# FAMIMO Common Software Documentation ($\alpha$2.0)

**Antoine Duchâteau, Edy Bertolissi**

IRIDIA, Université Libre de Bruxelles
50, av. Franklin Roosevelt
1050 Brussels, Belgium
Tel: +32 2 650 27 29, Fax: 32 2 650 27 15
email: bersini@ulb.ac.be

**Abstract**

Fuzzy systems are becoming more and more accepted, both in industry and in the academic control community, as an emerging tool for the facilitated design of complex non-linear feedback control systems. The scope of the FAMIMO project is to enrich the present knowledge in designing fuzzy systems for the reliable control of high-dimensional complex processes. One of the major goals of project has been set in making available analytic and computerised tools for assessing the stability of both the control and learning algorithms integrated in the future industrial exploitation of these methods.

This document provides the manual of the FAMIMO toolbox, a collection of tools, accessible through a user interface or the Matlab$^{TM}$command line, for the development of control systems for a large diversity of MIMO processes.

**Abstract**

Fuzzy systems are becoming more and more accepted, both in industry and in the academic control community, as an emerging tool for the facilitated design of complex non-linear feedback control systems. The scope of the FAMIMO project is to enrich the present knowledge in designing fuzzy systems for the reliable control of high-dimensional complex processes. One of the major goals of project has been set in making available analytic and computerised tools for assessing the stability of both the control and learning algorithms integrated in the future industrial exploitation of these methods.

This document provides the manual of the FAMIMO toolbox, a collection of tools, accessible through a user interface or the Matlab$^{TM}$command line, for the development of control systems for a large diversity of MIMO processes.

# Chapter 1

# Introduction

## 1.1 General information

This documentation is divided in three parts. The learning guide where the use of the tools is illustrated through several examples (right now, only the commented examples are available in the tutorial directory). The reference guide that lists the complete set of API. The user interface reference that explains the use of the graphical user interface (GUI).

It is recommended to step through the examples in order to have an overview of the philosophy and the possibilities of the toolbox. The reference guide is a concatenation of the Matlab$^{TM}$help available for each function. This reference is also available in html format.

The following features are still to be implemented:

- the control functionalities and the integration of the FAST toolbox.
- the Mixture of Experts model (trained by the em algorithm).
- A fast (we hope ten times faster) simulation of the model.
- A faster lazy algorithm.
- A better implementation of the parameters settings.

## 1.2 Installation

Before to use the tools or the GUI, the software needs to be installed. Inside Matlab$^{TM}$, move to the directory `identification` and type the "setup" command. The paths needed by the program will be set up automatically and the mex files will be built if the binaries are not provided. The mex files are already compiled for the linux, the macos and the solaris environment.

## 1.3 Bug report

This software is sort of alpha quality. We would be very happy to hear about bugs you noticed. Please send a mail with the bug description (or better, a mfile that makes the bug appear) to bugs@iridia.ulb.ac.be.

## 1.4 Help needed

Do not hesitate to contact Antoine Duchateau (aduchate@ulb.ac.be) or Edy Bertolissi (eberto@iridia.ulb.ac.be) if you need any for the understanding or the use of the toolbox.

# Chapter 2

# Learning Guide

The main scope behind NLMIMO is the integration of the software produced by the partners taking part in the FAMIMO project (ESPRIT LTR Project 21911) in a unique toolbox. In order to achieve this it has been necessary to design an extremely flexible system that can accommodate the needs of the different laboratories collaborating in the project. For this reason the NLMIMO toolbox has been designed using an object oriented approach, which allows modularity, flexibility and ease of extension. This has been possible thanks to the new features present in the Matlab$^{TM}$5.0 which allows the development of an object oriented software.

The NLMIMO toolbox is built around three main classes. The `system` class, which defines all the attributes which characterize a system, the `dataset` class, which allows the storage and the processing of all the data, and the `mapping` class, which defines the the modeling of the process. As depicted in figure **??** the `system` class is the main one of the toolbox, and the other ones are used to describe some of the features of the system. In this way the data and the parameterization of the model are separate from the description of the system, and therefore can be easily modified or updated without the need of altering the structure of the `system` class. Each class has methods for accessing, setting and displaying its attributes, and checking their integrity and consistency.

## 2.1 Building things

Once the general philosophy and the structure of the NLMIMO toolbox has been understood, it is time to put your hands on the keyboard and use this toolbox for the definition and the analysis of dynamic or static systems.

The nlmimo toolbox supports two alternative system representations: internal, and external representation. The Matlab$^{TM}$object oriented approach provides an ideal way to describe the rich set of information which characterize these representations.
We will start with the description of the external representation. There are several different alternatives constructors which can be used to build an external object. The common one is to define the a new object using the following syntax:

```
m=external('external object',2,3)
```

this creates a new object belonging to the class external whose name is "external object", with 2 inputs and 3 outputs. Matlab$^{TM}$displays the following information on the screen:

```
m =
A external system with the following attributes

        name: 'external object'
```

```
      n_in: 2
     n_out: 3
  dynamics: []
      data: []
         T: -1
   mapping: {}
     links: []
    limits: [2x5 double]
  userData: []
       opt: []
      date: [1999 7 26 8 53 44.8545]
```

It is possible to access the fields of the object using two different approaches; using the "." (dot) operator (object_name.field), or using the `get` method:

```
get(object_name,'field')
```

The first approach is useful when analyzing objects at the Matlab$^{TM}$prompt, but is discouraged in programs since it is against the object oriented programming paradigm used in the toolbox. If not differently specified all the fields of and object can be set using the `set` method, which allows to set the values of multiple attributes. This can be done in the following way:

```
set(object_name,'field', value)
set(object_name,'field_1', value_1,'field_2',value_2,...)
```

The `name` field of an external object contains its name, and `n_in` and `n_out` store the its number of inputs and outputs.

The `dynamics` field provides the dynamics description of the system. It is a structure including three subfields, `nu`, `ny` and `nd`, which define the regressors and therefore the dynamical behaviour of the system. `ny` is a $n\_out \star n\_out$ square matrix which contains the relationships of each output (each row) with all the other outputs. Each entry $ij$ indicates the number of past values associated to the output $j$ used in the regressor of the output $i$. `nu` is a $n\_out \star n\_in$ matrix defining the relationships between the outputs and each input. Each entry $ij$ indicates the number of past values associated to the input $j$ used in the regressor of the output $i$. Finally `nd` is a $n\_out \star n\_in$ matrix defining the delay of the inputs respect to the outputs. Each entry $ij$ indicates the delay of the input $j$ used in the regressor of the output $i$. For example the following dynamical description of a 2 inputs 3 outputs system:

$$\begin{aligned} y_1(t) &= F(y_1(t-1), y_1(t-2), y_2(t-1), u_1(t), u_1(t-1), u_1(t-2), u_2(t)) \\ y_2(t) &= F(y_1(t-1), y_2(t-1), y_3(t-1), u_1(t), u_1(t-1), u_2(t-2)) \\ y_3(t) &= F(y_1(t-1), y_2(t-1), y_2(t-2), y_3(t-1), u_2(t-2), u_2(t-3) \end{aligned}$$

is translated in the following set of matrices:

$$\texttt{ny} = \begin{pmatrix} 2 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 1 \end{pmatrix} end \qquad \texttt{nu} = \begin{pmatrix} 3 & 1 \\ 2 & 1 \\ 0 & 2 \end{pmatrix} \qquad \texttt{nd} = \begin{pmatrix} 0 & 0 \\ 0 & 2 \\ 0 & 2 \end{pmatrix}$$

The method `add_dynamics` allows to add the dynamical description to the object. It is possible to use one of the two following syntaxes:

```
m=add_dynamics(m,dyn)
```

which attaches to the object `m` the dynamic description of the system present in the structure `dyn` which must include the fields 'nu', 'ny' and 'nd', defining the appropriate matrixes of the inputs, outputs and delays of the system;

```
m=add_dynamics(m,ny,nu,nd)
```

which attaches to the the object `m` the dynamic description of the system as defined in the arrays `ny`, `nu`, and `nd`.

The `data` field stores an object of type "dataset", which contains all the input-output data of the system. Once the data has been stored in a "dataset" object and all the data preprocessing has been done it is possible to associate it to the external system using the following syntax:

```
m=add_data(m,data,t)
```

which associates the `data` object to the field 'data' of the model `m`. The optional variable `t` specifies the sampling time of the model (default continuous = 0).

`T` represents the sampling time of the system; -1 indicates that this quantity is undefined, 0 that the system is a continuous one, a number greater than zero indicates the sampling time of the discrete time system. This field can be defined using the `set` method.

The field `mapping` is a cell array which contains the input-output mappings of the system. Each of the elements present in this class belongs from a subclass of the "mapping" class. New mappings can be added to this list using the following syntax:

```
m=add_mapping(m,mapp)
```

which appends the mapping defined in `mapp` to the list of the mappings of the object. The clear the list it is possible to use the function:

```
m=add_mapping(m)
```

Once the list of the mappings have been defined it is necessary to associate each output to a mapping. This relation is stored in the `links` field, which is an array of size $n\_out \star 2$. The first column defines the number of the mapping, and the second one the number of the output to which the mapping is associated. Of course there could be mapping which are not associated to any output. This is especially the case during the modeling stage when different models are tested, and only some of them are active at each moment.

The `limits` field stores the superior and inferior limits of the input and outputs of the system. It is a array of size $2 \star (n\_in + n\_out)$. The top row stores the inferior limit and bottom one the superior limits of the inputs and the outputs of the system. The default values are `-Inf +Inf` for all the inputs and the outputs.

The `date` field reports the creation date the the system. The `opt` and `userData` fields are used to store optional information or used defined data, and can be useful especially during the development of user defined functions.

Once a new external object has been created it is possible to start to fill the different fields using the methods described in the previous section. Usually the first fields to be defined are the `dynamics` and `data` fields. Then as soon as the mappings are defined it is possible to add them to the list of the mapping of the system using the `add_mapping` method. The method get allows to retrieve all the mapping field stored in an external object, however the following syntaxes allow to see the contents of each mapping directly. For example:

```
m.mapping{1}
```

or even more shortly:

```
m{1}
```

allow to see the contents of the first mapping of the mapping list associated to the object `m`.

Before gpomg on it is essential to define the `links` field in order to associate each output to a mapping.
There are a series of method which have been defined for the processing of the external objects. First of all there is the `identify` method. It most general syntax follows:

```
m=identify(m,j,options)
```

which performs an identification of the output `j` of the system passing to the identification procedure the parameters contained in the structure `options`. This method is a wrapper since it calls the identification procedure associated with the mapping connected with the output `j` passing to this one the `option` structure and the data contained in the `data` field to perform the identification procedure. The contents of `option` must be consistent with the identification procedure associated with the particular mapping.

The quality of the result of the identification procedure can be assessed with methods for calculating the mean square error, `error`, and the simulation error `simulation_error` of the model respect the to the reference set of values given as output data or a validation set given as input of the method.

Another method defined by the class is `regress` which extracts the regressors from the data whose sintax follows:

```
[r_in,r_out]=regress(m,j)
```

This function uses the dynamical system representation stored in the `dynamics` field to extract the regressors form the input-output data stored in the `data` field. The function returns the input regressors `r_in` in a cell array. Each element of the cell array is a regressor associated with an output. Each regressor is an array of size $size\_regressor \star num_e xaples$, where $size\_regressor$ depends from the structure of the regressor defined in the dynamical description of the system, and $num\_exaples$ is the number of examples given in the input-output data set. `r_in` is a cell array returning the correspondent output of each regressor taken from the output data set. Each entry in the cell array is a vector of size $1 \star num\_examples$.

The method `eval` calculates the output for the output `j` of a system given the regressor using the syntax:

```
out=eval(m,regressor,j)
```

Other two method are used to calculate the derivatives of the system respect to the inputs (`jacob_inputs`) and the parameters (`jacob_parameters`). All these methods are wrappers since they call the equivalent methods defined in the single mappings defined inside the object.

The method `simulate` performs a simulation of the model using the input data and feeding back the results of the simulation to calculate at each step the regressors.

The class defines other methods for normalize a mapping in order to make fit its limits to the interval [0,1] (if the `limits` are finite), denormalise it, and to check the consistency of all the fields defined in an external object.
The internal class is quite similar to the external one, but it defines some other fields and methods. When a new internal object is created with the command

```
m=inetrnal('internal object',1,2,3)
```

this creates a new object belonging to the class internal whose name is "internal object", with 1 input, 2 outputs and 3 internal states. Matlab$^{TM}$displays the following on the screen:

```
m=
A internal system with the following attributes

          name: 'internal object'
          n_in: 1
         n_out: 2
       n_state: 3
         state: [0 0 0]
          data: []
             T: -1
       mapping: {1x1 cell  }
         links: [2x2 double]
        limits: [2x3 double]
      userData: []
  statemapping: {1x1 cell  }
    statelinks: [3x2 double]
   statelimits: [2x3 double]
           opt: []
          date: [1999 7 26 13 51 6.6021]
```

As it is possible to see the main difference between an internal and an external object is the absence of the `dynamic` field and the presence of all a series of field associated with the internal representation of the state of the system. The object is already initialized with linear mappings for describing the evolution of the internal state and its outputs. The field `n_states` defines the number of the internal states of the system, and the field `state` their initial values. The fields `statemapping`, `statelinks` and `statelimits` are similar to the `mapping`, `links` and `limits` fields but they refer to the internal state of the system. it is possible to add new mappings for the state using the `add_statemapping` method in a way identical to the `add_mapping` method. Almost all the methods defined in the external class are defined in the internal one as well. In addition the following methods are defined: `eval_state` for calculating the internal state of the system given a regressor, `jacob_cloop` for calculating the jacobian with respect to the state for a closed loop.

## 2.2 The mapping class and its subclasses

The mapping class and its subclasses allows great power and flexibility to the nlmimo toolbox. These classes allow the to define the modeling of a process using the approach which best suits the data and the the ability of the programmer. Different approaches can be used and the results of each mapping can be evaluated and compared to choose the best possible system description. Now in turn a description of all the mappings which have been defined so far in the toolbox will be given.

Besides providing a common framework for every linear and non linear mappings supported inside the `system` class and its subclasses. The mapping architecture supply the user with a complete set of stand alone classes. A dynamical model should always be defined using the `internal` or `external` to take profit of the dynamical representation facilities provided by these classes. A static model can be defined using the `external` class if integration with simulink, models combination or data set embedding are whished features. It can also be defined using

one of the `mapping` subclasses when such features are not necessary which allow user to choose a smaller data structure while still having access to the core functions of the toolbox.

In the following pages, we present how to use the mapping class in stand alone mode. Their use while embedded inside a `system` object is obvious if one keeps into mind that many `system` methods like `eval`, `identify`, ... are nothing more than wrapper/dispatcher which pass their arguments to the called `mapping` subclasses methods.

### 2.2.1 The linear class

The linear mapping provides a simple linear relationship between inputs and outputs. To create a new linear object it is possible to use the following instruction:

```
l=linear('linear object',2,3)
```

which creates a new linear object called "linear object" with 2 inputs and 3 outputs. Matlab$^{TM}$displays the following information on the screen:

```
l =

A linear object with the following attributes

           name: 'linear object'
           n_in: 2
          n_out: 3
         limits: [2x5 double]
       userData: []
            opt: []
           date: [1999 7 26 14 51 31.6432]
    optimparams: []
        linears: [3x3 double]
```

the `name` field stores the name of the object, `n_in` and `n_out` the number of input and output of the mapping. It is possible to notice that there is no definition of time in all the mapping objects. This is beacuse the mapping objects are better understood as input-output functions which receive a certain number of inputs and return a defined number of outputs. The `limits` field stores the limits of the inputs and the outputs of the mapping. It is a array of size $2 \star (n\_in + n_o ut)$. The top row stores the inferior limit and bottom one the superior of the inputs and the outputs of the system. The default values are `-Inf +Inf` for all the inputs and the outputs. The most important field is the `linears` one which stores the linear relationships between the input and the outputs of the system. The `date` field stores the creation time of the object. The `userData` and `opt` field could be used by the user during the developemnt of programs for storing information. The `optimparams` field is used to specify which subset of the parameters has to be optimised when the `identify` method is called. Similarly to the case of the subclasses of the `system` class, all the fields can be set using the `set` method, and retrived using the 'dot' or the `get` method.

The `linear` class defines a series of methods for performing the several actions on its objects.

The method `identify` whose syntax follows:

```
m = identify(m,in,out)
```

perform a least mean square fit of the input-output data contained in the `in` and `out` arrays. The first one must have a size of $num\_examples \star n\_in$ while the second one must be equal to $num\_examples \star n\_out$. The results of the least mean square fit are stored in the `linears` field.

