AMoC

The ACON* Model Classes

Michael Sturm, Till Brychcy, Clemens Kirchmair

October 20, 1997

Abstract

The ACON Model Classes (AMoC) are a collection of algorithms implemented in C++ for approximation of functions and modeling of technical processes. The algorithms are accessible from the MATLAB/SIMULINK environment via the MATLAB to C interface or by integrating the C++ classes in programs written by the user. The aim of this document is to introduce the design of AMoC and to show the user how to access the models realized in AMoC.

1. Introduction

The primary aim of this collection of C++ model classes is the integration of neural systems into a control environment. The commonly used MATLAB/SIMULINK program package has been chosen as a basic tool kit. The following topics had to be met designing AMoC:

- Availability of standard neural procedures
- Easy inclusion of existing methods and algorithms
- Integration with MATLAB/SIMULINK
- Compatibility with MATLAB/SIMULINK toolboxes
- Support stand alone applications without MATLAB/SIMULINK as well

* The research project ACON »*Adaptive Control*« is funded by the german BMBF »*Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie*« under grant 01IN510C8

2. Conception

To meet the above constraints, an abstract view of a model has been defined. The abstract interface of a model is implemented as a C++ class.

As MATLAB's programming interface is designed to interconnect with C routines, we had to implement an object handler to access C++ objects. From MATLAB's point of view a model is represented by an unique number, the model handle. The object handler realizes a mapping from model handles to C++ objects, which implement the models.

Figure 1 shows the different ways of interfacing the ACON Model Classes: The library can be accessed either directly by C++ programs or, as described before, via the object handler by MATLAB/SIMULINK.

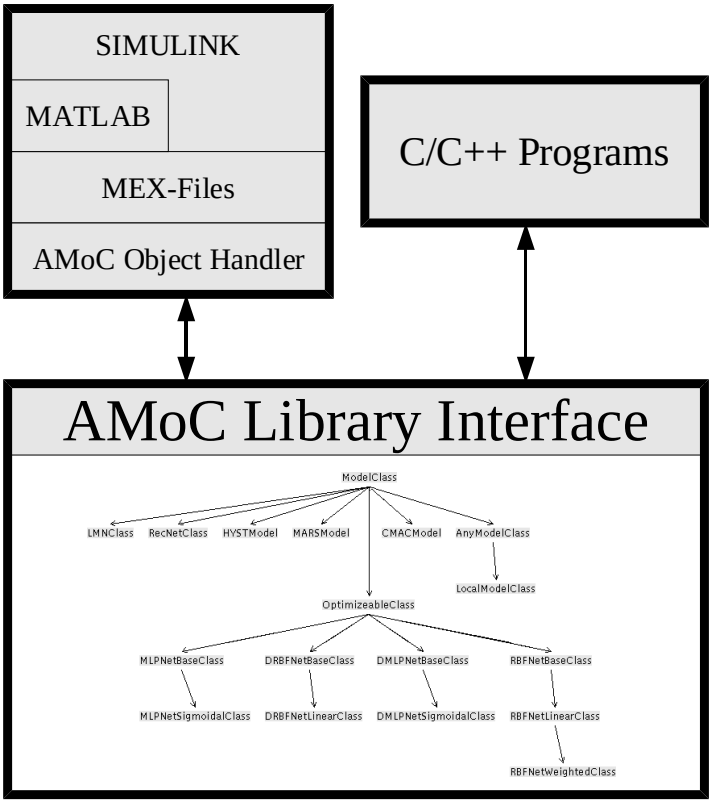

Figure 1: The different AMoC layers.

2.1. Formal View of the Model Hierarchy

Now the formal view of a model will be defined and further on extended to a special class that can be used together with standard optimizers.

2.1.1. Model

Definition

A discrete model is a function **f**, that maps an input vector *u* and a state vector *x* to an output vector *y* and a new state *x'*:

$$\mathbf{f}: (u, x) \in \Re^n \times \Re^s \rightarrow (y, x') \in \Re^m \times \Re^s$$

In a static model the mapping is state-independent. This reduces **f** to:

$$\mathbf{f}: u \in \Re^n \rightarrow y \in \Re^m$$

Properties

Each model must implement the function **f**, to provide a mapping as specified above. The (optional) model state has to be maintained internally by the model and can be accessed by special interface functions.

Often the goal is to train a model by example. If a model is capable of learning from samples, an optional training procedure may be made available.

