MorphoTools: a set of R functions for morphometric analysis

Petr Koutecký

University of South Bohemia, Faculty of Science, Department of Botany, Branišovská 1760, České Budějovice, CZ-37005, Czechia; e-mail: kouta@prf.jcu.cz

Version 1.01

České Budějovice, October 2014

Introduction

The aim of this paper is to provide a set of functions for performing distance-based morphometric analysis in R (R Core Team 2013). No previous knowledge of R is necessary. The functions includes data import from Excel or tab-delimited text files, descriptive statistics for populations and taxa, histograms of characters, correlation matrices of characters, cluster analysis, principal component analysis (PCA), linear discriminant analysis (LDA) with permutation tests, classificatory discriminant analysis and *k*-nearest neighbour classification. The use of the functions is demonstrated on a sample data set. Detailed descriptions of the functions and examples of the scripts for producing graphics are included in this manual.

Experienced R users are welcome to modify the functions according to their specific needs. I would also be grateful for any comments, improvements of the functions or suggestions for additions in future versions.

Citation

Koutecký P. (2014): MorphoTools: a set of R functions for morphometric analysis. – Plant Systematics and Evolution xxx: xxx-xxx; doi 10.1007/s00606-014-1153-2

Files

Four files can be downloaded from http://botanika.prf.jcu.cz/systematics/morphotools.html:

- this manual (MorphoTools.pdf)
- function definitions (MorphoTools.R) – this file can be directly used as source code in R
- sample data (SamplaData.txt) – tab-delimited text file containing morphological measurements (see the ‘Sample data’ paragraph for details)
- working protocol (protocol.R) – plain text file (.R extension, but can be opened as a .txt file) that includes all commands used in the analysis of the sample data (including graphics); the commands can be easily copied to R, or the whole protocol can be opened in specialised software such as RStudio (see Installation) and directly edited and run.

Sample data

The sample data include portions of data sets from previously published studies by Koutecký (2007) and Koutecký et al. (2012): 25 morphological characters (see the cited studies for details) of the vegetative (stems and leaves) and reproductive structures (capitula and achenes) of three diploid species of the *Centaurea phrygia* complex: *C. phrygia* L. s.str. (abbreviated “ph”), *C. pseudophrygia* C.A.Mey. (“ps”) and *C. stenolepis* A.Kern. (“st”). Moreover, a fourth group includes the putative hybrid *C. pseudophrygia* × *C. stenolepis* (“hybr”). The data represent 8, 12, 7 and 6 populations for each group, respectively, and 20 individuals per population, with one exception in which only 12 individuals were available. All morphological characters are either quantitative (direct measurements, counts or ratios) or binary (two characters states or presence/absence). In four characters of achenes, there are missing data because fruits were not available in all individuals. In two populations of *C. stenolepis* fruits were completely missing. In total, the data set includes 652 individuals (453 complete) from 33 populations (31 complete).

Styles used in this manual

`function`	The function described in the following paragraph.
`function(parameters)`	Examples of R commands and functions calls. Anything underlined must be replaced by a specific value/name.
`results`	Names of R functions; results and values printed on screen

Installation

R and additional packages can be downloaded from http://www.r-project.org/. Follow the instructions on the R web page. All functions described here were tested with R version 3.0.2.

In working with R, it is possible to directly use the console version you will install. However, other software can communicate with R and handle your data in more conveniently (if software operated by text commands can be called convenient at all). I recommend RStudio (http://www.rstudio.com/), which, among other features, allows for the easy editing of R scripts and the running of selected parts of the script, highlights the syntax, and provides an instant list of objects and functions in the current environment.

Apart from the base R installation, download and install the following additional packages: ade4, class, permut, scatterplot3d, vegan. The packages are most easily downloaded and installed from the menu Packages – Install packages.

After installing R and the packages, set the working directory (menu File – Change dir…), copy the file with the function definitions (MorphoTools.R) to the working directory and either read it as a source using the menu File – Source R code… or type:

```
source(“FILE”)
```

(replace FILE with the case-sensitive file name, and be careful not to omit the quotes; if the file is not in the R working directory, you must enter the whole path, e.g., “D:/R/FILE.R”)

Introductory notes (for inexperienced R users)

There are many text books / web pages that can be used to learn basics of the R environment, such as Getting started with R (Beckerman & Petchey 2012), quite comprehensive The R book (Crawley 2013) or Quick-R webpage (http://www.statmethods.net/) and manuals on R homepage (http://cran.r-project.org/manuals.html). Here, I highlight only a few points that are necessary to know for using MorhoTools functions.

- R is case sensitive (including the names of objects, functions and function parameters)
- Text values in function calls (such as external files to be imported, names of taxa to be analysed, etc.) must be enclosed in double quotes (e.g., “taxon”). Function parameters are enclosed in parentheses, while individual parameters are usually separated by commas. Be careful not to omit any of these symbols; missing quotes etc. are frequent causes of non-functioning code and error warnings.
- Note that R (unlike Windows) will not ask for confirmation if you command it to delete or overwrite something, and these operations are generally irreversible.
- Most of the functions return new data as results (such as ordination scores, coefficients for variables, etc.). You must assign the results to a new object using `<-` . When not assigned, the results are displayed on a screen and lost. The commands therefore usually have this structure:

  ```
  RESULTS<-FUNCTION(data,parameters)
  ```
- R has useful help for built-in functions. To see the help file, simply type a question mark followed by the function name:

  ```
  ?FUNCTION
  ```

- To see an entire whole object, simply write its name. (However, this procedure is not useful for large datasets.)
- To open table-like objects (data frames, matrices) in a separate window:
  ```
  View(data)                # note the capital V in the function name
  ```
- Editing objects (good for functions; to edit data tables, it is usually easier to open data in an external editor and import the new version into R):
  ```
  fix(data)
  ```
- List of objects and tidying up. To see all objects in the current R environment (i.e., all objects and functions you have imported or created):
  ```
  ls()
  ```
- To delete object(s), use the command rm:
  ```
  rm(object1,object2,object3,…)
  ```
- To delete everything in the current R environment:
  ```
  rm(list=ls())
  ```

Introductory notes on graphics (for inexperienced R users)

- R can produce many types of graphs. However, making graphics through text commands is a rather creative process, and the settings of graphical parameters (such as symbol size or colour, axis labels, etc.) must be adjusted to fit optimally with the given data. Therefore, graphics production cannot be automated as a function. Instead, examples of figures for the sample data and the annotated codes used to generate them are included in this manual.
- Many graphical parameters can be set within graph-making functions such as `plot`. Parameters may also be set using `par`, but such settings are permanent and will influence all further plots. Therefore, the default settings should be stored in a list and restored from it, if necessary.

  To see all parameters:
  ```
  par(no.readonly=T)
  ```
  To save the default values into a list:
  ```
  default.pars<-par(no.readonly=T)
  ```
  To set the values of parameters:
  ```
  par(fg=”blue”)          # changing the default (foreground) colour to blue
  par(fg=”blue”,lty=2)    # changing the default colour and the default line type to dashed
  ```
  To restore the default values:
  ```
  par(default.pars)
  ```
- The parameter `mar` controls the width of graph margins (bottom, left, top, right). The default values are `mar=c(5, 4, 4, 2)+0.1`, which allows for the display of axis labels, as well as the main title and subtitle. Depending on the size of your screen and graphics window, it may be useful to adjust these margins. (For example, I rarely use the main title and therefore usually reduce the free space at the upper margin.)
  ```
  par(mar=c(4.5,4.5,1,2))
  ```
- Graphics are easily exported from R in vector formats as metafiles (.emf) or PostScript files (.eps or .ps) or PDFs. Direct export to raster formats (.bmp, .jpg, .png, .tif) is also possible through the menu Save as… (appears when the window of the graphical device is active), but the resulting images are low resolution.
- To obtain high-quality raster images, it is necessary to wrap the script (one or more lines) used to produce the image inside additional two lines, one starting the special graphical device and the other closing it after the image is produced. There are four functions, `bmp, jpeg, png, tiff` with the same parameters: file name, width and height, specification of units for width and height (px [pixels], in [inches], cm, mm), and resolution in dpi. Note that there are also additional parameters, but the default settings usually work well.
  ```
  png("File.png",width=15,height=12,units="cm",res=600)
     plot...
     axis...
  dev.off()
  ```

Data structure

All functions expect the following data structure: (i) one individual is recorded per row; (ii) the first three columns include unique identifiers for individuals, populations and taxa/groups, respectively; (iii) starting from the fourth column, any number of quantitative or binary morphological characters may be recorded; (iv) the first row contains variable names. Any column names can be used (avoiding spaces and special characters); the first three columns will be renamed to “ID”, ”Population” and “Taxon” by the import functions, respectively, because these names are expected by the other functions. If mixed populations containing more taxa are present, they must be divided into (sub)populations with unique names (i.e., each population consists of a single taxon). The columns ID, Population and Taxon are expected to be factors (categorial variables) and not numeric variables; if numbers are used, they will be converted to factor levels by the import functions. If there are missing values in the data, they must be represented as empty cells or by the text NA (not quoted), not zero, space or any other character.

An example of tab-delimited data file:

```
ID      Population      Taxon   SN      SF      ST      SFT     ...
RTE1    RTE     hybr    35.2    23.6    58.8    0.40    ...
RTE2    RTE     hybr    39.0    11.8    50.8    0.23    ...
...
PREL31  PREL    st      47.2    9.1     56.3    0.16    ...
PREL32  PREL    st      57.8    4.0     61.8    0.06    ...
...
```

Typical workflow and functions

Data import and control

`read.morfodata` and `read.morfodata2` import data from tab-delimited text files; import from Excel spreadsheets through the clipboard in Windows also usually works well. These functions are basically the `read.delim` and `read.delim2` functions, respectively, but they also rename the first three columns and convert them into factors if necessary. The two functions differ only in the decimal separator used: dot (default in R) is expected by the former, comma by the latter. The functions have only one parameter: the name of the input file.

```
data<-read.morfodata(“file.txt”)
data<-read.morfodata2(“clipboard”)
```

To check the data structure:

```
str(data)
```

This command displays the object’s class (should be “data.frame”), dimensions, and a list of its columns and their types. The first three columns should be “factor”; check the number of levels (populations, taxa) in the second and third columns. All other columns should be “num” (numeric) or “int” (integer). The presence of a “factor” among the characters indicates a problem with data import (check the decimal used) or in the data themselves (some values may not have been recognised as numbers, causing the column to be imported as text).

