Permutation tests

Gavin L. Simpson

April 27, 2023

Welcome

Today's topics

- Restricted permutation tests
- PERMANOVA
- Distance-based RDA
- Diagnostics

Permutation tests

Permutation tests in vegan

RDA has lots of theory behind it, CCA bit less. However, ecological/environmental data invariably violate what little theory we have

Instead we use permutation tests to assess the *importance* of fitted models — the data are shuffled in some way and the model refitted to derive a Null distribution under some hypothesis of *no effect*

Permutation tests in vegan

What *is* shuffled and *how* is of **paramount** importance for the test to be valid

- No conditioning (partial) variables then rows of the species data are permuted
- With conditioning variables, two options are available, both of which *permute residuals* from model fits
    - The *full model* uses residuals from model $Y = X + Z + \varepsilon$
    - The *reduced model* uses residuals from model $Y = Z + \varepsilon$
- In **vegan** which is used can be set via argument `model` with `"direct"`, `"full"`, and `"reduced"` respectively

Permutation tests in vegan

A test statistic is required, computed for observed model & each permuted model

vegan uses a pseudo F statistic

$$F = \frac{\chi^2_{model}/df_{model}}{\chi^2_{resid}/df_{resid}}$$

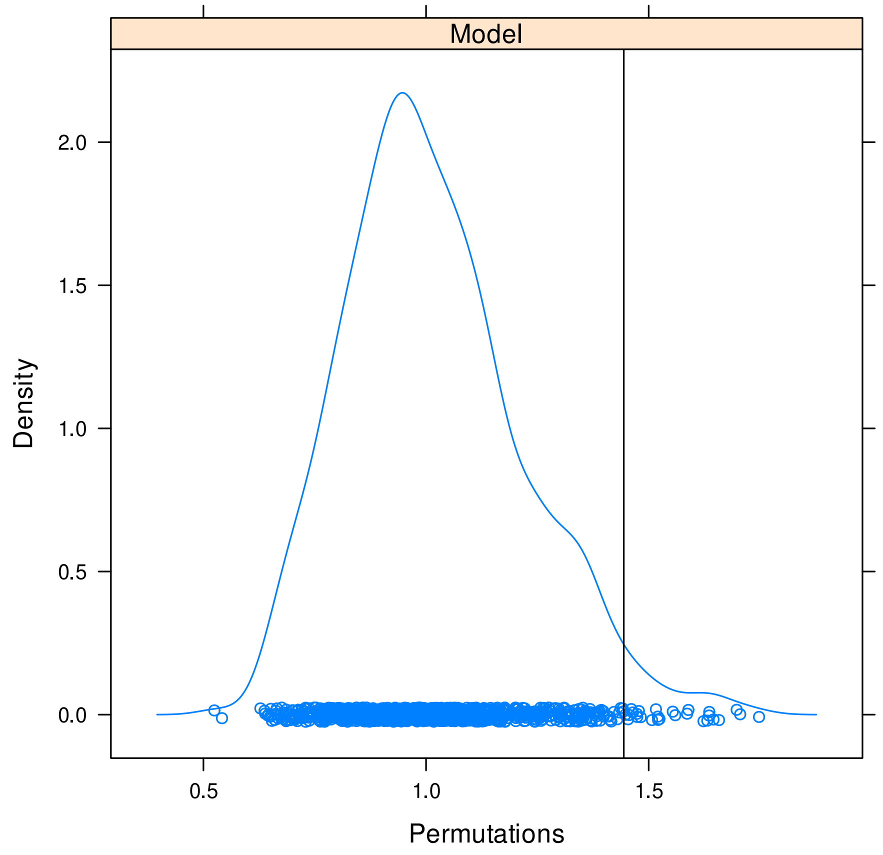
Evaluate whether F is unusually large relative to the null (permutation) distribution of F

Permutation tests in vegan

```r
cca1 <- cca(varespec ~ ., data = varechem)
pstat <- permustats(anova(cca1))
summary(pstat)
```

```
##
##       statistic    SES   mean lower median  upper Pr(perm)
## Model    1.4441 2.2865 1.0158       1.0011 1.3634    0.021 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Interval (Upper - Lower) = 0.95)
```

```r
densityplot(pstat)
```



Permutation tests in vegan: `anova()`

- The main user function is the `anova()` method
- It is an interface to the lower-level function `permutest.cca()`
- At its most simplest, the `anova()` method tests whether the **model** as a whole is significant

