

# 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](mailto:kouta@prf.jcu.cz)

Version 1.1

České Budějovice, Dec 2016

---

## 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. (2015): MorphoTools: a set of R functions for morphometric analysis. – Plant Systematics and Evolution 301: 1115-1121; doi 10.1007/s00606-014-1153-2

## Styles used in this manual

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

## Uptades

Compared to the previous version (1.01), the following changes were introduced:

- In cases there were several similar functions differing only in one or two parameters (such as reading the data with dot or comma as decimal separator) only one function is retained with additional parameters. This leads to fewer function at all and applies for: `read.morphodata`, `charhist`, `cormat`, `clust`, `classif.da`, `knn.select`, `knn.classif`.
- New functions were introduced to make graphical outputs from PCA and discriminant analysis: (a) scatterplots of ordination scores, (b) “arrow” diagram for characters, (c) biplots, (d) function adding labels to individual objects, (e) histograms of ordination scores for two-group discriminant analysis. For classificatory discriminant analysis, (f) a barplot of posterior probabilities is added. These functions replace or much shortens the original graphic scripts.
- In `read.morphodata` the spelling has been changed (from “morfodata”)
- The import function prints the structure `str(x)` of the imported data frame.
- `charhist` prints a taxon code as a part of the x-axis label.

## Files

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

- this manual (MorphoTools manual 1-1.pdf)
- function definitions (MorphoTools1-1.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 (MorphoTools working protocol 1-1.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).

## 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). If the data set does not contain population information, the second Population column should be included with any values (e.g.. copy the Taxon column there). The functions for populations will not work, but the others will. The columns ID, Population and Taxon are expected to be factors (categorical 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	...
...							

## 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 3.2.3.

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 do 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 `MorphoTools` 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, 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 (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 and small corrections of the data; 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 (including functions and all results you have created) in the current R environment:  
`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 can only partly 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))
```

- The parameter `mfrow=c(x, y)` divides the graphic window into specified number of subplots in  $x$  rows and  $y$  columns. Each subplot is then successively filled with one graph. This setting is typically used to produce several graphs of the same type or of related content (such as box-and-whisker plots for several morphological characters or plots of ordination scores for objects and correlations of characters with ordination axes).

```
par(mfrow=c(1,3))      # dividing the graphic window: three subplots in one row
  plot...              # graph 1
  plot...              # graph 2
  plot...              # graph 3
par(mfrow=c(1,1))      # restoring the default settings (one graph in the whole graphic window)
```

- 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)
```

```
  par...
  plot...
  axis...
dev.off()
```

## Typical workflow and functions

### *Data import and control*

`read.morphodata` imports data from tab-delimited text files; import from Excel spreadsheets through the clipboard in Windows also usually works well. The function is basically the `read.delim` and function, but it also renames the first three columns and convert them into factors if necessary. The decimal separator is controlled with a parameter `dec`: dot (default) or comma [in the earlier version of MorhoTools, this was made by two functions `read.morfodata` and `read.morpfodata2`, respectively]. If there is no population information, the population column may be omitted and the file imported with the parameter `popul=FALSE`; a copy of the taxon column will be appended as the population column.

```
data<-read.morphodata("file.txt")           # default decimal separator (dot)
data<-read.morphodata("file.txt", dec=",")   # setting decimal separator to comma
data<-read.morphodata("clipboard", popul=F)  # import from a clipboard, no population column
```

The functions also display the structure of the data using the function `str(data)`. The object's class (should be "data.frame"), dimensions, and a list of its columns and their types and also lists of populations and taxa are displayed. 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)
head(data); tail(data) # display the beginning or the end of the data, respectively
```

### *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 as a result of a function (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, COLUMNS]`. 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 4<sup>th</sup> 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 5<sup>th</sup>, 7<sup>th</sup> and 10<sup>th</sup> 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 three parameters: the name of the data to be exported, the name of the file to be created (“clipboard” is the default), and the decimal separator used (dot is the default but comma can also be used).

```
export.res(data) # makes export to the clipboard  
export.res(data, "clipboard") # the same  
export.res(data, "file.txt", dec=",") # makes export to a file; comma as the decimal separator
```



## 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)
```

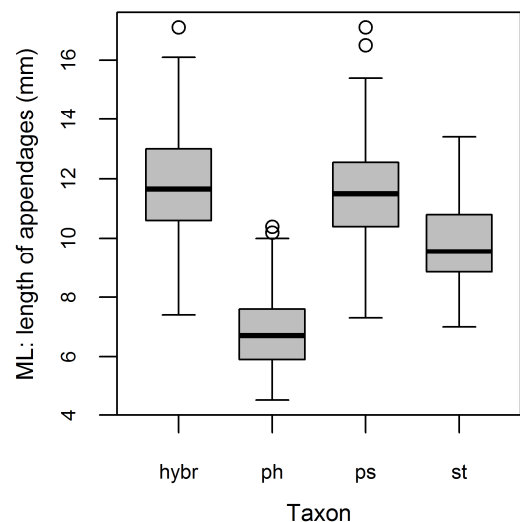
```
results3<-descr.pop(data)
```

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

```
boxplot(ML~Taxon,data=cent.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()` or consult e.g. <http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf>; 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`
  - `whisklty` – 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` plot histograms and the expected normal distribution of quantitative (not binary) characters, either for the whole dataset or for a specified taxon. The function show graphs for the characters sequentially and the next graph is initiated by pressing enter or clicking the mouse. The purpose of the function 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.

```
charhist\(data\) # the whole dataset (all taxa); default  
charhist\(data,taxon="taxon"\) # the same specified taxon  
charhist\(data,"taxon"\) # the same, shorter version
```

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` calculates the matrix of the correlation coefficients of the characters, Pearson's (default) or Spearman's. 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(data) # Pearson's correlation coefficients
results<-cormat(data,method="pearson") # the same
results<-cormat(data,method="spearman") # Spearman's correlation coefficients
```

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; see the function's help for details and use).