Brief summary of the data:

```
summary(data)
```

This command shows the levels and number of observations within the first 6 levels for factors and the min-max values, quartiles and medians for numeric variables. Checking the minimum and maximum values often helps to trace errors in data (missing decimal points, etc.). If there are missing data (NAs in R terminology) in some variable, the number of observations with missing data is indicated. Note that with other types of objects (such as computation results), the generic function `summary` provides different outputs specific to the type (class in R terminology) of the object.

Other useful functions:

```
dim(data)                 # dimensions (no. of rows/columns) of table-like objects
length(data)              # number of values in a vector or a list
levels(data$Population)   # list of populations
levels(data$Taxon)        # list of taxa
colnames(data)            # list of column names (characters)
```

Basic data manipulation

To make a copy of an object (of any type):

```
newdata<-data
```

Replacing an object with a new/modified one of the same name – for example, the results of functions that omit incomplete rows or modify some columns:

```
data<-FUNCTION(data)
```

Selecting part of the data. The easiest way to select part of a data frame is the use of subscripts, which take the form `DATA[ROWS,COLUMS]`. For example, `DATA[2,3]` selects the value on the second row in the third column. To access all rows of the third column, leave the rows parameter empty: `DATA[,3]` (Note that comma between rows and columns is still present!) Similarly, to select the second row with the values of all columns, use `DATA[2,]`. The range of numbers can be entered using a colon: `DATA[1:30,]` selects the values for the rows 1 to 30 and all columns. It is possible to use logical conditions (typically for rows); however, it is necessary to wrap the command using the `with` function (which has two parameters, the data and the expression to be evaluated; thus, in our case `DATA` must be repeated twice). Named elements (such as columns in a data frame) can also be referred using the symbol $, for example `DATA$Taxon`.

Note that after selecting part of the data, all factor levels from the original data are retained (for ID, Population and Taxon), even if there are no rows for a certain level in the new data. This situation can cause problems in further calculations. The `droplevels` function should be employed to remove the unused factor levels.

To select all data for a specified taxon (note the operator = = , "equals")

```
newdata<-with(data,data[Taxon=="taxon",])
newdata<-droplevels(newdata)
```

For more taxa (any number), use the operator | (logical OR)

```
newdata<-with(data,data[Taxon=="taxon1"|Taxon=="taxon2",])
newdata<-droplevels(newdata)
```

Removing part of the data is similar to selecting, only using negative indices. For example, `DATA[-(1:30),]` will remove the first 30 rows from the data frame and `DATA[,-4]` will remove the 4th column. To remove selected rows/columns that are not adjacent, use the function `c()` inside the subscript. For example, `DATA[,-c(5,7,10)]` removes the 5th, 7th and 10th columns.

To remove all data for a specified taxon (note the operator != , "not equal")

```
newdata<-with(data,data[Taxon!="taxon",])
newdata<-droplevels(newdata)
```

For more taxa (any number), use the operator & (logical AND)

```
newdata<-with(data,data[Taxon!="taxon1" & Taxon!="taxon2",])
newdata<-droplevels(newdata)
```

Missing data

MorphoTools functions can handle missing data; the omission of rows with missing data is usually included in the functions. To remove rows with missing data manually, use `na.omit` followed by `droplevels` to remove unused levels of the factors.

```
newdata<-na.omit(data)
newdata<-droplevels(newdata)
```

`na.meansubst` substitutes missing data using the average value of the respective character in the respective population. Generally, most of the multivariate analyses require a full data matrix. The preferred approach is to reduce the data set to complete rows only (i.e. perform the casewise deletion of missing data, as is incorporated within most of the functions) or to remove characters for which there are missing values (if they are concentrated in a few characters). The use of mean substitution, which introduces values that are not present in the original data, is justified only if (i) there are relatively few missing values, (ii) these missing values are scattered throughout many characters (each character includes only a few missing values) and (iii) removing all individuals or all characters with missing data would unacceptably reduce the data set.

```
newdata<-na.meansubst(data)
```

Exporting results

`export.res` can be used to export spreadsheet-like results (of the data frame or matrix classes) for most of the following functions; it will create a tab-delimited text file or copy the data to a clipboard (but the latter operation usually fails for large data). This function is only a wrapper for the `write.table` function with some parameters set differently from the default. The function has two parameters: the name of the data to be exported and the name of the file to be created; “clipboard” is the default for the latter.

```
export.res(data)                  # makes export to the clipboard
export.res(data,”clipboard”)      # the same
export.res(data,”file.txt”)       # makes export to a file
```

Descriptive statistics

`descr.all`, `descr.tax` and `descr.pop` calculate the descriptive statistics of each character in the whole dataset, each taxon and each population, respectively. The following statistics are included: number of observations, mean, standard deviation, and the percentiles 0% (minimum), 5%, 25% (lower quartile), 50% (median), 75% (upper quartile), 95% and 100% (maximum). The 5% and 95% percentiles are included because the trimmed range (without the most extreme 10% of values) is sometimes used in taxa descriptions, determination keys, etc. The results are formatted as data frames and can be exported using the `export.res` function.

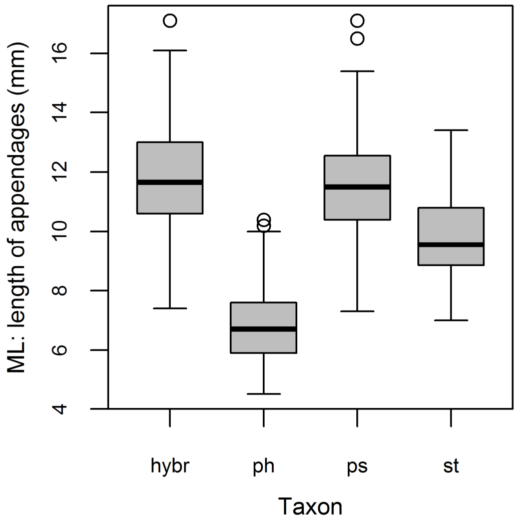
```
results1<-descr.all(data)
results2<-descr.tax(data)
results3<-descr.pop(data)
```

Box-and-whisker plots for characters (for example, the character ML from the sample data):

```
boxplot(ML~Taxon,data=indiv.orig,cex.axis=1,cex.lab=1,range=1.5,col="grey",
   xlab="Taxon",ylab="ML: length of appendages (mm)",
   pars=list(whisklty=1,boxwex=0.7))
```

The command can take one line or be divided into several lines; the only rules are that the parameters must be comma-separated and each line must end with a hard stop (enter). The parameters are as follows:

- The formula `y~x` defines the character to be plotted (y-axis) and grouping variable (x-axis), from `data`.
- `cex` (character expansion) – size of axis values (`cex.axis`) and axis labels (`cex.lab`) relative to the default.
- `range` – length of whiskers (not more than `range` times the interquartile range); values more distant from the lower/upper quartiles are drawn as outliers; `range=0` spreads the whiskers from the min to the max.
- `col` (colour) – fill of the boxes, which can be set



separately for each box using `col=c(“col”,”col”,…)`; to obtain a list of all colours use the command `colors()`; there are also a default 8 colours that can be coded by number instead of name (quotes are then not used): 1: black, 2: red, 3: green, 4: blue, 5: cyan, 6: magenta, 7: yellow, 8: grey.

- `xlab` and `ylab` – axis labels; similarly, `main` and `sub` define the graph main title (above the graph) and subtitle (under x-axis label), here, both are suppressed by the default settings `main=””` and `sub=””`.
- Names of groups on the x-axis are taken from the data (levels in the grouping variable) but can be overridden by the parameter `names=c(“name”,“name”,...)`.
- Other parameters are listed in `pars`
  - `whislty` – line type for whiskers (1: solid; 2: dashed [default], 3: dotted, etc.)
  - `boxwex` – relative width of boxes

Note that many other graphical parameters can be listed within `pars` (above, only those I consider important for the particular graph are used); see the help for all the parameters.

Population means

`popul.means` calculates the average value for each character in each population, with the pairwise deletion of missing data. The resulting data frame has populations as rows, and the first three columns are Population, Taxon, and N (the number of complete rows for each population). The data frame can be exported using the `export.res` function.

```
newdata<-popul.means(data)
```

If there are many missing data, some population may be represented only by a few complete individuals. In such a case, average values can be biased, and it can be useful to omit these populations from further analyses in which the populations are used as the operational taxonomic units (OTUs). To omit populations with less than *x* complete observations:

```
newdata<-with(data,data[!N<x,])
newdata<-droplevels(newdata)
```

`popul.otu` converts the data frame with the population means so that the populations can be used as OTUs in the multivariate analyses. This data frame must have the same structure as that of the individual data; i.e., the first three columns must be ID, Population and Taxon; in this case, ID contains the same data as Population. Note that when using populations as OTUs they are handled with the same weight in all analyses (disregarding population size, within-population variation, etc.)

```
newdata<-popul.otu(data)
```

Distribution of characters, transformations

`charhist` and **`charhist.t`** plot histograms and the expected normal distribution of quantitative (not binary) characters, the former for the whole dataset and the latter for a specified taxon. These functions show graphs for the characters sequentially and the next graph is initiated by pressing enter or clicking the mouse. The purpose of these functions is to examine normality of distribution, which is expected by some analyses (Pearson correlation coefficients, PCA, discriminant analyses). More precisely, within-group distribution should be normal, and thus one-taxon histograms are more relevant.

```
histchar(data)
histchar.t(data,”taxon”)
```

If a normality test of some character is needed, the Shapiro-Wilk test may be performed with the following R base function:

```
shapiro.test(data$character)
```

For a selected taxon only:

```
x<-with(data,data[Taxon==”taxon”])
shapiro.test(x$character)
```

Several other normality tests are available in the package `nortest` (Gross & Ligges 2012).