Permutation tests in vegan: `anova()`

$$F = \frac{1.4415/14}{0.6417/9} = 1.4441$$

```r
set.seed(42)
(perm <- anova(cca1))
```

```
## Permutation test for cca under reduced model
## Permutation: free
## Number of permutations: 999
##
## Model: cca(formula = varespec ~ N + P + K + Ca + Mg + S + Al + Fe + Mn + Zn + Mo + Baresoil +
Humdepth + pH, data = varechem)
##          Df ChiSquare      F Pr(>F)
## Model    14   1.44148 1.4441  0.029 *
## Residual  9   0.64171
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Restricted permutation tests

Restricted permutation tests

What *is* shuffled and *how* is of **paramount** importance for a valid test

Complete randomisation assumes a null hypothesis where all observations are *independent*

- Temporal or spatial correlation
- Clustering, repeated measures
- Nested sampling designs (Split-plots designs)
- Blocks
- …

Permutation *must* give null distribution of the test statistic whilst preserving the *dependence* between observations

Trick is to shuffle the data whilst preserving that dependence

Restricted permutations

Canoco has had restricted permutations for a *long* time. *vegan* has only recently caught up & we're not (quite) there yet

vegan used to only know how to completely randomise data or completely randomise within blocks (via `strata` in *vegan*)

The **permute** package grew out of initial code in the *vegan* repository to generate the sorts of restricted permutations available in Canoco

We have now fully integrated **permute** into *vegan*…

vegan depends on *permute* so it will already be installed & loaded when using *vegan*

Restricted permutations with permute

permute follows *Canoco* closely — at the (friendly!) chiding of Cajo ter Braak when it didn't do what he wanted!

Samples can be thought of as belonging to three levels of a hierarchy

- the *sample* level; how are individual samples permuted
- the *plot* level; how are samples grouped at an intermediate level
- the *block* level; how are samples grouped at the outermost level

Blocks define groups of plots, each of which can contain groups of samples

Restricted permutations with permute

Blocks are *never* permuted; if defined, only plots or samples *within* the blocks get shuffled & samples are **never** swapped between blocks

Plots or samples within plots, or both can be permuted following one of four simple permutation types

1. Free permutation (randomisation)
2. Time series or linear transect, equal spacing
3. Spatial grid designs, equal regular spacing
4. Permutation of plots (groups of samples)
5. Fixed (no permutation)

Multiple plots per block, multiple samples per plot; plots could be arranged in a spatial grid & samples within plots form time series

Blocks

Blocks are a random factor that does not interact with factors that vary within blocks

Blocks form groups of samples that are never permuted between blocks, only within blocks

Using blocks you can achieve what the `strata` argument used to in **vegan**; needs to be a factor variable

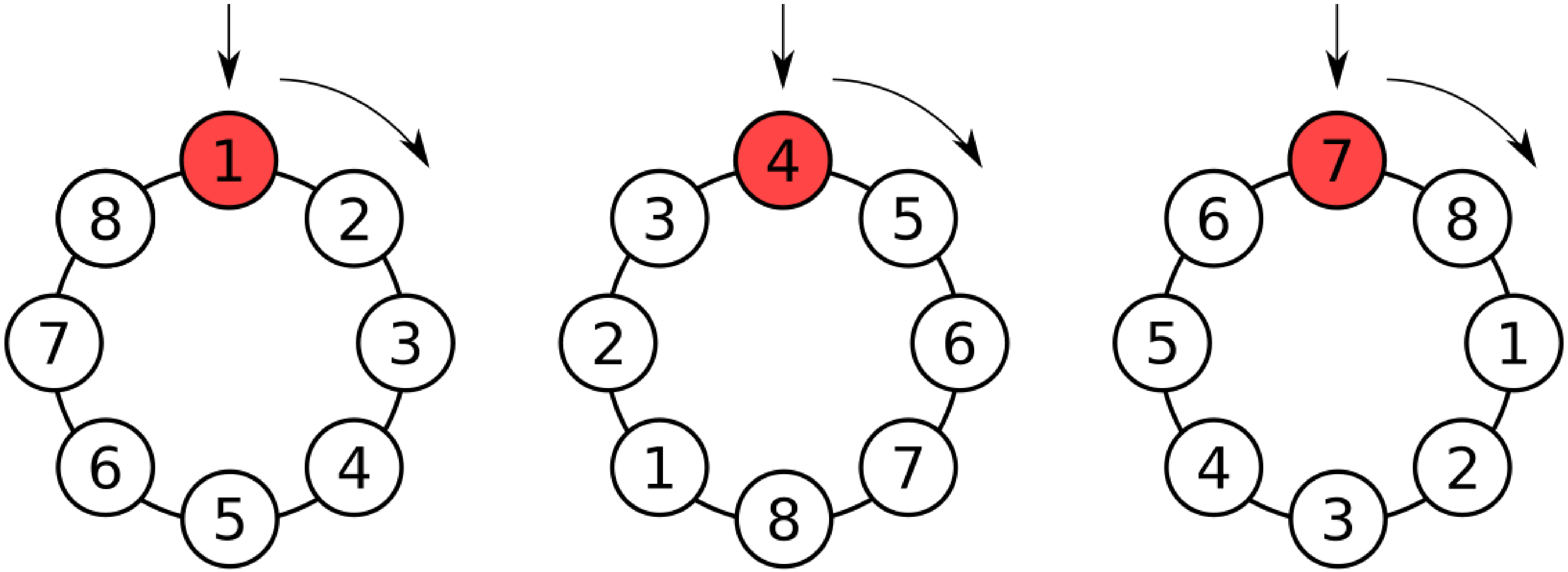
The variation *between* blocks should be excluded from the test; **permute** doesn't do this for you!

Use `+ Condition(blocks)` in the model formula where `blocks` is a factor containing the block membership for each observation

Time series & linear transects

Can link *randomly* starting point of one series to any time point of another series if series are stationary under H_0 that series are unrelated

Achieve this via cyclic shift permutations — wrap series into a circle



Time series & linear transects

Works OK if there are no trends or cyclic pattern — autocorrelation structure only broken at the end points *if* series are stationary

Can detrend to make series stationary but not if you want to test significance of a trend

