SOM Toolbox for Matlab 5

Juha Vesanto, Johan Himberg,
Esa Alhoniemi, and Juha Parhankangas

SOM Toolbox Team
Helsinki University of Technology
P.O.Box 5400, FIN-02015 HUT, Finland
somtlbx@mail.cis.hut.fi
http://www.cis.hut.fi/projects/somtoolbox/

Report A57
April 2000

ISBN 951-22-4951-0
ISSN 1456-2243
Libella Oy
Espoo 2000

SOM Toolbox for Matlab 5

Juha Vesanto, Johan Himberg,
Esa Alhoniemi, and Juha Parhankangas

SOM Toolbox Team
Helsinki University of Technology
P.O.Box 5400, FIN-02015 HUT, Finland

somtlbx@mail.cis.hut.fi
http://www.cis.hut.fi/projects/somtoolbox/

April 20, 2000

Abstract

Self-Organizing Map (SOM) is an unsupervised neural network method which has properties of both vector quantization and vector projection algorithms. The prototype vectors are positioned on a regular low-dimensional grid in an ordered fashion, making the SOM a powerful visualization tool.

SOM Toolbox is an implementation of the SOM and its visualization in the Matlab 5 computing environment. The Toolbox can be used to preprocess data, initialize and train SOMs using a range of different kinds of topologies, visualize SOMs in various ways, and analyze the properties of the SOMs and the data, e.g. SOM quality, clusters on the map, and correlations between variables. With data mining in mind, the Toolbox and the SOM in general are best suited for the data understanding phase, although they can also be used for modeling.

SOM Toolbox can be applied to the analysis of single table data with numerical variables. It is easily applicable to small data sets (under 10000 records) but can also be applied in case of medium sized data sets (upto 1000000 records). The Toolbox is mainly suitable for training maps with 1000 map units or less.

Contents

1 General

1.1 About this report

This report presents the SOM Toolbox (version 2) [23], hereafter simply called the Toolbox, for Matlab 5 computing environment. The SOM stands for Self-Organizing Map (also called Self-Organizing Feature Map, Kohonen map), a popular neural network based on unsupervised learning [12]. The Toolbox contains functions for the creation, visualization and analysis of Self-Organizing Maps. The Toolbox as well as this report and other related works are available free of charge under the GNU General Public License from

`http://www.cis.hut.fi/projects/somtoolbox/`

Below, guidelines are given how to obtain and install the Toolbox. In Section 2, the SOM is shortly introduced. Section 3 gives guidelines to how the Toolbox can and should be used. Section 4 gives detailed information about how the Toolbox really works: in particular, the structs and functions of the Toolbox are described.

The first version of the Toolbox was released in autumn 1997 [21]. Compared to that, this second version has some major changes as well as numerous smaller improvements and additions. The major changes are in the structs, normalization and visualization. See Appendix A for more details. The Toolbox is closely related to the SOM_PAK, a program package implementing the SOM algorithm in C-code [14]. The Toolbox contains functions for using SOM_PAK programs from Matlab, and has functions for reading and writing data files in SOM_PAK format, see Appendix B.

Notation. Primarily two font types are used in this report: regular and `fixed`. The latter is used to write out Matlab and shell commands and functions, string literals, file names and web addresses. Also sizes of matrices are typed with fixed font, for example `[1000 x 10]` means a matrix with 1000 rows and 10 columns.

1.2 About Matlab

Matlab is a widely used programming environment for technical computing by MathWorks, Inc. Matlab is available for Windows and various Unix platforms. An evaluation version can be downloaded from the Internet, see `http://www.mathworks.com/`.

Matlab features a high-level programming language, powerful visualization, graphical user interface tools and a very efficient implementation of matrix calculus. These are major advantages in the data mining research because they allow fast prototyping, testing and customizing of the algorithms. There are also a large number of toolboxes intended for a variety of modeling and analysis tasks. These toolboxes are based on a wide span of methodologies from statistical methods to Bayesian networks.

Matlab can accommodate continuous, categorial and symbolic data. The data model is a single table, but since the Matlab language is highly flexible it is possible to build more complicated models. Direct support for object oriented programming is available. With special toolboxes, connection to (ODBC) databases is possible.

In Matlab, the data mining process is typically highly interactive requiring adequate knowledge of programming from the user. However, using the graphical user interface tools it is possible to build simpler end-user interfaces and even create stand-alone applications.

In order to get the most out of this report, some prior familiarity with Matlab would be good. However, Matlab has a pretty extensive online help. If you encounter a function that you do not recognize, try the `help` command followed by the function name, for example

```
help plot
```

1.3 About SOM Toolbox

The Toolbox was born out of need for a good, easy-to-use implementation of the SOM in Matlab for research purposes. In particular, the researchers responsible for the Toolbox work in the field of data mining, and therefore the Toolbox is oriented towards that direction in the form of powerful visualization functions. However, also people doing other kind of research using SOM will probably find it useful — especially if they haven't yet made a SOM implementation of their own in Matlab environment. Since much effort has been put into making the Toolbox relatively easy to use, it can also be used for educational purposes.

The Toolbox can be used to preprocess data, initialize and train SOMs using a range of different kinds of topologies, visualize SOMs in various ways, and analyze the properties of the SOMs and data, for example SOM quality, clusters on the map and correlations between variables[1]. With data mining in mind, the Toolbox and the SOM in general is best suited for the data understanding/survey phase, although it can also be used for modeling [4, 24].

The Toolbox consists of two sets of functions: the basic package and the contributed functions. The basic package is meant to be self-sufficient and well-documented. Except for the lowest level subroutines, each function has a short help at the beginning of the file, and a longer help with full details immediately after it. Try `help` and `type` commands to view the helps. For example:

```
help som_hits % to see the short help
more on       % this may be necessary
type som_hits % to see the longer help
```

Often the help part is longer than the code itself. The contributed functions are add-ons to the Toolbox. While useful, they are not essential, and in most cases they are less well documented than the basic package.

The basic package is maintained and copyrighted by the SOM Toolbox Team. The copyright statement (GNU General Public License version 2 or later) allows you to freely make use of, modify and distribute the functions as long as the copyright statement is included in the distribution. Note that the license does not grant you the right to include the codes in your own propietary program to be sold as your own. Refer to the `Copyright.txt` and `License.txt` files in the SOM Toolbox distribution for the exact wording of do's and do-not's. For further details concerning licensing etc. please refer to the SOM Toolbox website.

The contributed functions may have their own copyright notices, but some do not. In such a case, it should be assumed that the GNU General Public License holds for them, too, but so that the author(s) of the file (listed in the file) have the Copyright. You are also welcome to contribute your codes to the Toolbox: just send your questions and/or contributions to `somtlbx@mail.cis.hut.fi`. If you rather present your work on your own site, please provide at least the URL of the site so we can link to it.

This report concentrates on the basic package, and only mentions some contributed functions.

1.4 System requirements

The primary requirement is that you have a Matlab, version 5.2 at least[2]. Just the basic Matlab is sufficient, no other toolboxes are needed.

Secondary requirement is to have enough memory: the Toolbox uses quite a lot of memory to speed things up. Matlab itself requires at least 16 MB of memory, but the suggestion is to

[1]... or components: for historical reasons the term "component" is sometimes used instead of "variable" in this report. Several scalar variables make up a vector. Thus the they are the components of the vector.

[2]Version 5.1 may work, too, if you remove the `try - catch` commands.

have much more than that. To use the Toolbox, we recommend at least 64 MBs of memory, preferably more. As an illustrative example, consider making a SOM from a data matrix of size `[10000 x 10]`. The data matrix alone requires almost 1 MB of memory, but what really consumes memory is training. The default number of map units for the above data would be 500. The training procedures would use one or more matrices of size `[500 x 500]`. The size of these matrices quickly becomes overwhelming as the number of map units increases. In the case above, the batch training procedure would reserve at least 10 MBs of memory.

Finally, the total diskspace required for the Toolbox itself is less than 2 MBs. The documentation takes a few MBs more.

1.5 Installation

The Toolbox can be downloaded for free from

`http://www.cis.hut.fi/projects/somtoolbox/`

Once you have downloaded the relevant files (e.g. `somtoolbox2.zip`), move them to an appropriate directory and decompress them using, e.g., `unzip`, `pkunzip` or `winzip`. After this, install the Toolbox like you would install any other toolbox. If you don't know how to do that, just make sure that when you want to utilize the SOM Toolbox, you're either in the Toolbox directory, or the directory is in your matlabpath (see commands `path`, `addpath` and `pathtool`).

On-line help is available from the SOM Toolbox website, but it may be more convenient to put them somewhere locally. You can do this by decompressing the documentation file (e.g. `somtoolbox2doc.zip`) in an appropriate directory.

2 Self-Organizing Map (SOM)

A SOM consists of neurons organized on a regular low-dimensional grid. The number of neurons may vary from a few dozen up to several thousand. Each neuron is represented by a d-dimensional weight vector (a.k.a. prototype vector, codebook vector) $\mathbf{m} = [m_1, \ldots, m_d]$, where d is equal to the dimension of the input vectors. The neurons are connected to adjacent neurons by a neighborhood relation, which dictates the topology, or structure, of the map. In the Toolbox, topology is divided to two factors: local lattice structure and global map shape. Examples of rectangular and hexagonal lattice structures are shown in Figure 1 and examples of different kinds of map shapes in Figure 2.

The SOM training algorithm resembles vector quantization (VQ) algorithms, such as k-means [6]. The important distinction is that in addition to the best-matching weight vector, also its topological neighbors on the map are updated: the region around the best-matching vector is stretched towards the presented training sample, as in Figure 3. The end result is that the neurons on the grid become ordered: neighboring neurons have similar weight vectors.

Since the weight vectors of the SOM have well-defined low-dimensional coordinates $\mathbf{r}_i$ on the map grid, the SOM is also a vector projection algorithm [8]. Together the prototype vectors and their projection define a low-dimensional map of the data manifold.

2.1 Sequential training algorithm

The SOM is trained iteratively. In each training step, one sample vector $\mathbf{x}$ from the input data set is chosen randomly and the distances between it and all the weight vectors of the SOM are calculated using some distance measure. The neuron whose weight vector is closest to the input vector $\mathbf{x}$ is called the Best-Matching Unit (BMU), denoted here by c:

$$||\mathbf{x} - \mathbf{m}_c|| = \min_i \{||\mathbf{x} - \mathbf{m}_i||\}, \tag{1}$$

where $||\cdot||$ is the distance measure, typically Euclidian distance. In the Toolbox, the distance computation is slightly more complicated because of two factors:

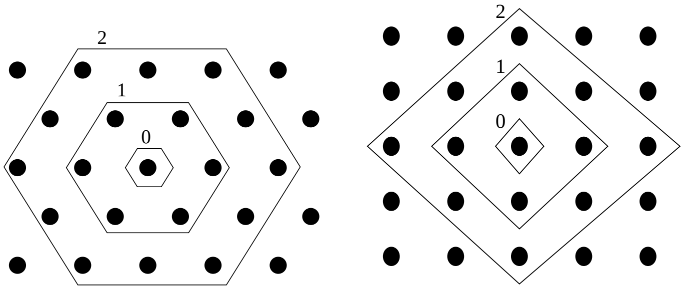
- Missing values: In the Toolbox, these are represented by the value `NaN` in the vector or data matrix. Missing components are handled by simply excluding them from the distance calculation (ie. it is assumed that their contribution to the distance $||\mathbf{x} - \mathbf{m}_i||$ is zero). Because the same variable(s) is ignored in each distance calculation (over which the minimum is taken), this is a valid solution [18].

- Mask: Each variable has an associated weighting factor, defined in the `.mask` field of map and training structs (see Section 4). This is primarily used in binary form for excluding certain variables from the BMU-finding process (1 for include, 0 for exclude). However, the mask can get any values, so it can be used for weighting variables according to their importance.

With these changes, the distance measure becomes:

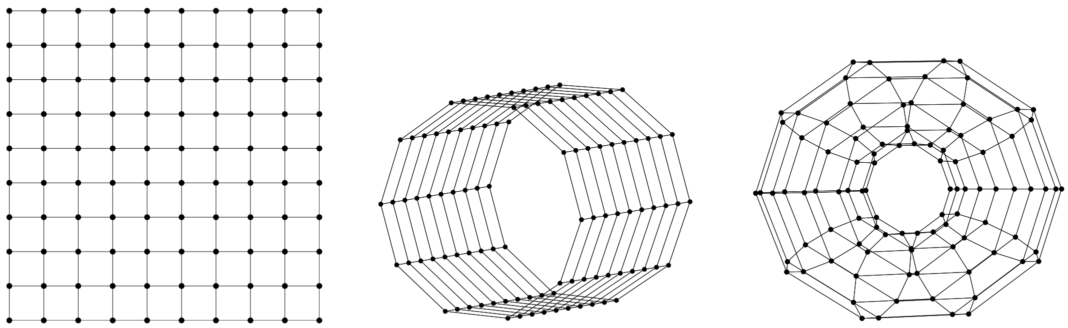
$$||\mathbf{x} - \mathbf{m}||^2 = \sum_{k \in K} w_k (x_k - m_k)^2, \tag{2}$$

where K is the set of known (not missing) variables of sample vector $\mathbf{x}$, x_k and m_k are kth components of the sample and weight vectors and w_k is the kth mask value (`mask(k)`).



(a) Hexagonal grid (b) Rectangular grid

Figure 1: Discrete neighborhoods (size 0, 1 and 2) of the centermost unit: (a) hexagonal lattice, (b) rectangular lattice. The innermost polygon corresponds to 0-neighborhood, the second to the 1-neighborhood and the biggest to the 2-neighborhood.



(a) Sheet (b) Cylinder (c) Toroid

Figure 2: Different map shapes. The default sheet shape (a), and two shapes where the map topology accommodates circular data: cylinder (b) and toroid (c).

After finding the BMU, the weight vectors of the SOM are updated so that the BMU is moved closer to the input vector in the input space. The topological neighbors of the BMU are treated similarly. This adaptation procedure stretches the BMU and its topological neighbors towards the sample vector as shown in Figure 3.

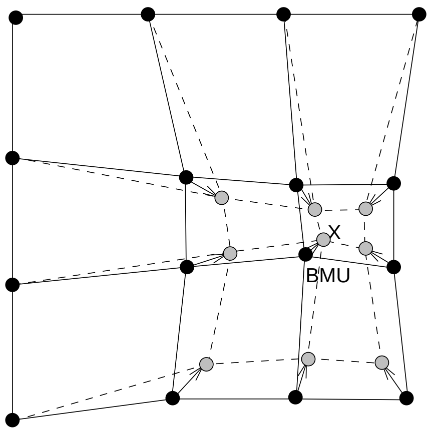


Figure 3: Updating the best matching unit (BMU) and its neighbors towards the input sample marked with $\mathbf{x}$. The solid and dashed lines correspond to situation before and after updating, respectively.

The SOM update rule for the weight vector of unit i is:

$$\mathbf{m}_i(t+1) = \mathbf{m}_i(t) + \alpha(t)h_{ci}(t)[\mathbf{x}(t) - \mathbf{m}_i(t)], \tag{3}$$

where t denotes time. The $\mathbf{x}(t)$ is an input vector randomly drawn from the input data set at time t, $h_{ci}(t)$ the neighborhood kernel around the winner unit c and $\alpha(t)$ the learning rate at time t, see Figures 4 and 5. The neighborhood kernel is a non-increasing function of time and of the distance of unit i from the winner unit c. It defines the region of influence that the input sample has on the SOM.

The training is usually performed in two phases. In the first phase, relatively large initial learning rate α_0 and neighborhood radius σ_0 are used. In the second phase both learning rate and neighborhood radius are small right from the beginning. This procedure corresponds to first tuning the SOM approximately to the same space as the input data and then fine-tuning the map.

