# Processing Sensor Data with the Common Sense Toolkit (CSTK)

## Elaboration according to the work on the CSTK

Martin Berchtold
Tutor: Kristof Van Laerhoven

March 4, 2005

**Abstract**

The Common Sense Toolkit (CSTK) is a middleware framework for developers to process sensor data. It consists of different interconnected C++ classes, configurable through a XML settings language. Consisting of different levels of processing algorithms, the CSTK can provide a bottom-up approach to resource efficient gather information from raw sensor readings. To visualize the information, several graphical visualization tools are provided. Algorithms implemented belong to the fields of clustering, classification, topological representation, memory organization and stochastic analysis.

# Contents

# 1    Introduction

The idea behind the Common Sense Toolkit (CSTK) is to provide a tool to process sensor data in real time on limited resources. Since most sensor arrays are getting more and more independent from wires to provide energy and communication, it becomes necessary to process the gained data nearby the sensor node to decentralize the processing. With this decentralization the processing can become more ad-hoc or even possible when no powerful computer is accessible.
The real time processing becomes also more and more a demand to be solved, because the limitation of memory on embedded devices prevents a collection of an appropriate data set. Analyzing the data in an online algorithm has also the advantage of no pre training phase, since the algorithm adapts during the normal use of the sensors.

For these reasons the CSTK architecture is held simple and the algorithms are implemented in the most possible efficient way. Implemented in C++ conform the C99 standard, the toolkit runs also on other platforms such as a Pocket-PC device without porting the code.

Each module of the toolkit can be interconnected in an arbitrary way, although the general structure is designed for a special processing structure. Each module is implemented in a class structure, which enables the user to reuse the classes in her own code.

## 1.1    The Structure of the CSTK

The CSTK structure's configuration is done with a XML settings language, which determines the input, the output and the settings of each algorithm and plotting module. To parse the XML file a special parser (XML-FILE-PARSER fig.1) module, consisting of different parser classes, which are specified for each algorithm, plotting and input class, is implemented in the toolkit. The parser passes the gained information on how to handle the processing to the actual processing modules.

The first step in processing is the collection of sensor data (SENSOR-DATA-PARSER fig.1). There are several ways of receiving the data, the most common is through a serial connection. With a internet connection several kinds of packages can be obtained, in the case of the CSTK the UDP protocol is chosen. For off-line processing the toolkit provides parsing of log-files to gain the data. The last method to obtain data is through simulation, which is mostly used to test algorithms on implementation and runtime performance.

This report starts with the data structure base, the vector classes (DATA CONTAINER with BASIC OPERATORS and FUNCTIONS fig.1). Since every algorithm can be standardized to retrieve, intermediate save and return data in a vector structure the CSTK bases on this container format. For different purposes, different vector structures have been implemented. The duality between values and operators that is used in the standard one dimensional structure of C++, is obtained in the higher dimensional structures of vectors and matrices. The operators and functions are implemented in the vector and matrix classes themselves, since the attribute belongs to the unit.

The main focus of this report is on the different algorithms (ALGORITHMS fig.1) used to process the sensor data. The algorithms implemented are described in manner of function, implementation architecture and test performance. Their development process is described to give the reader an inner view of the thoughts resulting in the present implementation. Many ideas that occurred during the development process were added and tested in usefulness. Some functions are not really useful for most purposes, but remain in the toolkit for reasons of completeness.

## 1.2    Development Process

During the implementation of the different classes a lot changes were made in the structure of the CSTK. The architecture was changed by the initiator Kristof Van Laerhoven in a mutual development process. The latest architecture aims on respective abstract base classes for every module class to provide a general interface, which makes it easier to handle then each module itself.
The main files for the algorithms have been centralized in the parsing classes to simplify the runtime architecture.
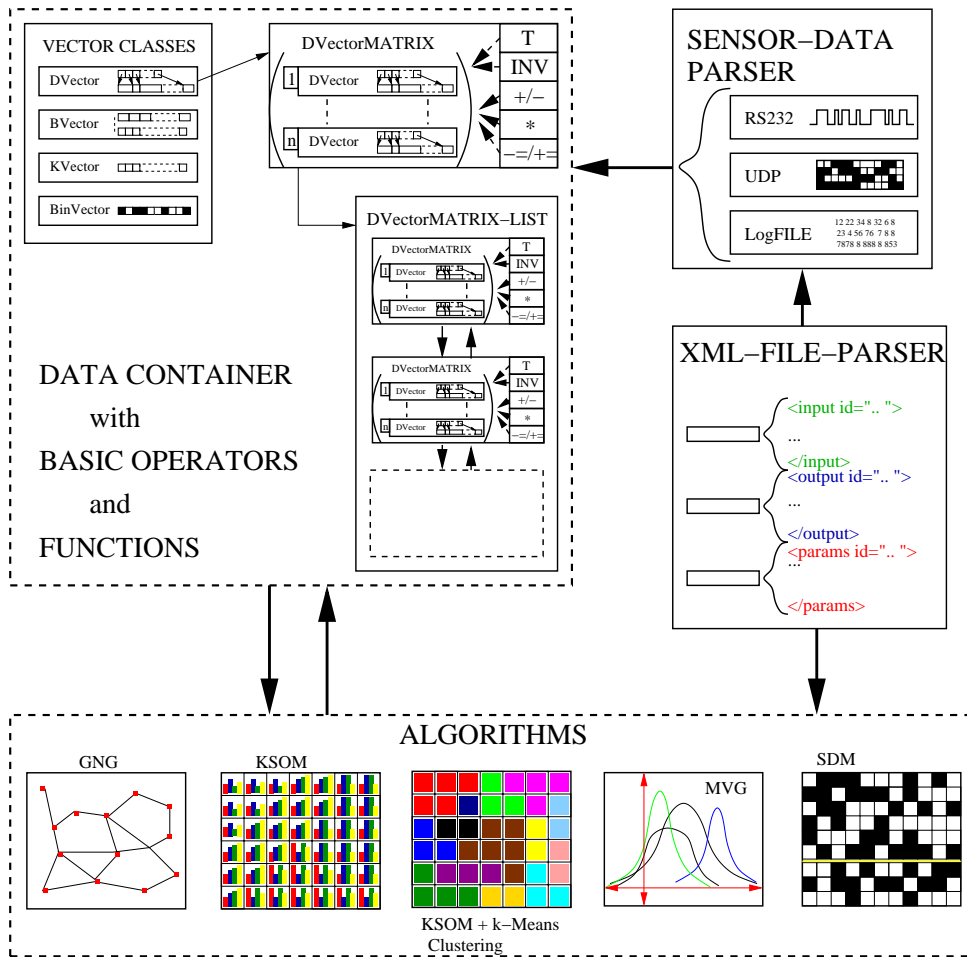
Figure 1: Different Modules of CSTK and the connections between them

The structure of the current CSTK is described in this report as far it was implemented by myself. The modules written by Kristof Van Laerhoven are only described as far as they were used. For further information the respective papers, web pages and the source code downloadable from sourceforge can be called in.

# 2 Data Structures

The base for every data processing is a container for input, output and intermediate data. In the case of the CSTK there were vector architectures chosen. For every special need there exists a special vector class in the toolkit, which uses the given memory space in the most efficient way.

## 2.1 BinVector - a vector for binary sensors

The BinVector architecture (Fig.2) is designed for binary sensor input, such as tilt switches, pressure switches and light barriers. Containing only a bit per value the memory usage is much less than a vector of boolean values. Typically a boolean type needs to be stored in one byte to rep-



Figure 2: The binary vector BinVector

resent one bit, while with the BinVector architecture the bits can be stored together in one byte. When the number of bits is not a multiple of eight, the last byte of the memory is only used partly.

## 2.2 BVector - a template vector

The BVector design (Fig.3) is a template class architecture. With each vector initialization the type of its elements can be specified. The BVector basically contains one sensor's output data that is
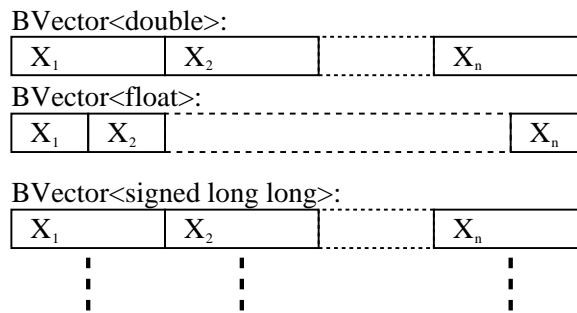


Figure 3: The template vector BVector

viewed over a time interval n. This is suitable if a window of a continuous input stream is required. When the combined output of a variety of different vectors should be analyzed every time they get polled, this architecture is unsuitable. For this purpose the DVector class was implemented: If there is an amount of DVectors of the same type, they can also be transformed into a set of BVectors. Each element representing one sensor (type) is put together in one BVector, which is producing the sensor logging over the time again.

## 2.3 KVector - a vector for eight bit values

KVector is a vector architecture that contains unsigned eight bit values. It is specialized for embedded devices with low resources. The same could be realized with BVector and DVector,
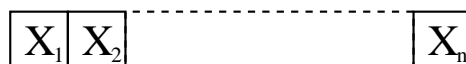


Figure 4: The unsigned integer vector KVector

however BVector is a bit slower due to the template implementation and DVector would need more memory due to the pointer architecture.

## 2.4 DVector - a heterogeneous vector for different sensors

The DVector architecture (Fig.5) is the most flexible and generic one. It is designed for sensor readings of different types of sensors that do not need the same types of memory. Each element's type can be exclusively specified. This becomes possible due to the pointer architecture of DVector: The vector itself is implemented as an array of pointers (Fig.5), which point to the real memory
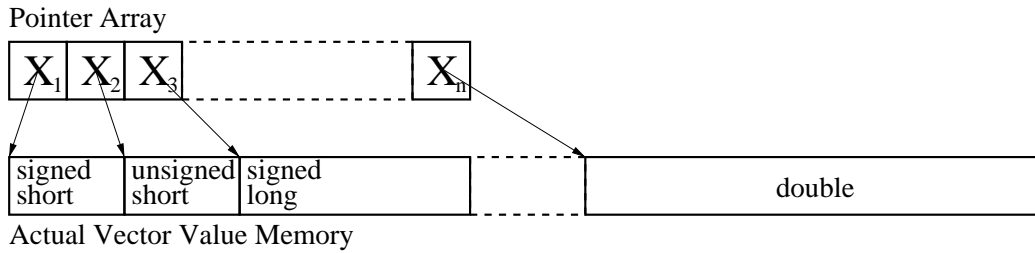


Figure 5: Example of a DVector object: a vector of pointers to variable types

location that contains the data. The data spaces types can can be specified separately for each element. Since this vector can provide a memory entry for a multiple sensor device at a specific point in time, it is the most often used one. Therefore, all algorithms described in the following sections are using the DVector architecture for input and output.

### 2.4.1 DVectorList - encapsulated DVector pointer list

The need for a DVectorList became obvious when the algorithm to calculate the covariance matrix was implemented. The input for the covariance calculation has to consist of vectors that show the development of a sensor array's output over the time. The list is implemented as a double linked list for reasons of completeness, despite the fact that the covariance calculation needs only a single linked one.

### 2.4.2 DVectorMatrix - encapsulated DVector array

The DVectorMatrix is a matrix that consists of row DVectors. Dimensions of the matrix can be specified through the horizontal and the vertical dimension, where the horizontal dimensions the dimension of the DVectors represents and the vertical dimension the dimension of the DVectors array. The need for the matrix implementation first became clear in the 'Multi Variate Gaussian' (MVG - see 3.3.2) model, as it needs various linear algebra operations to calculate the gaussian functions. The particular operators and functions are described in the following 'Mathematical Functions' section.