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

## Cluster analysis

`clust` performs agglomerative hierarchical clustering based on Euclidean distances. Typically, populations are used as OTUs. The function includes (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`. The clustering method is specified with the `method` parameter, one of: `average` (the default; can be also coded as `UPGMA`), `complete` or `ward`. For the Ward's method, there are two versions in R, the default (`ward.D`) is used. Note that there are other methods available but rarely used in morphometric analyses; see the help to the `hclust` function for details. Similarly, other than Euclidean distances might be implemented by a simple modification of the code (see the `dist` function).

```

results<-clust(data)                # UPGMA (default)
results<-clust(data,method="average") # the same, "UPGMA" can also be used
results<-clust(data,method="ward")   # Ward's method
results<-clust(data,method="complete") # complete linkage

```

**Dendrograms** are produced using `plot` (note that this is a generic R function, with settings depending on the class of the plotted object). For the populations clustered using Ward's method:

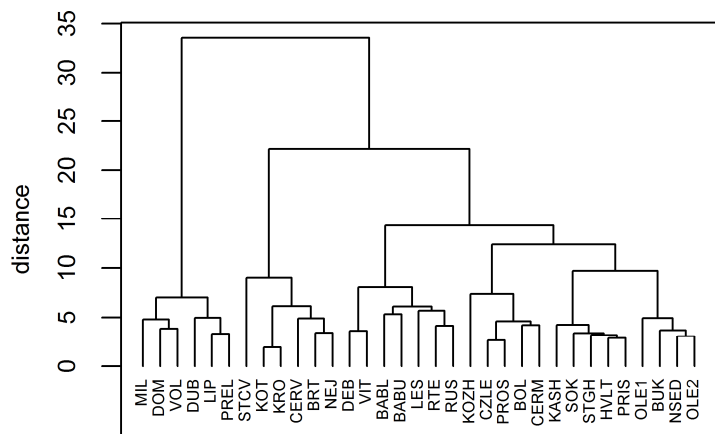
```

plot(results,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.



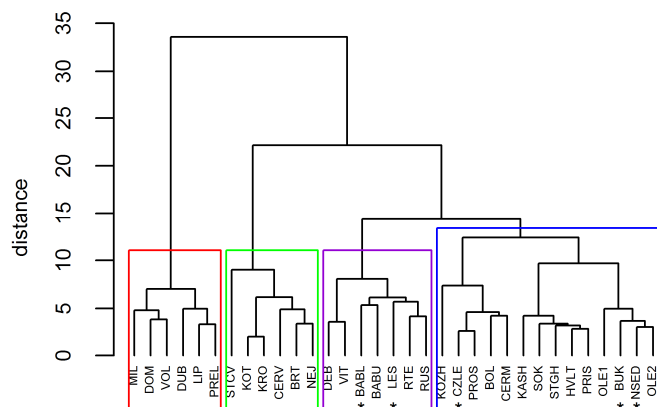
Rectangles highlighting certain clusters can be added using `rect.hclust`:

```

rect.hclust(results,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.



Additional content may be created using functions such as `text` or `arrows`. In the example figure, certain (“misclassified”) populations are highlighted by the symbol \* using `text`:

```
text(x=c(15,17,21,31,32),y=-4.3,"*",cex=1.25,col="black",adj=0.5)
```

- x and y coordinates of the text; might be single value or vectors when more text values is to be displayed; if the vectors differ in length, the shorter is recycled from the beginning; here, the y-coordinate is the same in all cases and the x-coordinates are positions of the populations to be highlighted
- the next parameter is the text (quoted) to be displayed; might also be a character vector, `c("text1","text2",...)` or as `.character(seq(1,5,0.5))` [numbers from 1 to 5 by 0.5]
- `cex` and `col` have their usual meaning
- `adj` is the horizontal adjustment of the text to the given coordinates (0: left adjusted; 0.5: centred; 1: right adjusted); note that there are other possibilities but are less suitable in this particular case
- many other parameters are available describing colour or font of the text, see the help to `text`

The use of `plot`, `rect.hclust` or `text` 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.hclust` function with different parameters can be used several times for one dendrogram. Similarly, `text` can be added in one or several calls. The example figure was created using this code (for comparison with the previous graph, `frame.plot=F` is used):

```
results<-cclust(data,method="ward")
par(mar=c(2,4.5,1,1))
plot(results,cex=0.6,cex.axis=0.8,cex.lab=0.8,frame.plot=F,hang=-0.1,
      main="",sub="",xlab="",ylab="distance")
rect.hclust(results,h=12,which=c(1,2,3),
            border=c("red","green","darkviolet"))
rect.hclust(results,h=14,which=4,border="blue")
text(x=c(15,17,21,31,32),y=-4.2,"*",cex=1,adj=0.5)
```

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 (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)
correlations<-pca.cor(results,n)
```

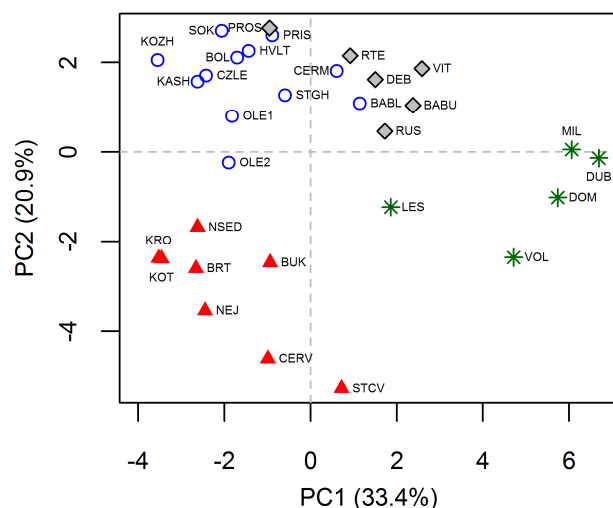
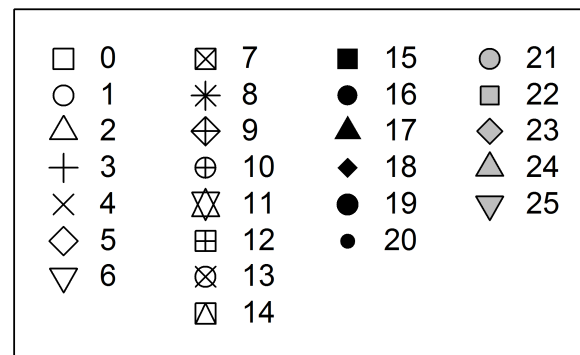
# extracts the first four principal components  
# extracts the specified *n* of principal components

`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.

`plot.scores` produces a scatterplot of the ordination scores, `point.labels` can add label to the data points, as described below. The function allows many graphical parameters of `plot`, with some additional features. Reference lines are added at 0 at both axes. Taxa can be marked with different symbols and/or colours. There are 21 different symbols available in R and 5 additional (21–25) symbols whose background colours can be set differently from their foreground colours; letters and many other characters can also be used as plotting symbols. Colours are coded by name (type `colors()` to see the list); 8 default colours may also be coded by numbers (1: black, 2: red, 3: green, 4: blue, 5: cyan, 6: magenta, 7: yellow, 8: grey). Symbol types (the `pch` parameter) and colours (the `col` parameter) can be generated automatically from levels in the variable Taxon using the function `as.numeric(scores$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 without a possibility of change. To use different symbols/colours, check levels of the Taxon variable and define vectors of values of `pch` and/or `col` for each level. Alternatively, these parameters can be defined directly in the `plot.scores` function call.

