Design decisions and implementation details in vegan

Jari Oksanen

processed with vegan 2.7-3 in R version 4.5.2 (2025-10-31) on March 3, 2026

Abstract

This document describes design decisions, and discusses implementation and algorithmic details in some vegan functions. The proper FAQ is another document.

Contents

1 Parallel processing **1**
1.1 User interface 1
1.1.1 Using parallel processing as default 1
1.1.2 Setting up socket clusters . . 2
1.1.3 Random number generation . 2
1.1.4 Does it pay off? 2
1.2 Internals for developers 2

2 Nestedness and Null models **3**
2.1 Matrix temperature 3

3 Scaling in redundancy analysis **4**

4 Weighted average and linear combination scores **5**
4.1 LC Scores are Linear Combinations . 6
4.2 Factor constraints 9
4.3 Conclusion 9

1 Parallel processing

Several **vegan** functions can perform parallel processing using the standard R package **parallel**. The **parallel** package in R implements the functionality of earlier contributed packages **multicore** and **snow**. The **multicore** functionality forks the analysis to multiple cores, and **snow** functionality sets up a socket cluster of workers. The **multicore** functionality only works in unix-like systems (such as MacOS and Linux), but **snow** functionality works in all operating systems. **Vegan** can use either method, but defaults to **multicore** functionality when this is available, because its forked clusters are usually faster. This chapter describes both the user interface and internal implementation for the developers.

1.1 User interface

The functions that are capable of parallel processing have argument `parallel`. The normal default is `parallel = 1` which means that no parallel processing is performed. It is possible to set parallel processing as the default in **vegan** (see § 1.1.1).

For parallel processing, the `parallel` argument can be either

1. An integer in which case the given number of parallel processes will be launched (value 1 launches non-parallel processing). In unix-like systems (*e.g.*, MacOS, Linux) these will be forked `multicore` processes. In Windows socket clusters will be set up, initialized and closed.

2. A previously created socket cluster. This saves time as the cluster is not set up and closed in the function. If the argument is a socket cluster, it will also be used in unix-like systems. Setting up a socket cluster is discussed in § 1.1.2.

1.1.1 Using parallel processing as default

If the user sets option `mc.cores`, its value will be used as the default value of the `parallel` argument in **vegan** functions. The following command will set up parallel processing to all subsequent **vegan** commands:

```
> options(mc.cores = 2)
```

The `mc.cores` option is defined in the **parallel** package, but it is usually unset in which case **vegan** will default to non-parallel computation. The `mc.cores` option can be set by the environmental variable `MC_CORES` when the **parallel** package is loaded.

R allows setting up a default socket cluster (`setDefaultCluster`), but this will not be used in **vegan**.

1.1.2 Setting up socket clusters

If socket clusters are used (and they are the only alternative in Windows), it is often wise to set up a cluster before calling parallelized code and give the pre-defined cluster as the value of the `parallel` argument in **vegan**. If you want to use socket clusters in unix-like systems (MacOS, Linux), this can be only done with pre-defined clusters.

If socket cluster is not set up in Windows, **vegan** will create and close the cluster within the function body. This involves following commands:

```
clus <- makeCluster(4)
## perform parallel processing
stopCluster(clus)
```

The first command sets up the cluster, in this case with four cores, and the second command stops the cluster.

Most parallelized **vegan** functions work similarly in socket and fork clusters, but in `oecosimu` the parallel processing is used to evaluate user-defined functions, and their arguments and data must be made known to the socket cluster. For example, if you want to run in parallel the `meandist` function of the `oecosimu` example with a pre-defined socket cluster, you must use:

```
> ## start up and define meandist()
> library(vegan)
> data(sipoo)
> meandist <-
      function(x) mean(vegdist(x, "bray"))
> library(parallel)
> clus <- makeCluster(4)
> clusterEvalQ(clus, library(vegan))
> mbc1 <- oecosimu(dune, meandist, "r2dtable",
                   parallel = clus)
> stopCluster(clus)
```

Socket clusters are used for parallel processing in Windows, but you do not need to pre-define the socket cluster in `oecosimu` if you only need **vegan** commands. However, if you need some other contributed packages, you must pre-define the socket cluster also in Windows with appropriate `clusterEvalQ` calls.

If you pre-set the cluster, you can also use **snow** style socket clusters in unix-like systems.

