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., Andrýsek J.

Mixtools

Application Program Interface

User's Guide

No. 2088 September 2003

Project GA ČR 102/03/0049, AV ČR S1075351 and S1075102

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

Aplication Program Interface

User's Guide

Contents

Acknowledgment

The work described in this report was done with support of GA CR 102/03/0049, AV CR S1075351, S1075102 and EU project IST ProDaCTools No. IST-1999-12058

1 Introduction

This section contains general considerations related to the approach of MATLAB MEX-functions, MATLAB Application Program Interface (API) and their use and modification in Mixtools.

The general MATLAB support is supposed to be known, the basic MathWorks texts available on distribution documentation CD are "External Interfaces" and "External Interfaces/API Reference"

1.1 MATLAB MEX-functions

MEX stands for MATLAB executable. MEX-functions are dynamically linked subroutines produced from C (or Fortran) source code that, when compiled, can be run from within MATLAB in the same way as MATLAB M-function or built-in functions. The Application Program Interface (API) functions provide functionality to transfer data between MEX-functions and MATLAB, and the ability to call MATLAB functions from C (or Fortran) code.

The main reasons to write MEX-functions are:

- the ability to call large existing C or FORTRAN routines directly from MATLAB without having to rewrite them as M-functions;
- speed - the bottleneck computations (like for-loops) can be rewritten as a MEX-function for efficiency;
- MEX-function can contain code necessary to connect MATLAB with a controlled real process.

From MATLAB version 6, the parsed code (internal representation of M-functions) reaches MEX-functions in speed. It means that the necessity to convert MATLAB M-functions to MATLAB MEX-functions looses importance.

1.2 Mixtools MEX-functions

The Mixtools toolbox contains more than 150 M-functions. Eighty of them has been converted into MEX-functions.

The recursive processing done by M-functions in Mixtools is very slow but the Mixtools MEX-functions can be 20 to 50 times faster. It is enabled by simplicity of Mixtools MEX interface that avoids calling of MATLAB services.

All Mixtools MEX-functions have M-function equivalent. It means that the MEX-functions are easy to be programmed. The M-version is not dependent on MATLAB version - it can be used for education. When moving Mixtools to a different platform, the M-version can be used until the MEX version is developed.

1.3 Mixtools API

The API supplied by MATLAB can be substituted by a (reduced) equivalent and used in stand-alone (MATLAB independent) ANSI C-coded environment. Codes of Mixtools MEX-functions can be used in the stand-alone application without any code change. This offers the possibility to get rid of the MATLAB dependency in real time applications.

The approach enables to write C-encoded programs without any use of MATLAB. The programming is easy and well described in MATLAB manuals.

The toolbox functions can be programmed and debugged in MATLAB comfortable environment and then, without any change of code, used in a MATLAB independent program.

The code runs on any platform because a very simple ANSI C code is used. No attempt has been made to support any graphics. Access to The code can be employed in a non MATLAB experimental environments (e.g. Scilab, Octave etc.)

The Mixtool philosophy is:

- initialization, learning, data analysis etc. is done in MATLAB environment;

- real time data processing is done outside MATLAB in stand-alone application;
- results are visualized in MATLAB comfortable environment.

This is supported by functions that transform MATLAB objects to stand-alone application and back.

ANSI C is employed in programming because the only structure used in API is *mxArray*. There are applications that cannot use C++ (special processors). Students that contribute to the toolbox often know only basic ANSI C.

An alternative way is use of MATLAB Compiler. The MATLAB compiler does not accelerate MATLAB processing too much (from version 6). The code generated is too complex and unreadable. Each MATLAB platform needs own libraries, the solution is expensive. It is almost impossible to write MATLAB independent programs in it. There is little experience with the MATLAB compiler.

The MATLAB compiler is expected to be employed to transfer various MATLAB GUI into stand-alone programs.

intro.tex by PN September 30, 2003

2 Mixtools MEX-functions

The MEX-functions can be called from any other MEX-function and from any stand-alone application program. The only data structure is *mxArray* - a special structure that contains MATLAB data (dimensions, data values etc.)

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" (listed in Appendix) contains functions prototypes and descriptive information.