```
levels(scores$Taxon) # four levels present in the sample data; levels ordered alphabetically
[1] "hybr" "ph" "ps" "st"
taxon.symb<-c(23,17,1,8)
taxon.col<-c("black","red","blue","darkgreen")
plot.scores(scores,axes=c(1,3),pch=17,col=taxon.col,labels=T,legend=F)
```

The function `plot.scores` is then called, with the following parameters:

- `scores`, a dataframe with ordination scores.
- `axes` defines ordination axes to plot; the form is `axes=c(x,y)`, where `x` and `y` are number of the horizontal and vertical axis, `c(1,2)` is the default.
- `pch` and `col` define point symbol and colours, respectively. If omitted, default values are used [solid circle, `pch=16`, and colour created using `as.numeric(scores$Taxon)`]. Single value can be specified (the same for all taxa) or a vector of the same length as is the number of taxa can be provided; the two parameters are independent and all combinations of definitions can be used.
- `xlab` and `ylab` define labels of the `x` and `y` axis; if omitted, default labels “axis 1” and “axis 2”, respectively, are displayed.
- `labels` (logical, TRUE/FALSE; the latter is the default) defines whether the sample names taken from the `scores$ID` column are displayed. This is intended for a quick orientation in the results, for presentations / publications more detailed work with text labels is needed, as described below.
- `legend` (logical, TRUE/FALSE; the latter is the default) defines if legend is displayed, if so, the additional parameter `legend.pos` defines the legend position. The legend is generated from Taxon levels using `pch` and `col` specified in the function call. The legend position can be defined either as the position of the legend upper left corner as `c(x,y)` or by a single word defining the legend’s alignment to the plot, possible values are `center` (the centre of the plot, not useful) `left`, `right`, `top`, `bottom` or any combination of them; `legend.pos="topright"` is the default. However, note that the legend in this form is intended mainly for quick orientation in the plot in cases in which `col` or `pch` are generated automatically; for finer control it is possible to set `legend=FALSE` and create legend directly using the `legend` function, as described below.

Numerous other graphical parameters of the generic `plot` function can also be specified. The most important include:

- `bg` sets the background colour for the symbols 21–25.
- `xlim,ylim` define the minimum and maximum for the `x`- and-`y` axis, respectively, in the form `c(min,max)`. Extending the axes can be useful to create free space for a legend or point labels.
- `main` and `sub` have their usual meaning (the main title and the subtitle below the `x` axis).
- `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).

```
plot.scores(scores) # all parameters default
taxon.col<- c("black","red","blue","darkgreen")
plot.scores(scores,axes=c(1,3),pch=17,col=taxon.col,labels=T) # col from a vector
plot.scores(scores,axes=c(2,3),labels=T,pch=c(1,3,8,21),cex=0.8,
  col=c("black","red","blue","darkgreen"),bg="grey") # both pch and col defined here
```

Text descriptions of data points (taken from the ID column of the dataframe with ordination scores) may be added using the function `point.labels`. This is only a wrapper for the `text` function that allows convenient work with only selected labels; i.e., for different groups of labels, different parameters (position, colour, size, etc.) can be defined.

```
point.labels(scores,axes=c(1,2),cex=0.8,col="red",pos=4,offset=0.4)
```

The function has four parameters:

- `scores` and `axes` are the same as in `plot.scores` (`axes` must be specified again here to get the coordinates of the labeled points).
- `names` defines the labels to work with; should be a character vector or `c(label1, label2, ...)`; if omitted, all labels are influenced. Alternatively, the labels can be selected using indices (row numbers) to `scores`.
- `excl` (logical) defines whether only specified labels are processed (FALSE; the default) or whether the specified labels are excluded (TRUE).

Numerous graphical parameters of the `text` function can be used

- `col` and `cex` have their usual meaning
- `font` defines the style of the text (1: plain, 2: bold, 3: italics, 4: bold italics)
- `pos` is the position of the label relative to its coordinates (1: below, 2: to the left, 3: above, 4: to the right; last is the default).
- `offset` – the offset (in fraction of character width) of the label from the position specified by `pos`; the default is 0.5.
- `adj` provides an alternative way of adjusting the label position; note that it is overridden with `pos` and thus `pos=NULL` must be specified. The parameter `adj` (single value for x axis or `c(x, y)` for both axes) have the values from 0 (left/bottom) to 1 (right/top); 0.5 means centred and values outside the interval  $<0, 1>$  will usually also work to make the label more distant from the point.

```
point.labels(scores, axes=c(1,3), cex=0.5, col="blue") # 1st and 3rd axis, all labels
point.labels(scores, names=c("POP1", "POP2"), excl=T) # all but two named populations
point.labels(scores, names=c("POP1", "POP2"), pos=1) # the two populations, below points
point.labels(scores[scores$ID==c(1,2,5), ], pos=2) # selecting populations using indices
```

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

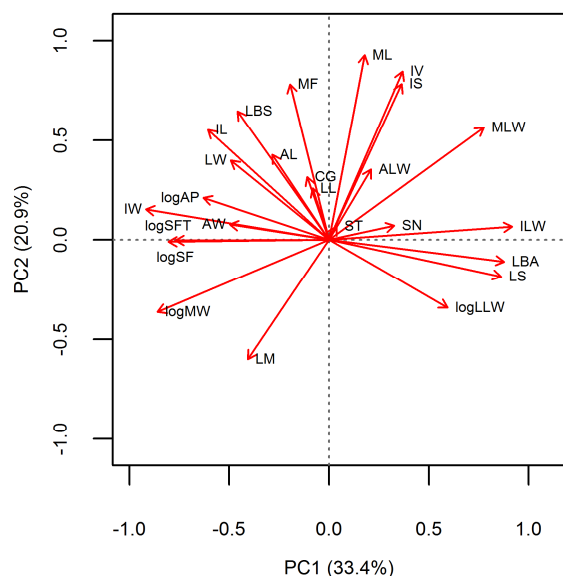
```
results<-pca.calc(data)
scores<-pca.scores(results, data)
plot.scores(scores, xlab="PC1 (33.4%)", ylab="PC2 (20.9%)", pch=c(23,17,1,8),
  col=c("black", "red", "blue", "darkgreen"), bg="grey", cex=1.25, xlim=c(-4,6.7),
  legend=F)
point.labels(scores,
  names=c("PROS", "SOK", "KASH", "BOL", "KRO", "DUB", "MIL", "CERM", "KOZH", "KOT"),
  excl=T, cex=0.75)
point.labels(scores, names=c("PROS", "SOK", "KASH", "BOL", "CERM"), pos=2, cex=0.75)
point.labels(scores, names=c("KRO", "MIL", "KOZH"), pos=3, offset=0.7, cex=0.75)
point.labels(scores, names=c("DUB", "KOT"), pos=1, offset=0.7, cex=0.75)
```

`plot.characters` draws character loadings as arrows. The function uses the base functions `plot` and `arrows` and allows most of their graphical parameters. The character names can be added using the `point.labels` function, as described above.

