Walter S. Mischo

Fachgebiet Regelsystemtheorie und Robotik
Technische Hochschule Darmstadt
Landgraf-Georg-Strasse 4
D-6100 Darmstadt

Tel. (49) 6151/16-5442
Fax. (49) 6151/293445

Introduction to AMS and User Manual for Its Software Simulation CAMS

Preliminary Version
Last Revision: July 15, 1991

Chapter 1

Introduction to AMS)

1.1 The Principal Function of AMS

1.1.1 Association

The principle of the association algorithm is shown in fig. ??.

The input space $\mathbf{S}$, which normally is a multidimensional interval in $\mathbf{R}^n$:

$$\mathbf{S} = [s_{1min}, s_{1max}] \times \ldots \times [s_{nmin}, s_{nmax}] \tag{1.1}$$

(mostly we consider: $s_{1min} = \ldots = s_{nmin} = 0$, $s_{1max} = \ldots = s_{nmax} \equiv s_{max}$) is discretized by a certain $\epsilon > 0$ (without loosing generality: $\epsilon \equiv 1$) and then by ϱ different grid systems each of them with a space of ϱ between the coordinate axes in each dimension. Each grid system is displaced against all others by a constant amount. For instance[1] let us assume a displacement in each dimension which agrees the ordinal number of each grid system, i.e. grid system 0 starts in point $(0, \ldots, 0)$, grid system 1 in $(1, \ldots, 1)$ (or more exact: $(\epsilon, \ldots, \epsilon)$, cf. remark on ϵ) and so on. Obviously we can calculate a new representation $\vec{a}_j$ of a point $\vec{s} \in \mathbf{S}$ in coordinates of the grid system j:

$$a_{ji} = \begin{cases} \lfloor \frac{s_i}{\varrho} \rfloor, & \text{if } (s_i)_\varrho \geq j - 1 \\ \lfloor \frac{s_i}{\varrho} \rfloor - 1, & \text{else} \end{cases} \tag{1.2}$$

Normally we transpose the indices to a_{ij} thus receiving the *association matrix* $\underline{A} = (a_{ij})_{j=1,\ldots,\varrho}^{i=1,\ldots,n}$. The columns of $\underline{A}$ could be understood as code for the "lower left corners" of ϱ ($\varrho \times \ldots \times \varrho$)-hypercubes (so called *association cells*) in $\mathbf{S}$, of which the intersection is a $(1 \times \ldots \times 1)$-hypercube representing $\vec{s}$ at discretization ϵ exactly.

The function defined in this manner has the following properties:

- It is easy to show, that the function is bijective.
- The input point $\vec{s}$ is – from a geometrical point of view – "distributed" over approximately[2] an $(2\varrho - 1 \times \ldots \times 2\varrho - 1)$-hypercube in $\mathbf{S}$ with center $\vec{s}$.

[1] Although this is not the best possibility

[2] The corners of the hypercube are not filled. This effect decreases with increasing ϱ.