```r
shuffle(10, control = how(within = Within(type = "series")))
```

```
##  [1]  2  3  4  5  6  7  8  9 10  1
```

Spatial grids

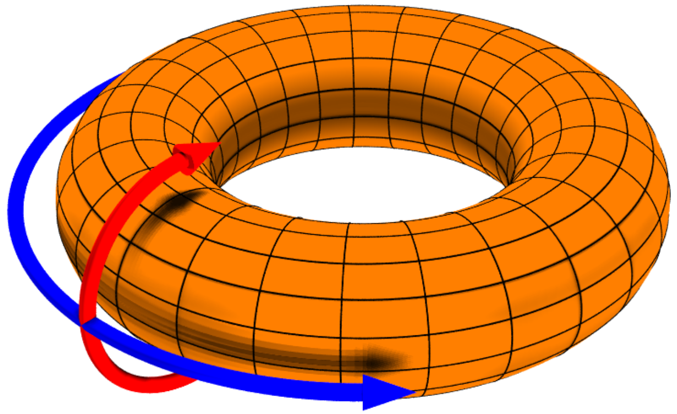
The trick of cyclic shifts can be extended to two dimensions for a regular spatial grid arrangement of points

Now shifts are *toroidal* as we join the end point in the *x* direction together and in the *y* direction together

Source: Dave Burke, Wikimedia CC BY

```r
set.seed(4)
h <- how(within = Within(type = "grid",
                         ncol = 3, nrow = 3))
perm <- shuffle(9, control = h)
matrix(perm, ncol = 3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```



Whole-plots & split-plots I

Split-plot designs are hierarchical with two levels of units

1. **whole-plots** , which contain
2. **split-plots** (the samples)

Permute one or both, but whole-plots must be of equal size

Essentially allows more than one error stratum to be analyzed

Test effect of constraints that vary *between* whole plots by permuting the whole-plots whilst retaining order of split-splots (samples) within the whole-plots

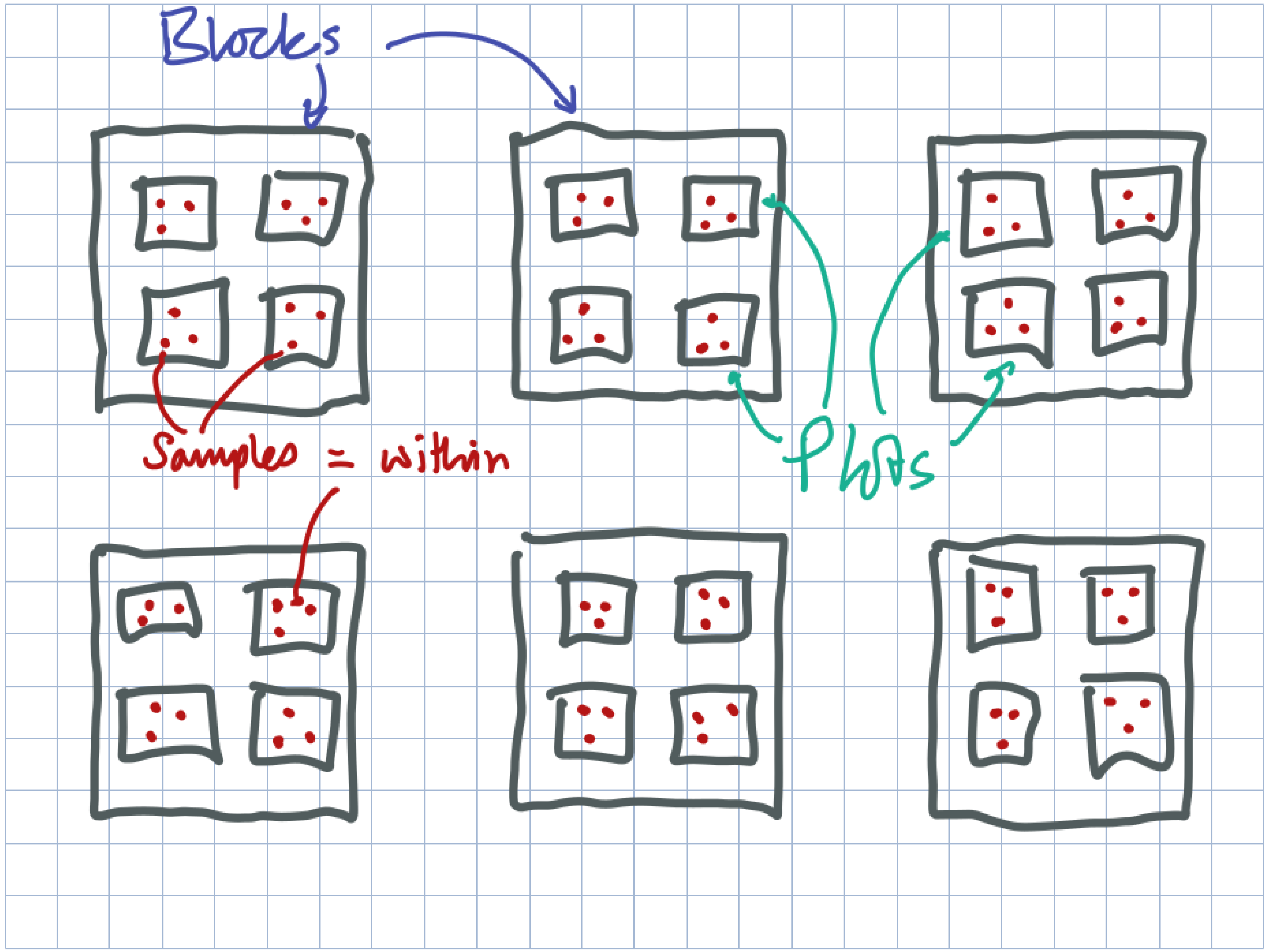
Test effect of constraints that vary *within* whole-plots by permuting the split-plots within whole-plots without permuting the whole-plots