The class provides a method for evaluating the mapping given an input, `eval`, method for normalising and denormalising the mapping ( if the `limits` are finite). The methods `jacob_inputs` and `jacob_params` are used to calculate the derivatives of the mapping respect to the inputs and the parameters. The method `check` is provided to check the consistency of all the fields defined in an external object.

### 2.2.2 The lazy class

The class lazy allows to describe dynamical systems using a 'lazy' representation, which is a kind of k-nearest neigbour. It works in a simple way. The input-output data is stored in a database. When a prediction of the putput is needed a local model is fitted through the closest points of the query contained in the database. In order to create a new lazy object it is possible to use the following instruction:

```
l=lazy('lazy object',2,3)
```

which creates a new lazy object called "lazy object" with 2 inputs and 3 outputs. Matlab$^{TM}$displays the following information on the screen:

```
A lazy object with the following attributes

          name: 'lazy object'
          n_in: 2
         n_out: 3
        limits: [2x5 double]
      userData: []
           opt: []
          date: [1999 7 27 8 7 13.6540]
   optimparams: []
        id_par: []
       cmb_par: 1
    examples_x: []
    examples_y: []
```

the `name` field stores the name of the object, `n_in` and `n_out` the number of inputs and outputs of the mapping. This class as well has the `limits`, `userData`, `opt`, `optimparams`, and `date` fields which have the same meaning of the ones defined in the `linear` class. The fileds `examples_x` and `examples_y`, contain the full input-output database, which is used by the algorithm to perform the local fitting when needed. The elements which describe how the lazy model will be built are defined in the `id_par` and `cmb_par` fields.

When the `identify` method is called:

```
m = identify(m,in_data,out_data,options)
```

no real identification is performed, since the models are built on query-by-query basis, but the `examples_x` field is inizialized with the data contained in `in_data` and the `examples_y` field with the one contained in the `out_data`. The dimension of `in_data` must be $num\_examples \star n\_in$, the one of `out_data` $num\_examples \star n\_out$. The `options` argument is a structure which may be used to specify the identification method and options. It can define two fields; `id_par` for the identification parameters and `cmb_par` for the model combination parameters.The first one

defines the range in which the number of neighbors is searched. The identification parameter can assume the following forms:

1. `id_par[3,3]` $\qquad \texttt{id\_par} = \begin{vmatrix} \texttt{idm0} & \texttt{idM0} & \texttt{valM0} \\ \texttt{idm1} & \texttt{idM1} & \texttt{valM1} \\ \texttt{idm2} & \texttt{idM2} & \texttt{valM2} \end{vmatrix}$

   where `[idm`$d$`,idM`$d$`]` is the range $\mathcal{K}(d)$ in which the best number of neighbors is searched when identifying the local model of degree $d$. The scalar `valM`$d$ defines the maximum number of neighbors on which the local model of degree $d$ is validated:

$$mse_d^{cv}(k) = \frac{1}{m_d(k)} \sum_{j=1}^{m_d(k)} e_{d,j}^{cv}(k). \tag{2.1}$$

   where $m(k) = \min(k, \texttt{valM}d)$.

2. `id_par[3,2]` $\qquad \texttt{id\_par} = \begin{vmatrix} \texttt{idm0} & \texttt{idM0} \\ \texttt{idm1} & \texttt{idM1} \\ \texttt{idm2} & \texttt{idM2} \end{vmatrix}$

   where `[idm`$d$`,idM`$d$`]` have the same meaning as in point 1, and `valM`$d$ assumes the default value of `idM`$d$ for all the values of $d$: for every local model considered, all the neighbors used in identification are used also in validation.

3. `id_par[3,1]` $\qquad \texttt{id\_par} = \begin{vmatrix} \texttt{c0} \\ \texttt{c1} \\ \texttt{c2} \end{vmatrix}$

   Here `idm`$d$ and `idM`$d$ are obtained according to the following formulas:

$$\begin{aligned} \texttt{idm}d &= \texttt{floor(3 * p}d\texttt{ * c}d\texttt{)} \\ \texttt{idM}d &= \texttt{ceil(5 * p}d\texttt{ * c}d\texttt{)} \end{aligned}$$

   where `p`$d$ is the number of parameter of the model of degree $d$. Recommended choice: `c`$d$ = 1. The validation parameters get the default value as in point 2.

`cmb_par` determines the behavior of the function for what concerns the combination/selection of the local models. If `cmb_par` is not given, the best model is selected among those identified as specified by `id_par`. In this case, the model combination reduces to a simple model selection. The default value for `cmb_par` is 1 as it will be clear from what follows. If given, `cmb_par` can assume the following to forms:

1. `cmb_par[3,1]` $\qquad \texttt{cmb\_par} = \begin{vmatrix} \texttt{cmb0} \\ \texttt{cmb1} \\ \texttt{cmb2} \end{vmatrix}$

   where `cmb`$d$ is the number of models of degree $d$ that will be included in the local combination. Each local model will be therefore a combination of the best `cmb0` models of degree 0, the best `cmb1` models of degree 1, and the best `cmb2` models of degree 2, all identified as specified by `id_par`.

2. `cmb_par[1,1]` $\qquad \texttt{cmb\_par} = \begin{vmatrix} \texttt{cmb} \end{vmatrix}$

   where `cmb` is the number of models that will be combined, disregarding any constraint on the degree of the models that will be considered. Each local model will be therefore a combination of the best `cmb` models, identified as specified by `id_par`.

It is important to notice that less computational expensive models are the constant ones, followd by the linear ones. The quadratic models are particularly heavy and should not be used for for a number of inputs which is greater than 7-8. The combination of models improves the performance of the lazy algorithm. For more details about the features of the algorithm see "The lazy learning toolbox" [**?**].

The `eval` method calculates the output of the lazy system for a given input. At each query point it builds a local model fitting the data close to the query point using the database of the examples. This function is quite computationally expensive in comparison with the `eval` methods defined in other classes such as linear and taksug, since all the database of the examples must be completely scanned and the model generated at each query.

The method `jacob_inputs` computes the jacobian of the outputs respect to the inputs, while the method `jacob_params` is only provided for consistency, buy returns an empty array, since it has no meaning for the lazy mapping.

Methods for normalising and denormalising the mapping ( if the `limits` are finite) are provided. Finally the the method `check` checks the consistency of all the fields defined in a lazy object.

### 2.2.3 The taksug class

The `taksug` class is used when an input-output relation inside the system is to be represented with a Takagi Sugeno fuzzy system:

```
ts = taksug('Takagi Sugeno Mapping',2,3)
```

produces a 2 inputs, 3 outputs TS fuzzy function. The object returned is as follows:

```
A taksug object with the following attributes

           name: 'Takagi Sugeno Mapping'
           n_in: 2
          n_out: 3
         limits: [2x5  double]
       userData: []
            opt: []
           date: [1999 7 27 17 19 19.5549]
    optimparams: []
        n_rules: 0
     model_code: {}
              m: 2
        centers: []
     ivariances: []
            rls: []
            mfs: {1x2  cell  }
        linears: []
```

Once again, the `name` field stores the name of the object, `n_in` and `n_out` the number of input and output of the mapping. This class as well has the `limits`, `userData`, `opt`, `optimparams`, and `date` fields which have the same meaning as in the `linear` class. `n_rules` is the number of rules of the TS function. It should not be changed by hand, `add_rules` and `rem_rules` take care of maintaining it coherent with the others attributes of the object. The same remark is appliable for the dimensions of the other fields specific to `taksug`.

Let's add a few rules to the TS system:

```
ts = add_rules(ts,5)

ts =

A taksug object with the following attributes

           name: 'Takagi Sugeno Mapping'
           n_in: 2
          n_out: 3
         limits: [2x5   double]
       userData: []
            opt: []
           date: [1999 7 27 17 19 19.5549]
    optimparams: []
        n_rules: 5
     model_code: {1x3   cell  }
              m: 2
        centers: [2x5   double]
     ivariances: [2x2x5 double]
            rls: []
            mfs: {1x2   cell  }
        linears: [3x3x5 double]
```

If we look inside `ts.model_code`, Matlab$^{TM}$returns:

```
ans =

    'productspace'    'inversedist'    'linear'
```

which means that the TS rules are defined in the multidimensional space formed by the product of the input spaces (`productspace`), that the degrees of fullfilment of the rules are computed on the basis of the inverse of the distance (`inversedist`) and that the consequents of the rules are linear functions (`linear`). We can either change this value by writing for example:

```
set(ts,'model_code',{'productspace' 'gaussian' 'linear'})
```

or we could have called the `add_rules` with an extra parameter:

```
ts = add_rules(ts,5,{'productspace' 'gaussian' 'linear'})
```

If we call add_rules and the specified rule types don't match the previously added ones, the following error message is issued:

```
??? Error using ==> taksug/add_rules
You cannot add rules of this type to the current model.
Remove first the previous ones
```

In this case, `rem_rules` has to be used to empty the rules before new types of rules can be added.

The fields `centers` and `ivariances` are used to describe rules defined in the cross product space:

$$\mathcal{R}^{(l)} : \text{IF } x_1, x_2, ..., x_n \text{ IS } A^{(l)} \text{ THEN } y = h^{(l)}(x)$$

while the fields `rls` and `mfs` are used for projected rules:

$$\mathcal{R}^{(l)} : \text{IF } x_1 \text{ IS } A_1^{(l)} \text{ AND } \ldots \text{AND } x_n \text{ IS } A_n^{(l)} \text{ THEN } y = h^{(l)}(x).$$

centers is a $n\_in \star (n\_rules)$ matrix containing for each rule, the center $c_l$ of the fuzzy set, antecedent of the rule $l$. ivariances is a $n\_in \star n\_in \star (n\_rules)$ matrix $\Lambda_l$ containing for each rule, the quadratic matrix defining the metric of the antecedent of the rule $l$. *linears* contains the parameters of the linear consequent $L_l$ of the rule. The output of the Takagi Sugeno system is therefore computed as this:

$$y = \sum_{l}^{l=n} e^{-(x-c_l)^T \Lambda_l (x-c_l)} * L_l * x \tag{2.2}$$

for model_code equal to {'productspace' 'gaussian' 'linear'} and

$$y = \sum_{l}^{l=n} \frac{1}{[1 + (x-c_l)^T \Lambda_l (x-c_l)]^{\frac{1}{m-1}}} * L_l * x \tag{2.3}$$

for model_code equal to {'productspace' 'inversedist' 'linear'}.

mfs is cell array (one cell for each input) whose the cells contain the description of the rukes along each input dimension. rls defines the rule base. Its size is $n\_rules \star (n\_in)$.

Let's take an example of a projected Takagi Sugeno system. This is the Matlab$^{TM}$session:

```
>>ts = taksug('Takagi Sugeno Mapping',2,3);
>>ts.limits(:,1:2) =[0 0;1 1];
>>ts = sets_grid(ts, 3, 5, 'trapezoidal');
>>ts = add_rules(ts, 15, {'projected' 'rulegrid' 'linear'})

ts =

A taksug object with the following attributes

           name: 'Takagi Sugeno Mapping'
           n_in: 2
          n_out: 3
         limits: [ 2x5    double]
       userData: []
            opt: []
           date: [1999 7 28 11 9 29.3285]
    optimparams: []
        n_rules: 15
     model_code: { 1x3    cell  }
              m: 2
        centers: []
     ivariances: []
            rls: [15x2    double]
            mfs: { 1x2    cell  }
        linears: [ 3x3x15 double]
```

The taksug object is created using the constructor taksug, limits are specified for the two inputs. On the basis of these limits, the toolbox is able to populate the inputs with fuzzy sets using the sets_grid method. In this case, 3 sets are put on the first dimension and 5 are put on the second one. The sets are trapezoidal. The other possible set types are constant, gaussian and s-shaped. Rules are added by means of the add_rules method. Note the projected option

and the `rulegrid` option which specifies that fuzzy sets previously created have to be used in order to create evenly distributed rules.
Now let's look inside the structure of the object we have defined:

```
>>ts.mfs{1}

ans =

    1.0000         0         0    0.1667    0.3333
    1.0000    0.1667    0.3333    0.6667    0.8333
    1.0000    0.6667    0.8333    1.0000    1.0000
```

The first column of `mfs` contain, for each set, the code representing the type of the set (trapezoidal, gaussian, s-shaped). The next four are used to specify the location and the extent of the rules.
`rls` is a list of the fuzzy sets associated to each rule:

```
>>ts.rls

ans =

     1     1
     2     1
     3     1
     1     2
     2     2
     3     2
     ...
```

which states for example that rule 6 has to be read as:

$$\mathcal{R}^{(6)} : \text{IF } x_1 \text{ IS } A_1^3 \text{ AND } x_2 \text{ IS } A_2^2 \text{ THEN } y = L_6 x.$$

As we have seen, fuzzy sets can be added by hand and there position and size can be changed using the `set` function. The same holds for the linear consequents or the rules. An alternate way of defining the rule base is to use the `identify` method.

```
m = identify(m,in,out,options)
```

optimise the TS function to fit the input-output data contained in the `in` and `out` arrays. `in` must have a size of $num\_examples \star n\_in$ while `out` must be equal to $num\_examples \star n\_out$.
`option` is a struct whose fields `method`, `n_rules`, `fuzzyness`, ... (see help of the taksug/identify method) let choose among different optimisation procedures and tune these procedure.
The class provides a method for evaluating the mapping given an input, `eval`, method for normalising and denormalising the mapping ( if the `limits` are finite). The methods `jacob_inputs` and `jacob_params` are used to calculate the derivatives of the mapping respect to the inputs and the parameters. The method `check` is provided to check the consistency of all the fields defined in the `taksug` object.

### 2.2.4 The mamdani class

The `mamdani` class used to represent Mamdani type of fuzzy systems is very similar to the `taksug` class.
The same methods are available though the calling convention can differ slightly. The following example should be compared to the one provided for the `taksug` class:

```
>>m = mamdani('Mamdani Mapping',3,2);
>>m.limits = [0 0 0 0 0;1 1 1 1 1];
>>m = sets_grid(m,3,2,3,1,2);
>>m = add_rules(m,-1,{'product' 'meancentroid'},'rulegrid')

m =

A taksug object with the following attributes

           name: 'Mamdani Mapping'
           n_in: 3
          n_out: 2
         limits: [ 2x5 double]
       userData: []
            opt: []
           date: [1999 7 28 14 17 18.6538]
    optimparams: []
        n_rules: 18
     model_code: { 1x2 cell  }
            rls: [18x5 double]
            mfs: { 1x5 cell  }
```

With respect to the `taksug` class, it has to be highlighted that only the fields used in the `projected` rules are present in the `mamdani` class. The `linears` attribute is not present and is replaced by extra fuzzy sets (one series for each output) stored inside `mfs`. The rules are of the form:

$$\mathcal{R}^{(l)} : \text{IF } x_1 \text{ IS } A_1^{(l)} \text{ AND } \ldots \text{AND } x_n \text{ IS } A_n^{(l)} \text{ THEN } y_1 \text{ IS } B_1^{(l)} \ldots y_m \text{ IS } B_m^{(l)}.$$

and `rls` contains for each rules the list of the fuzzy sets defining the antecedent as well as the *fuzzy sets defining the consequent* of the rule.
The `model_code` attribute, determines the T-norm used for the AND operator and the defuzzification method used during the fuzzy inference.
No identification method is defined yet.

## 2.3 Handling data with the `dataset` class

Dataset is quite a complete package to handle data. It allows to perform normalisation, statistical analysis, clustering, supervised classification of data. The description of the use of dataset exceeds the scope of this tutorial. We encourage the reader to refer to the dataset tutorial or the dataset documentation for extensive information.

In this tutorial, we will focus on the use of the `dataset` constructor specifically applied to build dataset used for regression puposes. The syntax of `dataset` is the following:

```
dataset(data)
```

`data` holds the raw data (one sample on each row, one column for each variable).

## 2.4 NLMIMO identification in a nutshell

The nlmimo toolbox allows a great flexibility for the identification of static and dynamic systems. In any case the main steps which are needed to build such a system follow (let's consider an external system):

- build the system (`external`) defining the appropriate number of inputs and outputs;
- put the data inside a `dataset` object and attach it to the external object with the appropriate sampling time;
- define the dynamics of the system;
- define the mappings which will be used by the system, and make them consistent with the dynamics description of the system;
- attach the mapping to the external object and link them to the outputs of the system;
- perform the identification;
- evaluate the identification results

if the results are not satisfactory (thing that usually happens at the beginning), change the mappings or even the dynamics of the system and try again.

# Chapter 3

# API Reference

## 3.1 Data Manipulation Routines

Data collected from real processes often need to be pre-processed before being used for modelisation or control. Filtering, elimination of outliers, selection and combination of data coming from different sources are common crucial tasks that have to be carried on at an early srage in the modelisation process.