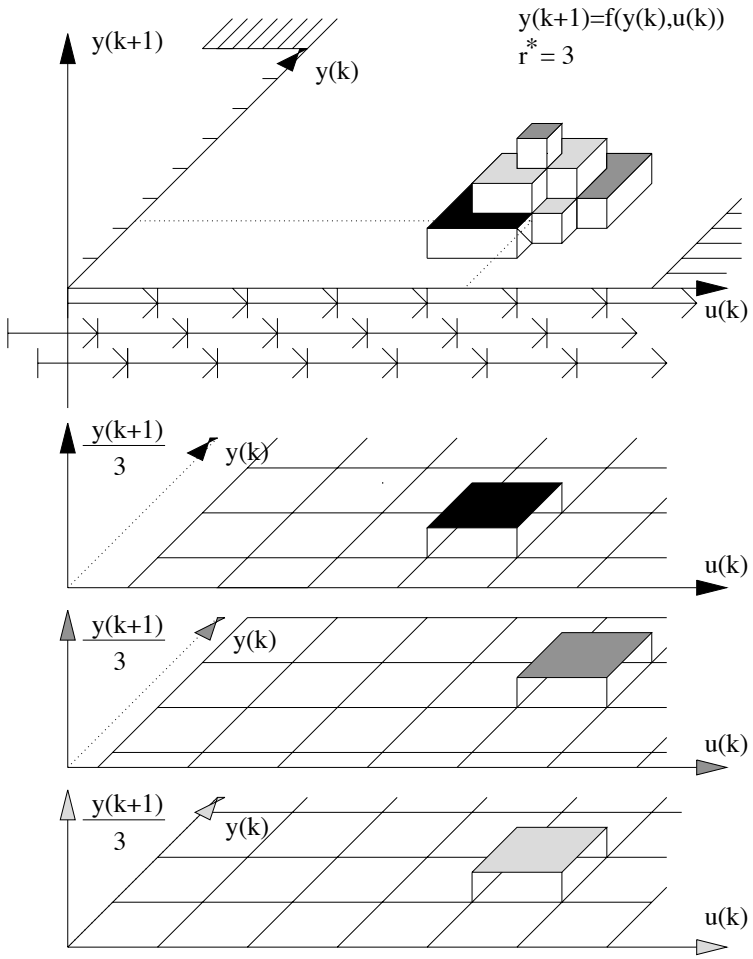

Figure 1.1: Principle of the association in AMS

- If we have two input points $\vec{s}_1$ and $\vec{s}_2$ then the number of equal columns in the related association matrices $\underline{A}_1$ and $\underline{A}_2$ (if the number anyway is > 0) is approximately proportional to the Euclidian distance between $\vec{s}_1$ and $\vec{s}_2$.

Thus this *association scheme* provides the desired behaviour of AMS: automatic interpolation based on the Euclidean distance[3], which we call *local generalization.*

1.1.2 Address Calculation

The code for each of the ϱ *association cells* is a vector of n entries, each of them with a value range of $-1, \ldots, \lfloor s_{max}/\varrho \rfloor$. So memory addressing can simply be done by calculation of the binary equivalent of a number with base $\sigma = \lfloor s_{max}/\varrho \rfloor + 2$:

$$r_j = \sum_{i=1}^{n} (a_{ij} + 1)\sigma^{i-1},\ j = 1, \ldots, \varrho \tag{1.3}$$

For this *direct addressing* method[4] it is therefore necessary to have[5]

$$R = \varrho \cdot \left(\lfloor \frac{s_{max}}{\varrho} \rfloor + 2\right)^n \tag{1.4}$$

memory cells. It is remarquable that this means an enormous saving of memory space, most of all if $\varrho \gg n$, compared with a simple tabulation of the desired functional mapping. If we compare the input value range per component with ϱ by

$$s_{max} = \delta\varrho,\ \delta > 1 \text{ (in general)}$$

we get a reduction factor for memory space of

$$\mu = \varrho^{n-1} \left(\frac{\lfloor \delta \rfloor}{\lfloor \delta \rfloor + 2} \right)^n \tag{1.5}$$

There are mainly two reasons for the application of *hash coding*, an addressing method, which converts the association matrix columns to memory addresses in a non-bijective manner:

1. In spite of the above reduction factor the required amount of memory could be too large and most of all:

2. In many applications is no need for storing all the possible data points. In this case hash coding provides a kind of "focusing" mechanism for the desired data out of a great input space (e.g. the operating points of a process to be modelled, cf. section ??).

As conclusion it can be pointed out, that in most of the applications for AMS a memory space $R' \leq R$ is sufficient.

[3]Another associative memory, the Sparse Distributed Memory by P. Kanerva ([?]) interpolates concerning the Hamming distance.

[4]The denotation *content addressing* is avoided here, because a lot of authors see hash coding also as method for content addressing.

[5]A more detailed formula for the used memory space is possible, but requires a more difficult coding

1.1.3 Training

For the training of AMS a *training set* is assumed, which consists of pairs $(\vec{s}^{(k)}, \hat{p}^{(k)})$, which e.g. can be understood as measured values of the process input[6] and output at time kT, with T as process sampling period.

Each of the ϱ memory locations[7] r_j contains a weight $w_{r_j}^{(k)}$ and an access counter $z_{r_j}^{(k)}$, which initially is set to 0 and incremented each time the weight $w_{r_j}^{(k)}$ is accessed in a training step.