Whole-plots & split-plots II

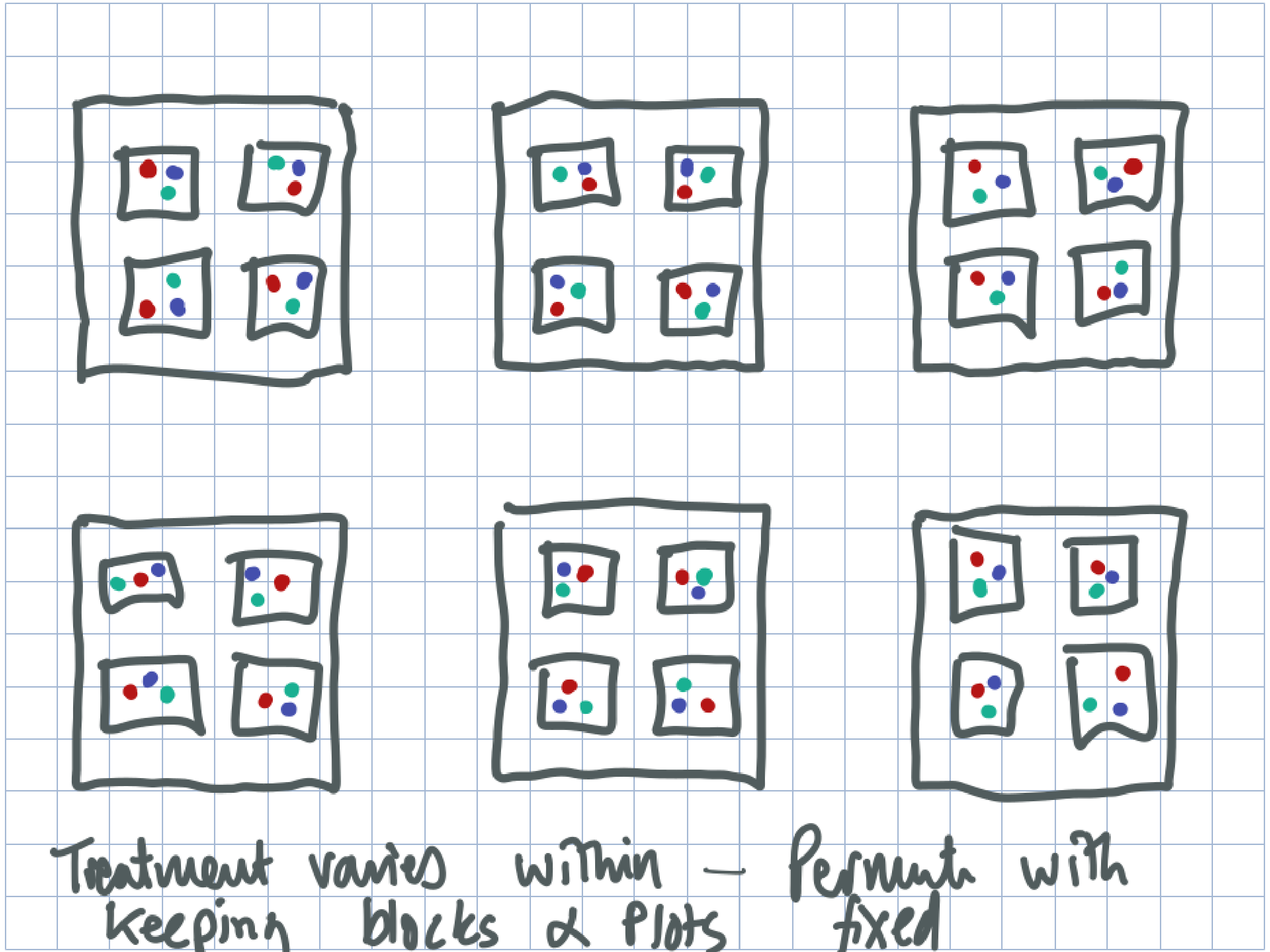
Whole-plots or split-plots, or both, can be time series, linear transects or rectangular grids in which case the appropriate restricted permutation is used

If the split-plots are parallel time series & `time` is an autocorrelated error component affecting all series then the same cyclic shift can be applied to each time series (within each whole-plot) (`constant = TRUE`)