### 2.4.3 DMatrixList - encapsulated DVectorMatrix pointer list

For the input of the MVG model there was a need for a DVectorMatrix list (DMatrixList) as well. The list element for the MVG input contains the covariance matrix and the mean vector produced during the calculation of the covariance matrix. The MVG scans over the list and finds the minimum according to the input vector. For usage within classification algorithms, the DMatrixList element contains also an integer value with a class number.

# 3 Mathematical Functions

For every calculation there is a need of operators and functions. The one dimensional structure of C++ provides operators for standard mathematical calculations, to which every mathematical calculation can be reduced to. Calculations are often implemented several times in different algorithms if a linear algebra architecture with vectors and matrices is in use . In this manner a definition of special vector and matrix operators and functions reduces code and makes it more readable. Due to the implementation of operators and functions in the DVector class and its derivates, these classes provide a solid base for most calculations done in the algorithms using these classes.

## 3.1 Basic Vector Operators

The first need of special vector operators occurred when the 'Multi Variate Gaussian' (MVG) model was implemented. The model needs several operators which permit the vector calculations without reducing the calculations to each element's operations in the respective function. Since all implemented algorithms use the DVector class, many calculations upon the DVector architecture, that have been implemented before the operator's existence, could have been redone with them.

### 3.1.1 Initial Design

The first implementation of vector operators was done in separate classes, that returned the result as a new vector. The problem with this approach was that the used operators needed to be exclusively included in the current running thread. It also occurred that the operator classes had to be included on different levels. The current running thread needed often algorithm classes which used the operators already. Since such double included files cause linking errors during the compilation, one including reference had always to be removed. The second big problem was that the performance was poorly due to a lot of vector copying. The respective operator class returns a DVector after the calculation. To prevent this vector from deletion, the vector had to be copied into a vector outside the operator class.

### 3.1.2 Vector Operators: DVector

Because of the previous reasons the operator had to be re-implemented. Since the calculations that can be done with a vector depend on the vector itself, the operators must be part of the particular vector. The overloading of the given operators for the vector classes was the nearest choice, so the formulas that use the operations look the same as those with normal one dimensional values. The first problem that became apparent with the overloaded vectors was that the defined methods to read the vectors elements could not be accessed. In the operator definition the input vectors could not be defined as constants, because the compiler could not determine if parameters are manipulated in the accessed method . Without defining the input vectors as constants, the assignments to a result vector after calculations was not possible. After identifying the impossibilities, the design of the operators was moved to a lower level, where the memory entries of the vectors could be directly manipulated. This issue is especially a difficulty with the DVector class, because the DVector itself is only an array of pointers. The actual values are saved in memory locations where the pointers point to, which is the part of the memory that has to be manipulated. Special private 'get' and 'set' methods had to be defined to manipulate the values according to the value's data type.

### 3.1.3 Operator's Return Type

After several changes in older classes and utilization in new implementations of classes the operators had been tested with the algorithm's test files. A problem occurred with the 'Kohonen Self Organizing Map' (KSOM) used by the graphical tool 'topoplot', where after a short while the visualizations got slower and slower, until the whole process was killed. The problem was that the operators returned the address of the vector and not the vector itself, which led to a memory leak as it was not possible to delete the vector from the operator function after returning it. With this mistake the memory got filled up with obsolete vectors. After re-implementing the operator to return the vector itself, the KSOM was running continuously without any further problems.

### 3.1.4 Beyond DVector

The last issue discussed in this section is the usage of the operators in the other vector implementations (BinVector, BVector and KVector).
The first implementation was in the DVector class, because these vectors are used by most algorithms as generic input and output.
It makes not much sense to implement the operators in the BinVector class, since there is no default multiplication or addition defined on binary vectors.
The earlier discussed duality between the DVector and the BVector classes raises the question if there should be a transformation between these two types. If there will be one, there is only the need for operators in the DVector class, because the BVector can be transformed to a DVector and be processed further on. Also it makes no sense to add or multiply two BVectors of different types,

since two different types represent mostly two different sensors or sensor groupings. Two different sensors are mostly not computable with each other.

The implementation of operators in KVector would also be possible, but what happens if the range of the type is overstepped with the calculation? Only the unsigned small integer type is used in KVector, so if the computations need to expand the type, the result must be stored in a DVector or a BVector anyway, and the KVector can be transformed in the first place.

## 3.2 Linear Algebra

The MVG model needed an implementation of a matrix class to store the result of the covariance calculations. Since all calculations that can be done with a matrix are specific for the particular matrix, the functions have been implemented in the DVectorMatrix class. Although this was not the first architecture of the functions, it is the most logical one. First the functions have been implemented as separate classes that use each other if results from other functions were needed. All of them used the DVectorMatrix class to store the output or get the input of the calculations.

### 3.2.1 Matrix Multiplication

The base of the matrix multiplication is the vector multiplication, since all the operations can be reduced to vector multiplications. It is left to the user to assure that the second matrix has to be transposed, because the operator multiplies the row vectors, not the row vector with the column vector. The matrices' dimensions also have to suit to each other (1st matrix: $n \times m$; 2nd matrix: $m \times l$) so the multiplication can be done. If the dimensions do not suit each other, the operator returns an empty matrix.

To make the section complete, it has to be mentioned that there are also operators for vector matrix multiplication, matrix-vector multiplication, matrix-matrix addition and subtraction. These are also based on the DVector operators.

### 3.2.2 Matrix Transposition

The matrix transposition is also a basic method used by many other linear algebra functions. The method makes a column vector out of a row vector according to the math standard (math not.: $(B^T) = (b^T)_{ij} = (b_{ji})$). The result of the transposition is an array of DVectors where each element of one vector has the same type. This fact raises again the question of transforming the transposed matrix into a BVector, since a BVector is constructed to contain elements of the same specified type.

### 3.2.3 Determinant Calculation

The determinant is mostly used by the matrix inversion to calculate the inversion and to determine if it can be calculated. The implementation uses an array - with dimension $n + m$ for $m$ number of columns and $n$ number of rows - to store the multiplicands which are later summed up. In two encapsulated 'for' loops the algorithm goes through the entire matrix. To do this in one pass the structure of the plus and minus multiplicands had to be found. So the element $x_{ij}$ (Fig.6) of the
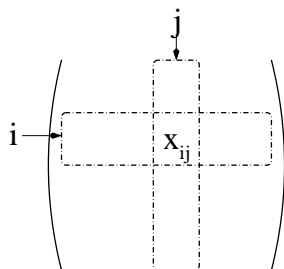


Figure 6: Standard Matrix

matrix is multiplied with the $|i - j|$th element of the array and saved there, when $(i - j) \leq 0$. If the result of $(i - j)$ is bigger than zero, the element is multiplied with the $(m - (i - j))$th element of the array, where $m$ is the number of columns of the array. For the negative multiplicands of

the calculation the $x_{ij}$th element is multiplied with the $(i + j + 1)$th element of the array, when $(i+j+1) \geq n$ for n the number of rows. If not, the element is multiplied with the $(n+(i+j+1))$th element of the array. In the end the array is summed up with the right sign of the elements. The determinant can of course only be calculated when the matrix is a square one, otherwise zero is returned. In the mathematical meaning when the determinant is zero the matrix is singular and cannot be inverted. This is also the fact when the matrix is not a square one.

### 3.2.4  Matrix Inversion

The matrix inversion was the most difficult part of the linear algebra implementations. It was not easy to find an algorithm that does the inversion in an efficient way. The algorithm which uses the calculation of the determinants of sub portions of the matrix (minor) is too calculation intensive for bigger matrices, since there has a determinant for every element in the inverted matrix to be calculated.

The shortcoming of the Gaussian elimination is that computations are performed that are already done. This shortcoming overcomes the Kron reduction as described in [Gra04] (Coleman-Shipley inversion), which reuses results that are already known. The Kron reduction pivots on each diagonal element. In some cases one pivot element can get zero, which ruins the whole algorithm since there are divisions by the pivot. In that case the more calculative intensive inversion through determinants has to be taken, which is done by the implemented inversion method itself.

## 3.3  Stochastic Analysis

The stochastic section is presently based on the multi variate Gaussian model according to Bayes' rule. The covariance matrix calculation is needed as input for the 'Multi Variate Gaussian' (MVG) model.

### 3.3.1  Covariance Matrix Calculation

The covariance matrix determines the multiple variances for a multi dimensional Gaussian function. The variances are calculated over a list of DVectors, where each DVector is one observation of a sensor-array at time t. First the mean values for each dimension of the DVectors has to be calculated by going through the list the first time. The second time the algorithm goes through the list the actual covariance matrix is calculated. The formula that is used is:

$$cov_{ij} = \frac{\sum_{k=1}^{n}(x_i(t_k) - \mu_i) \cdot (x_j(t_k) - \mu_j)}{n}, \quad i, j \in \{1, \cdots, m\} \tag{3.3.1.1}$$

$$CovarianceMatrix = \begin{pmatrix} cov_{11} & cov_{12} & \cdots & cov_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ cov_{m1} & cov_{m2} & \cdots & cov_{mm} \end{pmatrix} \tag{3.3.1.2}$$

$$\mu_i = \sum_{k=0}^{n} x_k \tag{3.3.1.3}$$

In the formula (3.3.1.1) and (3.3.1.3) $n$ determines the number of DVectors in the DVectorList. The equation (3.3.1.2) shows the final covariance matrix that is returned by the method, where $m$ is the dimension of the DVectors.

### 3.3.2  Multi Variate Gaussian Model

The 'Multi Variate Gaussian' model according to the Bayes' rule [Mit97] determines the best hypothesis from some space $H$, given the observed training data $D$. The Bayes' theorem is specified as follows:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)} \tag{3.3.2.1}$$

In equation (3.3.2.1) the posterior probability $P(h|D)$ is derived from the prior probability $P(h)$, together with $P(D)$ and $P(D|h)$. The initial probability that hypothesis $h$ holds is described through $P(h)$. The probability that the training data $D$ is observed is determined through $P(D)$. The next probability function in equation (3.3.2.1) is $P(D|h)$, which describes the probability of

observing data $D$ in a world where $h$ holds. The achievement of the whole equation is $P(h|D)$, which denotes the probability that $h$ holds under the observation of $D$.

The goal of the whole algorithm is now to find the maximum a posteriori (MAP) hypothesis. Given an amount of hypothesis H the MAP hypothesis can be calculated:

$$
\begin{aligned}
h_{MAP} &= \arg\max_{h \in H} P(h|D) \\
&= \arg\max_{h \in H} \frac{P(D|h)P(h)}{P(D)} \\
&= \arg\max_{h \in H} P(D|h)P(h) \qquad (3.3.2.2)
\end{aligned}
$$

Since $P(D)$ is a constant independent from $h$ it is dropped in the last step. Now it is assumed that every hypothesis in $h$ is equally probable a priori. $P(D|h)$ is often called the maximum likelihood hypothesis $h_{ML}$:

$$
h_{ML} = \arg\max_{h \in H} P(D|h) \qquad (3.3.2.3)
$$

With the Maximum Likelihood function can now a density function be inserted, in our case the Gaussian density function. Since the Gaussian function has to cover the multi dimensional input, the density function must be multi dimensional:

$$
P(\mathbf{x}) = \frac{1}{\sqrt{2\pi}\sigma} \cdot e^{-\frac{(\mathbf{X}-\mu)^2}{2\sigma^2}} \qquad (3.3.2.4)
$$

In the last equation the $\mathbf{x}$ is the input vector and $\sigma$ the variance of the Gaussian distribution. To derive a multi variate Gaussian distribution the different variances have to be inserted in the function, which is achieved with the covariance matrix $\Sigma$, yielding

$$
P(\mathbf{x}) = \frac{1}{\sqrt[n]{2\pi}|\Sigma^{-1}|} \cdot e^{-\frac{1}{2}(\mathbf{X}-\mu)^T \Sigma^{-1}(\mathbf{X}-\mu)} \qquad (3.3.2.5)
$$

Exchanging $P(D|h)$ in equation (3.3.2.3) with the multi variate Gaussian distribution (3.3.2.5), depending on the different mean vectors $\mu_i$ and covariance matrices $\Sigma_i$ according to the hypothesizes $h$, yields to equation (3.3.2.6).

$$
\begin{aligned}
h_{ML}(\mathbf{x}) &= \arg\max_{h \in H} \left( \frac{1}{\sqrt[n]{2\pi}|\Sigma_h^{-1}|} \cdot e^{(\mathbf{X}-\mu_h)^T \Sigma_h^{-1}(\mathbf{X}-\mu_h)} \right) \qquad (3.3.2.6) \\
&= \arg\max_{h \in H} \left( \ln(|\Sigma_h^{-1}| \cdot e^{(\mathbf{X}-\mu_h)^T \Sigma_h^{-1}(\mathbf{X}-\mu_h)}) \right) \\
&= \arg\max_{h \in H} \left( -(\mathbf{x}-\mu_h)^T \Sigma_h^{-1}(\mathbf{x}-\mu_h) - |\Sigma_h^{-1}| \right) \\
&= \arg\min_{h \in H} \left( (\mathbf{x}-\mu_h)^T \Sigma_h^{-1}(\mathbf{x}-\mu_h) + |\Sigma_h^{-1}| \right) \qquad (3.3.2.7)
\end{aligned}
$$

With the elimination of the constants and logarithmical adjustments the final equation (3.3.2.7) is achieved.