```
plot.characters(loadings, axes=c(1,3),
  col="black", length=0.05, labels=T)
```

The most important parameters include:

- `loadings`, a dataframe with the character loadings (the results of the `pca.cor` function).
- `axes=c(x,y)` defines ordination axes to plot; `c(1,2)` is the default.
- `col` is colour of the arrows, red is the default
- `lty` defines the arrow line style (1: solid (default), 2: dashed, 3: dotted, 4: dotdash, etc).
- `length` and `angle` defines the arrow head; the default values are 0.1 and 20, respectively





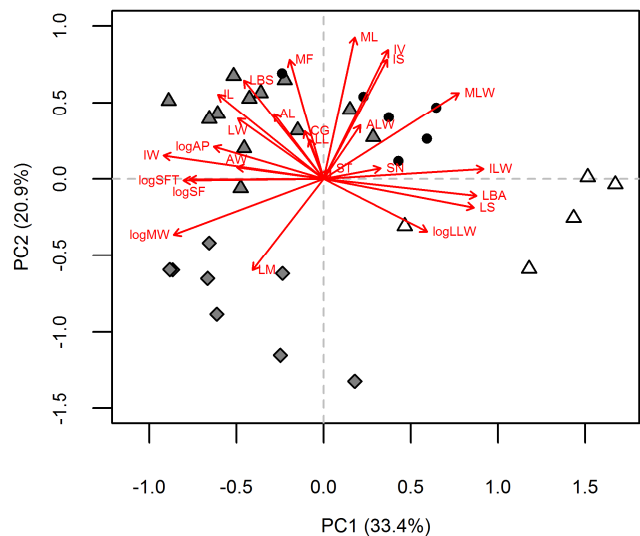
- `xlim`, `ylim` (`c(-1,1)` is the default), `xlab`, `ylob`, `cex`, etc. have their usual meaning
- `labels` (logical, TRUE/FALSE; the former is the default) defines whether the sample names taken from the `loadings$Character` column are displayed. This is intended for a quick orientation in the results, for presentation quality figures more detailed work with text labels using the `point.labels` function is needed, as described below.

The code used to produce the sample figure is:

```
loadings<-pca.cor(results)
plot.characters(loadings,length=0.05,xlab="PC1 (33.4%)",ylab="PC2 (20.9%)",
  labels=F)
point.labels(loadings,names=c("LW","logAP","IW","AW","logSF","logSFT"),
  excl=I,cex=0.75)
point.labels(loadings,names=c("LW","logAP","IW","AW"),cex=0.75,pos=2,
  offset=0.2)
point.labels(loadings,names=c("logSFT"),cex=0.75,pos=3)
point.labels(loadings,names=c("logSF"),cex=0.75,pos=1)
```

`plot.biplot` is the function to produce a biplot combining sample scores and character loadings. The function combines the `plot.scores` and `plot.characters` functions with the parameters:

- The first two parameters are `scores` and `loadings`
- Since the axes ranges are usually different between sample scores and characters (such as `<-4,6.5>` and `<-1,1>` for the x-axis, respectively, in the population sample data), it is usually necessary to rescale one of them. The scaling factor is defined by the `scale.fact` parameter: 0 – no change, > 0 sample scores are divided by the scaling factor, the scaling of axes for characters is retained; <0 character loadings are divided by the (absolute value) of the scaling factor, the scaling of axes for sample scores is retained.
- The other parameters are the same as in the `plot.scores` and `plot.characters` functions, except for the colour parameter that has to be specified separately as `col.point` and `col.arrows` (red as the default); `labels` and `legend` are set FALSE as the default.
- Labels can be added both to point and character arrows separately using the `point.labels` function, as described above.



```
plot.biplot(scores,loadings,pch=c(16,23,24,2),col.points="black",bg="grey50",
  scale.fact=4,cex=1.5,xlim=c(-1,1.7),ylim=c(-1.5,1),xlab="PC1 (33.4%)",
  ylab="PC2 (20.9%)")
point.labels(loadings,
  names=c("LW","logAP","IW","AW","logSF","logSFT","logMw"),excl=I,cex=0.75)
point.labels(loadings,names=c("logAP","IW","AW","logMw"),cex=0.75,pos=2,
  offset=0.2)
point.labels(loadings,names=c("LW","logSFT"),cex=0.75,pos=3)
point.labels(loadings,names=c("logSF"),cex=0.75,pos=1)
```

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. Note also that there are other function to produce biplots available in R, such as `biplot` and `biplot.princomp`.

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) and creating legend by the separate command is demonstrated:

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

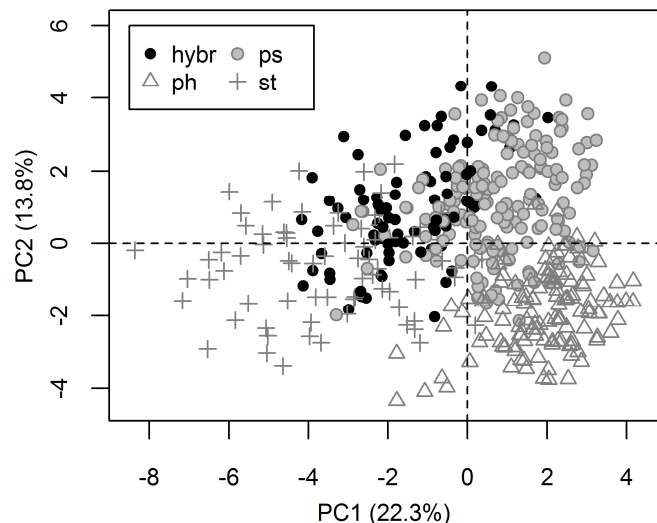
```
scores<-pca.scores(results,data)
```