The structure of MEX-functions is documented 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" , see Note 1
{
   #include "bldlmex.c"      // interface to MATLAB globals, see Note 2
   mexfun( ... );            // call MEX-function in library, see Note 3
}
#else                        // compiled if LIBRARY is defined
void mexfun( ... )           // maintained in library as "mexfun"
{
   // MEX - file interface
   ...
}

mxArray * mexfun1(...)       // access functions stored in prodact.lib, see Note 4
{                            // prototypes are transferred to mexlib.h
}
#endif                       // end processing of library code
```

Notes:

1. The arguments of the *mexFunction* are in this order:

```
int no,                     // number of outputs
mxArray *out[],             // array of pointers to outputs
int ni,                     // number of inputs
const mxArray *in[]         // array of pointers to inputs
```

The argument *in* is declared as *const*. This means that the values that are passed into the MEX-file should not be altered. Doing so can cause segmentation violations in MATLAB.

The outputs values in *out* are invalid when the MEX-file begins. The mxArrays they contain must be defined in the MEX-file otherwise segmentation violations occurs.

2. The *bldlmex.c* is interface to Mixtools global values used for data management. 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
mxArray *XDATA;             // data matrix for stand-alone C programs
mxArray *XTIME;             // TIME definition for stand-alone C programs
```

3. All Mixtools MEX-functions are held in *prodact.lib* under their names (in *API.lib* for stand-alone applications). Prototypes are defined in *mexlib.h*.

4. Auxiliary functions are created and stored in library *prodact.lib*. Prototypes are in *mexlib.h* listed in Appendix.

3 Mixtools Application Program Interface

The stand-alone application programs can

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

3.1 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';
mixdump(Mix, filename);
...
Mix = restore(filename);
```

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

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

3.2 Programming

In Mixtools API, a function can change the inputs if they are not passed to MATLAB. This is indicated by zero value of *no* and explicitly mentioned in *mexlib.h*.

The MATLAB MEX-functions are written using functions of MATLAB API. The API functions are substituted by an independent API encoded designed for Mixtools. The definitions are held in header file "mex.h".

The *mxArray* is a special structure containing representation of a MATLAB array. The mxArray structure is not know in MATLAB, its fields are accessed by API functions only.

In Mixtools API environment, the *mxArray* is known. However, a direct access to its fields is not recommended.

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 MATLAB descriptions 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)	

3.3 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 freeing of memory is required in the function body.

3.4 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.

3.5 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
```


4 Examples of programming with Mixtools API

4.1 API example: quasi-Bayes repeated estimation

This example consist of

- data sample is generated in MATLAB and dumped into file *data*;
- initial mixture is build in MATLAB and dumped into file $mix0$;
- quasi-Bayes estimation is done in MATLAB for later comparison with results of stand-alone processing;
- stand-alone corresponding processing is done and dumped into file mix.
- resulting mixtures are compared and displayed in Fig. 1.

The MATLAB script is

```
% example: standalone API programming
%
% Autor  : P. Nedoma
% Updated: September 2003
% Project: GA CR 102/03/0049, AV CR S1075351, S1075102, DESIGNER

prodini;
randn('seed', 321);                   % fix realization
ndat   = 2000;                        % size of data sample
ncom   = 3;                           % number of components
cove   = ltdl(0.2*[1 0.1; 0.1 1]);    % common component covariance
Sim    = statsim(ndat, ncom, cove);   % generate data sample

mixdump(DATA, 'data');                % dump DATA

Mix0   = genmixe(ncom);               % initial mixture
```

