The C Clustering Library

The University of Tokyo, Institute of Medical Science, Human Genome Center

Michiel de Hoon, Seiya Imoto, Satoru Miyano

23 September 2017

The C Clustering Library for cDNA microarray data.

Table of Contents

1 Introduction

Clustering is widely used in gene expression data analysis. By grouping genes together based on the similarity between their gene expression profiles, functionally related genes may be found. Such a grouping suggests the function of presently unknown genes.

The C Clustering Library is a collection of numerical routines that implement the clustering algorithms that are most commonly used. The routines can be applied both to genes and to arrays. The clustering algorithms are:

- Hierarchical clustering (pairwise centroid-, single-, complete-, and average-linkage);
- k-means clustering;
- Self-Organizing Maps;
- Principal Component Analysis.

To measure the similarity or distance between gene expression data, eight distance measures are available:

- Pearson correlation;
- Absolute value of the Pearson correlation;
- Uncentered Pearson correlation (equivalent to the cosine of the angle between two data vectors);
- Absolute uncentered Pearson correlation (equivalent to the cosine of the smallest angle between two data vectors);
- Spearman's rank correlation;
- Kendall's rank correlation τ;
- Euclidean distance;
- City-block distance.

This library was written in ANSI C and can therefore be easily linked to other C/C++ programs. Cluster 3.0 (`http://bonsai.hgc.jp/~mdehoon/software/cluster`) is an example of such a program. This library may be particularly useful when called from a scripting language such as Python (`http://www.python.org`), Perl (`http://www.perl.org`), or Ruby (`http://www.ruby.org`). The C Clustering Library contains wrappers for Python and Perl; interfaces to other scripting languages may be generated using SWIG (`http://www.swig.org`).

This manual contains a description of clustering techniques, their implementation in the C Clustering Library, the Python and Perl modules that give access to the C Clustering Library, and information on how to use the routines in the library from other C or C++ programs.

The C Clustering Library was released under the Python License.

Michiel de Hoon (`michiel.dehoon "AT" riken.jp`; `mdehoon "AT" cal.berkeley.edu`), Seiya Imoto, Satoru Miyano
Laboratory of DNA Information Analysis, Human Genome Center, Institute of Medical Science, University of Tokyo.

2 Distance functions

In order to cluster gene expression data into groups with similar genes or microarrays, we should first define what exactly we mean by *similar*. In the C Clustering Library, eight distance functions are available to measure similarity, or conversely, distance:

'c'   Pearson correlation coefficient;

'a'   Absolute value of the Pearson correlation coefficient;

'u'   Uncentered Pearson correlation (equivalent to the cosine of the angle between two data vectors);

'x'   Absolute uncentered Pearson correlation;

's'   Spearman's rank correlation;

'k'   Kendall's rank correlation τ;

'e'   Euclidean distance;

'b'   City-block distance.

The first six of these distance measures are related to the correlation coefficient, while the remaining three are related to the Euclidean distance. The characters in front of the distance measures are used as mnemonics to be passed to various routines in the C Clustering Library.

One of the properties one would like to see in a distance function is that it satisfies the triangle inequality:

$$d\left(\underline{u}, \underline{v}\right) \leq d\left(\underline{u}, \underline{w}\right) + d\left(\underline{w}, \underline{v}\right) \text{ for all } \underline{u}, \underline{v}, \underline{w}.$$

In everyday language, this equation means that the shortest distance between two points is a straight line.

Correlation-based distance functions usually define the distance d in terms of the correlation r as

$$d = 1 - r.$$

All correlation-based similarity measures are converted to a distance using this definition. Note that this distance function does not satisfy the triangle inequality. As an example, try

$$\underline{u} = \left(1, 0, -1\right);$$

$$\underline{v} = \left(1, 1, 0\right).$$

$$\underline{w} = \left(0, 1, 1\right);$$

Using the Pearson correlation, we find $d\left(\underline{u}, \underline{w}\right) = 1.8660$, while $d\left(\underline{u}, \underline{v}\right) + d\left(\underline{v}, \underline{w}\right) = 1.6340$. None of the distance functions based on the correlation coefficient satisfy the triangle inequality; this is a general characteristic of the correlation coefficient. The Euclidean distance and the city-block distance, which are *metrics*, do satisfy the triangle inequality. The correlation-based distance functions are sometimes called *semi-metric*.

2.1 Data handling

The input to the distance functions contains two arrays and two row or column indices, instead of two data vectors. This makes it easier to calculate the distance between two columns in the gene expression data matrix. If the distance functions would require two vectors, we would have to extract two columns from the matrix and save them in two vectors to be passed to the distance function. In order to specify if the distance between rows or between columns is to be calculated, each distance function has a flag *transpose*. If `transpose==0`, then the distance between two rows is calculated. Otherwise, the distance between two columns is calculated.

2.2 Weighting

For most of the distance functions available in the C Clustering Library, a weight vector can be applied. The weight vector contains weights for the elements in the data vector. If the weight for element i is w_i, then that element is treated as if it occurred w_i times in the data. The weight do not have to be integers. For the Spearman rank correlation and Kendall's rank correlation τ, discussed below, the weights do not have a well-defined meaning and are therefore not implemented.

2.3 Missing Values

Often in microarray experiments, some of the data values are missing. In the distance functions, we therefore use an additional matrix *mask* which shows which data values are missing. If `mask[i][j]==0`, then `data[i][j]` is missing, and is not included in the distance calculation.

2.4 The Pearson correlation coefficient

The Pearson correlation coefficient is defined as

$$r = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{x_i - \bar{x}}{\sigma_x} \right) \left(\frac{y_i - \bar{y}}{\sigma_y} \right)$$

in which $\bar{x}, \bar{y}$ are the sample mean of x and y respectively, and σ_x, σ_y are the sample standard deviation of x and y. The Pearson correlation coefficient is a measure for how well a straight line can be fitted to a scatterplot of x and y. If all the points in the scatterplot lie on a straight line, the Pearson correlation coefficient is either $+1$ or -1, depending on whether the slope of line is positive or negative. If the Pearson correlation coefficient is equal to zero, there is no correlation between x and y.

The *Pearson distance* is then defined as

$$d_{\mathrm{P}} \equiv 1 - r.$$

As the Pearson correlation coefficient lies between -1 and 1, the Pearson distance lies between 0 and 2.

Note that the Pearson correlation automatically centers the data by subtracting the mean, and normalizes them by dividing by the standard deviation. While such normalization may be useful in some situations (e.g., when clustering gene expression levels directly

instead of gene expression ratios), information is being lost in this step. In particular, the magnitude of changes in gene expression is being ignored. This is in fact the reason that the Pearson distance does not satisfy the triangle inequality.

2.5 Absolute Pearson correlation

By taking the absolute value of the Pearson correlation, we find a number between zero and one. If the absolute value is one, all the points in the scatter plot lie on a straight line with either a positive or a negative slope. If the absolute value is equal to zero, there is no correlation between x and y.

The distance is defined as usual as

$$d_A \equiv 1 - |r|\,,$$

where r is the Pearson correlation coefficient. As the absolute value of the Pearson correlation coefficient lies between 0 and 1, the corresponding distance lies between 0 and 1 as well.

In the context of gene expression experiments, note that the absolute correlation is equal to one if the gene expression data of two genes/microarrays have a shape that is either exactly the same or exactly opposite. The absolute correlation coefficient should therefore be used with care.

2.6 Uncentered correlation (cosine of the angle)

In some cases, it may be preferable to use the *uncentered correlation* instead of the regular Pearson correlation coefficient. The uncentered correlation is defined as

$$r_U = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{x_i}{\sigma_x^{(0)}} \right) \left(\frac{y_i}{\sigma_y^{(0)}} \right),$$

where

$$\sigma_x^{(0)} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} x_i^2};$$

$$\sigma_y^{(0)} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} y_i^2}.$$

This is the same expression as for the regular Pearson correlation coefficient, except that the sample means $\bar{x}, \bar{y}$ are set equal to zero. The uncentered correlation may be appropriate if there is a zero reference state. For instance, in the case of gene expression data given in terms of log-ratios, a log-ratio equal to zero corresponds to the green and red signal being equal, which means that the experimental manipulation did not affect the gene expression.

The distance corresponding to the uncentered correlation coefficient is defined as

$$d_U \equiv 1 - r_U,$$

where r_U is the uncentered correlation. As the uncentered correlation coefficient lies between -1 and 1, the corresponding distance lies between 0 and 2.

The uncentered correlation is equal to the cosine of the angle of the two data vectors in n-dimensional space, and is often referred to as such. (From this viewpoint, it would make more sense to define the distance as the arc cosine of the uncentered correlation coefficient).

2.7 Absolute uncentered correlation

As for the regular Pearson correlation, we can define a distance measure using the absolute value of the uncentered correlation:

$$d_{\mathrm{AU}} \equiv 1 - |r_{\mathrm{U}}|,$$

where r_{U} is the uncentered correlation coefficient. As the absolute value of the uncentered correlation coefficient lies between 0 and 1, the corresponding distance lies between 0 and 1 as well.

Geometrically, the absolute value of the uncentered correlation is equal to the cosine between the supporting lines of the two data vectors (i.e., the angle without taking the direction of the vectors into consideration).

2.8 Spearman rank correlation

The Spearman rank correlation is an example of a non-parametric similarity measure. It is useful because it is more robust against outliers than the Pearson correlation.

To calculate the Spearman rank correlation, we replace each data value by their rank if we would order the data in each vector by their value. We then calculate the Pearson correlation between the two rank vectors instead of the data vectors.

Weights cannot be suitably applied to the data if the Spearman rank correlation is used, especially since the weights are not necessarily integers. The calculation of the Spearman rank correlation in the C Clustering Library therefore does not take any weights into consideration.

As in the case of the Pearson correlation, we can define a distance measure corresponding to the Spearman rank correlation as

$$d_{\mathrm{S}} \equiv 1 - r_{\mathrm{S}},$$

where r_{S} is the Spearman rank correlation.

2.9 Kendall's rank correlation

Kendall's rank correlation, usually referred to as Kendall's τ, is another example of a non-parametric similarity measure. It is similar to the Spearman rank correlation, but instead of the ranks themselves only the relative ranks are used to calculate τ (see Snedecor & Cochran). As in the case of the Spearman rank correlation, the weights are ignored in the calculation.

We can define a distance measure corresponding to Kendall's τ as

$$d_{\mathrm{K}} \equiv 1 - \tau.$$

As Kendall's τ is defined such that it will lie between -1 and 1, the corresponding distance will be between 0 and 2.

2.10 Euclidean distance

The Euclidean distance is a true metric, as it satisfies the triangle inequality. In this software package, we define the Euclidean distance as

$$d = \frac{1}{n} \sum_{i=1}^{n} \left(x_i - y_i \right)^2 .$$

Only those terms are included in the summation for which both x_i and y_i are present. The denominator n is chosen accordingly.

In this formula, the expression data x_i and y_i are subtracted directly from each other. We should therefore make sure that the expression data are properly normalized when using the Euclidean distance, for example by converting the measured gene expression levels to log-ratios.

Unlike the correlation-based distance functions, the Euclidean distance takes the magnitude of the expression data into account. It therefore preserves more information about the data and may be preferable. De Hoon, Imoto, Miyano (2002) give an example of the use of the Euclidean distance for k-means clustering.

2.11 City-block distance

The city-block distance, alternatively known as the Manhattan distance, is related to the Euclidean distance. Whereas the Euclidean distance corresponds to the length of the shortest path between two points, the city-block distance is the sum of distances along each dimension. As gene expression data tend to have missing values, in the C Clustering Library we define the city-block distance as the sum of distances divided by the number of dimensions:

$$d = \frac{1}{n} \sum_{i=1}^{n} \left| x_i - y_i \right| .$$

This is equal to the distance you would have to walk between two points in a city, where you have to walk along city blocks. The city-block distance is a metric, as it satisfies the triangle inequality. As for the Euclidean distance, the expression data are subtracted directly from each other, and we should therefore make sure that they are properly normalized.

2.12 Calculating the distance between clusters

In the hierarchical clustering methods, the distance matrix between all genes/microarrays is first calculated, and at successive steps of the algorithm the new distance matrix is calculated from the previous distance matrix. In some cases, however, we would like to calculate the distance between clusters directly, given their members. For this purpose, the function `clusterdistance` can be used. This function can also be used to calculate the distance between two genes/microarrays by defining two clusters consisting of one gene/microarray each. While this function is not used internally in the C Clustering Library, it may be used by other C applications, and can also be called from Python (See ⟨undefined⟩ [Using the C Clustering Library with Python: Pycluster and Bio.Cluster], page ⟨undefined⟩) and Perl (See ⟨undefined⟩ [Using the C Clustering Library with Perl: Algorithm::Cluster], page ⟨undefined⟩).

The distance between two clusters can be defined in several ways. The distance between the arithmetic means of the two clusters is used in pairwise centroid-linkage clustering and in k-means clustering. For the latter, the distance between the medians of the two clusters can be used alternatively. The shortest pairwise distance between elements of the two clusters is used in pairwise single-linkage clustering, while the longest pairwise distance is used in pairwise maximum-linkage clustering. In pairwise average-linkage clustering, the distance between two clusters is defined as the average over the pairwise distances.

Prototype

```
double clusterdistance (int nrows, int ncolumns, double** data, int** mask,
double weight[], int n1, int n2, int index1[], int index2[], char dist, char
method, int transpose);
```

Arguments

- `int nrows;`
  The number of rows in the *data* matrix, equal to the number of genes in the gene expression experiment.

- `int ncolumns;`
  The number of columns in the *data* matrix, equal to the number of microarrays in the gene expression experiment.

- `double** data;`
  The data array containing the gene expression data. Genes are stored row-wise, while microarrays are stored column-wise. Dimension: `[nrows][ncolumns]`.

- `int** mask;`
  This array shows which elements in the *data* array, if any, are missing. If `mask[i][j]==0`, then `data[i][j]` is missing. Dimension: `[nrows][ncolumns]`.

- `double weight[];`
  The weights that are used to calculate the distance. Dimension: `[ncolumns]` if `transpose==0`; `[nrows]` if `transpose==1`.

- `int n1;`
  The number of elements in the first cluster.

- `int n2;`
  The number of elements in the second cluster.

- `int index1[];`
  Contains the indices of the elements belonging to the first cluster. Dimension: `[n1]`.

- `int index2[];`
  Contains the indices of the elements belonging to the second cluster. Dimension: `[n2]`.

- `char dist;`
  Specifies which distance measure is used. See ⟨undefined⟩ [Distance functions], page ⟨undefined⟩.

- `char method;`
  Specifies how the distance between clusters is defined:

  'a'   Distance between the two cluster centroids (arithmetic mean);

  'm'   Distance between the two cluster centroids (median);

's'	Shortest pairwise distance between elements in the two clusters;
'x'	Longest pairwise distance between elements in the two clusters;
'v'	Average over the pairwise distances between elements in the two clusters.

- `int transpose;`
  If `transpose==0`, the distances between rows in the data matrix are calculated. Otherwise, the distances between columns are calculated.

Return value

The distance between two clusters (`double`).

2.13 The distance matrix

The first step in clustering problems is usually to calculate the distance matrix. This matrix contains all the distances between the items that are being clustered. As the distance functions are symmetric, the distance matrix is also symmetric. Furthermore, the elements on the diagonal are zero, as the distance of an item to itself is zero. The distance matrix can therefore be stored as a ragged array, with the number of columns in each row equal to the (zero-offset) row number. The distance between items i and j is stored in location `[i][j]` if $j < i$, in `[j][i]` if $j > i$, while it is zero if $j = i$. Note that the first row of the distance matrix is empty. It is included for computational convenience, as including an empty row requires minimal storage.

Prototype

```
double** distancematrix (int nrows, int ncolumns, double** data, int** mask,
double weight[], char dist, int transpose);
```

Arguments

- `int nrows;`
  The number of rows in the data matrix, equal to the number of genes in the gene expression experiment.
- `int ncolumns;`
  The number of columns in the data matrix, equal to the number of microarrays in the gene expression experiment.
- `double** data;`
  The data array containing the gene expression data. Genes are stored row-wise, while microarrays are stored column-wise. Dimension: `[nrows][ncolumns]`.
- `int** mask;`
  This array shows which elements in the *data* array, if any, are missing. If `mask[i][j]==0`, then `data[i][j]` is missing. Dimension: `[nrows][ncolumns]`.
- `double weight[];`
  The weights that are used to calculate the distance. Dimension: `[ncolumns]` if `transpose==0`; `[nrows]` if `transpose==1`.
- `char dist;`
  Specifies which distance measure is used. See ⟨undefined⟩ [Distance functions], page ⟨undefined⟩.

- int *transpose*;
  If *transpose*==0, the distances between the rows in the data matrix are calculated. Otherwise, the distances between the columns are calculated.

Return value

A pointer to the distance matrix stored as a newly allocated ragged array. If insufficient memory is available to store the distance matrix, a NULL pointer is returned, and all memory allocated thus far is freed.

3 Partitioning algorithms

Partitioning algorithms divide items into k clusters such that the sum of distances over the items to their cluster centers is minimal. The number of clusters k is specified by the user. In the C Clustering Library, three partitioning algorithms are available:

- k-means clustering
- k-medians clustering
- k-medoids clustering

These algorithms differ in how the cluster center is defined. In k-means clustering, the cluster center is defined as the mean data vector averaged over all items in the cluster. Instead of the mean, in k-medians clustering the median is calculated for each dimension in the data vector. Finally, in k-medoids clustering the cluster center is defined as the item which has the smallest sum of distances to the other items in the cluster. This clustering algorithm is suitable for cases in which the distance matrix is known but the original data matrix is not available, for example when clustering proteins based on their structural similarity.