The characters that deviate most from the normal distribution can be transformed. If both individuals and populations are used as OTUs, both data sets should be transformed in the same way. Logarithmic, both natural (`log`) and common (`log10`), and square root (`sqrt`) transformations are the most often used. If there are zero values in the data, it is necessary to add some constant (for common logarithmic transformation, a value of 1 is often used, as this value becomes 0 after transformation). In case of frequency (percentage) data (such as portion of certain type of trichomes among all trichomes), sometimes no transformation is needed (if the span of values is not too big and the values are generally between 30 and 70%) or simple arcsin (angular) transformation is often sufficient. Examples are as follows:

```
data$character<-sqrt(data$character)
data$character<-log(data$character)
data$character<-log10(data$character+1)
data$character<-log10(10*data$character+1)
data$character<-asin(sqrt(data$character/100))
data$character<-asin(sqrt(data$character))     # as above for frequency from 0 to 1, not %
```

Using the same character names on the left side as on the right side, the function will overwrite the original values. Using different names, the function creates a new column and retains the original values in the data. I prefer the former option. In this case, it is useful to rename the transformed characters. First, use the `colnames` function to see the (ordered) column names. Alternatively, you can find the index of the requested character using `which`. Then, rename the columns using subscripts to `colnames` with the position (index) of each character. For example, renaming Char2 and Char5 from *data*:

```
colnames(data)
[1] “ID”     “Population”   “Taxon”    “Char1”    “Char2”     “Char3”      “Char4”
[8] “Char5” “Char6”
```

or

```
which(colnames(data)=="Char2")
[1] 5
which(colnames(data)=="Char5")
[1] 8
```

then

```
colnames(data)[c(5,8)] <- c(“logChar2”,”logChar5”)
```

The transformed data can be exported using the `export.res` function.

Correlations of characters

`cormat.p` and `cormat.s` calculate the matrices of the correlation coefficients of the characters (Pearson’s and Spearman’s, respectively). The results are formatted as data frames to allow export with the `export.res` function. Highly correlated characters (r > |0.95|) should not be used in principal component analysis and especially in discriminant analysis. Significance tests are not performed, as they are usually unnecessary for morphometric analysis.

```
results<-cormat.p(data)
results<-cormat.s(data)
```

If significance tests are needed, they can be computed using the `cor.test` function from the R base installation (which tests pairs of characters) or the `corr.test` function from the `psych` package (Rewelle 2013) (matrix of significance tests).

```
cor.test(data$char1,data$char2,method="pearson")
cor.test(data$char1,data$char2,method="spearman")
```

Cluster analysis

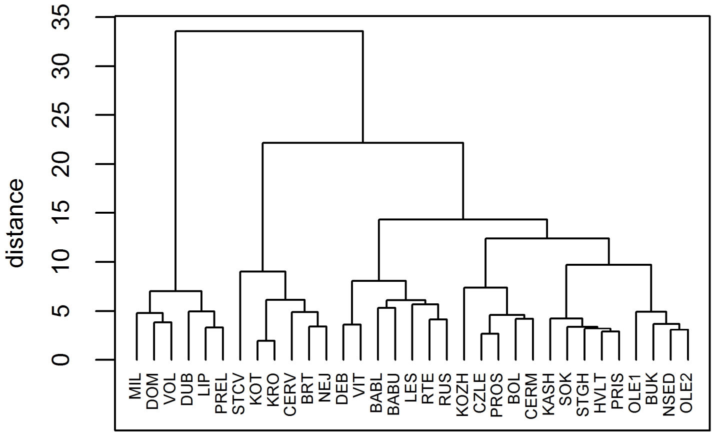
`clust.upgma` and `clust.ward` perform agglomerative hierarchical clustering based on Euclidean distances using UPGMA and Ward's method, respectively. Both functions include (i) standardisation of the characters to a zero mean and a unit standard deviation using the `scale` function, (ii) calculation of the distance matrix using the `dist` function and (iii) clustering using the function `hclust`. Typically, populations are used as OTUs.

```
results<-clust.upgma(data)
results<-clust.ward(data)
```

Dendrograms are produced using `plot` (note that this is a generic R function, with settings depending on the class of the plotted object):

```
plot(popul.ward,cex=0.6,cex.axis=0.8,cex.lab=0.8,frame.plot=T,hang=-0.1,
   main="",sub="",xlab="",ylab="distance")
```

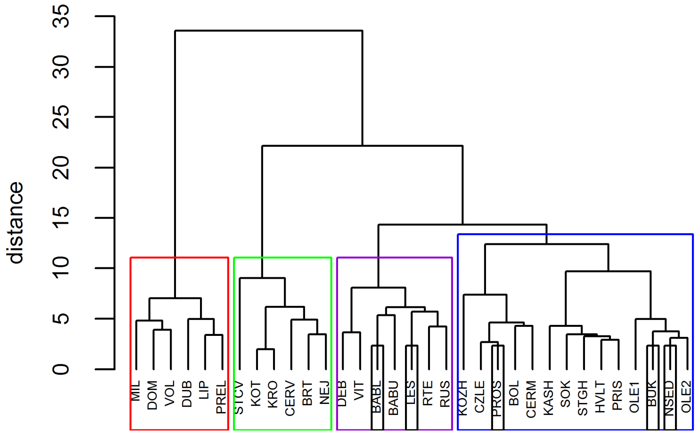
Important parameters:



- `data` – the first parameter (unlike other function, it is not called `data=data`).
- `cex`, `cex.axis`, `cex.lab` – relative size of OTU names, y-axis values and axis labels.
- `frame.plot` – logical (`T/F` or `TRUE/FALSE`), sets if the frame along the plot is drawn.
- `hang` – distance (fraction of the plot height) of the label from the last node of the dendrogram, i.e., length of the terminal branches; if negative (any value), the labels hang down from 0.
- `main`, `sub`, `xlab`, `ylab` – main title, subtitle, x-axis label, and y-axis label, respectively; `""` indicates no label.

Rectangles highlighting certain clusters can be added using `rect.hclust`:

```
rect.hclust(popul.ward,h=12,which=c(1,2,3),border=c("red","green","blue"))
```



- `data` – the same as in the function `plot`.
- h – linkage distance (in units of the y-axis) to cut the dendrogram; branches crossing this value define clusters.
- `which` – number of clusters to be drawn (from left to right); for example, there may be five clusters at `h=12`, but we want to draw only the first three. For one cluster, directly enter the number (e.g., `which=2`); for more clusters, wrap the numbers in the function `c()`, as above.
- `border` – colour of the rectangle; either single a value common to all clusters selected by `which` (e.g., `border = "green"`) or a separate colour for each cluster.

The use of both the `plot` and the `rect.hclust` functions requires some experimentation. After finding good parameters for `plot`, start with rectangles specifying only `h` (i.e., draw all clusters). Select the rectangles to draw using `which`, set `border`, and rerun both functions. The `rect.hlust` function with different parameters can be used several times for one dendrogram.

The example figure was created using this code (for comparison with the previous graph, `frame.plot=F` is used):

```
popul.ward<-clust.ward(popul)
plot(popul.ward,cex=0.6,cex.axis=0.8,cex.lab=0.8,frame.plot=F,hang=-0.1,
   main="",sub="",xlab="",ylab="distance")
rect.hclust(popul.ward,h=12,which=c(1,2,3),border=c("red","green",
   "darkviolet"))
rect.hclust(popul.ward,h=14,which=4,border="blue")
rect.hclust(popul.ward,h=2,which=c(14,16,21,30,31),border="black")
```

The red cluster is *Centaurea stenolepis*, the green is *C. phrygia* s.str., the violet is the hybrid *C. pseudophrygia* × *C. stenolepis* and the blue is *C. pseudophrygia*. The "incorrectly" clustered populations are highlighted as the black "clusters" (BABL and LES were originally classified as *C. pseudophrygia*, PROS as a hybrid and BUK and NSED as *C. phrygia* s.str.).

Principal component analysis (PCA)

`pca.calc` performs principal component analysis using the R base `prcomp` function after omitting the rows with missing data. Standardisation of characters to have a zero mean and a unit standard deviation is employed. Note that there are also other PCA functions in R, such as `princomp` and `dudi.pca` (the `ade4` package, Dray & Dufour 2007).

```
results<-pca.calc(data)
```

Variation explained by individual axes:

```
summary(results)
```

Note that the function `prcomp` stores only the square roots of eigenvalues (i.e., standard deviations of the principal components). To obtain the actual eigenvalues, use the `pca.eigen` function:

```
pca.eigen(results)
```

You can also plot the eigenvalues as a scree plot (the second command draws points connected with a line instead of bars):

```
plot(results)
plot(results,type="lines")
```

The correlation of the characters and ordination axes, i.e., character loadings, can be extracted using `pca.cor` with two parameters, PCA results and number of axes to save (*n*). For *n*, the default value is 4, but any value can be specified.

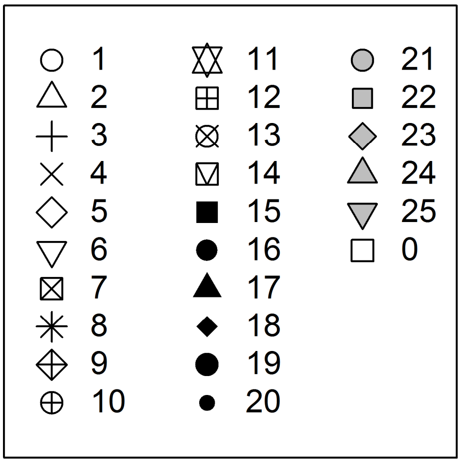
```
correlations<-pca.cor(results)          # extracts the first four principal components
correlations<-pca.cor(results,n)        # extracts the specified n of principal components
```

`pca.scores` computes the ordination scores of cases (objects, OTUs). Note that this function has three arguments: results (PCA definition), data, and number of axes to extract (*n*; default is four). The data may also contain additional (passive) samples that were not present in the original analysis. The function adds the first three columns (ID, Population, Taxon) from the data to the final data frame, which is useful for plotting taxa/populations with different symbols or colours.

```
scores<-pca.scores(results,data)
scores<-pca.scores(results,data,n)
```

Both coefficients of characters and case scores can be exported using `export.res` function.