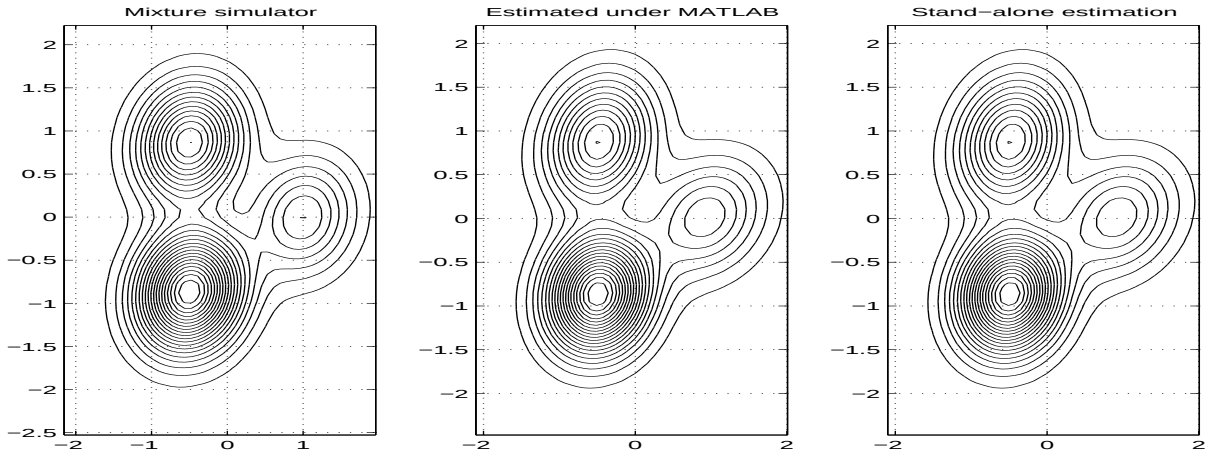

Figure 1: Quasi-Bayes estimation

subplot	*contains*
1	mixture simulator
2	mixture estimated in MATLAB
3	mixture estimated in stand-alone program

```
Mix0   = mix2mix(Mix0, 21);            % convert into convenient form
mixdump(Mix0, 'mix0');                 % dump initial mixture

% estimation under MATLAB
frg    = 1;                            % no forgetting
niter  = 20;                           % number of iterations
Mix    = mixestqb(Mix0,frg,ndat,niter);% quasi-Bayes repeated estimation

... plot simulator
... plot mixture estimated in MATLAB

% calling stand-alone main program equivalent to estimation above
% results are dumped in "mix"
!main

Mix1  = restore('mix');                % load results
Mix.states.mixll - Mix1.states.mixll   % compare log-likelihood
ans =
 -6.3665e-012

... plot mixture estimated in stand-alone program
```

The C-coded program *main* is:

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

void main(void)
```

```
{  mxArray *Mix0 = mixload("mix0");       // load initial matrix
   mxArray *frg, *ndat, *niter;
   mxArray *out[1];
   const mxArray *in[4];

   XDATA = mixload("data");               // load data sample
   XTIME = mxCreateDoubleMatrix(1,1,0);
#include "bldlmex.c"

// frg  = 1;                             % forgetting rate
// ndat = 1000;                          % size of data sample
// niter= 20;                            % number of iterations
   frg   = mxCreateScalarDouble(1);
   ndat  = mxCreateScalarDouble((double)mxGetN(XDATA));
   niter = mxCreateScalarDouble(20);

// 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);

// resuls saved as "mix"
   mixsave(out[0], "mix");

}
```

estqb.tex by PN September 30, 2003

4.1.1 API example: buffered quasi-Bayes etimation

With large data samples the Mixtools uses buffered processing. It means that at a time, only a portion of data sample is loaded and processed.

The example is similar to the previous one but buffered data processing is used. The data sample must be entered as binary file. It is done by:

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

The processing from the previous section is modified as:

```
Ndat  = {'data', 2};                  % alternative specification of "ndat"
DATA  = zeros(2,150);                 % allocate buffer
Mix   = mixestqb(Mix0, frg, Ndat, niter);% estimation
```

The main stand-alone program is

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

void main(void)
```

```
{  mxArray *Mix0 = mixload("mix0");    // load dumped mixture
   mxArray *out[1];
   const mxArray *in[4];
   mxArray *frg = mxCreateScalarDouble(1);

   mxArray *filename = mxCreateString("data"); // name of disk file
   mxArray *mdat     = mxCreateScalarDouble(2);
   mxArray *Ndat     = mxCreateCell(2);     // create cell argument
   mxSetCell(Ndat, 0, filename);            // set individual cells
   mxSetCell(Ndat, 1, mdat);

// build DATA, TIME
   XDATA = mxCreateDoubleMatrix(2, 150, 0); // buffer for data reading
   XTIME = mxCreateDoubleMatrix(1,1,0);     // processing TIME
   #include "bldlmex.c"

// buffered mixestqb
   in[0] = Mix0;                        // filling arguments
   in[1] = frg;                         // forgetting rate
   in[2] = Ndat;
   in[3] = mxCreateScalarDouble(20);    // number of iterations
   mixestqb(1, out, 4, in);             // iterative quasi-Bayes processing

// resuls
   mixsave(out[0], "mix1");             // dump result
}
```

The last processing calls the main stand-alone program *main* and compares the estimated mixtures. The result of stand-alone estimation is dumped into $mix1$ file.

```
!main
Mix1  = restore('mix1');               % load results
Mix.states.mixll - Mix1.states.mixll   % compare log-likelihood
ans =
 -6.3665e-012
```

bufqb.tex by PN September 30, 2003

4.1.2 API example: stand-alone on-line processing

Recursive estimation by *mixestim* is exposed. The data and initial mixture are dumped as in previous section:

```
randn('seed', 321);                    % fix realization
ndat   = 2000;                         % size of data sample
ncom   = 3;                            % number of components
cove   = ltdl(0.2*[1 0.1; 0.1 1]);     % common component covariance
Sim    = statsim(ndat, ncom, cove);    % generate data sample

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

Mix0   = genmixe(ncom);                % initial mixture
```

```
Mix0   = mix2mix(Mix0, 21);            % convert into convenient form
mixdump(Mix0, 'mix0');                 % dump initial mixture
```