The `dataset` class is used to handle the data. Once an object belonging to this class is initialized with raw data, it is possible to perform two basic set of operations:

Basic data manipulation: concatenation of data sets, simple filtering, data normalisation, extraction of a data subset, separation of data into subsets on the basis of criteria, split of the data set for cross validation purposes.

Complex data analysis: statistical analysis, crisp clustering, fuzzy clustering, supervised classification using neural networks, decision iridia0trees, nearest neighbours and bayesian methods.

The `dataset` class takes care of transparently loading and storing parts of the data set from and to the hard disk, allowing the whole toolbox to deal with very large data sets: as long as the user respects the calling convention of the dataset class for retrieving and storing data points, and he does not have to worry about the physical location of the data. If the needed data point is not in RAM, it is loaded from the hard disk and changes will be saved.

Besides the functionalities it provides, the use of a separate class for storing and manipulating data sets insure that every pieces of information related to the data - name of the variables, possible discrete values, range of the variables, origin of a piece of data - are kept in the same place. Moreover, the dataset methods continuously check the coherence of the dataset properties and guarantee that the information always makes sense.

# dataset

### Purpose

Creator of the dataset class

### Synopsis

```
data
data(d)
data(data,'attr1',val1,'attr2',val2,...)
data(data,symbols,vartypes,labels,variables)
data(classes,data,symbols,vartypes,labels,variables)
data(datafile,variables,symbols,vartypes,labels)
```

### Description

`data` creates an empty data object.

`data(d)` clone the data object or fix a broken data object from its struct form.

`data(data,'attr1',val1,'attr2',val2,...)` create a data object. `data` are the data (each piece of data is on a different line). The couples of `attr1`, `val1`, ... are used to set the attributes.

`data(data,symbols,vartypes,labels,variables)` create a data object. `data` are the data (each piece of data is on a different line). `labels` is a column vector of the labels (optional) corresponding to each piece of data. `variables` [cell], are the names of the columns of the data. `symbols` [cell] are the symbolic correspondance for the symbolic data. `vartypes` is a cell array containing the type of the variable for each column ('continuous','discrete','symbolic','class'). There can only be one 'class' variable at a time. If vartypes is not specified, the types are deduced from the `data` and the symbol field

`data(classes,data,symbols,vartypes,labels,variables)` create a data object. `data` are the data (each piece of data is on a different line). `labels` is a column vector of the labels (optional) corresponding to each piece of data. `classes` contains the class associated with each piece of data. `variables` [cell], are the names of the columns of the data. `symbols` [cell] are the symbolic correspondance for the symbolic data.

`data(datafile,variables,symbols,vartypes,labels)` create a data object from a mat datafile. `datafile` is the name of the mat file where the data are saved. The file must contain a series of variables referenced by `variables`[cell]. Each of this variable must be a column vector([double] or [cell]) of length equal to the number of data points.

### Methods

- `addclass` add a class to the data set (obsolete)
- `addfeature` Adds a new feature to the data set
- `addpoint` Add a point to the dataset
- `addsfeature` Adds a new symbolic feature to the data set

- `addvariable` Add a new variable(feature) to the data set
- `bootstrap` BOOTSTRAP creates n new datasets ready for Bagging purpose or cross validation
- `btstrap2` BTSTRAP2 creates n new datasets ready for Bagging purpose or cross validation
- `classgroup` Groups a few classes into more general classes
- `classpart` selects a part (of the classes) of the data set
- `csplit` Cut the dataset into n pieces for x validation purposes
- `cut` Cut the dataset into n pieces for x validation purposes
- `normalise` performs a denormalisation of the continuous data field of a dataset
- `display` displays the dataset object
- `double` converts the dataset object to double
- `export` Export a data set to a comma delimited text file
- `fs` FS performs a features selection on the dataset d
- `fuzzyknn` performs fuzzy nearest neightbours algorithm
- `fuzzyknn2` performs fuzzy nearest neightbours algorithm (matlab release)
- `get` Access/query dataset property values
- `getmiss` get missing values from a dataset
- `gfkclus` performs Gustaffson Kessel clustering
- `horzcat` Horizontal concatenation operator
- `join` joins two data sets
- `kmeans` performs kMeans clustering
- 
- `leave1out` LEAVE1OUT performs the leave-one-out validation and computes the confusion matrix, error rates and Kappa inside a report structure
- `mkmeans` performs multi prototypes clustering
- `normalise` NORMALISE performs the linear normalisation of a learning dataset and transforms the corresponding testing dataset if it is provided
- `normstd` NORMSTD performs a normalisation of the continuous features of a learning set and a testing set
- `pick` picks randomly chosen examples out of a data set
- `placekernels` Place kernels to be used with smoothknn
- `plot` plot a dataset
- `postmining` compute the confusion matrix and error rates between supervision and prediction
- `randfs1` RANDFS1 : a features subsets builder by sampling with replacement
- `randfs2` RANDFS2 : a features subsets builder by sampling without replacement
- `remPoint` remove a point from a data set
- `set` Set object properties
- `simple_knn` classifies a testing set using kNN
- `simple_knn2` classifies the query using kNN (matlab release)
- `simple_MLP` SIMPLE_MLP performs a one-hidden-layer neural network classification
- `simpleknn` classifies the query using kNN

- `simpleknn` classifies the query using kNN
- `size` get the size of the data set
- `smoothknn` classifies the query using smoothknn
- `subsref` overload the subscript reference operator
- `svm` Support Vectors Machine classification
- `svminit` Initialise the Support Vectors for the svm classification
- `vertcat` vertical concatenation operator

# addclass

### Purpose

add a class to the data set (obsolete)

### Synopsis

```
d = addClass(d,c)
```

### Description

`d = addClass(d,c)` adds class `c` to dataset `d`.

# addfeature

**Purpose**

Adds a new feature to the data set

**See also**

also addsfeature

# addpoint

### Purpose

Add a point to the dataset

### Synopsis

```
addpoint(d,varargin)
```

### Description

`addpoint(d,varargin)` is equivalent to [d;dataset(varargin)] and adds a point to the dataset.

### See also

also vertcat, join

# addsfeature

**Purpose**

Adds a new symbolic feature to the data set

**See also**

also addsfeature

# addvariable

### Purpose

Add a new variable(feature) to the data set

### Synopsis

```
d = addvariable(d,v)
```

### Description

`d = addvariable(d,v)` adds the variable `v` to the dataset d. Fills the corresponding data with zeros.

# bootstrap

### Purpose

BOOTSTRAP creates n new datasets ready for Bagging purpose or cross validation

### Synopsis

```
LS = BOOTSTRAP(D,N)
```

### Description

`LS = BOOTSTRAP(D,N)` `D` is a dataset object, `N` is the desired number of bootstraps. `LS` is a cell array containing `N` new bootstrapped datasets.

### See also

CUT, BTSTRAP2

# btstrap2

### Purpose

BTSTRAP2 creates n new datasets ready for Bagging purpose or cross validation

### Synopsis

```
[LS,LSBAR] = BTSTRAP2(D,N)
```

### Description

`[LS,LSBAR] = BTSTRAP2(D,N)` `D` is a dataset object, `N` is the desired number of bootstraps.
`LS` is a cell array containing `N` new bootstrapped datasets. `LSBAR` is the complementary cell array for each bootstrap.

### See also

CUT, BOOTSTRAP

# classgroup

### Purpose

Groups a few classes into more general classes

### Synopsis

```
out=classgroup(d,classes,groupclasses)
```

### Description

`out=classgroup(d,classes,groupclasses)` groups the groups of classes specified by the cell array `classes` into new classes specified by the cell array `groupclasses`.

### See also

also classpart

# classpart

### Purpose

selects a part (of the classes) of the data set

### Synopsis

```
out=classpart(d,classes)
```

### Description

`out=classpart(d,classes)` allows to extracts from `d` the data concerning the part `classes` of the classes.

### See also

classgroup

# csplit

### Purpose

Cut the dataset into n pieces for x validation purposes

### Synopsis

```
[pick, rest] = cut (d,n)
```

### Description

`[pick, rest] = cut (d,n)` cuts `d` in `n` sub-dataset, each of which is placed in one cell of the cell array `pick`. For each ¡code¿pickj¡/code¿, the rest of the data set is placed inside ¡code¿restj¡/code¿

### Remarks

The cutting process respects the classes repartition. No Randomization is done.

### See also

cut xval

# cut

### Purpose

Cut the dataset into n pieces for x validation purposes

### Synopsis

```
[pick, rest] = cut (d,n)
```

### Description

`[pick, rest] = cut (d,n)` cuts `d` in `n` sub-dataset, each of which is placed in one cell of the cell array `pick`. For each ¡code¿pickj¡/code¿, the rest of the data set is placed inside ¡code¿restj¡/code¿

### Remarks

The cutting process respects the classes repartition.

### See also

xval

# normalise

**Purpose**

performs a denormalisation of the continuous data field of a dataset

**Synopsis**

```
d = denormalise(d)
```

**Description**

`d = denormalise(d)` denormalises the dataset `d`.

# display

### Purpose

displays the dataset object

### Synopsis

```
display(d)
```

### Description

`display(d)` displays the dataset object `d`.

# double

### Purpose

converts the dataset object to double

### Synopsis

```
out=double(in)
```

### Description

`out=double(in)` gets the data out of the data set `in` and return it inside `out`.

# export

### Purpose

Export a data set to a comma delimited text file

### Synopsis

```
export(d,name)
```

### Description

`export(d,name)` export the dataset `d` to file named `name`.

# fs

### Purpose

FS performs a features selection on the dataset d

### Synopsis

```
newd=fs(d,s)
```

### Description

`newd=fs(d,s)`
`d` is a dataset object
`s` is a vector containing the index of the features to select.

`newd` is a dataset with selected features only.

### See also

pick, cut, get

# fuzzyknn

### Purpose

performs fuzzy nearest neightbours algorithm

### Synopsis

```
myclass = fuzzyknn(d,query,k)
```

### Description

`myclass = fuzzyknn(d,query,k)` returns inside `myclass` the result of the classification of the point `query` using fuzzy KNN method. `d` is the data set and `k` the number of neibourghs.

### See also

newexpert, simpleknn, smoothknn

# fuzzyknn2

### Purpose

performs fuzzy nearest neightbours algorithm (matlab release)

### Synopsis

```
myclass = fuzzyknn2(learn,test,k)
```

### Description

`myclass = fuzzyknn2(learn,test,k)` returns inside `myclass` the result of the classification of the dataset `test` using fuzzy KNN method. `learn` is also a dataset and `k` the number of neighbours.

### See also

fuzzyknn, simpleknn, smoothknn

# get

### Purpose

Access/query dataset property values

### Synopsis

```
d=get(data,label)
```

### Description

`d=get(data,label)` returns the value of the specified property `label` of the data set `d`. Possibles properties are:

| | |
|---|---|
| `nv`: | the number of variables |
| `ns`: | the number of examples |
| `nc`: | the number of classes |
| `contdata`: | the continuous data |
| `symbdata`: | the symbolic data |
| `classes`: | the classes |
| `classeslist`: | the list of classes |
| `data`: | the data attribute |
| `symbols`: | the symbols attribute |
| `vartypes`: | the vartypes attribute |
| `labels`: | the labels attribute |
| `variables`: | the labels attribute |

### See also

set

# getmiss

### Purpose

get missing values from a dataset

### Synopsis

```
miss=getmiss(d)
```

### Description

`miss=getmiss(d)`
`d` is a dataset `miss` is a vector containing the position of missing values.

### See also

get

# gfkclus

### Purpose

performs Gustaffson Kessel clustering

### Synopsis

```
[cg, v] = gfkclus(d,n,m)
```

### Description

`[cg, v] = gfkclus(d,n,m)` Finds the center of the clusters `cg` and their variances `v` using the Gustaffson Kessel clustering method. `d` is the data set, `n` the number of prototypes by cluster and m is the fuzzyness parameter (¿1). `cg` is a `n` rows matrix (one for each cluster).

### See also

mkmeans, fkmeans, gkclus

# horzcat

### Purpose

Horizontal concatenation operator

### Synopsis

```
z = horzcat(d1, d2, ...)
```

### Description

`z = horzcat(d1, d2, ...)` Concatenates two or more data sets `d1`, `d2` horizontally. redundant classes are converted to symbolic variables.

# join

### Purpose

joins two data sets

### Synopsis

```
d = join(d1,d2)
```

### Description

`d = join(d1,d2)` Joins two different data sets `d1` and `d2`, even if they are not compatible (they do not share the same variables). This can lead to a strongly sparse data set if the data sets are fairly different. Note also that the variables will be resorted during the process.

### See also

vertcat

# kmeans

### Purpose

performs kMeans clustering

### Synopsis

```
cg = kmeans(d,n)
```

### Description

`cg = kmeans(d,n)` Finds the center of the clusters `cg` using the kMeans clustering method. `d` is the data set and `n` the number of prototypes by cluster. `cg` is a `n` rows matrix (one for each cluster).

### See also

mkmeans, fkmeans, gkclus

---

**Purpose**

**See also**

# leave1out

### Purpose

LEAVE1OUT performs the leave-one-out validation and computes the confusion matrix, error rates and Kappa inside a report structure

### Synopsis

```
REPORT = LEAVE1OUT(ALGO, LEARNSET, P1, P2)
```

### Description

### See also

CUT, BOOTSTRAP.

# mkmeans

### Purpose

performs multi prototypes clustering

### Synopsis

```
[cg,cp] = mkmeans(d,n,p)
[cg,cp] = mkmeans(d,n,p,l)
```

### Description

`[cg,cp] = mkmeans(d,n,p)` Finds the center of the clusters `cg` and there associated prototypes `cp` using a new clustering method developped in IRIDIA. `d` is the data set, `n` the number of clusters and `p` the number of prototypes by cluster. `cg` is a `n` rows matrix (one for each cluster). `cp` is a `p` rows and `n` columns 3D matrix.

`[cg,cp] = mkmeans(d,n,p,l)` `l` is the weight of the center of the cluster in the definition of the cost function minimised by the clustering method.

### See also

kmeans, fkmeans, gkclus

# normalise

### Purpose

NORMALISE performs the linear normalisation of a learning dataset and transforms the corresponding testing dataset if it is provided

### Synopsis

```
LEARNSET = NORMALISE(LEARNSET); [LEARNSET, TESTSET] = NORMALISE(LEARNSET, TESTSET);
```

### Description

`LEARNSET = NORMALISE(LEARNSET); [LEARNSET, TESTSET] = NORMALISE(LEARNSET, TESTSET);`
each feature value will be linearly computed between 0 and 1.
Algorithm

$$newd = (d - min(d))/(max(d) - min(d)) \tag{3.1}$$

### See also

NORMSTD

# normstd

### Purpose

NORMSTD performs a normalisation of the continuous features of a learning set and a testing set

### Synopsis

```
LEARNSET = NORMSTD(LEARNSET); [LEARNSET, TESTSET] = NORMSTD(LEARNSET, TESTSET);
```

### Description

### Remarks

The normalisation transforms the data so that their mean is zero and standard deviation 1.

### See also

normalise

# pick

### Purpose

picks randomly chosen examples out of a data set

### Synopsis

```
[out,theRest]=pick(d,n)
```

### Description

`[out,theRest]=pick(d,n)` picks `n` randomly chosen example out of dataset `d`. The picked examples are rturned inside `out`, the others are inside `theRest`.

### See also

cut

# placekernels

### Purpose

Place kernels to be used with smoothknn

### Synopsis

```
kernels=placekernels(data)
kernels=placekernels(data,num)
```

### Description

`kernels=placekernels(data)` Build the metrics (the `kernels`) needed by smoothknn using a non supervised clustering algorithm (Gustaffson Kessel). The number of kernels for each class is (right now badly) guessed.

`kernels=placekernels(data,num)` Build the metrics (the `kernels`) needed by smoothknn using a non supervised clustering algorithm (Gustaffson Kessel). `num` kernels are computed for each class.

# plot

### Purpose

plot a dataset

### Synopsis

```
h=plot(d,colors)
h=plot(d,colors,classes)
h=plot(d,colors,classes,vars)
```

### Description

`h=plot(d,colors)` plot the dataset `d` using the cell array of colors `colors`. Every classes are plot among the two first variables. The handles of the plots for the different classes are returned inside h.

`h=plot(d,colors,classes)` Only the classes specified inside the cell array `classes` are plot with respect to the two first features.

`h=plot(d,colors,classes,vars)` The classes specified inside the cell array `classes` are plot with respect to the variables specified inside the array `vars`.