2.1.2. Optimizeable Model

Definition

An optimizeable model is a model whose mapping function **f** depends on a parameter vector Θ. The previous mapping gets extended to:

$$\mathbf{f}: (u, x, \Theta) \in \Re^n \times \Re^s \times \Re^p \rightarrow (y, x') \in \Re^m \times \Re^s$$

The Jacobian matrix for this mapping regarding the parameter vector Θ of length *p* is defined as:

$$\frac{d\mathbf{f}}{d\Theta} = \begin{pmatrix} \frac{\partial f_1}{\partial \Theta_1} & \cdots & \frac{\partial f_1}{\partial \Theta_p} \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial \Theta_1} & & \frac{\partial f_m}{\partial \Theta_p} \end{pmatrix}$$

The cost function **g** for the optimization process is defined using pairs of training samples (u, y_{Target}) and the prediction error $\mathbf{e}(y, y_{Target})$ as follows:

$$\mathbf{g}(u, \Theta, y_{Target}) := \mathbf{e}(\mathbf{f}(u, x, \Theta), y_{Target}), \text{ where } \mathbf{e}: (y, y_{Target}) \in \Re^m \times \Re^m \rightarrow \Delta \in \Re$$

For gradient based optimizers as addressed here, one needs the gradient of the cost function **g**:

$$\frac{d\,\mathbf{g}}{d\,\Theta} = \frac{d\,\mathbf{g}}{d\,\mathbf{f}} \cdot \frac{d\,\mathbf{f}}{d\,\Theta} = \frac{d\,\mathbf{e}}{d\,y} \cdot \frac{d\,\mathbf{f}}{d\,\Theta}$$

As can be seen, the gradient of the cost function **g** can be computed by multiplying the Jacobian matrix of the error function **e** with the Jacobian matrix of the mapping function **f**.

Please note that the cost function **g** depends on the state x of the model *implicitly*. Thus using state dependent models with this paradigm needs further considerations. (E.g. keeping the state untouched while multiple evaluations of **f** are done by an optimizer ...)

Properties

As an optimizeable model is derived from the model class, it possesses all properties of the former. In addition a parameter vector Θ is maintained internally by the model and accessed by special interface functions to allow an optimization process.

The optional training procedure mentioned above is implemented as a call to an optimizer here.

The Jacobian matrix of the mapping function is estimated by a forward difference method if it is not provided by the model.

Optimizer and Error Function

By now the following optimizers have been implemented:

- **Gradient descent:** Follow the cost function surface in negative gradient direction, update parameter vector Θ after *each* sample pair (u, y_{Target}) has been processed. Step size is dependent of the actual gradient size. See [15].
- **Batch gradient descent:** Same as gradient descent, but update parameter vector Θ after *all* sample pairs (u, y_{Target}) have been processed.
- **RProp:** Modified batch gradient descent, honors only direction of gradient and adapts step size to the cost function surface. A technical description of the algorithm can be found in [12] whereas [13] presents a comparison with other methods.
- **Levenberg-Marquardt:** For a description of this optimizer see [7] and [11]. We use the implementation from MINPACK [8] here.

Each optimizer needs an error function **e** to calculate the cost function **g**. So far there is only one error function implemented:

- **Mean squared error:** $\mathbf{e}\left(y, y_{Target}\right) \equiv \frac{1}{2}\left|y - y_{Target}\right|^2 \Rightarrow \frac{d\,\mathbf{e}}{d\,y} = y - y_{Target}$

For a more detailed description of the optimizer algorithms see the technical documentation of the AMoC library.

2.2. Implementation

This part describes the realization of the AMoC library interface.

2.2.1. C++ Object Classes

Conforming to the formal view of models, the abstract C++ class `ModelClass` defines an interface description for a model. It contains no data, but several interface methods to be realized in derived classes. The methods are divided into three groups:

1. **Pure virtual:** These functions *must* be implemented in a derived class, as they are essential for a model.
2. **Virtual:** This group of functions *can* be implemented in a derived class, as they are
   - Optional: The model may not necessarily need this function. (E.g. `put`/`get` for stream output/input)
   - Already implemented: The function has been implemented in `ModelClass` based on another function, but can be reimplemented in a derived class for better performance. (e.g. `vRecallSingle` based on `vRecall`)
3. **Normal:** The function is implemented in `ModelClass` based on another function. (e.g. `Save`/`Load` based on `put`/`get`)

Another class in the AMoC context is `OptimizeableClass`. It is derived from `ModelClass` and introduces the prior mentioned parameter vector handling and interface routines for optimization processes. Again there are pure virtual, virtual and normal functions.