2.2 Batch training algorithm

Also batch training algorithm is iterative, but instead of using a single data vector at a time, the whole data set is presented to the map before any adjustments are made — hence the name "batch". In each training step, the data set is partitioned according to the Voronoi regions of the map weight vectors, ie. each data vector belongs to the data set of the map unit to which it is closest. After this, the new weight vectors are calculated as:

$$\mathbf{m}_i(t+1) = \frac{\sum_{j=1}^{n} h_{ic}(t)\mathbf{x}_j}{\sum_{j=1}^{n} h_{ic}(t)}, \tag{4}$$

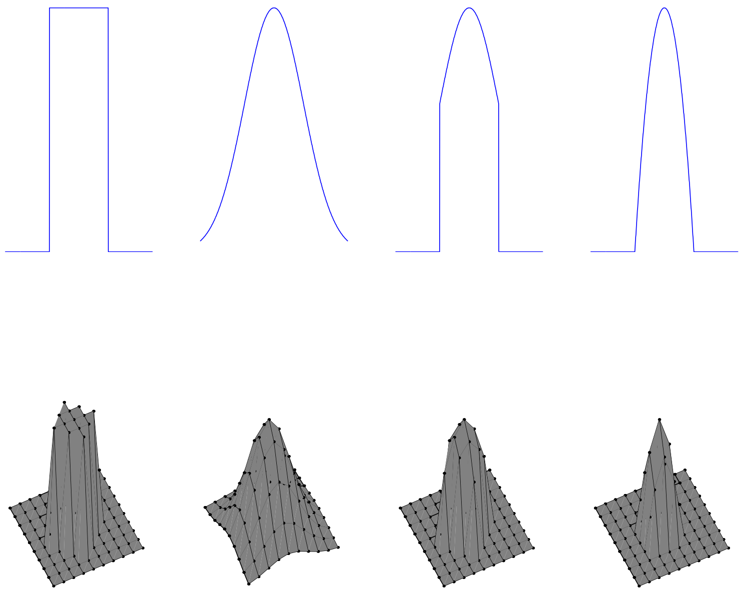


Figure 4: Different neighborhood functions. From the left `'bubble'` $h_{ci}(t) = \mathbf{1}(\sigma_t - d_{ci})$, `'gaussian'` $h_{ci}(t) = e^{-d_{ci}^2/2\sigma_t^2}$, `'cutgauss'` $h_{ci}(t) = e^{-d_{ci}^2/2\sigma_t^2}\mathbf{1}(\sigma_t - d_{ci})$, and `'ep'` $h_{ci}(t) = \max\{0, 1-(\sigma_t - d_{ci})^2\}$, where σ_t is the neighborhood radius at time t, $d_{ci} = ||\mathbf{r}_c - \mathbf{r}_i||$ is the distance between map units c and i on the map grid and $\mathbf{1}(x)$ is the step function: $\mathbf{1}(x) = 0$ if $x < 0$ and $\mathbf{1}(x) = 1$ if $x \geq 0$. The top row shows the function in 1- and the bottom row on a 2-dimensional map grid. The neighborhood radius used is $\sigma_t = 2$.

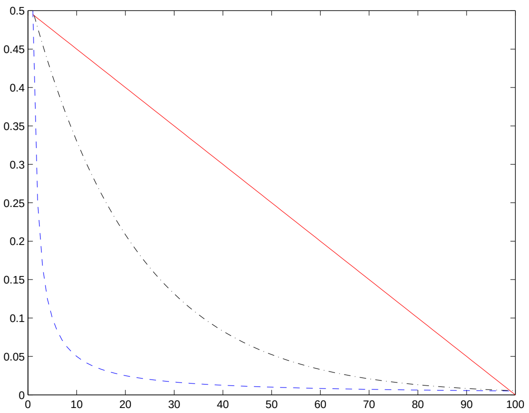


Figure 5: Different learning rate functions: `'linear'` (solid line) $\alpha(t) = \alpha_0\,(1 - t/T)$, `'power'` (dot-dashed) $\alpha(t) = \alpha_0(0.005/\alpha_0)^{t/T}$ and `'inv'` (dashed) $\alpha(t) = \alpha_0/(1 + 100\,t/T)$, where T is the training length and α_0 is the initial learning rate.

where $c = \arg\min_k\{||\mathbf{x}_j - \mathbf{m}_k||\}$ is the index of the BMU of data sample $\mathbf{x}_j$. The new weight vector is a weighted average of the data samples, where the weight of each data sample is the neighborhood function value $h_{ic}(t)$ at its BMU c. As in the sequential training algorithm, missing values are simply ignored in calculating the weighted average.

Notice that in the batch version of the k-means algorithm, the new weight vectors are simply averages of the Voronoi data sets. The above equation equals this if $h_{ic} = \delta(i, c)$.

Alternatively, one can first calculate the sum of the vectors in each Voronoi set:

$$\mathbf{s}_i(t) = \sum_{j=1}^{n_{V_i}} \mathbf{x}_j, \tag{5}$$

where n_{V_i} is the number of samples in the Voronoi set of unit i. Then, the new values of the weight vectors can be calculated as:

$$\mathbf{m}_i(t+1) = \frac{\sum_{j=1}^{m} h_{ij}(t)\mathbf{s}_j(t)}{\sum_{j=1}^{m} n_{V_j} h_{ij}(t)}, \tag{6}$$

where m is the number of map units. This is the way batch algorithm has been implemented in the Toolbox.

3 How to use the Toolbox

3.1 Data requirements

The kind of data that can be handled with the Toolbox is so-called spreadsheet or table data. Each row of the table is one data sample. The items on the row are the variables, or components, of the data set, see Figure 6. The variables might be the properties of an object, or a set of measurements measured at a specific time. The important thing is that every sample has the same set of variables. Thus, each column of the table holds all values for one variable. Some of the values may be missing, but the majority should be there.

The Toolbox can handle both numeric and symbolic data, but only the former is utilized in the SOM algorithm. Note that for a variable to be "numeric", the numeric representation must be meaningful: values 1, 2 and 4 corresponding to objects A, B and C should really mean that (in terms of this variable) B is between A and C, and that the distance between B and A is smaller than the distance between B and C. Identification numbers, error codes, etc. rarely have such meaning, and they should be handled as symbolic data.

In the Toolbox, symbolic data can be inserted into string labels associated with each data sample. Consider them like post-it notes attached to each sample. You can check on them later to see what was the meaning of some specific sample, but the SOM algorithm ignores them. If you need to utilize the symbolic variables in training the SOM, you can try converting them into numerical variables using, e.g., mapping or 1-of-n coding [17].

This does not mean that the symbolic data would be useless. The distribution of labels on the map can be investigated afterwards, see function `som_autolabel`. The contributed functions include several which make use of labels in a supervised manner during training, for example `som_supervised` and `lvq3`.

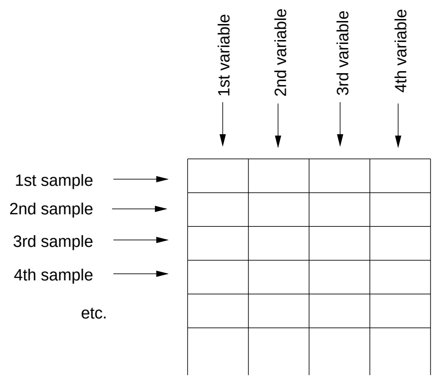


Figure 6: Table-format data: there can be any number of samples, but all samples have fixed length, and consist of the same variables.

3.2 Construction of data sets

First, the data has to be brought into Matlab. You can use normal Matlab functions like `load` and `fscanf`. In addition, the Toolbox has function `som_read_data` which can be used to read ASCII data files in SOM_PAK format (see Appendix B). Whichever the case, the following assumes that all your data is present in the Matlab workspace.

Data struct. Typically, the data is put into a so-called data struct, which is a Matlab struct defined in the Toolbox to group information related to a data set. It has fields for numerical data (`.data`), strings (`.labels`), as well as for information about data set and the individual variables (for more information, see Section 4). The format of numerical and string data is described below. If your data only consists of numerical values, you don't necessarily have to use data structs at all. Most functions accept numerical matrices (described below) as well.

Numerical data must be in a matrix of size `[dlen x dim]`, where `dlen` is the number of samples, and `dim` is the number of numerical variables. If `D` is such a matrix, each row `D(i,:)` of the matrix corresponds to one sample, and each column `D(:,i)` is the collection of all values of a single variable. If the value of some variables in some samples are missing, they can be replaced with `NaN`s. The map struct, which is used to hold all information of a SOM, has a similar field: `.codebook`. However, the `.codebook` field must not contain any `NaN`s.

A numerical matrix `D` can be converted into a data struct with `sD=som_data_struct(D)`, or inserted into an existing data struct with `sD=som_set(sD,'data',D)`.

String data are put in the field `.labels` of the data struct. This field is a cell array of strings (see commands `cell` and `cellstr`). The size of the array is `[dlen x ml]`, where `ml` is the maximum number of labels in a single sample. Note that strings of zero length (`''`) in the array are considered empty and are ignored by the functions. Each row of the array holds the labels for one sample so that labels `sD.labels(i,:)` and numerical values `sD.data(i,:)` correspond to each other. Also the map struct has `.labels` field.

If your string data is in this kind of format in variable `L`, you can insert it to an existing data struct with `sD=som_set(sD,'labels',L)` or along with the numerical data with `sD=som_data_struct(D,'labels',L)`. Alternatively, the labels can be inserted into the data struct with function `som_label` in a much more flexible manner. Of course, one could insert them by assignment:

```
% all labels
sD.labels = L;
% or single labels
sD.labels{i,1} = 'label';   % insert content 'label' into cell
sD.labels(i,1) = {'label'}; % insert a cell {'label'} into an array
```

3.3 Data preprocessing

Data preprocessing in general can be just about anything: simple transformations or normalizations performed on single variables, filters to remove uninteresting or erronous values, calculation of new variables from existing ones. In the Toolbox, only the first of these is implemented as part of the package. Specifically, the function `som_normalize` can be used to perform linear and logarithmic scalings and histogram equalizations of the numerical variables (the `.data` field).

Scaling of variables is of special importance in the Toolbox, since the SOM algorithm uses Euclidian metric to measure distances between vectors. If one variable has values in the range of $[0, ..., 1000]$ and another in the range of $[0, ..., 1]$ the former will almost completely dominate the map organization because of its greater impact on the distances measured. Typically, one would want the variables to be equally important. The default way to achieve this is to linearly scale all variables so that the variance of each is equal to one. This can be done simply with `sD = som_normalize(sD,'var')` or `D = som_normalize(D,'var')`.

One of the advantages of using data structs instead of simple data matrices is that the data structs retain information of the performed normalizations in the field `.comp_norm`. Using function `som_denormalize` one can reverse the normalization to get the values in the original scale: `sD = som_denormalize(sD)`. Also, one can repeat the exactly same normalizations to other data sets, for example `sD2 = som_normalize(sD2,sD.comp_norm)`.

All normalizations are single-variable transformations, so one can make one kind of normalization to one variable, and another type of normalization to another variable. Also, multiple normalizations one after the other can be made for each variable.

For example, consider a data set `sD` which has three numerical variables. You could do a histogram equalization to the first variable, a logarithmic scaling to the third variable, and finally a linear scaling to unit variance to all three variables:

```
sD = som_normalize(sD,'histD',1);
sD = som_normalize(sD,'log',3);
sD = som_normalize(sD,'var',1:3);
```

Your data doesn't necessarily have to be preprocessed at all before creating a SOM for it. However, in most real tasks preprocessing is important, even crucial part of the whole process [17].

3.4 Initialization and training

There are two initialization (random and linear) and two training (sequential and batch) algorithms implemented in the basic Toolbox[3]. The simplest way to initialize and train a SOM is to use function `som_make`:

```
sM = som_make(sD);
```

This function both initializes and trains the map. The training is done is two phases: rough training with large (initial) neighborhood radius and large (initial) learning rate, and fine-tuning with small radius and learning rate. By default linear initialization and batch training algorithm are used.

The `som_make` selects map size and training parameters automatically, although it has a number of arguments to give preferences of for example map size. If you want to have tighter control over the training parameters, you can use the relevant initialization and training functions directly. They are: `som_lininit, som_randinint, som_seqtrain` and `som_batchtrain`. In addition, the functions `som_topol_struct` and `som_train_struct` can be used to get default values for map topology and training parameters, respectively.

3.5 Visualization and analysis

The ordered SOM grid can be used as a convenient visualization platform for showing different features of the SOM (and the data) [22]. In the SOM Toolbox, there are a number of functions for the visualization of the SOM. Here, they are divided to three categories according to the different visual primitives:

1. cell visualizations which are based on showing the map grid as it is in the output space
2. graph visualizations which show a simple graph in the place of each map unit
3. mesh visualizations which show the map as a mesh or a scatter plot

[3] Contributed functions include some more.

3.5.1 Cell visualizations

Cell type visualizations show the SOM as it is in the output space: a regular lattice of cells the properties of which show the associated values. Note that these visualization only work for (1- or) 2-dimensional maps and that `'cell'` and `'toroid'` shapes are treated the same as `'sheet'`.

The basic tool is function `som_show`:

```
som_show(sM);
```

which by default shows first the U-matrix calculated based on all the variables and then the component planes.

- Unified distance matrix (U-matrix) visualizes distances between neighboring map units, and helps to see the cluster structure of the map: high values of the U-matrix indicate a cluster border, uniform areas of low values indicate clusters themselves [20].
- Each component plane shows the values of one variable in each map unit.

The values are shown using indexed color coding. For different colormaps, see commands `colormap`, `jet`, `hsv`, `hot`, `gray`. Also other types of planes are possible:

- An empty grid shows only the edges of the units. This may be used as a basis for labeling or other visualizations where the colored backgroud could be distracting.
- In color planes each unit is given a fixed color. This may be used to show for example clustering or other identification information for linking different visualizations [7, 10]. There are special tools in the contributed code that give this kind of color tools, for example `som_colorcode` and `som_clustercolor`.

The function `som_show` has various input arguments that may be used to control what kind of planes to show and in which order. The variable scales may be denormalized back to the original data scale (if possible) and there are various arguments that change the look of the visualizations in general, like orientation of colorbars.

A related function is `som_show_add` which sets additional information on a figure produced by `som_show`[4]: labels, hit histograms, and trajectories.

- Labeling, produced for example by function `som_autolabel`, is used to categorize the units (or some units) by giving them names.
- Hit histograms are actually markers that show the distribution of the best matching units for a given data set. Multiple histograms may be drawn and these are identified by different colors and/or markers. This makes it possible to compare different data sets by the distribution of their "hits" on a map. Hit histograms can be calculated using function `som_hits`.
- Trajectories show the best matching units for a data set that is time series (or any ordered series). It may be either a line connecting the consecutive best matching units or a "comet" trajectory where the current (first data sample) best matching unit has biggest marker and the oldest (last data sample) has smallest marker. There is a contributed function `som_trajectory` which can be used for interactive trajectory behaviour analysis and even for manually segmenting the map and the time series during the study of the trajectory.

[4] Some contributed functions, such as `som_trajectory`, also require that the Matlab figure on which they are applied has been made using `som_show`.

The som_show uses a basic routine som_cplane. This may be used to build customized cell style visualizations. Customization parameters include

- color of units
- size scaling of units
- locations of units
- prototypic shape of the unit (arbitrary polygon)
- form of units (by scaling the locations of vertices)

3.5.2 Graph visualizations

Graph visualizations are meant for drawing the SOM codebook as a set of conventional graphs. The idea is that each unit of the codebook is presented using for example pie chart, and the charts are positioned in the same way as the units are in the cell visualizations.

- Pie charts (som_pieplane) are ideal for showing proportional values. The color and size of pie slices can be altered using different arguments.
- Bar charts (som_barplane) are suitable for showing values in different categories. The color for each bar and the gap between bars can be specified. You should carefully read the different annotations on how this function scales the codebook vector values before visualization in order to avoid misinterpretations. See Section 4.2.7.
- Signal graph (som_plotplane) shows codebook vectors as simple line graphs. The color of the lines can be specified for each unit separately.