1.1.3 Random number generation

Vegan does not use parallel processing in random number generation, and you can set the seed for the standard random number generator. Setting the seed for the parallelized generator (L'Ecuyer) has no effect in **vegan**.

1.1.4 Does it pay off?

Parallelized processing has a considerable overhead, and the analysis is faster only if the non-parallel code is really slow (takes several seconds in wall clock time). The overhead is particularly large in socket clusters (in Windows). Creating a socket cluster and evaluating `library(vegan)` with `clusterEvalQ` can take two seconds or longer, and only pays off if the non-parallel analysis takes ten seconds or longer. Using pre-defined clusters will reduce the overhead. Fork clusters (in unix-likes operating systems) have a smaller overhead and can be faster, but they also have an overhead.

Each parallel process needs memory, and for a large number of processes you need much memory. If the memory is exhausted, the parallel processes can stall and take much longer than non-parallel processes (minutes instead of seconds).

If the analysis is fast, and function runs in, say, less than five seconds, parallel processing is rarely useful. Parallel processing is useful only in slow analyses: large number of replications or simulations, slow evaluation of each simulation. The danger of memory exhaustion must always be remembered.

The benefits and potential problems of parallel processing depend on your particular system: it is best to rely on your own experience.

1.2 Internals for developers

The implementation of the parallel processing should accord with the description of the user interface above (§ 1.1). Function `oecosimu` can be used as a reference implementation, and similar

interpretation and order of interpretation of arguments should be followed. All future implementations should be consistent and all must be changed if the call heuristic changes.

The value of the `parallel` argument can be `NULL`, a positive integer or a socket cluster. Integer 1 means that no parallel processing is performed. The "normal" default is `NULL` which in the "normal" case is interpreted as 1. Here "normal" means that R is run with default settings without setting `mc.cores` or environmental variable `MC_CORES`.

Function `oecosimu` interprets the `parallel` arguments in the following way:

1. `NULL`: The function is called with argument `parallel = getOption("mc.cores")`. The option `mc.cores` is normally unset and then the default is `parallel = NULL`.

2. Integer: An integer value is taken as the number of created parallel processes. In unix-like systems this is the number of forked multicore processes, and in Windows this is the number of workers in socket clusters. In Windows, the socket cluster is created, and if needed `library(vegan)` is evaluated in the cluster (this is not necessary if the function only uses internal functions), and the cluster is stopped after parallel processing.

3. Socket cluster: If a socket cluster is given, it will be used in all operating systems, and the cluster is not stopped within the function.

This gives the following precedence order for parallel processing (highest to lowest):

1. Explicitly given argument value of `parallel` will always be used.

2. If `mc.cores` is set, it will be used. In Windows this means creating and stopping socket clusters. Please note that the `mc.cores` is only set from the environmental variable `MC_CORES` when you load the **parallel** package, and it is always unset before first `require(parallel)`.

3. The fall back behaviour is no parallel processing.

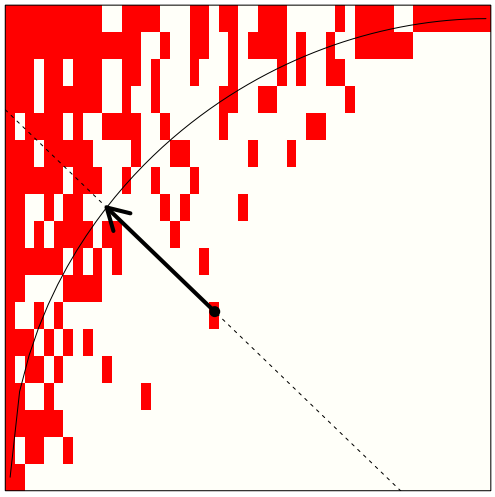

Figure 1: Matrix temperature for *Falco subbuteo* on Sibbo Svartholmen (dot). The curve is the fill line, and in a cold matrix, all presences (red squares) should be in the upper left corner behind the fill line. Dashed diagonal line of length D goes through the point, and an arrow of length d connects the point to the fill line. The "surprise" for this point is $u = (d/D)^2$ and the matrix temperature is based on the sum of surprises: presences outside the fill line or absences within the fill line.

2 Nestedness and Null models

Some published indices of nestedness and null models of communities are only described in general terms, and they could be implemented in various ways. Here I discuss the implementation in **vegan**.