**In the Implementation of the MVG Model** the input is a list of covariance matrices $\Sigma$ with belonging mean vectors $\mu$ and an input vector. The algorithm scans over the input list and finds the minimum argument according to the equation (3.3.2.7). The result of the scan is a value that is the class information according to the covariance matrix and its mean vector. To assure that the inversion of the covariance matrices is done only once per matrix the inverted matrices have to be saved already in the input matrix list. The inversion is not done in the MVG algorithm itself!

## 3.4 Test Results

The test results reference to the MVG model only, since this uses all the different operators, linear algebra methods and the covariance matrix.

### 3.4.1  Initial Tests with Random or Specified Data

The tests with predefined DVector lists have been pretty good. Several DVectorLists have been filled with manually specified values and subsequently have been fed in the covariance method. The inverted result of each covariance calculation went in a DMatrixList with its mean vectors and a class number. One of the DVectors from one of the lists is fed in the scan to determine if the scan delivers the right class number according to the covariance matrix that was calculated out of the list, which was correctly derived.

### 3.4.2  Tests with Actual Sensor Data

It was tried to classify a real time input of 2D accelerometer sensor data with the MVG model, by pre-classifying several motions like vertical and horizontal shaking, twisting, speeding up and slowing down of the sensor. In the actual test one of the motions classified before was repeated and its classification observed. The outcome of this test was not satisfying, since the to classifying vector was calculated out of the generated vector stream produced by the respective motion by calculating the mean vector. The mean vector can not represent the motion, every information is lost. A peak extraction could work in this case, but was not tested since there is no implementation in the DVector class yet for generating histograms.

# 4  Classification Algorithms

The algorithms, which classify data, work only supervised, since they need a data set that is already pre labelled with a class identifier. According to the distance to the pre labelled data, a decision for the new unclassified input data is made to which class it belongs to.

Two algorithms are implemented. The first one, the K Nearest Neighbors (KNN) classification is suitable for every type of input data. Sparse Distributed Memory (SDM), the second one, is only working on binary data.

## 4.1  K Nearest Neighbors (KNN)

The KNN algorithm is a very simple classification algorithm. It searches for the k nearest preclassified vectors in the search space and decides according to the normalized summed distances of similar class vectors, to which class the input vector belongs to.

### 4.1.1  Description of the Implementation Architecture

The input of the algorithm is a DVector list element (Fig.7), which consists of a DVector, pointers to the next element in the list and a value for the class information. If the value for the class information is already set ($classinf \neq -1$), the DVector list element is inserted in the already
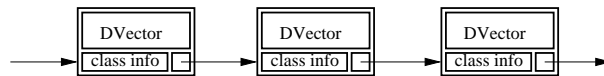


Figure 7: DVector list elements with class information

existing list and a non class value ($-1$) is returned. If the class has not been set ($classinf = -1$) the algorithm performs the distance scan for the $k$ smallest distances between the input DVector and the DVectors in the list, therefore the algorithm goes once through the list of pre classified DVectors. According to the specified distance measurement the maximum distance in the $k$ distances is exchanged against a currently scanned smaller distance. In each loop the biggest distance of the currently $k$ saved vectors is marked. When the end of the list is reached, the $k$ vectors with the smallest distances are found.

To determine the class with the highest occurrence, a normalized sum of same class distances is calculated. The class with the smallest value is the class that is returned by the algorithm.

### 4.1.2 Architecture Development Process

The first implementation of the KNN algorithm was slightly different, since there was a different construction method for the vector list. In that implementation the list was constructed outside the KNN class and the algorithm was operating upon it. The current implementation constructs the list inside the KNN class, by copying the input DVector list element in a list element that belongs to the class itself. After termination of the algorithm, the list is deleted in the destructor method. There was no obvious reason for storing the DVector list for future usage in other algorithms than the KNN one. Also there is a practical aspect for the user, since he has not to bother to link the list elements together and to delete the list after termination of the algorithm himself in his own main function.

The list architecture is different to the DVectorList, because there is no value for a class information belonging to each list element. Mainly the reason to have a different implementation for a DVector list for the KNN algorithm than the general one, was that the implementation of the KNN class was emerged before the implementation of the DVectorList.

## 4.2 Sparse Distributed Memory (SDM)

The Sparse Distributed Memory (SDM) [TAHH99] is a memory architecture for storing high dimensional binary vectors by distributing the data over the memory. Each memory entry consists of an address vector and a data vector. The address vectors are binary vectors that are normally smaller in dimensionality than the data vectors. The data vectors are vectors of floating point types that store the actual data.

Is a new binary input vector stored in the memory, then an address vector has to be associated with it. The associated address vector is in the beginning of the storage process compared with each memory locations address vector in terms of distance. For the distance measurement between
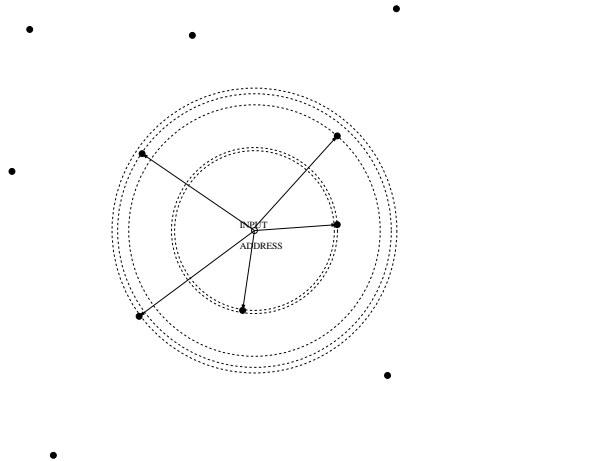


Figure 8: Address distance between input address and addresses of the memory where the input data is saved

the address vectors the Hamming distance is used, which is implemented in the BinVector class. According to the distance of the address vectors the data vectors are distributed over the memory. Each bit of the data vector is represented as a floating point value in the memory. If the respective bit of the input data vector is one, a positive value according to the address distance is saved on each entry of the memory. Is the bit zero, negative values are then distributed over the memory.

A data vector can be retrieved out of the memory with the inverse storage function. Again according to the distances, the values of every data memory location are summed up. If the respective value of the sum is bigger than zero the bit of the stored data vector was a one, was it smaller than zero it was a zero.

Even if the restored data vector is not exactly the same as the stored one, the distance between these two is mostly very small. If it isn't important to retrieve the exact input vector, much more vectors can be saved in the memory than there are entries in it.

The implemented and described version is an approach to the original memory model by Kanerva [Kan94]. His architecture saves the data in a certain radius around the address with the smallest distance to the input address. In this threshold radius the data is saved equally, not weighted. The CSTK includes also an implementation of this early version.

Several different implementations of this principle have been tested with different types of input data. The different implementations are described and the test results are shown in the following subsections.

### 4.2.1 Implementation of different distribution functions for SDM

Inspired by the article of Turk and Goerz [TG94], other distribution functions and distance measurements have been tested with the SDM implementation.

The distribution of the data over the memory was tested with the Gauss function. In the original implementation [TAHH99] the data was distributed linearly, in this function the data is distributed over the memory according to the Gauss distribution. The further away the address is according to the winner address, that has the smallest distance to the input address, the lower is the influence of the input data on the memory data.
The distribution is implemented with absolute and relative distance measurement. With the relative measurement the address with the lowest distance to the input address determines the normalization distance. All addresses with higher distances are getting normalized with the smallest distance by division through it (math not.: $RelativeDistance(a_i) = \frac{AbsoluteDistance(a_i)}{\min_j AbsoluteDistance(a_j)}$).

Another way of distributing the data is by a distance measurement between the address vectors other than the Hamming distance. The only other measurement that has been found was the NAND one, where the bits are NAND wise compared instead of XOR wise. The distribution is also implemented relatively and absolute.

**Test Results derived from Random Data** showed no improvement with other distribution functions compared to the linear one.
Tests with the NAND distance measurement have been especially disappointing. The data vectors which were retrieved from the memory showed nearly no similarity with the actual data that was originally stored in the memory.
The Gauss distribution showed similar results to the linear distribution. Since the calculation of the Gauss function is way more calculation intensive, the performance is even worse.

All three different distribution functions that have been tested with different parameters showed better performance and data retrieval with the absolute distance measurement.

### 4.2.2 The Self Organizing Sparse Distributed Memory

The Self Organizing SDM [HR02] works nearly in the same way as the normal SDM, despite of there being an initialization phase where the memory addresses get trained. In the training phase a subset of the input data is presented to the memory. According to the distance between the input address and the memory addresses local counter vectors are updated. After the whole subset of the input is presented to the memory the counter vectors associated with the addresses are inspected. Is the counter variable associated with the address bit is positive, the address bit is set to one. Is the counter variable negative, the address bit is set to zero. This basically works like the update of the data vectors described in the earlier sections.
Nodes that have been too less influenced by the input-data are getting deleted. This training phase is repeated until a satisfying convergence criterion is fulfilled.

**Test Results derived from Random Data** can't give much information about the efficiency of the algorithm, since the random addresses are getting influenced by random input. Despite of that the algorithm achieved much better results than the normal SDM implementation.

# 5    Clustering Algorithms

Is no class information provided with each data set, clustering algorithms can separate data without further knowledge. Separation and assignment of each data portion to its corresponding data set is done via similarity measurements and assumption of the number of different clusters.

The K-Means clustering algorithm, which is described in the following section, is a clustering algorithm, that initially starts with a specified number of cluster centers and guessed start data for the centers.

## 5.1    K-Means Clustering (KMeans)

The K-Means algorithm clusters input vectors according to their distance to the $k$ cluster centers. In the initialization phase the number ($k$) of centers is set and the centers vectors are set. Each center represents one cluster. With a new input vector the nearest center is searched which determines the cluster the vector belongs to. With the training method the algorithm adapts the centers by making the winning center more similar to the input vector (math not.: $center_i = center_i + \alpha(inputvector - center_i)$).

For the clustering the initialization of the cluster centers is very important. If the centers are to close the clusters that result could be better clustered as one big cluster.If one center has a too big distance to the input vectors, the center will never be activated therefor be useless. The number of clusters is also an important issue. If a input space with more regions than available cluster should be covered this will result in a useless clustering. Is the number of clusters is to high the clustering will result in clusters that should be united.

### 5.1.1    Implementation Architecture and Implementation upon KSOM

The implementation of the KMeans class is a straight forward implementation of the algorithm. There have been no special implementation issues to be solved.