A scatterplot of the ordination scores can be produced using the `plot` function. There are 20 different symbols in R and 5 additional (21–25) symbols whose background colours can be set differently from their foreground colours. Colours are coded by name (type `colors()` to see the list); 8 default colours may also be coded by



number(1: black, 2: red, 3: green, 4: blue, 5: cyan, 6: magenta, 7: yellow, 8: grey). Factor levels may be used in the variable Taxon to code symbols/colours by specifying the `pch` [symbols] and `col` [colour] arguments of `plot` (see below) using `as.numeric(Taxon)`, but this method has one major drawback. The `as.numeric` function will code the first level found in the specified variable as 1, the second as 2, etc., and these numbers will then be used to code the symbols/colours.

If you want use different symbols/colours, define a copy of the Taxon column in the data frame, then recode the levels to the desired numbers and use them for plotting. A new column, Symb or Col (or both), must be created using the `as.numeric` function, and the functions `recode.symb` or `recode.col` then set the numbers in the Symb/Col column according to the values in Taxon column.

```
data$Symb<-as.numeric(data$Taxon)
data<-recode.symb(data,"taxon",symbol)
data$Col<-data$Taxon                    # if colours will be coded by names
data<-recode.col(data,"taxon","colour")
data$Col<-as.numeric(data$Taxon)        # if colours will be coded by numbers
data<-recode.col(data,"taxon",colour)
```

Then use the `plot` function to create the scatterplot; to simplify the column names, it may be wrapped in the function `with`:

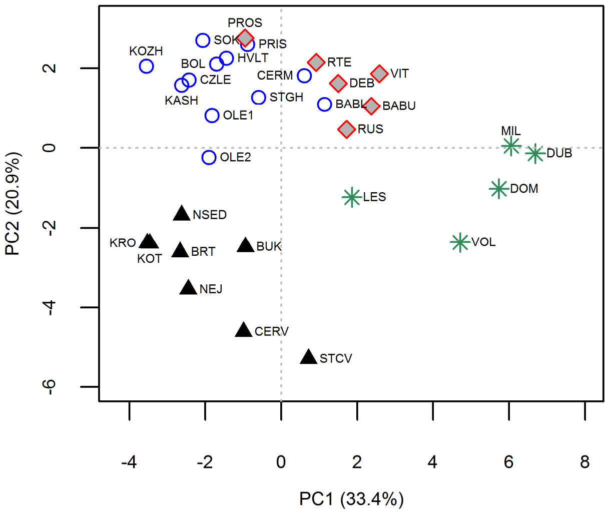
```
with(popul.pca.sc,plot(PC1,PC2,type="p",pch=Symb,bg="grey80",col=Col,
   xlim=c(-4.0,8.0),ylim=c(-6.0,3.0),cex=1.5,cex.axis=1,cex.lab=1,
   xlab="PC1 (33.4%)",ylab="PC2 (20.9%)"))
```

Important parameters:

- The first parameter of the `with` function defines the data to plot.
- The first and second parameter of the `plot` function represent the column names from the data that define the x- and y-coordinates, respectively; in the `PCA.calc` results, the columns are called PC1, PC2,…
- `pch` defines the symbols; it can be a single number (all symbols will be of the same type) or the column containing the symbol codes. When `pch` is omitted, the default symbol (1: open circle) will be used.
- `col` defines the colour of the symbols, in a similar way as `pch`.
- `bg` sets the background colour for symbols 21–25.
- `xlim,ylim` define the minimum and maximum for the x- and-y axis, respectively. This parameter can be omitted. Extending the axes can be useful to create free space for a legend or point labels, etc.
- `cex`, `cex.axis`, `cex.lab`, `cex.main` and `cex.sub` define the relative sizes of symbols, axis values, axis labels, main title and subtitle, respectively (last two are not used in an example above).
- `xlab` and `ylab` are labels for the x- and y-axis, respectively; if omitted, the respective column names are used. The parameters `main` (main title) and `sub` (subtitle) are similar (not used here).

Several other elements can be added into the existing plot by functions called after the `plot` function. The function `abline` adds lines at x=0 and y=0 (or any other position controlled by the parameters `h` and `v`); the parameter `lty` defines the line type (1: solid, 2: dashed, 3: dotted; three other types are available); `col` defines colour in usual way.

```
abline(h=0,v=0,lty=3,col="grey")
```



Text descriptions of points may be added using the function `text`. To display population names:

```
with(popul.pca.sc,text(PC1,PC2,
   Population,cex=0.5,adj=NULL,pos=4,
   offset=0,col="black"))
```

The function call is similar to the `plot` function, and its special parameters are as follow:

- The first two arguments are the x- and y-coordinates of the data point; the third argument is the variable containing labels.
- `adj` – adjustments of the label position, as `c(x,y)`
- `pos` – position of the label relative to the data point (1: below, 2: to the left, 3: above, 4: to the right); if specified, it overrides `adj`.
- `offset` – the offset (in fraction of character width) of the label from the position specified by `pos`; default is 0.5.
- Note that it is possible to select populations for labelling with the `text` function using subscripts (to data that are specified within the `with` function); rerunning the function with different settings of the `pos` and `offset` parameters for the selected populations can prevent the labels of individual characters from overlapping.

The whole code used to generate the figure for populations (to demonstrate various symbol types, colours and work with text labels):

```
popul.pca<-pca.calc(popul)
popul.pca.sc<-pca.scores(popul.pca,4,popul)
popul.pca.sc$Symb<-as.numeric(popul.pca.sc$Taxon)
popul.pca.sc$Col<-as.numeric(popul.pca.sc$Taxon)
popul.pca.sc<-recode.symb(popul.pca.sc,"ps",1)
popul.pca.sc<-recode.symb(popul.pca.sc,"st",8)
popul.pca.sc<-recode.symb(popul.pca.sc,"hybr",23)
popul.pca.sc<-recode.symb(popul.pca.sc,"ph",17)
popul.pca.sc<-recode.col(popul.pca.sc,"ps","blue")
popul.pca.sc<-recode.col(popul.pca.sc,"st","seagreen")
popul.pca.sc<-recode.col(popul.pca.sc,"hybr","red")
popul.pca.sc<-recode.col(popul.pca.sc,"ph","black")
with(popul.pca.sc,plot(PC1,PC2,type="p",pch=Symb,bg="grey70",col=Col,
   xlim=c(-4,8),ylim=c(-6,3),cex=1.5,cex.axis=0.9,cex.lab=0.9,
   xlab="PC1 (33.4%)",ylab="PC2 (20.9%)"))
abline(h=0,v=0,lty=3,col="grey")
with(popul.pca.sc[-c(3,6,13,14,15,16,18,24),],text(PC1,PC2,Population,cex=0.9,
   adj=NULL,pos=4,offset=0.2,col="black"))
with(popul.pca.sc[c(13,14),],text(PC1,PC2,Population,cex=0.9,adj=NULL,pos=1,
   offset=0.3,col="black"))
with(popul.pca.sc[c(3,6,16),],text(PC1,PC2,Population,cex=0.9,adj=NULL,pos=2,
   offset=0.3,col="black"))
with(popul.pca.sc[c(15,18,24),],text(PC1,PC2,Population,cex=0.9,adj=NULL,
   pos=3,offset=0.3,col="black"))
```

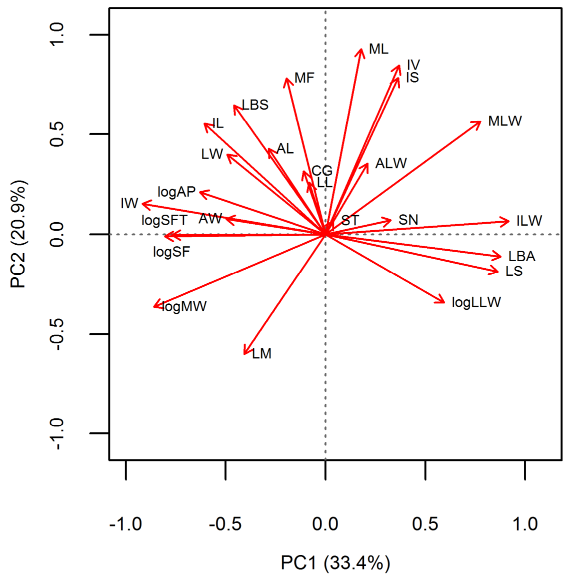
The ordination scores of characters may be drawn as arrows. An empty plot (drawn with only axes and their labels) must be prepared using the function `plot`, zero lines added using the `abline` function, and arrows and their labels added using the functions `arrows` and `text`. (Both functions must be called after `plot`, as they only add elements into the already existing plot.)