3.5.3 Mesh visualizations

The function som_grid can be used for drawing mesh (grid) style visualizations. The function is based on the idea that the visualization of a data set simply consists of a set of objects, each with a unique position, color and shape. See Matlab function scatter. In addition, connections between objects, for example neighborhood relations, can be shown using lines. With som_grid the user is able to assign arbitrary values to each of these properties. For example, x-, y-, and z-coordinates, object size and color can each stand for one variable, thus enabling the simultaneous visualization of five variables. The different options are:

- the position of an object can be 2- or 3-dimensional
- the color of an object can be freely selected from the RGB cube, although typically indexed color is used
- the shape of an object can be any of the Matlab plot markers ('.', '+', etc.)
- lines between objects can have arbitrary color, width and any of the Matlab line modes, e.g. '-'
- in addition to the objects, associated labels can be shown
- a surface between map units can be drawn in addition to the mesh grid

3.5.4 Analysis

For quantitative analysis of the SOM there are at the moment only a few tools. However, using low level functions, like `som_neighborhood`, `som_bmus` and `som_unit_dists`, it is easy to implement new analysis functions. Much research is being done in this area, and many new functions for the analysis will be added to the Toolbox in the future, for example tools for clustering and analysis of the properties of the clusters. Some such functions are already part of the contributed code, see Section 4.3.

3.6 Example

Here is a simple example of the usage of the Toolbox to make and visualize a SOM of a data set. As the example data, the well-known Iris data set is used [2]. This data set consists of four measurements from 150 Iris flowers: 50 Iris-setosa, 50 Iris-versicolor and 50 Iris-virginica. The measurements are length and width of sepal and petal leaves.

The data is in an ASCII file (in SOM_PAK format, see Appendix B). It is loaded into Matlab using `som_read_data` and normalized such that each variable has unit variance. Before normalization, an initial statistical look of the data set would be in order, for example using variable-wise histograms (see command `plotmatrix`). This information would provide an initial idea of what the data is about, and would indicate how the variables should be preprocessed. When the data set is ready, a SOM is trained. Since the data set had labels (the species identifiers), the map is also labelled using `som_autolabel`.

```
% make the data
sD = som_read_data('iris.data');
sD = som_normalize(sD,'var');
% make the SOM
sM = som_make(sD);
sM = som_autolabel(sM,sD,'vote');
```

After this, the SOM is visualized using `som_show`. The U-matrix is shown along with all four component planes. Also the labels of each map unit are shown on an empty grid using `som_show_add`. The values of components are denormalized so that the values shown on the colorbar are in the original value range. The visualizations are shown in Figure 7.

```
% basic visualization
som_show(sM,'umat','all','comp',1:4,'empty','Labels','norm','d');
som_show_add('label',sM,'subplot',6);
```

From the U-matrix it is easy to see that the top three rows of the SOM form a very clear cluster. By looking at the labels, it is immediately seen that this corresponds to the Setosa subspecies. The two other subspecies Versicolor and Virginica form the other cluster. The U-matrix shows no clear separation between them, but from the labels it seems that they correspond to two different parts of the cluster. From the component planes it can be seen that the petal length and petal width are very closely related to each other. Also some correlation exists between them and sepal length. The Setosa subspecies exhibits small petals and short but wide sepals. The separating factor between Versicolor and Virginica is that the latter has bigger leaves.

Component planes are very convenient when one has to visualize a lot of information at once. However, when only a few variables are of interest scatter plots are much more efficient. Figure 8 shows the PCA-projection of both data and the map grid.

```
% find PCA-projection of the data
[Pd,V,me] = pcaproj(sD,3);
```

```
% plot the map grid projection with
som_grid(sM,'Coord',pcaproj(sM,V,me),'marker','none',...
         'Label',sM.labels,'labelcolor','k');
% plot also the original data with color indicating subspecies
hold on, grid on
colD = [repmat([1 0 0],50,1); ...
        repmat([0 1 0],50,1); ...
        repmat([0 0 1],50,1)];
som_grid('rect',[150 1],'Line','none','Coord',Pd,'markercolor',colD)
```

Figure 9 visualizes all four variables of the SOM plus the subspecies information using three coordinates, marker size and color.

```
% denormalize the weight vectors
M = som_denormalize(sM.codebook,sM);
% make a cell array of marker types based on subspecies
colM = zeros(length(sM.codebook),3);
un = unique(sD.labels);
for i=1:3, ind = find(strcmp(sM.labels,un(i))); colM(ind,i) = 1; end
% plot the map
som_grid(sM,'Coord',M(:,2:4),'MarkerSize',(M(:,1)-4)*5,'Markercolor',colM);
% plot the data on top
hold on, grid on
D = som_denormalize(sD.data,sD);
som_grid('rect',[150 1],'Coord',D(:,2:4),'Marker','x',...
         'MarkerSize',(D(:,1)-4)*5,'Line','none','Markercolor',colD);
```

Figure 10 shows the same information using graphs.

```
% show the map grid and subspecies information
som_cplane(sM.topol.lattice,sM.topol.msize,colM);
% show the four variables with barcharts
hold on
som_barplane(sM.topol.lattice,sM.topol.msize,M,'w','unitwise');
```

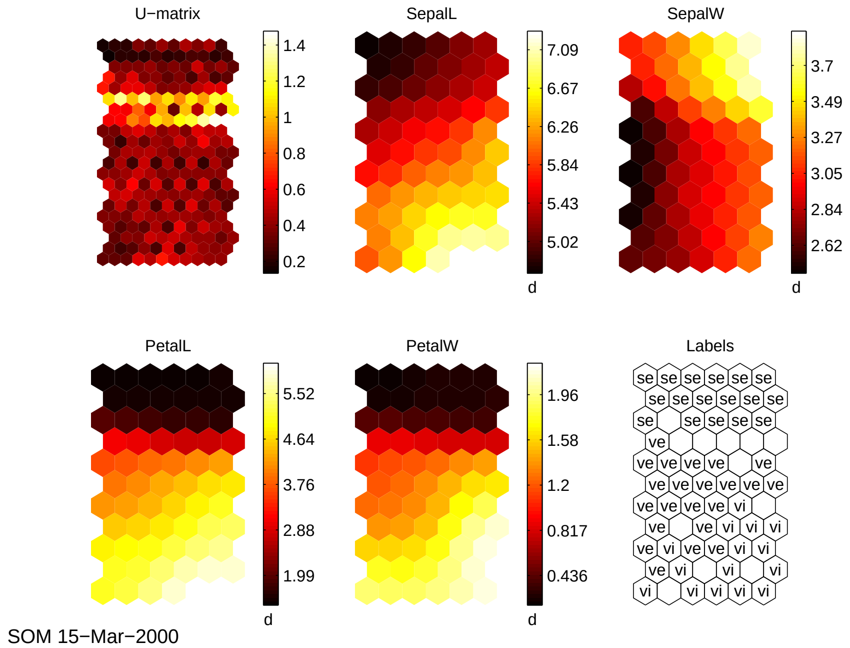


Figure 7: Visualization of the SOM of Iris data. U-matrix on top left, then component planes, and map unit labels on bottom right. The six figures are linked by position: in each figure, the hexagon in a certain position corresponds to the same map unit. In the U-matrix, additional hexagons exist between all pairs of neighboring map units. For example, the map unit in top left corner has low values for sepal length, petal length and width, and relatively high value for sepal width. The label associated with the map unit is 'se' (Setosa) and from the U-matrix it can be seen that the unit is very close to its neighbors.

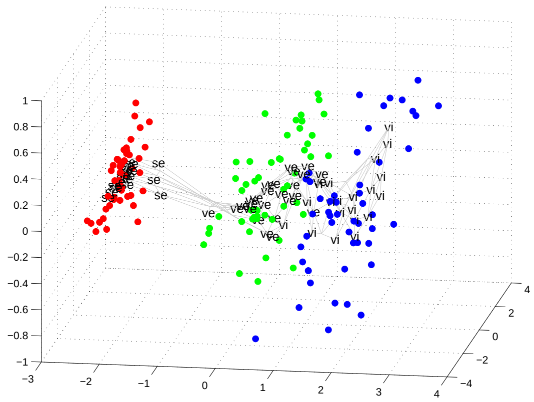


Figure 8: Projection of the Iris data set to the subspace spanned by its three eigenvectors with greatest eigenvalues. The three subspecies have been plotted using different colors. The SOM grid has been projected to the same subspace. Neighboring map units are connected with lines. Labels associated with map units are also shown.

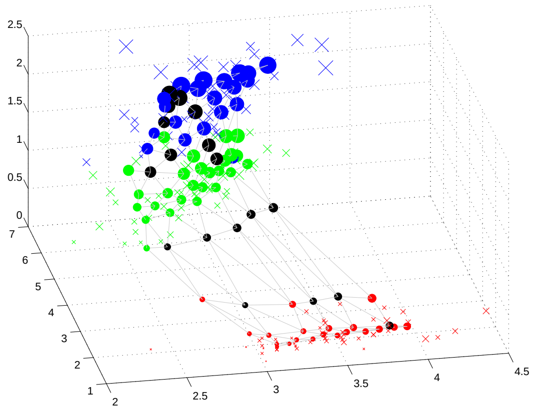


Figure 9: The four variables and the subspecies information from the SOM. Three coordinates and marker size show the four variables. Marker color gives subspecies. The data has been plotted on top with crosses.

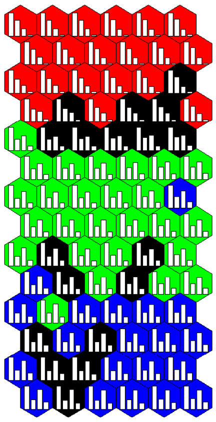


Figure 10: The four variables shown with barcharts in each map unit. In the background, color indicates the subspecies.

4 How the SOM Toolbox works

In this section, the structs and functions of the Toolbox are described to give you an idea of how the Toolbox really works. Additional details about the functions you can find from the help sections of the functions themselves.

To understand this section, you should have some familiarity with Matlab in general: what are structs, cells and different variable types. See functions `struct`, `cell` and `cellstr`. To understand the visualization functions, you have to be familiar with things like object handles, patch objects, etc. See `help graphics`. Finally, several abbreviations and standard variable names used in the functions are listed below:

variable	what it is
`sD`	data struct
`sM`	map struct
`sTo`	topology struct
`sTr`	train struct
`sN`	normalization struct
`sG`	grid struct
`sS`	some SOM Toolbox struct
`M`	map codebook `sM.codebook`
`D`	data matrix `sD.data`
`dim`	input space dimension
`dlen`	the number of data samples `[dlen,dim] = size(sD.data)`
`msize`	map size; sidelengths of the map grid `sTo.msize`
`munits`	number of map units `munits=prod(msize)`, `[munits,dim] = size(sM.codebook)`
`*`	when size is given, this means any size, e.g. `[dlen x *]`

4.1 Structs — the backbone of the Toolbox

The Toolbox uses structs to group related information, for example regarding SOMs and data sets, under a common variable. All structs have one field in common: the `.type` field which takes one of values `'som_data'`, `'som_map'`, `'som_topol'`, `'som_norm'`, `'som_train'` or `'som_grid'` indicating exactly which of the different structs it is.

For the creation of different structs there are special functions: `som_data_struct`, `som_map_struct`, `som_topol_struct`, `som_train_struct`, `som_norm_variable` and `som_grid`. Ultimately, though, all structs are created in function `som_set`, which is also in most cases used to change values of the fields of the structs. Of course, the field values could be changed by assignement, e.g. `sD.data = D`, but the use of `som_set` is encouraged since it checks the validity of the given field values[5].

[5]Except for the grid struct: for it the function `som_grid` should be used to set the values of the grid struct since the validity of the field values it checked there.

4.1.1 Data struct

Data struct groups all information related to one data set. The function `som_data_struct` is used to create a data struct from a given data matrix.

fieldname	type	size
`.type`	string ('`som_data`')	
`.name`	string	
`.data`	matrix	`[dlen x dim]`
`.labels`	cell matrix of strings	`[dlen x *]`
`.comp_names`	cellstr	`[dim x 1]`
`.comp_norm`	cell array of struct arrays	`[dim x 1]`
`.label_names`	cellstr	`[* x 1]`

`.type` field is the struct type identifier. Do not change it.

`.name` field is just a string that you can use to give an identifier to the particular data set. It is not used in the functions per se.

`.data` field is the data matrix, each row is one data vector and each column is one component. The matrix can have missing values indicated by `NaN`s.

`.labels` field contains the labels for each of the vectors. The ith row `.labels(i,:)` contains the labels for ith data vector `.data(i,:)`. Note that if some vectors have more labels than others, the others are given empty labels (`''`) to pad the `.labels` array up.

`.comp_names` field contains the names of the vector components. The default component names are `'Variable#'` where `#` is the order number of that component (ie. default value for variable number 2 is `'Variable2'`).

`.comp_norm` field contains normalization information on each component. Each cell of `.comp_norm` is a struct array of normalization structs. If no normalizations have been defined for the particular component, the cell is empty (`[]`).

`.label_names` is similar to `.comp_names` in that it holds names for different label variables. The field should only be used if all data vectors have the same number of labels, each corresponding to a certain symbolic variable.

4.1.2 Map struct

The map struct includes all information about a SOM. The function `som_map_struct` can be used to create a map struct. Also, the initialization functions `som_randinit` and `som_lininit` create a default map struct based on the given data struct: in this case the map topology is determined based on the data and the fields `.comp_names` and `.comp_norm` are copied from the training data struct into the map struct.

fieldname	type	size
`.type`	string (`'som_map'`)	
`.name`	string	
`.codebook`	matrix	`[munits x dim]`
`.topol`	topology struct	
`.labels`	cell matrix of strings	`[munits x *]`
`.neigh`	string	
`.mask`	vector	`[dim x 1]`
`.trainhist`	struct array of train structs	`[* x 1]`
`.comp_names`	cellstr	`[dim x 1]`
`.comp_norm`	cell array of struct arrays	`[dim x 1]`

`.type` field is the struct identifier. Do not change it.

`.name` field is just a string that you can use to give an identifier to the particular map. It is not used in the functions per se.

`.codebook` field is the codebook matrix. Each row `.codebook(i,:)` corresponds to the weight vector of one map unit. The order of map units in the codebook corresponds to the order of elements in a Matlab matrix (the linear index to matrix): the map is gone through column by column (as opposed to SOM_PAK where the map is gone through row by row). The codebook matrix must not contain any `NaN`s.

`.topol` field is the topology of the map: dimensions, lattice and shape of the map grid. See topology struct for more detailed information.

`.labels` field contains the labels for each of the map units. The ith row `.labels(i,:)` contains the labels for the ith map unit `.codebook(i,:)`. Note that if some vectors have more labels than others, the others are given empty labels (`''`) to pad the `.labels` array up.

`.neigh` field is the neighborhood function name: `'gaussian'`, `'cutgauss'`, `'bubble'` or `'ep'`. See Figure 4.

`.mask` field is the BMU search mask. The greater the mask value, the bigger the component's effect on map organization. Setting mask value to zero removes the effect of that component on organization.

`.trainhist` is a struct array of training structs, containing information on initialization and training, e.g., what was the name of the training data and which initialization/training algorithm was used. The first training struct contains information on initialization, the others on actual trainings. If the map has not been initialized, `.trainhist` is empty (`[]`). See training struct for more detailed information.

`.comp_names` field contains the names of the vector components. The default component names are `'Variable#'` where `#` is the order number of that component (ie. default value for variable number 2 is `'Variable2'`).

`.comp_norm` field contains normalization information on each component. Each cell of `.comp_norm` is a struct array of normalization structs. If no normalizations are performed for the particular component, the cell is empty (`[]`).