The on-line processing will be simulated as a recursive one. Under MATLAB, it means:

```
frg   = 1;
Mix = Mix0;
for TIME = Mix0.states.maxtd+1 : ndat
    Mix  = mixestim(Mix, frg);         % recursive mixture estimation
end
```

The *main* program is called and both mixtures are compared:

```
!main                                  % call stand-alone program
Mix1  = restore('mix');                % restore resulting mixture
equal(Mix, Mix1, 1e-11)                % the mixtures are equal
ans =
      1
```

The stand-alone program is coded as:

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

void main(void)
{  mxArray *Mix  = mixload("mix0");    // load initial mixture
   mxArray *frg  = mxCreateScalarDouble(1); // forgetting rate
   FILE    *file = fopen("data","rb"); // data will be read from file
   mxArray       *out[1];
   const mxArray *in[3];
   int    wspmemo;


// build DATA, TIME
   XDATA = mxCreateDoubleMatrix(2,1,0);// data buffer
   XTIME = mxCreateScalarDouble(1);    // processing time
   #include "bldlmex.c"

   in[1]   = frg;
   wspmemo = wsp;                      // save workspace pointer

  for(;;)                              // on-line processing
   {  fread(DATA, sizeof(double), 2, file); // get data column
      if (feof(file)) break;
      in[0] = Mix;
      mixestim(1, out, 2, in);         // mixture estimation
      space(1, out, wspmemo);          // make memory compact
      Mix = out[0];
   }
   fclose(file);
// resuls are saved
   mixsave(Mix, "mix");
}
```

5 Appendices

5.1 Mixtools quick reference

5.1.1 Mixtools Reference

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 = ...`	initialization of mixture estimation[a]
`mixinit(Mix0,frg,ndat,niter,opt,belief)`	
`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]	
`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] defaults: see Prediction operations. The functions allows definition of grid densities and ranges

[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 = ...`	estimate structure of a factor
`facstrid(Fac,Fac0,belief,nbest,nruns)`	

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 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)`	gennerate model structure of a 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 September 30, 2003

5.1.2 Mixtools Cryptonyms

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]
`facllls`	trial factor predictions $log(f(d_{t+1}\|fac, t+1))$
`comllls`	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
`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

5.1.3 Mixtools codes

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	for predictor types

5.2 List of Mixtools MEX-functions

algen	aloptim	arx2arx	arxc2can	arxc2fac	can2arxc
can2marg	com2can	com2com	com2pro	comarx	comarxls
diff_tg	fac2arxc	fac2ls	facarx	facarxls	facchng
facdpred	facflat	facfrg	facgmean	facsort	facstrid
facupdt	facvll	getdvect	inisyn	isdimeq	kldiscom
kldist	ld2ld	ld2ls	ld2v	ldinv	ldperm
ldupdt	ls2ld	mix2mix	mix2mixm	mix2pro	mixconst
mixcopy	mixdfms	mixest	mixestbb	mixestbq	mixestfe
mixestim	mixestmt	mixestqb	mixflat	mixflatv	mixfrg
mixgmean	mixgrid	mixpro	mixsimul	mixstats	noise
preaux	preaux1	pro2pre	pro2str	profix	profixn
protest	ricexp	ricpen	ricpenu	ricshift	soptim
statgrid	straux1	strmax	subsplix	synmixi	ud2ld
udform	udupdt	ufcgen	utinv		

5.3 Mixtools MEX header

```
/* @mex.h - Application Program Interface
   Project: GA CR 102/03/0049, AV CR S1075351, S1075102
*/
```

```
#define MAXPSI (500) /* maximum regressor length*/
#define MAXFAC (500) /* maximum factors or components*/
#define MAXCOM (500) /* maximum number of components */
#define MAXCHN (100) /* maximum number of channels in mixture */

/*#ifdef MATLAB65*/
  #undef  mexGetArrayPtr
  #define mexGetArrayPtr(a,b) mexGetVariablePtr((b),(a))
/*#else
  #define mexPutVariable(a,b,c) {mxSetName((c),(b));mexPutArray((c),(a));}
#endif
*/

#ifndef LIBRARY
   double        *DATA;          /* global data*/
   int            M_DATA;        /* DATA rows*/
   int            N_DATA;        /* DATA columns*/
   double        *TIME ;         /* simulation itime*/
   double         PSI[MAXPSI];   /* global extended regressor*/
   mxArray       *XDATA;         /* data matrix for standalone C*/
   mxArray       *XTIME;
#else
   extern double *DATA;
   extern int    M_DATA;
   extern int    N_DATA;
   extern double *TIME;
   extern double PSI[1];
   extern mxArray*XDATA;
   extern mxArray*XTIME;
#endif