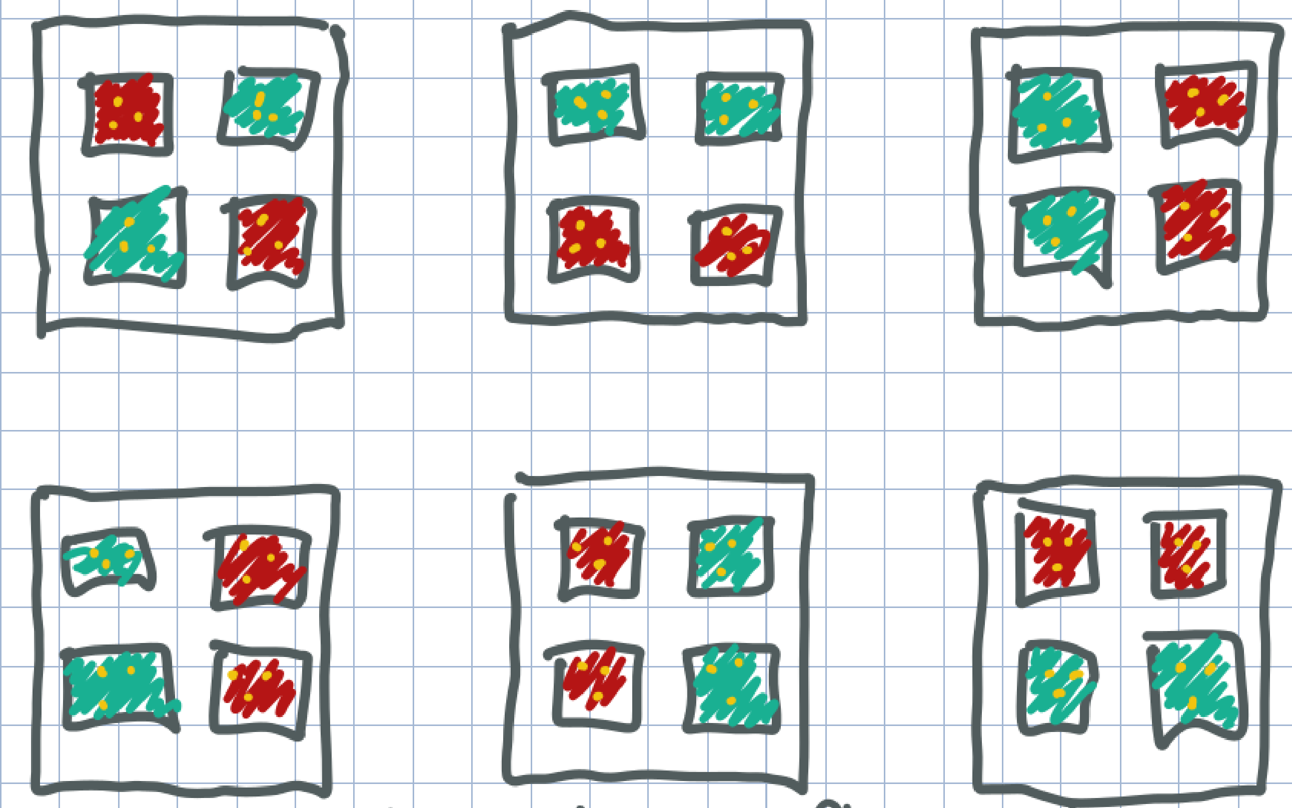
Split plot designs



Split plot designs


Treatment varies within — Permute with keeping blocks & plots fixed

Split plot designs

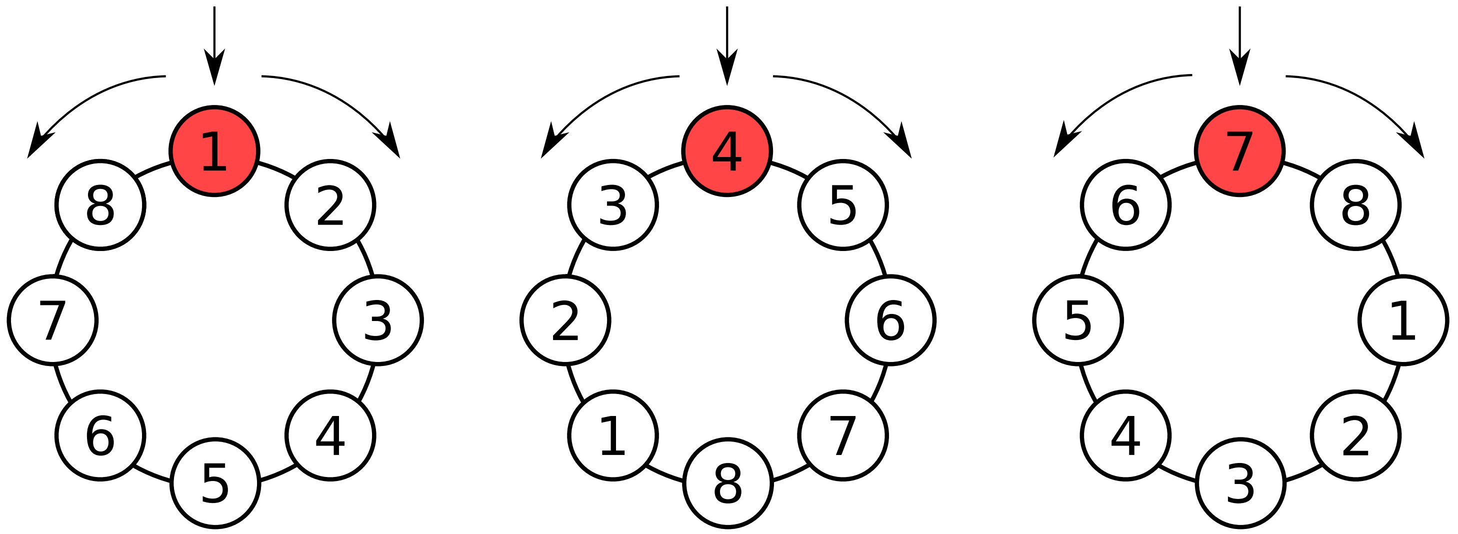


Treatment varies at the Plot level so we Permute whole Plots keeping within fixed

Mirrored permutations

Mirroring in restricted permutations allows for isotropy in dependencies by reflecting the ordering of samples in time or spatial dimensions

For a linear transect, technically the autocorrelation at lag *h* is equal to that at lag -*h* (also in a trend-free time series)