The functions `plot`, `abline` and `text` are called as above. The only difference is the use of `type="n"` as a parameter of `plot` (no points are drawn; this also renders useless any graphical parameters concerning the data points, such as `cex`, `col`, `pch`).

```
with(popul.pca.co,plot(PC1,PC2,type="n",xlim=c(-1,1.1),ylim=c(-1,1)))
abline(h=0,v=0,lty=3,col="grey")
with(popul.pca.co,arrows(0,0,PC1,PC2,length=0.15,lty=1,col="red"))
with(popul.pca.co[-c(2,4,6,13,23,25),],text(PC1,PC2,Character,pos=4,cex=0.75))
with(popul.pca.co[c(6,13,23,25),],text(PC1,PC2,Character,pos=2,offset=0.2,
   cex=0.75))
with(popul.pca.co[2,],text(PC1,PC2,Character,pos=1,offset=0.3,cex=0.75))
with(popul.pca.co[4,],text(PC1,PC2,Character,pos=3,offset=0.3,cex=0.75))
```

The arguments of the `arrows` function are as follow:

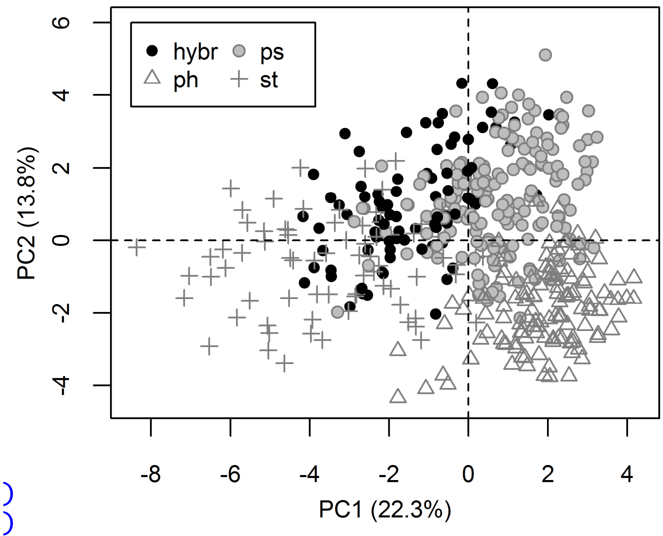
- The first two arguments (0,0 here) define the origin of the arrows. The next two arguments are variables in the data (here PC1, PC2) that define the end points of the arrows.
- `length` defines the size of the arrow heads.
- `lty` and `col` have their usual meanings (line type and colour, respectively)



Generally, both scripts may be combined to produce a biplot by making a plot of the populations and then adding arrows and their labels, omitting the second call of `plot` and `abline`. However, some rescaling of population scores to values comparable to the characters is usually necessary. (Here, the x-axis for the populations was <-4,8> while it was <-1,1> for the characters; thus, dividing population scores by 4 would create a better picture). This procedure can be conducted directly within the `plot` and `text` functions:

```
with(...,plot(PC1/4,PC2/4,...))
with(...,text(PC1/4,PC2/4,...))
```

However, if there are many characters and data points, they will most likely overlap. I therefore consider two separate scatterplots more informative than a biplot.



Here is a PCA of individuals, configured to highlight the position of the hybrids in a greyscale-only figure. Compared to the scatterplot of populations, individual labels are omitted (too many would be present), but use of a legend is demonstrated:

```
indiv.pca<-pca.calc(indiv)
pca.sc<-pca.scores(indiv.pca,4,indiv)
pca.sc$Symb<-as.numeric(pca.sc$Taxon)
pca.sc$Col<-as.numeric(pca.sc$Taxon)
levels(pca.sc$Taxon)
pca.sc<-recode.symb(pca.sc,"ph",2)
pca.sc<-recode.symb(pca.sc,"ps",21)
pca.sc<-recode.symb(pca.sc,"st",3)
pca.sc<-recode.symb(pca.sc,"hybr",16)
pca.sc<-recode.col(pca.sc,"ph","grey50")
pca.sc<-recode.col(pca.sc,"ps","grey50")
pca.sc<-recode.col(pca.sc,"st","grey50")
pca.sc<-recode.col(pca.sc,"hybr","black")
with(pca.sc,plot(PC1,PC2,type="p",pch=Symb,col=Col,bg="grey",
xlim=c(-8.5,4.3),ylim=c(-4.5,6),cex=1,cex.axis=1,cex.lab=1,
   xlab="PC1 (22.3%)",ylab="PC2 (13.8%)"))
abline(h=0,v=0,lty=2,col="black")
legend(-8.5,6.0,levels(pca.sc$Taxon),pch=c(16,2,21,3),
   col=c("black",rep("grey50",3)),pt.bg="grey",bty="o",cex=0.9,pt.cex=1,ncol=2)
```

The parameters of `legend`:

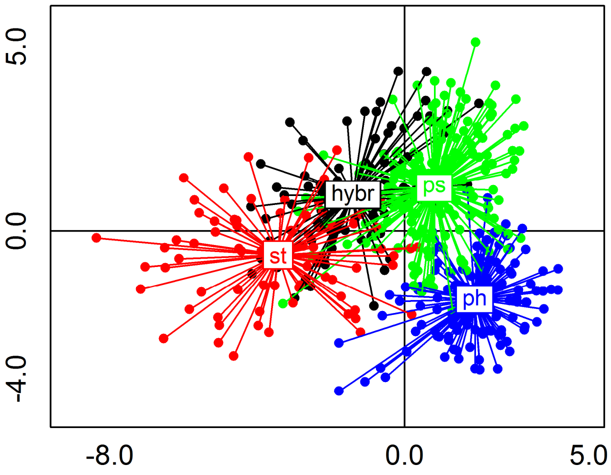
- The first two arguments indicate the position of the upper left corner of the legend. Note that it is possible to create free space for the legend by properly setting the `xlim` and `ylim` parameters of the `plot` function.
- The third argument defines the text labels displayed in the legend. Here, the labels are taken from the variable Taxon using the `levels` function. Alternatively, arbitrary text labels can be defined using the `c()` function:
  `c(“text1”,”text2”,...)`

- `pch` – symbols used in the legend (be careful to use the correct order respective to the Taxon levels).
- `col` – colour of symbols. Note that I used the function `rep` here, which has two arguments: what to repeat and how many times: `rep(“grey50”,3)` is equivalent to `“grey50”,“grey50”,“grey50”`.
- `pt.bg` is equivalent to `bg` in the `plot` function, i.e., the background colour of symbols 21–25. Note that the parameter `bg` is also present in the `legend` function, but it defines the background of the whole legend.
- `bty` defines the frame around the legend `(“o”` for rectangle or `“n”` for none)
- `cex` defines the relative size of the legend text, while `pt.cex` defines the size for legend symbols; if `pt.cex` is not specified (default), then `pt.cex = cex`.
- `ncol` defines the number of columns in the legend. The default is `ncol=1`, i.e., a vertical legend. `ncol=k`, where *k* is the number of levels, would produce a horizontal legend.
- Note that many other graphical parameters are present for `legend` that define legend text alignment, spacing or colour, legend frame type and colour, etc., but the default settings usually work well; see the help for details.

An interesting possibility for showing the overlap of groups is the “spider” diagram of the function `s.class` from the `ade4` package. Note that this package also includes other graphing functions (such as `s.arrow` for characters and `scatter` for biplots), as well as its own PCA function, `dudi.pca`.

```
library(ade4)
par(oma=c(2,2,1,1))
s.class(pca.sc,pca.sc$Taxon,xax=4,yax=5,pch=16,label=levels(pca.sc$Taxon),
   col=c("black","blue","green","red"),cstar=1,axesell=F,addaxes=T,grid=F,
   clabel=0.8,cpoint=0.8)
axis(1,pos=-4.7,cex.axis=1,lwd=0,at=c(-8,0,5),labels=c("-8.0","0.0","5.0"))
axis(2,pos=-9.5,cex.axis=1,lwd=0,at=c(-4,0,5),labels=c("-4.0","0.0","5.0"))
par(default.par)
```

Parameters of `s.class`:



- The first arguments the data to plot, and the second the variable that define the classes to plot.
- `xax` and `yax` are the numbers of columns in the data with x- and y-coordinates, respectively
- `pch` and `col` have their usual meaning and must be either a single number (the same for all classes) or values for each class enclosed in the `c()` function.
- `label` defines class labels, using either an expression (such as the function `levels` here) or a list of values for each class, similarly to `col` and `pch`.
- `cstar` defines the length of star “rays” from the interval <0,1>, `axesell=F` suppress inertia ellipses, `addaxes=T` draws axes, and `grid=F` suppress gridlines.
- `clabel` and `cpoint` defines the sizes of class labels and points, respectively (equivalents of the `cex` parameters of the `plot` function).

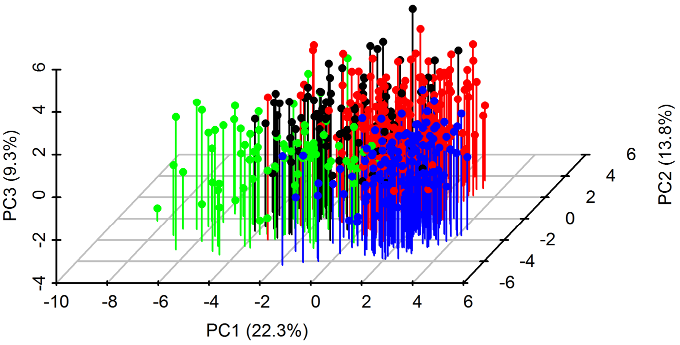
The function `s.class` does not print axis values. These values can be added using the generic function `axis`, although much experimentation is needed depending on the graph window size and shape. First, it is necessary to set the margins using `par(oma=...))` and then to use the `axis` function:

- the first argument is axis to be drawn (1: horizontal, 2: vertical)
- `pos` is the axis position on the perpendicular axis

- `cex.axis` sets the size of the axis labels as usual.
- `lwd=0` suppresses the axis to be draw (`lwd` is line width) so as not to interfere with the axes drawn by `s.class`
- tickmarks would be displayed if `lwd` were not zero, their length is controlled by the `tcl` parameter. However, it is difficult to find a combination of axis position, labels and tickmark length that creates a good figure. Therefore, tickmarks are omitted here.