Naturally all weights and counters which are not accessed by an actual training input $\vec{s}^{(k)}$ will not be affected, thus:

$$\left.\begin{array}{rcl} w_l^{(k)} & = & w_l^{(k-1)} \\ z_l^{(k)} & = & z_l^{(k-1)} \end{array}\right\} \forall l \in \{1,\ldots,R'\} \setminus \{r_1,\ldots,r_\varrho\}$$

There are two main methods for adjusting the accessed weights:

Training Exactly on Target: Procedure `adjwgh`

First the average of the activated weights is determined:

$$p^{(k-1)} = \frac{1}{\varrho}\left(\sum_{j=1}^{\varrho} \alpha_{r_j}^{(k-1)} w_{r_j}^{(k-1)} + (1-\alpha_{r_j}^{(k-1)})\hat{p}^{(k)}\right) \tag{1.6}$$

For previously untrained weights (access counter $z_{r_j}^{(k-1)} = 0$) is $\alpha_{r_j}^{(k-1)} = 0$ and the target value is inserted instead of the weight, the other are "validated" by $\alpha_{r_j}^{(k-1)} = 1$.

With the restriction

$$p^{(k)} \stackrel{!}{=} \hat{p}^{(k)} \tag{1.7}$$

each weight has to be corrected by

$$w_{r_j}^{(k)} = w_{r_j}^{(k-1)} + \hat{p}^{(k)} - p^{(k-1)} \tag{1.8}$$

and the associated access counter is incremented (although there is not so much use for it in this algorithm, but in the other one).

Training with Counters; Procedures `admzae, admwza`

This variant is useful for suppressing noise on the target signal $\hat{p}^{(k)}$ and makes more use of the access counters.

Here no average sum is determined. Instead an average of the previous weight value with the actual target value weighted by the access counter is done:

[6] No matter, whether the input really is vectorial or whether a scalar input and process history build up $\vec{s}$.

[7] This is a short-handed notation: of course the r_j are dependent on the actual training input $\vec{s}^{(k)}$: $r_j = r_j(\vec{s}^{(k)})$, regarding association and hash coding.

$$\begin{aligned} w_{r_j}^{(k)} &= \left(z_{r_j}^{(k-1)} w_{r_j}^{(k-1)} + \hat{p}^{(k)}\right) \Big/ \left(z_{r_j}^{(k-1)} + 1\right) \\ z_{r_j}^{(k)} &= z_{r_j}^{(k-1)} + 1 \end{aligned} \tag{1.9}$$

So the target values affecting the same weight are of an influence decreasing with time. Thus this algorithm serves as filter for noise with a mean value of 0. To avoid a total insensivity against slow time variance, the access counter normally is bound to a certain maximum z_{max} thus allowing a correction of $1/(z_{max}+1)$ the value of the total difference $\hat{p}^{(k)} - w_{r_j}^{(k-1)}$. It is obvious that this method takes a longer time for the weights to converge against the desired value as the method above.

1.1.4 Output

The output – or response – of AMS is determined by

$$p = \left(\sum_{j=1}^{\varrho} \alpha_{r_j}^{(k)} w_{r_j}^{(k)}\right) \Big/ \left(\sum_{j=1}^{\varrho} \alpha_{r_j}^{(k)}\right) \tag{1.10}$$

at "each time t with $kT < t < (k+1)T$".[8]

1.2 A New Approach to the Training Algorithm

1.2.1 AMS as Linear Equation System

As denoted in the previous section in in AMS ϱ addresses $r_j^{(k)}$ are calculated from an input $\vec{s}^{(k)}$. If we build up a row vector of dimension R', fill it with 0 on locations, where the weights are not affected by $\vec{s}^{(k)}$, and with 1 at the positions $r_j^{(k)}$ we get:

$$\underline{B}\,\underline{w} = \underline{\hat{p}} + \underline{r} \tag{1.11}$$

of which the number of equations is determined by the number of training points q, say. $\underline{w}$ contains all R' possible weights[9], $\underline{\hat{p}}$ all targets in the training point set. The associated input vectors are coded in the rows of $\underline{B}$. $\underline{r}$ is a remaining error vector to be minimized. Note, that μ from eq. (??) is the degree of overdetermination of system (??), if all possible input points are to be trained.