/* basic allocation/load constants*/
#define MIX_FIELDS       (5)      /* No. of mixture fields*/
#define STATES_FIELDS    (14)     /* No. of states fields*/
#define FAC_FIELDS       (5)      /* No. of factor fields*/

/* description of factor fields*/
   #define Fac_ychn    (0)
   #define Fac_str     (1)
   #define Fac_dfm     (2)
   #define Fac_type    (3)
   #define Fac_LD      (4)
   #define Fac_states  (5)

/* description of factor LS fields, ARX component uses ychns */
   #define LS_ychn    (0)
   #define LS_ychns   (0)
   #define LS_str     (1)
   #define LS_dfm     (2)
   #define LS_type    (3)
   #define LS_cove    (4)
   #define LS_Eth     (5)
   #define LS_Cth     (6)
   #define LS_states  (7)

/* description of  mixture fields*/
   #define Mix_Facs    (0)
   #define Mix_coms    (1)
   #define Mix_dfcs    (2)
   #define Mix_type    (3)
   #define Mix_states  (4)

/* fields of component type */
   #define Mix_Coms    (0)
   #define Mix_Cdfcs   (1)
   #define Mix_Ctype   (3)
   #define Mix_Cstates (4)

/* description of estimator states */
   #define states_mapping (0)
   #define states_modelled (2)
   #define states_notmodelled (3)
   #define states_facs    (4)
   #define states_maxtd   (5)
   #define states_dfm0    (6)
   #define states_dfcs0   (7)
```

```
/* description of statistics */
   #define states_comlls  (8)
   #define states_mixll   (9)
   #define states_comwgs  (10)
   #define states_facwgs  (11)
   #define states_faceps  (12)
   #define states_workspace (13)


/* simulator states */
   #define states_cumprob   (8)
   #define states_Facs      (9)
   #define states_active    (10)
   #define states_sortcoms  (1)


/* canonical mixture */
   #define Mixc_Cans   (0)
   #define Mixc_dfcs   (1)
   #define Mixc_type   (2)

/* states of 122  mixture */
   #define Mix122_modelled    (0)
   #define Mix122_notmodelled (1)
   #define Mix122_predicted   (2)
   #define Mix122_incondition (3)
   #define Mix122_zerodelayed (4)
   #define Mix122_comaux      (5)

/* control design */
#define states_pstr   (0)
#define states_pEth   (1)
#define  states_strc  (6)
#define  states_outs  (7)
#define  states_kc    (8)
#define  states_ufc   (9)
#define  states_UDc  (10)
#define  states_udca (11)
#define  states_kca  (12)
#define  states_uchn (13)
#define  states_pochn (14)

/* filters */
#define Filter_str  (0)
#define Filter_type (3)


/* Mexlib functions prototypes*/
/*double facupdt(mxArray *Fac, double *workspace);  factor update, return prediction*/
double gammaln(double);                          /* = gammaln.m*/
int    maxdelay(mxArray *Mix);                   /* max. delay in all factors*/
double redultx(double*, double *,double *r, double*,int *, int *, int *);
void   reformx(double *,double *,double *,int *,int *,int *);
double sumArray(mxArray*);             /* sum of array elements*/

/* macros*/
#ifndef API
#define error              mexErrMsgTxt
#define mxGetFieldN(X,i)   mxGetFieldByNumber(X,0,i)
#define mxGetFieldPr(X, i) (mxGetPr(mxGetFieldByNumber(X, 0, i)))
#define mxGetCellPr(X, i)  (mxGetPr(mxGetCell(X,i)))
#define LEN(X)             (mxGetM(X)*mxGetN(X))
#define mxSetFieldN(X,i,X1) (mxSetFieldByNumber(X,0,i,X1))
#endif
#define eps (2.22045e-016)  /* used to prevent frozen run */
#define mxReplaceFieldN(X,i,X1) {mxArray *CopyX=(X),
       *CopyX1=(X1),*Old;int Copyi=(i);Old=mxGetFieldN(CopyX,Copyi);
       if(Old)mxDestroyArray(Old);mxSetFieldN(CopyX,Copyi,CopyX1);}
#define mxReplaceCellN(X,i,X1) {mxArray *CopyX=(X),
       *CopyX1=(X1),*Old;int Copyi=(i);Old=mxGetCell(CopyX,Copyi);
       if(Old)mxDestroyArray(Old);mxSetCell(CopyX,Copyi,CopyX1);}
#define mxReplaceCell(X,i,X1) mxReplaceCellN(X,i,X1)