A 3D scatterplot can be produced using the `scatterplot3d` function from the `scatterplot3d` package (Ligges & Mächler 2003). Similarly to the process for 2D plotting, it is first necessary to define symbol types and colours:

```
levels(pca.sc$Taxon)
pca.sc<-recode.col(pca.sc,"ph","blue")
pca.sc<-recode.col(pca.sc,"ps","red")
pca.sc<-recode.col(pca.sc,"st","green")
pca.sc<-recode.col(pca.sc,"hybr","black")
library(scatterplot3d)
with(pca.sc,scatterplot3d(PC1,PC2,PC3,color=Col,pch=16,type="h",
   lty.hplot=1,grid=T,col.grid=”grey”,box=F,angle=60,scale.y=0.9,cex.symbols=1,
   cex.axis=0.9,cex.lab=0.9,mar=c(4,4,0,3),y.margin.add=0.3,xlab="PC1 (22.3%)",
   ylab="PC2 (13.8%)",zlab="PC3 (9.3%)"))
```



Then, it is necessary to load the scatterplot3d package using the `library` command. The parameters of the `scatterplot3d` function are as follow:

- The first three parameters represent the x-,y- and z-coordinates of the points, respectively.
- `color` has the same meaning as `col` in the other plotting functions.
- `bg` is also similar to its use in the `plot` function (symbols 21–25; not used here).
- `pch` has its usual meaning; note that in the present figure, only one character is used for all taxa.
- `type` is similar to the same parameter in the `plot` function; the relevant values are “`p`” for points only, “`h`” for points with vertical lines and “`n`” for nothing.
- `grid` is logical (`TRUE`/`FALSE`) – whether gridlines are drawn the on x-y plane.
- `col.grid` sets the colour of the gridlines if `grid=T`.
- `box` (logical) defines whether only the axes or all edges of a box are drawn around the graph.
- `angle` and `scale.y` control the “rotation” of the graph; the former is the angle between the x- and y-axis and the latter is the relative length of the y-axis.
- `mar` defines the width of the margins in a similar way as `par(mar=...)`; note that setting margins using `par` is not effective in the `scatterplot3d` package.
- `y.margin.add` adds extra space between the y-axis values and the axis label to avoid overlap.
- `cex.`, `xlab`, `ylab`, `zlab`, `sub`, and `main` all have their usual meanings.

Canonical discriminant analysis

(*also know as linear discriminant analysis or canonical variate analysis; CDA or LDA or CVA*)

Canonical discriminant analysis can be performed in several R packages. Here, the use of the `vegan` package (Oksanen et al. 2013) is demonstrated. The major advantage of `vegan` is that it uses permutation significance tests, which overcome the need for the normal distribution of characters. Standardisation of data is not necessary in discriminant analyses and thus the original values of characters are used.

`discr.calc` creates an object, `discr.data`, that contains two data frames (values of the characters and classification of objects into taxa) and passes them to the `cca` function from the `vegan` package, which performs the discriminant analysis.

```
results<-discr.calc(data)
```

`discr.sum` summarises the results and returns the eigenvalues of the canonical axes, the percentage of explained variation, the canonical correlation coefficients of the individual axes, and the permutation tests of the whole model and of the individual axes (using the functions `summary.cca`, `eigenvals`, `spenvcor` and `anova.cca`). Note that the axes are called CCA1, CCA2, etc. The eigenvalues in `vegan` (similarly as in Canoco) are equal to the squares of canonical correlation coefficients and are different from the eigenvalues printed by other software packages (see ter Braak & Šmilauer 2012 for details); the recalculation is returned as the first item by the `discr.sum` function. The number of permutations can be set using the parameter `perm`; the default value is 500. The number of permutations determines the lowest achievable significance level (1/n); setting more permutations allows for more precise tests, but the permutation procedure is rather slow in R.

```
discr.sum(results)                # using default number of permutations (500)
discr.sum(results,perm=1000)      # user-specified number of permutations
```

`discr.coef` returns the coefficients of discriminant function (regression coefficients) for individual characters (based on the function `coef`). These coefficients refer to centered but unstandardised characters [for characters A, B,.., the equation is (A – mean (A)) * coef A + (B – mean (B)) * coef B + …]. Because character means are needed to compute the object scores, they are returned as the first column of the resulting data frame. However, the calculation of canonical scores (even for new objects) is more easily achieved using the `discr.scores` function (see below). The results can be exported using the `export.res` function.

```
coefficients<-discr.coef(results)
```

`discr.taxa` returns the positions (centroids) of taxa (using the function `scores`); the results can be exported using the `export.res` function.

```
scores<-discr.taxa(results)
```

`discr.scores` computes the ordination scores of cases (OTUs). The function has two arguments, results (discriminant analysis definition) and new data. The function uses the generic function `predict` and can also calculate scores for (passive) samples that were not present in the original ordination. This approach is advantageous for testing the positions of “atypical” populations (e.g., putative hybrids) or for assessing the taxonomic positions of selected individuals (e.g., type herbarium specimens). The function adds the first three columns (ID, Population and Taxon) from the data to the final data frame, which is useful for plotting taxa/populations using different symbols or colours. The resulting data frame can be exported using the `export.res` function.

```
scores<-discr.scores(results,data)
```

`discr.bip` returns the “biplot scores”, i.e., the contribution of characters to individual axes (also called factor structure or total structure coefficients by other software packages). Compared to usual biplot scores (as returned by the `summary.cca` function), these values are standardised by within group variance instead of by total variance to better reflect the relative importance of the characters (see Lepš & Šmilauer 2003 for details). The resulting data frame can be exported using the `export.res` function.

```
scores<-discr.bip(results)
```

`discr.test` performs significance tests of individual characters. Two types of tests are computed. First, the marginal effects (i.e., when a character is alone in the model) are computed using the generic function `add1`. Second, the unique contributions of the characters (i.e., the addition of each character into the model with all other characters) are tested using the `anova.cca` function with the setting `by=”margin”`; note that these contributions are called marginal effects in `vegan` terminology. The latter analogous to the standardised coefficients of the discriminant function computed by several other software packages. Similarly to the `discr.sum` function, the number of permutations (500 is default) should be chosen with respect to the desired significance level. Although the algorithm is rather slow, calculation should not take more than a few minutes, even with 1000 permutations (unless the dataset is extremely large). As many tests (one for each character) are performed, some correction of significance levels should be considered. The results of the `discr.test` function consist of a list containing two data frames. These data frames can be exported separately using the `export.res` function by adding a subscript to the list (note the double brackets). They can also be exported together, but the names of the columns are changed during the export.

```
tests<-discr.test(results)
tests<-discr.test(results,perm=1000)         # user-specified number of permutations
export.res(tests[[1]])                       # marginal effects
export.res(tests[[2]])                       # unique contributions
```

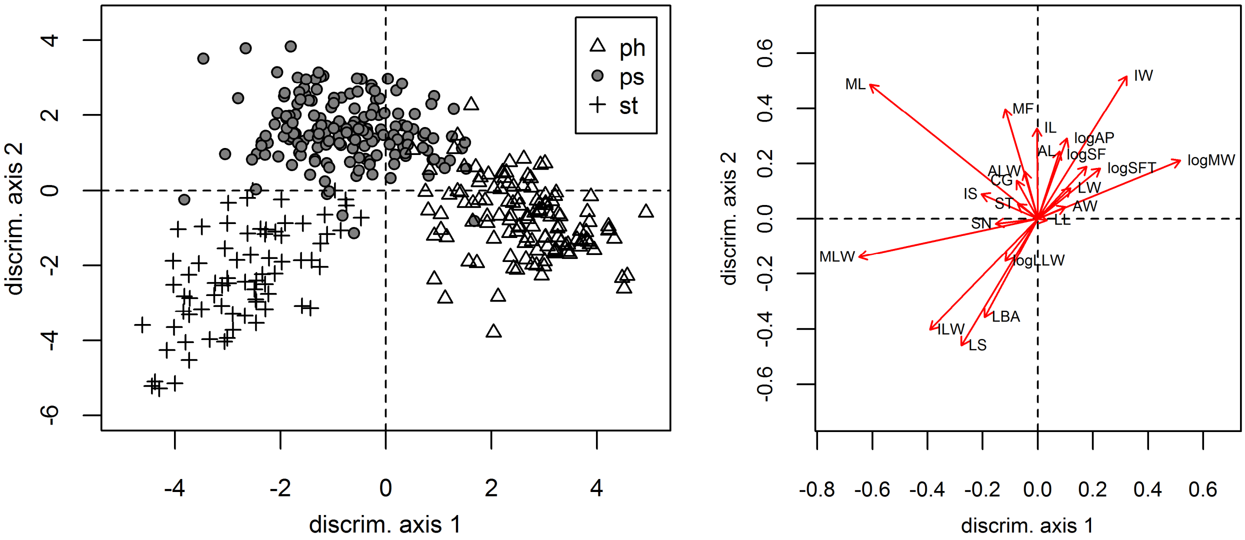
Forward selection of characters is performed by the `discr.step` function, which is based on the `ordistep` function. Two models can be specified by the parameter `dir`: `dir=”forward”` performs the addition of characters starting with a null model, while with `dir=”both”`, the function alternates the addition and removal of one character and stops when no significant change is possible in either direction. Direction (default is forward) and number of permutations (default is `perm = 500`) can be specified, and the threshold significance level can also be set (default is `p=0.05`). The inclusion of a character into/removal from the model is based on permutation tests; the function also print AIC values (similarly to other stepwise models in R), but these values are not meaningful for cca ordination (see the R help for details).