Mirrored permutations

Hence the series `(1, 2, 3, 4)` and `(4, 3, 2, 1)` are equivalent fom this point of view & we can draw permutations from either version

Similar argument can be made for spatial grids

Using `mirror = TRUE` then can double (time series, linear transects) or quadruple (spatial grids) the size of the set of permutations

Sets of permutations — no free lunch

Restricted severely reduce the size of the set of permutations

As the minimum *p* value obtainable is $1/np$ where np is number of allowed permutations (including the observed) this can impact the ability to detect signal/pattern

If we don't want mirroring

- in a time series of 20 samples the minimum *p* is 1/20 (0.05)
- in a time series of 100 samples the minimum *p* is 1/100 (0.01)
- in a data set with 10 time series each of 20 observations (200 total), if we assume an autocorrelated error component over all series (`constant = TRUE`) then there are only 20 permutations of the data and minimum *p* is 0.05

Sets of permutations — no free lunch

When the set of permutations is small it is better to switch to an exact test & evaluate all permutations in the set rather than randomly sample from the set

Use `complete = TRUE` in the call to `how()` — perhaps also increase `maxperm`

Designing permutation schemes

In **permute**, we set up a permutation scheme with `how()`

We sample from the permutation scheme with

- `shuffle()`, which gives a single draw from scheme, or
- `shuffleSet()`, which returns a set of `n` draws from the scheme

`allPerms()` can generated the entire set of permutations — **note** this was designed for small sets of permutations & is slow if you request it for a scheme with many thousands of permutations!

Designing permutation schemes

`how()` has three main arguments

1. `within` — takes input from helper `Within()`
2. `plots` — takes input from helper `Plots()`
3. `blocks` — takes a factor variable as input

```
plt ← gl(3, 10)
h ← how(within = Within(type = "series"), plots = Plots(strata = plt))
```

Designing permutation schemes

Helper functions make it easy to change one or a few aspects of permutation scheme, rest left at defaults

```
args(Within)
```

```
## function (type = c("free", "series", "grid", "none"), constant = FALSE,
##     mirror = FALSE, ncol = NULL, nrow = NULL)
## NULL
```

```
args(Plots)
```

```
## function (strata = NULL, type = c("none", "free", "series", "grid"),
##     mirror = FALSE, ncol = NULL, nrow = NULL)
## NULL
```

Designing permutation schemes

`how()` has additional arguments, many of which control the heuristics that kick in to stop you shooting yourself in the foot and demanding 9999 permutations when there are only 10

- `complete` should we enumerate the entire set of permutations?
- `minperm` lower bound on the size of the set of permutations at & below which we turn on complete enumeration

```
args(how)
```

```
## function (within = Within(), plots = Plots(), blocks = NULL,
##     nperm = 199, complete = FALSE, maxperm = 9999, minperm = 5040,
##     all.perms = NULL, make = TRUE, observed = FALSE)
## NULL
```

Time series example I

Time series within 3 plots, 10 observation each

```r
plt <- gl(3, 10)
h <- how(within = Within(type = "series"),
         plots = Plots(strata = plt))
set.seed(4)
p <- shuffle(30, control = h)
do.call("rbind", split(p, plt)) ## look at perms in context
```

```
##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## 1    9   10    1    2    3    4    5    6    7     8
## 2   14   15   16   17   18   19   20   11   12    13
## 3   24   25   26   27   28   29   30   21   22    23
```

Time series example II

Time series within 3 plots, 10 observation each, same permutation within each

```
plt ← gl(3, 10)
h ← how(within = Within(type = "series", constant = TRUE),
         plots = Plots(strata = plt))
set.seed(4)
p ← shuffle(30, control = h)
do.call("rbind", split(p, plt)) ## look at perms in context
```

```
##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## 1    9   10    1    2    3    4    5    6    7     8
## 2   19   20   11   12   13   14   15   16   17    18
## 3   29   30   21   22   23   24   25   26   27    28
```

Ohraz Case Study

Restricted permutations | Ohraz

Now we've seen how to drive **permute**, we can use the same `how()` commands to set up permutation designs within **vegan** functions

Analyse the Ohraz data Case study 5 of Leps & Smilauer

Repeated observations of composition from an experiment

- Factorial design (3 replicates)
- Treatments: fertilisation, mowing, *Molinia* removal

Test 1 of the hypotheses

> There are *no* directional changes in species composition in time that are common to all treatments or specific treatments

Restricted permutations | Ohraz

Analyse the Ohraz data Case study 5 of Leps & Smilauer

```r
## load vegan
library("vegan")

## load the data
spp <- read.csv("data/ohraz-spp.csv", header = TRUE, row.names = 1)
env <- read.csv("data/ohraz-env.csv", header = TRUE, row.names = 1)
molinia <- spp[, 1]
spp <- spp[, -1]

## Year as numeric
env <- transform(env, year = as.numeric(as.character(year)))
```

Restricted permutations | Ohraz

```
c1 <- rda(spp ~ year + year:mowing + year:fertilizer + year:removal + Condition(plotid), data =
env)
(h <- how(within = Within(type = "free"), plots = Plots(strata = env$plotid, type = "none")))
```

```
##
## Permutation Design:
##
## Blocks:
##   Defined by: none
##
## Plots:
##   Plots: env$plotid
##   Permutation type: none
##   Mirrored?: No
##
## Within Plots:
##   Permutation type: free
##
## Permutation details:
##   Number of permutations: 199
##   Max. number of permutations allowed: 9999
##   Evaluate all permutations?: No.  Activation limit: 5040
```