The expectation-maximization (EM) algorithm is commonly used to find the partitioning into k groups. The first step in the EM algorithm is to create k clusters and randomly assign items (genes or microarrays) to them. We then iterate:

- Calculate the centroid of each cluster;
- For each item, determine which cluster centroid is closest;
- Reassign the item to that cluster.

The iteration is stopped if no further item reassignments take place.

As the initial assignment of items to clusters is done randomly, usually a different clustering solution is found each time the EM algorithm is executed. To find the optimal clustering solution, the k-means algorithm is repeated many times, each time starting from a different initial random clustering. The sum of distances of the items to their cluster center is saved for each run, and the solution with the smallest value of this sum will be returned as the overall clustering solution.

How often the EM algorithm should be run depends on the number of items being clustered. As a rule of thumb, we can consider how often the optimal solution was found. This number is returned by the partitioning algorithms as implemented in this library. If the optimal solution was found many times, it is unlikely that better solutions exist than the one that was found. However, if the optimal solution was found only once, there may well be other solutions with a smaller within-cluster sum of distances.

3.1 Initialization

The k-means algorithm is initialized by randomly assigning items (genes or microarrays) to clusters. To ensure that no empty clusters are produced, we use the binomial distribution to randomly choose the number of items in each cluster to be one or more. We then randomly permute the cluster assignments to items such that each item has an equal probability to be in any cluster. Each cluster is thus guaranteed to contain at least one item.

3.2 Finding the cluster centroid

The centroid of a cluster can be defined in different ways. For k-means clustering, the centroid of a cluster is defined as the mean over all items in a cluster for each dimension separately. For robustness against outliers, in k-medians clustering the median is used instead of the mean. In k-medoids clustering, the cluster centroid is the item with the smallest sum of distances to the other items in the cluster. The C Clustering Library provides routines to calculate the cluster mean, the cluster median, and the cluster medoid.

3.2.1 Finding the cluster mean or median

The routine `getclustercentroids` calculates the centroids of the clusters by calculating the mean or median for each dimension separately over all items in a cluster. Missing data values are not included in the calculation of the mean or median. Missing values in the cluster centroids are indicated in the array *cmask*. If for cluster i the data values for dimension j are missing for all items, then `cmask[i][j]` (or `cmask[j][i]` if `transpose`==1) is set equal to zero. Otherwise, it is set equal to one. The argument *method* determines if the means or medians are calculated. If a memory allocation error occurred, `getclustercentroids` returns 0; this can happen only if the cluster medians are being calculated. If no errors occur, `getclustercentroids` returns 1;

Prototype

int getclustercentroids(int `nclusters`, int `nrows`, int `ncolumns`, double** `data`, int** `mask`, int `clusterid[]`, double** `cdata`, int** `cmask`, int `transpose`, char `method`);

Arguments

- int `nclusters`;
  The number of clusters.

- int `nrows`;
  The number of rows in the data matrix, equal to the number of genes in the gene expression experiment.

- int `ncolumns`;
  The number of columns in the data matrix, equal to the number of microarrays in the gene expression experiment.

- double** `data`;
  The data array containing the gene expression data. Genes are stored row-wise, while microarrays are stored column-wise. Dimension: `[nrows][ncolumns]`.

- int** `mask`;
  This array shows which elements in the *data* array, if any, are missing. If `mask[i][j]`==0, then `data[i][j]` is missing. Dimension: `[nrows][ncolumns]`.

- int `clusterid[]`;
  The cluster number to which each item belongs. Each element in this array should be between 0 and `nclusters`-1 inclusive. Dimension: `[nrows]` if `transpose`==0, or `[ncolumns]` if `transpose`==1.

- double** `cdata`;
  This matrix stores the centroid information. Space for this matrix should be

allocated before calling `getclustercentroids`. Dimension: [*nclusters*][*ncolumns*] if `transpose==0` (row-wise clustering), or [*nrows*][*nclusters*] if `transpose==1` (column-wise clustering).

- `int** cmask`;
  This matrix stores which values in *cdata* are missing. If `cmask[i][j]==0`, then `cdata[i][j]` is missing. Space for *cmask* should be allocated before calling `getclustercentroids`. Dimension: [*nclusters*][*ncolumns*] if `transpose==0` (row-wise clustering), or [*nrows*][*nclusters*] if `transpose==1` (column-wise clustering).

- `int transpose`;
  This flag indicates whether row-wise (gene) or column-wise (microarray) clustering is being performed. If `transpose==0`, rows (genes) are being clustered. Otherwise, columns (microarrays) are being clustered.

- `char method`;
  If `method=='a'`, the cluster centroids are calculated as the arithmetic mean over each dimension. For `method=='m'`, the cluster centroids are calculated as the median over each dimension.

Return value

This routine returns 1 if successful, and 0 in case of a memory allocation error.

3.2.2 Finding the cluster medoid

The cluster medoid is defined as the item which has the smallest sum of distances to the other items in the cluster. The `getclustermedoids` routine calculates the cluster medoids, given to which cluster each item belongs.

Prototype

`void getclustermedoids(int nclusters, int nelements, double** distance, int clusterid[], int centroids[], double errors[]);`

Arguments

- `int nclusters`;
  The number of clusters.

- `int nelements`;
  The total number of elements that are being clustered.

- `double** distmatrix`;
  The distance matrix. The distance matrix is symmetric and has zeros on the diagonal. To save space, the distance matrix is stored as a ragged array. Dimension: [*nelements*][] as a ragged array. The number of columns in each row is equal to the row number (starting from zero). Accordingly, the first row always has zero columns.

- `int clusterid[]`;
  The cluster number to which each element belongs. Dimension: [*nelements*].

- `int centroid[]`;
  For each cluster, the element of the item that was determined to be its centroid. Dimension: [*nclusters*].

- `double errors[];`
  For each cluster, the sum of distances between the items belonging to the cluster and the cluster centroid. Dimension: [*nclusters*].

3.3 The EM algorithm

The EM algorithm as implemented in the C Clustering Library first randomly assigns items to clusters, followed by iterating to find a clustering solution with a smaller within-cluster sum of distances. During the iteration, first we find the centroids of all clusters, where the centroids are defined in terms of the mean, the median, or the medoid. The distances of each item to the cluster centers are calculated, and we determine for each item which cluster is closest. We then reassign the items to their closest clusters, and recalculate the cluster centers.

All items are first reassigned before recalculating the cluster centroids. If unchecked, clusters may become empty if all their items are reassigned. For k-means and k-medians clustering, the EM routine therefore keeps track of the number of items in each cluster at all times, and prohibits the last remaining item in a cluster from being reassigned to a different cluster. For k-medoids clustering, such a check is not needed, as the item that functions as the cluster centroid has a zero distance to itself, and will therefore not be reassigned to a different cluster.

The EM algorithm terminates when no further reassignments take place. We noticed, however, that for some sets of initial cluster assignments, the EM algorithm fails to converge due to the same clustering solution reappearing periodically after a small number of iteration steps. In the EM algorithm as implemented in the C Clustering Library, the occurrence of such periodic solutions is checked for. After a given number of iteration steps, the current clustering result is saved as a reference. By comparing the clustering result after each subsequent iteration step to the reference state, we can determine if a previously encountered clustering result is found. In such a case, the iteration is halted. If after a given number of iterations the reference state has not yet been encountered, the current clustering solution is saved to be used as the new reference state. Initially, ten iteration steps are executed before resaving the reference state. This number of iteration steps is doubled each time, to ensure that periodic behavior with longer periods can also be detected.

3.4 Finding the optimal solution

3.4.1 k-means and k-medians

The optimal solution is found by executing the EM algorithm repeatedly and saving the best clustering solution that was returned by it. This can be done automatically by calling the routine `kcluster`. The routine to calculate the cluster centroid and the distance function are selected based on the arguments passed to `kcluster`.

The EM algorithm is then executed repeatedly, saving the best clustering solution that was returned by these routines. In addition, `kcluster` counts how often the EM algorithm found this solution. If it was found many times, we can assume that there are no other solutions possible with a smaller within-cluster sum of distances. If, however, the solution was found only once, it may well be that better clustering solutions exist.

Prototype

void kcluster (int *nclusters*, int *nrows*, int *ncolumns*, double** *data*, int** *mask*, double *weight*[], int *transpose*, int *npass*, char *method*, char *dist*, int *clusterid*[], double* *error*, int* *ifound*);

Arguments

- int *nclusters*;
  The number of clusters k.

- int *nrows*;
  The number of rows in the data matrix, equal to the number of genes in the gene expression experiment.

- int *ncolumns*;
  The number of columns in the data matrix, equal to the number of microarrays in the gene expression experiment.

- double** *data*;
  The data array containing the gene expression data. Genes are stored row-wise, while microarrays are stored column-wise. Dimension: [*nrows*][*ncolumns*].

- int** *mask*;
  This array shows which elements in the *data* array, if any, are missing. If *mask*[i][j]==0, then *data*[i][j] is missing. Dimension: [*nrows*][*ncolumns*].

- double *weight*[];
  The weights that are used to calculate the distance. Dimension: [*ncolumns*] if *transpose*==0; [*nrows*] if *transpose*==1.

- int *transpose*;
  This flag indicates whether row-wise (gene) or column-wise (microarray) clustering is being performed. If *transpose*==0, rows (genes) are being clustered. Otherwise, columns (microarrays) are being clustered.

- int *npass*;
  The number of times the EM algorithm should be run. If *npass* > 0, each run of the EM algorithm uses a different (random) initial clustering. If *npass* == 0, then the EM algorithm is run with an initial clustering specified by *clusterid*. Reassignment to a different cluster is prevented for the last remaining item in the cluster in order to prevent empty clusters. For *npass*==0, the EM algorithm is run only once, using the initial clustering as specified by *clusterid*.

- char *method*;
  Specifies whether the arithmetic mean (*method*=='a') or the median (*method*=='m') should be used to calculate the cluster center.

- char *dist*;
  Specifies which distance function should be used. The character should correspond to one of the distance functions that are available in the C Clustering Library. See ⟨undefined⟩ [Distance functions], page ⟨undefined⟩.

- int *clusterid*[];
  This array will be used to store the cluster number to which each item was assigned by the clustering algorithm. Space for clusterid should be allocated before calling kcluster. If *npass*==0, then the contents of *clusterid* on input is used as the

initial assignment of items to clusters; on output, *clusterid* contains the optimal clustering solution found by the EM algorithm. Dimension: [`nrows`] if `transpose`==0, or [`ncolumns`] if `transpose`==1.

- `double* error`;
  The sum of distances of the items to their cluster center after k-means clustering, which can be used as a criterion to compare clustering solutions produced in different calls to `kcluster`.

- `int* ifound`;
  Returns how often the optimal clustering solution was found. In case of an inconsistency in the input arguments (specifically, if *nclusters* is larger than the number of elements to be clustered), `*ifound` is set to 0. If a memory allocation error occurred, `*ifound` is set to -1.

3.4.2 k-medoids

The kmedoids routine performs k-medoids clustering on a given set of elements, using the distance matrix and the number of clusters passed by the user. Multiple passes are being made to find the optimal clustering solution, each time starting from a different initial clustering.

Prototype

```
void kmedoids (int nclusters, int nelements, double** distance, int npass, int
clusterid[], double* error, int* ifound);
```

Arguments

- `int nclusters`;
  The number of clusters to be found.

- `int nelements`;
  The number of elements to be clustered.

- `double** distmatrix`;
  The distance matrix. The distance matrix is symmetric and has zeros on the diagonal. To save space, the distance matrix is stored as a ragged array. Dimension: [`nelements`][] as a ragged array. The number of columns in each row is equal to the row number (starting from zero). Accordingly, the first row always has zero columns.

- `int npass`;
  The number of times the EM algorithm should be run. If `npass` > 0, each run of the EM algorithm uses a different (random) initial clustering. If `npass` == 0, then the EM algorithm is run with an initial clustering specified by *clusterid*.

- `int clusterid[]`;
  This array will be used to store the cluster number to which each item was assigned by the clustering algorithm. Space for `clusterid` should be allocated before calling `kcluster`. On input, if `npass`==0, then *clusterid* contains the initial clustering assignment from which the clustering algorithm starts; all numbers in *clusterid* should be between 0 and `nelements`-1 inclusive. If `npass`!=0, *clusterid* is ignored on input. On output, *clusterid* contains the number of the cluster to which each item was assigned in the optimal clustering solution. The number of a cluster is defined as the item number of the centroid of the cluster. Dimension: [`nelements`].

- `double* error;`
  The sum of distances of the items to their cluster center after k-means clustering, which can be used as a criterion to compare clustering solutions produced in different calls to `kmedoids`.

- `int* ifound;`
  Returns how often the optimal clustering solution was found. In case of an inconsistency in the input arguments (specifically, if *nclusters* is larger than *nelements*), *ifound* is set to 0. If a memory allocation error occurred, *ifound* is set to -1.

3.5 Choosing the distance measure

Whereas all eight distance measures are accepted for k-means, k-medians, and k-medoids clustering, using a distance measure other than the Euclidean distance or city-block distance with k-means or k-medians is in a sense inconsistent. When using the distance measures based on the Pearson correlation, the data are effectively normalized when calculating the distance. However, no normalization is applied when calculating the centroid in the k-means or k-medians algorithm. From a theoretical viewpoint, it is best to use the Euclidean distance for the k-means algorithm, and the city-block distance for k-medians.

4 Hierarchical clustering

Hierarchical clustering methods are inherently different from the k-means clustering method. In hierarchical clustering, the similarity in the expression profile between genes or experimental conditions are represented in the form of a tree structure. This tree structure can be shown graphically by programs such as TreeView and Java TreeView, which has contributed to the popularity of hierarchical clustering in the analysis of gene expression data.

The first step in hierarchical clustering is to calculate the distance matrix, specifying all the distances between the items to be clustered. Next, we create a node by joining the two closest items. Subsequent nodes are created by pairwise joining of items or nodes based on the distance between them, until all items belong to the same node. A tree structure can then be created by retracing which items and nodes were merged. Unlike the EM algorithm, which is used in k-means clustering, the complete process of hierarchical clustering is deterministic.

Several flavors of hierarchical clustering exist, which differ in how the distance between subnodes is defined in terms of their members. In the C Clustering Library, pairwise single, maximum, average, and centroid linkage are available.

- In pairwise single-linkage clustering, the distance between two nodes is defined as the shortest distance among the pairwise distances between the members of the two nodes.
- In pairwise maximum-linkage clustering, alternatively known as pairwise complete-linkage clustering, the distance between two nodes is defined as the longest distance among the pairwise distances between the members of the two nodes.
- In pairwise average-linkage clustering, the distance between two nodes is defined as the average over all pairwise distances between the elements of the two nodes.
- In pairwise centroid-linkage clustering, the distance between two nodes is defined as the distance between their centroids. The centroids are calculated by taking the mean over all the elements in a cluster. As the distance from each newly formed node to existing nodes and items need to be calculated at each step, the computing time of pairwise centroid-linkage clustering may be significantly longer than for the other hierarchical clustering methods. Another peculiarity is that (for a distance measure based on the Pearson correlation), the distances do not necessarily increase when going up in the clustering tree, and may even decrease. This is caused by an inconsistency between the centroid calculation and the distance calculation when using the Pearson correlation: Whereas the Pearson correlation effectively normalizes the data for the distance calculation, no such normalization occurs for the centroid calculation.

For pairwise single-, complete-, and average-linkage clustering, the distance between two nodes can be found directly from the distances between the individual items. Therefore, the clustering algorithm does not need access to the original gene expression data, once the distance matrix is known. For pairwise centroid-linkage clustering, however, the centroids of newly formed subnodes can only be calculated from the original data and not from the distance matrix.

The implementation of pairwise single-linkage hierarchical clustering is based on the SLINK algorithm (R. Sibson, 1973), which is much faster and more memory-efficient than a straightforward implementation of pairwise single-linkage clustering. The clustering result produced by this algorithm is identical to the clustering solution found by the conventional

single-linkage algorithm. The single-linkage hierarchical clustering algorithm implemented in this library can be used to cluster large gene expression data sets, for which conventional hierarchical clustering algorithms fail due to excessive memory requirements and running time.

4.1 Representing a hierarchical clustering solution

The result of hierarchical clustering consists of a tree of nodes, in which each node joins two items or subnodes. Usually, we are not only interested in which items or subnodes are joined at each node, but also in their similarity (or distance) as they are joined. To store one node in the hierarchical clustering tree, we make use of a struct `Node`, which has the following members:

- `int left;`
  First item or subnode.

- `int right;`
  Second item or subnode.

- `double distance;`
  The distance between the two items or subnodes that were joined.

Each item and subnode is represented by an integer. For hierarchical clustering of n items, we number the original items $\{0, \ldots, n-1\}$, nodes are numbered $\{-1, \ldots, -(n-1)\}$. Note that the number of nodes is one less than the number of items.

A hierarchical clustering tree can now be written as an array of `Node` structs. The `treecluster` routine allocates this array, and returns a `Node*` pointer to the first element. The calling function is responsible for deallocating the `Node*` array.

4.2 Performing hierarchical clustering: `treecluster`

The `treecluster` routine implements pairwise single-, complete, average-, and centroid-linkage clustering. A pointer *distmatrix* to the distance matrix can be passed as one of the arguments to `treecluster`; if this pointer is `NULL`, the `treecluster` routine will calculate the distance matrix from the gene expression data using the arguments *data*, *mask*, *weight*, and *dist*. For pairwise single-, complete-, and average-linkage clustering, the `treecluster` routine ignores these four arguments if *distmatrix* is given, as the distance matrix by itself is sufficient for the clustering calculation. For pairwise centroid-linkage clustering, on the other hand, the arguments *data*, *mask*, *weight*, and *dist* are always needed, even if *distmatrix* is available.