4.1.3 Topology struct

All information about the map topology is grouped into the topology struct. Topology struct itself is the `.topol` field of the map struct. The function `som_topol_struct` creates a default topology struct based on given data (ie. it sets `.msize` to a suitable value, see Section 4.2.1).

fieldname	type	size
`.type`	string (`'som_topol'`)	
`.msize`	vector	`[* x 1]`, $* \geq 2$
`.lattice`	string	
`.shape`	string	

`.type` field is the struct identifier. Do not change it.

`.msize` field gives the map dimensions. The map side lengths are in matrix-notation order, ie. first y-side, then x-side and then all other dimensions (if any). The map dimensions must be positive integers.

`.lattice` field gives the local topology type of the map: hexagonal (`'hexa'`) or rectangular (`'rect'`). See Figure 1.

`.shape` field gives the global topology type of the map: a rectangular sheet (`'sheet'`), cylinder (`'cyl'`) or toroid (`'toroid'`). See Figure 2.

4.1.4 Normalization struct

Normalization struct groups the necessary details to perform or inverse a single normalization operation for a single variable. Normalization structs are found from both map and data structs from the `.comp_norm` field. This field is a cell array, one cell for each variable. Each cell is a struct array of normalization structs. Thus, normalizations are defined for each variable separately and each variable can have as many normalization operations as desired. The function `som_norm_variable` creates and manages the normalization structs themselves, while functions `som_normalize` and `som_denormalize` manage the normalization operations of the map and data structs. These are in the `.comp_norm` field, which is a cell array of normalization struct arrays.

fieldname	type	size
`.type`	string (`'som_norm'`)	
`.method`	string	
`.params`	varies	
`.status`	string	

`.type` field is the struct identifier. Do not change it.

`.method` field specifies the normalization operation: `'var'`, `'range'`, `'log'`, `'logistic'`, `'histD'` or `'histC'`. For a more detailed description of the operations, see Section 4.2.4.

`.params` field depends on the individual method: different methods need different kinds of transformation parameters, which are preserved in this field.

`.status` field indicates the status of the normalization for the specific data set. Status can be either `'uninit'`, `'undone'` or `'done'`.

4.1.5 Training struct

Training struct gathers together information about a specific initialization or training (to be) performed on a map. The function `som_train_struct` creates a default training struct based on given map, data and other arguments.

In certain situations some of the fields of the training struct are empty (`[]`, `''` or `NaN`) either because they are obsolute (like `.alpha_ini` in case of batch algorithm) or because there's no value to give (like the fields `.data_name` and `.time` before training).

fieldname	type	size
`.type`	string (`'som_norm'`)	
`.algorithm`	string	
`.data_name`	string	
`.mask`	vector	`[dim x 1]`
`.neigh`	string	
`.radius_ini`	scalar	
`.radius_fin`	scalar	
`.alpha_ini`	scalar	
`.alpha_type`	string	
`.trainlen`	scalar	
`.time`	string	

`.type` field is the struct identifier. Do not change it.

`.algorithm` field specifies which training/initialization algorithm was used. Typically, it has one of the values `'randinit'`, `'lininit'`, `'batch'` or `'seq'`.

`.data_name` gives the name of the data used in training. This is either the `.name` field of a data struct, or the variable name of a data matrix.

`.mask` gives the BMU-search mask used in the training.

`.neigh` gives the neighborhood function used in the training.

`.radius_ini` is the initial neighborhood function radius used in the training. Similarly `.radius_fin` is the final neighborhood radius used in the training. By default radius goes linearly from `.radius_ini` to `.radius_fin`, although any other progression can be used by setting the radius values by hand.

`.alpha_ini` is the initial learning rate at the beginning of the training.

`.alpha_type` is the learning rate function type. Implemented learning rate functions are linearly decreasing (`'linear'`), reciprocally decreasing (`'inv'`), and exponentially decreasing (`'power'`). See Figure 5.

`.trainlen` is the training length in epochs.

`.time` is the date and time when the initialization or training took place.

4.1.6 Grid struct

The `som_grid` visualization function has so many arguments that it was necessary to put them in a struct of their own. The function `som_grid` is used for both constructing and visualizing these structs. See Figures 8 and 9 for examples of these visualizations.

Notice that some fields have two or even more alternate value types. Several of the fields below have values of type "RGB triple" coding colors in RGB format. The triple is a vector of size `[1 x 3]` with values between $[0,1]$ indicating the presence of red, green and blue components in the color. Alternatively, the color may typically be given as a color specification string (see `plot`).

fieldname	type	size
`.type`	string (`'som_grid'`)	
`.lattice`	string	
	(sparse) matrix	`[munits x munits]`
`.shape`	string	
`.msize`	vector	`[1 x 2]`
`.coord`	matrix	`[munits x 2]` or `[munits x 3]`
`.line`	string	
`.linecolor`	string	
	RGB triple	
	(sparse) matrix	`[munits x munits x 3]`
`.linewidth`	scalar	
	(sparse) matrix	`[munits x munits]`
`.marker`	string	
	char/cell array	`[munits x 1]`
`.markersize`	scalar	
	vector	`[munits x 1]`
`.markercolor`	string	
	RGB triple	
	RGB triples	`[munits x 3]`
`.surf`	empty	
	vector	`[munits x 1]`
	RGB triples	`[munits x 3]`
`.label`	empty	
	char array	`[munits x 1]`
	cell array	`[munits x *]`
`.labelcolor`	string	
	RGB triple	
`.labelsize`	scalar	

`.type` field is the struct identifier. Do not change it.

`.lattice` gives the local topology of the map: which map units are connected to which. If a string, it is either `'rect'` or `'hexa'` and has then same meaning as the `.lattice` field of the topology struct. In matrix form, the element (i,j) is 1 if units i and j are connected to each other, otherwise 0.

`.shape` field gives the global topology type of the map: a rectangular sheet (`'sheet'`), cylinder (`'cyl'`) or toroid (`'toroid'`). Together with `.msize` field and `.lattice` field (with value `'rect'` or `'hexa'`) it defines which map units are connected to each other, and what are the default coordinates of map units.

`.msize` field gives the map dimensions. The map side lengths are in matrix-notation order, ie. first y-side, then x-side and then all other dimensions (if any). Of course, the map dimensions must be positive integers.

`.coord` field gives coordinates for map units. The coordinates may be either 2- or 3-dimensional.

`.line` field gives the linetype used for the connecting lines, e.g. `'-'`, see `plot`. A special value is `'none'` which leaves the lines out.

`.linecolor` field gives the color of the lines. By default, the same color is used for all lines. However, if the field is a matrix or cell array of RGB triples, each connection can have its own color. The colors are positioned in the matrix as in the matrix form of `.lattice` field.

`.linewidth` field gives the width of the lines. Also these can be individually defined for each connection.

`.marker` field gives the marker type used for map units. It can be any of the standard markers (see `plot` function) or `'none'` in which case the marker is not shown. Each map unit can have an individual marker by using a cell or char array of the markers as the value for this field.

`.markersize` field gives marker size. This can be individually defined for each map unit.

`.markercolor` field gives the color of markers. Each map unit can have its own color.

`.surf` field is empty by default. If given a value, a surface is drawn in addition to the map grid. See function `surf`. If field value is a vector, indexed colors are used in the surface. If it is a matrix, the matrix rows should define RGB colors for each of the map units.

`.label` field is empty by default. If given a value, the strings in the field are plotted in addition to the map grid. The strings can be given as a char or cell array. With cell array, multiple strings per map unit can be given.

`.labelcolor` field gives the color of the labels. All labels have the same color.

`.labelsize` field gives the font size. All labels have the same size.

4.2 Functions — the meat

On the skeleton formed by the structs, the functions provide the actual meat of the Toolbox. Below, a short intro to each function is given. For more information, see the help sections and the code in the functions themselves.

In many functions optional arguments are given as argument ID, argument value pairs (`argID, value`), for example `'msize',[10 12]` to specify the map grid size. This way the optional arguments may be given in any order, and any of them can left out (to be given a default value). Of course, this makes function calls slightly longer, but usually the added flexibility more than makes up for that. Additionally, many values, such as strings `'hexa'` or `'toroid'` and (usually) structs are unambiguous: they can only mean one thing within the context of the function. In such cases, the argument ID can be omitted.

In function descriptions below, arguments in brackets `[]` are optional.

4.2.1 Creating and managing structs

The functions `som_set`, `som_info`, `som_map_struct`, `som_data_struct`, `som_topol_struct` and `som_train_struct` are used to create, set values of and display the contents of map, data, topology and train structs. The four latter functions should be used to create the structs as they provide default values for all fields of the structs. The function `som_set` should be used to set the values of their fields afterwards.

The functions to deal with normalization structs are in Section 4.2.4. The function to deal with grid struct (ie. `som_grid`) is presented in Section 4.2.7.

For compatibility with SOM Toolbox version 1, the functions `som_vs1to2` and `som_vs2to1` convert data structs from version 1 to version 2 and the other way around.

som_set(sS, [field, contents, ...]) creates the structs and, except for the grid struct, checks them to ensure that any values given to the fields of those structs are correct. You can also use it to check the validity of the structs. In general, though, structs should be created using functions like `som_data_struct`.

```
sD  = som_set('som_data');          % create data struct
sTo = som_set(sTo,'msize',[10 20]); % set value of a field
som_set(sM);                        % check validity of sM
```

som_info(sS,[level]) displays information about the Toolbox struct(s) in a user-friendly manner. The required argument is naturally the struct itself, a struct array or a cell array of structs. The level of detail displayed can be varied with the optional second argument. The number of different levels varies between 1-4.

```
som_info(sD,4); % display complete info on the data struct
```

som_map_struct(dim, [[argID,] value, ...]) creates a map struct, described in the previous section. The only required argument is the map dimension. Most of the other fields of the map struct can be given values using optional arguments of the function. If they are left unspecified, default values are used.

```
% create map struct for 10-dimensional data
sM = som_map_struct(10);
% create map struct and specify the mask to use
sM = som_map_struct(4,'mask',[1 1 0 1]);
```

som_data_struct(D, [argID, value, ...]) creates a data struct described in the previous section. The required argument is the data matrix (the `.data` field). Most of the other fields of the data struct can be given values using optional arguments of the function. If they are left unspecified, default values are used.

```
% create data struct from the data matrix
sD = som_data_struct(D);
% create data struct and specify component names to use
sD = som_data_struct(D,'comp_names',{'1','2','3'});
```

som_topol_struct([[argID,] value, ...]) is used to create a topology struct. It tries to give sensible values for the map topology (ie. map size). Unless otherwise specified, a 2-dimensional `'sheet'` map with `'hexa'` lattice is used. If data matrix `D` or the number of data samples `dlen` is given, the function tries to determine a sensible map grid size. For the total number of map units, a heuristic formula of $m = 5\sqrt{n}$ is used. The ratio of the sidelengths is based on the ratio between two biggest eigenvalues of the covariance matrix of the given data, and the actual sidelengths are then set so that their product is as close to the desired `munits` as possible. The desired number of map units can also be given as an argument.

```
% create topology struct
sTo = som_topol_struct('msize',[10 10],'rect');
% select default topology based on the data
sTo = som_topol_struct('data',D);
% as above, but a preferred number of map units is given
sTo = som_topol_struct('data',D,'munits',200);
```

som_train_struct([[argID,] value, ...]) is used to set or fill out sensible values for the fields of a training struct: the training parameters. Often the parameters depend on the properties of the map and the training data. These are given as optional arguments to the function. Also a partially filled training struct can be given, in which case only its empty fields (field value is `[]` or `''` or `NaN`) are supplimented with default values.

Parameter	condition	default value
`.algorithm`	not initialized	`som_lininit`
	initialized	`som_batchtrain`
`.alpha_type`	always	`'inv'`
`.neigh`	always	`'gaussian'`
`.alpha_ini`	finetuning phase	0.05
	otherwise	0.5
`.radius_ini`	has been initialized randomly	max(msize)/4
	has been initialized linearly	max(msize)/4
	has been trained	σ_T
`.radius_fin`	rough training phase	$\max(1, \sigma_0/4)$
	otherwise	1
`.trainlen`	rough training phase	$10m/n$
	finetuning phase	$40m/n$
	otherwise	$50m/n$

```
sTr = som_train_struct(sM,sD);  % default training parameters
sTr = som_train_struct(sTr);    % fill out empty fields of sTr
```

som_vs1to2(sS) allows conversion of Toolbox version 1 structs to Toolbox version 2. There are quite a lot of changes between the versions, especially in the map struct, and this function makes it easy to update the structs, see Appendix A.

```
sM2 = som_vs1to2(sM1); % convert version 1 map struct to version 2
```

som_vs2to1(sS) allows conversion of structs from Toolbox version 2 to Toolbox version 1. There are quite a lot of changes between the versions, especially in the map struct. This function makes it possible to use the old functions with new structs, see Appendix A. Note that part of the information is lost in the conversion. Especially, training history is lost, and the normalization is, except in the simplest cases (like all have `'range'` or `'var'` normalization) also lost.

```
sM1 = som_vs2to1(sM2); % convert version 2 map struct to version 1
```

4.2.2 Map grid and neighborhood functions

Functions `som_unit_coords`, `som_unit_dists`, `som_unit_neighs`, `som_neighborhood` and `som_connection` are very fundamental functions that define the topology of the map grid. They are primarily used by the training and visualization functions. A beginning user of the Toolbox has little use for them, but an advanced user making new analysis, visualization or training tools will probably find them useful.

In the functions below, the required argument `topol` can be either a map struct, a topology struct or a vector giving the map grid size (`sTo.msize`). In the last case, also the lattice and shape should be specified.

som_unit_coords(topol, [lattice], [shape]) calculates the map grid coordinates of the units of a SOM based on the given topology. The coordinates are such that they can be used to position map units in space. In case of `'sheet'` shape they can be (and are) used to measure interunit distances. For this reason, in case of `'hexa'` lattice, the x-coordinates of every other row are shifted by $+0.5$, and the y-coordinates are multiplied by $\sqrt{0.75}$. This is done to make distances of a unit to all its six neighbors equal. See also function `som_vis_coords`.

```
Uc = som_unit_coords(sM); % map unit coordinates on the grid
```

som_unit_dists(topol,[lattice],[shape]) calculates the distances between the units of a SOM with given topology. In case of `'sheet'` shape, the distances are simply calculated as Euclidian distance between map unit coordinates given by `som_unit_coords`. In case of `'cyl'` and `'toroid'` shapes, the distances are taken as minimum of several distances measured when the map grid is shifted in different positions, see Figure 11. This function is utilized by the training algorithms when they need to evaluate the neighborhood function.

```
Ud = som_unit_dists(sM); % distances between map units
```

som_unit_neighs(topol,[lattice],[shape]) returns a sparse connection matrix indicating for each map unit which map units are its immediate neighbors (1-neighborhood). The matrix has value 1 for the neighbors and 0 for all others (including the unit itself).

```
Ne1 = som_unit_neighs(sM); % neighboring map units
find(Ne1(1,:))             % find neighbors of map unit 1
```

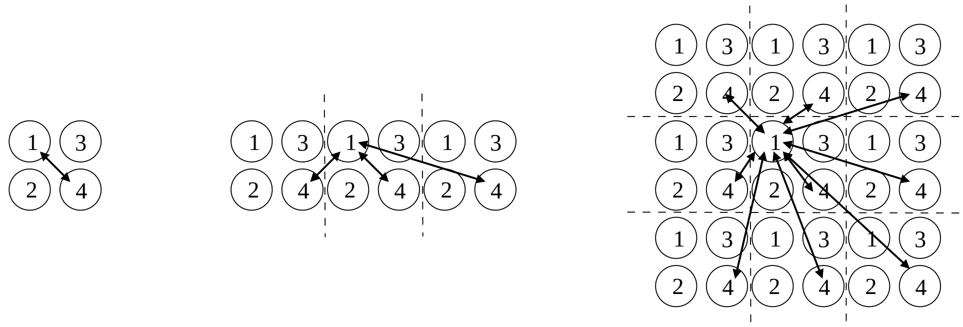


Figure 11: How distances between map units 1 and 4 are measured in case of different map shapes: 'sheet' on the left, 'cyl' in the middle and 'toroid' on the right. In the two latter cases, the coordinates of map units are first calculated as in the case of 'sheet' shape. Then the map grid is shifted to different positions and the minimum of these distances is taken.

som_neighborhood(Ne1,[n]) calculates the n-neighborhoods upto given maximum n for each map unit, see Figure 1. The first argument is the output of `som_unit_neighs` function: a connection matrix giving the units in 1-neighborhood of each unit. The neighborhoods are defined as shortest-path distances between map units. Due to recursive calculation procedure, the function is pretty slow especially with large n.

```
% neighbors upto 3-neighborhood
Ne = som_neighborhood(Ne1,3);
% find map units within 3-neighborhood of map unit 1
find(Ne(1,:)<=3)
```

som_connection(topol) returns a (sparse) matrix indicating which map units are connected to which. This function is used by the visualization functions. Does basically the same as `som_unit_neighs`, but faster. This is because the algorithm to derive the connections is based on indices rather than on calculation of distances between all map units. The connections are defined only in the upper triangular part to save some memory.

```
Ne1 = som_connection(sM);  % neighboring map units
```

som_vis_coords(lattice,msize) returns the coordinates of the map units as used in the visualizations. This function is used by the visualization functions. It is similar to `som_unit_coords` except that it is implemented more efficiently and the coordinates are slightly different (and that it only supports 'sheet' shape).