The purpose of the K-Means algorithms implementation is basically based on the usage upon topological maps. Is data sorted, further knowledge of its clusters is useful to stabilize the preprocessing algorithm or to provide sub areas that can be labelled.
In this manner the K-Means algorithm was placed upon a KSOM to cluster the trained topology of the map. For each neuron (explanation in next section) it is decided to which cluster it belongs to, according to the K-Means function. The cluster centers are specified by choosing respective neurons as initial vectors. The outcome of the clustering is visualized via different background colors of each neuron (see Appendix C).

# 6    Topological Maps

Topological maps are used for dimension reduction of high dimensional data. The input data of the maps is mostly not visualizable due to the limitation of our 3D environment and because of that not analyzable through the human mind.

To reduce the dimensionality of the data a mapping onto a lower dimensional topology is performed, through assigning each input vector to its closest map representative. These representatives are moved towards the input data to build up the topology representing the data.

Two different topological maps are going to be introduced in this section, one with a firm grid, the Kohonen Self-Organizing Map (KSOM), and the other with a growing adapting grid, the Growing Neural Gas (GNG) algorithm.

## 6.1    Kohonen Self-Organizing Maps (KSOM)

The KSOM is a one layered neural network that organizes the topology of a grid via training input vectors. The output neurons have relations to their neighbors which forms a grid (Fig.9). Each neuron is with a code book vector associated. When a new vector is fed in the network it
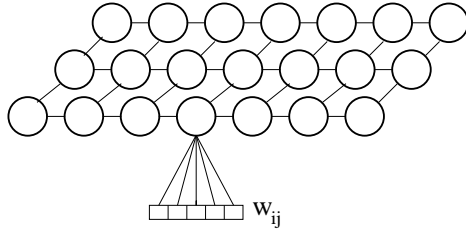
Figure 9: 2-dimensional KSOM grid with code book vector $\omega_{ij}$ of $neuron_{ij}$

is compared with every neuron's code book vector (Fig.9). The neuron with the smallest distance to the input vector wins. Now the winning neuron is made more similar to the input vector by adding a portion of the difference between input and code book vector to the code book vector. The neighbors are also made more similar to the input vector by a decreasing rate proportional to the distance to the winning neuron in the grid. The mathematical description of the code book vector update for a two dimensional grid is:

$$\omega_{ij} = \omega_{ij} + lr(t) \cdot nb(\omega_{winner}, \omega_{ij}) \cdot (v_{input} - \omega_{ij}) \tag{6.1.0.1}$$

The learning rate $(lr(t))$ determines how far the code book vector is changed. Since there can occur an overtraining of the map, the learning rate decreases over time $(lr(t-1) > lr(t) > lr(t+1))$. A neighbor function $(nb(\omega_{winner}, \omega_{ij}))$ determines the distance of the respective neuron to the winning neuron in the grid. There are different functions to determine the neighbor radius:

$$nb_{linear}(\omega_{win}, \omega_{ij}) = \frac{1}{1+ \parallel \omega_{win} - \omega_{ij} \parallel} \tag{6.1.0.2}$$

$$nb_{gauss}(\omega_{win}, \omega_{ij}) = e^{-\frac{\parallel \omega_{win} - \omega_{ij} \parallel}{2\sigma^2}} \tag{6.1.0.3}$$

$$nb_{mex.hat}(\omega_{win}, \omega_{ij}) = \begin{cases} nb_{gauss}(\omega_{win}, \omega_{ij}), & \text{if } \parallel \omega_{win} - \omega_{ij} \parallel \leq \varepsilon \\ -\frac{nb_{gauss}(\omega_{win}, \omega_{ij})}{\varrho}, & \text{if } \varepsilon < \parallel \omega_{win} - \omega_{ij} \parallel \leq (3\varepsilon + 1) \\ 0, & \text{else} \end{cases} \tag{6.1.0.4}$$

The Mexican hat neighbor function is a special function for KSOM, since there is belt around the winning node of nodes which are moved away from the input vector.

### 6.1.1 Description of the Implementation Architecture

In the current implementation there is not much difference to the previously described algorithm. One feature is again the ability of the user to choose different distance measurements (Manhattan, Chebychev, Minkowski and Euclid). Another option is to automatically decrease the learning rate $(lr(t))$ inside the KSOM class via different cascading functions. The used functions for the learning rate are:

$$lr_{linear}(t) = \begin{cases} -(\frac{1}{c} \cdot t) + lr_{const}, & \text{if } (\frac{1}{c} \cdot t) < lr_{const} \\ 0, & \text{else} \end{cases} \tag{6.1.1.1}$$

$$lr_{logarithmic}(t) = \frac{1}{c \cdot \ln t} + lr_{const} \tag{6.1.1.2}$$

$$lr_{exponential}(t) = \frac{1}{c \cdot e^t} + lr_{const} \tag{6.1.1.3}$$

The determination of the time variable $t$ is here the difficult part, since there is no general rule when a new time epoch should be stepped on.

### 6.1.2 Improvements of the KSOM Algorithm

All improvements of the KSOM architecture were implemented in a child class (KSOMfct) of the KSOM class, which leaves the choice to the user if the original algorithm or an improved one should be used.

One improvement to obtain an already trained topology of a map with further input is to use a

16

different neighbor relation. The used relation is using the distance between the code book vectors instead of the distance between the nodes in the grid. This special neighbor relation measurement needs a pre trained map to work on.

### 6.1.3 Test Results

The initial tests have been performed with a four dimensional simulated data input stream and a switch of the neighbor measurement after 2000 input vectors. As in figure 10 can be seen, there is a nicely developed topology of the map after the initial phase of 2000 input vectors. After the long
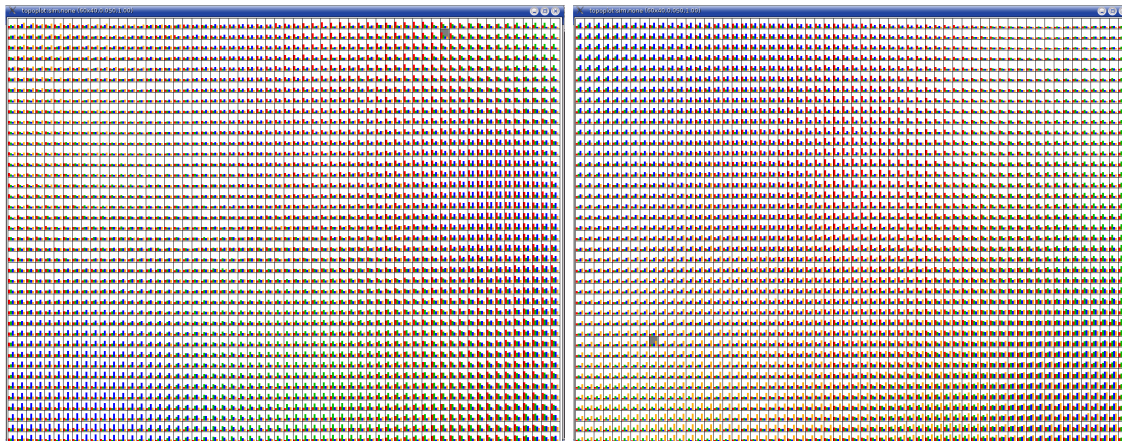


Figure 10: Training of a KSOM with 60x40 nodes: (left) trained with 2000 input vectors and normal grid neighbor measurement, (right) after 10h training with code-book-vector neighbor measurement based on the pre-trained map on the left side

time training with the different neighbor measurement there is still a topology visible in which different clusters can be determined. With the normal neighbor measurement the map would have been overtrained and nearly no different clusters would be recognizable.

## 6.2 Growing Neural Gas (GNG)

The Growing Neural Gas algorithm [Fri95] is one of the newest approaches in the field of topological maps. The topology of the map is not known in advance and is growing over the time. Comparable to the KSOM the GNG algorithm adept's the neurons code book vectors to the input training data, unlike the KSOM it increases the amount of neurons and connections over the time as long as necessary.

In the beginning the algorithm starts with two neurons and a connection (edge) between them. With each new input vector $\xi$ the nearest $\omega_{s_1}$ and the second nearest code book vector $\omega_{s_2}$ are found and their neurons $s_1$ and $s_2$ connected via an edge when there is no existing one. The nearest neuron and its direct topological neighbor neurons - neurons which are reachable over an edge - code book vectors are updated according to equation (6.2.0.1) and (6.2.0.2).

$$\Delta\omega_{s_1} = \epsilon_b(\xi - \omega_{s_1}) \tag{6.2.0.1}$$

$$\Delta\omega_n = \epsilon_n(\xi - \omega_n) \text{ for all direct neighbors } n \text{ of } s_n \tag{6.2.0.2}$$

To get information about where to insert a new neuron an error variable is associated with each neuron, which is updated for every winner neuron $s_1$ with a $\Delta error$ (6.2.0.3).

$$\begin{aligned} error_{s_1}(t+1) &= error_{s_1}(t) + \Delta error \\ &= error_{s_1}(t) + \| \omega_{s_1} - \xi \|^2 \end{aligned} \tag{6.2.0.3}$$

With every new epoch a new node is inserted in between the node with the highest error value and its neighbor with the largest error. The edge that connects this neighbors is deleted and two new connections to the new node are inserted. The new nodes code book vector is initialized with the mean vector ($\omega_{new} = 0.5(\omega_{max.error} + \omega_{neighbor_{max.error}})$) of the code book vectors from the two old nodes.

All edges are associated with an age value, that keeps track of wether the edge is still usable or not. The edges age emanating from a winner neuron are increased. If an edge reaches an age higher than a certain constant it is deleted. If a neuron exists without an emanating edge it is deleted as well.

### 6.2.1 Description of the Implementation Architecture

It was not easy to find an architecture that realizes the highly flexible structure of the GNG network, since the number of nodes and especially their edges is not known in advance. The only way to do this was with lists of pointers. The double linked node pointer list's element (Fig.11) consists of a DVector representing the code book vector, a pointer to the next and the last element in the list and a pointer to the first element of the corresponding edge list. The edges are basically elements consisting of four pointers: two to create the double linked edge list and two to point to the nodes the edge interconnects. To use doubly linked lists was necessary because of deletion reasons, since there have unused edges and nodes to be deleted. List elements that stand somewhere in the list can only be removed in doubly linked lists. Since there is not known which nodes need to be connected there has to keep track of the edges for each node. This means a bit of an overhead, because there is always an incoming and an outgoing edge per interconnection of two nodes. There was not a more efficient way of implementing the algorithm that could be thought of.

### 6.2.2 Test Results and Runtime Notes

The first tests have been performed with a two dimensional simulated input stream, so the plotting needed no dimension reduction.