The `treecluster` routine will complete faster if it can make use of a previously calculated distance matrix passed as the *distmatrix* argument. Note, however, that newly calculated distances are stored in the distance matrix, and its elements may be rearranged during the clustering calculation. Therefore, in order to save the original distance matrix, it should be copied before `treecluster` is called. The memory that was allocated by the calling routine for the distance matrix will not be deallocated by `treecluster`, and should be deallocated by the calling routine after `treecluster` returns. If *distmatrix* is `NULL`, however, `treecluster` takes care both of the allocation and the deallocation of memory for the distance matrix. In that case, `treecluster` may fail if not enough memory can be allocated for the distance matrix, in which case `treecluster` returns `NULL`.

Prototype

`tree* treecluster (int nrows, int ncolumns, double** data, int** mask, double weight[], int transpose, char dist, char method, double** distmatrix);`

Arguments

- `int nrows;`
  The number of rows in the *data* matrix, equal to the number of genes in the gene expression experiment.

- `int ncolumns;`
  The number of columns in the *data* matrix, equal to the number of microarrays in the gene expression experiment.

- `double** data;`
  The data array containing the gene expression data. Genes are stored row-wise, while microarrays are stored column-wise. Dimension: `[nrows][ncolumns]`.

- `int** mask;`
  This array shows which elements in the *data* array, if any, are missing. If `mask[i][j]==0`, then `data[i][j]` is missing. Dimension: `[nrows][ncolumns]`.

- `double weight[];`
  The weights that are used to calculate the distance. Dimension: `[ncolumns]` if `transpose==0`; `[nrows]` if `transpose==1`.

- `int transpose;`
  This flag indicates whether row-wise (gene) or column-wise (microarray) clustering is being performed. If `transpose==0`, rows (genes) are being clustered. Otherwise, columns (microarrays) are being clustered.

- `char dist;`
  Specifies which distance measure is used. See ⟨undefined⟩ [Distance functions], page ⟨undefined⟩.

- `char method;`
  Specifies which type of hierarchical clustering is used:
    - `'s'`: pairwise single-linkage clustering
    - `'m'`: pairwise maximum- (or complete-) linkage clustering
    - `'a'`: pairwise average-linkage clustering
    - `'c'`: pairwise centroid-linkage clustering

- `double** distmatrix;`
  The distance matrix, stored as a ragged array. This argument is optional; if the distance matrix is not available, it can be passed as `NULL`. In that case, `treecluster` will allocate memory space for the distance matrix, calculate it from the gene expression data, and deallocate the memory space before returning. If the distance matrix happens to be available, the hierarchical clustering calculation can be completed faster by passing it as the *distmatrix* argument. Note that the contents of the distance matrix will be modified by the clustering algorithm in `treecluster`. The memory that was allocated for the distance matrix should be deallocated by the calling routine after `treecluster` returns. Dimension: Ragged array, `[nrows][]` if `transpose==0`, or `[ncolumns][]` if

`transpose`==1. In both cases, the number of columns in each row is equal to the row number (starting from zero). Accordingly, the first row always has zero columns.

Return value

A pointer to a newly allocated `tree` structure that describes the calculated hierarchical clustering solution. If `treecluster` fails due to a memory allocation error, it returns `NULL`.

4.3 Sorting a hierarchical clustering tree: `sorttree`

A hierarchical clustering solution of n elements can be drawn as 2^{n-1} different trees. While all of these trees represent the same hierarchical clustering solution, they differ from each other by the ordering of the left and right subnode at each node, which is arbitrary. The routine `sorttree` sorts a hierarchical clustering tree by visiting each node and if needed switching the left and right subnode such that the average order value of the left node is less than or equal to the average order value of the right node. Here, the order value of the elements is specified by the user. Effectively, the `sorttree` routine finds the dendrogram such that the left-to-right order of the elements tend to have increasing order values, while remaining consistent with the hierarchical clustering solution.

Prototype

`int sorttree(const int `*`nnodes`*`, Node* `*`tree`*`, const double `*`order`*`[], int `*`indices`*`[]);`

Arguments

- `int `*`nnodes`*`;`
  The number of nodes in the hierarchical clustering tree.

- `Node* `*`tree`*`;`
  The hierarchical clustering solution. Each node `tree[i]` in the array describes one linking event, with `tree[i].left` and `tree[i].right` containing the numbers of the nodes that were joined. The original elements are numbered {0, ..., `nelements`-1}, nodes are numbered {-1, ..., -(`nelements`-1)}. Note that the number of nodes is one less than the number of elements. The `sorttree` function performs no error checking of the *tree* structure. Dimension: [`nnodes`].

- `const double `*`order`*`[];`
  The order values of the items. Dimension: [`nnodes`+1]. If `order` is NULL, the tree is not reordered, and the `indices[]` of the current tree are calculated.

- `int `*`indices`*`[];`
  The indices of each item after sorting, with item `indices[i]` appearing at position i (counting left-to-right) in the hierarchical clustering tree after sorting. Memory space for `indices` should be allocated before `sorttree` is called. Dimension: [`nnodes`+1].

Return value

The `sorttree` routine returns 1 if successful, and 0 in case of a memory allocation error.

4.4 Cutting a hierarchical clustering tree: `cuttree`

The tree structure generated by the hierachical clustering routine `treecluster` can be further analyzed by dividing the genes or microarrays into n clusters, where n is some positive integer less than or equal to the number of items that were clustered. This can be achieved by ignoring the top $n - 1$ linking events in the tree structure, resulting in n separated subnodes. The items in each subnode are then assigned to the same cluster. The routine `cuttree` determines to which cluster each item is assigned, based on the hierarchical clustering result stored in the tree structure. Clusters are numbered from 0 to $n - 1$ in the left-to-right order in which they appear in the hierarchical clustering tree.

Prototype

```
void cuttree (int nelements, Node* tree, unsigned int nclusters, int
clusterid[]);
```

Arguments

- `int nelements`;
  The number of elements for which the clustering results are stored in the hierarchical clustering result *tree*.

- `Node* tree`;
  The hierarchical clustering solution. Each node `tree[i]` in the array describes one linking event, with `tree[i].left` and `tree[i].right` containing the numbers of the nodes that were joined. The original elements are numbered {0, ..., `nelements`-1}, nodes are numbered {-1, ..., -(`nelements`-1)}. Note that the number of nodes is one less than the number of elements. The `cuttree` function performs no error checking of the *tree* structure. Dimension: [`nelements`-1].

- `unsigned int nclusters`;
  The desired number of clusters. The number of clusters should be positive, and less than or equal to *nelements*.

- `int clusterid[]`;
  The cluster number to which each element is assigned. Memory space for `clusterid` should be allocated before `cuttree` is called. Dimension: [`nelements`].

5 Self-Organizing Maps

Self-Organizing Maps (SOMs) were invented by Kohonen to describe neural networks (see for instance Kohonen, 1997). Tamayo (1999) first applied Self-Organizing Maps to gene expression data.

SOMs organize items into clusters that are situated in some topology. Usually a rectangular topology is chosen. The clusters generated by SOMs are such that neighboring clusters in the topology are more similar to each other than clusters far from each other in the topology.

The first step to calculate a SOM is to randomly assign a data vector to each cluster in the topology. If genes are being clustered, then the number of elements in each data vector is equal to the number of microarrays in the experiment.

An SOM is then generated by taking genes one at a time, and finding which cluster in the topology has the closest data vector. The data vector of that cluster, as well as those of the neighboring clusters, are adjusted using the data vector of the gene under consideration. The adjustment is given by

$$\Delta \underline{x}_{\mathrm{cell}} = \tau \cdot \left(\underline{x}_{\mathrm{gene}} - \underline{x}_{\mathrm{cell}} \right).$$

The parameter τ is a parameter that decreases at each iteration step. We have used a simple linear function of the iteration step:

$$\tau = \tau_{\mathrm{init}} \cdot \left(1 - \frac{i}{n} \right),$$

in which τ_{init} is the initial value of τ as specified by the user, i is the number of the current iteration step, and n is the total number of iteration steps to be performed. While changes are made rapidly in the beginning of the iteration, at the end of iteration only small changes are made.

All clusters within a radius R are adjusted to the gene under consideration. This radius decreases as the calculation progresses as

$$R = R_{\mathrm{max}} \cdot \left(1 - \frac{i}{n} \right),$$

in which the maximum radius is defined as

$$R_{\mathrm{max}} = \sqrt{N_x^2 + N_y^2},$$

where (N_x, N_y) are the dimensions of the rectangle defining the topology.

The routine `somcluster` carries out the complete SOM algorithm. First it initializes the random number generator. The distance function to be used is specified by *dist*. The node data are then initialized using the random number generator. The order in which genes or microarrays are used to modify the SOM is also randomized. The total number of iterations is specified by *niter*, given by the user.

Prototype

```
void somcluster (int nrows, int ncolumns, double** data, int** mask, double
weight[], int transpose, int nxgrid, int nygrid, double inittau, int niter,
char dist, double*** celldata, int clusterid[][2]);
```

Arguments

- `int nrows;`
  The number of rows in the data matrix, equal to the number of genes in the gene expression experiment.

- `int ncolumns;`
  The number of columns in the data matrix, equal to the number of microarrays in the gene expression experiment.

- `double** data;`
  The data array containing the gene expression data. Genes are stored row-wise, while microarrays are stored column-wise. Dimension: `[nrows][ncolumns]`.

- `int** mask;`
  This array shows which elements in the *data* array, if any, are missing. If `mask[i][j]==0`, then `data[i][j]` is missing. Dimension: `[nrows][ncolumns]`.

- `double weight[];`
  The weights that are used to calculate the distance. Dimension: `[ncolumns]` if `transpose==0`; `[nrows]` if `transpose==1`.

- `int transpose;`
  This flag indicates whether row-wise (gene) or column-wise (microarray) clustering is being performed. If `transpose==0`, rows (genes) are being clustered. Otherwise, columns (microarrays) are being clustered.

- `int nxgrid;`
  The number of cells horizontally in the rectangular topology containing the clusters.

- `int nygrid;`
  The number of cells vertically in the rectangular topology containing the clusters.

- `double inittau;`
  The initial value for the parameter τ that is used in the SOM algorithm. A typical value for *inittau* is 0.02, which was used in Michael Eisen's Cluster/TreeView program.

- `int niter;`
  The total number of iterations.

- `char dist;`
  Specifies which distance measure is used. See ⟨undefined⟩ [Distance functions], page ⟨undefined⟩.

- `double*** celldata;`
  The data vectors of the clusters in the rectangular topology that were found by the SOM algorithm. These correspond to the cluster centroids. The first dimension is the horizontal position of the cluster in the rectangle, the second dimension is the vertical position of the cluster in the rectangle, while the third dimension is the dimension along the data vector. The `somcluster` routine does not allocate storage space for the *celldata* array. Space should be allocated before calling `somcluster`. Alternatively, if *celldata* is equal to `NULL`, the `somcluster` routine allocates space for *celldata* and frees it before returning. In that case, `somcluster` does not return the data vectors of the clusters that were found. Dimension: `[nxgrid][nygrid][ncolumns]` if `transpose==0`, or `[nxgrid][nygrid][nrows]` if `transpose==1`.

- `int clusterid[][2];`
  Specifies the cluster to which a gene or microarray was assigned, using two integers to identify the horizontal and vertical position of a cell in the grid for each gene or microarray. Gene or microarrays are assigned to clusters in the rectangular grid by determining which cluster in the rectangular topology has the closest data vector. Space for the *clusterid* argument should be allocated before calling `somcluster`. If *clusterid* is `NULL`, the `somcluster` routine ignores this argument and does not return the cluster assignments. Dimension: `[nrows][2]` if `transpose`==0; `[ncolumns][2]` if `transpose`==1.

6 Principal Component Analysis

Principal Component Analysis (PCA) is a widely used technique for analyzing multivariate data. A practical example of applying Principal Component Analysis to gene expression data is presented by Yeung and Ruzzo (2001).

In essence, PCA is a coordinate transformation in which each row in the data matrix is written as a linear sum over basis vectors called principal components, which are ordered and chosen such that each maximally explains the remaining variance in the data vectors. For example, an $n \times 3$ data matrix can be represented as an ellipsoidal cloud of n points in three dimensional space. The first principal component is the longest axis of the ellipsoid, the second principal component the second longest axis of the ellipsoid, and the third principal component is the shortest axis. Each row in the data matrix can be reconstructed as a suitable linear combination of the principal components. However, in order to reduce the dimensionality of the data, usually only the most important principal components are retained. The remaining variance present in the data is then regarded as unexplained variance.

The principal components can be found by calculating the eigenvectors of the covariance matrix of the data. The corresponding eigenvalues determine how much of the variance present in the data is explained by each principal component.

Before applying principal component analysis, typically the mean is subtracted from each column in the data matrix. In the example above, this effectively centers the ellipsoidal cloud around its centroid in 3D space, with the principal components describing the variation of points in the ellipsoidal cloud with respect to their centroid.

The function `pca` below first uses the singular value decomposition to calculate the eigenvalues and eigenvectors of the data matrix. The singular value decomposition is implemented as a translation in C of the Algol procedure `svd` (Golub and Reinsch, 1970), which uses Householder bidiagonalization and a variant of the QR algorithm. The principal components, the coordinates of each data vector along the principal components, and the eigenvalues corresponding to the principal components are then evaluated and returned in decreasing order of the magnitude of the eigenvalue. If data centering is desired, the mean should be subtracted from each column in the data matrix before calling the `pca` routine.

Prototype

`int pca(int nrows, int ncolumns, double** u, double** v, double* w);`
applies Principal Component Analysis to the data matrix *u*.

Arguments

- `int nrows`
  The number of rows in `u`.

- `int ncolumns`
  The number of columns in `u`.

- `double** u;`
  On input: `u` is the rectangular matrix to which Principal Component Analysis is to be applied. The function assumes that the mean has already been subtracted of each column, and hence that the mean of each column is zero.

On output:
If **nrows** ≥ **ncolumns**, then on output **u** contains the coordinates with respect to the principal components.
If **nrows** < **ncolumns**, then on output **u** contains the principal component vectors.
Dimension: **[nrows][ncolumns]**.

- **double** v;**
Unused on input.
On output:
If **nrows** ≥ **ncolumns**, then on output **v** contains the principal component vectors.
If **nrows** < **ncolumns**, then on output **v** contains the coordinates with respect to the principal components.
Dimension: **[n][n]**, where $n = \min(\textit{nrows}, \textit{ncolumns})$.

- **double* w;**
Unused on input.
On output: The eigenvalues corresponding to the the principal component vectors.
Dimension: **[n]**, where $n = \min(\textit{nrows}, \textit{ncolumns})$.

If **nrows** ≥ **ncolumns**, then on output the dot product $u \cdot v$ reproduces the data originally passed in through u. If **nrows** < **ncolumns**, then on output the dot product $v \cdot u$ reproduces the data originally passed in through u.

Return value

The function returns 0 if successful, -1 if memory allocation fails, and a positive integer if the singular value decomposition fails to converge.

7 The random number generator

The random number generator in the C Clustering Library is used to initialize the k-means/medians/medoids clustering algorithm and Self-Organizing Maps (SOMs), as well as to randomly select a gene or microarray in the calculation of a SOM.

We need both a generator of uniform random deviates, and a generator whose deviates follow the binomial distribution. The latter can be built, given a uniform random number generator, using the BTPE algorithm by Kachitvichyanukul and Schmeiser (1988).

The uniform random number generator in the C Clustering Library is described by L'Ecuyer (1988). The random number generator is initialized automatically during its first call. As the random number generator by L'Ecuyer uses a combination of two multiplicative linear congruential generators, two (integer) seeds are needed for initialization, for which we use the system-supplied random number generator `rand` (in the C standard library). We initialize this generator by calling `srand` with the epoch time in seconds, and use the first two random numbers generated by `rand` as seeds for the uniform random number generator in the C Clustering Library.

8 Using the C Clustering Library with Python: Pycluster and Bio.Cluster

The C Clustering Library is particularly useful when used as a module to a scripting language. This chapter describes the routines in the C Clustering Library as seen from Python.

To make the routines available to Python, install Pycluster (see Section 10.1 [Installing the C Clustering Library for Python], page 73). From Python, then type

```
>>> from Pycluster import *
```

Pycluster is also available as part of Biopython (`http://www.biopython.org`), in which case you should use

```
>>> from Bio.Cluster import *
```

This will give you access to the clustering routines in the C Clustering Library directly from Python.

The function

```
>>> version()
```

returns the version number of the C Clustering Library as a string.

8.1 Partitioning algorithms

8.1.1 k-means and k-medians clustering: `kcluster`

clusterid, *error*, *nfound* = kcluster (data, nclusters=2, mask=None, weight=None, transpose=0, npass=1, method='a', dist='e', initialid=None) implements the k-means and k-medians clustering algorithms.

Arguments

- `data`
  Array containing the gene expression data, where genes are stored row-wise and microarray experiments column-wise.

- `nclusters`
  The number of clusters k.

- `mask`
  Array of integers showing which data are missing. If `mask[i][j]==0`, then `data[i][j]` is missing. If `mask==None`, then there are no missing data.

- `weight`
  contains the weights to be used when calculating distances. If `weight==None`, then equal weights are assumed.

- `transpose`
  Determines if genes or microarrays are being clustered. If `transpose==0`, genes (rows) are being clustered. If `transpose==1`, microarrays (columns) are clustered.