For 'rect' lattice, the coordinates of the map units and their subscript indices on the map grid are the same[6]. In 'hexa' lattice the x-coordinate of the nodes on even numbered rows (ie. y-coordinate is even) is incremented by 0.5. The hexagons are thus not unilateral but "a bit too tall". In visualization this is taken care of by changing the axis ratios to compensate for it. The motivation for using these coordinates instead of the proper ones (as calculated in `som_unit_coords`) is purely simplicity. For example, if you want to plot something on top of the third unit of the first column, the coordinates are (1,3) instead of (0,1.7321).

[6] Except that they are swapped: subscript indeces (1,2) correspond to coordinates (2,1). This swapping is simply due to the change in notation: in matrix subscript indices the vertical position is given first and then the horizontal position, and in coordinate notation the other way around.

```
Co = som_vis_coords('hexa',[10 6]); % unit coordinates in visualization
```

4.2.3 Initialization and training functions

The function som_make is the basic function to use when creating and training a SOM. It is a convenient tool that combines the tasks of creating, initializing and training a SOM in two phases — rough training and finetuning — based on given training data. Using som_make saves you a lot of typing.

Apart from collecting training parameters and managing the order of operations, som_make itself does little. For initalization and training it actually calls functions som_lininit, som_randinit, som_seqtrain, som_batchtrain and som_sompaktrain, which do the actual work. Of course, if you want to have tighter control over the training procedure, you can use these functions directly.

Each of the five initialization and training functions adds an entry to the .trainhist field of the map struct so that it is possible to check out later what has been done to the map. Saving this information is important because the default training parameters, provided by som_train_struct, depend on previous trainings (or lack of them).

som_make(D, [[**argID,**] **value, ...**]**)** creates, initilizes and trains a SOM from the given data using default parameters. Default size and shape come from som_topol_struct function and default training parameters from som_train_struct function. The optional arguments are used to modify these default values. The training is accomplished in two phases: 'rough' and 'finetune' phase.

```
% create and train SOM with default parameters
sM = som_make(sD);
% create and train SOM, the desired number of map units is about 200
sM = som_make(sD,'munits',200);
```

som_lininit(D,[[**argID,**] **value, ...**]**)** (creates and) initializes a SOM linearly. The initialization is made by first calculating the eigenvalues and eigenvectors of the given data. Then, the map weight vectors are initialized along the mdim greatest eigenvectors of the covariance matrix of the training data, where mdim is the dimension of the map grid, typically 2.

```
sM = som_lininit(sD);                % create and initialize a map
sM = som_lininit(sD,'msize',[4 9]); % the same, but the map size is given
sM = som_lininit(sD,sM);             % initialize the given map
```

som_randinit(D,[[**argID,**] **value, ...**]**)** initializes a SOM with random values. For each component x_i, the values of map weight vectors are uniformly distributed in the range of $[\min(x_i), \max(x_i)]$.

```
sM = som_randinit(sD);                % create and initialize a map
sM = som_randinit(sD,'msize',[4 9]); % the same, but the map size is given
sM = som_randinit(sD,sM);             % initialize the given map
```

som_batchtrain(sM, D, [[**argID,**] **value, ...**]**)** trains the SOM with the given data using the batch training algorithm. Using optional arguments the training parameters can be specified, otherwise default values are used (see function som_train_struct).

```
% train SOM with default parameters
sM = som_batchtrain(sM,sD);
% train SOM so that neighborhood radius goes from 4 to 1
sM = som_batchtrain(sM,sD,'radius',[4 1]);
```

som_seqtrain(sM, D, [[argID,] value, ...]) trains the SOM with the given data using the sequential training algorithm. Using optional arguments the training parameters can be specified, otherwise default values are used (see function `som_train_struct`).

```
% train SOM with default parameters
sM = som_seqtrain(sM,sD);
% train SOM so that neighborhood radius goes from 4 to 1
sM = som_seqtrain(sM,sD,'radius',[4 1]);
```

som_sompaktrain(sM, D, [[argID,] value, ...]) trains the SOM with the given data using the SOM_PAK implementation of the sequential training algorithm. This function simply writes data and initial codebook into a temporary file, calls the program `vsom` of the SOM_PAK package and reads the trained map. As with `som_batchtrain` and `som_seqtrain`, default training parameters from `som_train_struct` are used unless they are specified using the optional arguments. Notice that the program `vsom` must be in the search path in the shell. Alternatively, the path can be put to global variable `SOM_PAKDIR`. See also Section 4.3.7.

```
% train SOM with default parameters
sM = som_sompaktrain(sM,sD);
% train SOM so that neighborhood radius goes from 4 to 1
sM = som_sompaktrain(sM,sD,'radius',[4 1]);
```

4.2.4 Normalization functions

Normalizing the variables is important so that none of them has an overwhelming influence on the training result. Normalization is performed on a data set or data struct. When a SOM is trained, the normalization information is copied to the map struct. Saving this information is essential in order to be able to denormalize the values, or to repeat the normalization to other data sets.

The functions that manage normalization operations of map and data structs are `som_normalize` and `som_denormalize`. Both functions use `som_norm_variable` to do the actual normalizations. The basic user should have little need to use `som_norm_variable` directly, unless there is really a need to denormalize or normalize a single variable.

som_normalize(sS,[method],[comps]) is used to (initialize and) add, redo and apply normalizations on data/map structs and data sets. The performed normalizations are added to the `.comp_norm` fields of data/map struct. The function actually uses function `som_norm_variable` to handle the normalization operations, and only handles the data struct specific stuff itself.

```
% normalize all variables of data
sD = som_normalize(sD,'log');
% normalize only variables 1 and 3
sD = som_normalize(sD,'log',[1,3]);
% redo normalizations after possible denormalizations
sD = som_normalize(sD);
% repeat the normalization in sD on a new data matrix Dn
Dn = som_normalize(Dn,sD);
```

som_denormalize(sS,[[argID,] value, ...]) is used to undo normalizations. For a data or map struct, all normalizations in the `.comp_norm` field are undone and, thus, the values in the original data context are returned. Note that the operations are only undone — not removed — unless you give the `'remove'` argument. The function actually uses function som_norm_variable to handle the normalization operations, and only handles the data struct specific stuff itself.

```
sD = som_denormalize(sD);           % denormalize all variables
sD = som_denormalize(sD,[1,3]);     % denormalize only variables 1 and 3
D = som_denormalize(D,sD);          % denormalize data matrix D
sD = som_denormalize(sD,'remove'); % remove normalizations
```

som_norm_variable(x, method, operation) is used to initialize, apply and undo normalizations on scalar variables. It is the low-level function that upper-level functions som_normalize and som_denormalize utilize to actually do or undo the normalizations.

```
% initialize a normalization
[dummy,sN] = som_norm_variable(x1,'log','init');
% initialize and do normalization
[x1new,sN] = som_norm_variable(x1,'log','do');
% repeat the same normalization to x2
x2new = som_norm_variable(x2,sN,'do');
```

The five implemented normalization methods are:

- 'var' normalizes the variance of the variable to unity, and its mean to zero. This is a simple linear transformation: $x' = (x - \bar{x})/\sigma_x$, where $\bar{x}$ is the mean of the variable x and σ_x is its standard deviation. The transformation parameters in .params field are the mean and standard deviation of the variable.
- 'range' scales the variable values between [0,1] with a simple linear transformation: $x' = (x - \min(x))/(\max(x) - \min(x))$. The transformation parameters are the minimum value and range $(\max(x) - \min(x))$ of the variable. Note that if the transformation is applied to new data with values outside the original minima and maxima, the transformed values will be also outside the [0,1] range.
- 'log' is a logarithmic transformation. This is useful if the values of the variable are exponentially distributed with a lot of small values, and increasingly smaller number of big values. This transformation is a good way to get more resolution to the low end of that vector component. What is actually done is a non-linear transformation: $x' = \ln(x - \min(x) + 1)$, where ln is the natural logarithm. The resulting values will be non-negative. Beware, though: if the transformation is applied to a new data set with values below $\min(x) - 1$, the results will be complex numbers. The transformation parameter is the minimum value $\min(x)$, so if you know its true value beforehand, you might want to set the parameter by hand.
- 'logistic' or softmax normalization. This normalization ensures that all values, from $-\inf$ to inf are within the range [0,1]. The transformation is more or less linear in the middle range (around mean value), and has a smooth nonlinearity at both ends which ensures that all values are within the range. The data is first scaled as in variance normalization: $\hat{x} = (x - \bar{x})/\sigma_x$. Then the logistic function is applied: $x' = 1/(1 + e^{-\hat{x}})$. The transformation parameters are the mean value $\bar{x}$ and standard deviation σ_x of the original values x, just like in the 'var' normalization.
- 'histD' is discrete histogram equalization. Orders the values and replaces each value by its ordinal number. Finally, scales the values linearly so that they are between [0,1]. Useful for both discrete and continuous variables, but as the transformation parameters are all unique values of the initialization data set, it may use considerable amounts of

memory. If the variable can get more than a few values (say, 20), it might be better to use 'histC' method below. Another important note is that this method is not exactly revertible if it is applied to values which are not part of the original value set.

- 'histC' is continuous histogram equalization. Actually, this is a partially linear transformation which tries to do something like histogram equalization. The value range is divided to a number of bins such that the number of values in each bin is (almost) the same. The values are transformed linearly in each bin. For example, values in bin number 3 are scaled between [3,4[. Finally, all values are linearly scaled between [0,1]. The number of bins is the square root of the number of unique values in the initialization set, rounded up. The resulting histogram equalization is not as good as the one that 'histD' makes, but the benefit is that it is exactly revertible, even outside the original value range.

4.2.5 File read and write functions

The functions `som_read_data`, `som_write_data`, `som_read_cod` and `som_write_cod` are primarily provided for compatibility with SOM_PAK. However, especially `som_read_data` has more use, since it is relatively easy to export data from other programs, for example MS-Excel, to an ASCII format that `som_read_data` can read.

In general, if you need to save your data or maps from Matlab workspace, use Matlab's basic functions `save` and `load`.

som_read_data(filename, [dim], [missing]) reads data from an ASCII file. The file must be in SOM_PAK format (with a few exceptions, see Appendix B).

```
% read ASCII data file
sD = som_read_data('process.data');
% give also the vector dimension
sD = som_read_data('process.data',10);
% give missing value identifier ('x')
sD = som_read_data('process.data','x');
```

som_write_data(data,filename,[missing]) writes data into ASCII data file in SOM_PAK format (see Appendix B). Since the format does not support information on normalizations, that information is lost, as well as the data name. The component names are written on a comment line which begins with `'#n '` and label names on a comment line beginning with `'#l '`. Any spaces (`' '`) in the component names are replaced with underscores (`'_'`). This function is only offered for compatibility with SOM_PAK. In general, when saving data in files, use `save filename.mat sD`. This is faster and retains all information of the data struct.

```
% write ASCII data file in SOM_PAK formta
som_write_data(sD,'process.data');
% give missing value identifier ('x')
som_write_data(sD,'process.data','x');
```

som_read_cod(filename) reads a SOM codebook from an ASCII file. The file must be in SOM_PAK format (with a few exceptions, see Appendix B).

```
sD = som_read_cod('process.cod'); % read ASCII SOM (codebook) file
```

som_write_cod(sM,filename) writes a SOM codebook into an ASCII file in SOM_PAK format (see Appendix B). Since the format does not support information on normalizations, training history or mask, that information is lost, as well as the map name. Shapes other than `'sheet'` and neighborhoods other than `'bubble'` and `'gaussian'` as well as higher than 2-dimensional map grids are not supported at all. The component names are written on a comment line which begins with `'#n '`. Any spaces (`' '`) in the component names are replaced with underscores (`'_'`). This function is only offered for compatibility with SOM_PAK. In general, when saving maps in files, use `save filename.mat sM`. This is faster and retains all information of the map struct.

```
% write ASCII SOM (codebook) file in SOM_PAK format
som_write_cod(sM,'process.cod');
```

4.2.6 Label functions

Handling labels is not quite as straightforward as handling the numerical matrices is. For this reason, a special function `som_label` has been made to easily insert and remove labels from map and data structs. The function `som_autolabel` is a more sophisticated tool using which labels from one data or map struct can be transferred to another data or map struct.

som_label(sTo, mode, inds, [labels]) can be used to give and remove labels in map and data structs. Of course the same operation could be done by hand, but this function offers an alternative and hopefully slightly user-friendlier way to do it.

```
% give units 1 and 10 label 'x'
sM = som_label(sM, 'add', [1; 10], 'x');
% clear all labels from the data
sD = som_label(sD, 'clear', 'all');
% prune empty labels out of all map units
sM = som_label(sM, 'prune', 'all');
% replace existing labels in data vectors 1-10 with 'topten'
sD = som_label(sD, 'replace', [1:10]', 'topten');
% the same done with two function calls
sD = som_label(sD, 'clear', [1:10]');
sD = som_label(sD, 'add', [1:10]', 'topten');
```

som_autolabel(sTo, sFrom, [mode], [inds]) automatically labels given map or data struct based on an already labelled data or map struct. Basically, for each vector in `sFrom` the best match is found from among the vectors in `sTo`, and the labels in `sFrom` are added to the corresponding vector in `sTo`. The actual labels to add are selected based on the mode:

`'add'`	all labels from sFrom are added to sTo — even if there would be multiple instances of the same label
`'add1'`	as `'add'`, but only one instance of each label is kept
`'freq'`	only one instance of each label is kept and `'(#)'`, where `#` is the frequency of the label, is added to the end of the label. Labels are ordered according to frequency.
`'vote'`	only the label with most instances is added. In case of a draw, the first encountered label is used.

Note that these operations do not effect the old labels of `sTo`: they are left as they were.

```
% label map based on the data using 'add' mode
sM = som_autolabel(sM,sD);
```

```
% label data based on map
sD = som_autolabel(sD,sM);
% label map using the 5th column of labels in the data and 'vote' mode
sM = som_autolabel(sM,sD,'vote',[5]);
```

4.2.7 Visualization functions

The visualization functions in the Toolbox can basically be divided to three groups:

- The `som_show` family (`som_show`, `som_show_add`, `som_show_clear` and `som_recolorbar` and some functions in the contributed code) which are high-level tools for making cell-style visualizations. The `som_show` function sets several tags in the figures, and the other functions utilize heavily these tags. These functions are very SOM-specific.
- Generic functions: `som_cplane`, `som_barplane`, `som_pieplane`,`som_plotplane` and `som_grid`. These functions can be easily used for purposes other than just visualizing a SOM. For example, the function `som_pieplane` can be used for visualizing n different sized pie charts in arbitrary locations with any data.
- Low-level "internal" subfunctions used by the functions above.