/* MEX functions prototypes  */
void     arx2arx (int, mxArray**, int, const mxArray**);
mxArray *arx2arx1(mxArray *Com);
```

```
void     arxc2can(int, mxArray**, int, const mxArray**);
void     arxc2fac(int, mxArray**, int, const mxArray**);
mxArray *arxc2fac1(mxArray *Com);
void     can2arxc(int, mxArray**, int, const mxArray**);
mxArray *can2arxc1(mxArray *);
void     can2marg(int, mxArray**, int, const mxArray**);
void     com2com (int, mxArray**, int, const mxArray**);
mxArray *com2com1(mxArray *Com, int type);
void     com2pro (int, mxArray**, int, const mxArray**);
void     comarx  (int, mxArray**, int, const mxArray**);
mxArray *comarx1 (mxArray *ychns, mxArray *str);
void     comarxls(int, mxArray**, int, const mxArray**);
mxArray *comarxls1(mxArray *ychns,mxArray *str);
void     defaults(int, mxArray**, int, const mxArray**);
void     defaults1(char def,double *a,double *b, double *c);
void     dispv(char *, double *, int);
void     fac2arxc(int, mxArray**, int, const mxArray**);
mxArray *fac2arxc1(mxArray *Fac);
void     fac2ls  (int, mxArray**, int, const mxArray**);
void     fac2ls1(mxArray *Fac,mxArray **qEth,mxArray **qCth, mxArray **qcove, double *qdfm);
void     facarx  (int, mxArray**, int, const mxArray**);
mxArray *facarx1(int qychn,mxArray *qstr,mxArray *qLD,double qdfm);
void     facarxls(int, mxArray**, int, const mxArray**);
mxArray *facarxls1(int qychn,mxArray *qstr,double qcove,
                   mxArray *qEth, double qdfm, double dia);
void     facdpred(int, mxArray**, int, const mxArray**);
void     facflat( int no, mxArray **out, int ni, const mxArray **in);
void     facflat1(mxArray *Fac, mxArray *FacA, double rate);
void     facfrg(int, mxArray**, int, const mxArray**);
void     facfrg1(mxArray *Fac, double rate, mxArray *Faca);
void     facmean(int, mxArray**, int, const mxArray**);
void     facgmean1( mxArray *Fac, mxArray *Fac1, mxArray *Fac2,
                    double lambda, double lambda1);
void     facsort (int, mxArray**, int, const mxArray**);
void     facstrid(int, mxArray**, int, const mxArray**);
void     facvll( int no, mxArray **out, int ni, const mxArray **in);
double   facvll0(double *pLD, double *dfm, int nPsi);
double   facvll1(mxArray *Fac);
double   facvll2(mxArray *Fac1, mxArray *Fac2);
void     facupdt (int, mxArray**, int, const mxArray**);
double   facupdt1(const mxArray *Fac, double weight);
double   facupdt2(const mxArray *Fac, double weight, double *workspace);
void     getdvect(int, mxArray**, int, const mxArray**);
int      getdvect1(mxArray*Fac);
void     getrgr(double*str, int len);
void     iterplot1(mxArray *Mix, mxArray *Mix0, int iter, int itime);
void     kldiscom(int, mxArray**, int, const mxArray**);
void     kldist  (int, mxArray**, int, const mxArray**);
void     ld2ld   (int, mxArray**, int, const mxArray**);
void     ld2ld1(mxArray *LD0,mxArray **qLD,mxArray **qD); /* parameters strs and str not supported*/
void     ld2ls   (int, mxArray**, int, const mxArray**);
void     ld2ls1(mxArray *LD0,double dfm,int ny,mxArray **qEth,mxArray **qCth,mxArray **qcove);
void     ld2v    (int, mxArray**, int, const mxArray**);
mxArray *ld2v1(mxArray *);
void     ldinv   (int, mxArray**, int, const mxArray**);
void     ldinv1(mxArray *LD0,mxArray **qLD,mxArray **qD);
void     ldupdt  (int, mxArray**, int, const mxArray**);
double   ldupdt1( mxArray *LD, mxArray *dvect, double weight);/*in place*/
double   ldupdt2(double *LD, double *dvect, double weight, int nPsi);
void     ls2ld   (int, mxArray**, int, const mxArray**);
mxArray *ls2ld1(mxArray *Eth,mxArray *Cth,mxArray *cove,double dfm, mxArray **qD);
void     metropol(int no, mxArray **out, int ni, const mxArray **in);
void     mix2mix (int, mxArray**, int, const mxArray**);
mxArray *mix2mix1(mxArray *Mix, int typ);
void     mix2pro (int, mxArray**, int, const mxArray**);
void     mixcon1 (int, mxArray**, int, const mxArray**);
void     mixconst(int, mxArray**, int, const mxArray**);
void     mixcon1 (int, mxArray**, int, const mxArray**);
mxArray *mixcon11(mxArray *Coms, mxArray *dfcs);
void     mixdfms (int, mxArray**, int, const mxArray**);
void     mixdfms1( mxArray *Mix,double *s, double *s0);
void     mixest  (int, mxArray**, int, const mxArray**);
void     mixestbb(int, mxArray**, int, const mxArray**);
void     mixestbq(int, mxArray**, int, const mxArray**);
void     mixestim(int, mxArray**, int, const mxArray**);
void     mixestqb(int, mxArray**, int, const mxArray**);
void     mix2mixm(int, mxArray**, int, const mxArray**);
```