- `npass`
  The number of times the k-means clustering algorithm is performed, each time with a different (random) initial condition. If `initialid` is given, the value of `npass` is ignored and the clustering algorithm is run only once, as it behaves deterministically in that case.

- `method`
  describes how the center of a cluster is found:
    - `method=='a'`: arithmetic mean;
    - `method=='m'`: median.

  For other values of `method`, the arithmetic mean is used.
- `dist`
  defines the distance function to be used:
    - `dist=='c'`: correlation;
    - `dist=='a'`: absolute value of the correlation;
    - `dist=='u'`: uncentered correlation;
    - `dist=='x'`: absolute uncentered correlation;
    - `dist=='s'`: Spearman's rank correlation;
    - `dist=='k'`: Kendall's τ;
    - `dist=='e'`: Euclidean distance;
    - `dist=='b'`: City-block distance.
- `initialid`
  Specifies the initial clustering to be used for the EM algorithm. If `initialid==None`, then a different random initial clustering is used for each of the `npass` runs of the EM algorithm. If `initialid` is not `None`, then it should be equal to a 1D array containing the cluster number (between `0` and `nclusters-1`) for each item. Each cluster should contain at least one item. With the initial clustering specified, the EM algorithm is deterministic.

Return values

This function returns a tuple (`clusterid`, `error`, `nfound`).

- `clusterid`
  An array containing the number of the cluster to which each gene/microarray was assigned.
- `error`
  The within-cluster sum of distances for the optimal clustering solution.
- `nfound`
  The number of times the optimal solution was found.

8.1.2 *k*-medoids clustering: `kmedoids`

clusterid, *error*, *nfound* = kmedoids (distance, nclusters=2, npass=1,
initialid=None)
implements the *k*-medoids clustering algorithm.

Arguments

- `distance`
  The matrix containing the distances between the elements. You can specify the distance matrix in three ways:
    - as a 2D Numerical Python array (in which only the left-lower part of the array will be accessed):

```
distance = array([[0.0, 1.1, 2.3],
                   [1.1, 0.0, 4.5],
                   [2.3, 4.5, 0.0]])
```

- as a 1D Numerical Python array containing consecutively the distances in the left-lower part of the distance matrix:
  ```
  distance = array([1.1, 2.3, 4.5])
  ```

- as a list containing the rows of the left-lower part of the distance matrix:
  ```
  distance = [array([]),
              array([1.1]),
              array([2.3, 4.5])
             ]
  ```

These three expressions correspond to the same distance matrix.

- nclusters
  The number of clusters k.

- npass
  The number of times the k-medoids clustering algorithm is performed, each time with a different (random) initial condition. If initialid is given, the value of npass is ignored, as the clustering algorithm behaves deterministically in that case.

- initialid
  Specifies the initial clustering to be used for the EM algorithm. If initialid==None, then a different random initial clustering is used for each of the npass runs of the EM algorithm. If initialid is not None, then it should be equal to a 1D array containing the cluster number (between 0 and nclusters-1) for each item. Each cluster should contain at least one item. With the initial clustering specified, the EM algorithm is deterministic.

Return values

This function returns a tuple (*clusterid, error, nfound*).

- *clusterid*
  An array containing the number of the cluster to which each item was assigned, where the cluster number is defined as the item number of the item representing the cluster centroid.

- *error*
  The within-cluster sum of distances for the optimal k-medoids clustering solution.

- *nfound*
  The number of times the optimal solution was found.

8.2 Hierarchical clustering

The pairwise single-, maximum-, average-, and centroid-linkage clustering methods are accessible through the function treecluster. The hierarchical clustering routines can be applied either on the original gene expression data, or (except for centroid-linkage clustering) on the distance matrix directly. The tree structure generated by treecluster can be cut in order to separate the elements into a given number of clusters.

8.2.1 Representing a hierarchical clustering solution

A hierarchical clustering solution is represented using two Python classes: `Node` and `Tree`.

The class `Node`

The Python class `Node` corresponds to the C struct `Node` described above. A `Node` object has three attributes:

- left
- right
- distance

Here, `left` and `right` are integers referring to the two items or subnodes that are joined at this node, and `distance` is the distance between them. The items being clustered are numbered from 0 to (*number of items* - 1), while clusters are numbered -1 to -(*number of items*-1).

To create a new `Node` object, we need to specify `left` and `right`; `distance` is optional.

```
>>> from Pycluster import *
>>> Node(2,3)
(2, 3): 0
>>> Node(2,3,0.91)
(2, 3): 0.91
```

The attributes `left`, `right`, and `distance` of an existing `Node` object can be modified directly:

```
>>> node = Node(4,5)
>>> node.left = 6
>>> node.right = 2
>>> node.distance = 0.73
>>> node
(6, 2): 0.73
```

An error is raised if `left` and `right` are not integers, or if `distance` cannot be converted to a floating-point value.

The class `Tree`

The Python class `Tree` represents a full hierarchical clustering solution. A `Tree` object can be created from a list of `Node` objects:

```
>>> nodes = [Node(1,2,0.2), Node(0,3,0.5), Node(-2,4,0.6), Node(-1,-3,0.9)]
>>> tree = Tree(nodes)
>>> print tree
(1, 2): 0.2
(0, 3): 0.5
(-2, 4): 0.6
(-1, -3): 0.9
```

The `Tree` initializer checks if the list of nodes is a valid hierarchical clustering result:

```
>>> nodes = [Node(1,2,0.2), Node(0,2,0.5)]
>>> Tree(nodes)
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in ?
cluster.error: Inconsistent tree
```

Individual nodes in a `Tree` object can be accessed using square brackets:

```
>>> nodes = [Node(1,2,0.2), Node(0,-1,0.5)]
>>> tree = Tree(nodes)
>>> tree[0]
(1, 2): 0.2
>>> tree[1]
(0, -1): 0.5
>>> tree[-1]
(0, -1): 0.5
```

As a `Tree` object is read-only, we cannot change individual nodes in a `Tree` object. However, we can convert the tree to a list of nodes, modify this list, and create a new tree from this list:

```
>>> tree = Tree([Node(1,2,0.1), Node(0,-1,0.5), Node(-2,3,0.9)])
>>> print tree
(1, 2): 0.1
(0, -1): 0.5
(-2, 3): 0.9
>>> nodes = tree[:]
>>> nodes[0] = Node(0,1,0.2)
>>> nodes[1].left = 2
>>> tree = Tree(nodes)
>>> print tree
(0, 1): 0.2
(2, -1): 0.5
(-2, 3): 0.9
```

This guarantees that any `Tree` object is always well-formed.

A `Tree` object has two methods (`scale` and `cut`, described below). The `treecluster` function returns `Tree` objects.

8.2.2 Performing hierarchical clustering: `treecluster`

tree = treecluster(data=None, mask=None, weight=None, transpose=0,
method='m', dist='e', distancematrix=None)
implements the hierachical clustering methods.

Arguments

- `data`
  Array containing the gene expression data, where genes are stored row-wise and microarray experiments column-wise. Either `data` or `distancematrix` should be `None`.

- `mask`
  Array of integers showing which data are missing. If `mask[i][j]==0`, then `data[i][j]` is missing. If `mask==None`, then there are no missing data.

- `weight`
  contains the weights to be used when calculating distances. If `weight==None`, then equal weights are assumed.
- `transpose`
  Determines if genes or microarrays are being clustered. If `transpose==0`, genes (rows) are being clustered. If `transpose==1`, microarrays (columns) are clustered.
- `method`
  defines the linkage method to be used:
  - `method=='s'`: pairwise single-linkage clustering
  - `method=='m'`: pairwise maximum- (or complete-) linkage clustering
  - `method=='c'`: pairwise centroid-linkage clustering
  - `method=='a'`: pairwise average-linkage clustering
- `dist`
  defines the distance function to be used:
  - `dist=='c'`: correlation;
  - `dist=='a'`: absolute value of the correlation;
  - `dist=='u'`: uncentered correlation;
  - `dist=='x'`: absolute uncentered correlation;
  - `dist=='s'`: Spearman's rank correlation;
  - `dist=='k'`: Kendall's τ;
  - `dist=='e'`: Euclidean distance;
  - `dist=='b'`: City-block distance.
- `distancematrix`
  The distance matrix. Either `data` or `distancematrix` should be `None`. If `data` is `None` and `distancematrix` is given, the arguments `mask`, `weights`, `transpose`, and `dist` are ignored. Note that pairwise single-, maximum-, and average-linkage clustering can be calculated from the distance matrix, but pairwise centroid-linkage cannot.

  If not `None`, `distancematrix` should correspond to a distance matrix, which can be specified in three ways:
  - as a 2D Numerical Python array (in which only the left-lower part of the array will be accessed):
    ```
    distance = array([[0.0, 1.1, 2.3],
                      [1.1, 0.0, 4.5],
                      [2.3, 4.5, 0.0]])
    ```
  - as a 1D Numerical Python array containing consecutively the distances in the left-lower part of the distance matrix:
    ```
    distance = array([1.1, 2.3, 4.5])
    ```
  - as a list containing the rows of the left-lower part of the distance matrix:
    ```
    distance = [array([]),
                array([1.1]),
                array([2.3, 4.5])
               ]
    ```

These three expressions correspond to the same distance matrix. As `treecluster` may shuffle the values in the distance matrix as part of the clustering algorithm, be sure to save this array in a different variable before calling `treecluster` if you need it later.

Return values

This function returns a `Tree` object. This object contains *number of items* – 1 nodes, where the number of items is the number of genes if genes were clustered, or the number of microarrays if microarrays were clustered. Each node describes a pairwise linking event, where the node attributes `left` and `right` each contain the number of one gene/microarray or subnode, and `distance` the distance between them. Genes/microarrays are numbered from 0 to (*number of items* – 1), while clusters are numbered –1 to -(*number of items*–1).

8.2.3 Scaling a hierarchical clustering tree: *tree*.scale

To display a hierarchical clustering solution with Java TreeView, it is better to scale all node distances such that they are between zero and one. This can be accomplished by calling the `scale` method on an existing `Tree` object.

tree.scale()

scales the distances in the hierarchical clustering tree such that they are all between zero and one. This function takes no arguments, and returns `None`.

8.2.4 Sorting a hierarchical clustering tree: *tree*.sort

indices = *tree*.sort(order)

sorts the hierarchical clustering tree such that the elements tend to increase in the order value from left to right in the dendrogram, where the order value of each element is specified by the user. This is accomplished by visting each node in the hierarchical clustering tree, verifying if the average order value of the left node is less than or equal to the average order value of the right node, and switching the left and right node otherwise. The method returns the indices of the elements in the left-to-right order after sorting.

Arguments

- *tree*

  The `Tree` object *tree* contains the hierarchical clustering result generated by `treecluster`.

- order

  The order values of the elements. If `order` is `None`, the tree is not reordered, and the *indices* of the current tree are calculated.

Return values

This function returns the array *indices*.

- *indices*

  An array indicating the order of the elements in the dendogram after sorting, with element *indices*[i] occurring at position i in the left-to-right order in the dendogram.

8.2.5 Cutting a hierarchical clustering tree: `tree.cut`

clusterid = `tree.cut([nclusters])`
groups the items into **nclusters** clusters based on the tree structure generated by the hierarchical clustering routine **treecluster**.

Arguments

- *tree*
  The **Tree** object *tree* contains the hierarchical clustering result generated by **treecluster**.
- **nclusters**
  The desired number of clusters. If specified, **nclusters** should be positive, and less than or equal to the number of elements. If **nclusters** is not specified, the number of elements is used as the default value.

Return values

This function returns the array *clusterid*.

- *clusterid*
  An array containing the number of the cluster to which each gene/microarray is assigned. Clusters are numbered from 0 through **nclusters**-1 in the left-to-right order in which they appear in the hierarchical clustering tree.

8.3 Self-Organizing Maps: `somcluster`

clusterid, *celldata* = `somcluster(data, mask=None, weight=None, transpose=0, nxgrid=2, nygrid=1, inittau=0.02, niter=1, dist='e')`
implements a Self-Organizing Map on a rectangular grid.

Arguments

- `data`
  Array containing the gene expression data, where genes are stored row-wise and microarray experiments column-wise.
- `mask`
  Array of integers showing which data are missing. If `mask[i][j]==0`, then `data[i][j]` is missing. If `mask==None`, then there are no missing data.
- `weight`
  contains the weights to be used when calculating distances. If `weight==None`, then equal weights are assumed.
- `transpose`
  Determines if genes or microarrays are being clustered. If `transpose==0`, genes (rows) are being clustered. If `transpose==1`, microarrays (columns) are clustered.
- `nxgrid, nygrid`
  The number of cells horizontally and vertically in the rectangular grid, on which the Self-Organizing Map is calculated.
- `inittau`
  The initial value for the parameter τ that is used in the SOM algorithm. The default value for `inittau` is 0.02, which was used in Michael Eisen's Cluster/TreeView program.

- `niter`
  The number of iterations to be performed.
- `dist`
  defines the distance function to be used:
    - `dist=='c'`: correlation;
    - `dist=='a'`: absolute value of the correlation;
    - `dist=='u'`: uncentered correlation;
    - `dist=='x'`: absolute uncentered correlation;
    - `dist=='s'`: Spearman's rank correlation;
    - `dist=='k'`: Kendall's τ;
    - `dist=='e'`: Euclidean distance;
    - `dist=='b'`: City-block distance.

Return values

This function returns the tuple *clusterid, celldata*.

- *clusterid*
  An array with two columns, where the number of rows is equal to the number of genes or the number of microarrays depending on whether genes or microarrays are being clustered. Each row contains the x and y coordinates of the cell in the rectangular SOM grid to which the gene or microarray was assigned.
- *celldata*
  An array with dimensions (`nxgrid`, `nygrid`, *number of microarrays*) if genes are being clustered, or (`nxgrid`, `nygrid`, *number of genes*) if microarrays are being clustered. Each element `[ix][iy]` of this array is a 1D vector containing the gene expression data for the centroid of the cluster in the grid cell with coordinates (*ix,iy*).

8.4 Finding the cluster centroids: `clustercentroids`

`cdata, cmask = clustercentroids(data, mask=None, clusterid=None, method='a', transpose=0)`
calculates the cluster centroids.

Arguments

- `data`
  Array containing the gene expression data, where genes are stored row-wise and microarray experiments column-wise.
- `mask`
  Array of integers showing which data are missing. If `mask[i][j]==0`, then `data[i][j]` is missing. If `mask==None`, then there are no missing data.
- `clusterid`
  Vector of integers showing to which cluster each element belongs. If `clusterid` is not given, then all elements are assumed to belong to the same cluster.
- `method`
  Specifies whether the arithmetic mean (`method=='a'`) or the median (`method=='m'`) is used to calculate the cluster center.

- transpose
  Determines if gene or microarray clusters are being considered. If `transpose==0`, then we are considering clusters of genes (rows). If `transpose==1`, then we are considering clusters of microarrays (columns).

Return values

This function returns the tuple *cdata, cmask*.

- *cdata*
  A 2D array containing the centroid data. The dimensions of this array are (*number of clusters, number of microarrays*) if genes were clustered, or (*number of genes, number of clusters*) if microarrays were clustered. Each row (if genes were clustered) or column (if microarrays were clustered) contains the averaged gene expression data for corresponding to the centroid of one cluster that was found.

- *cmask*
  This matrix stores which values in *cdata* are missing. If *cmask*[i][j]==0, then *cdata*[i][j] is missing. The dimensions of this array are (*number of clusters, number of microarrays*) if genes were clustered, or (*number of genes, number of clusters*) if microarrays were clustered.

8.5 The distance between two clusters: `clusterdistance`

distance = clusterdistance(data, mask=None, weight=None, index1=0, index2=0,
method='a', dist='e', transpose=0)
calculates the distance between two clusters.

Arguments

- data
  Array containing the gene expression data, where genes are stored row-wise and microarray experiments column-wise.

- mask
  Array of integers showing which data are missing. If `mask[i][j]==0`, then `data[i][j]` is missing. If `mask==None`, then there are no missing data.

- weight
  contains the weights to be used when calculating distances. If `weight==None`, then equal weights are assumed.

- index1
  is a list containing the indices of the elements belonging to the first cluster. A cluster containing only one element *i* can be represented as either a list [*i*], or as an integer *i*.

- index2
  is a list containing the indices of the elements belonging to the second cluster. A cluster containing only one element *i* can be represented as either a list [*i*], or as an integer *i*.

- method
  Specifies how the distance between clusters is defined:
    - `method=='a'`: Distance between the two cluster centroids (arithmetic mean);

- `method=='m'`: Distance between the two cluster centroids (median);
- `method=='s'`: Shortest pairwise distance between elements in the two clusters;
- `method=='x'`: Longest pairwise distance between elements in the two clusters;
- `method=='v'`: Average over the pairwise distances between elements in the two clusters.
- `dist`
  defines the distance function to be used:
  - `dist=='c'`: correlation;
  - `dist=='a'`: absolute value of the correlation;
  - `dist=='u'`: uncentered correlation;
  - `dist=='x'`: absolute uncentered correlation;
  - `dist=='s'`: Spearman's rank correlation;
  - `dist=='k'`: Kendall's τ;
  - `dist=='e'`: Euclidean distance;
  - `dist=='b'`: City-block distance.
- `transpose`
  Determines if gene or microarray clusters are being considered. If `transpose==0`, then we are considering clusters of genes (rows). If `transpose==1`, then we are considering clusters of microarrays (columns).

Return values

This function returns the distance between the two clusters.

8.6 Calculating the distance matrix: `distancematrix`

matrix = distancematrix(data, mask=None, weight=None, transpose=0, dist='e')
returns the distance matrix between gene expression data.