2.1 Matrix temperature

The matrix temperature is intuitively simple (Fig. 1), but the the exact calculations were not explained in the original publication (Atmar and Patterson, 1993). The function can be implemented in many ways following the general principles. Rodríguez-Gironés and Santamaria (2006) have seen the original code and reveal more details of calculations, and their explanation is the basis of the implementation in **vegan**. However, there are still some open issues, and probably **vegan** func-

tion `nestedtemp` will never exactly reproduce results from other programs, although it is based on the same general principles.[1] I try to give main computation details in this document — all details can be seen in the source code of `nestedtemp`.

- Species and sites are put into unit square (Rodríguez-Gironés and Santamaria, 2006). The row and column coordinates will be $(k - 0.5)/n$ for $k = 1 \ldots n$, so that there are no points in the corners or the margins of the unit square, and a diagonal line can be drawn through any point. I do not know how the rows and columns are converted to the unit square in other software, and this may be a considerable source of differences among implementations.

- Species and sites are ordered alternately using indices (Rodríguez-Gironés and Santamaria, 2006):

$$\begin{aligned} s_j &= \sum_{i|x_{ij}=1} i^2 \\ t_j &= \sum_{i|x_{ij}=0} (n-i+1)^2 \end{aligned} \tag{1}$$

  Here x is the data matrix, where 1 is presence, and 0 is absence, i and j are row and column indices, and n is the number of rows. The equations give the indices for columns, but the indices can be reversed for corresponding row indexing. Ordering by s packs presences to the top left corner, and ordering by t pack zeros away from the top left corner. The final sorting should be "a compromise" (Rodríguez-Gironés and Santamaria, 2006) between these scores, and **vegan** uses $s+t$. The result should be cool, but the packing does not try to minimize the temperature (Rodríguez-Gironés and Santamaria, 2006). I do not know how the "compromise" is defined, and this can cause some differences to other implementations.

- The following function is used to define the fill line:

$$y = (1 - (1 - x)^p)^{1/p} \tag{2}$$

  This is similar to the equation suggested by Rodríguez-Gironés and Santamaria (2006, eq. 4), but omits all terms dependent on the numbers of species or sites, because I could not understand why they were needed. The differences are visible only in small data sets. The y and x are the coordinates in the unit square, and the parameter p is selected so that the curve covers the same area as is the proportion of presences (Fig. 1). The parameter p is found numerically using R functions `integrate` and `uniroot`. The fill line used in the original matrix temperature software (Atmar and Patterson, 1993) is supposed to be similar (Rodríguez-Gironés and Santamaria, 2006). Small details in the fill line combined with differences in scores used in the unit square (especially in the corners) can cause large differences in the results.

- A line with slope $= -1$ is drawn through the point and the x coordinate of the intersection of this line and the fill line is found using function `uniroot`. The difference of this intersection and the row coordinate gives the argument d of matrix temperature (Fig. 1).

- In other software, "duplicated" species occurring on every site are removed, as well as empty sites and species after reordering (Rodríguez-Gironés and Santamaria, 2006). This is not done in **vegan**.

3 Scaling in redundancy analysis

This chapter discusses the scaling of scores (results) in redundancy analysis and principal component analysis performed by function `rda` in the **vegan** library.

Principal component analysis decomposes a centred data matrix $\mathbf{X} = \{x_{ij}\}$ into K orthogonal components so that $x_{ij} = \sqrt{n-1}\sum_{k=1}^{K} u_{ik}\sqrt{\lambda_k}v_{jk}$, where u_{ik} and v_{jk} are orthonormal coefficient matrices and λ_k are eigenvalues. In **vegan** the eigenvalues sum up to variance of the data, and therefore we need to multiply with the square root of degrees of freedom $n-1$. Orthonormality means that sums of squared columns is one and their cross-

[1] function `nestedness` in the **bipartite** package is a direct port of the original BINMATNEST program of Rodríguez-Gironés and Santamaria (2006).

product is zero, or $\sum_i u_{ik}^2 = \sum_j v_{jk}^2 = 1$, and $\sum_i u_{ik}u_{il} = \sum_j v_{jk}v_{jl} = 0$ for $k \neq l$. This is a decomposition, and the original matrix is found exactly from the singular vectors and corresponding singular values, and first two singular components give the rank = 2 least squares estimate of the original matrix.