```
plot.scores(scores,pch=c(16,2,21,3),col=c("black",rep("grey30",3)),bg="grey",
  cex=1.25,xlim=c(-8.5,4.3),ylim=c(-4.5,6),xlab="PC1 (22.3%)",
  ylab="PC2 (13.8%)",legend=F)
```

```
legend(-8.5,6.0,levels(scores$Taxon),pch=c(16,2,21,3),
  col=c("black",rep("grey30",3)),bty="o",pt.bg="grey",cex=0.9,pt.cex=1,ncol=2)
```

The parameters of `legend`:

- The first two arguments indicate the position (x and y) 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.scores` function. Alternatively, legend position can be defined by a single word, as described above (top, bottom, right, left, and their combinations).
- 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 (only be careful to use the correct order respective to the labels).
- `col` – colour of symbols. Note that I used the function `rep` here, which has two arguments: what to repeat and how many times: `rep("grey30",3)` is equivalent to `"grey30","grey30","grey30"`. The colours can also be generated automatically in the same way as colours of the data points, i.e. the first level (hybr) will have colour 1 (“black”), the second (“ph”) the colour 2 (red), etc.; the command must extract levels (taxa) from the `Taxon` column and convert them from a character vector (does not have levels defined) to a factor: `col=as.factor(levels(scores$Taxon))`
- `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.

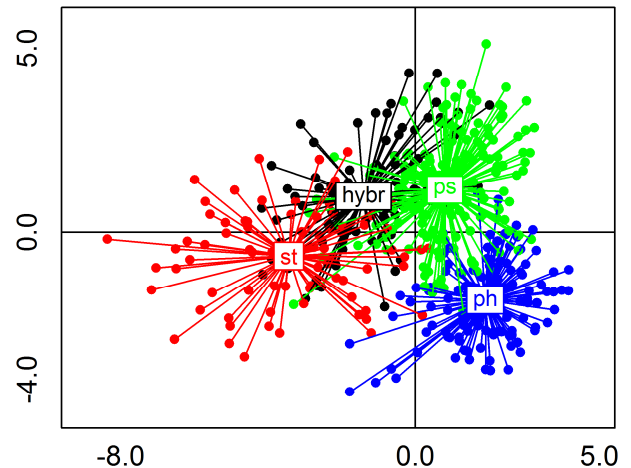


An interesting possibility for showing the overlap of groups is the “spider” diagram of the function `s.class` from the `ade4` package. Note that this package also includes other graphing functions (such as `s.arrow` for characters and `scatter` for biplots), as well as its own PCA function, `dudi.pca`.

```
library(ade4)
par(oma=c(2,2,1,1))
s.class(scores, scores$Taxon, xax=4, yax=5, pch=16,
        col=c("black", "blue", "green", "red"), label=levels(scores$Taxon), 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"))
```

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 the 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 `tc1` 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. If they should be different for each taxon, it is necessary to extract Taxon column from the dataframe, replace the values with colours/symbols and then use the modified vector in a function call with the parameters `pch` or `col`. Note that for 2D plots, this operation is included within the function `plot.scores`.

```

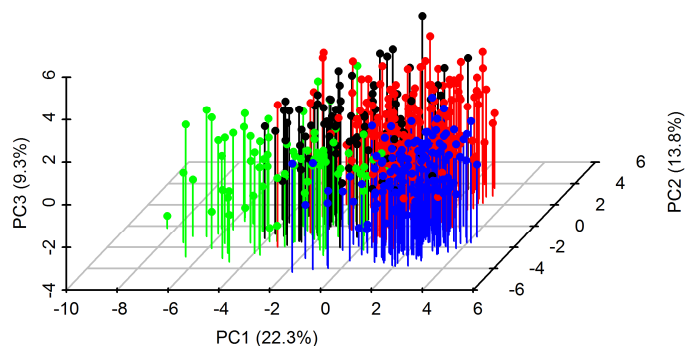
levels(scores$Taxon) # prints the levels of the Taxon column on a screen
taxa<-levels(scores$Taxon) # extracts the levels of the Taxon column to a vector
taxa.colours<-c("black","blue","red","green") # defines the colour for each taxon
taxa.ind<-as.character(scores$Taxon) # extracts the Taxon column to a character vector
# the following two lines makes the replacement (might be one line, then omit the brackets {} )
for (i in 1:length(taxa)) {
  taxa.ind<-gsub((taxa)[i],taxa.colours[i],individ.taxon)}
# if numbers are used as codes (such as for symbols), one additional line is needed
taxa.ind<-as.numeric(taxa.ind)
# 3D plot
library(scatterplot3d)
with(scores,scatterplot3d(PC1,PC2,PC3,color=taxa.ind,pch=16,type="h",
  lty.hp=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%)"))

```

First, the `scatterplot3d` package is loaded using the `library` command. The `scatterplot3d` function is then run; for simpler writing of the column names, it can be wrapped in the `with` function. This function has two parameters: an object to work with and the command to run. When not using `with`, it would be necessary to use the full reference to the columns with ordination scores, such as `scores$PC1` instead of `PC1`, etc.

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 known 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 centred but unstandardised characters [for characters A, B,..., the equation is  $(A - \text{mean}(A)) * \text{coef A} + (B - \text{mean}(B)) * \text{coef B} + \dots$ ]. 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.

```
dscores<-discr.taxa(results) # not using scores here as it is a function in vegan
```

`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.