Arguments

- `data`
  Array containing the gene expression data, where genes are stored row-wise and microarray experiments column-wise.
- `mask`
  Array of integers showing which data are missing. If `mask[i][j]==0`, then `data[i][j]` is missing. If `mask==None`, then there are no missing data.
- `weight`
  contains the weights to be used when calculating distances. If `weight==None`, then equal weights are assumed.
- `transpose`
  Determines if genes or microarrays are being clustered. If `transpose==0`, genes (rows) are being clustered. If `transpose==1`, microarrays (columns) are clustered.
- `dist`
  defines the distance function to be used:
  - `dist=='c'`: correlation;

- `dist=='a'`: absolute value of the correlation;
- `dist=='u'`: uncentered correlation;
- `dist=='x'`: absolute uncentered correlation;
- `dist=='s'`: Spearman's rank correlation;
- `dist=='k'`: Kendall's τ;
- `dist=='e'`: Euclidean distance;
- `dist=='b'`: City-block distance.

Return values

- *matrix* is a list of 1D arrays containing the distance matrix between the gene expression data. The number of columns in each row is equal to the row number. Hence, the first row has zero elements. An example of the return value is
  ```
  matrix = [array([]),
            array([1.]),
            array([7., 3.]),
            array([4., 2., 6.])]
  ```
  This corresponds to the distance matrix

$$\begin{pmatrix} 0 & 1 & 7 & 4 \\ 1 & 0 & 3 & 2 \\ 7 & 3 & 0 & 6 \\ 4 & 2 & 6 & 0 \end{pmatrix}$$

8.7 Principal Component Analysis: `pca`

columnmean, coordinates, components, eigenvalues = pca(data)
applies Principal Component Analysis to the rectangular matrix `data`.

Arguments

- `data`
  Array containing the data to which the Principal Component Analysis is to be applied.

Return values

The function returns a tuple *columnmean, coordinates, components, eigenvalues*.

- *columnmean*
  Array containing the mean over each column in `data`.
- *coordinates*
  The coordinates of each row in `data` with respect to the principal components.
- *components*
  The principal components.
- *eigenvalues*
  The eigenvalues corresponding to each of the principal components.

The original matrix `data` can be recreated by calculating *columnmean* + dot(*coordinates, components*).

8.8 Handling Cluster/TreeView-type files

Cluster/TreeView are GUI-based codes for clustering gene expression data. They were originally written by Michael Eisen (`http://rana.lbl.gov`) while at Stanford University. Pycluster contains functions for reading and writing data files that correspond to the format specified for Cluster/TreeView. In particular, by saving a clustering result in that format, TreeView can be used to visualize the clustering results. We recommend using Alok Saldanha's Java TreeView program (`http://jtreeview.sourceforge.net/`), which can display hierarchical as well as k-means clustering results.

8.8.1 The `Record` class

An object of the class `Record` contains all information stored in a Cluster/TreeView-type data file. To store the information contained in the data file in a `Record` object, we first open the file and then read it:

```
>>> import Pycluster
>>> handle = open("mydatafile.txt")
>>> record = Pycluster.read(handle)
```
This two-step process gives you some flexibility in the source of the data. For example, you can use

```
>>> import gzip # Python standard library
>>> handle = gzip.open("mydatafile.txt.gz")
```
to open a gzipped file, or

```
>>> import urllib # Python standard library
>>> handle = urllib.urlopen("http://somewhere.org/mydatafile.txt")
```
to open a file stored on the Internet before calling `read`.

If you're using the C Clustering Library from Biopython, you should use:

```
>>> from Bio import Cluster
>>> record = Cluster.read(handle)
```

The `read` command reads the tab-delimited text file `mydatafile.txt` containing gene expression data in the format specified for Michael Eisen's Cluster/TreeView program. For a description of this file format, see the manual to Cluster/TreeView. It is available at Michael Eisen's lab website (`http://rana.lbl.gov/manuals/ClusterTreeView.pdf`) and at our website (`http://bonsai.hgc.jp/~mdehoon/software/cluster/cluster3.pdf`).

A `Record` object stores the following information:

- `data`
  The data array containing the gene expression data. Genes are stored row-wise, while microarrays are stored column-wise.

- `mask`
  This array shows which elements in the `data` array, if any, are missing. If `mask[i,j]==0`, then `data[i,j]` is missing. If no data were found to be missing, `mask` is set to `None`.

- `geneid`
  This is a list containing a unique description for each gene (i.e., ORF numbers).

- `genename`
  This is a list containing a description for each gene (i.e., gene name). If not present in the data file, `genename` is set to `None`.
- `gweight`
  The weights that are to be used to calculate the distance in expression profile between genes. If not present in the data file, `gweight` is set to `None`.
- `gorder`
  The preferred order in which genes should be stored in an output file. If not present in the data file, `gorder` is set to `None`.
- `expid`
  This is a list containing a description of each microarray, e.g. experimental condition.
- `eweight`
  The weights that are to be used to calculate the distance in expression profile between microarrays. If not present in the data file, `eweight` is set to `None`.
- `eorder`
  The preferred order in which microarrays should be stored in an output file. If not present in the data file, `eorder` is set to `None`.
- `uniqid`
  The string that was used instead of UNIQID in the data file.

   After loading a `Record` object, each of these attributes can be accessed and modified directly. For example, the data can be log-transformed by taking the logarithm of `record.data`.

8.8.2 Performing hierarchical clustering

`tree = record.treecluster(transpose=0, method='m', dist='e')`
applies hierachical clustering to the data contained in the `Record` object.

Arguments

- `transpose`
  Determines if genes or microarrays are being clustered. If `transpose==0`, genes (rows) are being clustered. If `transpose==1`, microarrays (columns) are clustered.
- `method`
  defines the linkage method to be used:
  - `method=='s'`: pairwise single-linkage clustering
  - `method=='m'`: pairwise maximum- (or complete-) linkage clustering
  - `method=='c'`: pairwise centroid-linkage clustering
  - `method=='a'`: pairwise average-linkage clustering
- `dist`
  defines the distance function to be used:
  - `dist=='c'`: correlation;
  - `dist=='a'`: absolute value of the correlation;
  - `dist=='u'`: uncentered correlation;
  - `dist=='x'`: absolute uncentered correlation;

- `dist=='s'`: Spearman's rank correlation;
- `dist=='k'`: Kendall's τ;
- `dist=='e'`: Euclidean distance;
- `dist=='b'`: City-block distance.

Return values

This method returns a `Tree` object containing *number of items* - 1 nodes, where the number of items is the number of genes if genes were clustered, or the number of microarrays if microarrays were clustered. Each node describes a pairwise linking event, where the node attributes `left` and `right` each contain the number of one gene/microarray or subnode, and `distance` the distance between them. Genes/microarrays are numbered from 0 to (*number of items* - 1), while clusters are numbered -1 to -(*number of items*-1).

8.8.3 Performing k-means or k-medians clustering

clusterid, *error*, *nfound* = *record*.kcluster(nclusters=2, transpose=0, npass=1, method='a', dist='e', initialid=None)
applies k-means or k-medians clustering to the data contained in the `Record` object.

Arguments

- `nclusters`
  The number of clusters k.

- `transpose`
  Determines if genes or microarrays are being clustered. If `transpose==0`, genes (rows) are being clustered. If `transpose==1`, microarrays (columns) are clustered.

- `npass`
  The number of times the k-means clustering algorithm is performed, each time with a different (random) initial condition. If `initialid` is given, the value of `npass` is ignored and the clustering algorithm is run only once, as it behaves deterministically in that case.

- `method`
  describes how the center of a cluster is found:
  - `method=='a'`: arithmetic mean;
  - `method=='m'`: median.

  For other values of `method`, the arithmetic mean is used.

- `dist`
  defines the distance function to be used:
  - `dist=='c'`: correlation;
  - `dist=='a'`: absolute value of the correlation;
  - `dist=='u'`: uncentered correlation;
  - `dist=='x'`: absolute uncentered correlation;
  - `dist=='s'`: Spearman's rank correlation;
  - `dist=='k'`: Kendall's τ;
  - `dist=='e'`: Euclidean distance;

- `dist=='b'`: City-block distance.

- `initialid`
  Specifies the initial clustering to be used for the EM algorithm. If `initialid==None`, then a different random initial clustering is used for each of the `npass` runs of the EM algorithm. If `initialid` is not `None`, then it should be equal to a 1D array containing the cluster number (between `0` and `nclusters-1`) for each item. Each cluster should contain at least one item. With the initial clustering specified, the EM algorithm is deterministic.

Return values

This function returns a tuple (*clusterid, error, nfound*).

- *clusterid*
  An array containing the number of the cluster to which each gene/microarray was assigned.

- *error*
  The within-cluster sum of distances for the optimal clustering solution.

- *nfound*
  The number of times the optimal solution was found.

8.8.4 Calculating a Self-Organizing Map

clusterid, celldata = record.somcluster(transpose=0, nxgrid=2, nygrid=1, inittau=0.02, niter=1, dist='e')
calculates a Self-Organizing Map on a rectangular grid, using the gene expression data in the `Record` object *record*.

Arguments

- `transpose`
  Determines if genes or microarrays are being clustered. If `transpose==0`, genes (rows) are being clustered. If `transpose==1`, microarrays (columns) are clustered.

- `nxgrid, nygrid`
  The number of cells horizontally and vertically in the rectangular grid, on which the Self-Organizing Map is calculated.

- `inittau`
  The initial value for the parameter τ that is used in the SOM algorithm. The default value for `inittau` is 0.02, which was used in Michael Eisen's Cluster/TreeView program.

- `niter`
  The number of iterations to be performed.

- `dist`
  defines the distance function to be used:
    - `dist=='c'`: correlation;
    - `dist=='a'`: absolute value of the correlation;
    - `dist=='u'`: uncentered correlation;
    - `dist=='x'`: absolute uncentered correlation;

- `dist=='s'`: Spearman's rank correlation;
- `dist=='k'`: Kendall's τ;
- `dist=='e'`: Euclidean distance;
- `dist=='b'`: City-block distance.

Return values

This function returns the tuple *clusterid, celldata*.

- *clusterid*
  An array with two columns, where the number of rows is equal to the number of genes or the number of microarrays depending on whether genes or microarrays are being clustered. Each row contains the x and y coordinates of the cell in the rectangular SOM grid to which the gene or microarray was assigned.

- *celldata*
  An array with dimensions (`nxgrid`, `nygrid`, *number of microarrays*) if genes are being clustered, or (`nxgrid`, `nygrid`, *number of genes*) if microarrays are being clustered. Each element `[ix][iy]` of this array is a 1D vector containing the gene expression data for the centroid of the cluster in the grid cell with coordinates (*ix,iy*).

8.8.5 Finding the cluster centroid

cdata, cmask = record.clustercentroids(clusterid=None, method='a', transpose=0)
calculates the cluster centroids.

Arguments

- clusterid
  Vector of integers showing to which cluster each element belongs. If `clusterid` is not given, then all elements are assumed to belong to the same cluster.

- method
  Specifies whether the arithmetic mean (`method=='a'`) or the median (`method=='m'`) is used to calculate the cluster center.

- transpose
  Determines if gene or microarray clusters are being considered. If `transpose==0`, then we are considering clusters of genes (rows). If `transpose==1`, then we are considering clusters of microarrays (columns).

Return values

This function returns the tuple *cdata, cmask*.

- *cdata*
  A 2D array containing the centroid data. The dimensions of this array are (*number of clusters, number of microarrays*) if genes were clustered, or (*number of genes, number of clusters*) if microarrays were clustered. Each row (if genes were clustered) or column (if microarrays were clustered) contains the averaged gene expression data for corresponding to the centroid of one cluster that was found.

- *cmask*
  This matrix stores which values in *cdata* are missing. If *cmask*[i][j]==0, then

> `cdata[i][j]` is missing. The dimensions of this array are (*number of clusters, number of microarrays*) if genes were clustered, or (*number of genes, number of clusters*) if microarrays were clustered.

8.8.6 Calculating the distance between two clusters

`distance` = `record`.`clusterdistance(index1=0, index2=0, method='a', dist='e', transpose=0)`
calculates the distance between two clusters.

Arguments

- `index1`
  is a list containing the indices of the elements belonging to the first cluster. A cluster containing only one element *i* can be represented as either a list `[i]`, or as an integer *i*.
- `index2`
  is a list containing the indices of the elements belonging to the second cluster. A cluster containing only one element *i* can be represented as either a list `[i]`, or as an integer *i*.
- `method`
  Specifies how the distance between clusters is defined:
  - `method=='a'`: Distance between the two cluster centroids (arithmetic mean);
  - `method=='m'`: Distance between the two cluster centroids (median);
  - `method=='s'`: Shortest pairwise distance between elements in the two clusters;
  - `method=='x'`: Longest pairwise distance between elements in the two clusters;
  - `method=='v'`: Average over the pairwise distances between elements in the two clusters.
- `dist`
  defines the distance function to be used:
  - `dist=='c'`: correlation;
  - `dist=='a'`: absolute value of the correlation;
  - `dist=='u'`: uncentered correlation;
  - `dist=='x'`: absolute uncentered correlation;
  - `dist=='s'`: Spearman's rank correlation;
  - `dist=='k'`: Kendall's τ;
  - `dist=='e'`: Euclidean distance;
  - `dist=='b'`: City-block distance.
- `transpose`
  Determines if gene or microarray clusters are being considered. If `transpose==0`, then we are considering clusters of genes (rows). If `transpose==1`, then we are considering clusters of microarrays (columns).

Return values

This function returns the distance between the two clusters.

8.8.7 Calculating the distance matrix

matrix = *record*.distancematrix(transpose=0, dist='e')
returns the distance matrix between gene expression data.

Arguments

- transpose
  Determines if genes or microarrays are being clustered. If transpose==0, genes (rows) are being clustered. If transpose==1, microarrays (columns) are clustered.
- dist
  defines the distance function to be used:
    - dist=='c': correlation;
    - dist=='a': absolute value of the correlation;
    - dist=='u': uncentered correlation;
    - dist=='x': absolute uncentered correlation;
    - dist=='s': Spearman's rank correlation;
    - dist=='k': Kendall's τ;
    - dist=='e': Euclidean distance;
    - dist=='b': City-block distance.

Return values

- *matrix* is a list of 1D arrays containing the distance matrix between the gene expression data. The number of columns in each row is equal to the row number. Hence, the first row has zero elements. An example of the return value is
  matrix = [array([]),
         array([1.]),
         array([7., 3.]),
         array([4., 2., 6.])]
  This corresponds to the distance matrix

$$\begin{pmatrix} 0 & 1 & 7 & 4 \\ 1 & 0 & 3 & 2 \\ 7 & 3 & 0 & 6 \\ 4 & 2 & 6 & 0 \end{pmatrix}$$

8.8.8 Saving the clustering result

record.save(jobname, geneclusters, expclusters)
writes the text file *jobname*.cdt, *jobname*.gtr, *jobname*.atr, *jobname**.kgg, and/or *jobname**.kag for subsequent reading by the Java TreeView program. If geneclusters and expclusters are both None, this method only writes the text file *jobname*.cdt; this file can subsequently be read into a new Record object.

Arguments

- jobname
  The string jobname is used as the base name for names of the files that are to be saved.

- geneclusters
  This argument describes the gene clustering result. In case of k-means clustering, this is a 1D array containing the number of the cluster each gene belongs to. It can be calculated using `kcluster`. In case of hierarchical clustering, `geneclusters` is a `Tree` object.

- expclusters
  This argument describes the clustering result for the experimental conditions. In case of k-means clustering, this is a 1D array containing the number of the cluster each experimental condition belongs to. It can be calculated using `kcluster`. In case of hierarchical clustering, `expclusters` is a `Tree` object.

8.8.9 Example calculation

This is an example of a hierarchical clustering calculation, using single linkage clustering for genes and maximum linkage clustering for experimental conditions. As the Euclidean distance is being used for gene clustering, it is necessary to scale the node distances `genetree` such that they are all between zero and one. This is needed for the Java TreeView code to display the tree diagram correctly. To cluster the experimental conditions, the uncentered correlation is being used. No scaling is needed in this case, as the distances in `exptree` are already between zero and two. The example data `cyano.txt` can be found in the `data` subdirectory.

```
>>> from Pycluster import *
>>> handle = open("cyano.txt")
>>> record = read(handle)
>>> genetree = record.treecluster(method='s')
>>> genetree.scale()
>>> exptree = record.treecluster(dist='u', transpose=1)
>>> record.save("cyano_result", genetree, exptree)
```

This will create the files `cyano_result.cdt`, `cyano_result.gtr`, and `cyano_result.atr`.

Similarly, we can save a k-means clustering solution:

```
>>> from Pycluster import *
>>> handle = open("cyano.txt")
>>> record = read(handle)
>>> (geneclusters, error, ifound) = record.kcluster(nclusters=5, npass=1000)
>>> (expclusters, error, ifound) = record.kcluster(nclusters=2, npass=100,
transpose=1)
>>> record.save("cyano_result", geneclusters, expclusters)
```

This will create the files `cyano_result_K_G2_A2.cdt`, `cyano_result_K_G2.kgg`, and `cyano_result_K_A2.kag`.

9 Using the C Clustering Library with Perl: Algorithm::Cluster

Algorithm::Cluster is a Perl wrapper extension of the C Clustering Library written by John Nolan of the University of California, Santa Cruz.

Algorithm::Cluster requires Perl 5.6.0 at a minimum. It will not compile properly with 5.005_03 or previous versions or Perl. It has been tested on Win32, Mac OS X, Linux, OpenBSD and Solaris.