# postmining

**Purpose**

compute the confusion matrix and error rates between supervision and prediction

**See also**

confmat

# randfs1

### Purpose

RANDFS1 : a features subsets builder by sampling with replacement

### Synopsis

```
FS = RANDFS1(D,PROP,NB)
```

### Description

```
FS = RANDFS1(D,PROP,NB)
```

`FS` = cell array containing the subsets of features
`D` = training set build with 'dataset'
`PROP` = percentage of the total number of attributes
`NB` = number of subsets

### Examples

fs = randfs1(train,0.5,10);
data=get(train,'contdata');
d=data(:,fs1);

### See also

randfs2

# randfs2

### Purpose

RANDFS2 : a features subsets builder by sampling without replacement

### Synopsis

```
FS = RANDFS2(D,PROP,NB)
```

### Description

```
FS = RANDFS2(D,PROP,NB)
```

`FS` = cell array containing the subsets of features
`D` = training set build with 'dataset'
`PROP` = percentage of the total number of attributes
`NB` = number of subsets

### Examples

fs = randfs1(train,0.5,10);
data=get(train,'contdata');
d=data(:,fs1);

### See also

randfs1

# remPoint

### Purpose

remove a point from a data set

### Synopsis

`remPoint(d,num)`

### Description

`remPoint(d,num)` remove point number `num` from dataset `d`.

### See also

addpoint, addvariable

# set

### Purpose

Set object properties

### Synopsis

```
set(d,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)
m_out = set(d,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)
```

### Description

`set(d,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)` sets the value of the attribute `AttrName1`, `AttrName2`, ... associated to the object `d`.

`m_out = set(d,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)` sets the value of the attribute `AttrName1`, `AttrName2`, ... associated to the object `d` and return the modified model inside m_out. The original model is not modified. The following `AttrName` are recognised:

| | |
|---|---|
| `contdata`: | the continuous data |
| `symbdata`: | the symbolic data |
| `classes`: | the classes |
| `labels`: | the labesl of the data points |
| `vartypes`: | the types of the variables |

### See also

get

# simple_knn

### Purpose

classifies a testing set using kNN

### Synopsis

```
[out]=simpleknn(data,test,n)
```

### Description

`[out]=simpleknn(data,test,n)` The classification of the dataset `test` is performed using a `n` nearest neighbours classifier from the dataset DATA.

### See also

smoothknn, fuzzyknn

# simple_knn2

### Purpose

classifies the query using kNN (matlab release)

### Synopsis

```
[out]=simple_knn2(data,query)
[out]=simpleknn(data,query,n)
```

### Description

`[out]=simple_knn2(data,query)` classifies the point `query` and returns the result inside classes `out`.

`[out]=simpleknn(data,query,n)` The classification is performed using a `n` nearest neighbours classifier.

### See also

smoothknn, simpleknn, fuzzyknn

# simple_MLP

### Purpose

SIMPLE_MLP performs a one-hidden-layer neural network classification

### Synopsis

```
CLASS = SIMPLE_MLP(LEARNSET,TESTSET,ALGO,NBEPOCHS,GOAL)
```

### Description

```
CLASS = SIMPLE_MLP(LEARNSET,TESTSET,ALGO,NBEPOCHS,GOAL)
```

LEARNSET, TESTSET are 'dataset' objects.
GOAL : first stopping condition, the global error rate. Common value is 0.01.
NBEPOCHS : second stopping condition, the number of epochs. Common value is 50.
ALGO :
- 'trainlm' Levenberg-Marquardt, slow and memory consuming but efficient.
- 'trainrp' Resilient backpropagation, fast, NBEPOCHS bigger.
- 'trainbfg' BFGS Quasi-Newton, fast, NBEPOCHS bigger.

'Neural Networks' Toolbox required.

### See also

BAG_MLP, BAGFS_MLP, MFS_MLP, simple_knn, simple_c45, simple_lda

# simpleknn

### Purpose

classifies the query using kNN

### Synopsis

```
[out,confid]=simpleknn(data,query)
[out,confid]=simpleknn(data,query,n)
```

### Description

`[out,confid]=simpleknn(data,query)` classifies the point `query` and returns the result inside classes `out` with a confidence `confid`. The classification is done by means of a 2 nearest neighbours method.

`[out,confid]=simpleknn(data,query,n)` The classification is performed using a `n` nearest neighbours classifier.

### See also

smoothknn, fuzzyknn

# simpleknn

### Purpose

classifies the query using kNN

### Synopsis

```
[out]=simpleknn(data,query)
[out]=simpleknn(data,query,n)
```

### Description

`[out]=simpleknn(data,query)` classifies the point `query` and returns the result inside classes `out` with a confidence `confid`. The classification is done by means of a 2 nearest neighbours method.

`[out]=simpleknn(data,query,n)` The classification is performed using a `n` nearest neighbours classifier.

### See also

smoothknn, fuzzyknn

# size

### Purpose

get the size of the data set

### Synopsis

```
out=size(d)
out=size(d,n)
```

### Description

`out=size(d)` returns the dimensions (number of examples, number of variables) of the database `d`.

`out=size(d,n)` returns either the number of examples(`n` = 1) or the number of variables(`n` = 2) of the database `d`.

### See also

also get

# smoothknn

### Purpose

classifies the query using smoothknn

### Synopsis

```
[out,confid]=smoothknn(data,query)
[out,confid]=smoothknn(data,query,kernels)
[out,confid]=smoothknn(data,query,kernels,ratio)
```

### Description

`[out,confid]=smoothknn(data,query)` classifies the point whose the features are equal to `query` inside classes `out` with a confidence `confid`. The classification is done by means of a kNN with fixed metric using the dataset `data`.

`[out,confid]=smoothknn(data,query,kernels)` The classification is performed using an adaptive metric based on kernels.

`[out,confid]=smoothknn(data,query,kernels,ratio)` Ratio increases the smoothness of the algorithm by extending the zone of influence of each point (equivalent to take a wider neibourghood).

### See also

simpleknn, fuzzyknn

# subsref

### Purpose

overload the subscript reference operator

### Synopsis

```
function out=subsref(d,s)
```

### Description

`function out=subsref(d,s)` subsref(d,s) will return either a sub-datatset or a part of the dataset object depending on the subscript method. `d` is the data set and `s` is the reference specification structure.

### See also

also `subsref`

# svm

### Purpose

Support Vectors Machine classification

### Synopsis

```
class = svm(d, data, sv, act)
```

### Description

`class = svm(d, data, sv, act)` classifies the piece of data `data` using dataset `d` and support vectors `sv`. `act` is the activation.

### See also

svminit

# svminit

### Purpose

Initialise the Support Vectors for the svm classification

### Synopsis

```
sv = svminit(d, ker, bound)
```

### Description

`sv = svminit(d, ker, bound)` computes the support vectors needed to perform support machine classification and return them inside `sv`. `d` is the data set, `ker` specifies the type of kernel.

### See also

svm

# vertcat

### Purpose

vertical concatenation operator

### Synopsis

```
z = vertcat(d1, d2, ...)
```

### Description

`z = vertcat(d1, d2, ...)` concatenates two or more data sets `d1`, `d2` etc. Symbolic codes are reorganised in order to avoid redundancies.

### See also

also

## 3.2 High Level Routines

### 3.2.1 `System` Abstract Class

The `system` class centralizes the information needed to describe a general static or dynamic system. It defies the number of inputs and outputs of the system and their limits. All the input-output data associated with the system is stores in an attribute of type `dataset`. Since a system can be represented using an internal (state space based) or external (input-output based) representation the `system` class has been defined as an abstract class and captures the common features of these two alternative representations. Two subclasses, the `external` and the `internal` one, implement these alternative descriptions of a system.

# system

### Purpose

Constructor for the system abstract class

### Synopsis

```
m=system
m=system(name,n_in,n_out)
```

### Description

`m=system` creates a new general system skeleton

`m=system(name,n_in,n_out)` creates a new general system with `n_in` inputs and `n_out` outputs. The name `name` is associated to it.

### Methods

- `add_data` add a input-output data set to the model
- `add_mapping` add a mapping to a system object
- `clone_output` clone the mapping associated to an output
- `denormalise` normalises the system M
- `display` Display an object of class system
- `eval` computes the value of the system for some regressor
- `get` gets the value of the attribute of an object
- `identify` Identify the system from data (abstact method)
- `interp_model` computes the jacobian of the model by interpoling local models
- `jacob_inputs` computes the jacobian of the model
- `jacob_params` computes the jacobian of the model
- `normalise` normalises the system M
- `plot` plot the system
- `set` Set object properties

### See also

# add_data

### Purpose

add a input-output data set to the model

### Synopsis

```
m=add_data(m)
m=add_data(m,data,t)
```

### Description

`m=add_data(m)` removes all the input-output data from the model `M`.

`m=add_data(m,data,t)` puts `data` into the field 'data' of the model `m`, `data` must be a data object. The optional variable `t` specifies the sampling time of the model (default continuous = 0).

### Remarks

This function is used to inizialize the System 'data' field with the model input-output data. The data is stored in the 'data' field which store.

# add_mapping

### Purpose

add a mapping to a system object

### Synopsis

```
m=add_mapping(m)
m=add_mapping(m,mapping)
```

### Description

`m=add_mapping(m)` clears the mapping field of the object

`m=add_mapping(m,{mapping})`

### Remarks

This method adds a mapping to the list of the current mappings of the system object. The links field, which connects the mapping with the output, must be set using the `set` method.

# clone_output

### Purpose

clone the mapping associated to an output

### Synopsis

```
m = clone_output(m,j,label)
```

### Description

`m = clone_output(m,j,label)` clones the mapping associated to the output `j`, and gives it the new name specified by `new_name`.

# denormalise

### Purpose

normalises the system M

### Synopsis

```
m=normalise(m)
```

### Description

`m=normalise(m)` denormalises the system `m`

### Remarks

This function denormalises a mapping in order to make fit its limits to the original values before normalisation.

# display

**Purpose**

Display an object of class system

**Synopsis**

```
display(m)
```

**Description**

`display(m)` Display the object m of class system

**See also**

# eval

### Purpose

computes the value of the system for some regressor

### Synopsis

```
out=eval(m,r,j)
```

### Description

`out=eval(m,r,j)` returns the value `out` of the output(s) `j` of mapping `m` given the input regressor `r`.

# get

### Purpose

gets the value of the attribute of an object

### Synopsis

```
d = get(m,label,opt)
```

### Description

`d = get(m,label,opt)` gets the value of the attribute `label` associated to the object `m` and return it inside `d`. The following codes are recognised:

| | |
|---|---|
| `name`: | returns the name of the object |
| `n_in`: | returns the number of inputs of the system |
| `n_out`: | returns the number of outputs of the system |
| `data`: | returns the data set linked to the system |
| `T`: | returns the sampling period of the discrete system, 0 if continuous |
| `mapping`: | returns the array of mappings |
| `links`: | returns the links array which defines which output is computed by which mapping |
| `limits`: | returns the bounds of the inputs and outputs |
| `userData`: | returns the 'userData' field |
| `opt`: | returns the 'opt' field |
| `date`: | returns the date of creation of the object |
| `mapfield`: | let the user access directly property of the mappings. When used, opt must contain a 2 elements cell array `n` 'property'. `n` is the index of the mapping and 'property' is the name of the property which has to be accessed. An even more convenient way to access mappings' fields is a achieved through the syntax sn.property. |
| `numparams`: | returns the number of parameters to be optimised |
| `params`: | returns the parameters to be optimised |

### See also

set

# identify

### Purpose

Identify the system from data (abstact method)

### Synopsis

```
m = identify(m,j,options)
```

### Description

`m = identify(m,j,options)` performs the identification of the model `m` using the data embeded inside the object for the output `j`. Use `options` in order to specify the identification method and options.

# interp_model

### Purpose

computes the jacobian of the model by interpoling local models

### Synopsis

```
interp_model(m,r,j)
```

### Description

`interp_model(m,r,j)` computes for output `j` the jacobian of the model `m` with respect to the regressor of the system at point `r`.

### Remarks

The procedure for computing the jacobian depends on the underlying representation of the model. The result `out` is an array of size number of outputs * number of inputs containing the derivatives of the outputs with respect to the inputs.

# jacob_inputs

**Purpose**

computes the jacobian of the model

**Synopsis**

```
jacob_inputs(m,r,j)
```

**Description**

`jacob_inputs(m,r,j)` computes for output `j` the jacobian of the model `m` with respect to the regressor of the system at point `r`.

**Remarks**

The procedure for computing the jacobian depends on the underlying representation of the model. The result `out` is an array of size number of outputs * number of inputs containing the derivatives of the outputs with respect to the inputs.

# jacob_params

### Purpose

computes the jacobian of the model

### Synopsis

```
jacob_params(m,r)
jacob_params(m,r,j)
```

### Description

`jacob_params(m,r)` computes for each output the jacobian of the model with respect to the parameters at regressor `r`.

`jacob_params(m,r,j)` computes for output(s) `j` the jacobian of the model with respect to the parameters at regressor `regressor`.

# normalise

**Purpose**

normalises the system M

**Synopsis**

```
m=normalise(m)
```

**Description**

`m=normalise(m)` normalises the system `m`

**Remarks**

This function normalizes a mapping in order to make fit its limits to the interval [0,1]

# plot

**Purpose**

plot the system

**Synopsis**

```
f=plot(m)
```

**Description**

```
f=plot(m)
```

**See also**

# set

### Purpose

Set object properties

### Synopsis

```
set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)
m_out = set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)
```

### Description

`set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)` sets the value of the attribute `AttrName1`, `AttrName2`, ... associated to the object `m`.

`m_out = set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)` sets the value of the attribute `AttrName1`, `AttrName2`, ... associated to the object `m` and return the modified model inside m_out. The original model is not modified. The following `AttrName` are recognised:

| | |
|---|---|
| `name`: | sets the name of the system |
| `n_in`: | sets the number of inputs of the system |
| `n_out`: | sets the number of outputs of the system |
| `data`: | sets the data set linked to the system |
| `T`: | sets the sampling period of the discrete system, 0 if continuous |
| `mapping`: | sets the array of mappings |
| `links`: | sets the links array which defines which output is computed by which mapping |
| `limits`: | sets the limits of the domain of definition of the mapping. The value must be a `2*n_in` matrix where the upper and lever saturation level are |
| `userData`: | sets the 'userData' field |
| `opt`: | sets the 'opt' field |
| `date`: | sets the date of creation of the object |
| `mapfield`: | let the user access directly property of the mappings. When used, `value` must contain a 3 elements cell array `n` 'property' `mapvalue`. `n` is the index of the mapping, 'property' is the name of the property which has to be changet and `mapvalue` is the new value. An even more convenient way to change mappings' fields is a achieved through the syntax sn.property = mapvalue. |
| `params`: | sets the parameters to be optimised |

### See also

get

### 3.2.2 `Internal` Class

In the case of a system which uses an external representation the input-output relation is of the type:

$$\frac{dy}{dt} = f(y, u)$$

in the case of a continuous system, and

$$y(t+1) = f(y(t), u(t))$$

in the case of a discrete time system.

The different input-output relations, which are represented using an objects belonging to one of the subclasses of the `mapping` class, are stored inside an attribute of the `external` class. If a system is multi-outputs, a different relation (a linear one, a fuzzy representation, a non-parametric approach) can be defined for each output. This allows a very general representation and lets the user experiment different types and sizes of models.

# internal

### Purpose

Constructor for the internal class

### Synopsis

```
m=internal
m=internal(name,n_in,n_out)
m=internal(name,n_in,n_out,n_state)
```

### Description

`m=internal` creates a new general system skeleton

`m=internal(name,n_in,n_out)` creates a new general system with `n_in` inputs and `n_out` outputs. The name `name` is associated to it.

`m=internal(name,n_in,n_out,n_state)` creates a new general system with `n_in` inputs, `n_out` outputs and `n_state` states. The name `name` is associated to it.

### Methods

- `add_statemapping` add a mapping for describing the state of an internal object
- `conicity` compute the conicity stabity criterium
- `display` Display an object of class internal
- `eval_state` computes the value of the state of a system for some regressor
- `get` gets the value of the attribute of an object
- `glob_indice` Compute the third stability indice for a closed loop
- `identify` Identify the system from data (abstact method)
- `jacob_cloop` Compute the jacobian with respect to the state for a closed loop
- `jacob_inputs` computes the jacobian of the model
- `jacob_params` computes the jacobian of the model
- `loc_indices` Compute the first and the second stability indices for a closed loop
- `regress`
- `seek_eq` Use the optimisation toolbox to find a cl
- `set` Set object properties
- `stateregress` Compute the regressor for the state mapping

### See also

also

# add_statemapping

### Purpose

add a mapping for describing the state of an internal object

### Synopsis

```
m=add_statemapping(m)
m=add_statemapping(m,mapping)
```

### Description

`m=add_statemapping(m)` clears the statemapping field of the object

`m=add_statemapping(m,{mapping})`

### Remarks

This method adds a mapping to the list of the current mappings attached to the state of the internal object. The links field, which connects the mapping with the state output, must be set using the `set` method.