As the trained GNG network (Fig.12) shows, the simulated input stream is nearly evenly spread over the two dimensional input space. To make the network in the beginning fast growing and in the end stabilize to a static state, the insertion of nodes is linearly decreasing over the time.
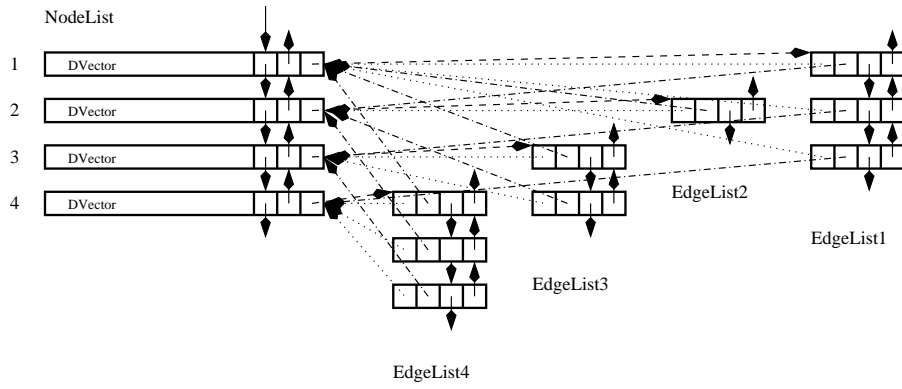
Figure 11: GNG node list and edge lists
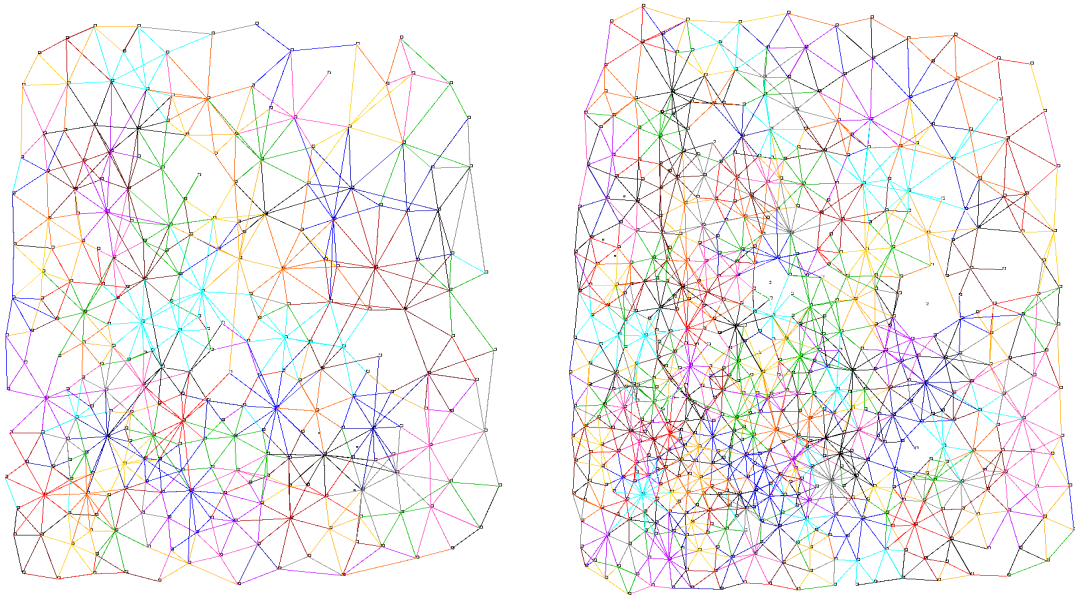


Figure 12: (right) GNG network after short time training with 2-dimensional simulated input stream - 253 nodes; (left) GNG network after long time training with 2-dimensional simulated input stream - 563 nodes

Since edges and nodes are getting deleted more frequently in the starting phase of the algorithm and nodes are getting inserted more rarely, the maximum age must be at least be so high that the number of nodes is stable and increasing over the time. If the maximum allowed age of the edges is too low, the algorithm is not achieving the desired goal of covering the input data's space.

# 7    Conclusion and Summary of the Implementations

A lot has been implemented, vector and matrix operators and functions, stochastic calculations, algorithms to classify and cluster data and different topological maps. With each implementation a development process was passed with many changes in efficiency, flexibility, usability and even changes in whole architectures.

The DVector and its derivates' code has changed the most: errors in designs and implementations of the operators were discovered and changed, algorithms for the manipulating functions have been changed in efficiency and robustness in calculations and memory leaks have been patched.

For many algorithms special functions that improve the outcome have been discovered while working with them. These functions were implemented and tested on usability and documented here.

Interconnections between algorithms have been implemented for the KSOM and the K-Means clustering with visualization. Similar classes inherit same functions to reduce code, which is especially the case for the SDM architecture.

In the end a collaborative work was prepared for the 'International Workshop on Wearable and Implantable Body Sensor Networks' and submitted. The produced paper is attached in appendix C and can be seen as cross test of many previously described algorithms in a real life problem.

# References

[Fri95]    Bernd Fritzke. A growing neural gas network learns topologies. *Advances in Neural Information Processing Systems 7, MIT Press, Cambridge MA*, 1995.

[Gra04]    W. Mack Grady. *Power System Notes*, chapter 4.3 Kron Reduction. WorldWideWeb, 2004.

[HR02]    Emma Hart and Peter Ross. Exploiting the analogy between immunology and sparse distributed memories: a system for clustering non-stationary data. *Napier University, Scotland*, 2002.

[Kan94]    Pentti Kanerva. The spatter code for encoding concepts at many levels. *ICANN'94, Proceedings of the International Conference on Artificial Neural Networks*, 1994.

[Mit97]    Tom M. Mitchell. *Machine Learning*, chapter 6.2 Bayes Theorem. McGraw-Hill Book Co, 1997.

[TAHH99]    David J. Willshaw Tim A. Hely and Gillian M. Hayes. A new approach to kanerva's sparse distributed memory. *IEEE Transactions on Neural Networks, Vol.XX, No.Y*, 1999.

[TG94]    Andreas Turk and Guenter Goertz. Kanerva's sparse distributed memory: An object-orientated implementation on the connection machine. *ICANN'94, Proceedings of the International Conference on Artificial Neural Networks*, 1994.

# A Relevant Header Files

## A.1 DVector

```
/************************************************************************
                        dvector.h  -  v.1.10
    begin               : Aug 17 2002
    copyright           : (C) 2002-2003 by Kristof Van Laerhoven
    email               :
 ************************************************************************/

/************************************************************************
 *                                                                      *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or   *
 *   (at your option) any later version.                                *
 *                                                                      *
 ************************************************************************/

#ifndef DVECTOR_H
#define DVECTOR_H

#include <stdio.h> // NULL, sprintf, ...
#include <math.h>
#include "vect_types.h"  // datacell
#include "datacell.h"

#define abs(a) ((a)>0)?(a):-(a)

/**** DVector Definition ************************************************
 * Use this class for input or data vectors for your algorithms. It     *
 * supports mixed types, and stores a pointer per component.            *
 ************************************************************************/
class DVector {
 public:
   DVector();
   DVector(vei_t nsize);
   DVector(const DVector& vec);
   ~DVector();

   void create(vei_t nsize);

   void set_comp(f_64b value, char type, vei_t comp);
   void set_comp(DataCell value, vei_t comp); // allocate a component

   f_64b get_comp(vei_t comp);         // return a component as a 64 bit float
   char  get_type(vei_t comp);                   // return the type
   vei_t get_dim();                              // return the vector dimension

   oas_t dis_eucl(DVector& invec);          // Euclidean distance
   oas_t dis_manh(DVector& invec);          // Manhattan Distance
   oas_t dis_cheb(DVector& invec);          // Chebychev Distance
   oas_t dis_mink(DVector& invec, vei_t exp); // Minkowski Distance

   char* to_string(void);

   bool operator==(const DVector& vec);

   /*********************************************************************************
    *    The equal operator is just a loop that reads the elements of the source vector  *
    *    (vec) and writes the values in the destination vector (this).              *
    *    The types are the same than the types from the input vector,              *
    *    which made it possible to use the set function that has only one          *
    *    vector element as input.                                                  *
    *********************************************************************************/
   DVector& operator=(const DVector& vec);

   /*********************************************************************************
    *    This operator was more difficult to implement, because there has an actual   *
    *    calculation to be done. The get function returns only an element           *
    *    as a double, which makes it not possible to use this method. If            *
    *    the types of the two input vectors are the same, the type is taken         *
    *    over for the new edition of the first input vector. Are the types          *
    *    different, the get function is used and all the elements of the            *
    *    saving vector are becoming the double type. There is not jet a             *
    *    safety treatment if the vector types are the same, but the calculation     *
    *    causes an overflow of the type.                                            *
    *********************************************************************************/
   DVector& operator+=(const DVector& vec);

   /*********************************************************************************
    *    This operator is the same as the previous operator, except that there is done  *
    *    a minus instead of the plus.                                              *
    *********************************************************************************/
   DVector& operator-=(const DVector& vec);

   /*********************************************************************************
    *    The friend specifier specifies that the operator is not inside the vector class,*
    *    but it belongs to the class, it is a friend of it. Friend Methods and operators  *
    *    can use private declarations of the class. The friend specification is needed,  *
    *    because with it the definition of operators with two input variables is first  *
    *    possible. The implementation is nearly the same as the '+=' operator despite   *
    *    of the destination vector where the results are saved. This vector has to be   *
    *    constructed in the operator itself instead of returning the first input vector  *
    *    with 'this'. It works the same way as the according mathematic operator, each  *
    *    element of the first vector is added to the element of the second vector that  *
    *    stands on the same position.                                              *
    *********************************************************************************/
   friend DVector operator+(const DVector& vec1, const DVector& vec2);

   /*********************************************************************************
```

```
  *     This operator is the same as the previous operator, except that there is done   *
  *     a minus instead of the plus.                                                    *
  *************************************************************************************/
  friend DVector operator-(const DVector& vec1, const DVector& vec2);

  /*************************************************************************************
  *     The value is multiplied with each element of the input vector and the result is *
  *     saved in a newly constructed result vector. Since the type of the value has to  *
  *     be known in advance, the biggest type (double) had to be chosen. With this fact *
  *     the get function could be used within a loop. The operator returns again a      *
  *     vector constructed inside the operator.                                         *
  *************************************************************************************/
  friend oas_t operator*(const DVector& vec1, const DVector& vec2);

  /*************************************************************************************
  *     This is the Vector multiplication operator according to the math standard, that *
  *     multiplies each element and sums up the result of the multiplications. Since the*
  *     type of the multiplications result is not known in advance , the biggest type   *
  *     (double) is chosen. With this abstraction the get function can be               *
  *     used within a simple loop.                                                      *
  *************************************************************************************/
  friend DVector operator*(const oas_t val, const DVector& vec);

 private:
   DVector& set(const DVector& vec, vei_t i);
   friend oas_t get(const DVector& vec, vei_t i);

   void **vct;                            // the pointer array
   char *types;                           // the types for each pointer
   vei_t vctsize;
   char *strout;                          // pointer to outoput buffer
};
#endif
```