Note that the `discr.step` function is intended to print the results on a screen. The results of stepwise selection using the `discr.step` function may be assigned as a new discriminant analysis, however, the functions for characters (`discr.coef`, `discr.bip`) will not work properly, as they look for (the original set of) characters stored in `discr.data`. Therefore, a new data set with only selected characters should be prepared and the `discr.calc` function then used to produce a new `discr.data` list.

```
discr.step(results)                    # default settings
discr.step(results,perm=200)           # reducing the number of permutations
discr.step(results,perm=1000,dir=”both”,p=0.05/22)
                                       # all parameters set manually, Bonferroni correction of
                                       significance levels used (22 characters present)
newresults<-discr.step(results)        # saves the results as a new discriminant analysis
```

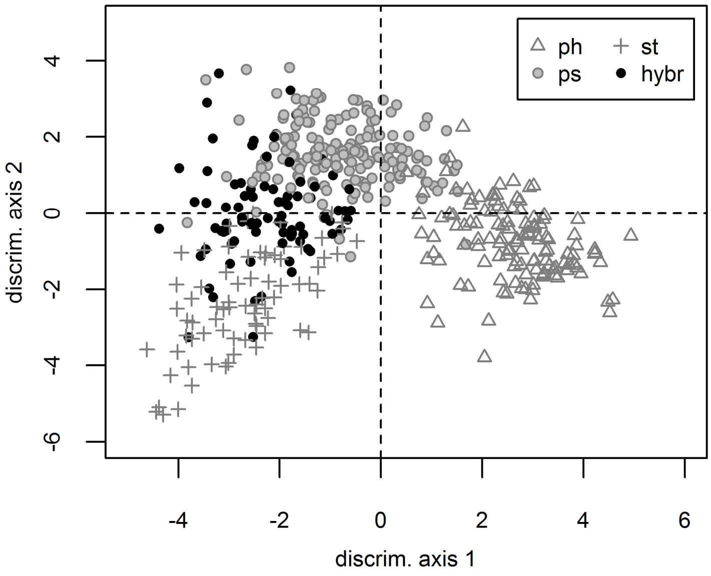
Graphics can be produced in a similar way as in PCA. The code to produce a scatterplot of individuals (without hybrids, which were excluded from the analysis) and a plot of characters is as follows:

```
# individuals (from discr.inds3)
discr.inds3$Symb<-as.numeric(discr.inds3$Taxon)
discr.inds3$Col<-as.numeric(discr.inds3$Taxon)
levels(discr.inds3$Taxon)
discr.inds3<-recode.symb(discr.inds3,"ph",2)
discr.inds3<-recode.symb(discr.inds3,"ps",21)
discr.inds3<-recode.symb(discr.inds3,"st",3)
with(discr.inds3,plot(CCA1,CCA2,type="p",pch=Symb,col="black",bg="grey50",
   xlim=c(-5,5),ylim=c(-6,4.5),cex=1,cex.axis=1,cex.lab=1,
   xlab="discrim. axis 1",ylab="discrim. axis 2"))
abline(h=0,v=0,lty=2,col="black")
legend(3.75,4.5,c("ph","ps","st"),pch=c(2,21,3),col="black",bty="o",
   pt.bg="grey50",cex=0.9,pt.cex=1,ncol=1)
# characters (from discr.indb)
with(discr.indb,plot(CCA1,CCA2,type="n",asp=1,xlim=c(-0.7,0.65),
   ylim=c(-0.6,0.6)))
abline(h=0,v=0,lty=2,col="black")
with(discr.indb,arrows(0,0,CCA1,CCA2,length=0.15,lty=1,col="red"))
with(discr.indb[-c(1,2,3,13,14,16,18,19,21),],text(CCA1,CCA2,Character,pos=4,
   cex=0.75))
with(discr.indb[c(1,3,13,14,16,18,19,21),],text(CCA1,CCA2,Character,pos=2,
   offset=0.2,cex=0.75))
with(discr.indb[2,],text(CCA1,CCA2,Character,pos=3,offset=0.2,cex=0.75))
par(default.par)
```



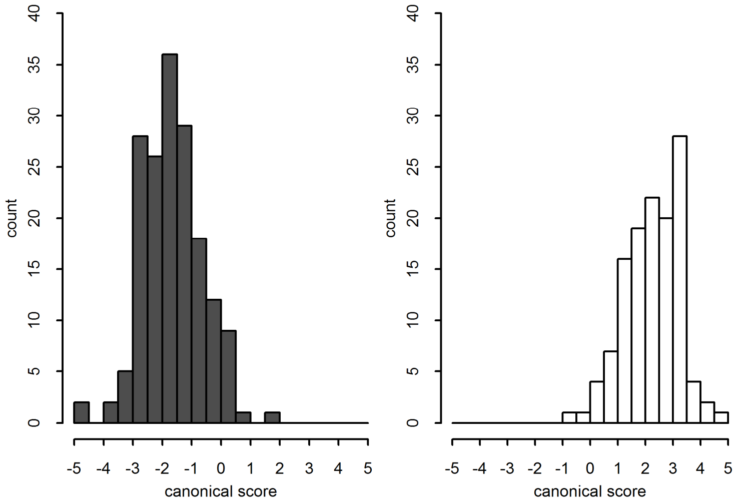
A scatterplot of individuals, with a focus on putative hybrids treated as passive samples:

```
discr.indsh<-discr.scores(discr.ind,indivx)
discr.indsh$Symb<-as.numeric(discr.indsh$Taxon)
discr.indsh$Col<-as.numeric(discr.indsh$Taxon)
levels(discr.indsh$Taxon)
discr.indsh<-recode.symb(discr.indsh,"ph",2)
discr.indsh<-recode.symb(discr.indsh, "ps",21)
discr.indsh<-recode.symb(discr.indsh,"st",3)
discr.indsh<-recode.symb(discr.indsh,"hybr",16)
discr.indsh<-recode.col(discr.indsh,"ph","grey50")
discr.indsh<-recode.col(discr.indsh,"ps","grey50")
discr.indsh<-recode.col(discr.indsh,"st","grey50")
discr.indsh<-recode.col(discr.indsh,"hybr","black")
with(discr.indsh,plot(CCA1,CCA2,type="p",pch=Symb,col=Col,bg="grey",
   xlim=c(-5,6),ylim=c(-6,5),cex=1,cex.axis=1,cex.lab=1,xlab="discrim. axis 1",
   ylab="discrim. axis 2"))
abline(h=0,v=0,lty=2,col="black")
legend(2.9,5.0,c("ph","ps","st","hybr"),pch=c(2,21,3,16),bty="o",
   col=c(rep("grey50",3),"black"),pt.bg="grey",cex=0.9,pt.cex=1,ncol=2)
```



If there are only two groups (for example, only *C. phrygia* and *C. pseudophrygia* from the sample data are analysed), there is only one canonical axis, and canonical scores are plotted as a histogram. The easiest way is to plot two separate histograms for each group:

```
par(mfrow=c(1,2))
with(discr.ind2s,hist(discr.ind2s[Taxon=="ps",4],plot=T,axes=F,freq=T,
   col="grey30",border="black",breaks=seq(-5.0,5.0,0.5),xlim=c(-5.0,5.0),
   ylim=c(0,40),main="",xlab="canonical score",ylab="count",cex.lab=0.75))
axis(1,at=(seq(-5.0,5.0,1)),labels=as.character(seq(-5.0,5.0,1)),tcl=-0.3,
   cex.axis=0.75)
axis(2,at=(seq(-0,40,5)),labels=as.character(seq(0,40,5)),tcl=-0.3,
   cex.axis=0.75)
with(discr.ind2s,hist(discr.ind2s[Taxon=="ph",4],plot=T,axes=F, freq=T,
   col="white",border="black",breaks=seq(-5.0,5.0,0.5),xlim=c(-5.0,5.0),
   ylim=c(0,40),main="",xlab="canonical score",ylab="count",cex.lab=0.75))
axis(1,at=(seq(-5.0,5.0,1)),labels=as.character(seq(-5.0,5.0,1)),tcl=-0.3,
   cex.axis=0.75)
axis(2,at=(seq(-0,40,5)),labels=as.character(seq(0,40,5)),tcl=-0.3,
   cex.axis=0.75)
par(mfrow=c(1,1))
```



The first parameter, `mfrow`, divides the drawing space into two parts (one row, two columns) in which two independent graphs can be plotted. The last line restores the default settings. The function `hist` plots a histogram. Most of the parameters are similar to those of the `plot` function, with some differences and specific parameters:

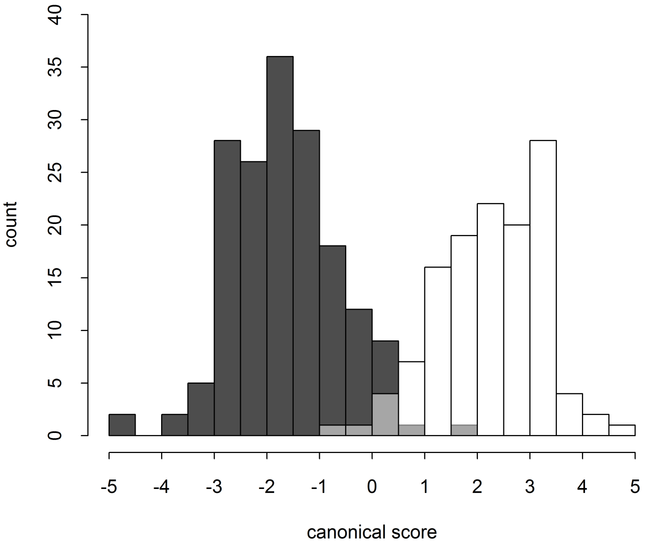
- `breaks` provides one way of defining histogram classes (columns). Here, the function `seq` is used (with three parameters: minimum, maximum, and step).
- `col` defines the inner fill of the columns, and the outline is defined by `border`.
- Axes must be drawn for the whole range of both groups, including the area in which no data are present (i.e., the area occupied by the other group). This situation is generally handled poorly by R functions. Therefore, it is better to suppress the axes to be drawn by the `hist` function (the parameter `axes=F)` and draw the axes manually using the `axis` function. The first parameter determines the axis to be drawn (1: x, 2: y), `at` and `tck` parameters define the positions and

size of the tickmarks, respectively, `labels` sets the labels for each tickmark, and `cex.labels` defines the size of the labels. The number of labels must be the same as the number of tickmarks, and any text string can be used as a label (i.e., `labels=c(“text1”,”text2”,…)` would be possible); here, both the position of tickmarks and the labels are generated by the function `seq` as a regular sequence from -5 to 5 by 1 (for the x-axis) and from 0 to 40 by 5 for (for the y-axis).