# conicity

### Purpose

compute the conicity stabity criterium

### Synopsis

```
conicity(plant,range,params,controller)
conicity(plant,range,controller)
```

### Description

`conicity(plant,range,params,controller)` coputes the conicity stability criterium for the plant `plant` in the range specified by the array `range` with the static controller `controller`.

### Remarks

This method is applicable only if the plant system is has single takagi sugeno mapping for all the outputs or it is a linear one. In addition the controller must be a static one. The outputs of the function are: CONICITY INDEX = CONIC DESVIATION / CONIC ROBUSTNESS CONIC DEVIATION = GAIN(CONTROLLER - CENTRE) CONIC ROBUSTNESS = 1 / GAIN(FEEDB((A,B,C,D),CENTRE) where `centre` is the center of the cone which is calculated with the method specified in the field `search_method` of the `params` structure. Possible values for this field are:
`linearization`: which uses a linarization technique for finding the center of the cone

if no `search_method` is specified the linearization techique is used.

# display

**Purpose**

Display an object of class internal

**Synopsis**

```
display(m)
```

**Description**

`display(m)` Display the object m of class internal

**See also**

# eval_state

### Purpose

computes the value of the state of a system for some regressor

### Synopsis

```
out=eval_state(m,r,j)
```

### Description

`out=eval_state(m,r,j)` returns the value `out` of the state(s) `j` of mapping `m` given the input regressor `r`.

# get

### Purpose

gets the value of the attribute of an object

### Synopsis

```
d = get(m,label,opt)
```

### Description

`d = get(m,label,opt)` gets the value of the attribute `label` associated to the object `m` and return it inside `d`. The following codes are recognised:

| | |
|---|---|
| `name`: | returns the name of the object |
| `n_in`: | returns the number of inputs of the system |
| `n_out`: | returns the number of outputs of the system |
| `data`: | returns the data set linked to the system |
| `T`: | returns the sampling period of the discrete system, 0 if continuous |
| `mapping`: | returns the array of mappings |
| `links`: | returns the links array which defines which output is computed by which mapping |
| `limits`: | returns the bounds of the inputs and outputs |
| `userData`: | returns the 'userData' field |
| `opt`: | returns the 'opt' field |
| `date`: | returns the date of creation of the object |
| `mapfield`: | let the user access directly property of the mappings. When used, opt must contain a 2 elements cell array `n` 'property'. `n` is the index of the mapping and 'property' is the name of the property which has to be accessed. An even more convenient way to access mappings' fields is a achieved through the syntax sn.property. In both syntax, if `n` is negative, the ¡CODE¿abs(n)¡/CODE¿ mapping describing the state is accessed. |
| `numparams`: | returns the number of parameters to be optimised |
| `params`: | returns the parameters to be optimised |
| `n_state`: | returns the number of states |
| `state`: | returns the state vector of the system |
| `statemapping`: | returns the mappings used to describe the state |
| `statelinks`: | returns the links array which defines which state is computed using which statemapping |
| `statelimits`: | returns the bounds of the states |
| `statemapfield`: | let the user access directly property of the statemappings. When used, opt must contain a 2 elements cell array `n` 'property'. `n` is the index of the mapping and 'property' is the name of the property which has to be accessed. |

### See also

set

# glob_indice

### Purpose

Compute the third stability indice for a closed loop

### Synopsis

```
[i1,i2] = glob_indice(m, c, y_d, v, limits)
```

### Description

`[i1,i2] = glob_indice(m, c, y_d, v, limits)` `m` is the model (external) and `c` is the controller. `y_d` is the desired output and `v` are the perturbations.

### See also

local_indice

# identify

### Purpose

Identify the system from data (abstact method)

### Synopsis

```
m = identify(m,in,out,options)
```

### Description

`m = identify(m,in,out,options)` performs the identification of the model `m` using the data embeded inside the object. Use `options` in order to specify the identification method and options.

# jacob_cloop

**Purpose**

Compute the jacobian with respect to the state for a closed loop

**Synopsis**

```
out = jacob_cloop(m,c,yd,v)
```

**Description**

```
out = jacob_cloop(m,c,yd,v)
```

**See also**

also

# jacob_inputs

### Purpose

computes the jacobian of the model

### Synopsis

```
jacob_inputs(m,r,j)
```

### Description

`jacob_inputs(m,r,j)` computes for output `j` the jacobian of the model `m` with respect to the regressor of the system at point `r`.

### Remarks

The procedure for computing the jacobian depends on the underlying representation of the model. The result `out` is an array of size number of outputs * number of inputs containing the derivatives of the outputs with respect to the inputs.

# jacob_params

### Purpose

computes the jacobian of the model

### Synopsis

```
jacob_params(m,r)
jacob_params(m,r,j)
```

### Description

`jacob_params(m,r)` computes for each output the jacobian of the model with respect to the parameters at regressor `r`.

`jacob_params(m,r,j)` computes for output(s) `j` the jacobian of the model with respect to the parameters at regressor `regressor`.

# loc_indices

### Purpose

Compute the first and the second stability indices for a closed loop

### Synopsis

```
[i1,i2] = loc_indices(m,c,yd,v,eq_m,eq_c)
```

### Description

`[i1,i2] = loc_indices(m,c,yd,v,eq_m,eq_c)` `m` is the model (external) and `c` is the controller. `y_d` is the desired output and `v` are the perturbations.

### See also

glob_indice

# regress

### Purpose

### Synopsis

```
[ir, or] = regress(m,j)
```

### Description

```
[ir, or] = regress(m,j)
```

### See also

# seek_eq

### Purpose

Use the optimisation toolbox to find a cl

### Synopsis

`seek_eq(m,c,yd,options)`

### Description

`seek_eq(m,c,yd,options)` Finds the equilibrium of the closed loop formed by the model `m` and the controller `c` for a set point of `yd` and a perturbation of `v`. `options` contains all the options of the algorithm.

### Remarks

-loop equil.

### See also

# set

### Purpose

Set object properties

### Synopsis

```
set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)
m_out = set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)
```

### Description

`set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)` sets the value of the attribute `AttrName1`, `AttrName2`, ... associated to the object `m`.

`m_out = set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)` sets the value of the attribute `AttrName1`, `AttrName2`, ... associated to the object `m` and return the modified model inside m_out. The original model is not modified. The following `AttrName` are recognised:

| | |
|---|---|
| `name`: | sets the name of the system |
| `n_in`: | sets the number of inputs of the system |
| `n_out`: | sets the number of outputs of the system |
| `data`: | sets the data set linked to the system |
| `T`: | sets the sampling period of the discrete system, 0 if continuous |
| `mapping`: | sets the array of mappings |
| `links`: | sets the links array which defines which output is computed by which mapping |
| `limits`: | sets the limits of the domain of definition of the mapping. The value must be a `2*n_in` matrix where the upper and lever saturation level are |
| `userData`: | sets the 'userData' field |
| `opt`: | sets the 'opt' field |
| `date`: | sets the date of creation of the object |
| `mapfield`: | let the user access directly property of the mappings. When used, `value` must contain a 3 elements cell array `n` 'property' `mapvalue`. `n` is the index of the mapping, 'property' is the name of the property which has to be changet and `mapvalue` is the new value. An even more convenient way to change mappings' fields is a achieved through the syntax sn.property = mapvalue. In both syntax, if `n` is negative, the ¡CODE¿abs(n)¡/CODE¿ mapping describing the state is accessed. |
| `params`: | sets the parameters to be optimised |
| `n_state`: | sets the number of states |
| `state`: | sets the state vector of the system |
| `statemapping`: | sets the mappings used to describe the state |
| `statelinks`: | sets the links array which defines which state is computed using which statemapping |
| `statelimits`: | sets the bounds of the states |

### See also

get

# stateregress

### Purpose

Compute the regressor for the state mapping

### Synopsis

```
[ir, or] = stateregress(m,j)
```

### Description

`[ir, or] = stateregress(m,j)` does exactly the same as regress for the state mapping

### See also

regress

### 3.2.3 `External` Class

In the case of internal representation the system is described by the following input output relationships:

$$\dot{x} = a(x) + b(u)$$

$$y = c(x) + d(u)$$

or in the case of a continuous time system

$$x(t+1) = a(x(t)) + b(u(t))$$

$$y(t) = c(x(t)) + d(u(t))$$

in the case of a discrete time system.

In the class `internal`, `mapping` objects are used to describe the evolution of the state variables in the same way they are used for the outputs. It is possible to choose a different model for each of the internal variables of the system as well as for its outputs.

# external

### Purpose

Constructor for the external class

### Synopsis

```
m=external
m=external(m)
m=external(name,n_in,n_out)
```

### Description

`m=external` creates a new general external skeleton.

`m=external(m)` clone the external object `m`. This can be also used to transform a structure in a external object since the common fields are used to inizialize the external object.

`m=external(name,n_in,n_out)` creates a new general system with `n_in` inputs and `n_out` outputs. The name `name` is associated to it.

### Methods

- `add_dynamics` add the dynamical description of the model
- `build_regressors` build the regresors for a MIMO system
- `check` checks the consistency of all the fields of a external object
- `display` Display an object of class external
- `error` computes the mean square error of the model
- `errordb` Graphical User Interface for inspecting the errors of the mappings
- `get` gets the value of the attribute of an object
- `identify` Identify the system from data
- `regress` extracts the regressors from the data
- `set` Set object properties
- `simulate` simulates the model
- `simulation_error` computes the mean square error of the simulated model

# add_dynamics

### Purpose

add the dynamical description of the model

### Synopsis

```
m=add_dynamics(m,dyn)
m=add_dynamics(m,ny,nu,nd)
```

### Description

`m=add_dynamics(m,dyn)` attaches to the model `m` the dynamic description of the model present in the structure `dyn` which must include the fields 'nu', 'ny' and 'nd', defining the appropriate matrixes of the inputs, outputs and delays of the system.

`m=add_dynamics(m,ny,nu,nd)` attaches to the model `m` the dynamic description of the system as defined in the arrays `ny`, `nu`, and `nd`.

### Remarks

This is done by inizializing the 'nu', 'ny' and 'nd' subfields of the 'dynamics' field of the model. These define the structure of the regressors and therefore the dynamical behaviour of the system. 'ny' is a `num_out * num_out` square matrix which contains the relationships of each output (each row) with all the other outputs. Each entry `ij` indicates the number of past values associated to the output `j` used in the regressor of the output `i`. 'nu' is a `num_out * num_in` matrix defining the relationships between the outputs and each input. Each entry `ij` indicates the number of past values associated to the input `j` used in the regressor of the output `i`. Finally 'nd' is a `num_out * num_in` matrix defining the delay of the inputs respect to the outputs. Each entry `ij` indicates the delay of the input `j` used in the regressor of the output `i`.

### See also

get

# build_regressors

### Purpose

build the regresors for a MIMO system

### Synopsis

```
[in,out]=build_regressors(m,z,subset);
```

### Description

`[in,out]=build_regressors(m,z,subset);` given the inputs outputs `z`=[outputs inputs], computes `subset` regressors.

### Remarks

For z=[y u]; ino(t)=[y1(t-1),y1(t-2),...,y1(t-ny(o,1)),y2(t-1),y2(t-2),...,y2(t-ny(o,2)), ... u1(t-nd(O,1)),u1(t-1-nd(O,1)),...,u1(t-nu(o,1)-nd(O,1)), ...] outo(t)=yo(t) It works even if y1 ... are all vectors (useful for the computation of the derivatives)

# check

### Purpose

checks the consistency of all the fields of a external object

### Synopsis

`check(m)`

### Description

`check(m)` check the consistency of a external object.

### Remarks

This function has been provided in order to make the external class more rouboust since it checks a external object for consistency of the data defined in its fields. If a inconsistency is found, an error message is displayed.

# display

### Purpose

Display an object of class external

### Synopsis

```
display(m)
```

### Description

`display(m)` Display the object m of class external

# error

### Purpose

computes the mean square error of the model

### Synopsis

```
err=error(m,j)
err=error(m,data,j)
```

### Description

`err=error(m,j)` returns the mean square error of the model respect the output `j`.

`err=error(m,data,j)` get the error of the model respect the data `data` and the output `j`. `data` must be an object of the class dataset.

### Remarks

This function evaluates the mean square error of the model respect to the reference set of values given as output data. This is performed by the function by calling the 'eval' method, obtaining the extimated output of the model and comparing it with the reference one.

### See also

simulation_error

# errordb

**Purpose**

Graphical User Interface for inspecting the errors of the mappings

**See also**

# get

### Purpose

gets the value of the attribute of an object

### Synopsis

```
d = get(m,label,options)
```

### Description

`d = get(m,label,options)` gets the value of the attribute `label` associated to the object `m` and return it inside `d`. The following codes are recognised:

| | |
|---|---|
| `name`: | returns the name of the object |
| `n_in`: | returns the number of inputs of the system |
| `n_out`: | returns the number of outputs of the system |
| `data`: | returns the data set linked to the system |
| `T`: | returns the sampling period of the discrete system, 0 if continuous |
| `mapping`: | returns the array of mappings |
| `links`: | returns the links array which defines which output is computed by which mapping |
| `limits`: | returns the bounds of the inputs and outputs |
| `userData`: | returns the 'userData' field |
| `opt`: | returns the 'opt' field |
| `date`: | returns the date of creation of the object |
| `mapfield`: | let the user access directly property of the mappings. When used, opt must contain a 2 elements cell array `n` 'property'. `n` is the index of the mapping and 'property' is the name of the property which has to be accessed. An even more convenient way to access mappings' fields is a achieved through the syntax sn.property. |
| `numparams`: | returns the number of parameters to be optimised |
| `params`: | returns the parameters to be optimised |
| `dynamics`: | returns the dynamical description of the regressors |
| `horiz_length`: | returns the width of the time window used to compute the outputs |

### See also

set

# identify

### Purpose

Identify the system from data

### Synopsis

```
m = identify(m)
m = identify(m,j,options)
```

### Description

`m = identify(m)` performs the identification of the model `m` using the data embedded inside the object for all the outputs of the system, using the defautl values of the identification methods.

`m = identify(m,j,options)` performs the identification of the model `m` using the data embeded inside the object for the outputs listed in the array `j`. Use `options` in order to specify the identification method and options.

### See also

eval

# regress

### Purpose

extracts the regressors from the data

### Synopsis

```
[r_in,r_out]=get_regressors(m)
[r_in,r_out]=get_regressors(m,j)
```

### Description

`[r_in,r_out]=get_regressors(m)` extracts the regressors from the model `m`, for all the outputs of the system.

`[r_in,r_out]=get_regressors(m,j)` extracts the regressors from the model `m`, for all the outputs of the system defined in the array `j`.

### Remarks

This function uses the dynamical system representation stored in the 'dynamics' field to extract the regressors form the input-output data stored in the 'data' field. The function returns the input regressors `r_in` in a cell array. Each element of the cell array is a regressor associated with an output. Each regressor is an array of size `size_regressor * num_exaples`, where `size_regressor` depends from the structure of the regressor defined in the dynamical description of the system, and `num_exaples` is the number of exaples given in the input-output data set. `r_in` is a cell array returing the corrispondent output of each regressor taken from the output data set. Each entry in the cell array is a vector of size `1 * num_exaples`.

# set

### Purpose

Set object properties

### Synopsis

```
set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)
m_out = set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)
```

### Description

`set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)` sets the value of the attribute `AttrName1`, `AttrName2`, ... associated to the object `m`.

`m_out = set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)` sets the value of the attribute `AttrName1`, `AttrName2`, ... associated to the object `m` and return the modified model inside m_out. The original model is not modified. The following `AttrName` are recognised:

| | |
|---|---|
| `name`: | sets the name of the system |
| `n_in`: | sets the number of inputs of the system |
| `n_out`: | sets the number of outputs of the system |
| `data`: | sets the data set linked to the system |
| `T`: | sets the sampling period of the discrete system, 0 if continuous |
| `mapping`: | sets the array of mappings |
| `links`: | sets the links array which defines which output is computed by which mapping |
| `limits`: | sets the limits of the domain of definition of the mapping. The value must be a `2*n_in` matrix where the upper and lever saturation level are |
| `userData`: | sets the 'userData' field |
| `opt`: | sets the 'opt' field |
| `date`: | sets the date of creation of the object |
| `mapfield`: | let the user access directly property of the mappings. When used, `value` must contain a 3 elements cell array `n` 'property' `mapvalue`. `n` is the index of the mapping, 'property' is the name of the property which has to be changet and `mapvalue` is the new value. An even more convenient way to change mappings' fields is a achieved through the syntax sn.property = mapvalue. |
| `params`: | sets the parameters to be optimised |
| `dynamics`: | sets the dynamics of the system (please use add_dynamics instead) |

### See also

get

# simulate

### Purpose

simulates the model

### Synopsis

```
[res,m]=simulate(m)
[res,m]=simulate(m,time)
```

### Description

`[res,m]=simulate(m)` simulate the model `m` using the input data.