Restricted permutations | Ohraz

```r
set.seed(42)
anova(c1, permutations = h, model = "reduced")
```

```
## Permutation test for rda under reduced model
## Plots: env$plotid, plot permutation: free
## Permutation: none
## Number of permutations: 199
##
## Model: rda(formula = spp ~ year + year:mowing + year:fertilizer + year:removal +
Condition(plotid), data = env)
##          Df Variance      F Pr(>F)
## Model     4   158.85 6.4247  0.005 **
## Residual 90   556.30
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Restricted permutations | Ohraz

```r
set.seed(24)
anova(c1, permutations = h, model = "reduced", by = "axis")
```

```
## Permutation test for rda under reduced model
## Forward tests for axes
## Plots: env$plotid, plot permutation: free
## Permutation: none
## Number of permutations: 199
##
## Model: rda(formula = spp ~ year + year:mowing + year:fertilizer + year:removal +
Condition(plotid), data = env)
##          Df Variance       F Pr(>F)
## RDA1      1    89.12 14.4173  0.005 **
## RDA2      1    34.28  5.5458  0.005 **
## RDA3      1    26.52  4.2900  0.025 *
## RDA4      1     8.94  1.4458  0.485
## Residual 90   556.30
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Hierarchical analysis of crayfish

Hierarchical analysis of crayfish

Variation in communities may exist at various scales, sometimes hierarchically

A first step in understanding this variation is to test for its exisistence

In this example from Leps & Smilauer (2014) uses crayfish data from Spring River, Arkansas/Missouri, USA, collected by Dr. Camille Flinders.

567 records of 5 species, each sub-divided into *Large* & *Small* individuals

Hierarchical analysis of crayfish

```r
## load data
crayfish <- head(read.csv("data/crayfish-spp.csv")[, -1], -1)
design <- read.csv("data/crayfish-design.csv", skip = 1)[, -1]

## fixup the names
names(crayfish) <- gsub("\\.", "", names(crayfish))
names(design) <- c("Watershed", "Stream", "Reach", "Run",
                   "Stream.Nested", "ReachNested", "Run.Nested")
```

Crayfish — Unconstrained

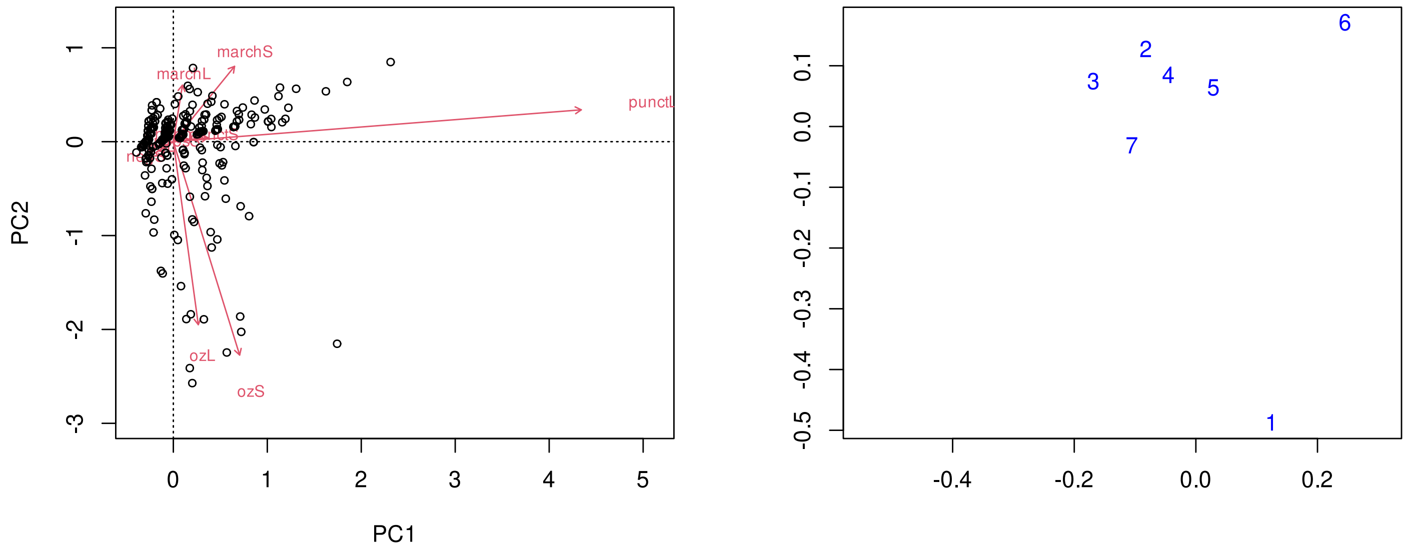
A number of samples have 0 crayfish, which excludes unimodal methods

```
m.pca <- rda(crayfish)
summary(eigenvals(m.pca))
```

```
## Importance of components:
##                          PC1    PC2    PC3    PC4     PC5     PC6     PC7
## Eigenvalue            3.5728 1.8007 1.1974 0.9012 0.79337 0.38886 0.28132
## Proportion Explained  0.3818 0.1924 0.1280 0.0963 0.08478 0.04155 0.03006
## Cumulative Proportion 0.3818 0.5742 0.7022 0.7985 0.88325 0.92480 0.95486
##                           PC8     PC9      PC10
## Eigenvalue            0.21225 0.20528 0.0048809
## Proportion Explained  0.02268 0.02194 0.0005216
## Cumulative Proportion 0.97754 0.99948 1.0000000
```