```
dscores<-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.

```
dscores<-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 (but not identical) 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.

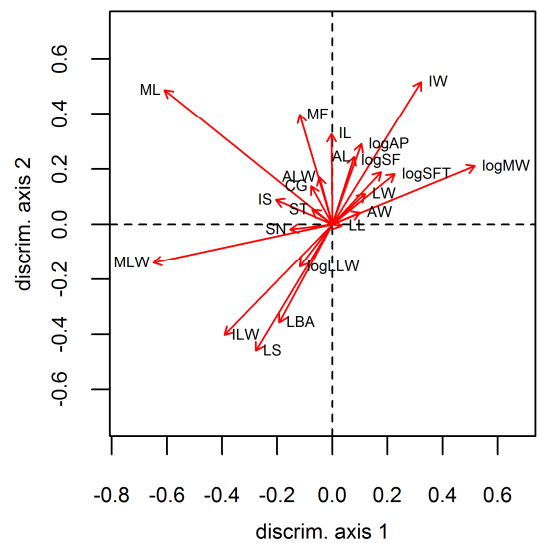
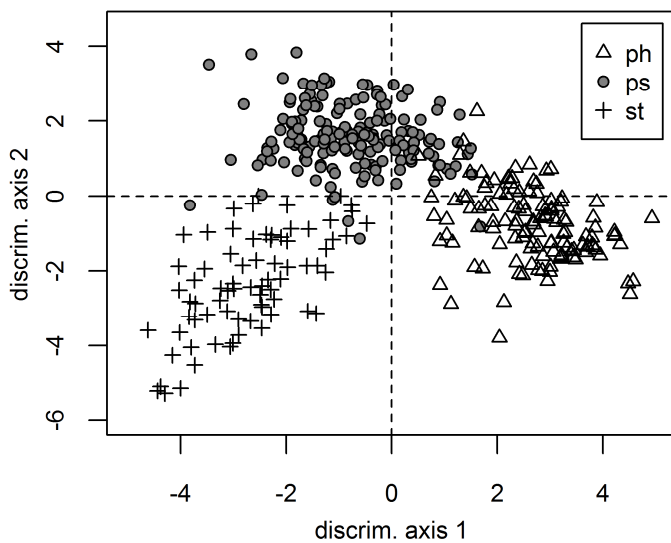
```
discr.step(results)                # default settings
discr.step(results,perm=200)       # reducing the number of permutations
discr.step(results,perm=1000,dir="both",p=0.05/22)
                                     # all parameters set manually, Bonferroni correction of
                         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

Graphics with three and more groups can be produced in a similar way as in PCA, using the functions `plot.scores`, `plot.characters`, `plot.biplot` and `plot.labels`; see the examples below. For two groups, there is only one canonical axis and the ordination scores are better presented as histograms (sample scores) / bar plots (characters).



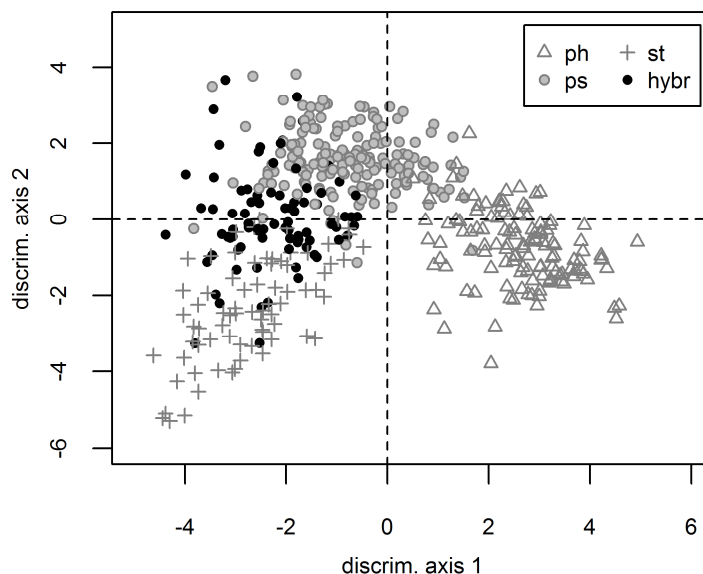
Discriminant analysis of the three species, hybrids excluded:

```
# individuals
levels(dscores$Taxon) # ph ps st
plot.scores(dscores, pch=c(2,21,3), col="black", bg="grey50",
  xlab="discrim. axis 1", ylab="discrim. axis 2", legend=F)
legend(3.6,4.4, c("ph", "ps", "st"), pch=c(2,21,3), col="black", pt.bg="grey50",
  bty="o", cex=1, pt.cex=1, ncol=1)
# characters
plot.characters(characters, xlab="discrim. axis 1", ylab="discrim. axis 2",
  xlim=c(-0.8,0.7), ylim=c(-0.6,0.6))
point.labels(characters,
  names=c("logSF", "CG", "ALW", "ST", "IS", "SN", "MLW", "AL"), excl=I,
  cex=0.8, offset=0.4)
point.labels(characters, names=c("logSF"), pos=3, cex=0.8, offset=0.4)
point.labels(characters, names=c("ALW", "CG", "ST", "IS", "SN", "MLW", "AL"),
  pos=2, cex=0.8, offset=0.3)
```



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

```
plot.scores(dscores, pch=c(16,2,21,3), col=c("black", rep("grey50", 3)), bg="grey",
  xlab="discrim. axis 1", ylab="discrim. axis 2", xlim=c(-5,6), ylim=c(-6,5),
  legend=E)
legend(2.9,5, c("ph", "ps", "st", "hybr"), pch=c(2,21,3,16),
  col=c(rep("grey50", 3), "black"), pt.bg="grey", bty="o", cex=0.9, pt.cex=1, ncol=2)
```

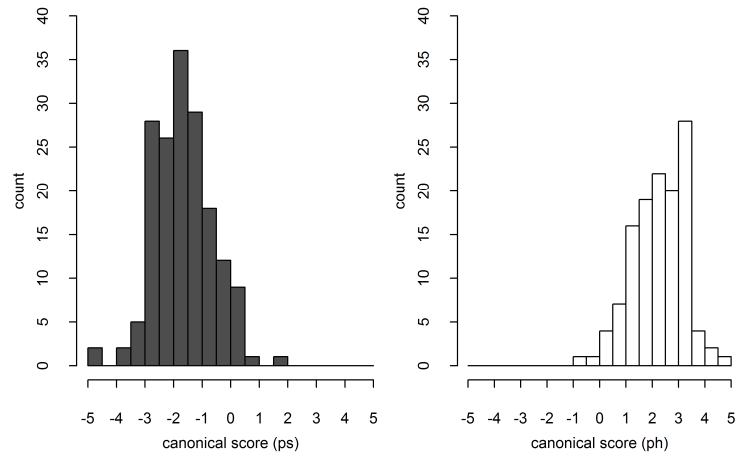


If there are only two groups (for example, only *C. phrygia* and *C. pseudophrygia* from the sample data are analysed), there is only one canonical axis, and canonical scores are plotted as a histogram. One possibility is to a separate histogram for each group, both histogram having the same axes; this is done by the function `discr.2hist`:

```
discr.2hist(dscores,breaks=seq(-5,5,0.5),labels=seq(-5,5,1),ylim=c(0,40),
            col=c("grey30","white"),tcl=-0.3,tcl.min=0)
```