Now we can formulate a new version of the training algorithm in section ?? as iterative solution of system (??) (with respect to the fact that in the simple case of AMS the factor $1/\underline{b}_k\underline{b}_k^T$ becomes $1/\varrho$):

$$\underline{w}^{(k)} = \underline{w}^{(k-1)} + \frac{1}{\underline{b}_k\underline{b}_k^T}\left(\hat{p}_k - \underline{b}_k \cdot \underline{w}^{(k-1)}\right) \cdot \underline{b}_k^T \tag{1.12}$$

[8]For learning control it is of a great practical interest to access AMS several times between two training steps

[9]In the algorithm of section ?? we store ϱ times the value of each weight for simplicity and numerical stability

where $\underline{b}_k$ is the k'th row of $\underline{B}$. Note that the index k and the discrete time k are "the same" in the meaning that not only the solution method is iterative, but also the equations are "presented" one after another in real applications.

This is the algorithm of *Kaczmarz* for the iterative solution of linear equation systems. It is described in [?] together with a convergence proof. This convergence proof is also done and extended to modified training algorithms in [?].

1.2.2 Modifikation of the Linear Equation System

It is here possible to change the coefficients of the linear equation system to improve the convergence of the solution algorithm ignoring the usual transformation rules, because the task of AMS only requires a function mapping. The matrix $\underline{B}$ of the underlying equation system can – from this point of view – be choosed nearly arbitrary. The association algorithm in section ?? realizes a geometrical relation to the actual input point and leads to coefficients, which are either 1 or 0. The improvement is to choose new coefficients at the locations where previously were 1's, which are in general $\neq 1$.

A Small Example Let us demonstrate the effect of this method with an simple example of an "AMS-like" equation system (something like $\varrho = 3$), but which is quadratic and can be solved.

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix} \underline{w} = \begin{pmatrix} 100 \\ 200 \\ 300 \\ 400 \\ 500 \end{pmatrix}$$

can be solved by the Kaczmarz-algorithm in 961 iterations (solution: $\underline{w} = 1/3 \cdot (0,\ 600,\ -300,\ 300,\ 900)^T$), the (for AMS-purposes) nearly equivalent system:

$$\begin{pmatrix} 0.2 & 2 & 0.2 & 0 & 0 \\ 0 & 0.2 & 2 & 0.2 & 0 \\ 0 & 0 & 0.2 & 2 & 0.2 \\ 0.2 & 0 & 0 & 0.2 & 2 \\ 2 & 0.2 & 0 & 0 & 0.2 \end{pmatrix} \underline{w} = \begin{pmatrix} 100 \\ 200 \\ 300 \\ 400 \\ 500 \end{pmatrix}$$

with the solution: $\underline{w} = 1/2.4 \cdot (556.18,\ 43.82,\ 205.62,\ 300,\ 394.38)^T$ was solved in 142 iterations, which is more than 6 times faster.

Neurobiological Reasoning As a model of the human cerebellum Albus showed ([?]) a simplified scheme like fig. ?? as base for his CMAC memory, which can be seen also as a further improvement of the well-known *Perceptron* ([?]). Like in AMS the activation of the *weight synapses* is here assumed to be binary (0 or 1).

But it is obvious, that other – more continously activating – functions are really used by the nature, because the activation is done by spike rates on the fibers which

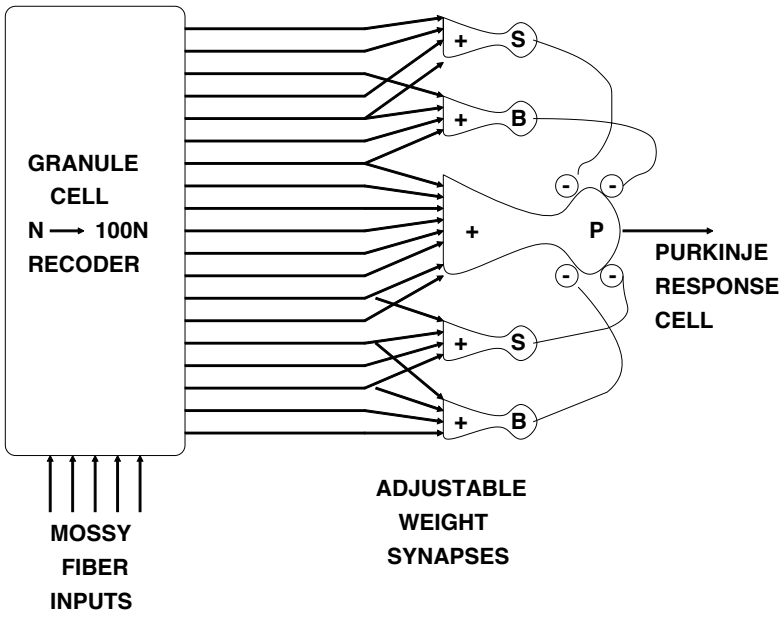

Figure 1.2: Cerebellum model of *Albus*

can be seen as pulse frequency modulated signals on an electrical career. And there is no reason, why the possibility of nearly continous changing pulse rates should not be used in reality. Thus the non-binary weighting in the above system matrix $\underline{B}$ can be reasonable also from this point of view.