`[res,m]=simulate(m,time)` simulate the model `m` for the time horizon `time`

### Remarks

This function performs a simulation of the model `m` using the input data and feeding back the results of the simulation to calculate at each step the regressors. The result of the simulation `res` is a `simulation_time * num_outputs` array.

# simulation_error

### Purpose

computes the mean square error of the simulated model

### Synopsis

```
m=simulation_error(m,j,'plot')
m=simulation_error(m,data,j)
```

### Description

`m=simulation_error(m,j,'plot')` get the simulation error of the model `m` respect the output `j`. If the instead of the number of the output the label 'all' is given to the function the error respect all the outputs is returned in an array. The optional label `plot` may be specified if a plot of the original data and the simulation is desired.

`m=simulation_error(m,data,j)` get the error of the simulated model `m` respect the output `j` and the data `data` input-output set. `data` must be a structure defining the 'dat_in' and 'dat_out' fields which contain the input output data. If the instead of the number of the output `j` the label 'all' is given to the function the error respect all the outputs is returned in an array. The optional label ¡code¿'plot'¡/code¿ may be specified if a plot of the original data and the simulation is desired.

### Remarks

This function performs a simulation of the model using the `simulate` method and then calculates the mean square error by comparing the simulated output with the value of the output contained in the output data set.

### See also

error

## 3.3 Low Level Routines

### 3.3.1 `Mapping` Abstract Class

The `mapping` class is used to define the modeling of a process. Since, as describe in section , it is possible to choose among different possible alternatives for defining the mapping between a specified number of inputs and outputs, the `mapping` abstract class, defines all the features which are common to all the possible mapping, leaving the definition of the details, which are dependent from the particular descriptor, to its subclasses which define the different approaches which can be used for process modeling. The `mapping` class stores in its attributes the number of input and the number of output parameters of the mapping, as well as their limits. The main abstract methods defined by the class are `identify`, for performing the identification of the model, `eval` which is used to evaluate the mapping given the input values, and `jacob_inputs` and `jacob_params`, for computing the jacobian of the model respect the inputs and the parameters. The mapping could be normalizing and denormalizing as needed. All these are defined as abstract methods, that is define only the calling convention and the returned values, and need to be implemented at the level of the subclasses since the operations which they perform depend on the particular descriptor.

Virtually every type of system representation can be implemented as a subclass of the `mapping` class. At present the following mappings are available in the toolbox:

- linear systems
- Mamdani fuzzy systems
- Takagi-Sugeno fuzzy systems
- lazy learning local modeling (integrated from [**?**])
- mixture of experts modeling

New mappings are easily added by defining other subclasses of the `mapping` class.
With the exception of the linear models, all the mappings may be used to define nonlinearities. They can be seen as different ways of parametrising nonlinearities. These methods have been shown to be universal approximators for certain classes of functions. For example fuzzy logic systems are universal approximators for continuous functions defined on compact sets [**?**]. This means that these methods are equivalent with the respect to the nonlinearity which they approximate. From the process point of view it is the nonlinearity that matters, not the way in which it is parametrized. However from the point of view of the designer, which parameterization is used can be very important.

# mapping

### Purpose

Constructor for the mapping abstract class

### Synopsis

```
m=mapping
m=mapping(name,n_in,n_out)
```

### Description

`m=mapping` creates a new general mapping skeleton

`m=mapping(name,n_in,n_out)` creates a new general mapping with `n_in` inputs and `n_out` outputs. The name `name` is associated to it.

### Methods

- `check` checks the consistency of all the field of a mapping object
- `denormalise` normalises the model M
- `display` Display an object of class mapping
- `error` Computes the squared error of the mapping
- `eval` computes the value of the mapping for some input
- `get` gets the value of the attribute of an object
- `identify` Identify the mapping from data (abstact method)
- `jacob_inputs` computes the jacobian of the model
- `jacob_params` computes the jacobian of the model
- `lev_marq` Performs a Levenbergh Marquardt optimisation of the mapping
- `normalise` normalises the model M
- `set` Set object properties

### See also

# check

### Purpose

checks the consistency of all the field of a mapping object

### Synopsis

`check(m)`

### Description

`check(m)` check the consistency of a mapping object.

### Remarks

This function has been provided in order to make the mapping class more robust since it checks a mapping object for consistency of the data defined in its fields. If a inconsistency is found, an error message is displayed.

# denormalise

### Purpose

normalises the model M

### Synopsis

```
m=normalise(m)
```

### Description

`m=normalise(m)` denormalises the model `m`

### Remarks

This function denormalises a mapping in order to make fit its limits to the original values before normalisation.

# display

**Purpose**

Display an object of class mapping

**Synopsis**

```
display(m)
```

**Description**

`display(m)` Display the object m of class mapping

**See also**

# error

### Purpose

Computes the squared error of the mapping

### Synopsis

```
out=error(m,in,out)
```

### Description

`out=error(m,in,out)` for each output, the mean squared error is computed, comparing predictions made on th basis of inputs `in` to reference outputs `out`.

### See also

# eval

### Purpose

computes the value of the mapping for some input

### Synopsis

```
out=eval(m,regressor,j)
```

### Description

`out=eval(m,regressor,j)` returns the value `out` of the output(s) `j` of mapping `m` given the input(s) `x`.

# get

### Purpose

gets the value of the attribute of an object

### Synopsis

```
d = get(m,label,opt)
```

### Description

`d = get(m,label,opt)` gets the value of the attribute `label` associated to the object `m` and return it inside `d`. The following codes are recognised:

| | |
|---|---|
| `name`: | returns the name of the object |
| `n_in`: | returns the number of inputs of the mapping |
| `n_out`: | returns the number of outputs of the mapping |
| `userData`: | returns the 'userData' field |
| `opt`: | returns the 'opt' field |
| `date`: | returns the date of creation of the object |
| `limits`: | returns the limits of the mapping |
| `optimparams`: | returns the indices of the parameters to be optimised |
| `numparams`: | returns the number of parameters to be optimised |

### See also

set

# identify

### Purpose

Identify the mapping from data (abstact method)

### Synopsis

```
m = identify(m,in,out,options)
```

### Description

`m = identify(m,in,out,options)` performs the identification of the model `m` using data inside `in` and `out`. Use `options` in order to specify the identification method and options.

# jacob_inputs

### Purpose

computes the jacobian of the model

### Synopsis

```
jacob_inputs(m,x)
jacob_inputs(m,x,j)
```

### Description

`jacob_inputs(m,x)` computes for each output the jacobian of the model with respect to the x (the input of the mapping) at point `x`.

`jacob_inputs(m,x,j)` computes for output J the jacobian of the model with respect to the x (the input of the mapping) at point `x`.

### Remarks

The procedure for computing the jacobian depends on the underlying representation of the model. The result `out` is an array of size number of outputs * number of inputs containing the derivatives of the outputs with respect to the inputs.

# jacob_params

### Purpose

computes the jacobian of the model

### Synopsis

```
jacob_params(m,x)
jacob_params(m,x,j)
```

### Description

`jacob_params(m,x)` computes for each output the jacobian of the model with respect to the parameters at input `x`.

`jacob_params(m,x,j)` computes for output(s) `j` the jacobian of the model with respect to the parameters at input `regressor`.

# lev_marq

### Purpose

Performs a Levenbergh Marquardt optimisation of the mapping

### Synopsis

```
function m = lev_marq(m, in, out, cost)
```

### Description

`function m = lev_marq(m, in, out, cost)` Optimises the mapping `m` with respect to input/outpu couples ¡CODE¿in/out¡/CODE¿. This method is mainly used internaly. Identify should be prefered.

### See also

identify

# normalise

**Purpose**

normalises the model M

**Synopsis**

```
m=normalise(m)
```

**Description**

`m=normalise(m)` normalises the model `m`

**Remarks**

This function normalizes a mapping in order to make fit its limits to the interval [0,1]

# set

### Purpose

Set object properties

### Synopsis

```
set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)
m_out = set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)
```

### Description

`set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)` sets the value of the attribute `AttrName1`, `AttrName2`, ... associated to the object `m`.

`m_out = set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)` sets the value of the attribute `AttrName1`, `AttrName2`, ... associated to the object `m` and return the modified model inside m_out. The original model is not modified. The following `AttrName` are recognised:

| | |
|---|---|
| `name`: | sets the name of the object |
| `n_in`: | sets the number of inputs of the model. This sets the saturation level of the inputs to -Inf, Inf. |
| `n_out`: | sets the number of outputs of the model |
| `userData`: | sets the 'userData' field |
| `opt`: | sets the 'opt' field |
| `date`: | sets the date of creation of the object |
| `limits`: | sets the limits of the domain of definition of the mapping. The value must be a `2*n_in` matrix where the upper and lever saturation level are defined. |
| `optimparams`: | sets the indices of the parameters to be optimised |

### See also

get

### 3.3.2 `Lazy` Class

the lazy learning is a local modeling technique that defers processing of the training data until a query explicitly needs to be answered. This means that all the training data needs to be stored in memory and accessed when a query is made. Given two variables $\mathbf{x} \in \Re_m$ and $y \in \Re$, and considering the mapping $f : \Re_m \rightarrow \Re$ which is known only through a set of examples $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, which could be affected by noise, the aim is to find the best possible estimate $\hat{y} = \beta\mathbf{q}$ associated the the query $\mathbf{q}$ using a model linear in the parameters $\beta$. This can be done by minimizing the following cost expression

$$C(\mathbf{q}) = \sum_{i=1}^{n}[(y_i - \mathbf{x}_i^T\beta)^2 K(d(\mathbf{x}_i, \mathbf{q}))]$$

where $d(\mathbf{x}_i, \mathbf{q})$ is the distance from the query point of the $i^{th}$ example, and $K(.)$ is a weighting function, or kernel function, which is used to calculate a weight for that data point from the distance. In this way the linear local model can be specialized to the query by emphasizing nearby points and discarding more distant ones. In matrix notation the solution of the previous weighted least squares problem is given by:

$$\beta = (\mathbf{X}^T\mathbf{W}^T\mathbf{W}\mathbf{X})^{-1}\mathbf{X}^T\mathbf{W}^T\mathbf{W}\mathbf{y}$$

where $\mathbf{X}$ is a matrix whose $i^{th}$ row is $x_i^T$, $\mathbf{y}$ is a vector whose $i^{th}$ element is $y_i$, $\mathbf{W}$ is a diagonal matrix whose $i^{th}$ diagonal elements is $w_{ii} = \sqrt{k(d(\mathbf{x}_i, \mathbf{x}_q))}$. Once the local approximation of the polynomial has been computed the prediction is given by

$$\hat{y} = \beta\mathbf{q}$$

In these systems, no parameter identification process is needed since the answer is produced upon request of information, and it is calculated starting from the database of examples which must be retained into memory.
The `lazy` class is based on the *lazy learning toolbox* [**?**, **?**], whose functions have been integrated into an implementation of the `mapping` class.
The complete documentation of the *lazy learning toolbox* is joined in the annexes.

# lazy

### Purpose

Constructor for the lazy abstract class

### Synopsis

```
m=lazy
m=lazy(name,n_in,n_out)
```

### Description

`m=lazy` creates a new general lazy skeleton

`m=lazy(name,n_in,n_out)` creates a new general lazy with `n_in` inputs and `n_out` outputs. The name `name` is associated to it.

### Remarks

This class implements the famimo class and allows to describe dynamical systems using a 'lazy' representation. It works in a very simple way. The input-output data is stored in a database. When a prediction of the putput is needed a local model is fitted through the closest points of the query contained in the database.
The elements that describe the lazy model associated with each output are defined by the following attributes:

`id_par`: identification parameters
`cmb_par`: model combination parameters

### Methods

- `check` checks the consistency of all the field of a lazy object
- `denormalise` denormalises the model M
- `display` Display an object of class lazy
- `eval` computes the value of the lazy for some input
- `get` gets the value of the attribute of an object
- `identify` Identify the lazy model from data
- `jacob_inputs` computes the jacobian of the model
- `jacob_params` computes the jacobian of the model
- `normalise` normalises the model M
- `set` Set object properties

### See also

mapping

# check

### Purpose

checks the consistency of all the field of a lazy object

### Synopsis

`check(m)`

### Description

`check(m)` check the consistency of a lazy object.

### Remarks

This function has been provided in order to make the lazy class more robust since it checks a lazy object for consistency of the data defined in its fields. If a inconsistency is found, an error message is displayed.

# denormalise

### Purpose

denormalises the model M

### Synopsis

```
m=denormalise(m)
```

### Description

`m=denormalise(m)` denormalises the model `m`

### Remarks

This function denormalises a lazy in order to make fit its limits from [0,1] to the original values, before normalisation.

# display

### Purpose

Display an object of class lazy

### Synopsis

```
display(m)
```

### Description

`display(m)` Display the object `m` of class lazy

# eval

### Purpose

computes the value of the lazy for some input

### Synopsis

```
out=eval(m,regressor)
out=eval(m,regressor,j)
```

### Description

`out=eval(m,regressor)` returns the value `out` of the output(s) of lazy `m` given the input(s) `x`.

`out=eval(m,regressor,j)` returns the value `out` of the output(s) `j` of lazy `m` given the input(s) `x`.

### Remarks

This function returns the output of system as computated by the model, given a regressor.

# get

### Purpose

gets the value of the attribute of an object

### Synopsis

```
d = get(m,label,opt)
```

### Description

`d = get(m,label,opt)` gets the value of the attribute `label` associated to the object `m` and return it inside `d`. The following codes are recognised:

| | |
|---|---|
| `name`: | returns the name of the object |
| `n_in`: | returns the number of inputs of the lazy |
| `n_out`: | returns the number of outputs of the lazy |
| `userData`: | returns the 'userData' field |
| `opt`: | returns the 'opt' field |
| `date`: | returns the date of creation of the object |
| `limits`: | returns the limits of the lazy |
| `optimparams`: | returns the indices of the parameters to be optimised |
| `numparams`: | returns the number of parameters to be optimised |
| `id_par`: | returns the identification paramenters |
| `cmb_par`: | returns the combination paramenters of the models |
| `examples_x`: | the database of the input examples |
| `examples_y`: | the database of the output examples |

### See also

set

# identify

### Purpose

Identify the lazy model from data

### Synopsis

```
m = identify(m,in,out,options)
```

### Description

`m = identify(m,in,out,options)` performs the identification of the model `m` using data inside `in` and `out`. Define the structure `options` in order to specify the identification method and options.
Accepted fields are:

| | |
|---|---|
| `id_par`: | identification parameters |
| `cmb_par`: | model combination parameters |

# jacob_inputs

### Purpose

computes the jacobian of the model

### Synopsis

```
jacob_inputs(m,x)
jacob_inputs(m,x,j)
```

### Description

`jacob_inputs(m,x)` computes for each output the jacobian of the model with respect to the x (the input of the lazy) at point `x`.

`jacob_inputs(m,x,j)` computes for output J the jacobian of the model with respect to the x (the input of the lazy) at point `x`.

### Remarks

The procedure for computing the jacobian depends on the underlying representation of the model. The result `out` is an array of size number of outputs * number of inputs containing the derivatives of the outputs with respect to the inputs.

# jacob_params

### Purpose

computes the jacobian of the model

### Synopsis

```
jacob_params(m,x)
jacob_params(m,x,j)
```

### Description

`jacob_params(m,x)` computes for each output the jacobian of the model with respect to the parameters at input `x`. It returns an empty array.

`jacob_params(m,x,j)` computes for output(s) `j` the jacobian of the model with respect to the parameters at input `regressor`. It returns an empty array.

### Remarks

This function does not have any meaning for lazy systems, and it is provided only for compatibility reasons.

# normalise

**Purpose**

normalises the model M

**Synopsis**

```
m=normalise(m)
```

**Description**

`m=normalise(m)` normalises the model `m`

**Remarks**

This function normalizes a lazy in order to make fit its limits to the interval [0,1]

# set

### Purpose

Set object properties

### Synopsis

```
set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)
m_out = set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)
```

### Description

`set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)` sets the value of the attribute `AttrName1`, `AttrName2`, ... associated to the object `m`.