The parameters of the function are:

- `scores` – a dataframe with the ordination scores. The function internally computes means for both groups to draw the group with lower values first; the group codes (from the taxon column) are displayed in the x-axis label.
- `breaks` defines the range of the x-axis (to cover all values in the data) and the step; here it is generated using the function `seq` as `seq(min,max,step)`. If omitted, default values generated by the `hist` function are used.
- `labels` defines the labels on the x-axis. When omitted, all values defined with `breaks` are labelled. In the example above, the labels are generated so that only every second position is labelled.
- `col` and `border` define colours of the fill and the outline of the bars, respectively. When omitted, default settings (from `par`) are used (usually used for `border`, where black is the default). When single value is supplied, it will be used for both histograms. When there are two values `c(value1,value2)`, they will be used for the left and the right histogram, respectively.
- `tcl` defines size of the labelled (main) tickmarks, the negative values cause the tickmarks to be oriented outside of the plot (down and left on the x- and y-axis, respectively); `tcl=-0.5` is the default.
- `tcl.min` defines the size of unlabelled (minor) tickmarks in the similar way; `tcl.min=0` can be used to suppress them at all.
- `ylim` defines the range of y axis, as `c(0,max)`. When omitted, it is determined by the function internally as the maximum count within an interval rounded up to the nearest ten.
- other usual graphical parameters can be specified, such as `cex.axis` and `cex.labels`. The y-axis limit (`ylim`) is usually different for both groups (depending on the number of observations available), it is therefore useful to define it manually and re-run the function; it will then be common for both groups.



Both histograms can be combined into one using semi-transparent colours to show the groups overlap; see the next page.

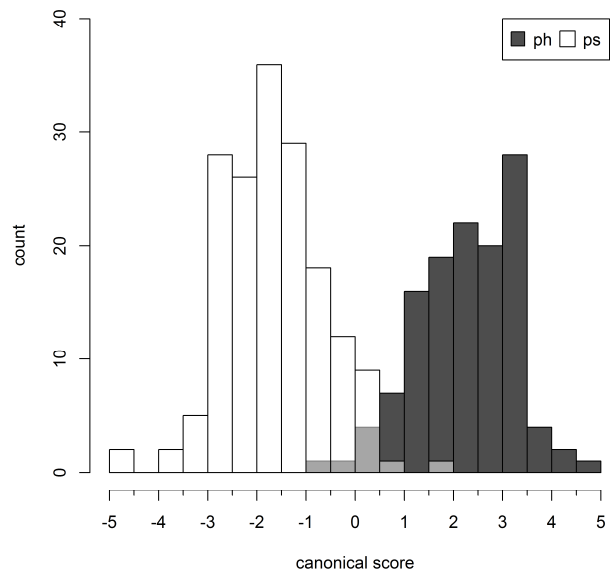
A histogram with semi-transparent colours allows for both groups to be combined into one figure using the `discr.hist` function; the semi-transparent colour showing the overlaps. However, note that in this case, the graph cannot be exported to a metafile (does not support semi-transparent colours). Here, the script defines export to .png (the first and the last line); the plot margins (and many other `par` settings) should be defined after the `png` call, because many general `par` settings are not effective for the PNG graphical device (similar also `tiff`, `bmp` or `jpg`, as described in the introduction).

```
png("file.png",width=16,height=14,units="cm",res=600)
par(mar=c(4.5,4.5,1,1)) # reducing the upper and right margins
discr.hist(dscores,breaks=seq(-5,5,0.5),labels=seq(-5,5,1),tcl.min=0.25,
  col=c("grey30","white",0.5),legend=F)
legend("topright",levels(dscores$Taxon),pch=22,pt.bg=c("grey30","white"),
  pt.cex=2,bty="o",ncol=2)
dev.off()
```

Many parameters are similar to the previous function `discr.2hist`:

- the first parameters is the dataframe with the ordination scores.
- `breaks`, `labels`, `tcl`, `tcl.min` and `ylim` are similar as in the two histogram function.
- `col` has three values that defines colours of the first and second histogram and opacity for the second. The order of taxa is respective to `levels(scores$Taxon)`, i.e. to an alphabetical order of the taxon codes. Opacity has the range from 0 (fully transparent) to 1 (no transparency), 0.5 is a good starting value.
- Other graphical parameters such as `border`, `cex.axis` or `cex.lab` may be supplied.
- The logical parameter `legend` controls whether a legend is drawn (`FALSE` is the default); if `TRUE` it draws a simple legend in the upper right corner (`legend.pos="topright"`). The definition of the legend's position is similar to the `plot.scores` function; the text labels are taken from `levels(scores$Taxon)`. However, it is intended mainly as a quick information on what colour is used for what taxon; for presentation quality figures it is possible to set the automatic legend off and use the `legend` command. The parameters of `legend` are similar as elsewhere and only a few specific features are commented here:
  - It is useful to set `pch=22`, i.e. a square that can be filled with a specified colour.
  - The outline colour is controlled with the `col` parameter and should be the same as `border` in the `discr.hist` call; usually both can be omitted and the default foreground colour is used (black if you have not changed it using `par`).
  - Size of the symbols (squares) is controlled by the parameters `pt.cex`, the values 1.5 or 2 usually works well.
  - The fill is defined by the `pt.bg` parameter and the same two colours as used for the histogram (the `col` parameter of the `discr.hist` function) should be used. However, the tone of the second colour (if other than white) depends on opacity and should be adjusted using the function `adjustcolor` with two parameters set (colour and opacity; note that the function has many other parameters that are not used in this particular case).

```
legend(...,pt.bg=c("colour1",adjustcolor("colour2",alpha.f=opacity)),...)
```



