pdf, 12

quasi-Bayes estimation, 11

static factor, 16

TIME, 15

xcove, 17

Bibliography

[1] James O. Berger. *STATISTICAL DECISION THEORY AND BAYESIAN ANALYSIS*. Springer-Verlag, New York, 1985.

[2] MathWorks col. *MATLAB THE LANGUAGE OF TECHNICAL COMPUTING*. The MathWorks, Inc., Natick, 1998.

[3] M. Kárný, J. Böhm, T.V. Guy, L. Jirsa, A. Kanouras, I. Nagy, P. Nedoma, A. Quinn, L. Tesař, D. Parry, M. Tichý, and M. Valečková. *PRODACTOOL BACKGROUND – THEORY, ALGORITHMS AND SOFTWARE*. 2001. Draft of the report, 260 pp.

[4] Ivan Nagy Miroslav Kárný. *DYNAMIC BAYESIAN DECISION-MAKING*. UTIA AV, 1999.

[5] P. Nedoma, M. Kárný, I. Nagy, and M. Valečková. *MIXTOOLS. MATLAB TOOLBOX FOR MIXTURES*. Number 1995. Prague, 2000.

[6] Karel Pala. *ÚVOD DO SYSTEMU LATEX*. Ediční středisko ČVUT, Praha, 1990.

[7] V. Peterka. *BAYESIAN SYSTEM IDENTIFICATION*. Pergamon Press, Oxford, 1981.

[8] B. Stroustrup. *THE C++ PROGRAMMING LANGUAGE*. Addison-Wesley, New York, Bonn, Tokyo, 1991.