To install Algorithm::Cluster on UNIX, Linux, Mac OS X, or Cygwin, you can download the source code from `http://bonsai.hgc.jp/~mdehoon/software/cluster`. You will need an ANSI C compiler; GNU's free `gcc` compiler will do. Unpack the distribution and type the following familiar commands to compile, test and install the module:

```
perl Makefile.PL
make
make test
make install
```

You can also use the CPAN shell from within Perl to download the module and install it.

Algorithm::Cluster offers the following functions:

- `kcluster`
- `kmedoids`
- `treecluster`
- `somcluster`
- `clustercentroids`
- `clusterdistance`
- `distancematrix`
- `pca`
- `mean`
- `median`

You can also use Algorithm::Cluster to read expression data files and to write the clustering solution to a file; see below for details.

9.1 Using named parameters

Most of the interface functions in Algorithm::Cluster expect named parameters. This is implemented by passing a hash as a parameter to the function. For example, if a Perl function `sign_up()` accepts three named parameters, `name`, `program` and `class`, then you can invoke the function like this:

```
$return_value = sign_up(
        'name' => 'Mr. Smith',
        'program' => 'Biology',
        'class' => 'Intro to Molecular Biology',
);
```

When the function parses its parameters, it will create a hash, on the fly, with three keys. The function can access the values by referring to the hash.

This is convenient for several reasons. First, it means that you can pass the parameters in any order. Both invocations below are valid:

```
$return_value = sign _up(
        'class' => 'Intro to Molecular Biology',
        'name' => 'Ms. Jones',
        'program' => 'Biology',
);
$return_value = sign _up(
        'name' => 'Miss Chen',
        'program' => 'Biology',
        'class' => 'Intro to Molecular Biology',
);
```

If the function defines default values for parameters, you can also leave some parameters out, and the function will still know which parameter is which:

```
$return_value = sign_up(
        'name' => 'Ms. Jones',
);
```

You can define the hash on your own, and pass this to the function. This is useful if your function accepts a long list of parameters, and you intend to call it several times, (mostly) reusing the same values. You can implement your own defaults:

```
%student = (
        'name' => 'Mr. Smith',
        'program' => 'Biology',
        'class' => 'Intro to Molecular Biology',
);
$return_value = sign_up(%student);

$hash{student} = 'Ms. Jones';
$return_value = sign_up(%student);

$hash{student} = 'Miss Chen';
$return_value = sign_up(%student);
```

9.2 References to arrays, and two-dimensional matrices

Perl implements two-dimensional matrices using references. For example, a reference to a one-dimensional array (a row) can be defined like this:

```
$row = [ 1, 2 ];
```

In this example, $row itself is not an array, but a reference to the array (1, 2). (The square brackets indicate that we want to create a reference.) $row->[0] equals 1 and $row->[1] equals 2. A 3×2 matrix of integers can be defined like this:

```
$row0 = [ 1, 2 ];
$row1 = [ 3, 4 ];
$row2 = [ 5, 6 ];
$data = [ $row0, $row1, $row2 ];
```

Or, more succinctly:

```
 $data = [
     [ 1, 2 ],
     [ 3, 4 ],
     [ 5, 6 ],
];
```

In this example, `$data->[0]->[1]` equals 2, while `$data->[2]->[0]` equals 5.

Many of the functions available in Algorithm::Cluster expect data to be in the form of a two dimensional array, like the example above. Some functions also return references to data structures like this.

9.3 Partitioning algorithms

9.3.1 The *k*-means clustering algorithm: `kcluster`

The function `kcluster()` implements the *k*-means clustering algorithm. In the example invocation below, we have created `$param{data}` as an empty matrix, because that is the default value, but you must populate `$param{data}` with real data, in order to invoke `kcluster()` properly.

```
 my %param = (
         nclusters => 2,
         data => [[]],
         mask => '',
         weight => '',
         transpose => 0,
         npass => 10,
         method => 'a',
         dist => 'e',
         initialid => [],
);
my ($clusters, $error, $found) = kcluster(%param);
```

Arguments

- `data`
  A reference to a two-dimensional matrix containing the gene expression data, where genes are stored row-wise and microarray experiments column-wise.

- nclusters
  The number of clusters k.

- mask
  A reference to a two-dimensional matrix of integers showing which data are missing. If `$param{mask}->[i]->[j]==0`, then `$param{data}->[i]->[j]` is missing. If `mask` is `''` (i.e., the null string, and not a reference at all), then there are no missing data.

- weight
  A reference to an array containing the weights to be used when calculating distances. If `weight` equals `''` (i.e., the null string, and not a reference at all), then equal weights are assumed. If `transpose==0`, the length of this array must equal the number of columns in the data matrix. If `transpose==1`, the length of the `weight` array should equal the number of rows in the data matrix. If `weight` has a different length, the entire array will be ignored.

- transpose
  Determines if genes or microarrays are being clustered. If `$param{transpose}==0`, genes (rows) are being clustered. If `$param{transpose}==1`, microarrays (columns) are clustered.

- npass
  The number of times the k-means clustering algorithm is performed, each time with a different (random) initial condition. If the argument `initialid` is given, the value of `npass` is ignored and the clustering algorithm is run only once, as it behaves deterministically in that case.

- method
  A one-character flag, indicating how the center of a cluster is found:
    - `'a'`: arithmetic mean
    - `'m'`: median

  For any other values of method, the arithmetic mean is used.

- dist
  A one-character flag, defining the distance function to be used:
    - `'c'`: correlation
    - `'a'`: absolute value of the correlation
    - `'u'`: uncentered correlation
    - `'x'`: absolute uncentered correlation
    - `'s'`: Spearman's rank correlation
    - `'k'`: Kendall's τ;
    - `'e'`: Euclidean distance
    - `'b'`: City-block distance

  For other values of `dist`, the default (Euclidean distance) is used.

- initialid
  An optional parameter defining the initial clustering to be used for the EM algorithm. If `initialid` is not specified, then a different random initial clustering is used for each of the `npass` runs of the EM algorithm. If `initialid` is specified, then it should be

equal to a 1D array containing the cluster number (between `0` and `nclusters-1`) for each item. Each cluster should contain at least one item. With the initial clustering specified, the EM algorithm is deterministic.

Return values

This function returns a list of three items: *$clusterid*, *$error*, *$nfound*.

- *$clusterid*
  A reference to an array with a length equal to the number of rows in the data array. Each element in the *clusterid* array contains the number of the cluster to which each gene/microarray was assigned.
- *$error*
  The within-cluster sum of distances of the optimal clustering solution that was found.
- *$nfound*
  The number of times the optimal solution was found.

9.3.2 The k-medoids algorithm: `kmedoids`

The function `kmedoids()` implements the k-means clustering algorithm. In the example invocation below, we have created the distance matrix `$param{distances}` as an empty matrix, because that is the default value, but you must populate `$param{distances}` with real data, in order to invoke `kmedoids()` properly.

```
my %param = (
        nclusters => 2,
        distances => [[]],
        npass => 10,
        initialid => [],
);
my ($clusters, $error, $found) = kmedoids(%param);
```

Arguments

- `nclusters`
  The number of clusters k.
- `distances`
  A list containing the distance matrix between the elements. An example of a distance matrix is:
  `$distances = [[], [1.1], [1.0, 4.5], [2.3, 1.8, 6.1]];`

- `npass`
  The number of times the k-medoids clustering algorithm is performed, each time with a different (random) initial condition. If `initialid` is given, the value of *npass* is ignored, as the clustering algorithm behaves deterministically in that case.
- `initialid`
  Specifies the initial clustering to be used for the EM algorithm. If `initialid` is not specified, then a different random initial clustering is used for each of the `npass` runs of the EM algorithm. If `initialid` is specified, then it should be equal to a 1D array

containing the cluster number (between `0` and `nclusters-1`) for each item. Each cluster should contain at least one item. With the initial clustering specified, the EM algorithm is deterministic.

Return values

This function returns a list of three items: *$clusterid*, `$error`, `$nfound`.

- `$clusterid`
  $clusterid is a reference to an array with a length equal to the number of rows of the distance matrix. Each element in the *clusterid* array contains the number of the cluster to which each gene/microarray was assigned. The cluster number is defined as the number of the gene/microarray that is the centroid of the cluster.

- `$error`
  `$error` is the within-cluster sum of distances of the optimal clustering solution that was found.

- `$nfound`
  `$nfound` is the number of times the optimal solution was found.

9.4 Hierarchical clustering: `treecluster`

The pairwise single-, maximum-, average-, and centroid-linkage clustering methods are accessible through the function `treecluster`. The hierarchical clustering routines can be applied either on the original gene expression data, or (except for centroid-linkage clustering) on the distance matrix directly. The tree structure generated by `treecluster` can be cut in order to separate the elements into a given number of clusters.

9.4.1 Representing a hierarchical clustering solution

A hierarchical clustering solution is represented using two Perl classes: `Node` and `Tree`.

The class `Node`

The Perl class `Node` corresponds to the C struct `Node` described above. A `Node` object has three attributes:

- left
- right
- distance

Here, `left` and `right` are integers referring to the two items or subnodes that are joined at this node, and `distance` is the distance between them. The items being clustered are numbered from `0` to (*number of items* - 1), while clusters are numbered `-1` to `-(`*number of items*`-1)`.

To create a new `Node` object, we need to specify `left`, `right`, and `distance`:

```
use Algorithm::Cluster;
my $node = Algorithm::Cluster::Node->new(3, 4, 2.7);
print "Left:", $node->left, "\n";
print "Right:", $node->right, "\n";
print "Distance:", $node->distance, "\n";
```

Executing this code prints

```
Left:3
Right:4
Distance:2.7
```

The attributes `left`, `right`, and `distance` of an existing `Node` object can be modified by using the `set_left`, `set_right`, and `set_distance` methods:

```
use Algorithm::Cluster;
my $node = Algorithm::Cluster::Node->new(3, 4, 2.7);
$node->set_left(2);
$node->set_right(1);
$node->set_distance(2.1);
```

The class `Tree`

The Perl class `Tree` represents a full hierarchical clustering solution. A `Tree` object can be created from an array of `Node` objects:

```
use Algorithm::Cluster;
my $node1 = Algorithm::Cluster::Node->new(1,2,3.1);
my $node2 = Algorithm::Cluster::Node->new(-1,3,5.3);
my $node3 = Algorithm::Cluster::Node->new(4,0,5.9);
my $node4 = Algorithm::Cluster::Node->new(-2,-3,7.8);
my @nodes = [$node1,$node2,$node3,$node4];
my $tree = Algorithm::Cluster::Tree->new(@nodes);
```

The `Tree` initializer checks if the list of nodes is a valid hierarchical clustering result:

```
use Algorithm::Cluster;
my $node1 = Algorithm::Cluster::Node->new(1,2,0.3);
my $node2 = Algorithm::Cluster::Node->new(1,3,0.7);
my @nodes = [$node1,$node2];
my $tree = Algorithm::Cluster::Tree->new(@nodes);
```
results in the following error message:

```
the array of nodes passed to Algorithm::Cluster::Tree::new do not represent a
valid tree
```

Individual nodes in a `Tree` object can be accessed using the `get` method. Continuing the previous example, the following code prints `0.7`:

```
my $node3 = $tree->get(1);
print $node3->distance;
```

As a `Tree` object is read-only, we cannot change individual nodes in a `Tree` object. However, we can convert the tree to a list of nodes, modify this list, and create a new tree from this list:

```
use Algorithm::Cluster;
my $node1 = Algorithm::Cluster::Node->new(0,1,0.3);
my $node2 = Algorithm::Cluster::Node->new(2,3,0.7);
```

```
my $node3 = Algorithm::Cluster::Node->new(-1,-2,0.9);
my $nodes = [$node1, $node2, $node3];
my $tree = Algorithm::Cluster::Tree->new($nodes);
my $i;
my $n = $tree->length;
my $node;
my $nodes2 = [];
for ($i = 0; $i < $n; $i++) {
        $nodes2->[$i] = $tree->get($i);
}
$node = $nodes2->[1];
$node->set_left(-1);
$node = $nodes2->[2];
$node->set_left(2);
my $tree2 = Algorithm::Cluster::Tree->new($nodes2);
```

This guarantees that any `Tree` object is always well-formed.

A `Tree` object has three methods (`scale`, `sort`, and `cut`, described below). The `treecluster` function returns `Tree` objects.

9.4.2 Performing hierarchical clustering: `treecluster`

The pairwise single-, maximum-, average-, and centroid-linkage clustering methods are accessible through the function `treecluster`.

```
my %param = (
        data => [[]],
        mask => '',
        weight => '',
        transpose => 0,
        dist => 'e',
        method => 's',
);

my $tree = Algorithm::Cluster::treecluster(%param);
```

Arguments

- `data`
  A reference to a two-dimensional matrix containing the gene expression data, where genes are stored row-wise and microarray experiments column-wise. It is also possible to pass a user-defined distance matrix as a lower-diagonal matrix, for example:
  ```
  $data = [
          [],
          [ 3.4],
          [ 4.3, 10.1],
          [ 3.7, 11.5, 1.0]
          ];
  ```

If the `data` argument has this form, it will be interpreted as a distance matrix instead of a raw data matrix, and the arguments `mask`, `weight`, `transpose`, and `dist` will be ignored.

- `mask`
  A referene to a two-dimensional matrix of integers showing which data are missing. If `$param{mask}->[i]->[j]==0`, then `$param{data}->[i]->[j]` is missing. If `mask` is `''` (i.e., the null string, and not a reference at all), then there are no missing data.

- `weight`
  A reference to an array containing the weights to be used when calculating distances. If `weight` equals `''` (i.e., the null string, and not a reference at all), then equal weights are assumed. If `transpose==0`, the length of this array must equal the number of columns in the data matrix. If `transpose==1`, the length of the `weight` array should equal the number of rows in the data matrix. If `weight` has a different length, the entire array will be ignored.

- `transpose`
  Determines if genes or microarrays are being clustered. If `$param{transpose}==0`, genes (rows) are being clustered. If `$param{transpose}==1`, microarrays (columns) are clustered.

- `dist`
  A one-character flag, defining the distance function to be used:
  - `'c'`: correlation
  - `'a'`: absolute value of the correlation
  - `'u'`: uncentered correlation
  - `'x'`: absolute uncentered correlation
  - `'s'`: Spearman's rank correlation
  - `'k'`: Kendall's τ;
  - `'e'`: Euclidean distance
  - `'b'`: City-block distance

- `method`
  Specifies which type of hierarchical clustering is used:
  - `'s'`: pairwise single-linkage clustering
  - `'m'`: pairwise maximum- (or complete-) linkage clustering
  - `'a'`: pairwise average-linkage clustering
  - `'c'`: pairwise centroid-linkage clustering

  Pairwise centroid-linkage clustering is not available if a user-defined distance matrix is passed via `data`.

Return values

This function returns a `Tree` object. This object contains *number of items* - 1 nodes, where the number of items is the number of genes if genes were clustered, or the number of microarrays if microarrays were clustered. Each node describes a pairwise linking event, where the node attributes `left` and `right` each contain the number of one gene/microarray or subnode, and `distance` the distance between them. Genes/microarrays are numbered

from 0 to (`number of items` - 1), while clusters are numbered -1 to -(`number of items`-1).

9.4.3 Scaling a hierarchical clustering tree: `$tree->scale`

To display a hierarchical clustering solution with Java TreeView, it is better to scale all node distances such that they are between zero and one. This can be accomplished by calling the `scale` method on an existing `Tree` object.

`$tree->scale`
scales the distances in the hierarchical clustering tree such that they are all between zero and one. This function takes no arguments.

9.4.4 Sorting a hierarchical clustering tree: `$tree->sort`

`my @indices = $tree->sort($order)`
sorts the hierarchical clustering tree such that the elements tend to increase in the order value from left to right in the dendrogram, where the order value of each element is specified by the user. This is accomplished by visting each node in the hierarchical clustering tree, verifying if the average order value of the left node is less than or equal to the average order value of the right node, and switching the left and right node otherwise. The method returns the indices of the elements in the left-to-right order after sorting.

Arguments

- `$tree`
  The `Tree` object `$tree` contains the hierarchical clustering result generated by `treecluster`.

- `$order`
  The order values of the elements. If `$order` is not specified, the tree is not reordered, and the `@indices` of the current tree are calculated.

Return values

This function returns the array `@indices`.

- `@indices`
  An array indicating the order of the elements in the dendogram after sorting, with element `$indices[$i]` occurring at position i in the left-to-right order in the dendogram.

9.4.5 Cutting a hierarchical clustering tree: `$tree->cut`

`my @clusterid = $tree->cut($nclusters)`
groups the items into `$nclusters` clusters based on the tree structure generated by the hierarchical clustering routine `treecluster`.

Arguments

- `$tree`
  The `Tree` object `$tree` contains the hierarchical clustering result generated by `treecluster`.

- `$nclusters`
  The desired number of clusters; `$nclusters` should be positive, and less than or equal

to the number of elements. If $nclusters$ is not specified, the number of elements is used as the default value.

Return values

This function returns the array `@clusterid`.

- `@clusterid`
  Array containing the number of the cluster to which each gene/microarray is assigned. Clusters are numbered from 0 through $nclusters$-1 in the left-to-right order in which they appear in the hierarchical clustering tree.