`m_out = set(m,'AttrName1',AttrValue1,'AttrName2',AttrValue2,...)` sets the value of the attribute `AttrName1`, `AttrName2`, ... associated to the object `m` and return the modified model inside m_out. The original model is not modified. The following `AttrName` inherided from the mapping class are recognised:

| | |
|---|---|
| `name`: | sets the name of the object |
| `n_in`: | sets the number of inputs of the model. This sets the saturation level of the inputs to -Inf, Inf. |
| `n_out`: | sets the number of outputs of the model |
| `userData`: | sets the 'userData' field |
| `opt`: | sets the 'opt' field |
| `date`: | sets the date of creation of the object |
| `limits`: | sets the limits of the domain of definition of the lazy. The value must be a `2*n_in` matrix where the upper and lever saturation level are defined. |
| `optimparams`: | sets the indices of the parameters to be optimised |

The following codes specific to the lazy class are recognised:

| | |
|---|---|
| `id_par`: | identification parameters |
| `cmb_par`: | model combination parameters |

The identification parameter can assume the following forms:

$$id\_par = [idmidMvalM] \tag{3.2}$$

where [idmX,idMX] is the range in which the best number of neighbors is searched when identifying the local model of degree X and where valMX is the maximum number of neighbors used in validation for the model of degree X. This means that the model of degree X identified with k neighbors, is validated un the first v neighbors, where v=min(k,valMX).

$$id\_par = [idmidM] \tag{3.3}$$

where idmX and idMX have the same role they have in point 1, and valMX is by default set to idMX: each model is validated on all the neighbors used in identification.

$$id\_par = [c] \tag{3.4}$$

Here idmX and idMX are obtained according to the following fotmulas:

$$idmX = 3 * no_parX * cX idMX = 5 * no_parX * cX \tag{3.5}$$

where no_parX is the number of parameter of the model of degree X. Recommended choice: cX = 1. As far as the valMX are concerned, they get the default value as in point 2. The default value for cmb_par is 1 as it will be clear from what follows. If given, cmb_par can assume the following to forms:

$$cmb\_par = [cmb] \tag{3.6}$$

where cmbX is the number of models of degree X that will be included in the local combination. Each local model will be therfore a combination of "the best cmb0 models of degree 0", "the best cmb1 models of degree 1", and "the best cmb2 models of degree 1" identified as specified by id_par.

$$cmb_par = [cmb] \tag{3.7}$$

where cmb is the number of models that will be combined, disregarding any constraint on the degree of the models that will be considered. Each local model will be therfore a combination of "the best cmb models", identified as specified by id_par.

### 3.3.3 `Taksug` Class

the Takagi-Sugeno fuzzy systems are a different type of fuzzy systems which use rules of the form:

$$\mathcal{R}^{(l)} : \text{IF } x_1 \text{ IS } A_1^{(l)} \text{ AND } \ldots \text{AND } x_n \text{ IS } A_n^{(l)} \text{ THEN } y = h^{(l)}(x)$$

where $\mathbf{x_i} \in \Re$ are the inputs (antecedent) variables, and $y \in \Re$ is the output (consequent). $A_i^{(l)}$ are antecedent fuzzy sets of the $l$th rule, defined by a membership function

$$\mu_{A_i^{(l)}}(x_i) : \Re \rightarrow [0, 1]$$

In many cases the output function $h^{(l)}(x)$ is a linear combination of the input variables plus a constant term.

$$h^{(l)}(x) = a_0^{(l)} + a_1^{(l)} x_1 + \ldots + a_n^{(l)} x_n$$

In the special case when all the $l_1 \ldots l_n = 0$, and therefore the consequents are constant functions, this becomes a special case of the Mandami system with the consequent fuzzy sets being fuzzy singletons. The system output is a weighted average of the individual rule outputs, similar to the fuzzy-mean defuzzification formula:

$$y = \sum_{l=1}^{M} \frac{\mu_{A^{(l)}}(x)}{\sum_{k=1}^{M} \mu_{A^{(k)}}(x)} h^{(l)}(x)$$

where the weights $\mu_{A^{(l)}}(x)$ are computed according to

$$\mu_{A^{(l)}}(x) = \prod_{i=1}^{n} \mu_{A_i^{(l)}}(x_i)$$

This approach allows to model a system by means of the decomposition of a nonlinear system into a collection of local linear models. It allows a more accurate representation of systems, since the rules are usually simple linear subsystems, and not constant values like in the case of the Mamdani fuzzy systems. Since this approach imposes to structure the problem in a series of local models, Takagi-Sugeno models can be more easily constructed from numerical data than Mamdani models, which are essentially structure free.

The Features of this class are similar to the ones defined in the `mamdani` class. It is possible to add and delete rules, and to define their shape. Since these type of models are usually constructed from numerical data several identification procedures are available for performing the parameter identification procedure.

# taksug

### Purpose

class for implementing the taksug fuzzy functions

### Synopsis

```
m=taksug(varargin)
m=taksug(taksug_obj)
m=taksug(structure)
m=taksug(name, n_inputs, n_outputs)
```

### Description

`m=taksug(varargin)` returns an empty taksug object.

`m=taksug(taksug_obj)` returns a copy of the `taksug_obj` object.

`m=taksug(structure)` initializes a new taksug object with the appropriate fields set according to the values defined in the fields of `structure`.

`m=taksug(name, n_inputs, n_outputs)` initializes a new taksug object called `name`, with number of inputs `n_inputs` and number of outputs `n_outputs`.

### Remarks

The taksug class is a child of the mapping class and allows the definition of a static system using a fuzzy rule based Takagi-Sugeno (TS) representation. The class inherits all the fields of the mapping class and defined a series of new attibutes required for implementing the TS representation of a model. At the top level the following fields are defined:

| | |
|---|---|
| `n_rules`: | defines the number of fuzzy rules of the model |
| `model_code`: | defines the structure of the fuzzy rules |
| `centers`: | position of the center of each rule |
| `ivariances`: | inverse of the projected covariance matrix of the cluster associated to each rule. In other words, it is the quadratic matrix defining the local metric (or the shape) of the rule |
| `linears`: | hyperplane associated to each rule (consequence of the rule) |
| `m`: | index of the fuzzyness of the model |
| `mfs`: | description of the fuzzy sets while they are defined along each dimension. |
| `rls`: | description of the rules base. |

### Methods

- `add_rules` adds specified number of rules
- `add_sets` Add fuzzy sets along one or more dimensions of the input space
- `check` checks the consistency of all the field of a taksug object
- `denormalize` denormalizes the model
- `display` Display an object of class taksug

- `drawcontrols` Draw controls which allow to control the plot behaviour
- `eval` computes the value of the model
- `fit_linears` identifies the consequent part of a fuzzy model from data
- `get` gets the value of the attribute of an object
- `identify` identifies a fuzzy model from data
- `interp_model` Computes a local fuzzy combination of the consequents of the rules
- `jacob_inputs` computes the jacobian of the model
- `jacob_params` computes the jacobian of the model
- `lev_marq` Performs a Levenbergh Marquardt optimisation of the mapping
- `membership` returns the membership of a rule
- `normalise` normalises the model M
- `plot` performs a plot of the a taksug model
- `plot` performs a plot of the a taksug model
- `rem_rules` remove the specified rules from the taksug object
- `set` set object properties
- `sets_grid` Remove every fuzzy sets and replace them by evenly spaced ones

# add_rules

### Purpose

adds specified number of rules

### Synopsis

```
add_rules(m,n)
add_rules(m,n,model_code)
```

### Description

`add_rules(m,n)` add `n` rules to the model `m`. When rules are added the `optimparams` property of the model is set to contain the parameters of the newly added rules. This allow to optimise by default every parameters of a TS system.

`add_rules(m,n,model_code)` add `n` rules to the model `m`, `model_code` is a cell array x y z. x is the type of TS model ('crossproduct' or 'projected'). For 'crossproduct' models, y is the type of the membership functions ('gaussian', 'inversedist':

$$\frac{1}{distanceofmahalanobis^{\frac{1}{m-1}}} \tag{3.8}$$

and z is the type of the consequents ('linear'). For 'projected' systems, y specify the way rules are initially positioned ('standard': no positionment, 'rulegrid': rules evenly disposed on a grid defined by the fuzzy sets) and z is the type of the consequents ('linear'). Only rules of the same type can co-exist in a TS model.

### Remarks

This function is used to specify the number and shape of the rules associated to the TS model.

# add_sets

### Purpose

Add fuzzy sets along one or more dimensions of the input space

### Synopsis

```
m = add_sets(m,i,n,t)
```

### Description

`m = add_sets(m,i,n,t)` adds `n` sets to the dimension `i` of the Takagi Sugeno Function `m`. The sets are of type t Accepted values for `t` are 'constant, 'trapezoidal','gaussian' and 's-shaped'. This method is used to define the fuzzy sets for the 'projected' type of fuzzy model.

### See also

add_rules

# check

### Purpose

checks the consistency of all the field of a taksug object

### Synopsis

`check(m)`

### Description

`check(m)` check the consistency of a taksug object.

### Remarks

This function has been provided in order to make the taksug class more robust since it checks a taksug object for consistency of the data defined in its fields. If a inconsistency is found, an error message is displayed.

# denormalize

### Purpose

denormalizes the model

### Synopsis

```
m=denomalise(m)
```

### Description

`m=denomalise(m)` denormalzes the model `m`.

### Remarks

This function can be applied to denormalize a model that has been previously normalised with the function `normalize` restoring the original shape of the local representation of the model.

### See also

normalize

# display

**Purpose**

Display an object of class taksug

**Synopsis**

```
display(m)
```

**Description**

`display(m)` Display the object m of class taksug

**See also**

# drawcontrols

### Purpose

Draw controls which allow to control the plot behaviour

### Synopsis

```
drawcontrols(m,f,a,bounds)
```

### Description

`drawcontrols(m,f,a,bounds)` binds a series of controls drawed inside `bounds`, to the figure `f` and the axes `a`.

### See also

plot

# eval

### Purpose

computes the value of the model

### Synopsis

```
out=eval(m,x,j)
```

### Description

`out=eval(m,x,j)` returns the value `out` for the output(s) `j` of model `m` given the input `x`.

### Remarks

This function returns the output of system as computated by the model, given a regressor.

# fit_linears

### Purpose

identifies the consequent part of a fuzzy model from data

### Synopsis

```
m=fit_linears(m,in,out,type)
```

### Description

`m=fit_linears(m,in,out,type)` identify model `m` on the basis of the data inside `in` and `out`.`type` is used to specify the type of the fit. 0 is the defaults and leads to a classical LMS fit. 1 leads to a local fit.

# get

### Purpose

gets the value of the attribute of an object

### Synopsis

```
d = get(m,label)
```

### Description

`d = get(m,label)` gets the value of the attribute `label` associated to the object `m` and return it inside `d`. The following codes are recognised:

| | |
|---|---|
| `name`: | returns the name of the object |
| `n_in`: | returns the number of inputs of the mapping |
| `n_out`: | returns the number of outputs of the mapping |
| `userData`: | returns the 'userData' field |
| `opt`: | returns the 'opt' field |
| `date`: | returns the date of creation of the object |
| `limits`: | returns the limits of the mapping |
| `optimparams`: | returns the indices of the parameters to be optimised |
| `numparams`: | returns the number of parameters to be optimised |
| `n_rules`: | returns the number of rules associated to the taksug model |
| `model_code`: | returns the code of the model associated to the taksug model |
| `m`: | returns the fuzziness associated to the taksug model |
| `centers`: | returns the positions of the centers of the rules associated to the taksug model |
| `ivariances`: | returns the invariances of the rules associated to the taksug model |
| `linears`: | returns the linears of associated to the taksug model |
| `rls`: | returns the fuzzy sets associated to the taksug model |
| `mfs`: | returns the definition of the rules of the taksug model |
| `params`: | returns every parameters to be optimised in the shape of a vector |

### See also

set

# identify

### Purpose

identifies a fuzzy model from data

### Synopsis

```
m=identify(m,in,out,options)
```

### Description

`m=identify(m,in,out,options)` identify model `m` on the basis of the data inside `in` and `out` using method specified in `options`. The accepted fields for options are:

| | |
|---|---|
| `method`: | identification method(see below). |
| `n_rules`: | number of rules. |
| `min_n_rules`: | min. nb. of rules (incremental methods only) |
| `max_n_rules`: | max. nb. of rules (incremental methods only) |
| `fuzzyness`: | fuzzyness of the model (usually greater than 1) |
| `tolerance`: | minimal improvement for recursive techniques before the algorithm stops. |
| `seed`: | seed for random generator |
| `lin_fit`: | type of linear fitting: 0: normal, 1: local |
| `rule_type`: | rule type code (see `add_rules` for more info) |

Methods of identification are:

| | |
|---|---|
| `cluslms`: | GK clustering followed by a LMS fit |
| `fmclust`: | Robert Babuska's identification method |
| `incrsie`: | Incremental identification method (Siemens) |
| `tsgklmxv`: | incremental method developped in IRIDIA |
| `cluslev`: | GK clust. init. and Lev.-Marq. optimisation |
| `randlm`: | random init. and Lev.-Marq. optimisation |

# interp_model

### Purpose

Computes a local fuzzy combination of the consequents of the rules

### Synopsis

```
l=interp_model(m,x,j)
```

### Description

`l=interp_model(m,x,j)` computes for `taksug` model `m`, a fuzzy combination of the consequents of the rules at point `x`. `l` is equal to a linear combination of the parameters of the consequents for each rule. The coefficient of the linar combination are equal to the membership of `x` to the different rules.

### See also

also jacob_params

# jacob_inputs

### Purpose

computes the jacobian of the model

### Synopsis

```
jacob_inputs(m,x)
jacob_inputs(m,x,j)
```

### Description

`jacob_inputs(m,x)` computes for each output the jacobian of the model with respect to the x (the input of the linear) at point `x`.

`jacob_inputs(m,x,j)` computes for output J the jacobian of the model with respect to the x (the input of the linear) at point `x`.

### Remarks

The procedure for computing the jacobian depends on the underlying representation of the model. The result `out` is an array of size number of outputs * number of inputs containing the derivatives of the outputs with respect to the inputs.

# jacob_params

### Purpose

computes the jacobian of the model

### Synopsis

```
jacob_params(m,x)
jacob_params(m,x,j)
```

### Description

`jacob_params(m,x)` computes for each output the jacobian of the model with respect to the parameters at input `x`.

`jacob_params(m,x,j)` computes for output(s) `j` the jacobian of the model with respect to the parameters at input `x`.

# lev_marq

### Purpose

Performs a Levenbergh Marquardt optimisation of the mapping

### Synopsis

```
function m = lev_marq(m, in, out, cost)
```

### Description

`function m = lev_marq(m, in, out, cost)` Optimises the mapping `m` with respect to input/outpu couples ¡CODE¿in/out¡/CODE¿. This method is mainly used internaly. Identify should be prefered.

### See also

identify

# membership

### Purpose

returns the membership of a rule

### Synopsis

```
out=membership(m,x,j)
```

### Description

`out=membership(m,x,j)` returns the membership of the rules for the model `m`.

# normalise

### Purpose

normalises the model M

### Synopsis

```
m=normalise(m)
```

### Description

`m=normalise(m)` normalises the model `m`

# plot

### Purpose

performs a plot of the a taksug model

### Synopsis

```
f=plot(m)
f=plot(m,options)
```

### Description

`f=plot(m)` opens a new figure if needed and plots the model `m`. A complete user interface is set to allow advanced visualisation options to be selected.

`f=plot(m,options)` opens a new figure if needed and plots the model `m`. `options` are passed to the drawing engine.

# plot

### Purpose

performs a plot of the a taksug model

### Synopsis

```
f=plot(m)
f=plot(m,data)
f=plot(m,data,options)
```

### Description

`f=plot(m)` opens a new figure if needed and plots the model `m`. A complete user interface is set to allow advanced visualisation options to be selected.

`f=plot(m,data)` opens a new figure if needed and plots the model `m`. The data specified in `data` are plotted on the same graph in order to visually X validate the model.

`f=plot(m,data,options)` opens a new figure if needed and plots the model `m`. The data specified in `data` are plotted on the same graph in order to visually X validate the model. `options` are passed to the drawing engine.

# rem_rules

### Purpose

remove the specified rules from the taksug object

### Synopsis

```
m=rem_rules(m)
m=rem_rules(m,n)
```

### Description

`m=rem_rules(m)` remove all rules from model `m`.

`m=rem_rules(m,n)` remove rules number `n` from model `m`.