## A.2   DVectorMatrix

```
/********************************************************************
                   dvectormatrix.h  -  v.0.1
                   -------------------------
     begin                : Mo Oct 04 2004
     copyright            : (C) 2004 by Martin Berchtold
     email                :
 ********************************************************************/

/********************************************************************
 *                                                                  *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                            *
 *                                                                  *
 ********************************************************************/
#ifndef DVECTORMATRIX_H
#define DVECTORMATRIX_H

#include <stdio.h>
#include "cstk_base/vector/dvector.h"
/*************************************************************************************
 *      List of DVector's with pointer to next and last element in list.            *
 *************************************************************************************/
class DVectorList
{
    public:
        DVectorList() {vector=NULL;next=NULL;}
        DVectorList(vei_t nsize) {vector = new DVector(nsize);}

        ~DVectorList() {if (vector) delete vector;}

        void create(vei_t nsize) {vector = new DVector(nsize);}

        //DVector that contains the data
        DVector *vector;

        //pointer to the next DVectorList element in the list
        DVectorList *next;
        //pointer to the lsat DVectorList element in the list
        DVectorList *last;
};
/*************************************************************************************
 *      DVectorMatrix definition that contains an array of DVector rows             *
 *      Information of the number of rows are saved in vesize and the number of     *
 *      columns in hosize                                                           *
 *************************************************************************************/
class DVectorMatrix
{
    public:
        DVectorMatrix();
        DVectorMatrix(vei_t hsize, vei_t vsize);
        // Copy constructor
        DVectorMatrix(const DVectorMatrix& input);

        ~DVectorMatrix();

        void create(vei_t hsize, vei_t vsize);

        /*************************************************************
         *    Matrix transposition that makes out of the column elements row vectors.  *
         *    Result is a Matrix that has instad of various type DVectors rows of homogenous *
         *    DVectors that contain only one data type each              *
```

```
                ******************************************************************************/
                DVectorMatrix T();

                /******************************************************************************
                *   Matrix inversion according to the Coleman-Shipley inversion (Kron reduction)  *
                *   algorithm. If one of the pivot elemnts gets zero the determinant inversion     *
                *   is done instead.                                                               *
                ******************************************************************************/
                DVectorMatrix INV();

                /******************************************************************************
                *   Straight forward implementation of the math algorithm that multiplies the     *
                *   diagonals of the matrix and adds and subtracts the results                     *
                ******************************************************************************/
                oas_t det();

                /******************************************************************************
                *   This equal operator does the same as the copy constructor, it uses the        *
                *   equal operator of DVector to hand over each DVector of the matrix to           *
                *   DVectors in the new matrix                                                     *
                ******************************************************************************/
                DVectorMatrix& operator=(const DVectorMatrix& b);

                /******************************************************************************
                *   The multiplication between matrices multiplies each row vector from the        *
                *   first matrix with each row vector of the second matrix using the DVector       *
                *   operator multiply. Dimensions have to fit and the second matrix has to be      *
                *   transposed!                                                                    *
                ******************************************************************************/
                friend DVectorMatrix& operator*(const DVectorMatrix& mat1, const DVectorMatrix& mat2_T);

                /******************************************************************************
                *   The multiplication between a matrix and a vector multiplies each row vector    *
                *   from the matrix with the vector using the DVector multiplication operator.      *
                *   The dimenson of the row vectors and the vector have to be the same             *
                ******************************************************************************/
                friend DVector& operator*(const DVector& vec, const DVectorMatrix& mat_T);
                friend DVector& operator*(const DVectorMatrix& mat_T, const DVector& vec);

                /******************************************************************************
                *   The plus and minus operators use also the plus and minus operators fom the     *
                *   DVector class to add and subtract the row vectors.                             *
                ******************************************************************************/
                friend DVectorMatrix& operator+(const DVectorMatrix& mat1, const DVectorMatrix& mat2);
                friend DVectorMatrix& operator-(const DVectorMatrix& mat1, const DVectorMatrix& mat2);

                //DVector that contains the data
                DVector *vector;
                //storage for the dimension information of the matrix
                vei_t hosize, vesize;
        private:
                //matrix inversion according to the determinant inversion
                DVectorMatrix detinv(DVectorMatrix& mat);
                //cofactor calculation needed for the determinant inversion
                oas_t cofactor(DVectorMatrix& mat, vei_t i, vei_t j);
};
/************************************************************************************************
*       List of DVectorMatrices with pointer to next element in list,                             *
*       a DVector element associated with each Matrix and a integer variable                      *
*       for class informations                                                                    *
************************************************************************************************/
class DMatrixList
{
        public:
                DMatrixList();
                DMatrixList(vei_t hsize, vei_t vsize);

                ~DMatrixList();

                void create(vei_t hsize, vei_t vsize);

                //DVectorMatrix that contains the data of the list element
                DVectorMatrix *matrix;
                //pointer to the next DMatrixList element in the list
                DMatrixList *next;
                //pointer to the last DMatrixList element in the list
                DMatrixList *last;
                //special mean vector needed for the MVG
                DVector *mue;
                //class information needed for MVG
                vei_t classinf;
};
#endif
```

## A.3  Stochastic

```
/***************************************************************
                    stochastics.h  -  v.0.1
    begin              : Mon Oct 04 2004
    copyright          : (C) 2004 by Martin Berchtold,
    email              :
 ***************************************************************/

/***************************************************************
 *                                                             *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
```

```
 *    (at your option) any later version.                                        *
 *                                                                               *
 ********************************************************************************/
#ifndef STOCHASTICS_H
#define STOCHASTICS_H

#include <stdio.h>
#include <string.h>
#include <float.h>
#include <math.h>

#include "cstk_base/types.h"
#include "cstk_base/matrix/dvectormatrix.h"

/********************************************************************************
 *    Abstract base class for all stochastical analyses classes. The methods    *
 *    are matrix, vector and value corresponding to the return values.          *
 ********************************************************************************/
struct Stochastics{
    virtual ~Stochastics() {};

    virtual DVectorMatrix matrix(DVectorList& veclist)=0;
    virtual DVector vector(DVector& vec)=0;
    virtual vei_t value(DMatrixList& matlist, DVector& I)=0;
};
#endif
```

## A.3.1  Covariance

```
/********************************************************************
                       covariance.h  -  v.0.1
                       --------------------
     begin                 : Mo Oct 04 2004
     copyright             : (C) 2004 by Martin Berchtold
     email                 :
 ********************************************************************/
/********************************************************************
 *                                                                 *
 *    This program is free software; you can redistribute it and/or modify  *
 *    it under the terms of the GNU General Public License as published by  *
 *    the Free Software Foundation; either version 2 of the License, or     *
 *    (at your option) any later version.                          *
 *                                                                 *
 ********************************************************************/
#ifndef COVARIANCE_H
#define COVARIANCE_H

#include "cstk_base/stochastics/stochastics.h"

/********************************************************************************
 *      This class produces out of a DVectorList a covariance matrix.          *
 ********************************************************************************/
class Covariance : virtual public Stochastics
{
    public:
        Covariance();
        ~Covariance();
        /********************************************************************
         *        This is the main method that outputs as return value the  *
         *        calculated covariace matrix. The input of the method is a  *
         *        list of DVectors that represent a senor-arrays output over *
         *        the time.                                                  *
         ********************************************************************/
        DVectorMatrix matrix(DVectorList& veclist);

        //empty return method since there is no vector as result possible
        DVector vector(DVector& vec) {return vec;}
        //empty return method since there is no value as result possible
        vei_t value(DMatrixList& matlist, DVector& I) {return 0;}

        //over the time mean vector produced in calculations, needed for MVG
        DVector *mean;
    protected:
        //pointer to the resulting covariance matrix
        DVectorMatrix *matrix_cov;
    private:
        //calculation of the i'th row j'th column covariance matrix element
        oas_t cov(vei_t i, vei_t j, DVectorList& veclis);
};
#endif
```

## A.3.2  Multi Variate Gaussian

```
/********************************************************************
                       mvg.h  -  v.0.1
                       --------------------
     begin                 : Mo Oct 08 2004
     copyright             : (C) 2004 by Martin Berchtold
     email                 :
 ********************************************************************/
/********************************************************************
 *                                                                 *
 *    This program is free software; you can redistribute it and/or modify  *
 *    it under the terms of the GNU General Public License as published by  *
```

```
#ifndef MVG_H
#define MVG_H

#include "cstk_base/stochastics/stochastics.h"

/**************************************************************************************
 *   Simple class that performs a scan with a multi variate Gaussian model           *
 *   on a covariance matrix list according to a input vector. The result of the      *
 *   scan is a class information belonging to the 'winning' covariance matrix         *
 *   and mean vector tuple.                                                           *
 **************************************************************************************/
class MVG : virtual public Stochastics
{
    public:
        MVG();
        ~MVG();

        //empty return method since there is no matrix as result possible
        DVectorMatrix matrix(DVectorList& veclist) {DVectorMatrix mat; return mat;}
        //empty return method since there is no vector as result possible
        DVector vector(DVector& vec) {return vec;}

        /**************************************************************************************
         *    The method that performs the scan returns a value containing the               *
         *    class information of the hypothesis with the highest probability.              *
         **************************************************************************************/
        vei_t value(DMatrixList& matlist, DVector& I);
};
#endif
```

## A.4    K Nearest Neighbors

```
/**********************************************************************
                          __knn.h_v.2.01_____
      begin                 : Sept 23 2004
      copyright             : (C) 2004 by Martin Berchtold, Kristof Van Laerhoven
      email                 :
 **********************************************************************/
```

```
#ifndef KNN_H
#define KNN_H

#include "cstk_base/types.h"
#include "cstk_base/vector/dvector.h"

#define DIS_MANH 0
#define DIS_CHEB 1
#define DIS_EUCL 2
#define DIS_MINK 3

/**************************************************************************************
 *   Single linked DVector list element with copy constuctor and a vlue for          *
 *   class informations.                                                             *
 **************************************************************************************/
struct VectorPoCl {
        VectorPoCl() {vector=NULL;next=NULL;}
        VectorPoCl(vei_t nsize) {vector = new DVector(nsize);next = NULL;}
        VectorPoCl(const VectorPoCl& vect)
        {
            vector = new DVector(*(vect.vector));
            next = vect.next;
            vec_class = vect.vec_class;
        }
        ~VectorPoCl() {if (vector) delete vector;}

        void create(vei_t nsize) {vector = new DVector(nsize);}
        DVector *vector;
        VectorPoCl *next;
        vei_t vec_class;
};
/**************************************************************************************
 *   K neares neighbor algorithm that cunstructs a DVector list out of elements       *
 *   with a class information and determines the class of elements where the          *
 *   class value is not set. The class information is determined through the k        *
 *   nearest neighbors in the list of classified elements.                           *
 **************************************************************************************/
class KNN {
    private:
        // number k of the neighbors which count
        ve_t knn;
        // pointer to the first element in the list
        VectorPoCl* first;
        // pointer to the currently observed element in the list
        VectorPoCl* current;
```

```
        /*******************************************************************************
         *    This method determines the distance between the DVectors in the classified   *
         *    list and the input DVector according to the selected vector distance          *
         *    measurement. The exponent value is used in the Minkowski distance measurement. *
         *******************************************************************************/
        f_32b det_rad(const VectorPoCl& datav, char seldis, vei_t exp);

        vei_t *kclasses;
        f_32b *kdist;
        f_32b wincl;    // last overall winner's weight
        f_32b allcl;    // sum of all winner's weight
    public:
        KNN();
        KNN(vei_t k);
        ~KNN();

        /*******************************************************************************
         *    Main method which determines if the element should be saved in the list     *
         *    or if the class information that is not yet set should be determined. If      *
         *    the element should be saved it is done in this method by setting the next     *
         *    pointer of the element and the first pointer of the algorithm. Should the     *
         *    class information should be retrieved, the get_k_dis method is accessed.      *
         *******************************************************************************/
        vei_t access(const VectorPoCl& datav, char dist=DIS_EUCL, vei_t expo=2);

        /*******************************************************************************
         *    This method determines the class of the input element according to its       *
         *    k nearest neighbors and returns the result. It is the actual implementation   *
         *    of the KNN algorithm.                                                         *
         *******************************************************************************/
        vei_t get_k_dis(const VectorPoCl& datav, char dist=DIS_EUCL, vei_t expo=2);

        // returns the i'th class value of the k neighbours to the last input element
        vei_t get_class(vei_t i) { return (i<knn)?kclasses[i]:-1; };
        // returns the i'th distance value of the k neighbours to the last input element
        f_32b get_dist(vei_t i) { return (i<knn)?kdist[i]:-1; };
        // returns the confidence of the last retrieved class information
        f_32b get_conf() { return (wincl/allcl); };

        u_32b num_prototypes;
};
#endif
```