## 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` returns 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 mode of cross-validation is set by the parameter `crossval`. The default "indiv" (alternatively also "ind" or "ID") 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 value "pop" sets whole populations as leave-out units. The latter 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)           # default settings
results<-classif.da(data,crossval="indiv") # the same
results<-classif.da(data,crossval="ind")  # the same
results<-classif.da(data,crossval="ID")   # the same
results<-classif.da(data,crossval="pop")  # populations as leave-out units
```

`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, entirely new data, 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.

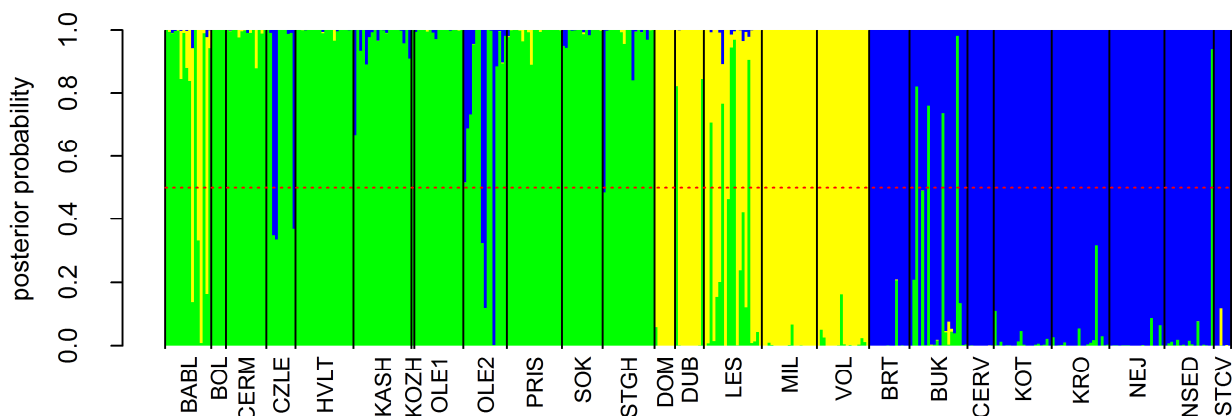
```
matrix<-classif.pmatrix(results)
```

`classif.barplot` allow graphical output of classificatory discriminant analyses. The function draws a barplot with posterior probabilities of classification of each individual into each group (from the results of the `classif.da` function). This can help to identify incorrectly or uncertainly (correctly, but with not a high probability) classified individual or populations on a first glance. The parameters of the function are as follows:

- The data frame with classification results (from the function `classif.da` or `classif.samp`). Note that if there are more than several hundreds of individuals, the bars will be very narrow and it will probably be useful to split the data into several figures (e.g. selecting data for individual taxa using subscripts).  
When plotted, the data are sorted alphabetically according to taxon codes and then population names if different order is not defined by the following two parameters.
- `tax.ord` sets the user specified order in the plot, as `c("taxon1", "taxon2", ...)`, using the taxa codes from the Taxon column of the data.
- `poplabels` define population labels, as an existing character vector or `c("population1", "population2", ...)`. If omitted, the population names are taken from the Population column of the data. If used, populations will be drawn in the specified order, overriding the `tax.ord` settings or the default order.
- `col` defines colours of each taxon; the order of taxa is either alphabetical or that defined by the `tax.ord` parameter. If omitted, the default method of the `barplot` function is used, which is a grey-scale palette.
- `col.lines` defines colour of lines separating individual populations; the standard foreground colour (which is usually black) is the default.
- `cex.axis`, `cex.lab` and `tcl` define size of the y-axis values, labels and tickmarks in a usual way
- `cex.pop` define size of population labels (x-axis)
- `ylab` has its usual meaning, "posterior prob." is the default
- `ref.line` (logical) defines whether a reference line at certain probability (y-axis) is drawn; the default is FALSE.
- `ref.pars` is a list (see the example below) of additional parameters to the reference line; the parameters can be: `y` (the probability value to be used, 0.5 is the default) and `col`, `lwd` and `lty` with their usual meaning (line colour, width and type; the defaults are foreground, 1 and 2 (dashed), respectively).

An example is the classification of 3 species (without hybrids) from the sample data, obtained using `classif.da` with `crossval="indiv"`:

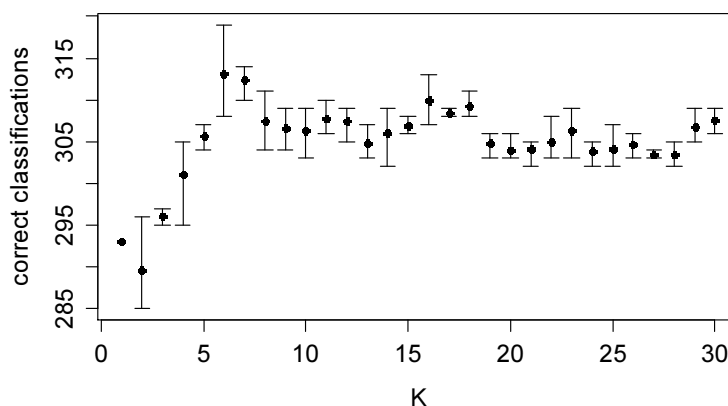
```
classif.barplot(classification, tax.ord=c("ps", "st", "ph"),
  col=c("green", "yellow", "blue"), cex.axis=0.8, cex.lab=0.8, cex.pop=0.8,
  ref.line=T, ref.pars=list(col="red", lty=3))
```



## *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` search for the optimal *k* for the given data set. The functions can use two cross-validation methods (by individuals and populations) in a similar way as the classificatory discriminant analysis functions. The functions compute the 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)
```

```
# default settings
```

```
knn.select(data, crossval="indiv")
```

```
# the same
```

```
knn.select(data, crossval="pop")
```

```
# populations as leave-out units
```

`knn.classif` performs *knn* classification for the specified *k*; the cross-validation method is specified 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(data,k,crossval="indiv")
```

```
results<-knn.classif(data,k,crossval="pop")
```

```
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, 2<sup>nd</sup> 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

Introduction.....	1
Files.....	1
Sample data.....	2
Installation.....	3
Introductory notes .....	3
Data structure .....	2
Typical workflow and functions .....	5
Data import and control .....	5
Basic data manipulation .....	5
Missing data .....	6
Exporting results .....	6
Descriptive statistics .....	7
Population means .....	7
Distribution of characters, transformations.....	8
Correlations of characters .....	9
Cluster analysis .....	10
Principal component analysis (PCA) .....	12
Canonical discriminant analysis.....	19
Classificatory discriminant analysis.....	23
k-nearest neighbours classification .....	25
Acknowledgements.....	27
References.....	27