### See also

# set

### Purpose

set object properties

### Synopsis

```
m=set(m,label1,value1,label2,value2,...)
```

### Description

`m=set(m,label1,value1,label2,value2,...)` sets the value `value` of the attribute `label` associated to the object `m`. The following codes are recognised:

| | |
|---|---|
| `name`: | sets the name of the object |
| `n_in`: | sets the number of inputs of the model. This sets the saturation level of the inputs to -Inf, Inf. |
| `n_out`: | sets the number of outputs of the model |
| `userData`: | sets the 'userData' field |
| `opt`: | sets the 'opt' field |
| `date`: | sets the date of creation of the object |
| `limits`: | sets the limits of the mapping |
| `optimparams`: | sets the indices of the parameters to be optimised |
| `model_code`: | sets the type of rules associated to the taksug model |
| `m`: | sets the fuzziness associated to the taksug model |
| `centers`: | sets the positions of the centers of the rules associated to the taksug model |
| `center`: | sets the positions of one center of the rules associated to the taksug model. `value` is of the form number value. |
| `ivariances`: | sets the invariances of the rules associated to the taksug model |
| `ivariance`: | sets the positions of one ivariance of the rules associated to the taksug model. `value` is of the form number value. |
| `linears`: | sets the linears of associated to the taksug model |
| `linear`: | sets the positions of one linear of the rules associated to the taksug model. `value` is of the form number value. |
| `rls`: | sets the fuzzy sets associated to the taksug model |
| `rl`: | sets one fuzzy set associated to the taksug model |
| `mfs`: | sets the definition of the rules of the taksug model |
| `mf`: | sets the definition of one rule of the taksug model |
| `params`: | sets every parameters to be optimised providing a vector |
| `optimselect`: | basically achieves the same result as `optimparams` but lets the user perform a grouped selection of the parameters (all the consequents parameters, all the centers,...), `value` is 4 bits binary number dcba. a is set for selecting the centers, b for the ivariances, c for the consequents except the offsets, d for the offsets |

### Remarks

General purpose function for setting the values of the fields of a taksug object. For setting the rules, use the `add_rules` and `rem_rules` function. This method extends the 'set' method defined in the mpping class.

**See also**

get, add_rules, rem_rules

# sets_grid

### Purpose

Remove every fuzzy sets and replace them by evenly spaced ones

### Synopsis

```
m = sets_grid(m, d1, d2, d3, d4..., t)
```

### Description

`m = sets_grid(m, d1, d2, d3, d4..., t)` places `d1` sets of type `t` along the dimension `1`, `d2` sets along the dimension `2`, etc... The position of the sets is computed on the basis of the limits attribute. The previously defined rules and sets are removed.

### See also

add_setss

### 3.3.4 `Mamdani` Class

Mamdani fuzzy systems, also known as linguistics fuzzy systems, are composed by four different elements: a fuzzifier, a rule base, an inference engine, and a defuzzifier. The inputs are transformed in fuzzy numbers by the fuzzifier, which converts numeric values $x^\star$ into fuzzy sets $\mu_A(x)$. Usually singleton fuzzifiers are the predominant ones, since their use simplifies the computations in the fuzzy system considerably. This means that the inputs of the fuzzy system are real numbers and not fuzzy sets. The fuzzy rule base consists of fuzzy IF-THEN rules and membership functions, characterizing the fuzzy set. More precisely a fuzzy rule base, $\mathcal{R}$, is a set of rules $\mathcal{R}^{(l)}$, $l = 1, 2, \ldots, K$ of the form:

$$\mathcal{R}^{(l)} : \text{IF } x_1 \text{ IS } A_1^{(l)} \text{ AND } \ldots \text{AND } x_n \text{ IS } A_n^{(l)} \text{ THEN } y \text{ IS } B^{(l)}$$

where $\mathbf{x_i} \in \Re$ are the input (antecedent) variables, and $y \in \mathcal{Y} \in \Re$ is the output (consequent). $A_i^{(l)}$ and $B^{(l)}$ are *linguistic terms* (labels) defined by fuzzy sets $\mu_{A_i^{(l)}}(x_i) : \Re \rightarrow [0, 1]$ and $\mu_{B^{(l)}}(y) : \mathcal{Y} \rightarrow [0, 1]$. The inference engine takes the input of the fuzzy system, and uses the rule based systems to calculate the output of the system. Each rule is evaluated and the output suggested by the rule is inferred using fuzzy operators. Then the predictions of each rules are aggregated together in a fuzzy set. The defuzzifier has the task of taking transforming this fuzzy set into a numeric value which is the output of the system. These type of fuzzy systems are well suited for encoding imprecise knowledge expressed in the form of IF-THEN rules. These rules are easily understood by the user since they do not imply any mathematical representation. The type of interpolation between the rules depends on their shape and and the choice of the fuzzy logic operators and defuzzification procedures [**?**].

The class defines methods for adding, and removing rules. It is possible to choose the shape of the membership functions by composition of simpler shapes. In this way it is possible to select gaussian, sigmoidal, triangular, and several variations of rectangular membership functions. The inference engine uses .... for calculating the contribution of each rule. Freedom about the defuzzification method is also granted. Center of gravity, center average, and maximum defuzzifiers are implemented in this class.

# mamdani

### Purpose

build a mamdami object

### Synopsis

```
m=mamdani(varargin)
```

### Description

```
m=mamdani(varargin)
```

### Methods

- `add_rules` add specified number of rules
- `add_sets` Add fuzzy sets along one or more dimension of the space
- `check` checks the consistency of all the field of a mamdani object
- `denormalize` denormalises the model
- `display` Display an object of class mamdani
- `eval` computes the value of the model
- `get` gets the value of the attribute of an object
- `identify` identifies a fuzzy model from data
- `jacob_inputs` computes the jacobian of the model
- `jacob_params` computes the jacobian of the model
- `membership` Compute the degree of fullfilment of the rules of a mamdani function
- `normalise` normalises the model M
- `plot` performs a plot of the a mamdani model
- `rem_rules` remove the specified rules from the mamdani object
- `rem_sets` remove the specified sets from the mamdani object
- `rules_parser` parses rules in order to populate the rules base of a mamdani model
- `set` set object properties
- `sets_grid` Remove every fuzzy sets and replace them by evenly spaced ones

### See also

# add_rules

### Purpose

add specified number of rules

### Synopsis

```
add_rules(m,n)
add_rules(m,n,model_code)
add_rules(m,n,model_code,mode)
```

### Description

`add_rules(m,n)` add `n` rules to the model `m`.

`add_rules(m,n,model_code)` add `n` rules to the model `m`, `model_code` is a cell array of two elements x y where x is the T-norm used (only 'product' is currently implemented) and y is defuzzification method ('meancentroid').

`add_rules(m,n,model_code,mode)` mode can take the value 'standard' or 'rulegrid'. In lhe latter case, the position of the rules is initialised in order to fill the space according to the position of the fuzzy sets along the the different dimensions.

### Remarks

This function is used to specify the number and shape of the rules associated to the TS model.

# add_sets

### Purpose

Add fuzzy sets along one or more dimension of the space

### Synopsis

```
m = add_sets(m,i,n,t)
m = add_sets(m,i,n,t)
```

### Description

`m = add_sets(m,i,n,t)` adds `n` sets to the dimension `i` of the Mamdani Function `m`. The sets are of type t Accepted values for `t` are 'constant, 'trapezoidal','gaussian' and 's-shaped'.

`m = add_sets(m,i,n,t)` where `n` where n is a cell array of linguistic values attaches the corresponding sets to the dimension ¡CODE¿i¡CODE¿.

### See also

# check

### Purpose

checks the consistency of all the field of a mamdani object

### Synopsis

`check(m)`

### Description

`check(m)` check the consistency of a mamdani object.

### Remarks

This function has been provided in order to make the mamdani class more robust since it checks a mamdani object for consistency of the data defined in its fields. If a inconsistency is found, an error message is displayed.

# denormalize

### Purpose

denormalises the model

### Synopsis

```
m=denomalise(m)
```

### Description

`m=denomalise(m)` denormalzes the model `m`.

### Remarks

This function can be applied to denormalise a model that has been previously normalised with the function `normalise` restoring the original shape of the local representation of the model.

### See also

normalize

# display

**Purpose**

Display an object of class mamdani

**Synopsis**

```
display(m)
```

**Description**

`display(m)` Display the object m of class mamdani

**See also**

# eval

### Purpose

computes the value of the model

### Synopsis

```
out=eval(m,x,j)
```

### Description

`out=eval(m,x,j)` returns the value `out` for the output(s) `j` of model `m` given the input `x`.

### Remarks

This function returns the output of system as computated by the model, given a regressor.

# get

### Purpose

gets the value of the attribute of an object

### Synopsis

```
d = get(m,label)
```

### Description

`d = get(m,label)` gets the value of the attribute `label` associated to the object `m` and return it inside `d`. The following codes are recognised:

| | |
|---|---|
| `name`: | returns the name of the object |
| `n_in`: | returns the number of inputs of the mapping |
| `n_out`: | returns the number of outputs of the mapping |
| `userData`: | returns the 'userData' field |
| `opt`: | returns the 'opt' field |
| `date`: | returns the date of creation of the object |
| `limits`: | returns the limits of the mapping |
| `optimparams`: | returns the indices of the parameters to be optimised |
| `numparams`: | returns the number of parameters to be optimised |
| `n_rules`: | returns the number of rules associated to the taksug model |
| `model_code`: | returns the code of the model associated to the |
| `rls`: | returns the fuzzy sets associated to the taksug model |
| `mfs`: | returns the definition of the rules of the taksug model |
| `params`: | returns every parameters to be optimised in the shape of a vector |

### See also

set

# identify

### Purpose

identifies a fuzzy model from data

### Synopsis

```
m=identify(m,in,out,options)
```

### Description

`m=identify(m,in,out,options)` identify model `m` on the basis of the data inside `in` and `out` using method specified in `options`.

# jacob_inputs

### Purpose

computes the jacobian of the model

### Synopsis

```
jacob_inputs(m,x)
jacob_inputs(m,x,j)
```

### Description

`jacob_inputs(m,x)` computes for each output the jacobian of the model with respect to the x (the input of the linear) at point `x`.

`jacob_inputs(m,x,j)` computes for output J the jacobian of the model with respect to the x (the input of the linear) at point `x`.

### Remarks

The procedure for computing the jacobian depends on the underlying representation of the model. The result `out` is an array of size number of outputs * number of inputs containing the derivatives of the outputs with respect to the inputs.

# jacob_params

### Purpose

computes the jacobian of the model

### Synopsis

```
jacob_params(m,x)
jacob_params(m,x,j)
```

### Description

`jacob_params(m,x)` computes for each output the jacobian of the model with respect to the parameters at input `x`.

`jacob_params(m,x,j)` computes for output(s) `j` the jacobian of the model with respect to the parameters at input `x`.

# membership

### Purpose

Compute the degree of fullfilment of the rules of a mamdani function

### Synopsis

```
out = membership(m,x)
```

### Description

```
out = membership(m,x)
```

### See also

# normalise

### Purpose

normalises the model M

### Synopsis

```
m=normalise(m)
```

### Description

`m=normalise(m)` normalises the model `m`

# plot

### Purpose

performs a plot of the a mamdani model

### Synopsis

```
f=plot(m)
f=plot(m,options)
```

### Description

`f=plot(m)` opens a new figure if needed and plots the model `m`. A complete user interface is set to allow advanced visualisation options to be selected.

`f=plot(m,options)` opens a new figure if needed and plots the model `m`. `options` are passed to the drawing engine.

# rem_rules

### Purpose

remove the specified rules from the mamdani object

### Synopsis

```
m=rem_rules(m)
m=rem_rules(m,n)
```

### Description

`m=rem_rules(m)` remove all rules from model `m`.

`m=rem_rules(m,n)` remove rules number `n` from model `m`.

### See also

# rem_sets

### Purpose

remove the specified sets from the mamdani object

### Synopsis

```
m=rem_sets(m,i)
m=rem_sets(m,i,n)
```

### Description

`m=rem_sets(m,i)` remove all sets from output i of model `m`.

`m=rem_sets(m,i,n)` remove sets number `n` from output i of model `m`.

### See also

# rules_parser

### Purpose

parses rules in order to populate the rules base of a mamdani model

### Synopsis

```
m=rules_parser(m,rules)
```

### Description

```
m=rules_parser(m,rules)
```

### See also

# set

### Purpose

set object properties

### Synopsis

```
m=set(m,label1,value1,label2,value2,...)
```

### Description

`m=set(m,label1,value1,label2,value2,...)` sets the value `value` of the attribute `label` associated to the object `m`. The following codes are recognised:

| | |
|---|---|
| `name`: | sets the name of the object |
| `n_in`: | sets the number of inputs of the model. This sets the saturation level of the inputs to -Inf, Inf. |
| `n_out`: | sets the number of outputs of the model |
| `userData`: | sets the 'userData' field |
| `opt`: | sets the 'opt' field |
| `date`: | sets the date of creation of the object |
| `limits`: | sets the limits of the mapping |
| `optimparams`: | sets the indices of the parameters to be optimised |
| `model_code`: | sets the type of rules associated to the mapping model |
| `rls`: | sets the fuzzy sets associated to the mapping model |
| `rl`: | sets one fuzzy set associated to the mapping model |
| `mfs`: | sets the definition of the rules of the mapping model |
| `mf`: | sets the definition of one rule of the mapping model |
| `ling`: | sets the linguistical representation of the sets (the name of the sets) |
| `varnames`: | sets the name of the variables |
| `mftype`: | sets the type of a set. `value` is 3 elements cell array a b c where a is the number of the input, b is the index of the set and c is the type ('constant', 'trapezoidal', 's-shaped' or 'gaussian') |
| `params`: | sets every parameters to be optimised providing a vector |

### Remarks

General pupose function for setting the values of the fields of a mamdani object. For setting the rules, use the `add_rules`, `rem_rules` and `add_sets` function. This method extends the 'set' method defined in the mapping class.

### See also

get, add_dynamics, add_data, add_rules

# sets_grid

### Purpose

Remove every fuzzy sets and replace them by evenly spaced ones

### Synopsis

```
m = sets_grid(m, d1, d2, d3, d4..., t)
```

### Description

`m = sets_grid(m, d1, d2, d3, d4..., t)` places `d1` sets of type `t` along the dimension `1`, `d2` sets along the dimension `2`, etc... The position of the sets is computed on the basis of the limits attribute. The previously defined rules and sets are removed.

### See also

add_setss

# Chapter 4

# The graphical User Interface

The Famimo toolbox has been equipped with a graphical user interface, allowing a user friendly approach for the development of MIMO control systems. A series of windows and buttons guide the user during the specification of the characteristics of the model, granting the possibility of selecting different approaches for the solution of the control system.

In order to start the user interface the user must type the command `setup` at the Matlab$^{TM}$prompt from the directory where the toolbox has been installed, then the command `fmgui` starts the graphical user interface. Therefore typing

```
>> setup
>> fmgui
```

the window reported in figure 4.1 will appear on the screen.
It is possible to identify two sets of buttons of the left, the action buttons on the uppermost

Figure 4.1: The initial appearance of the Famimo toolbox GUI.

part, and the model buttons on the lower part. The model buttons allow the user to choose among the different models supported by the toolbox. At present the following models are supported:

- Tagaki-Sugeno models

Figure 4.2: The Dynamics data screenshot of the Famimo toolbox GUI.

- lazy models

Once the model has been selected the action buttons guide the user in the process of defining the model, identifying it, designing the controller and verifying its stability. The data is inserted in the appropriate fields displayed in the top frame on the right, while the bottom frame provides an up to date description of the model.

**Model data**
Once the *Data* button as been selected (default condition when the system is started), the Famimo Toolbox GUI appears as in figure 4.1, and it is possible to proceed in the definition of the input and output variables of the model. It is necessary to specify the name of the file where the model data is saved and the name of the input and the output variables; If the data is already present in the memory of Matlab$^{TM}$it is possible to specify their names leaving out the *Data File* field. The *Apply* button updates the model with inputs from the user. Pushing the *Load Config.* button it is possible to load in memory configurations saved in advance.

**Dynamics data**
When the *Dynamics* button has been selected the Famimo Toolbox will appear as in figure 4.2. The user is prompted to insert the matrix of the inputs of the system, the outputs and the delays. If no information is provided the system implements a static mapping. When all the required information has been inserted it is possible to push the *Apply* button for updating the model.

**Identification**
When the *Identification* button has been selected the use interface changes as in figure 4.3. At this point the user is allowed to choose the method for identifying the model (the range of selections depends from the model which has been chosen), inserting the required parameters. The *Apply* button starts the model identification procedure.

**Visualisation**
Once the model has been identified it is possible to appreciate the results of the identification

Figure 4.3: The Identification screenshot of the Famimo toolbox GUI.

Figure 4.4: The Visualization screenshot of the Famimo toolbox GUI.

process by selecting the *Visualisation* button which changes the window as depicted in figure 4.4. The user has the option of validating the results of the identification with a data set different from the one used in the identification phase. Once the parameters have been set it is possible to press the *Apply* button.

A plot of the model appears in the visualisation window (figure 4.5) the controls at the bottom of the the window let the user (from left to right):

- Choose between a mapping representation (the model is plotted into the regressors space) and a dynamical representation (a simulation of the model is plotted through the time).

Figure 4.5: The visualisation window.

- Choose what is being represented (Data Sli(ce) lets the user choose between seeing all the data or only a part of them when the model is represented as a projection.
- Choose the projection plan (if needed)
- Choose between a two and a three dimensional visualisation.
- Specify the axes limits, rotate and zoom the model.

In this way it is possible to select the features of the visualization according to the needs of the user.

**Control**

Not yet implemented.

**Stability**

Not yet implemented.

# Contents