```
void     mixfix  (int, mxArray**, int, const mxArray**);
void     mixflat (int, mxArray**, int, const mxArray**);
void     mixflatv(int, mxArray**, int, const mxArray**);
void     mixfrg  (int, mxArray**, int, const mxArray**);
void     mixfrg1(mxArray *Mix, double rate, double rate1, mxArray *MixA);
void     mixgmean(int, mxArray**, int, const mxArray**);
void     mixgmean1(mxArray *Mix,mxArray *Mix1, mxArray* Mix2,
         double lambda, double lambda1);
void     mixgrid (int, mxArray**, int, const mxArray**);
void     mixpro  (int, mxArray**, int, const mxArray**);
void     mixstats(int, mxArray**, int, const mxArray**);
void     mixupdt(int, mxArray**, int, const mxArray**);
void     mixupdt1(mxArray *, int);
void     profix  (int, mxArray**, int, const mxArray**);
void     profixn (int, mxArray**, int, const mxArray**);
void     statgrid(int, mxArray**, int, const mxArray**);
void     strmax  (int, mxArray**, int, const mxArray**);
void     ud2ld(  int,  mxArray**, int, const mxArray**);
void     ud2ld1(mxArray *UD0,mxArray **qU,mxArray **qD);
void     udform  (int, mxArray**, int, const mxArray**);
void     udupdt  (int, mxArray**, int, const mxArray**);
void     udupdt1(mxArray *UD,mxArray *dvect,double weight);
void     utinv   (int, mxArray**, int, const mxArray**);
void     protest (int, mxArray**, int, const mxArray**);
void     mixcopy(mxArray *source, mxArray *target);
void     algen   (int, mxArray**, int, const mxArray**);
void     inisyn  (int, mxArray**, int, const mxArray**);
void     aloptim (int, mxArray**, int, const mxArray**);
void     diff_tg (int, mxArray**, int, const mxArray**);
mxArray *diff_tg1 (mxArray*,mxArray*);
void     com2can(int no, mxArray **out, int ni, const mxArray **in);

/* used in mixcopy dll - copies ARX statistics between mixtures, in place allowed */
void mixcopyARX  (int, mxArray**, int, const mxArray**);
/* copies numerical contents of fields specified FF->F */
void copyto(mxArray *F, mxArray *FF, int n);
void pause1(void);

/* data management */
#ifdef LIBRARY
extern int  fromdisk;
/* extern FILE *file; defined in mexlib.c */
extern int ndat_true;
extern int itime;
extern int  maxtd;
#else
int  fromdisk;
int  ndat_true;
int  itime;
int  maxtd;
#endif
/* at the beginning of each processing step call:
   if (fromdisk) tuk(); else itime = (int)TIME[0];
   after processing call:
   if (fromdisk) tak();
*/
void tuk(void);
void tak(void);

/* where arguments are processed call:
   ndat = tukinit(Mix, Ndat);
*/
int tukinit(mxArray *Mix, mxArray *Ndat);

/* where arguments are processed call:
   ndat = tukinit0(Ndat);
   if only ndat is of interest
*/
int tukinit0(mxArray *ndat);

void settime(int itime);

#include "operace.h"
#endif
```

5.4 Mixtools API header

```
/* @mex.h - Application Program Interface
   project ProDaCTools
*/
#include <stdio.h>


/* mxArray definition*/
typedef struct mxarray
{  int     M;                /* number of rows*/
   int     N;                /* number of columns*/
   int     type;             /* mxArray type*/
   double  *pr;              /* pointer to data if numeric*/
   struct  mxarray **fields; /* fields or cells if not numeric*/
   char **fieldnames;        /* fieldnames of struct matrices*/
}  mxArray;

/* struct mxarray types */
#define mxCHAR      (0)
#define mxDOUBLE    (1)
#define mxCOMPLEX   (2)
#define mxCELL      (11)
#define mxSTRUCT    (12)