The coefficients u_{ik} and v_{jk} are scaled to unit length for all axes k. Eigenvalues λ_k give the information of the importance of axes, or the 'axis lengths.' Instead of the orthonormal coefficients, or equal length axes, it is customary to scale species (column) or site (row) scores or both by eigenvalues to display the importance of axes and to describe the true configuration of points. Table 1 shows some alternative scalings. These alternatives apply to principal components analysis in all cases, and in redundancy analysis, they apply to species scores and constraints or linear combination scores; weighted averaging scores have somewhat wider dispersion.

In community ecology, it is common to plot both species and sites in the same graph. If this graph is a graphical display of PCA, or a graphical, low-dimensional approximation of the data, the graph is called a biplot. The graph is a biplot if the transformed scores satisfy $x_{ij} = c\sum_k u_{ij}^* v_{jk}^*$ where c is a scaling constant. In functions `princomp`, `prcomp` and `rda` with `scaling = "sites"`, the plotted scores define a biplot so that the eigenvalues are expressed for sites, and species are left unscaled.

There is no natural way of scaling species and site scores to each other. The eigenvalues in redundancy and principal components analysis are scale-dependent and change when the data are multiplied by a constant. If we have percent cover data, the eigenvalues are typically very high, and the scores scaled by eigenvalues will have much wider dispersion than the orthonormal set. If we express the percentages as proportions, and divide the matrix by 100, the eigenvalues will be reduced by factor 100^2, and the scores scaled by eigenvalues will have a narrower dispersion. For graphical biplots we should be able to fix the relations of row and column scores to be invariant against scaling of data. The solution in R standard function `biplot` is to scale site and species scores independently, and typically very differently (Table 1), but plot each independently to fill the graph area. The solution in Canoco and `rda` is to use proportional eigenvalues $\lambda_k / \sum \lambda_k$ instead of original eigenvalues. These proportions are invariant with scale changes, and typically they have a nice range for plotting two data sets in the same graph.

The **vegan** package uses a scaling constant $c = \sqrt[4]{(n-1)\sum \lambda_k}$ in order to be able to use scaling by proportional eigenvalues (like in Canoco) and still be able to have a biplot scaling. Because of this, the scaling of `rda` scores is non-standard. However, the `scores` function lets you to set the scaling constant to any desired values. It is also possible to have two separate scaling constants: the first for the species, and the second for sites and friends, and this allows getting scores of other software or R functions (Table 2).

The scaling is controlled by three arguments in the `scores` function in **vegan**:

1. `scaling` with options `"sites"`, `"species"` and `"symmetric"` defines the set of scores which is scaled by eigenvalues (Table 1).
2. `const` can be used to set the numeric scaling constant to non-default values (Table 2).
3. `correlation` can be used to modify species scores so that they show the relative change of species abundance, or their correlation with the ordination (Table 1). This is no longer a biplot scaling.

4 Weighted average and linear combination scores

Constrained ordination methods such as Constrained Correspondence Analysis (CCA) and Redundancy Analysis (RDA) produce two kind of site scores (ter Braak, 1986; Palmer, 1993):

- LC or Linear Combination Scores which are linear combinations of constraining variables.
- WA or Weighted Averages Scores which are such weighted averages of species scores that are as similar to LC scores as possible.

Many computer programs for constrained ordinations give only or primarily LC scores following recommendation of Palmer (1993). However, functions `cca` and `rda` in the **vegan** package use primarily WA scores. This chapter explains the reasons for this choice.

Table 1: Alternative scalings for RDA used in the functions `prcomp` and `princomp`, and the one used in the **vegan** function `rda` and the proprietary software Canoco scores in terms of orthonormal species (v_{ik}) and site scores (u_{jk}), eigenvalues (λ_k), number of sites (n) and species standard deviations (s_j). In `rda`, const $= \sqrt[4]{(n-1)\sum\lambda_k}$. Corresponding negative scaling in **vegan** is derived dividing each species by its standard deviation s_j (possibly with some additional constant multiplier).