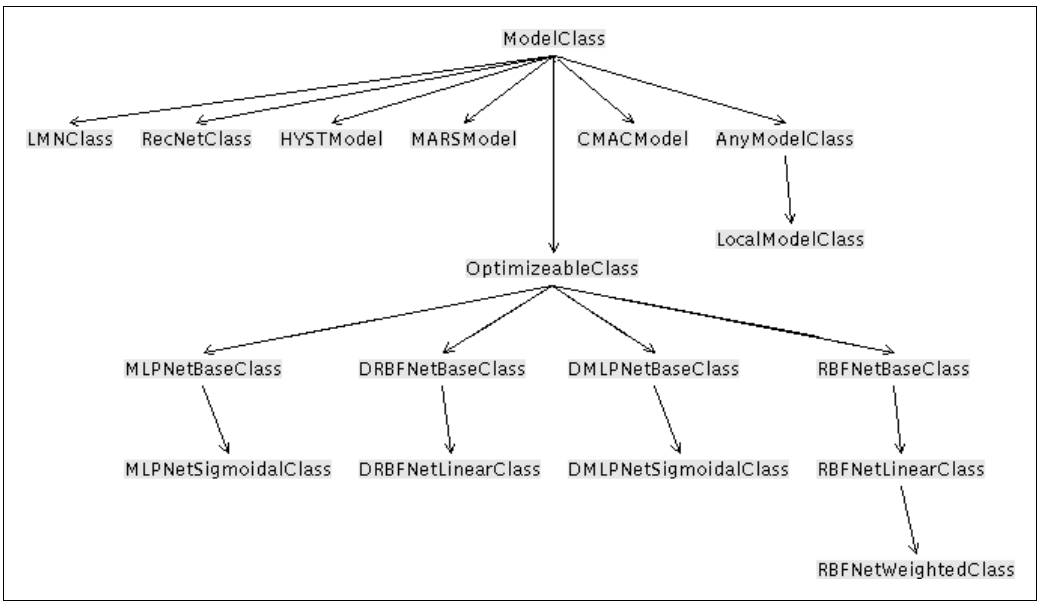

Figure 2: The AMoC class hierarchy.

Figure 2 shows the AMoC class hierarchy as implemented so far. One can see some models derived directly from `ModelClass`. This is due to the discussed state dependency, an inaccessible parameter vector or a specialized training algorithm. All models derived from `OptimizeableClass` use a standard optimizer (Set to a gradient descent optimizer by default).

To provide a simple interface, overloading capabilities of C++ are used. For example the function `Recall` defined in `ModelClass` has three different call signatures:

```
double Recall(const double* const u, const unsigned int i = 0)
void Recall(const double* const u, double* const y)
void Recall(const double* const U, double* const Y, const unsigned int
        nData)
```

The first member function takes an input vector *u*, calculates the `i`'th component of the corresponding output vector *y* and returns it. The second member function takes an input vector *u* and calculates the corresponding output vector *y*. The third member function takes *U*, an array of `nData` input vectors and calculates the corresponding output vectors *Y*. Thus the user just needs to know *one* function name: `Recall`

Unfortunately, some limitations in the C++ inheritance and overloading mechanism force the usage of a second indirection level if one wants to overload virtual functions with the same name only partially: the overloading of functions defined in a base class by a child's member function shadows *all* functions with the same name of the base class. In the above example the three different signatures act only as inlined wrappers to the `protected virtual` functions named `vRecall...(...)`. Those virtual functions then have different names to overcome the overloading problem.

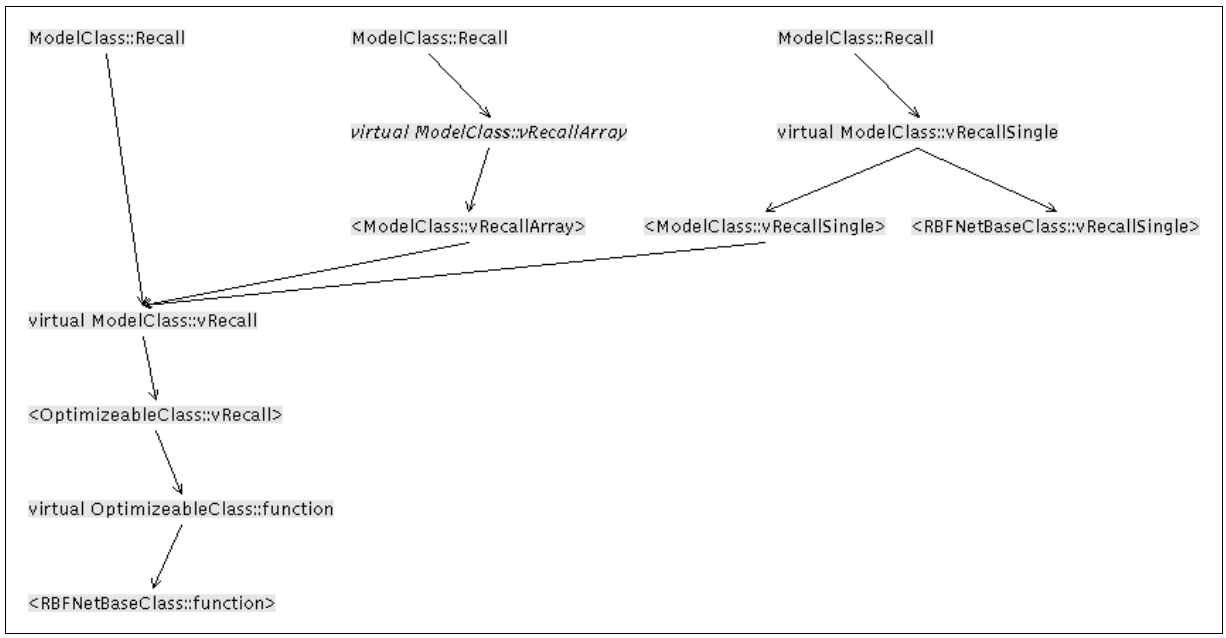

Figure 3: An example for the call of a virtual function (`vRecall`).