1.2.3 The Fire Rate Concept

Distance Between Input and Association Cell If we remember the geometrical explication of the AMS-association in fig. ?? for a two-dimensional case, we can see the column position of the related coefficient in $\underline{B}$ as code for one of the shown squares in the input space, the row position is the time, when the activating input occures. From geometry it is now easy to determine a measure of activation for each association cell (square) via a distance between the center point of each square and the actual input point. As an example an association area for $\varrho = 5$ is shown in fig. ??.

The distance provided in this way can be the Euclidean distance as shown in fig. ??, simplier and also applicable is the so-called "city-block" distance, which easily can be calculated during the association (using already calculated terms) by

$$d_{r_j} = \sum_{i=1}^{n} d_{r_j,i}, \text{ with: } d_{r_j,i} = \begin{cases} |(s_i)_\varrho - (j-1) - \lceil \varrho/2 \rceil|, & \text{if } (s_i)_\varrho \geq j-1 \\ |(s_i)_\varrho - (j-1) + \lceil \varrho/2 \rceil|, & \text{else} \end{cases} \tag{1.13}$$

This distance has for a given ϱ a maximum value of

$$d_{max} = \begin{cases} \frac{n\varrho}{2} & \text{if } \varrho \text{ is even} \\ \frac{n(\varrho-1)}{2} & \text{else} \end{cases} \tag{1.14}$$

Figure 1.3: Defining a distance between input point and association cells

and the sum over all distances in an actual association area is constant and given by

$$\sum_{j=1}^{\varrho} d_{r_j} = \begin{cases} \frac{n\varrho^2}{4} & \text{if } \varrho \text{ is even} \\ \frac{n(\varrho^2-1)}{4} & \text{else} \end{cases} \tag{1.15}$$

Fire Rate of an Assocation Cell The distance from eq. (**??**) can be used to calculate an activity for each association cell, which is qualitatively a monotone decreasing function of the distance. Although it is perhaps possible to determine some criterions for this function from conditioning restrictions on the matrix $\underline{B}$, this would lead to a great amount of computation with respect to the high dimensions of the problem.

In practical tests it has been demonstrated, that the influence of this function on the training process can be derived qualitatively.

We tried the following functions (with descriptive keywords and "inspiration" sources):

(i) $\phi(d_{r_j}) = A \cdot \mathrm{si}(2\pi d_{r_j}/d_{max})$ ("mexican hat function", sampling theorem in the time domain)

(ii) $\phi(d_{r_j}) = A \cdot (1 + \cos(\pi d_{r_j}/d_{max}))$ ("cosine roll-off", filters)

(iii) $\phi(d_{r_j}) = A \cdot \exp(-(a d_{r_j}/d_{max})^2), a = 2.83, 5.1$ (Gauss-distribution)

with some constant A (amplitude), which in practice determines the discretisation of the function values, because they of course are realized by table look-up in the software simulation.

Time Variance of the Fire Rate To obtain a time variant behaviour (a dependency on training point density) these fire rates are weighted with the actual access counter value z_{r_j} resulting in the applied fire rate f_{r_j}:

$$f_{r_j} = \frac{z_{r_j}\phi_{r_j} + \phi_{mean}}{z_{r_j} + 1} \tag{1.16}$$

with ϕ_{mean} normally set to $A/2$.

This method is of importance, if the fire rate function leads to coefficients = 0 for higher distances of the association cells from the input point.

1.2.4 The New Training Algorithm

The training algorithm with fire rates corresponding to that of section ?? can easily be derived from the formulation of the *Kaczmarz* algorithm, eq. (??). But it is instructive to derive it in a manner like in ??.