	Site scores u^*_{ik}	**Species scores** v^*_{jk}
`prcomp, princomp`	$u_{ik}\sqrt{n-1}\sqrt{\lambda_k}$	v_{jk}
`stats::biplot`	u_{ik}	$v_{jk}\sqrt{n}\sqrt{\lambda_k}$
`stats::biplot, pc.biplot=TRUE`	$u_{ik}\sqrt{n-1}$	$v_{jk}\sqrt{\lambda_k}$
`rda, scaling="sites"`	$u_{ik}\sqrt{\lambda_k/\sum\lambda_k}\times$ const	$v_{jk}\times$ const
`rda, scaling="species"`	$u_{ik}\times$ const	$v_{jk}\sqrt{\lambda_k/\sum\lambda_k}\times$ const
`rda, scaling="symmetric"`	$u_{ik}\sqrt[4]{\lambda_k/\sum\lambda_k}\times$ const	$v_{jk}\sqrt[4]{\lambda_k/\sum\lambda_k}\times$ const
`rda, correlation=TRUE`	u^*_{ik}	$\sqrt{\sum\lambda_k/(n-1)}s_j^{-1}v^*_{jk}$

Table 2: Values of the `const` argument in **vegan** to get the scores that are equal to those from other functions and software. Number of sites (rows) is n, the number of species (columns) is m, and the sum of all eigenvalues is $\sum_k\lambda_k$ (this is saved as the item `tot.chi` in the `rda` result)

	Scaling	**Species constant**	**Site constant**
vegan	any	$\sqrt[4]{(n-1)\sum\lambda_k}$	$\sqrt[4]{(n-1)\sum\lambda_k}$
`prcomp`, `princomp`	1	1	$\sqrt{(n-1)\sum_k\lambda_k}$
Canoco v3	`-1, -2, -3`	$\sqrt{n-1}$	$\sqrt{n}$
Canoco v4	`-1, -2, -3`	$\sqrt{m}$	$\sqrt{n}$

Briefly, the main reasons are that

- LC scores *are* linear combinations, so they give us only the (scaled) environmental variables. This means that they are independent of vegetation and cannot be found from the species composition. Moreover, identical combinations of environmental variables give identical LC scores irrespective of vegetation.

- McCune (1997) has demonstrated that noisy environmental variables result in deteriorated LC scores whereas WA scores tolerate some errors in environmental variables. All environmental measurements contain some errors, and therefore it is safer to use WA scores.

This article studies mainly the first point. The users of **vegan** have a choice of either LC or WA (default) scores, but after reading this article, I believe that most of them do not want to use LC scores, because they are not what they were looking for in ordination.

4.1 LC Scores are Linear Combinations

Let us perform a simple CCA analysis using only two environmental variables so that we can see the constrained solution completely in two dimensions:

```
> library(vegan)
> data(varespec)
> data(varechem)
> orig <- cca(varespec ~ Al + K, varechem)
```

Function `cca` in **vegan** uses WA scores as default. So we must specifically ask for LC scores (Fig. 2).

```
> plot(orig, dis=c("lc","bp"))
```

What would happen to linear combinations of LC scores if we shuffle the ordering of sites in species data? Function `sample()` below shuffles the indices.

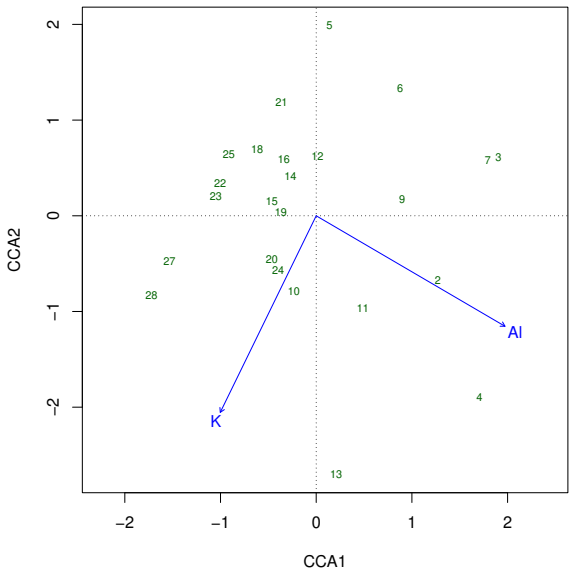

Figure 2: LC scores in CCA of the original data.

```
> i <- sample(nrow(varespec))
> shuff <- cca(varespec[i,] ~ Al + K, varechem)
```

It seems that site scores are fairly similar, but oriented differently (Fig. 3). We can use Procrustes rotation to see how similar the site scores indeed are (Fig. 4).

```
> plot(procrustes(scores(orig, dis="lc"),
                  scores(shuff, dis="lc")))
```