Figure 3 shows the resulting call structure for our example. Here a case is illustrated, that has been mentioned above. A virtual function of the base class gets substituted by the derived class for better performance:

In `RBFNetbaseClass` the function `vRecallSingle` has been redefined, to avoid the calculation of all output dimensions, as done by the base implementation. So `RBFNetbaseClass` calls `RBFNetbaseClass::vRecallSingle`, its own implementation. Other classes still call `ModelClass::vRecallSingle`, the realization based on `ModelClass::vRecall`.

2.2.2. MEX Wrapper

As already mentioned in the introduction, a special object handler has been designed to extend MATLAB's programming interface to C++ routines. The handler allows to register, access and delete models (C++ objects). On the MATLAB level, an object is represented by a handle, a unique double number. The handle is generated by a nontrivial algorithm and checked for consistency each time a model operation is done. This assures that accidentally generated handles (e.g. by programming failures on the MATLAB level) are rejected.

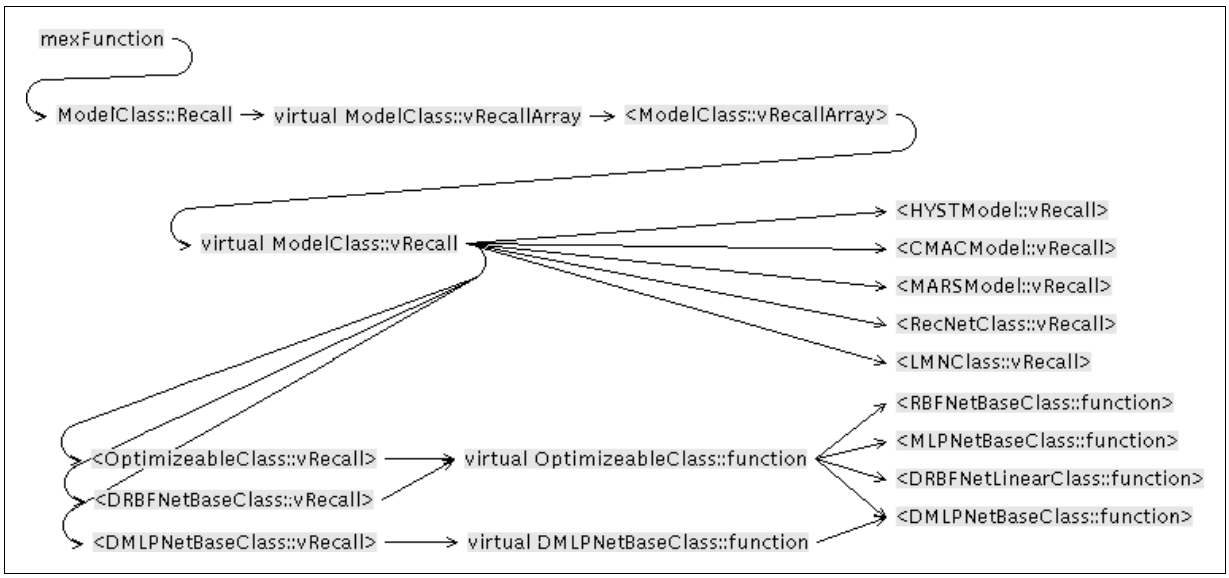

Figure 4: The call of a MEX function: `querymodel` (≡ `vRecallArray`) is mapped to the real computational routines by the virtual call mechanism of C++

The main object type maintained by the handler is `ModelClass`. As some of the implemented models are `OptimizeableClass` objects and MATLAB has some optimizers available, several operations specific for optimizeable models have been made available. As many C++ compilers do not support RTTI (Run Time Type Identification) yet, the downcast from `ModelClass` to `OptimizeableClass` may go wrong if one uses `OptimizeableClass`-specific commands with a handle representing only a `ModelClass` object.

2.2.3. Mapping C++ to MATLAB and vice versa

This section will give a short overview about the different ways of handling objects in MATLAB and C++. Primary the object handling in C++ is done by object pointers. The MATLAB object handler now maps ID's to object pointers for C++ usage. Registering a new object with the handler returns an unique ID containing a coded index into an array of object pointers.

The following table illustrates the different views of objects:

	C++		***MATLAB***	
	Program code	**Result**	**MEX command**	**Result**
Construction	**new** *Type*(...)	*Pointer* to object	*Type*(...)	*ID* of object
Destruction	**delete** *Pointer*	none	**destroymodel**(*ID*)	none
Call method	*Pointer->Method*(...)	any	*Method*(*ID*, ...)	any

Example:

Generate an RBF network with one input, ten hidden and one output unit. The initial value for σ is set to 2.0. Initialize and train the net with 100 samples of the **sin** function from the interval $[-\pi, \pi]$:

```
#include <math.h>
#include "rbfwght.H"

// number of samples
const int SAMPLES = 100;

// the sample array
double* x = new double[SAMPLES];
double* y = new double[SAMPLES];

// the rbf network
RBFNetWeightedClass net(1,10,1,2);

// set the samples
for(int i=0; i<SAMPLES; i++) {
    x[i]=-M_PI+i*M_PI*2/(SAMPLES-1);
    y[i]=sin(x[i]);
};

// initialize the net
net.TrainInit(x, y, SAMPLES);
```

C++ version:

```
// train the net
net.Train(x, y, SAMPLES);
```

MATLAB version:

```
>> SAMPLES=100;
>> x=linspace(-pi,pi,SAMPLES);
>> y=sin(x);
>> net=rbflin(1,10,1,2);
>> initmodel(net,x,y);
>> trainmodel(net,x,y,[]);
```

3. Available Models

This section describes the models and methods available in the AMoC library from a MATLAB point of view. The commands are grouped by function as explained below.