Example

```
use Algorithm::Cluster;
my $node1 = Algorithm::Cluster::Node->new(0,1,0.3);
my $node2 = Algorithm::Cluster::Node->new(2,3,0.7);
my $node3 = Algorithm::Cluster::Node->new(-1,-2,0.9);
my $nodes = [$node1, $node2, $node3];
my $tree = Algorithm::Cluster::Tree->new($nodes);
my @a = $tree->cut(3);
my $i = 0;
foreach (@a) {
        print "Element $i: Cluster $_\n";
        $i++;
}
 prints
Element 0: Cluster 0
Element 1: Cluster 0
Element 2: Cluster 1
Element 3: Cluster 2
```

9.5 Self-Organizing Maps: `somcluster`

The `somcluster()` function implements a Self-Organizing Map on a rectangular grid.

```
 my %param = (
        data => [[]],
        mask => '',
        weight => '',
        transpose => 0,
        nxgrid => 10,
        nygrid => 10,
        niter => 100,
        dist => 'e',
);
my ($clusterid) = Algorithm::Cluster::somcluster(%param);
```

Arguments

- `data`
  A reference to a two-dimensional matrix containing the gene expression data, where genes are stored row-wise and microarray experiments column-wise.

- `mask`
  A reference to a two-dimensional matrix of integers showing which data are missing. If `$param{mask}->[i]->[j]==0`, then `$param{data}->[i]->[j]` is missing. If `mask` is `''` (i.e., the null string, and not a reference at all), then there are no missing data.

- `weight`
  A reference to an array containing the weights to be used when calculating distances. If `weight` equals `''` (i.e., the null string, and not a reference at all), then equal weights are assumed. If `transpose==0`, the length of this array must equal the number of columns in the data matrix. If `transpose==1`, the length of the `weight` array should equal the number of rows in the data matrix. If `weight` has a different length, the entire array will be ignored.

- `transpose`
  Determines if genes or microarrays are being clustered. If `$param{transpose}==0`, genes (rows) are being clustered. If `$param{transpose}==1`, microarrays (columns) are clustered.

- `nxgrid, nygrid`
  Both parameters are integers, indicating the number of cells horizontally and vertically in the rectangular grid, on which the Self-Organizing Map is calculated.

- `inittau`
  The initial value for the neighborhood function, as given by the parameter τ. The default value for `inittau` is 0.02, which was used in Michael Eisen's Cluster/TreeView program.

- `niter`
  The number of iterations to be performed.

- `dist`
  A one-character flag, defining the distance function to be used:
    - `'c'`: correlation
    - `'a'`: absolute value of the correlation
    - `'u'`: uncentered correlation
    - `'x'`: absolute uncentered correlation
    - `'s'`: Spearman's rank correlation
    - `'k'`: Kendall's τ;
    - `'e'`: Euclidean distance
    - `'b'`: City-block distance

Return values

This function returns one value, *$clusterid*, which is a reference to a two-dimensional matrix. If `$param{transpose}==0`, then the number of rows in *$clusterid* equals the number of rows (genes) in the original data array; if `$param{transpose}==1`, then then the number of rows

in *$clusterid* equals the number of columns (microarrays) in the original data array. Each row in the array *clusterid* contains the x and y coordinates of the cell in the rectangular SOM grid to which the gene or microarray was assigned.

9.6 Finding the cluster centroids: `clustercentroids`

```
my %param = (
        data => [[]],
        mask => '',
        clusterid => undef,
        method => 'a',
        transpose => 0,
);
my ($cdata, $cmask) = clustercentroids(%param)
```
calculates the cluster centroids.

Arguments

- `data`
  Array containing the gene expression data, where genes are stored row-wise and microarray experiments column-wise.

- `mask`
  Array of integers showing which data are missing. If `$param{mask}->[$i]->[$j]==0`, then `$param{data}->[$i]->[$j]` is missing. If `mask` is '', then there are no missing data.

- `clusterid`
  Vector of integers showing to which cluster each element belongs. If `clusterid` is not given, then all elements are assumed to belong to the same cluster.

- `method`
  Specifies whether the arithmetic mean (`method=>'a'`) or the median (`method=>'m'`) is used to calculate the cluster center.

- `transpose`
  Determines if gene or microarray clusters are being considered. If `transpose==0`, then we are considering clusters of genes (rows). If `transpose==1`, then we are considering clusters of microarrays (columns).

Return values

This function returns the list *cdata, cmask*.

- *$cdata*
  A 2D array containing the centroid data. The dimensions of this array are (*number of clusters, number of microarrays*) if genes were clustered, or (*number of genes, number of clusters*) if microarrays were clustered. Each row (if genes were clustered) or column (if microarrays were clustered) contains the averaged gene expression data for corresponding to the centroid of one cluster that was found.

- *$cmask*
  This matrix stores which values in *$cdata* are missing. If *$cmask*[$i][$j]==0, then *$cdata*[$i][$j] is missing. The dimensions of this array are (*number of clusters,*

number of microarrays) if genes were clustered, or (*number of genes, number of clusters*) if microarrays were clustered.

9.7 The distance between two clusters: `clusterdistance`

The `clusterdistance` routine calculates the distance between two clusters, between a cluster and an item, or between two items.

```
my %param = (
        data => [[]],
        mask => '',
        weight => '',
        cluster1 => [],
        cluster2 => [],
        dist => 'e',
        method => 'a',
        transpose => 0,
);
```

```
my ($distance) = Algorithm::Cluster::clusterdistance(%param);
```

Arguments

- `data`
  A reference to a two-dimensional matrix containing the gene expression data, where genes are stored row-wise and microarray experiments column-wise.

- `mask`
  A reference to a two-dimensional matrix of integers showing which data are missing. If `$param{mask}->[i]->[j]==0`, then `$param{data}->[i]->[j]` is missing. If `mask` is `''` (i.e., the null string, and not a reference at all), then there are no missing data.

- `weight`
  A reference to an array containing the weights to be used when calculating distances. If `weight` equals `''` (i.e., the null string, and not a reference at all), then equal weights are assumed. If `transpose==0`, the length of this array must equal the number of columns in the data matrix. If `transpose==1`, the length of the `weight` array should equal the number of rows in the data matrix. If `weight` has a different length, the entire array will be ignored.

- `cluster1`
  contains the indices of the elements belonging to the first cluster, or alternatively an integer to refer to a single item.

- `cluster2`
  contains the indices of the elements belonging to the second cluster, or alternatively an integer to refer to a single item.

- `dist`
  A one-character flag, defining the distance function to be used:
    - `'c'`: correlation
    - `'a'`: absolute value of the correlation

- 'u': uncentered correlation
- 'x': absolute uncentered correlation
- 's': Spearman's rank correlation
- 'k': Kendall's τ;
- 'e': Euclidean distance
- 'b': City-block distance

- method
  A one-character flag, indicating how the center of a cluster is found:
  - 'a': Distance between the two cluster centroids (arithmetic mean)
  - 'm': Distance between the two cluster centroids (median)
  - 's': Shortest distance between elements in the two clusters
  - 'x': Longest pairwise distance between elements in the two clusters
  - 'v': Average over the pairwise distances between elements in the two clusters.

  For any other values of method, the arithmetic mean is used.

- transpose
  Determines if the distance between genes or between microarrays should be
  calculated. If $param{transpose}==0, the function calculates the distance between
  the genes (rows) specified by $param{cluster1} and $param{cluster2}. If
  $param{transpose}==1, the function calculates the distance between the microarrays
  (columns) specified by $param{cluster1} and $param{cluster2}.

Return values

The distance between the clusters indicated by cluster1 and cluster2.

9.8 Calculating the distance matrix: distancematrix

The function distancematrix() calculates the distance matrix between the gene expression
data and returns it as a ragged array.

```
my %param = (
        data => [[]],
        mask => '',
        weight => '',
        transpose => 0,
        dist => 'e',
);
my $distancematrix = distancematrix(%param);
```

Arguments

- data
  A reference to a two-dimensional matrix containing the gene expression data, where
  genes are stored row-wise and microarray experiments column-wise.

- mask
  A reference to a two-dimensional matrix of integers showing which data are missing. If

$param{mask}->[i]->[j]==0, then $param{data}->[i]->[j] is missing. If mask is '' (i.e., the null string, and not a reference at all), then there are no missing data.

- weight
  A reference to an array containing the weights to be used when calculating distances. If weight equals '' (i.e., the null string, and not a reference at all), then equal weights are assumed. If transpose==0, the length of this array must equal the number of columns in the data matrix. If transpose==1, the length of the weight array should equal the number of rows in the data matrix. If weight has a different length, the entire array will be ignored.

- transpose
  Determines if the distances between genes or microarrays should be calculated. If $param{transpose}==0, the distances between genes (rows) are calculated. If $param{transpose}==1, the distances are calculated between microarrays (columns).

- dist
  A one-character flag, defining the distance function to be used:
  - 'c': correlation
  - 'a': absolute value of the correlation
  - 'u': uncentered correlation
  - 'x': absolute uncentered correlation
  - 's': Spearman's rank correlation
  - 'k': Kendall's τ;
  - 'e': Euclidean distance
  - 'b': City-block distance For other values of dist, the default (Euclidean distance) is used.

Return values

This function returns the *$distancematrix*, an array of rows containing the distance matrix between the gene expression data. The number of columns in each row is equal to the row number. Hence, the first row has zero elements.

9.9 Principal Component Analysis: pca

my ($columnmean, $coordinates, $components, $eigenvalues) = pca($data);
applies Principal Component Analysis to the rectangular matrix data.

Arguments

- $data
  A reference to a two-dimensional matrix containing the gene expression data, where genes are stored row-wise and microarray experiments column-wise.

Return values

This function returns a list of four items: *$columnmean*, *$coordinates*, *$components*, *$eigenvalues*.

- *$columnmean*
  A reference to an array containing the mean over each column in $data.

- *$coordinates*
  A reference to an array containing the coordinates of each row in `$data` with respect to the principal components.
- *$components*
  A reference to an array containing the principal components.
- *$eigenvalues*
  A reference to an array containing the eigenvalues corresponding to each of the principal components.

9.10 Auxiliary functions

`median($data)`
Returns the median of the data. `$data` is a reference to a (one-dimensional) array of numbers.

`mean($data)`
Returns the mean of the data. `$data` is a reference to a (one-dimensional) array of numbers.

`version()`
Returns the version number of the C Clustering Library as a string.

9.11 Handling Cluster/TreeView-type files

Cluster/TreeView are GUI-based codes for clustering gene expression data. They were originlly written by Michael Eisen (`http://rana.lbl.gov`) while at Stanford University. Algorithm::Cluster contains functions for reading and writing data files in the format specifies for Cluster/TreeView. In particular, by saving a clustering result in that format, we can use TreeView to visualize the clustering results. We recommend using Alok Saldanha's Java TreeView program (`http://jtreeview.sourceforget.net`), which can display hierarchical as well as *k*-means clustering clustering results.

9.11.1 The `Record` class

An object of the class `Record` contains all information stored in a Cluster/TreeView-type data file. To store the information contained in the data file in a `Record` object, we create a `Record` object, open the file, and then read the file contents into the `Record` object:

```
use Algorithm::Cluster::Record;
my $record = Algorithm::Cluster::Record->new();
open INPUT, "mydatafile.txt";
$record->read(*INPUT);
```

The `read` command reads the tab-delimited text file `mydatafile.txt` containing gene expression data in the format specified for Michael Eisen's Cluster/TreeView program. For a description of this file format, see the manual to Cluster/TreeView. It is available at Michael Eisen's lab website (`http://rana.lbl.gov/manuals/ClusterTreeView.pdf`) and at our website (`http://bonsai.hgc.jp/~mdehoon/software/cluster/cluster3.pdf`).

A `Record` object stores the following information:

- `data`
  The data array containing the gene expression data. Genes are stored row-wise, while microarrays are stored column-wise.
- `mask`
  This array shows which elements in the `data` array, if any, are missing. If $record\{mask\}[\$i][\$j]==0$, then $record\{data\}[\$i][\$j]$ is missing. If no data were found to be missing, $record\{mask\}$ remains `undef`.
- `geneid`
  This is a list containing a unique description for each gene (i.e., ORF numbers).
- `genename`
  This is a list containing a description for each gene (i.e., gene name). If not present in the data file, $record\{genename\}$ remains `undef`.
- `gweight`
  The weights that are to be used to calculate the distance in expression profile between genes. If not present in the data file, $record\{gweight\}$ remains `undef`.
- `gorder`
  The preferred order in which genes should be stored in an output file. If not present in the data file, $record\{gorder\}$ remains `undef`.
- `expid`
  This is a list containing a description of each microarray, e.g. experimental condition.
- `eweight`
  The weights that are to be used to calculate the distance in expression profile between microarrays. If not present in the data file, $record\{eweight\}$ remains `undef`.
- `eorder`
  The preferred order in which microarrays should be stored in an output file. If not present in the data file, $record\{eorder\}$ remains `undef`.
- `uniqid`
  The string that was used instead of UNIQID in the data file.

After loading a `Record` object, each of these attributes can be accessed and modified directly. For example, the data can be log-transformed by taking the logarithm of `record->data`.

9.11.2 Performing hierarchical clustering

```
my %param = (
        transpose => 0,
        method => 'a',
        dist => 'e',
);
my $tree = $record->treecluster(%param);
```
applies hierachical clustering to the data contained in the `Record` object.

Arguments

- `transpose`
  Determines if genes or microarrays are being clustered. If `transpose==0`, genes (rows) are being clustered. If `transpose==1`, microarrays (columns) are clustered.

- `method`
  defines the linkage method to be used:
    - `method=='s'`: pairwise single-linkage clustering
    - `method=='m'`: pairwise maximum- (or complete-) linkage clustering
    - `method=='c'`: pairwise centroid-linkage clustering
    - `method=='a'`: pairwise average-linkage clustering
- `dist`
  defines the distance function to be used:
    - `dist=='c'`: correlation;
    - `dist=='a'`: absolute value of the correlation;
    - `dist=='u'`: uncentered correlation;
    - `dist=='x'`: absolute uncentered correlation;
    - `dist=='s'`: Spearman's rank correlation;
    - `dist=='k'`: Kendall's τ;
    - `dist=='e'`: Euclidean distance;
    - `dist=='b'`: City-block distance.

Return values

This function returns a `Tree` object. This object contains *number of items* − 1 nodes, where the number of items is the number of genes if genes were clustered, or the number of microarrays if microarrays were clustered. Each node describes a pairwise linking event, where the node attributes `left` and `right` each contain the number of one gene/microarray or subnode, and `distance` the distance between them. Genes/microarrays are numbered from 0 to (*number of items* − 1), while clusters are numbered −1 to −(*number of items*−1).

9.11.3 Performing *k*-means or *k*-medians clustering

```
my %param = (
        nclusters => 2,
        transpose => 0,
        npass => 1,
        method => 'a',
        dist => 'e',
        initialid => undef,
);
my ($clusterid, $error, $nfound) = $record->kcluster(%param);
```

applies *k*-means or *k*-medians clustering to the data contained in the `Record` object.

Arguments

- `nclusters`
  The number of clusters *k*.

- `transpose`
  Determines if genes or microarrays are being clustered. If `transpose==0`, genes (rows) are being clustered. If `transpose==1`, microarrays (columns) are clustered.

- `npass`
  The number of times the k-means clustering algorithm is performed, each time with a different (random) initial condition. If `initialid` is given, the value of `npass` is ignored and the clustering algorithm is run only once, as it behaves deterministically in that case.
- `method`
  describes how the center of a cluster is found:
    - `method=='a'`: arithmetic mean;
    - `method=='m'`: median.

  For other values of `method`, the arithmetic mean is used.
- `dist`
  defines the distance function to be used:
    - `dist=='c'`: correlation;
    - `dist=='a'`: absolute value of the correlation;
    - `dist=='u'`: uncentered correlation;
    - `dist=='x'`: absolute uncentered correlation;
    - `dist=='s'`: Spearman's rank correlation;
    - `dist=='k'`: Kendall's τ;
    - `dist=='e'`: Euclidean distance;
    - `dist=='b'`: City-block distance.
- `initialid`
  Specifies the initial clustering to be used for the EM algorithm. If `initialid==None`, then a different random initial clustering is used for each of the `npass` runs of the EM algorithm. If `initialid` is not `None`, then it should be equal to a 1D array containing the cluster number (between `0` and `nclusters-1`) for each item. Each cluster should contain at least one item. With the initial clustering specified, the EM algorithm is deterministic.

Return values

This function returns three variables (*$clusterid*, *$error*, *$nfound*).

- *$clusterid*
  An array containing the number of the cluster to which each gene/microarray was assigned.
- *$error*
  The within-cluster sum of distances for the optimal clustering solution.
- *$nfound*
  The number of times the optimal solution was found.

9.11.4 Calculating a Self-Organizing Map

```
my %param = (
        transpose => 0,
        nxgrid => 2,
```

```
        nygrid => 1,
        inittau => 0.02,
        niter => 1,
        dist => 'e',
);
my @clusterid = $record->somcluster(%param);
```
calculates a Self-Organizing Map on a rectangular grid, using the gene expression data in
the `Record` object *$record*.

Arguments

- `transpose`
  Determines if genes or microarrays are being clustered. If `transpose==0`, genes (rows)
  are being clustered. If `transpose==1`, microarrays (columns) are clustered.

- `nxgrid, nygrid`
  The number of cells horizontally and vertically in the rectangular grid, on which the
  Self-Organizing Map is calculated.

- `inittau`
  The initial value for the parameter τ that is used in the SOM algorithm. The default
  value for `inittau` is 0.02, which was used in Michael Eisen's Cluster/TreeView program.

- `niter`
  The number of iterations to be performed.

- `dist`
  defines the distance function to be used:
    - `dist=='c'`: correlation;
    - `dist=='a'`: absolute value of the correlation;
    - `dist=='u'`: uncentered correlation;
    - `dist=='x'`: absolute uncentered correlation;
    - `dist=='s'`: Spearman's rank correlation;
    - `dist=='k'`: Kendall's τ;
    - `dist=='e'`: Euclidean distance;
    - `dist=='b'`: City-block distance.