There is a small difference, but this will disappear if we use Redundancy Analysis (RDA) instead of CCA (Fig. 5). Here we use a new shuffling as well.

```
> tmp1 <- rda(varespec ~ Al + K, varechem)
> i <- sample(nrow(varespec)) # Different shuffling
> tmp2 <- rda(varespec[i,] ~ Al + K, varechem)
```

LC scores indeed are linear combinations of constraints (environmental variables) and *independent of species data*: You can shuffle your species data, or change the data completely, but the LC scores will be unchanged in RDA. In CCA the LC scores are *weighted* linear combinations with site totals of species data as weights. Shuffling species data in CCA changes the weights, and this can cause changes in LC scores. The magnitude of changes depends on the variability of site totals.

The original data and shuffled data differ in their goodness of fit:

```
> orig
```

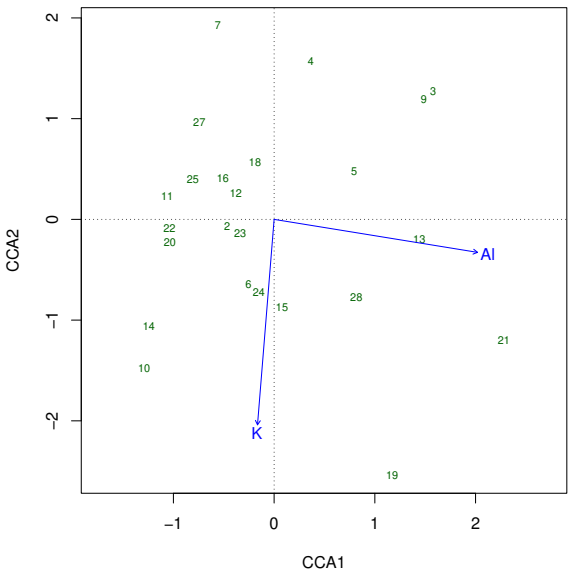

Figure 3: LC scores of shuffled species data.

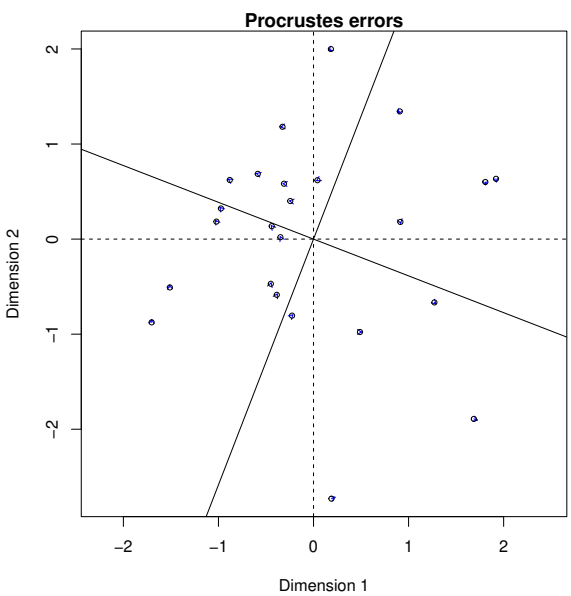

Figure 4: Procrustes rotation of LC scores from CCA of original and shuffled data.

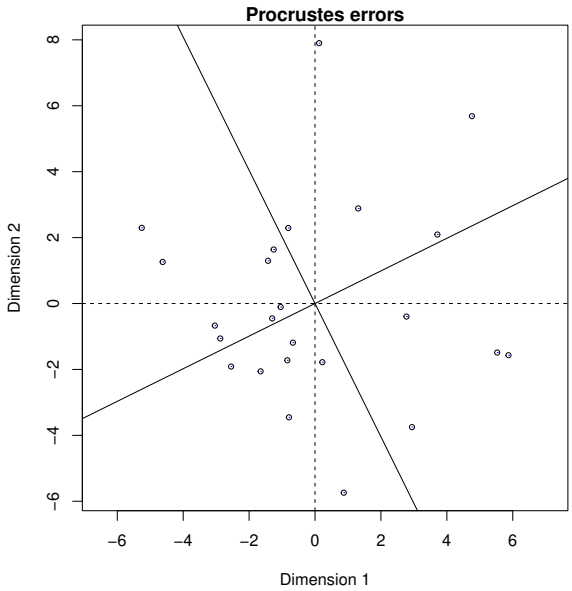

Figure 5: Procrustes rotation of LC scores in RDA of the original and shuffled data.