All functions use the same coordinate system for the positions of map units (unless the coordinates are explicitly set as is possible in the case of generic functions). The coordinate system insures that if two visualizations of the map of same size are drawn on the same figure they will match.

```
som_barplane('hexa',[10 5],rand(50,4));
hold on;
som_cplane('hexa',[10 5],'none');
som_grid('hexa',[10 5])
```

For example, `som_grid` can be used for plotting labels on top a visualization made using `som_cplane`. The coordinates are specified by function `som_vis_coords`. Most visualizations have been implemented so that after processing the input — data, coordinates, colors and other parameters — the visualization can be executed with a single `patch` command. This makes drawing for example multiple pie charts, bar charts or signal charts much faster since the overhead created by loops or axis generation is avoided.

som_show(sM, [[argID,] value, ...]) is the basic visualization function of the Toolbox. It is used to show component planes, U-matrices as well as empty planes and fixed-color planes. It has several auxiliary functions (e.g. `som_show_add`, `som_show_clear` and `som_recolorbar` listed below) which utilize the tags in the figure set by the `som_show` function. The function has a number of arguments which can be used to control type and order of the different visualizations and additional information printed on the figure. For an example see Figure 7.

```
% basic usage
som_show(sM);
% different planes
som_show(sM,'comp',[1 3 2 6],'umat',{[1 2],'1,2 only'},...
         'empty','Empty plane','color',rand(prod(sM.topol.msize),3));
% additional options
som_show(sM,'size',0.8,'bar','horiz','edge','off','footnote','');
```

som_show_add(mode, D, [[argID,] value, ...]) adds hits, labels and trajectories on a figure created with `som_show`. The first argument defines what kind of markers to add and the second arguments gives the markers or their places. The optional arguments are used to further modify the markers, for example their color, size and type. There are four basic modes: `'label'` to display labels, `'hit'` to display hit histograms, `'traj'` to display line trajectories and `'comet'` to display comet-like trajectories.

```
% basic usage
som_show_add('label',sM);
som_show_add('hit',som_hits(sM,sD));
som_show_add('traj',som_bmus(sM,sD));
som_show_add('comet',som_bmus(sM,sD));
% additional options
som_show_add('comet',som_bmus(sM,sD),'markersize', [15 10],...
             'markercolor','w','edgecolor','k','subplot',4);
```

som_show_clear([type],[p]) clears hits, labels or trajectories created by `som_show_add`. The type of markers to remove and the subplots from which to remove them can be specified with the optional arguments.

```
som_show_clear; % clear all markers
```

som_recolorbar([p], [ticks], [scaling], [labels]) refreshes the colorbars in the figure created by `som_show`. Refreshing is necessary if you have changed the colormap (see `colormap` command). Using optional arguments, the properties of the colormaps — tick placements, tick labels and whether normalized or denormalized values are shown — can be controlled.

```
% refresh colorbar
colormap(jet);
som_recolorbar;
% set colorbar ticks
colormap(jet(3));
som_recolorbar(2,'border','denormalized',{{'min' 'med1' 'med2' 'max'}});
```

som_cplane(lattice, msize, color, [s], [pos]) is the basic building block of any sheet shaped component plane or U-matrix style visuzalization. It draws the unit edges and sets colors for units. The unit markers may be located according to the standard `'hexa'` or `'rect'` lattice topologies. The units may also be given an arbitrary form instead of hexagon or rectagle and/or arbitrary positions (except in U-matrix visualization). Unit coloring may be done using fixed RGB colors or indexed colors or the units may be drawn as transparent. Se Figures 7 and 10.

```
% basic component plane
som_cplane(sM.topol.lattice,sM.topol.msize,sM.codebook(:,1));
% U-matrix
u=som_umat(sM);
h=som_cplane([sM.topol.lattice 'U'], sM.topol.msize, u(:));
% turn off edgecolor
set(h,'edgecolor','none');
% show value with size instead of the color
m = sM.codebook(:,1); m = m-min(m); m = m/max(m);
som_cplane(sM.topol.lattice,sM.topol.msize,'w',sqrt(m));
```

som_grid(sS,[[argID,] value, ...]) is the basic function for showing mesh style visualizations. The units may be drawn using different markers and colors, in different sizes and in different locations in 2D or 3D. However, the topological neighborhood is limited to be 2-dimensional. The connections between these units may be drawn using lines having different thicknesses and colors. By default, the connections are defined in function `som_connection`, but they may also be defined by the user. Labels may be plotted on the units. It is possible also to draw a surface between the units. The surface coloring is either indexed (one value per unit) or fixed RGB (a `[1 x 3]` RGB triple per unit). This function is very versatile. See Figures 8 and 9 for examples of these visualizations.

```
% plot SOM map grid
som_grid(sM);
% plot SOM map grid in alternate coordinates
som_grid(sM,'coord',sM.codebook(:,[1 2 3]));
% plot one component plane as a surface
S = som_grid(sM,'surf',sM.codebook(:,4),'marker','none','line','none');
% change the coordinates
S = som_grid(S,'coord',sM.codebook(:,[1 2 3]));
% plot labels on top of som_cplane visualization
som_cplane(sM.topol.lattice,sM.topol.msize,'none'); hold on
som_grid(sM,'Label',sM.labels,'Labelcolor','b');
```

som_barplane(lattice, msize, data, [color], [scaling], [gap], [pos]) can be used to show a bar chart in the place of each map unit. It can be used, for example, to show the prototype vectors. The appearence of the bar chart can be modified: the color of the bar for each variable, the gap between bars and the scaling of maximum and minumum values may be set. There are three different ways in which the bars in each unit can be scaled:

- `'none'`: the bars representing variables in each unit are drawn using the data without any scaling: the upper edge of the unit (in `'rect'` lattice) corresponds to value +0.625 and lower edge to value -0.625. Values that exceed these limits will cause the bars to range outside the unit limits.
- `'varwise'`: each variable is first scaled so that its maximum absolute value is one. The bars in each unit show now the value of each variable in that unit in relation to the overall maximum value of that variable.
- `'unitwise'`: variable values are scaled in each unit separately so that the heights of the bars show their magnitude relative to each other. This is sensible if the variables are directly comparable with each other.

The base line (zero level) in each unit is set according to the data (codebook) values. If the variable has a negative value the bar is drawn downward from this reference level and for positive values the bars will be drawn upward, respectively. The location of the zero level depends on the varible values in the unit and on the scaling mode. For example, if all values are negative, the line will be on the upper edge of the unit and if all values are positive it will be on the lower edge.

```
% basic usage
som_barplane(sM.topol.lattice,sM.topol.msize,sM.codebook);
% more options
som_barplane('hexa',[5 5],randn(25,10),jet(10),'unitwise',0);
% change the coordinates of map units
som_barplane('none',rand(25,2)*10,randn(25,10),jet(10),'unitwise',0);
```

som_pieplane(lattice, msize, data, [color], [s], [pos]) can be used to show a pie chart in the place of each map unit. This is very similar to `som_barplane` except that the optional arguments are slightly different. Argument `s` can be used to give individual size to each pie chart. An additional constraint is that negative values are not allowed in the data.

```
% basic usage
som_pieplane(sM.topol.lattice,sM.topol.msize,sM.codebook);
% more options
som_pieplane('hexa',[5 5],rand(25,5),jet(5),rand(25,1));
% change the coordinates of map units
som_pieplane('rect',rand(25,2)*10,rand(25,5));
```

som_plotplane(lattice, msize, data, [color], [scaling], [pos]) can be used to show a plot chart in the place of each map unit. This is also very similar to `som_barplane`.

```
% basic usage
som_plotplane(sM.topol.lattice,sM.topol.msize,sM.codebook);
% more options
som_plotplane('hexa',[5 5],rand(25,5),jet(25));
% worms on vacation!
h=som_plotplane('rect',rand(40,2)*10,randn(40,6),[0.8 0.4 0.3]);
set(h,'Linewidth',5)
```

In addition, there are some very low-level functions used by the visualization functions. To indicate this, the command names start with `vis_` instead of the usual `som_`. There is no reason why the user could not use these if he/she finds them useful, but they are often doing something very specific and their documentation is not as detailed as that of the user level functions. The functions are:

- **vis_patch**: gives the patch vertex coordinates for creating rectangles and hexagons in the visualizations.
- **vis_som_show_data**: is an integral part of all functions that are related to `som_show`. It reads the information stored to the UserData field of the figure, and checks if the figure really is a valid `som_show` figure. It also returns the handles to the subplots in the original order.
- **vis_valuetype**: This function makes type checks in visualization routines. It serves two purposes. The first purpose is to make validity checks for error handling, so that more informative error messages can be returned to the user than the reports of the syntax failures somewhere in the code during running. The second purpose is to help programming the overloaded functions in visualizations (different operations based on the input argument type).
- **vis_PlaneAxisProperties**: Subfunction for visualizations (`som_cplane`, `som_barplane`, `som_plotplane`,`som_pieplane` and `som_grid`). Sets the axis scalings and other axis properties that are common for these functions.
- **vis_footnote**: sets the movable text to the `som_show` figure.
- **vis_footnoteButtonDownFcn**: a subfunction for `vis_footnote`.

4.2.8 Miscalleous functions

som_bmus(sMap, sData, [which], [mask]) returns the indeces and corresponding quantization errors (Euclidian distances) of the vectors in the first argument, that were closest to (best matched) the vectors in the second argument. Also other than best match, for example second and third best match, can be returned. Note that the mask, if given or present in either of the given structs, is used to weight the distances, and thus the quantization errors are actually weighted distances. This function is typically used to find BMUs from a map for the vectors in a given data set.

```
% find the BMU in sM for each vector in sD
bmus = som_bmus(sM,sD);
% find the second- and third-best matching units in sM for D,
% as well as the corresponding quantization errors
[bmus,qerrs] = som_bmus(sM,D,[2,3]);
% find BMUs from sM for vectors in D, using the given mask
bmus = som_bmus(sM,D,1,[1 1 0 0 1]);
```

som_hits(sM, sD, [mode]) calculates the number of “hits” in each map unit, ie. a data histogram. The function uses `som_bmus` to find the BMUs, and calculates the number of times each map unit was the BMU. Also fuzzy responses can be calculated.

```
% data histogram
hits = som_hits(sM,D);
% fuzzy response of data sample
resp = som_hits(sM,D(1,:),'fuzzy');
```

som_quality(sM, D) provides two measures for the quality of the map. The measures are data-dependent: they measure the map in terms of the given data (typically the training data). The function returns two measures: average quantization error and topographic error.

- Average quantization error is simply the average distance (weighted with the mask) from each data vector to its BMU.
- Topographic error gives the percentage of data vectors for which the BMU and the second-BMU are not neighboring map units [11].

```
[qe,te] = som_quality(sM,sD); % error measures for sM, given sD
```

som_umat(sMap, [[argID,] value, ...]) returns the unified distance matrix (U-matrix) of the given SOM. The U-matrix is an important visualization method [20]. U-matrix gives the distances between neighboring map units. For example, in the case of `[1 x 5]` -sized map $[\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, \mathbf{m}_4, \mathbf{m}_5]$, the U-matrix is a `[1 x 9]` vector $[u_1, u_{12}, u_2, u_{23}, u_3, u_{34}, u_4, u_{45}, u_5]$, where $u_{ij} = ||\mathbf{m}_i - \mathbf{m}_j||$ is the distance between two neighboring map units and u_i is a unit-specific value, for example mean distance from that unit to all neighboring units: $u_3 = (u_{23} + u_{34})/2$. The SOM grid can be at most 2-dimensional.

```
% typically, the function is used simply like this
U = som_umat(sM);
% below, the unit-specific value is 'max', and a mask is specified
U = som_umat(sM,'max','mask',[1 1 0 1]);
% to get unit-specific values only
Um = U(1:2:end,1:2:end);
```

4.3 Contributed functions

The contributed functions differ from the basic SOM Toolbox functions in that they are not copyrighted by the SOM Toolbox team, and are typically less well documented than the basic package. Although the contributed functions are not essential, many of them are very handy in practice.

The list below is by no means conclusive as new functions will be added from time to time. You are also welcome to make contributions to the package. Just send your functions and/or questions about this to the email address given at the beginning of this report, or visit our website and read the guidelines there.

4.3.1 Demos

It is a good idea to start using the Toolbox by going through these demos. They introduce the SOM algorithm and the SOM Toolbox to you, and you can refer to them later to see how some utterly cool visualization was done.

- **som_demo1**: SOM Toolbox demo 1 — basic properties
- **som_demo2**: SOM Toolbox demo 2 — basic usage
- **som_demo3**: SOM Toolbox demo 3 — visualization
- **som_demo4**: SOM Toolbox demo 4 — data analysis

4.3.2 Clustering functions

The SOM algorithm is only one way to segment/cluster the data. It has many variants, and there are a lot of completely different clustering algorithms. Below are some implemented algorithms. The neat thing about Matlab is that the functions are easy to modify.

- **som_prototrain**: a simple version of sequential training algorithm which is easy to modify
- **kmeans**: k-means algorithm [6]. Note that if you set the final neighborhood radius to zero during training a SOM, the final energy function equals that of the k-means.
- **kmeans_clusters**: try and evaluate several k-means partitionings
- **neural_gas**: neural gas vector quantization algorithm [16]

4.3.3 Modeling functions

Although the SOM is not specifically meant for modeling, it is very easy to build local and nearest-neighbor models on top of the SOM. Below are some functions which can be used for modeling.

- **lvq1**: LVQ1 classification algorithm [12]
- **lvq3**: LVQ3 classification algorithm [12]
- **knn**: k-NN classification algorithm, see e.g. [3]
- **som_supervised**: supervised SOM algorithm [12]

- **som_estimate_gmm**: create Gaussian mixture model on top of SOM [1]
- **som_probability_gmm**: evaluate the Gaussian mixture model created with `som_estimate_gmm`

4.3.4 Projection functions

Vector projection algorithms find low-dimensional coordinates for a set of high-dimensional vectors such that the "shape" of the data cloud is preserved as well as possible. Vector projection is typically used for dimensionality reduction and to make visualization of originally high-dimensional data sets possible.

- **pcaproj**:principal component projection algorithm
- **cca**: Curvilinear Component Analysis (CCA) projection algorithm [5]
- **sammon**: Sammon's mapping projection algorithm [19]

4.3.5 Auxiliary visualization functions

The functions below are auxiliary visualization tools. Rather than make actual visualizations, most of them produce powerful visualization elements, like color codings, to be used with the actual visualization functions. There are also some GUIs which help in utilizing the standard visualizations.

- **som_colorcode**: create color coding for map/2D data
- **som_normcolor**: calculates fixed RGB colors that are similar to indexed colors with the specified colormap. The function is offered because some visualization functions (as `som_grid`) can't use colormap based indexed colors if the underlying Matlab function (e.g. `plot`) do not use them.
- **som_clustercolor**: color coding which depends on clustering structure
- **som_select**: GUI for manual selection of map units
- **som_trajectory**: launches a GUI for presenting comet-trajectories
- **vis_trajgui**: the actual GUI started by `som_trajectory`

4.3.6 Graphical user interface functions

While Matlab offers possibilities to create quite powerful graphical user interfaces (GUIs), only a couple have yet been implemented for the Toolbox. This is primarily because the authors themselves use almost exclusively the commands themselves, as this is quicker and more flexible.