In the following description there are often data samples mentioned. These are stored in MATLAB matrices in the following way: Each data sample vector is represented by a matrix **column**. Thus given n_{Data} sample vectors of dimension n_{Input}, they are represented by a $n_{Input} \times n_{Data}$ Matrix. The following convention is valid for the next subsections:

Upper case letters: *multiple* input data sample vectors stored in a matrix as mentioned above.

Lower case letters: *single* vectors, may be row or column vectors.

3.1. Generator Commands

These commands generate the different models available and return a model ID for each generated object.

3.1.1. Model Generators

CMAC

CMAC Models are an implementation of the extended CMAC. They are described in detail in [4].

```
id = cmac(U, Y, learnrate, type(0=binary,1=tde,2=atde), cellsize,
        rho(=overlap));
```

Hysteresis

There is an implementation of the static linear and scalar Preisach Model of hysteresis accessible from matlab via the `hyst` generators. The computation of the hysteresis values is based on a lookup table which can be given explicitly or initialized with random values. A detailed discussion of the Preisach model of hysteresis can be found in [6].

1. Build a hysteresis model with lookup table of length `grid*grid` that is initialized using random values. This feature makes not much sense until there will be a possibility to adapt the lookup table to given training samples (on our ToDo list). The parameter α_0 of the Preisach Model is set to `a0`.

   ```
   id = hyst(a0, grid);
   ```

2. Build a hysteresis model with given `lookuptable`. The lookup table must be a (N×N)-Matrix. The parameter α_0 of the Preisach Model is set to `a0`.

   ```
   id = hyst(a0, lookuptable);
   ```

MARS

Mars Models are an implementation of J. Friedman's MARS (Multivariate Adaptive Regression Splines [5]) method. Currently Friedman's original FORTRAN implementation is used. `U` contains the inputs, `Y` the outputs; `nk` is the maximum number of basis functions and `mi` is the maximum interaction between basis functions. Note that this kind of model cannot be trained later (you have to specify ALL the data in `U` and `Y` at creation time).

```
id = mars(U, Y, nk, mi);
```

Recurrent Network

A very general class of recurrent networks [3] is supplied by the "recurrent"-generator. You can create a network by specifying a desired `ConnectionMatrix` and the indices of input and output neurons (`InputNeurons`, `OutputNeurons`) as well as `Speed`, the number of internal ticks per global time step. In this case, all neurons will have a sigmoidal activation function. Usually, however, you might want to create a network using the C++ or Tcl-Interfaces (as described in the technical documentation of the class `RecNetClass`) or with the graphical network creator and save it as a file. You can use this network in MATLAB by using the second generator function.

```
id = recurrent(ConnectionMatrix, InputNeurons, OutputNeurons, Speed);
id = recurrent('filename');
```

3.1.2. Optimizeable Model Generators

Radial Basis Function Network

The following generators returns an ID for an RBF network. A formal description of generalized RBF networks can be found in [9] and [10]. There are two different kinds of RBF nets: one with a linear output layer and another with a normalized output layer (`rbflin`, `rbfwght`). There are four ways of calling the generators `rbf*`:

1. Generate an RBF net with `nInput` input neurons, `nHidden` hidden neurons and `nOutput` output neurons. Set the initial sigma to `sigma`. Preset the hidden positions and the connection matrix with zero:

```
id = rbflin(nInput, nHidden, nOutput, sigma);
id = rbfwght(nInput, nHidden, nOutput, sigma);
```

2. Generate an RBF net by reading it from a file:

```
id = rbflin('filename');
id = rbfwght('filename');
```

3. Generate an RBF net with `nHidden` hidden neurons, determine number of input and output neurons by inspecting `U` and `Y`. Set the initial sigma to `sigma`. Preset the hidden positions by clustering `U` and the connection matrix with a pseudoinverse:

```
id = rbflin(U, nHidden, Y, sigma);
id = rbfwght(U, nHidden, Y, sigma);
```

4. Same as 3, but in addition train the RBF net `steps` times with the given data. The parameter `learnparams` contains options for the used optimizer as described with `trainmodel`:

```
id = rbflin(U, nHidden, Y, sigma, steps, learnparams);
id = rbfwght(U, nHidden, Y, sigma, steps, learnparams);
```

Multi Layer Perceptron

The following generator returns an ID for a multi layer perceptron [14] with sigmoidal activation function in the hidden layers. The input and output layer have linear activation functions. There are two ways of calling the generators `mlp`:

1. Generate a multi layer perceptron by reading it from a file:

```
id = mlp('filename');
```

2. Generate a multi layer perceptron with a given layer structure:

```
id = mlp([nInput nHidden ... nOutput]);
```

Dynamic Radial Basis Function Network

With the `drbf` generator a dynamic radial basis function network can be built. A DRBF network consists of a linear input and output layer and a dynamic RBF layer as described in [2]. The dynamic is realized using a linear filter of order 2 with the filter polynomials A and B. There are three ways to build a DRBF net:

1. By calling the first generator a DRBF net with `nInput` input neurons, `nHidden` RBF neurons and `nOutput` output neurons can be built. There is one global σ for all RBF neurons which is set to `sigma_init`.

```
id = drbf([nInput nHidden nOutput], sigma_init);
```

2. Same as first generator call except `sigma_min` which specifies a lower bound for σ. σ is guaranteed to be greater or equal to `sigma_min`. `sigma_min` is set to `0.1` by default.

```
id = drbf([nInput nHidden nOutput], sigma_init, sigma_min);
```

3. Using the third generator the order of the linear filter can be changed. By default `degree` is set to `3` which means the linear filter is of order `2`.

```
id = drbf([nInput nHidden nOutput], sigma_init, sigma_min, degree);
```

Dynamic Multi Layer Perceptron

The `dmlp` generators make dynamic multilayer perceptrons available. Like in a DRBF net the hidden layers constitute the dynamic part of the network. The neurons in the input and output layer are linear, in the hidden layers a sigmoidal function is used as activation function. DMLP networks are introduced in [1].

1. The first generator expects a vector of integers as argument. The `i-th` component of this vector specifies the number of neurons in layer `i` beginning with the input layer. The vector must at least be of length `3`.

```
id = dmlp([nInput nHidden ... nOutput]);
```

2. By supplying the optional argument `degree` the order of the neurons linear filters can be changed. `degree` is set to `3` by default.

```
id = dmlp([nInput nHidden ... nOutput], degree);
```

3.2. Model Methods

The next commands build the common interface of MATLAB to all the models mentioned above.

3.2.1. Methods for all Models

These commands can be applied to all ID's returned by a **model** generator.

Querying Model Properties

Retrieve the input output dimensions of the model corresponding to `id`. The input dimension is returned in `sizes[0]`, the output dimension in `sizes[1]`.

```
sizes = infomodel(id);
```

Retrieve the model state of the model with the handle `id`. The current state is returned in `state`.

```
state = getmodelstate(id);
```

Set the state of the model with the handle `id`. If `state` is not given, the model's default state is set.

```
setmodelstate(id, state);
setmodelstate(id);
```

Saving and Loading Models

Save the model with the handle `id` in the file `filename`. An optional `iomode` argument may be given, where the value 0 means `ASCII` and 1 means `binary` save mode. The default mode is `ASCII`.

```
savemodel(id, filename [, iomode (0=ascii, 1=binary)]);
```

Load a model from the file `filename`. There are two variants: if an `id` is given, the load method of the corresponding model is called. If no `id` is given, the model type will be determined by the file content. A new model of this type will be constructed. The `id` is returned to the caller after reading the file content.

```
loadmodel(id, 'filename');
id=loadmodel('filename');
```

Initializing and Training Models

Initialize a model with the handle `id` with sample data pairs (`U`, `Y`).

```
initmodel(id, U, Y);
```

Train a model with the handle `id` with sample data pairs (`U`, `Y`). Additional parameters for training may be given in the vector `learnparams`. If no parameters shall be given, an empty vector has to be provided.

```
trainmodel(id, U, Y, learnparams);
```

Prediction with a Model

Query a model with the handle `id` at the input data points `U`. The result **f**(*u*, *x*) for each input vector *u* is returned as the matrix `Y`. Please note, that the internal state *x* is updated while querying multiple input vectors, with each single query!

```
Y = querymodel(id, U);
```

3.2.2. Methods for Optimizeable Models

These commands can be applied to all ID's returned by an **optimizeable model** generator. They are designed to be used as part of an optimization process with the optimizers provided by MATLAB, and therefore take an externally provided parameter set `par` ($\equiv \Theta$).

Computing the Jacobian

The next command calculates the Jacobian matrix $\frac{d\,\mathbf{f}}{d\,\Theta}(u)$ of the model with the handle `id` at the input data point `u` for the parameter set `par`. The Jacobian matrix is returned in `j`. If the additional return argument `y` is presented, the function value $\mathbf{f}(u, x, \Theta)$ is returned as well.

```
j = jac(id, par, u);
[j, y] = jac(id, par, u);
```

Accessing the Parameter Set

Access the current parameter set `par` for the optimizeable model with the handle `id`.

```
par = getpars(id);
par = setpars(id);
```

Model Prediction as Function of the Parameter Set

This function is almost identically to `querymodel(id, u)`, but takes an externally specified parameter set. Given the parameter set `par` the optimizeable model with the handle `id` calculates $\mathbf{f}(u, x, \Theta)$ at the input data point `u`. The result vector is returned in `y`.

```
y = func(id, par, u);
```

3.3. Model Management

These commands grant access to the handle management. Most of this is hidden for the user.