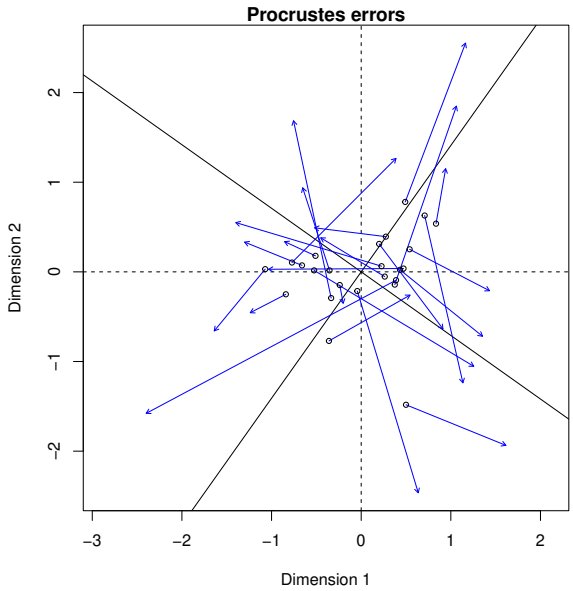

Figure 6: Procrustes rotation of WA scores of CCA with the original and shuffled data.

```
Call: cca(formula = varespec ~ Al + K, data =
varechem)

               Inertia Proportion Rank
Total           2.0832     1.0000
Constrained     0.4760     0.2285    2
Unconstrained   1.6072     0.7715   21

Inertia is scaled Chi-square

Eigenvalues for constrained axes:
  CCA1   CCA2
0.3608 0.1152

Eigenvalues for unconstrained axes:
   CA1    CA2    CA3    CA4    CA5    CA6    CA7
0.3748 0.2404 0.1970 0.1782 0.1521 0.1184 0.0836
   CA8
0.0757
(Showing 8 of 21 unconstrained eigenvalues)
```

```
> shuff
```

```
Call: cca(formula = varespec[i, ] ~ Al + K, data =
varechem)

               Inertia Proportion Rank
Total          2.08320    1.00000
Constrained    0.18780    0.09015    2
Unconstrained  1.89540    0.90985   21

Inertia is scaled Chi-square

Eigenvalues for constrained axes:
   CCA1    CCA2
0.09594 0.09186

Eigenvalues for unconstrained axes:
   CA1    CA2    CA3    CA4    CA5    CA6    CA7
0.4863 0.3296 0.2156 0.1823 0.1628 0.1119 0.0864
   CA8
0.0843
(Showing 8 of 21 unconstrained eigenvalues)
```

Similarly their WA scores will be (probably) very different (Fig. 6).

The example used only two environmental variables so that we can easily plot all constrained axes. With a larger number of environmental variables the full configuration remains similarly unchanged, but its orientation may change, so that two-dimensional projections look different. In the full space, the differences should remain within numerical accuracy:

```
> tmp1 <- rda(varespec ~ ., varechem)
> tmp2 <- rda(varespec[i,] ~ ., varechem)
> proc <- procrustes(scores(tmp1, dis="lc", choi=1:14),
                     scores(tmp2, dis="lc", choi=1:14))
> max(residuals(proc))
```

```
[1] 2.793485e-14
```

In `cca` the difference would be somewhat larger than now observed 2.7935e-14 because site weights used for environmental variables are shuffled with the species data.

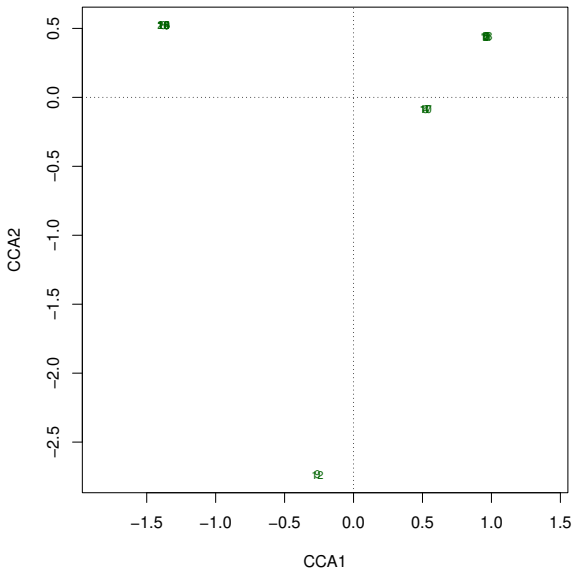

Figure 7: LC scores of the dune meadow data using only one factor as a constraint.