- **som_gui**: SOM initialization and training GUI
- **preprocess**: data set preprocessing tool GUI

4.3.7 SOM_PAK interface

These tools can be used to call SOM_PAK programs from the Matlab. They essentially gather the parameters, convert them, save the data and map, run the programs in shell, and read the answers back to Matlab. To work, the SOM_PAK functions need to be on the search path in the shell. Alternatively, the path can be put to global variable `SOM_PAKDIR`. See also function `som_sompaktrain`.

- **sompak_gui**: GUI for using SOM_PAK from Matlab
- **sompak_init, sompak_init_gui**: call SOM_PAK's initialization programs from Matlab (command-line and GUI)
- **sompak_sammon, sompak_sammon_gui**: call SOM_PAK's `sammon` program from Matlab (command-line and GUI)
- **sompak_train, sompak_train_gui**: call SOM_PAK's training program from Matlab (command-line and GUI)
- **sompak_rb_control**: an auxiliary function for `sompak_*_gui` functions

5 Performance

5.1 Computational complexity

As C-code, one epoch of sequential training algorithm can be implemented as:

```
for (j=0; j<n; j++) {           /* go through the data */
  bmu=-1; min=1000000;          /* or some other big enough number */
  for (i=0; i<m; i++) {         /* find the BMU */
    dist=0;
    for (k=0; k<d; k++) { diff = X[j][k] - M[i][k]; dist += diff*diff; }
    if (dist<min) { min=dist; bmu=i; }
  }
  for (i=0; i<m; i++) {         /* update */
    h = alpha*exp(U(bmu,i)/r);
    for (k=0; k<d; k++) M[i][k] -= h*(M[i][k] - X[j][k]);
  }
}
```

where X[j][k] is the kth component of jth data sample, M[i][k] is the kth component of map unit i and U is a table of squared map grid distances between map units calculated beforehand. Gaussian neighborhood function is assumed here and the radius r corresponds to $-2r(t)^2$ in Figure 4. Correspondingly, one epoch of batch training algorithm is:

```
for (i=0; i<m; i++) { vn[i] = 0; for (k=0; k<d; k++) S[i][k] = 0; } /* initialize */
for (j=0; j<n; j++) {           /* go through the data */
  bmu=-1; min=1000000;          /* or some other big enough number */
  for (i=0; i<m; i++) {         /* find the BMU */
    dist=0;
    for (k=0; k<d; k++) { diff = X[j][k] - M[i][k]; dist += diff*diff; }
    if (dist<min) { min=dist; bmu=i; vn[bmu]++; }
  }
  for (k=0; k<d; k++) S[bmu][k] += X[j][k]; /* Voronoi region sum */
}
for (i=0; i<m; i++) for (k=0; k<d; k++) M[i][k] = 0; /* initialize */
for (i1=0; i1<m; i1++) {        /* update */
  htot = 0;
  for (i2=0; i2<m; i2++) {
    h = exp(U[i1][i2]/r);
    for (k=0; k<d; k++) M[i1][k] += h*S[i2][k];
    htot += h*vn[i2];
  }
  for (k=0; k<d; k++) M[i1][k] /= htot;
}
```

There are $6nmd+2nm$ floating point operations (additions, substractions, multiplications, divisions or exponents) in the sequential algorithm and $3nmd + (2d+5)m^2 + (n+m)d$ operations in the batch training algorithm. Thus, the computational complexity of one epoch of sequential training is about $\mathcal{O}(nmd)$ and, if $n \gg m$, the computational complexity of batch training is half of the sequential one.

In the Toolbox, things are a bit different from the above C-code. For example, as discussed in Section 2 the distance metric is slightly different from Euclidian, see Eq. 2. This increases computational complexity somewhat, but the algorithm still remains essentially $\mathcal{O}(nmd)$ in complexity.

If default parameter values listed in Section 4.2.1 are used, one can also calculate the complexity of the whole training process in the Toolbox. The number of map units m is proportional to $\sqrt{n}$ and the number of epochs is proportional to m/n. Thus the total complexity is $\mathcal{O}(nd)$ making SOM applicable also to relatively large data sets, although training huge maps is time-consuming. Of course, in some cases the number of map units needs to be selected differently, e.g. $m = 0.1n$ in which case the complexity is $\mathcal{O}(n^2 d)$.

There are also some faster variants of the SOM [13, 9]. These are basically based on speeding up the winner search from $\mathcal{O}(md)$ to $\mathcal{O}(log(m)d)$ by investigating only a small number of map units.

The memory consumption depends on whether the interunit distances (the matrix `U` above) in the output space are calculated beforehand or not. If they are, the memory consumption scales quadratively with the number of map units. If not, the consumption scales linearly, but $6m$ additional floating point operations per training step are needed (assuming a 2-dimensional map grid is used). Note that the memory requirements due to the training data have been ignored. While just one training sample is needed at a time, in practice as much of the data as possible is usually kept in the main memory to reduce the overhead due to disc-access (or some other mass storage device) time.

5.2 Test set-up

While insightful, these estimates give no indication of the actual training times involved. To get an idea of this some performance tests were made. The purpose was only to evaluate the computational load of the algorithms. No attempt was made to compare the quality of the resulting mappings, primarily because there is no uniformly recognized "correct" method to evaluate it.

The tests were run in a machine with 3 GBs of memory and 16 250 MHz R10000 64-bit CPUs (one of which was used by the test process) running IRIX 6.5 operating system. The Matlab version was 5.3. The different parameters of the test (data size, training length, etc.) are listed in Table 1. The test parameters included different data set and map sizes and three training functions: `som_batchtrain`, `som_seqtrain` and `vsom`, the last of which was a C-program of the SOM_PAK package used through `som_sompaktrain`. The training length, 10 epochs, is fairly standard, although with very large data sets it is perhaps excessive.

The computing times measured for `vsom` do not include the time needed for writing and reading the files from Matlab. This took between 0.2 and 50 seconds, depending primarily on the size of the data set. Especially with large maps, this is irrelevantly small time when compared with overall training time. However, the measured times do contain the time used by `vsom` itself for file I/O. Thus, there is some small overhead in the computing times for `vsom` with respect to computing times of `som_batchtrain` and `som_seqtrain`.

5.3 Results

Figure 12 shows the test results. Some typical results are listed in Table 2. For a data of size `[3000 x 30]` and 300 map units, the training times for `'gaussian'` neighborhood were 8, 77 and 43 seconds, for `som_batchtrain`, `som_seqtrain` and `vsom`, respectively. For the largest investigated case with data of size `[30000 x 100]` and 1000 map units the times were 8 minutes, 2.2 hours and 47 minutes, respectively.

The `som_batchtrain` is almost always considerably faster (and `som_seqtrain` slower) than the others: upto 20 times faster than `som_seqtrain` and upto 11 times faster than `vsom`, with mean values being 11 and 4. The only case where this is not so is when the number of map units greatly exceeds the number of data samples $n < m$ (top right corner of Figure 12) — a very unlikely case in practice.

Figure 13 shows some typical computing times as a function of the number of data samples, map units and data dimension. Computing time scales almost linearly with respect to dimension and number of data samples (on this parameter range), but grows faster with the number of map units.

5.4 Additional tests

Neighborhood function. Since `'gaussian'` neighborhood function is computationally quite heavy, some tests were performed with the `'bubble'` neighborhood function. In Matlab, the change did not produce any improvements. However, in `vsom` the effect was significant: changing to `'bubble'` neighborhood dropped the training time by about 30% on the average.

Precompiled code. One of the weak points of Matlab is that loops are relatively slow when compared with precompiled code (like C-programs). Matlab has a compiler (`mcc` by MathWorks) which we used to precompile the training functions to so-called mex-files (short for Matlab executable). However, it appears that this is really beneficial only if there are a lot of loops in the function: we observed some benefits with respect to `som_seqtrain` (in the order of 20%; the longer the training, the greater the benefit) but none with respect to `som_batchtrain`. The C-code that the compiler produces does not seem to be very optimized, though. We believe that the computing times could be dropped more if the algorithm were handcoded.

Workstation. Some tests were also performed in a workstation with a single 350 MHz Pentium II CPU and 256 MBs of memory. The tests were performed both in Linux and Windows NT operating systems. The relative computation times for IRIX, Linux and Windows respectively were

- 1:5:2.5 for `som_batchtrain`
- 1:3.3:2.7 for `som_seqtrain`
- 1:1.7:4 for `vsom`

In effect, IRIX was fastest, and Windows was faster than Linux, except in the case of `vsom`. Of course, these results reflect differences between underlying hardware, operating system and

Table 1: Performance test parameters.

parameter	different values used in the test
data dimension	10, 30, 50, 100
data length	300, 1000, 3000, 10000, 30000
number of map units	30, 100, 300, 1000
training function	`som_batchtrain`
	`som_seqtrain`
	`vsom`
neighborhood function	`'gaussian'`
training length	10 epochs

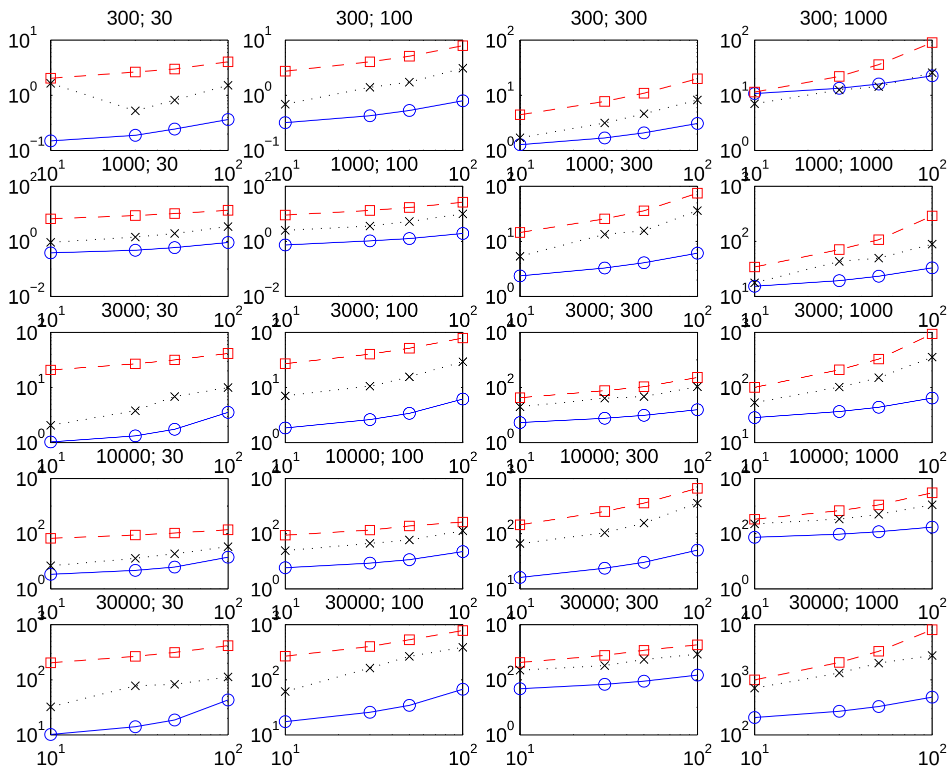


Figure 12: Computing time (in seconds) as a function of data dimension. Both axis are logarithmic. The subplot titles $n; m$ show the number of data samples (n) and map units (m). Times for batch training algorithm are shown with circles and solid line, times for sequential training with squares and dashed line, and times for `vsom` with crosses and dotted line.

optimization of the Matlab itself in different platforms and it is impossible to say how much each factor affected the results. The lower amount of memory in the workstation certainly had an effect with large data and map sizes. Since the performace of `vsom` in Linux was not effected by this, with large maps `vsom` was approximately as fast or even faster than `som_batchtrain`. The slowness of `vsom` in Windows may own something to the fact that the Windows version of `vsom` was compiled back in 1996, while the Unix versions were compiled just recently.

Version 1. To see how much progress has been done in SOM Toolbox development, some test runs were also made with the corresponding functions in version 1. The batch algorithm in version 2 is 70% faster and sequential algorithm about 800% faster than in version 1. The bigger the map and data sizes, the bigger the difference. In addition to the increase in speed, the memory consumption of `som_seqtrain` and especially `som_batchtrain` is only fractions of what it was before.

Table 2: Some computing times (10 epochs of training).

data	map units	`som_batchtrain`	`som_seqtrain`	`vsom`
`[300 x 30]`	100	0.4 s	4.0 s	1.4 s
`[1000 x 30]`	100	1.0 s	13 s	3.6 s
`[3000 x 30]`	100	2.6 s	40 s	11 s
`[10000 x 30]`	100	8.6 s	2.3 min	44 s
`[3000 x 10]`	300	5.4 s	43 s	20 s
`[3000 x 30]`	300	7.7 s	1.3 min	41 s
`[3000 x 50]`	300	9.8 s	1.8 min	47 s
`[3000 x 100]`	300	16 s	3.8 min	1.8 min
`[30000 x 30]`	30	14 s	4.4 min	1.3 min
`[30000 x 30]`	100	26 s	6.7 min	2.7 min
`[30000 x 30]`	300	1.1 min	13 min	5.4 min
`[30000 x 30]`	1000	4.5 min	34 min	22 min

5.5 Memory

The major deficiency of the Toolbox, and especially of batch training algorithm, is the expenditure of memory. A rough estimate of the amount of memory used by `som_batchtrain` is given by: $8 \times (5md + 4nd + 3m^2)$ bytes, where m is the number of map units, n is the number of data samples and d is the input space dimension. Especially the last term limits the usability of the Toolbox considerably. Consider a relatively small data set `[3000 x 10]` and 300 map units. The amount of memory required is still moderate, in the order of 3 MBs. However, increasing the map size to 3000 map units, the memory requirement is almost 220 MBs, 99% of which comes from the last term of the equation. The sequential algorithm is less extreme requiring only one half or one third of this.

5.6 Applicability

Applicability of the Toolbox in a given data mining task depends strongly on the characteristics of the problem. In general the Toolbox is intended for analysis of single table data with numerical (continuous or discrete) variables.

The maximum number of samples that the Toolbox can comfortably handle depends on the available hardware, but for example with Pentium II processor and about 128 MBs of memory handling data upto size 10000 samples is easy and even fast (batch training takes less than 1 minute). After 30000 samples, memory starts to become a problem. In our own work, sample sizes upto half a million samples have been handled. Here, it is assumed that the number of variables is moderate, say less than a few hundred.

A more strict limitation is the number of map units. The computational load grows perhaps quadratively with the number of map units (see Figure 13), as do memory requirements. As a general guideline, the Toolbox can comfortably handle maps with less than 1000 units.

In case memory becomes a real problem, it might be advisable to use SOM_PAK. It is slower than batch training (but faster than sequential training), but it uses memory much more conservatively, and can use buffered data.

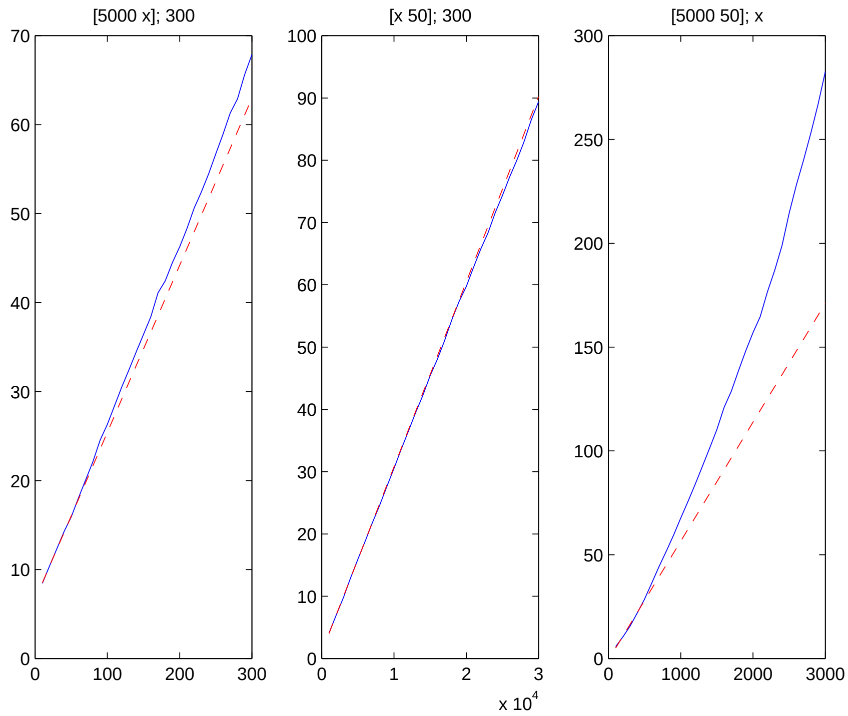


Figure 13: Computing time of `som_batchtrain` (in seconds) as a function of data dimension (on the left, with 5000 data samples and 300 map units), the number of data samples (in the center, with 300 map units, dimension 50), number of map units (in the center, with 5000 data samples, dimension 50) and the number of data samples (on the right, with 300 map units, dimension 50). The dashed line is a line fitted to the data using five first samples.