Get all Model IDs

Return all existing model handles in `id`.

```
id = getmodels();
```

Delete Models

Remove the models with the handles `id0`, `id1`, … . This calls the destructor of the associated models and frees any allocated memory used by this specific instantiation.

```
destroymodel([id0 id1 ... ]);
```

4. The SIMULINK Blocks

To simplify the access to the AMoC library we provide two SIMULINK blocks for training and recall of models: The *TrainModelBlock* and the *QueryModelBlock*.

4.1. The TrainModelBlock

This block has two input and one output lines. The lower input line has to be connected to the target vector *y*, while the upper input line must provide the input vector *u*. The output line gives the current model prediction for *y* and therefore has the same dimension as the lower input line. If the block is opened, there are three parameter sections: The first one takes a model ID, which must correspond to the dimension of the input and output lines. The second one can be used to specify a training parameter vector. The third parameter sets the sample time rate.

4.2. The QueryModelBlock

Opposite to the *TrainModelBlock*, the *QueryModelBlock* possesses just one input and one output line. The input line must be connected to the input vector *u*. The output line then provides the current prediction for *y*. The block takes two arguments, the model ID and the sampling time rate. The input and output line dimensions must correspond to the ones of the model addressed by the given handle.

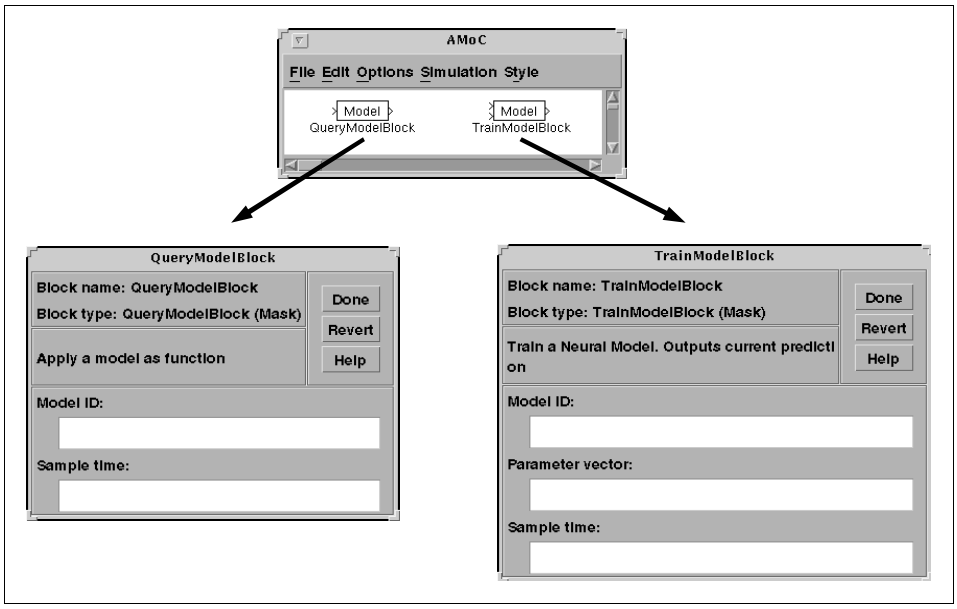

Figure 5: The AMoC SIMULINK blocks.

Example:

Figure 6 shows an example for the usage of the above introduced SIMULINK blocks. A signal **u**(*t*) is fed into a system with the differential equation:

$$\dot{\mathbf{y}}(t) = \mathbf{u}(t) - \mathbf{n}\left(\mathbf{y}(t)\right)$$

The function **n**(*x*) is a nonlinear mapping that has to be learned. We used the AMoC hysteresis model as example data generator. For learning **n**(*t*), we used another AMoC model, the ellipsoidal local models[2] described in [16]. As can be seen, all the preprocessing is done by SIMULINK. The input to the hysteresis gets delayed twice and is fed into a multiplexer to form the input vector for the *TrainModelBlock*. The target output vector is the output of the hysteresis and directly connected. The progress of learning is visualized, the current prediction error **E**(t) is calculated and plotted.