Return values

This function returns an array with two columns, where the number of rows is equal to the
number of genes or the number of microarrays depending on whether genes or microarrays
are being clustered. Each row contains the x and y coordinates of the cell in the rectangular
SOM grid to which the gene or microarray was assigned.

9.11.5 Finding the cluster centroid

```
my %param = (
        clusterid => undef,
        method => 'a',
        transpose => 0,
);
```

my (*$cdata*, *$cmask*) = *record*->clustercentroids(%param);
calculates the cluster centroids.

Arguments

- clusterid
  Vector of integers showing to which cluster each element belongs. If clusterid is not given, then all elements are assumed to belong to the same cluster.

- method
  Specifies whether the arithmetic mean (method=='a') or the median (method=='m') is used to calculate the cluster center.

- transpose
  Determines if gene or microarray clusters are being considered. If transpose==0, then we are considering clusters of genes (rows). If transpose==1, then we are considering clusters of microarrays (columns).

Return values

This function returns the list *$cdata*, *$cmask*.

- *$cdata*
  A 2D array containing the centroid data. The dimensions of this array are (*number of clusters*, *number of microarrays*) if genes were clustered, or (*number of genes*, *number of clusters*) if microarrays were clustered. Each row (if genes were clustered) or column (if microarrays were clustered) contains the averaged gene expression data for corresponding to the centroid of one cluster that was found.

- *$cmask*
  This matrix stores which values in *cdata* are missing. If *$cmask*[$i][$j]==0, then *$cdata*[$i][$j] is missing. The dimensions of this array are (*number of clusters*, *number of microarrays*) if genes were clustered, or (*number of genes*, *number of clusters*) if microarrays were clustered.

9.11.6 Calculating the distance between two clusters

```
 my %param = (
        index1 => [],
        index2 => [],
        method => 'a',
        dist => 'e',
        transpose => 0,
);
```
my *$distance* = *record*->clusterdistance(%param);
calculates the distance between two clusters.

Arguments

- index1
  is a list containing the indices of the elements belonging to the first cluster, or alternatively an integer to refer to a single item.

- `index2`
  is a list containing the indices of the elements belonging to the second cluster, or alternatively an integer to refer to a single item.
- `method`
  Specifies how the distance between clusters is defined:
    - `method=='a'`: Distance between the two cluster centroids (arithmetic mean);
    - `method=='m'`: Distance between the two cluster centroids (median);
    - `method=='s'`: Shortest pairwise distance between elements in the two clusters;
    - `method=='x'`: Longest pairwise distance between elements in the two clusters;
    - `method=='v'`: Average over the pairwise distances between elements in the two clusters.
- `dist`
  defines the distance function to be used:
    - `dist=='c'`: correlation;
    - `dist=='a'`: absolute value of the correlation;
    - `dist=='u'`: uncentered correlation;
    - `dist=='x'`: absolute uncentered correlation;
    - `dist=='s'`: Spearman's rank correlation;
    - `dist=='k'`: Kendall's τ;
    - `dist=='e'`: Euclidean distance;
    - `dist=='b'`: City-block distance.
- `transpose`
  Determines if gene or microarray clusters are being considered. If `transpose==0`, then we are considering clusters of genes (rows). If `transpose==1`, then we are considering clusters of microarrays (columns).

Return values

This function returns the distance between the two clusters.

9.11.7 Calculating the distance matrix

```
my %param = (
        transpose => 0,
        dist => 'e',
);
my $matrix = record->distancematrix(%param);
```
returns the distance matrix between gene expression data.

Arguments

- `transpose`
  Determines if genes or microarrays are being clustered. If `transpose==0`, genes (rows) are being clustered. If `transpose==1`, microarrays (columns) are clustered.
- `dist`
  defines the distance function to be used:
    - `dist=='c'`: correlation;

- `dist=='a'`: absolute value of the correlation;
- `dist=='u'`: uncentered correlation;
- `dist=='x'`: absolute uncentered correlation;
- `dist=='s'`: Spearman's rank correlation;
- `dist=='k'`: Kendall's τ;
- `dist=='e'`: Euclidean distance;
- `dist=='b'`: City-block distance.

Return values

This function returns the *$distancematrix*, an array of rows containing the distance matrix between the gene expression data. The number of columns in each row is equal to the row number. Hence, the first row has zero elements.

9.11.8 Saving the clustering result

```
my %param = (
        jobname => '',
        geneclusters => [],
        expclusters => [],
);
$record->save(%param);
```

writes the text file *jobname*.`cdt`, *jobname*.`gtr`, *jobname*.`atr`, *jobname**.`kgg`, and/or *jobname**.`kag` for subsequent reading by the Java TreeView program. If `geneclusters` and `expclusters` are both `None`, this method only writes the text file *jobname*.`cdt`; this file can subsequently be read into a new `Record` object.

Arguments

- `jobname`
  The string `jobname` is used as the base name for names of the files that are to be saved.
- `geneclusters`
  This argument describes the gene clustering result. In case of k-means clustering, this is a 1D array containing the number of the cluster each gene belongs to. It can be calculated using `kcluster`. In case of hierarchical clustering, `geneclusters` is a `Tree` object.
- `expclusters`
  This argument describes the clustering result for the experimental conditions. In case of k-means clustering, this is a 1D array containing the number of the cluster each experimental condition belongs to. It can be calculated using `kcluster`. In case of hierarchical clustering, `expclusters` is a `Tree` object.

9.11.9 Example calculation

This is an example of a hierarchical clustering calculation, using single linkage clustering for genes and maximum linkage clustering for experimental conditions. As the Euclidean distance is being used for gene clustering, it is necessary to scale the node distances `genetree` such that they are all between zero and one. This is needed for the Java TreeView code to display the tree diagram correctly. To cluster the experimental conditions, the uncentered

correlation is being used. No scaling is needed in this case, as the distances in `exptree` are already between zero and two. The example data `cyano.txt` can be found in the `data` subdirectory.

```
use Algorithm::Cluster::Record;
my $record = Algorithm::Cluster::Record->new();
open INPUT, "cyano.txt";
$record->read(*INPUT);
my $genetree = $record->treecluster(method=>'s');
my $exptree = $record->treecluster(dist=>'u', transpose=>1);
$record->save(jobname=>"cyano_result", geneclusters=>$genetree,
expclusters=>$exptree);
```

This will create the files `cyano_result.cdt`, `cyano_result.gtr`, and `cyano_result.atr`.

Similarly, we can save a k-means clustering solution:

```
use Algorithm::Cluster::Record;
my $record = Algorithm::Cluster::Record->new();
open INPUT, "cyano.txt";
$record->read(*INPUT);
my ($geneclusters, $error, $ifound) = $record->kcluster(nclusters=>5,
npass=>1000);
my ($expclusters, $error, $ifound) = $record->kcluster(nclusters=>2,
npass=>100, transpose=>1);
$record->save(jobname=>"cyano_result", geneclusters=>$geneclusters,
expclusters=>$expclusters);
```

This will create the files `cyano_result_K_G2_A2.cdt`, `cyano_result_K_G2.kgg`, and `cyano_result_K_A2.kag`.

10 Compiling and linking

In the instructions below, `<version>` refers to the version number. The C Clustering Library complies with the ANSI-C standard since version 1.04. As of version 1.06, the C Clustering Library makes use of `autoconf`/`automake` to make the installation process easier. To install the library for use with Python, use the `setup.py` script instead, as described below. To install the library for use with Perl, use the `Makefile.PL` script.

10.1 Installing the C Clustering Library for Python

Pycluster is available as part of the Biopython distribution and as a separate package. As of version 1.41, Pycluster uses the "new" Numerical Python (version 1.1.1 or later), which you should install before installing Pycluster.

To install Pycluster as a separate package, download `Pycluster-<version>.tar.gz` from `http://bonsai.hgc.jp/~mdehoon/software/cluster`. Unpack this file:
`gunzip Pycluster-<version>.tar.gz`
`tar -xvf Pycluster-<version>.tar`
and change to the directory `Pycluster-<version>`. Type
`python setup.py install`
from this directory. This will compile the library and install it for use with Python. To test your installation, you can run `python setup.py test`
If the installation was successful, you can remove the directory `Pycluster-<version>`. For Python on Windows (run from a DOS command window, or with a graphical user interface such as IDLE, PyCrust, PyShell, or PythonWin), a binary installer is available from `http://bonsai.hgc.jp/~mdehoon/software/cluster`.

Installation instructions for Biopython are available from the Biopython website (`http://www.biopython.org`).

10.2 Installing the C Clustering Library for Perl

To install the C Clustering Library for Perl, download `Algorithm-Cluster-<version>.tar.gz` from `http://bonsai.hgc.jp/~mdehoon/software/cluster` or from CPAN. Next, unpack this file with
`gunzip Algorithm-Cluster-<version>.tar.gz`
`tar -xvf Algorithm-Cluster-<version>.tar`
and change to the directory `Algorithm-Cluster-<version>`. Type
`perl Makefile.PL`
which will create a Makefile. To compile and install, type
`make`
`make install`
from this directory. You can execute
`make test`
to run some scripts that test the Algorithm::Cluster module. Some example Perl scripts can be found in the `perl/examples` subdirectory. If the installation was successful, you can remove the directory `Algorithm-Cluster-<version>`.

10.3 Accessing the C Clustering Library from C/C++

To call the routines in the C Clustering Library from your own C or C++ program, simply collect the relevant source files and compile them together with your program. The figure below shows the dependency structure for the source files in the C Clustering Library.
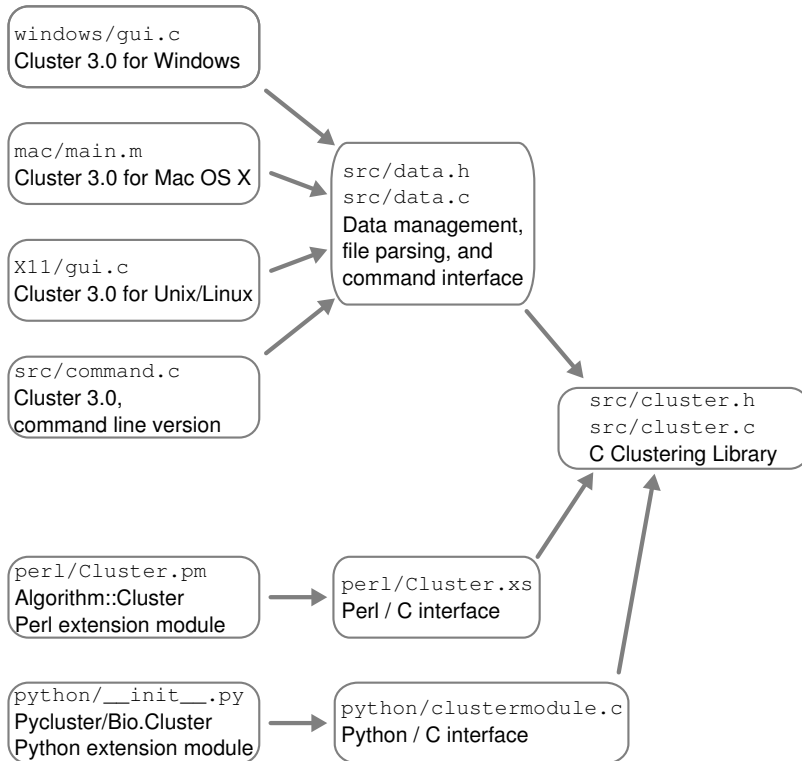
To use the routines in the C Clustering Library, put
```
#include <cluster.h>
```
in your source code. If your program is written in C++, use
```
extern "C" {
#include <cluster.h>
}
```
instead. To compile a C or C++ program with the C Clustering Library, add the relevant source files to the compile command. For example, a C program `myprogram.c` can be compiled and linked by
```
gcc -o myprogram myprogram.c cluster.c
```
An example C program that makes use of the C Clustering Library can be found in the `example` subdirectory.

10.4 Installing Cluster 3.0 for Windows

The easiest way to install Cluster 3.0 for Windows is to use the Windows installer (`http://bonsai.hgc.jp/~mdehoon/software/cluster`). The executable `cluster.com` can be used both as a GUI and as a command line program. To start Cluster 3.0 as a GUI program, simply double-click on `cluster.com`. If you want to use Cluster 3.0 from the

command prompt as a command line program, you may need to give the full path to the executable (e.g., `C:\Program Files\Stanford University\Cluster 3.0\cluster.com`; the exact path may be different on your computer). Type `cluster.com --help` for an overview of all command line options.

If you want to compile Cluster 3.0 from the source, change to the `windows` directory, and type `make`. This will compile the C Clustering Library, the Cluster 3.0 GUI, the Windows help files and the documentation. To compile the GUI, you need an ANSI C compiler such as GNU `gcc`. To compile the resources needed for the GUI, you will need the GNU program `windres`. To generate the help files, you need the HTML Help SDK, which can be downloaded from Microsoft. You will also need GNU makeinfo.

To generate the Windows installer, type
```
make clustersetup.exe
```
For this, you will need the Inno Setup Compiler, which can be downloaded from `http://www.jrsoftware.org`.

10.5 Installing Cluster 3.0 for Mac OS X

Cluster 3.0 for Mac OS X can be installed most easily by using the prebuilt package that is available at `http://bonsai.hgc.jp/~mdehoon/software/cluster`. After installing, you can start Cluster 3.0 as a GUI program by double-clicking on its icon. To run Cluster 3.0 as a command line program (e.g., from the Terminal), most likely you will need to give the full path to the executable (e.g., `/Applications/Cluster.app/Contents/MacOS/Cluster`). If you want to recompile Cluster 3.0, it is easiest to use the Project Builder and Interface Builder that are part of Mac OS X. The directory `mac` contains the project file that was used.

10.6 Installing Cluster 3.0 for Linux/Unix

Cluster 3.0 was ported to Linux/Unix using the Motif libraries. Motif is installed on most Linux/Unix computers. You will need a version compliant with Motif 2.1, such as Open Motif (`http://www.opengroup.org`), which is available at `http://www.motifzone.net`. Currently, LessTif (`http://www.lesstif.org`) does not work correctly with Cluster 3.0.

To install Cluster 3.0 on Linux/Unix, type
```
./configure
make
make install
```
This will create the executable `cluster` and install it in `/usr/local/bin`. Some auxiliary files will be installed in `/usr/local/cluster`. The executable can be used both as a GUI program and as a command line program. Type
```
cluster --help
```
for more information about running Cluster 3.0 as a command line program.

10.7 Installing Cluster 3.0 as a command line program

Cluster 3.0 can also be installed without GUI support. In this case, Cluster 3.0 can only be run as a command line program, in which the action taken by the program depends on the command line parameters. To install Cluster 3.0 as a command line program, the Motif

libraries are not needed. Simply download the source code for the C Clustering Library from our website `http://bonsai.hgc.jp/~mdehoon/software/cluster`, unpack and untar the file, and change to the directory `cluster-<version>`. Then type

`./configure --without-x`

`make`

`make install`

For the last step, you may need superuser privileges. For more information about the command line options, check the Cluster 3.0 manual.

Bibliography

Brown, P. O., and Botstein, D. (1999). Exploring the new world of the genome with DNA microarrays. *Nature Genetics* **21** (Supplement 1), 33–37.

De Hoon, M. J. L., Imoto, S., and Miyano, S. (2002). Statistical analysis of a small set of time-ordered gene expression data using linear splines. *Bioinformatics* **18** (11), 1477–1485.

De Hoon, M. J. L., Imoto, S., Nolan, J., and Miyano, S. (2004). Open source clustering software. *Bioinformatics*, **20** (9), 1453–1454.

Eisen, M. B., Spellman, P. T., Brown, P. O., and Botstein, D. (1998). Cluster analysis and display of genome-wide expression patterns. *Proceedings of the National Academy of Science USA* **95** (25), 14863–14868.

Golub, G. H. and Reisch, C. (1971). Singular value decomposition and least squares solutions. In *Handbook for Automatic Computation*, **2**, (Linear Algebra) (J. H. Wilkinson and C. Reinsch, eds), 134–151. New York: Springer-Verlag.

Hartigan, J. A. (1975). *Clustering algorithms* (New York: Wiley).

Jain, A. K. and Dubes, R. C. (1988). *Algorithms for clustering data* (Englewood Cliffs, N.J.: Prentice Hall).

Kachitvichyanukul, V. and Schmeiser, B. W. (1988). Binomial Random Variate Generation. *Communications of the ACM* **31** (2), 216–222.

Kohonen, T. (1997). *Self-organizing maps*, 2nd Edition (Berlin; New York: Springer-Verlag).

L'Ecuyer, P. (1988) Efficient and Portable Combined Random Number Generators. *Communications of the ACM* **31** (6), 742–749,774.

Sibson, R. (1973). SLINK: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, **16** (1), 30–34.

Snedecor, G. W. and Cochran, W. G. (1989). *Statistical methods* (Ames, Iowa: Iowa State University Press).

Tamayo, P., Slonim, D., Mesirov, J., Zhu, Q., Kitareewan, S., Dmitrovsky, E., Lander, E., and Golub, T. (1999). Interpreting patterns of gene expression with self-organizing maps: Methods and application to hematopoietic differentiation. *Proceedings of the National Academy of Science USA*, **96** (6), 2907–2912.

Tryon, R. C., and Bailey, D. E. (1970). *Cluster analysis* (New York: McGraw-Hill).

Tukey, J. W. (1977). *Exploratory data analysis* (Reading, Mass.: Addison-Wesley Pub. Co.).

Yeung, K. Y., and Ruzzo, W. L. (2001). Principal Component Analysis for clustering gene expression data. *Bioinformatics*, **17** (9), 763–774.