6 Conclusion

In this report, the SOM Toolbox version 2 for Matlab 5 has been introduced. Matlab as a computing environment is extremely versatile but for this reason also requires high degree of user interaction.

The SOM is an excellent tool in the visualization of high dimensional data. As such it is most suitable for data understanding phase of the knowledge discovery process, although it can be used for data preparation, modeling and classification as well. The data model is a single table with numerical variables. The Toolbox is easily applicable to small data sets (under 10000 records) but can also be applied in case of medium sized data sets (upto 1000000 records). An important limiting factor is, however, the map size. The Toolbox is mainly suitable for training maps with 1000 map units or less. If this kind of parameter values are used, especially the batch training algorithm is very efficient, being considerably faster than `vsom`, a commonly used implementation of sequential SOM algorithm in C-code.

In future work, our research will concentrate on the quantitative analysis of SOM mappings, especially analysis of clusters and their properties. New functions and graphical user interface tools will be added to the Toolbox to increase its usefulness in data mining. Also outside contributions to the Toolbox are welcome. It is our hope that the SOM Toolbox promotes the utilization of SOM algorithm — in research as well as in industry — by making its best features more readily accessible.

Acknowledgments

This work has been mostly carried out in "Adaptive and Intelligent Systems Applications" technology program of Technology Development Center of Finland, and the EU financed Brite/Euram project "Application of Neural Network Based Models for Optimization of the Rolling Process" (NEUROLL). The cooperation and financing of Jaakko Pöyry Consulting, UPM-Kymmene, Rautaruukki Raahe Steel, Laboratory of Metallurgy at Helsinki University of Technology and the Technology Development Center of Finland (TEKES) are gratefully acknowledged.

We would also like to thank Mr. Kimmo Kiviluoto, Mr. Jukka Parviainen and Mr. Mika Pollari whose handwork is also present in the Toolbox, and all other people who have contributed to the Toolbox, given bug reports or other valuable feedback.

References

[1] Esa Alhoniemi, Johan Himberg, and Juha Vesanto. Probabilistic Measures for Responses of Self-Organizing Map Units. In H. Bothe, E. Oja, E. Massad, and C. Haefke, editors. *Proceeding of the International ICSC Congress on Computational Intelligence Methods and Applications (CIMA'99).* pages 286–290, ICSC Academic Press, 1999.

[2] Anderson E. The Irises of the Gaspe Peninsula. Bull. American Iris Society; 1935;59:2-5.

[3] Christopher M. Bishop. *Neural Networks for Pattern Recogition.* Oxford University Press, 1995.

[4] Pete Chapman, Julian Clinton, Thomas Khabaza, Thomas Reinartz, and Rüdiger Wirth. *The CRISP-DM process model.* Technical report, CRISM-DM consortium, March 1999. A discussion paper available from http://www.crisp-dm.org.

[5] P. Demartines and J. Hérault. Curvilinear Component Analysis: A Self-Organizing Neural Network for Nonlinear Mapping of Data Sets. *IEEE Transactions on Neural Networks*, 8(1):148–154, January 1997.

[6] Robert M. Gray. Vector quantization. *IEEE ASSP Magazine*, pages 4–29, April 1984.

[7] Johan Himberg. A SOM based cluster visualization and its application for false coloring. Accepted for publication in International Joint Conference in Neural Networks (IJCNN), 2000.

[8] Samuel Kaski. *Data Exploration Using Self-Organizing Maps.* PhD thesis, Helsinki University of Technology, 1997. Acta Polytechnica Scandinavica: Mathematics, Computing and Management in Engineering, 82.

[9] Samuel Kaski. Fast winner search for SOM-based monitoring and retrieval of high-dimensional data. In *Proceedings of ICANN99, Ninth International Conference on Artificial Neural Networks*, volume 2, pages 940–945. IEE, London, 1999.

[10] Samuel Kaski, Jarkko Venna, and Teuvo Kohonen. Coloring that reveals high-dimensional structures in data. In T. Gedeon, P. Wong, S. Halgamuge, N. Kasabov, D. Nauck, and K. Fukushima, editors, *Proceedings of ICONIP'99, 6th International Conference on Neural Information Processing*, volume II, pages 729–734. IEEE Service Center, Piscataway, NJ, 1999.

[11] Kimmo Kiviluoto Topology Preservation in Self-Organizing Maps. In *Proceeding of International Conference on Neural Networks (ICNN'96)*, 1996, pages 294–299.

[12] Teuvo Kohonen. *Self-Organizing Maps*, volume 30 of *Springer Series in Information Sciences.* Springer, Berlin, Heidelberg, 1995.

[13] Pasi Koikkalainen. Fast deterministic self-organizing maps. In *Proceedings of International Conference on Artificial Neural Networks (ICANN'95)*, pages 63–68, 1995.

[14] Kohonen T., Hynninen J., Kangas J., Laaksonen J. SOM_PAK: The Self-Organizing Map Program Package, Technical Report A31, Helsinki University of Technology, 1996, `http://www.cis.hut.fi/research/som_lvq_pak.shtml`

[15] Teuvo Kohonen, Samuel Kaski, Krista Lagus, Jarkko Salojärvi, Jukka Honkela, Vesa Paatero, and Antti Saarela. Self organization of a massive document collection. *IEEE Transactions on Neural Networks.* Accepted for publication.

[16] T. M. Martinetz, S. G. Berkovich, and K. J. Schulten. "Neural-gas" network for vector quantization and its application to time-series prediction. *IEEE Transactions on Neural Networks*, 4(4):558–569, 1993.

[17] Dorian Pyle. *Data Preparation for Data Mining.* Morgan Kaufmann Publishers, 1999.

[18] T. Samad and S. A. Harp. Self-Organization with Partial Data. *Network*, 3:205–212, 1992. IOP Publishing Ltd.

[19] John W. Sammon, Jr. A Nonlinear Mapping for Data Structure Analysis. *IEEE Transactions on Computers*, C-18(5):401–409, May 1969.

[20] A. Ultsch and H. P. Siemon, "Kohonen's Self Organizing Feature Maps for Exploratory Data Analysis," in *Proceedings of International Neural Network Conference (INNC'90)*, Dordrecht, Netherlands, 1990, pp. 305–308, Kluwer.

[21] Juha Vesanto, Esa Alhoniemi, Johan Himberg, Kimmo Kiviluoto, and Jukka Parviainen. Self-Organizing Map for Data Mining in Matlab: the SOM Toolbox. *Simulation News Europe*, (25):54, March 1999.

[22] Juha Vesanto. SOM-Based Data Visualization Methods. *Intelligent Data Analysis*, 3(2):111–126, 1999.

[23] Juha Vesanto, Johan Himberg, Esa Alhoniemi, and Juha Parhankangas. Self-organizing map in Matlab: the SOM Toolbox. In *Proceedings of the Matlab DSP Conference 1999*, pages 35–40, Espoo, Finland, November 1999.

[24] Juha Vesanto. Using SOM in data mining. Licentiate's thesis in the Helsinki University of Technology, April 2000.

A Changes with respect to version 1

Version 2 of SOM Toolbox is considerably different from version 1. The major changes are in structs, preprocessing and visualization. Then there are a number of minor changes in the way the functions work, and also in the names of the functions. In addition, there are a lot of new auxiliary functions.

A.1 Changes in structs

The most essential changes are in the structs. Therefore, two functions have been provided `som_vs2to1` and `som_vs1to2` which can be used to convert between version 1 and 2 structs.

Map struct The changes in the map struct are:

- the shape of `.codebook` matrix has been changed from `[ydim x xdim x ... zdim x dim]` to `[munits x dim]`. The change is the same as the one performed by the `reshape` function.
- the shape of `.labels` cell array has been changed correspondingly. It is no longer a cell array of cell strings of size `[ydim x xdim x ... zdim]` but a cell array of strings of size `[munits x ml]` where `ml` is the maximum number of labels in a single map unit.
- the topology information (fields `.msize, .lattice` and `.shape`) have been moved from the map struct to a separate topology struct (see below). The map struct has a field `.topol` which contains this struct.
- fields `.init_type`, `.train_type` and `.data_name` have been removed, and the field `.train_sequence` has been changed to field `.trainhist` which contains an array of training structs (see below).
- field `.normalization` field has been replaced with `.comp_norm` which contains a cell array of length `dim`, one for each variable. Each cell contains a struct array of normalization structs (see below).

Data struct The changes in the map struct are:

- field `.normalization` field has been replaced with `.comp_norm` which contains a cell array of length `dim`, one for each variable. Each cell contains a struct array of normalization structs (see below).

Topology struct This is a new struct, which contains the fields `.msize`, `.lattice` and `.shape` previously part of map struct. There's also a change in the values that the `.shape` field can get. The value `'rect'` referring to the shape of the map has been changed to `'sheet'`.

Training struct There are a number of changes in the contents of this struct. Field `.data_name` from map struct has been brought here, and fields `.neigh` and `.mask` have been copied here (also from map struct). Field `.qerror` has been dropped out.

Normalization struct The normalization struct now has normalization information for only one variable (as opposed to version 1, when it had information on all variables). For further information see notes on preprocessing below.

A.2 Preprocessing

The functions involved with preprocessing have completely changed. Instead of `som_normalize_data` and `som_denormalize_data`, the functions are `som_normalize`, `som_denormalize` and `som_norm_variable`. The functions `som_normalize` and `som_denormalize` envelope the operations of adding and removing normalization operations to data and map structs, and call `som_norm_variable` to do the actual work.

Function `som_norm_variable` performs normalization operations for a single variable at a time. The possible actions are `'init'`, `'do'` and `'undo'`, and the possible normalization operations are `'var'`, `'range'`, `'log'`, `'logistic'`, `'histD'` and `'histC'`. Of these `'var'` equals `'som_var_norm'`, `'range'` equals `'som_lin_norm'` and `'histD'` equals `'som_hist_norm'` of version 1. Normalization of vectors to unit length (`'som_unit_norm'`) is *not* implemented in version 2. New normalization operations in version 2 are `'log'` and `'histC'`. The normalization operations are detailed in Section 4.2.4.

A.3 Function names

There are a number of changes in the function names. The table below lists all such 1st version functions in alphabetical order. On the right is the corresponding 2nd version function. In some cases the corresponding function is missing.

Version 1 function	Version 2 function
`som_addhits`	`som_show_add`
`som_addlabels`	`som_show_add`
`som_addtraj`	`som_show_add`
`som_cca`	`cca`
`som_clear`	`som_show_clear`
`som_create`	`som_map_struct`
`som_demo`	run the demos directly
`som_denormalize_data`	`som_denormalize`
`som_doit`	-
`som_figuredata`	`vis_som_show_data`
`som_init`	`som_make`
`som_train`	`som_make`
`som_normalize_data`	`som_normalize`
`som_pca`	`pcaproj`
`som_manualclassify`	`som_select`
`som_name`	`som_set`
`som_plane`	`som_cplane`
`som_plane3`	-
`som_planeH`	`som_cplane`
`som_planeL`	`som_grid`
`som_planeT`	`som_show_add`
`som_planeU`	`som_cplane`
`som_profile`	`som_plotplane`
`som_projection`	run the projections directly
`som_sammon`	`sammon`
`som_showgrid`	`som_grid`
`som_showtitle`	`som_footnote`
`som_showtitleButtonDownFcn`	`som_footnoteButtonDownFcn`
`som_trainops`	-
`som_unit_distances`	`som_unit_dists`
`som_unit_neighborhood`	`som_neighborhood`
`somui_it`	`som_gui`
`somui_vis`	-

B File formats

B.1 Data file format (.data)

The first line contains the input space dimension and nothing else. For example:

```
6
```

Alternatively, the first line can be omitted. In that case, the input dimension can be given as an explicit argument to function `som_read_data`, or if that is not done, the function tries to determine the input dimension from the first data lines encountered.

After the first line come data lines, comment lines or empty lines. Each data line contains one data vector and its labels. From the beginning of the line, first are values of the vector components separated by whitespaces, then labels, again separated by whitespaces. If there are missing values in the vector, they should be indicated with a specific string given as an argument to `som_read_data`. By default the string is `'NaN'`. For example:

```
10 15 NaN 1e-6 45 -12 label1 label2 longer_label
```

Comment lines start with `'#'`. Comment lines as well as empty lines are ignored, except if the comment line starts with `'#n '` or `'#l '`. In the former case the line should contain names of the vector components separated by whitespaces. In the latter case, the names of labels are given. For example:

```
#n power CUI moment energy residual output
```

Example data file:

```
---->8---->8---->8---->8---->8---->8----
3
#n height weight fingers
#l nimi
## height in cm, weight in kg
170 50 10  Mika
175 68 10  Toni
160 45  9  Yksi Puuttuu
NaN 89 10
---->8---->8---->8---->8---->8---->8----
```

B.2 Map file format (.cod)

The first line must contain the input space dimension, lattice type (`'rect'` or `'hexa'`), map grid size in x-direction, map grid size in y-direction, and neighborhood function (`'bubble'` or `'gaussian'`), in that order. For example:

```
6 rect 10 8 bubble
```

The following lines are data lines, comment lines or empty lines. Each data line contains the weight vector of one map unit and its labels. From the beginning of the line, first are values of the vector components separated by whitespaces, then labels, again separated by whitespaces. The data lines must not contain any missing components. For example:

```
10 15 13.4 1e-6 45 -12 label1 label2 longer_label
```

The order of map units in the file are one row at a time from right to left, from the top to the bottom of the map (x-direction first, then y-direction, note that this is opposite order from the one the map units are in the `.codebook` field of map struct). Naturally, the number of data lines should be equal to the product of map grid sizes on the first line.

Comment lines start with `'#'`. Comment lines as well as empty lines are ignored, except if the comment line starts with `'#n '`. In that case the line should contain names of the vector components separated by whitespaces.

Example codebook file:

```
---->8---->8---->8---->8---->8---->8----
3 rect 1 2 gaussian
#n height weight fingers
## height in cm, weight in kg
  167 59.2038 9.6888 Mika Toni
169.5  6.7962e+1  9.8112
---->8---->8---->8---->8---->8---->8----
```

B.3 Deviation from SOM_PAK format

Although the file formats are very close to those used in SOM_PAK, they are not entirely the same.

- SOM_PAK file format features that are not supported include:
  - data file `'weight'` identifier
  - map file `'fixed'` identifier

  Both `'weight'` and `'fixed'` are treated as labels if encountered.

- The Toolbox has special meaning for comment lines starting with `'#n '` or `'#l '`.

- To make data reading faster, there's a small kludge. When reading in the data files, the maximum value Matlab is able to deal with (`realmax`) should not appear in the input file. This is because function `sscanf` is not able to read `NaN`s: they are in the reading phase converted to the value of `realmax` with precision of 100 digits.