/*  API definitions */
#define  mxGetPr(X)        ((X->pr))            /* get data pointer*/
#define  mxGetPi(X)        ((X->pr))            /* get imaginary part pointer*/
#define  mxGetScalar(X)    ((*(X)->pr))         /* get scalar from matrix*/
#define  mxGetCell(X,i)    (X->fields[i])       /* get cell i-th (from 0)*/
#define  mxSetCell(X,i,X1) (X->fields[i] = X1)  /* set cell i-th*/
#define  mxGetCellPr(X, i) (mxGetPr(mxGetCell(X,i))) /* get data pointer of a cell array */
#define  mxSetFieldN(X,i,X1)(X->fields[i] = X1) /* set structure i-th structure field*/

/* Mixtools additional functions for structures
#define  mxGetType(X)      (X->type)
#define  mxGetFieldN(X,i)  (X->fields[i])       /* get i-th field*/
#define  mxGetFieldPr(X,i) (X->fields[i]->pr)   /* get i-th field pointer*/
void     mxUpdtFieldN(mxArray*Xt, int field, mxArray*Xs); /* source_>target*/

/* allocation functions */
#define  mxCreateDoubleMatrix(M, N, T) alloc_array(M, N, mxDOUBLE)

/* simplified MATLAB functions */
#define  mxCreateCell(N)               alloc_array(1, N, mxCELL)
#define  mxCreateCellMatrix(M,N)       alloc_array(M, N, mxCELL)
#define  mxCreateStruct(N)             alloc_array(1, N, mxSTRUCT)
mxArray *mxDuplicateArray(mxArray *);
void     mxDestroyArray(mxArray *X);   /* dummy destroy*/
#define  mxFree(X)            ;
#define  mxSetPr(X,x)         (X->pr = x)
#define  mxSetM(X,x)          (X->M  = x)
#define  mxSetN(X,x)          (X->N  = x)
#define  mxGetNumberOfFields(X)  (mxGetN((X)))
mxArray *mxCreateStructMatrix(int m, int n, int nfields, const char **names);
mxArray *mxCreateScalarDouble(double);

void    *mxCalloc(unsigned int n, unsigned int size); /* basic allocation function*/
int      mxGetM(mxArray *Mix);       /* get matrix No. of rows*/
int      mxGetN(mxArray *Mix);       /* get matrix No. of columns*/

/* query API defined functions */
#define mxIsDouble(X)   ( (X->type==mxDOUBLE)    ?1:0)
#define mxIsComplex(X)  ( (X->type==mxCOMPLEX)   ?1:0)
#define mxIsNumeric(X)  ( (X->type==mxDOUBLE)    ?1:0)
#define mxIsCell(X)     ( (X->type==mxCELL)      ?1:0)
#define mxIsStruct(X)   ( (X->type==mxSTRUCT)    ?1:0)
#define mxIsChar(X)     ( (X->type==mxCHAR)      ?1:0)
#define mxIsEmpty(X)    ( (LEN(X)==0)            ?1:0)

/* Mixtools API memory management (malloc substitution) */
#define WORKSPACE_LENGTH (10000000) /* heap allocation*/
#ifdef LIBRARY
extern char workspace[];
extern int   wsp;
extern int maxwsp;
extern int DEBUG;
#else
```

```
char   workspace[WORKSPACE_LENGTH];
int    wsp = 0;                   /*pointer to free workspace*/
int maxwsp=0;
int DEBUG=0;
#endif
char *aloc(int);                       /* internal allocation routine*/
mxArray *alloc_array(const int m, const int n, const int type);

/* auxiliary functions and definitions */
#define mxSetScalar(X,S)  (X->pr[0]=S) /* not in MATLAB API*/
void prt(char *mess, mxArray *X);      /* debugging prints*/
void prtmx(mxArray *X);                /* debugging prints*/
void error(char*);                     /* set error flag, finish processing*/
#define LEN(X)            (X->M*X->N)  /* numerical matrix size*/
#define mxGetNumberOfElements(X) ((int)(X->M*X->N))

/* apilib.c library prototypes */
/* load (restore) mixture structure from dumpfile */
mxArray *mixload (const char *filename); /* load dumped matrix*/
void     mixsave (const mxArray *mix, const char *filename); /* save object*/
#define ITEM_ID_MAX 3
mxArray *_load_item (FILE *f);
void _save_item (const mxArray *array_ptr, FILE *f);
void mexCallMATLAB( int no, mxArray **out, int ni,mxArray **in,const char *fun);
mxArray *mxCreateString(char *str);
int mxGetString(mxArray *X, char *a, int maxlen);
int size(mxArray *X);
int mxAddField(mxArray *X, const char *field_name);
```