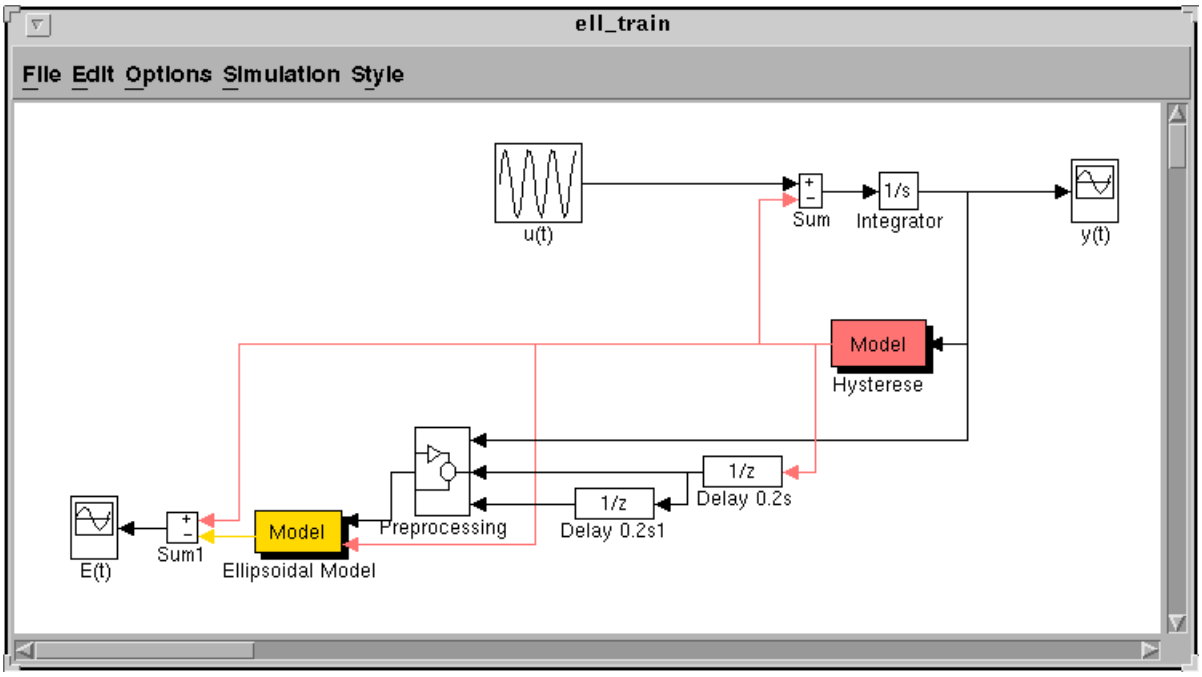

Figure 6: An example for the AMoC SIMULINK block usage.

2 See http://wwwbrauer.informatik.tu-muenchen.de/~sturm/ell/ for an interactive presentation of the results.

Table of Contents

References

[1] Ayoubi, M. (1997): Das dynamische Perzeptronmodell zur experimentellen Modellbildung nichtlinearer Prozesse. In *Informatik Forschung und Entwicklung*, 12(1), pp. 14 - 22, Springer Verlag, Berlin, Heidelberg.

[2] Ayoubi, M., and Isermann, R. (1995): Radial Basis Function Networks with Distributed Dynamics for Nonlinear Dynamic System Identification. In *Third European Congress on Intelligent Techniques and Soft Computing*, EUFIT'95.

[3] Brychcy, T. (1997): *Vorstrukturierte Verallgemeinerte Rekurrente Neuronale Netze (FKI-223-97).* Technische Universität München.

[4] Eldracher, Martin (1995): A Fuzzy-Neural-CMAC function approximator. In *Fuzzy-Neuro-Systeme '95. Theorie und Anwendungen. Tagungsband zum 3. Workshop. 15. bis 17. November 1995*. Darmstadt, pp. 207-214. Gesellschaft für Informatik e.V., Fachausschuß 1.2 „Inferenzsysteme" (Edt.). TH Darmstadt, Sekretariat am Institut für Regelungstechnik, Technische Hochschule Darmstadt.

[5] Friedman, J.H. (1991): Multivariate Adaptive Regression Splines. In *The Annals of Statistics*, Vol. 19 (1), pp. 1-141.

[6] Mayergoyz, I.D. (1991): *Mathematical Models of Hysteresis*. Springer Verlag, Berlin, Heidelberg, New York.

[7] Moré, J. J. (1978): The Levenberg-Marquardt Algorithm: Implementation and Theory. In *G.A. Watson*, pp. 105-116. Springer Verlag, Berlin.

[8] Moré, J.J., Garbow, B.S., and Hillstrom, K.E. (1980): *User Guide for MINPACK-1 (Report ANL-80-74).* Argonne National Laboratory.

[9] Poggio, T., and Giorosi, F. (1989): *A Theory of Networks for Approximation and Learning (A.I. Memo No. 1140).* Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

[10] Poggio, T., and Girosi, F. (1990): Networks for Approximation and Learning. In *Proceedings of the IEEE*, Vol. 78, pp. 1481--1497.

[11] Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P. (1994): *Numerical Recipes in C*. Cambridge University Press.

[12] Riedmiller, M. (1994): *Rprop - Description and Implementation Details*. Institut für Logik, Komplexität und Deduktonssysteme, Universität Karlsruhe.

[13] Riedmiller, M., and Braun, H. (1993): A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm. In *Proceedings of the IEEE International Conference on Neural Networks 1993*, ICNN 93.

[14] Rosenblatt, F. (1958): The Perceptron: A Probabilistic Model For Information Storage and Organization in the Brain. In *Psychological Review (65)*, pp. 386-408.

[15] Rumelhart, D.E., and McClelland, J.L. (1986): *Parallel Distributed Processing*. MIT Press.

[16] Sturm, M., and Brychcy, T. (1997): On-Line Prozeßraumkartierung mit ellipsoider Vektorquantisierung zur lokalen Modellbildung. In *Fuzzy-Neuro-Systeme ´97 - Computational Intelligence.* A. Grauel/W.Becker/F.Belli (Edt.).