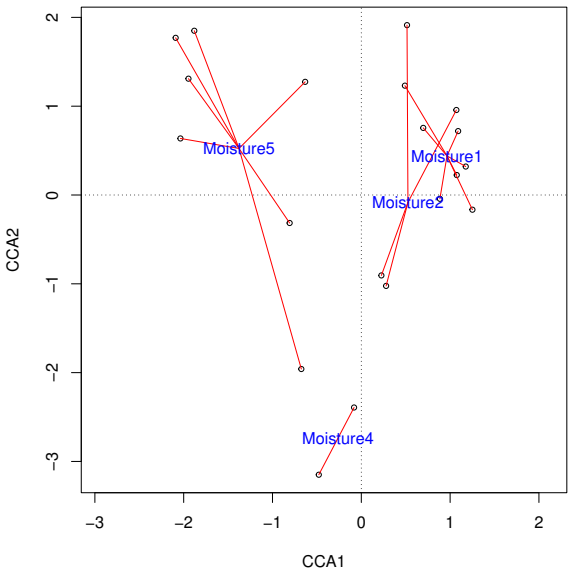

Figure 8: A "spider plot" connecting WA scores to corresponding LC scores. The shorter the web segments, the better the ordination.

4.2 Factor constraints

It seems that users often get confused when they perform constrained analysis using only one factor (class variable) as constraint. The following example uses the classical dune meadow data (Jongman *et al.*, 1987):

```
> data(dune)
> data(dune.env)
> orig <- cca(dune ~ Moisture, dune.env)
```

When the results are plotted using LC scores, sample plots fall only in four alternative positions (Fig. 7). In the previous chapter we saw that this happens because LC scores *are* the environmental variables, and they can be distinct only if the environmental variables are distinct. However, normally the user would like to see how well the environmental variables separate the vegetation, or inversely, how we could use the vegetation to discriminate the environmental conditions. For this purpose we should plot WA scores, or LC scores and WA scores together: The LC scores show where the site *should* be, the WA scores shows where the site *is*.

Function `ordispider` adds line segments to connect each WA score with the corresponding LC (Fig. 8).

```
> plot(orig, display="wa", type="points")
> ordispider(orig, col="red")
> text(orig, dis="cn", col="blue")
```

This is the standard way of displaying results of discriminant analysis, too. Moisture classes `1` and `2` seem to be overlapping, and cannot be completely separated by their vegetation. Other classes are more distinct, but there seems to be a clear arc effect or a "horseshoe" despite using CCA.

4.3 Conclusion

LC scores are only the (weighted and scaled) constraints and independent of vegetation. If you plot them, you plot only your environmental variables. WA scores are based on vegetation data but are constrained to be as similar to the LC scores as only possible. Therefore **vegan** calls LC scores as `constraints` and WA scores as `site scores`, and uses primarily WA scores in plotting. However, the user makes the ultimate choice, since both scores are available.

References

Atmar W, Patterson BD (1993). "The measure of order and disorder in the distribution of species in fragmented habitat." *Oecologia*, **96**, 373–382.

Jongman RH, ter Braak CJF, van Tongeren OFR (1987). *Data analysis in community and landscape ecology.* Pudoc, Wageningen.

McCune B (1997). "Influence of noisy environmental data on canonical correspondence analysis." *Ecology*, **78**, 2617–2623.

Palmer MW (1993). "Putting things in even better order: The advantages of canonical correspondence analysis." *Ecology*, **74**, 2215–2230.

Rodríguez-Gironés MA, Santamaria L (2006). "A new algorithm to calculate the nestedness temperature of presence–absence matrices." *Journal of Biogeography*, **33**, 921–935.

ter Braak CJF (1986). "Canonical correspondence analysis: a new eigenvector technique for multivariate direct gradient analysis." *Ecology*, **67**, 1167–1179.