A histogram with semitransparent colours allows for both groups to be combined into one figure. However, note that in this case, the graph cannot be exported to a metafile. (This procedure does not support semitransparent colours.) Export to pdf or raster formats works well:

```
png("Discr2.png",width=16,height=15,units="cm",res=600)
with(discr.ind2s,hist(discr.ind2s[Taxon=="ps",4],plot=T,col="grey30",
   border="black",freq=T,breaks=seq(-5.0,5.0,0.5),xlim=c(-5.0,5.0),
   ylim=c(0,40),main="",xlab="canonical score",ylab="count",axes=F))
with(discr.ind2s,hist(discr.ind2s[Taxon=="ph",4],plot=T,col=rgb(1,1,1,0.5),
   border="black",freq=T,breaks=seq(-5.0,5.0,0.5),xlim=c(-5.0,5.0),
   axes=F,add=T))
axis(1,at=(seq(-5.0,5.0,1)),labels=as.character(seq(-5.0,5.0,1)),
   tcl=-0.3,cex.axis=1)
axis(2,at=(seq(-0,40,5)),labels=as.character(seq(0,40,5)),
   tcl=-0.3,cex.axis=1)
dev.off()
```

The first function, `hist`, plots the histogram for one group (preferably the darker colour); it is necessary to define the ranges of x- and y-axes so that they cover both groups, as in the case of the two separate histograms. The second call of the `hist` function draws the second histogram, which is semi-transparent, and adds it into the current plot (the parameter `add=T`). This semi-transparency is achieved by applying the function `rgb` to the `col` parameter of the `hist` function. The function `rgb` has four parameters: the first three are red, green and blue on a scale from 0 to 1, while the fourth parameter is opacity on a scale from 0 to 1. For colours, a scale from 0 to 255 can also be used, but then an additional parameter, `maxColorValue=255`, must be placed inside the `rgb` function. The axes are then added using the `axis` function.



Classificatory discriminant analysis

Classificatory discriminant analysis with cross-validation, based on the `lda` function from the package `MASS` (Venables & Ripley 2002) is used.

`classif.da` and `classif.da.1` return the original three columns from the data (ID, Population, Taxon), a column with the classification from discriminant analysis, the posterior probabilities of classification into each group (taxon) and a column with the correctness of classification (true/false). The results are formatted as a data frame and may be exported using the `export.res` function. The two functions differ in the mode of crossvalidation. `classif.da.1` uses the standard one-leave-out method. However, as some hierarchical structure is usually present in the data (individuals from a population are not completely independent observations, as they are closer to each other than to individuals from other populations), the function `classif.da` uses whole populations as leave-out units. This method does not allow classification if there is only one population for a taxon and is more sensitive to "atypical" populations, which usually leads to a somewhat lower classification success rate.

```
results<-classif.da(data)
results<-classif.da.1(data)
```

`classif.samp` conducts the classificatory discriminant analysis of a sample set based on an independent training set. This function may be used to classify hybrid populations, type herbarium specimens, etc. Note that there can be groups (taxa) in the sample set that are not present in the training set (such as the hybrids from the sample data, which were classified using data from the three pure taxa); however, the last column in the results is then meaningless (The classification will always be evaluated as "false".) The results can be exported using the `export.res` function.

```
results<-classif.samp(sample_data,training_data)
```

`classif.matrix` formats the results of the above three functions as a classification matrix of taxa. For each taxon from the original data, the function shows the number of classifications into all taxa present and the percentage of correct classifications; the total percentage of correct classifications over all taxa is also computed. The results can be exported using the `export.res` function.

```
matrix<-classif.matrix(results)
```

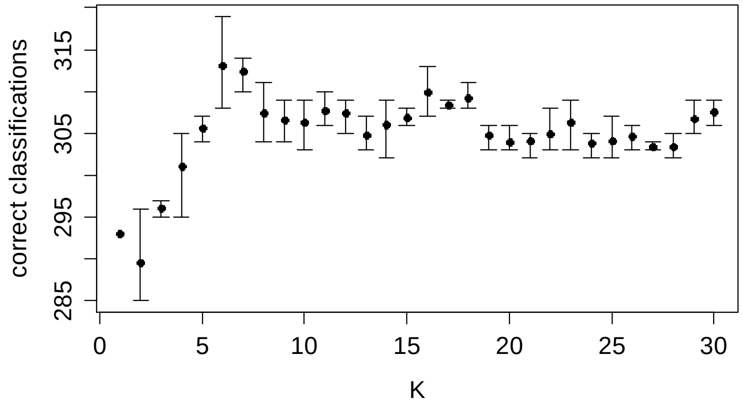
`classif.pmatrix` computes the classification matrix for populations; for each population, it shows the numbers of classifications into all taxa present in the data and the percentage of correct classifications. The results can be exported using the `export.res` function. The detailed classification of populations is very useful, as it can reveal some atypical or incorrectly assigned populations. For example, most of the populations (excluding hybrids) from the sample data are successfully classified using the `classif.da` function (generally over 70% correct classifications and often 100%), but two populations (BABL and LES) have only 31% and 15% of correct classifications, respectively; the first one is probably influenced by hybridization, while the latter is "atypical" within the sample dataset (it may represent different subspecies). When hybrid populations from the sample data are analysed (using `classif.samp` function), some individuals are usually classified as *C. pseudophrygia* and some as *C. stenolepis* (and none as *C. phrygia*) within each population, which clearly demonstrates the intermediacy of these populations.

```
matrix<-classif.pmatrix(results)
```

k-nearest neighbours classification

This method is used as a nonparametric alternative to classificatory discriminant analysis. It classifies an individual according to the *a priori* classification of its *k* neighbours (by Euclidean distance) using a majority vote. The functions are based on the `knn` and `knn.cv` functions from the package `class` (Venables & Ripley 2002). As the *k*-nearest neighbour method uses Euclidean distance, the characters are standardised to a zero mean and a unit variation in all of the functions below.

`knn.select` and `knn.select.1` search for the optimal *k* for the given data set. The functions only differ in cross-validation method in a similar way as the classificatory discriminant analysis functions. The functions the compute number of correctly classified individuals for *k* values from 1 to 30 and highlight the value with the highest success rate. Ties (i.e., when there are the same numbers of votes for two or more groups) are broken at random, and thus several iterations may yield different results. Therefore, the functions compute 10 iterations, and the average success rates for each *k* are used; the minimum and maximum success rates for each *k* are also displayed as error bars. Note that several *k* values may have nearly the same success rates; if this is the case, the similarity of iterations may also be considered. In the example displayed here, the highest average value is for *k* = 6, but individual solutions have both considerably higher and considerably lower success rates; in comparison, *k* = 7 has nearly the same average success rate, but the individual iterations are more similar to each other.



```
knn.select(data)
knn.select.1(data)
```

`knn.classif` and `knn.classif.1` perform knn classification for the specified *k*; differing only by cross-validation method (as above). The results are similar to those for classificatory discriminant analysis: they include the first three columns from the data (ID, Population, Taxon), a column with the knn classification, the proportion of votes for the "winning" group (analogous to the posterior probabilities in the discriminant analysis) and a column with the correctness of classifications (true/false). The results can be formatted as classification matrices using the `classif.matrix` or `classif.pmatrix` functions and both the knn results and the matrices can be exported using the `export.res` function.

```
results<-knn.classif(data,k)
results<-knn.classif.1(data,k)
matrix<-classif.matrix(results)
matrix<-classif.pmatrix(results)
```

`knn.samp` conducts knn classification from a sample set using an independent training set (similarly to the `classif.samp` function for classificatory discriminant analysis). The result can be accessed using the `classif.matrix` or `classif.pmatrix` functions and exported using the `export.res` function.

```
results<-knn.samp(sample_data,training_data,k)
matrix<-classif.pmatrix(results)
```

Acknowledgements

I am obliged to Petr Šmilauer, Jan Š. Lepš, Filip Kolář, Pavel Kúr and Tomáš Urfus for valuable advice on statistical methods used or for conducting some analyses in other software packages for comparison. Filip Kolář also came up with the name MorphoTools.

References

ter Braak C. J. F. & Šmilauer P. (2012): Canoco reference manual and user's guide: software for ordination (version 5.0). – Microcomputer Power, Ithaca.

Beckerman A. P. & Petchey O. L. (2012): Getting started with R: an introduction for biologists. – Oxford University Press, Oxford.

Crawley M. J. (2013): The R Book, 2nd edition. – Wiley, Hoboken.

Dray S. & Dufour A. B. (2007): The ade4 package: implementing the duality diagram for ecologists. – Journal of Statistical Software 22: 1–20.

Gross J. & Ligges U. (2012): nortest: Tests for Normality. R package, version 1.0-2. – URL: http://CRAN.R-project.org/package=nortest

Koutecký P. (2007): Morphological and ploidy level variation of *Centaurea phrygia* agg. (Asteraceae) in the Czech Republic, Slovakia and Ukraine. – Folia Geobot. 42: 77–102.

Koutecký P., Štěpánek J. & Baďurová T. (2012): Differentiation between diploid and tetraploid *Centaurea phrygia*: mating barriers, morphology and geographic distribution. – Preslia 84: 1–32.

Lepš J. & Šmilauer P. (2003): Multivariate analysis of ecological data using CANOCO. – Cambridge Univ. Press, Cambridge.

Ligges U. & Mächler M. (2003): scatterplot3d - an R package for visualizing multivariate data. – Journal of Statistical Software 8: 1–20.

Oksanen J., Blanchet F. G., Kindt R., Legendre P., Minchin P. R., O'Hara R. B., Simpson G. L., Solymos P., Stevens M. H. H. & Wagner H. (2013): vegan: community ecology package. Version 2.0-10. – URL: http://CRAN.R-project.org/package=vegan

R Core Team (2013). R: A language and environment for statistical computing. – R Foundation for Statistical Computing, Vienna, Austria. URL: http://www.R-project.org/.

Revelle W. (2013): psych: procedures for personality and psychological research, version 1.4.1. – Northwestern University, Evanston, Illinois, USA – URL: http://CRAN.R-project.org/package=psych

Venables W. N. & Ripley B. D. (2002): Modern Applied Statistics with S. Ed. 4. – Springer, New York.

Contents