## A.5  Sparse Distributed Memory

### A.5.1  SDMA

```
/**************************************************************************
                        _sdma.h_-__v.1.00_
     begin              : Jun 01 2004
     copyright          : (C) 2002-2004 by Martin Berchtold
     email              :
 **************************************************************************/

/**************************************************************************
 *                                                                        *
 *   This program is free software; you can redistribute it and/or modify *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                        *
 **************************************************************************/

#ifndef SDMA_H
#define SDMA_H

#include <stdlib.h>
#include "cstk_base/vector/binvector.h" // for addresses and data
#include "cstk_base/vector/bvector.h" // for the counters

typedef struct SDMA_PARAM
{
    // size of the memory in data and address vector tupels
    vei_t size;
    // size of the address vectors
    vei_t asize;
    // size of the data vectors
    vei_t dsize;
    // threshold percentage for distribution of the data according to the distance
    // of the address vectors
    oas_t thresholdp;
    // threshold for SOSDM
    vei_t threshold;
};
/*******************************************************************************
 *   Sparse Distributed Memory Approach implementation that distributes the        *
 *   input data vector weighted over the whole memory.                             *
 *******************************************************************************/
class SDMA
{
 public:
    SDMA();
    SDMA(vei_t nsize, vei_t nasize, vei_t ndsize, oas_t thp=0);
    ~SDMA();

        /*******************************************************************************
```

```
        *    Method to create the SDMA with specification for the size of the         *
        *    data and address vectors, size of the memory in entries and              *
        *    threshold percentage for the distribution of the input data.             *
        ******************************************************************************/
        void create(vei_t nsize, vei_t nasize, vei_t ndsize, oas_t thp=0);
        /******************************************************************************
        *    This method initializes the data and address vectors of the memory       *
        *    with random bits.                                                         *
        ******************************************************************************/
        void random_init();

        /******************************************************************************
        *    With this method address and data vector tupels are deleted from          *
        *    the memory according to the usage value.                                  *
        ******************************************************************************/
        vei_t remove(oas_t usage);
        /******************************************************************************
        *    Method to store data vectors in the memory according to the address       *
        *    vector distances.                                                         *
        ******************************************************************************/
        vei_t store(BinVector& v1, BinVector& b1, bool det_radius);
        /******************************************************************************
        *    Method to retrieve data vectors out of the memory according to the        *
        *    address vector distances.                                                 *
        ******************************************************************************/
        vei_t retrieve(BinVector& v1, BinVector& tsum,
                       BVector<oas_t>& tempsum, bool det_radius);
    protected:
        BinVector       *av; // all the addresses are binary vectors
        BVector<oas_t> *av_tmp; // temporary data vectors
        BVector<oas_t> *dvd; // all the data vectors are vectors of counters

        // determines the smallest distance of input address and memory addresses
        // for relative distribution of the data vector
        vei_t radius(BinVector& v1);

        oas_t tmp;

        SDMA_PARAM par;
};
#endif
```

## A.5.2   SDMA Functions

```
/*****************************************************************************
                        _sdmafct.h__-__v.1.00_
        begin                 : Jun 01 2004
        copyright             : (C) 2002-2004 by Martin Berchtold
        email                 :
 *****************************************************************************/

/*****************************************************************************
 *                                                                           *
 *    This program is free software; you can redistribute it and/or modify   *
 *    it under the terms of the GNU General Public License as published by    *
 *    the Free Software Foundation; either version 2 of the License, or        *
 *    (at your option) any later version.                                     *
 *                                                                           *
 *****************************************************************************/

#ifndef SDMAFCT_H
#define SDMAFCT_H

#include "sdma.h"
#define e 2.7182818284
#define pi 3.1415926535

/*******************************************************************************
 *    Child class of SDMA that realizes several different store/retrieve methods  *
 *    for different distribution functions. The initialization method and the     *
 *    constructors are used form the parent class.                                *
 *******************************************************************************/
class SDMAfct : public SDMA
{
 public:
    SDMAfct();
    SDMAfct(vei_t nsize, vei_t nasize, vei_t ndsize);
    ~SDMAfct();

    // gauss distribution
    vei_t store_gauss(BinVector& v1, BinVector& b1, bool det_radius,
                      oas_t density);
    vei_t retrieve_gauss(BinVector& v1, BinVector& tsum,
                         BVector<oas_t>& tempsum, bool det_radius,
                         oas_t density);

    // NAND distance measurement
    vei_t store_nand(BinVector& v1, BinVector& b1, bool det_radius);
    vei_t retrieve_nand(BinVector& v1, BinVector& tsum,
                        BVector<oas_t>& tempsum, bool det_radius);
 private:
    vei_t radius_nand(BinVector& v1);
};
#endif
```

## A.5.3 Self Organizing SDM

```
/************************************************************************
                          _sosdm.h_-__v.1.00_
                          -----------------
    begin               : Jun 01 2004
    copyright           : (C) 2002-2004 by Martin Berchtold
    email               :
 ************************************************************************/

/************************************************************************
 *                                                                      *
 *    This program is free software; you can redistribute it and/or modify *
 *    it under the terms of the GNU General Public License as published by *
 *    the Free Software Foundation; either version 2 of the License, or  *
 *    (at your option) any later version.                               *
 *                                                                      *
 ************************************************************************/

#ifndef SOSDM_H
#define SOSDM_H

#include "sdma.h"
#define e 2.7182818284
#define pi 3.1415926535

/*************************************************************************
 *    Self Organizing Sparse Distributed Memory which inherits the initialization *
 *    method form SDMA and uses the parameter type. The self organizing aspect of *
 *    this algorithm is focused on the addresses of the memory, which are moved *
 *    to the input addresses during an initialization phase.            *
 *************************************************************************/
class SOSDM : public SDMA
{
 public:
    SOSDM();
    ~SOSDM();

    void create_SO(vei_t thrs);
    vei_t store_SO(BinVector& v1, BinVector& b1);
    vei_t retrieve_SO(BinVector& v1, BinVector& tsum,
                      BVector<oas_t>& tempsum);
    void address_update();
    void del_datavec();
    void del_errovec();
    void destroy_errvec();
 private:
    vei_t max_invdist(BinVector& v1);
    oas_t signal_thres(BinVector& v1, BinVector& v2,
                       vei_t maxdis);
    void error_calc(BinVector& v1, BinVector& v2, oas_t sigthr,
                    vei_t cunter);

    BVector<oas_t> *errv; // all error vectors are vectors of counters

    oas_t total_sig;
};
#endif
```

## A.6 K-Means Clustering

```
/************************************************************************
                          _____kmeans.h_____
                          -------------------
    begin               : Sept 28 2004
    copyright           : (C) 2004 by Martin Berchtold
    email               :
 ************************************************************************/

/************************************************************************
 *                                                                      *
 *    This program is free software; you can redistribute it and/or modify *
 *    it under the terms of the GNU General Public License as published by *
 *    the Free Software Foundation; either version 2 of the License, or  *
 *    (at your option) any later version.                               *
 *                                                                      *
 ************************************************************************/

#ifndef KMEANS_H
#define KMEANS_H

#include "cstk_base/types.h"
#include "cstk_base/vector/dvector.h"

#define DIS_MANH 0
#define DIS_CHEB 1
#define DIS_EUCL 2
#define DIS_MINK 3

/*************************************************************************
 *    The K-Means clustering clusters the input data in k different clusters. *
 *************************************************************************/

class KMeans
{
    public:
        KMeans(); //empty initialization
        //initialize values - specifies k, alpha(multipl. constant) and distance measurement
        KMeans(vei_t buckets, oas_t alpha, ve_t seldist=DIS_EUCL, vei_t exp=2);
        ~KMeans();

        //initialize values - specifies k, alpha(multipl. constant) and distance measurement
```

```
        void create(vei_t buckets, oas_t alpha, ve_t seldist=DIS_EUCL, vei_t exp=2);
        void initialize(DVector& min, DVector& max); //initializes buckets vectors randomly
        void initialize(DVector& bvec, vei_t bnum);  //initializes buckets with input vector
        /*************************************************************************
        *    This method updates the bucket with the smallest distance to the   *
        *    input vector by a small portion of the difference between input    *
        *    vector and bucket vector.                                          *
        *************************************************************************/
        vei_t update_bucket(DVector& vec);
        /*************************************************************************
        *    Method to retrieve the class information from the bucket with the  *
        *    smallest distance to the input vector.                             *
        *************************************************************************/
        vei_t retrieve_cluster(DVector& vec);

        DVector* bucket;
    private:
        //returns the distance between the two input vectors according to the selected distance
        oas_t det_rad(DVector& vector, DVector& datav, ve_t seldis, vei_t exp);

        vei_t bucket_num, exponent;
        oas_t alpval;
        ve_t selected_dist;
};
#endif
```

## A.7 Kohonen Self Organizing Map

```
/****************************************************************************
                            ksom.h  -  v.1.00
                            -----------------
    begin                : Mon Oct 25 2004
    copyright            : (C) 2004 by Martin Berchtold
    email                :
 ****************************************************************************/

/****************************************************************************
 *                                                                          *
 *    This program is free software; you can redistribute it and/or modify  *
 *    it under the terms of the GNU General Public License as published by   *
 *    the Free Software Foundation; either version 2 of the License, or      *
 *    (at your option) any later version.                                    *
 *                                                                          *
 ****************************************************************************/
#ifndef KSOM_H
#define KSOM_H

#include <stdlib.h>  // rand
#include <math.h>

#include "cstk_base/vector/dvector.h"   // for the types & basic vector
#include "cstk_base/vector/kvector.h"   // x/y winner and max

#define DIS_MANH 0
#define DIS_CHEB 1
#define DIS_EUCL 2
#define DIS_MINK 3

#define LIN 0
#define LOG 1
#define EXP 2

#define EUCLNB 0
#define MANHNB 1
#define CHEBNB 2
#define MINKNB 3
#define MEXNB 4
#define GAUSSNB 5

typedef struct KSOM_PARAM
{
    /*---parameters used for automatig determination of the learning rate---*/
    /*----------------------------------------------------------------------*/
    /*automatic decreasing learning rate*/
    bool autol;
    /*specification of the function that decreases the learning rate*/
    ve_t lfct;
    /*constant that is used by learning rate calculations*/
    oas_t c;
    /*number of epochs trained jet*/
    vei_t epoch;
    /*----------------------------------------------------------------------*/

    /*---parameters to determine the neighbourhood---*/
    /*-----------------------------------------------*/
    /*specification of neighbourhood function*/
    ve_t nfct;
    /*neighbourhood radius*/
    oas_t nb_radius;
    /*---------------------------------------------*/

    /*---specification of the distance masurement used to determine the winner neuron---*/
    /*----------------------------------------------------------------------------------*/
    /*selection of the distance masurement*/
    ve_t dist;
    /*exponent for Minkowski distance*/
    vei_t minkexp;
    /*----------------------------------------------------------------------------------*/

    /*---parameters for the maxican hat neighbour function---*/
```

```
    /*------------------------------------------------------*/
    /*parameter that specifies the entrance of the lower zero part of the maxican hat function*/
    oas_t d;
    /*constant for mexican hat function for f(x)<0*/
    oas_t roh;
    /*------------------------------------------------------*/
};
/***************************************************************************
*   Kohonen Self Organizing Map (2-dimensional) implementation with several        *
*   different neighborhood functions, automatically determination of the           *
*   learning rate and different measurements for the distances of neurons in       *
*   the grid.                                                                       *
***************************************************************************/
class KSOM {
 public:
    KSOM();
    /* create the map with the right dimensions and all parameters*/
    KSOM(vei_t x, vei_t y, vei_t n, ve_t distance=DIS_EUCL, ve_t neighbourfct=EUCLNB,
         bool autolearn=false, ve_t learnfct=LIN);
    ~KSOM();
    /* create the map with the right dimensions and all parameters*/
    void create(vei_t x, vei_t y, vei_t n, ve_t distance=DIS_EUCL, ve_t neighbourfct=EUCLNB,
                bool autolearn=false, ve_t learnfct=LIN);
    /* randomize the contents of the map's cells, prototyp vector spezializes the range of each dimension*/
    void initRandom(DVector& prototyp);
    /* feed a vector into the map, lr specifies the learning rate when autolearn=false*/
    void feed(DVector& vec, float lr);
    /* get the value of the cell x/y*/
    oas_t getCell(vei_t x, vei_t y, vei_t i);

    /* winner coords on the map*/
    KVector *win_xy;
    vei_t winner_x, winner_y;
    vei_t max_x, max_y;

    KSOM_PARAM par;
 protected:
    oas_t det_dis(DVector& vec1, DVector& vec2);
    oas_t det_dis(KVector& vec1, KVector& vec2);
    oas_t det_nb(KVector& vec1, KVector& vec2, ve_t fct);

    oas_t det_lr(oas_t lr);
    KVector  *max_xy;
    DVector *vect;
};
/***************************************************************************
*   This is a child class of the KSOM class that implements different              *
*   functions to improve the KSOM algorithm.                                       *
***************************************************************************/
class KSOMfct : public KSOM
{
 public:
    /***************************************************************************
    *   This method realizes a different way of training the neuron grid using         *
    *   the distance of the input vector to the neurons in the grid to distribute      *
    *   the data over the map. No determination of a winner neuron is needed in        *
    *   this case. The new feed method works only on a pre trained map.                *
    ***************************************************************************/
    void feed_NoWinner(DVector& vec, float lr);
 private:
    oas_t det_nb(oas_t dist, ve_t fct);
};
#endif
```