Average of the activated weights, now with continous activation (the careless notation of the fire rate f_{r_j} is replaced by $f_{r_j}^{(k)}$ to denote the dependency on an actual input $\vec{s}^{(k)}$):

$$p^{(k-1)} = \frac{\sum_{j=1}^{\varrho}(\alpha_{r_j}^{(k-1)} f_{r_j}^{(k-1)} w_{r_j}^{(k-1)} + (1 - \alpha_{r_j}^{(k-1)}) f_{r_j}^{(k-1)} \hat{p}^{(k)}}{\sum_{j=1}^{\varrho} f_{r_j}^{(k-1)}} \tag{1.17}$$

with $\alpha_{r_j}^{(k-1)}$ as in section ??.

For the weight correction there are now two restrictions:

$$p^{(k)} \stackrel{!}{=} \hat{p}^{(k)} \tag{1.18}$$

as in section ?? and additionally:

$$w_{r_j}^{(k)} - w_{r_j}^{(k-1)} = C^{(k)} \cdot f_{r_j}^{(k-1)} \equiv c_{r_j}^{(k)} \tag{1.19}$$

with a constant $C^{(k)}$. From this restrictions we can calculate $C^{(k)}$:

$$C^{(k)} = \frac{\sum_{i=1}^{\varrho} \left(f_{r_j}^{(k-1)}\right)^2}{\sum_{i=1}^{\varrho} f_{r_j}^{(k-1)}} \left(\hat{p}^{(k)} - p^{(k-1)}\right) \tag{1.20}$$

Writing this result in a vectorial formulation we get in fact the *Kaczmarz* algorithm eq. (??) (the square sum in eq. (??) corresponds to the $\underline{b}_k \underline{b}_k^T$ in eq. (??)).

Chapter 2

User Manual of CAMS

2.1 Introduction

CAMS is a software library. There are functions to install, store, restore and access an AMS-type memory. The user interface is contructed via two C-headers named `usertype.h` – containing the data types used by CAMS – and alternatively `install.h`" – the function external declarations for "traditional" C – or `instansi.h` – the function prototypes for ANSI-C.

CAMS is a machine independent implementation with one exception: it will (probably) only run on 32-Bit architectures with sufficient main memory (¿ 1 MByte), which is nearly no restriction today. For implementations on some 16-Bit architectures (e.g. PC-AT) it will probably be sufficient to change some type declarations from `int` to `long`. At first the data types used by CAMS are shown and then the interface descriptions for all CAMS-functions are given with ANSI-C style function headers and explications of the used parameters.

2.2 Data Types of CAMS

```
typedef int     INPUT; /* 32-Bit input */
```

32 Bit integer input components to AMS.

```
typedef float   OUTPUT; /* 32-Bit floating point output */
#define OUTPUT_FORM "%f" /* format element for printing
 * OUTPUT values */

#ifdef SUN_OS
#define OUTPUT_MIN MINFLOAT
#define OUTPUT_MAX MAXFLOAT
#endif

#ifdef HELIOS
#define OUTPUT_MIN FLT_MIN
```

```
#define OUTPUT_MAX FLT_MAX
#endif
```

32 floating point output and target values for AMS. Additionally minimum and maximum values are given here for two operating systems, other values can be added arbitrarely and a format element for formatted printing of the outputs.

```
typedef float   INFACTOR; /* 32-Bit floating point input
 * weighting factor */
```

Floating point factors are used for scaling the AMS input values.

```
typedef unsigned long ADRESS; /* 32-Bit adress */
```

This is the address type of CAMS, which mainly shows the dependence of the right function on the 32 Bit organization.

```
typedef short   FLAG; /* ''boolean'' type */
```

For `TRUE/FALSE` values.

```
typedef short   SMALL; /* small integers */
```

For small integer values.

```
typedef char   *POINTER; /* universal pointer type */
```

Opaque pointer value as return value from CAMS-installation procedures.

```
typedef struct {
    OUTPUT          output; /* output value */
    float           trainind; /* training index */
    float           ctmean; /* counter mean value */
}               ASSOCOUT; /* output from AMS */
```

The output structure of CAMS. The response of CAMS consists of one such record for each output vector component. For training steps a counter mean value is returned `trainind`, which gives an idea about the average number of former training steps applied to the actually trained region. In normal read access the outputs (`output`) are accompaigned by training indices (`trainind`) showing a degree of quality of the output value.