# B   UML Diagrams

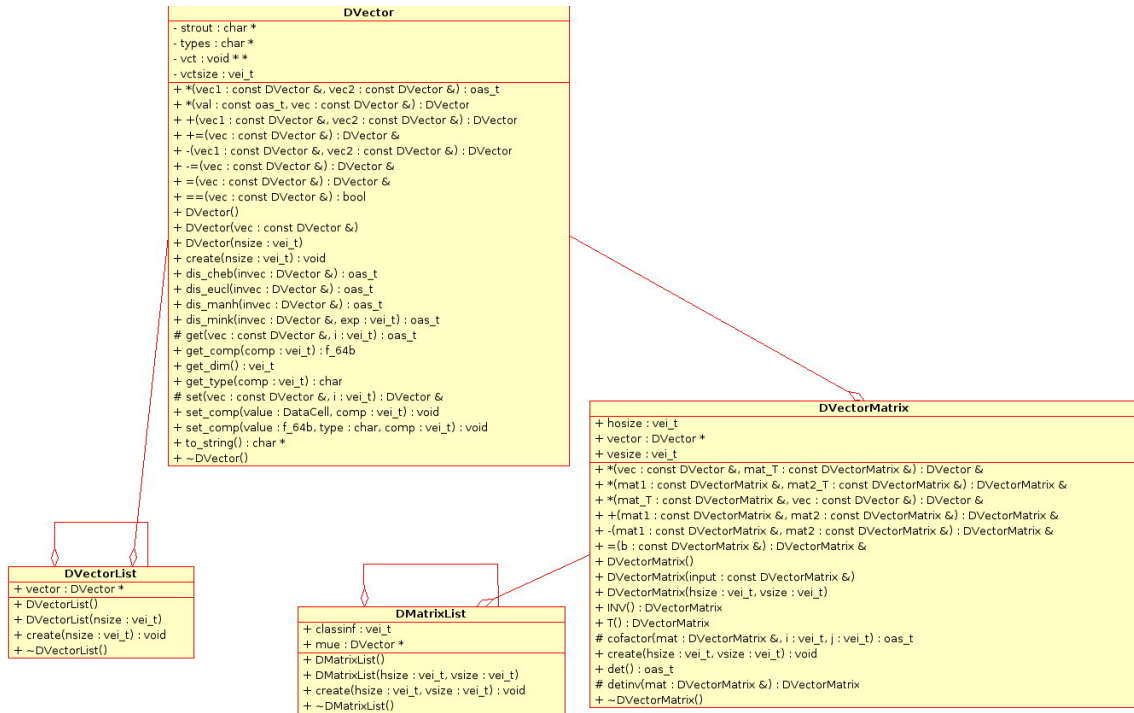## B.1   DVector, DVectorList, DVectorMatrix and DMatrixList



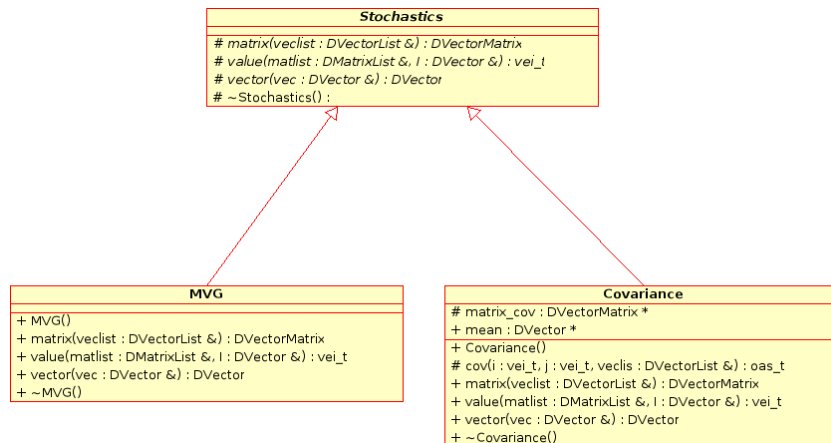Figure 13: DVector UML Diagram - with derivations

## B.2   Stochastic



Figure 14: Stochastic UML Diagram

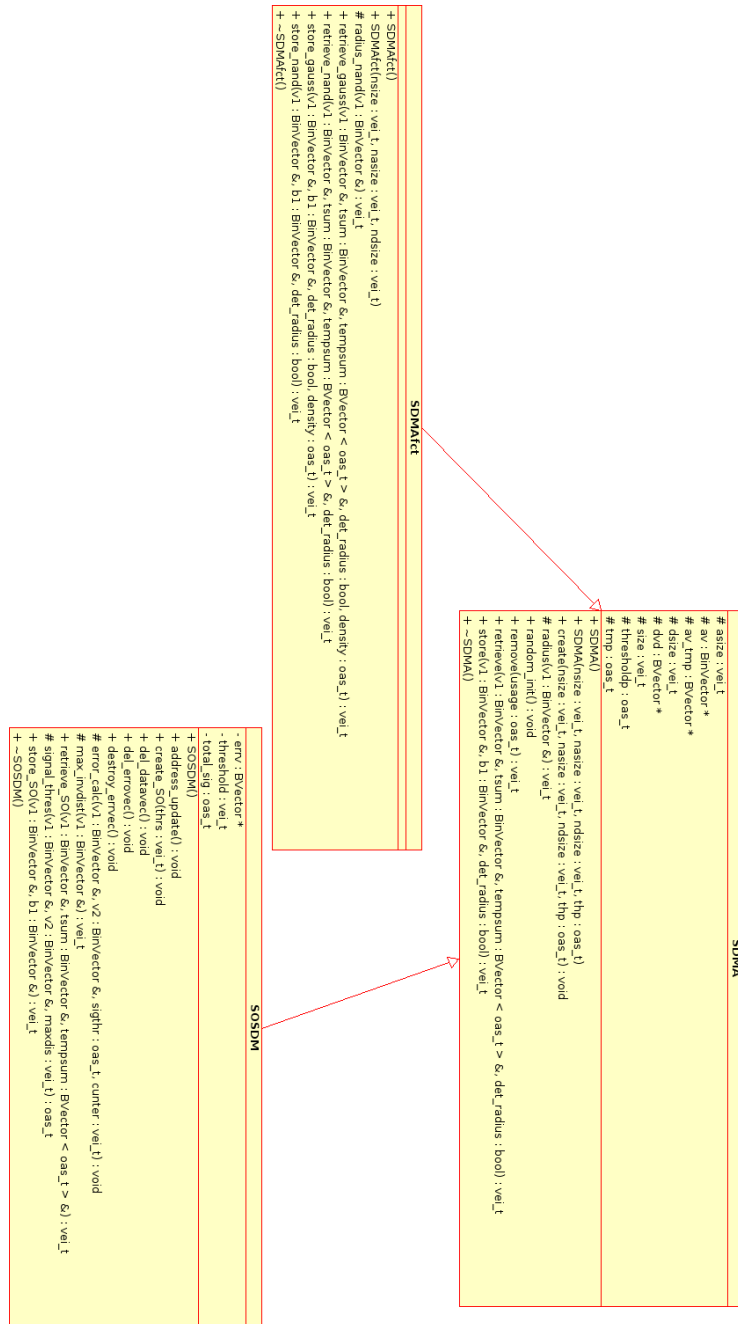## B.3 Sparse Distributed Memory



Figure 15: Sparse Distributed Memory UML Diagram - all classes
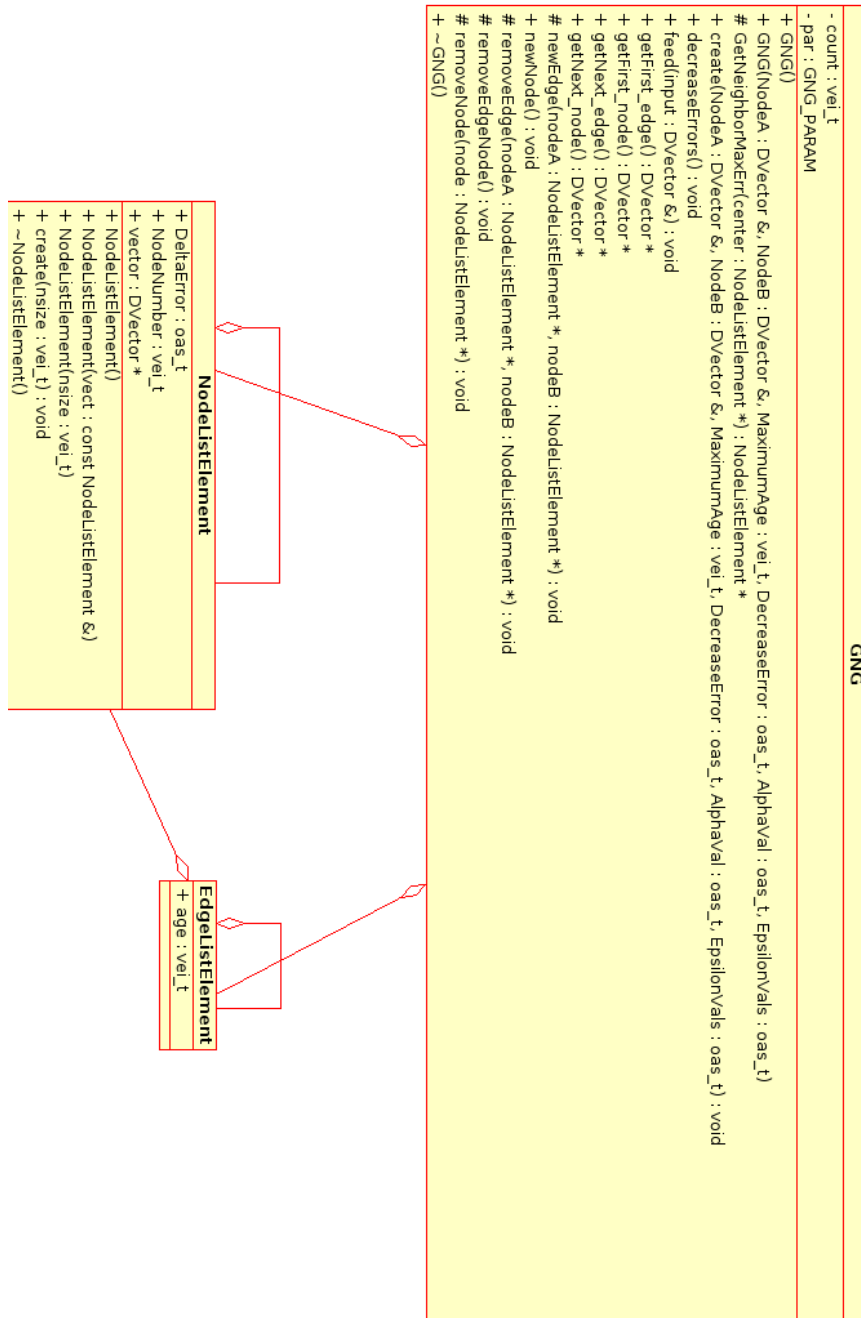
## B.4  Growing Neural Gas



Figure 16: Growing Neural Gas UML Diagram - with list classes

# C Paper Submission: REAL-TIME ANALYSIS OF COR-RELATIONS BETWEEN ON-BODY SENSOR NODES (WITH TOPOLOGICAL MAP ARCHITECTURES)

The attached paper was submitted to the 'International Workshop on Wearable and Implantable Body Sensor Networks' at 8th of February 2005. Since the assumptions of this paper are based on the use of the CSTK algorithms and show their behavior in execution it is useful to attach it here. Each test done in this paper could have been placed in the corresponding sections, but this would have broken the contiguity of the statement.