Crayfish — Unconstrained

```r
layout(matrix(1:2, ncol = 2))
biplot(m.pca, type = c("text", "points"), scaling = "species")
set.seed(23)
ev.pca <- envfit(m.pca ~ Watershed, data = design, scaling = "species")
plot(ev.pca, labels = levels(design$Watershed), add = FALSE)
layout(1)
```



Crayfish — Watershed scale

```r
m.ws <- rda(crayfish ~ Watershed, data = design)
m.ws
```

```
## Call: rda(formula = crayfish ~ Watershed, data = design)
##
##                Inertia Proportion Rank
## Total           9.3580     1.0000
## Constrained     1.7669     0.1888    6
## Unconstrained   7.5911     0.8112   10
## Inertia is variance
##
## Eigenvalues for constrained axes:
##   RDA1   RDA2   RDA3   RDA4   RDA5   RDA6
## 0.7011 0.5540 0.3660 0.1064 0.0381 0.0013
##
## Eigenvalues for unconstrained axes:
##    PC1    PC2    PC3    PC4    PC5    PC6    PC7    PC8    PC9   PC10
## 3.0957 1.2109 0.9717 0.7219 0.5333 0.3838 0.2772 0.2040 0.1879 0.0048
```

Crayfish — Watershed scale

```r
summary(eigenvals(m.ws, constrained = TRUE))
```

```
## Importance of components:
##                          RDA1   RDA2   RDA3   RDA4    RDA5      RDA6
## Eigenvalue             0.7011 0.5540 0.3660 0.1064 0.03814 0.0012791
## Proportion Explained  0.3968 0.3135 0.2072 0.0602 0.02159 0.0007239
## Cumulative Proportion 0.3968 0.7103 0.9175 0.9777 0.99928 1.0000000
```

Crayfish — Watershed scale

```r
set.seed(1)
ctrl <- how(nperm = 499, within = Within(type = "none"),
            plots = with(design, Plots(strata = Stream, type = "free")))
(sig.ws <- anova(m.ws, permutations = ctrl))
```

```
## Permutation test for rda under reduced model
## Plots: Stream, plot permutation: free
## Permutation: none
## Number of permutations: 499
##
## Model: rda(formula = crayfish ~ Watershed, data = design)
##           Df Variance      F Pr(>F)
## Model      6   1.7669 21.724  0.002 **
## Residual 560   7.5911
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Crayfish — Stream scale

```r
m.str <- rda(crayfish ~ Stream + Condition(Watershed), data = design)
m.str
```

```
## Call: rda(formula = crayfish ~ Stream + Condition(Watershed), data =
## design)
##
##                Inertia Proportion Rank
## Total           9.3580     1.0000
## Conditional     1.7669     0.1888    6
## Constrained     1.1478     0.1227   10
## Unconstrained   6.4433     0.6885   10
## Inertia is variance
## Some constraints or conditions were aliased because they were redundant
##
## Eigenvalues for constrained axes:
##   RDA1   RDA2   RDA3   RDA4   RDA5   RDA6   RDA7   RDA8   RDA9  RDA10
## 0.4928 0.2990 0.2058 0.0782 0.0372 0.0224 0.0063 0.0030 0.0029 0.0002
##
## Eigenvalues for unconstrained axes:
##    PC1    PC2    PC3    PC4    PC5    PC6    PC7    PC8    PC9   PC10
## 2.7853 0.8528 0.7737 0.6317 0.5144 0.2808 0.2517 0.1923 0.1559 0.0046
```

Crayfish — Stream scale

```r
summary(eigenvals(m.str, constrained = TRUE))
```

```
## Importance of components:
##                          RDA1   RDA2   RDA3    RDA4    RDA5    RDA6     RDA7
## Eigenvalue             0.4928 0.2990 0.2058 0.07824 0.03719 0.02235 0.006326
## Proportion Explained   0.4293 0.2605 0.1793 0.06816 0.03240 0.01947 0.005511
## Cumulative Proportion  0.4293 0.6898 0.8691 0.93731 0.96971 0.98918 0.994694
##                            RDA8     RDA9     RDA10
## Eigenvalue             0.003042 0.002894 0.0001546
## Proportion Explained   0.002651 0.002521 0.0001347
## Cumulative Proportion  0.997344 0.999865 1.0000000
```

Crayfish — Stream scale

```
set.seed(1)
ctrl ← how(nperm = 499, within = Within(type = "none"),
           plots = with(design, Plots(strata = Reach, type = "free")),
           blocks = with(design, Watershed))
(sig.str ← anova(m.str, permutations = ctrl))
```

```
## Permutation test for rda under reduced model
## Blocks:  with(design, Watershed)
## Plots: Reach, plot permutation: free
## Permutation: none
## Number of permutations: 499
##
## Model: rda(formula = crayfish ~ Stream + Condition(Watershed), data = design)
##           Df Variance      F Pr(>F)
## Model     14   1.1478 6.9477  0.004 **
## Residual 546   6.4433
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Crayfish — Reach scale

```
(m.re ← rda(crayfish ~ Reach + Condition(Stream), data = design))
```

```
## Call: rda(formula = crayfish ~ Reach + Condition(Stream), data =
## design)
##
##                Inertia Proportion Rank
## Total           9.3580     1.0000
## Conditional     2.9148     0.3115   20
## Constrained     1.4829     0.1585   10
## Unconstrained   4.9603     0.5301   10
## Inertia is variance
## Some constraints or conditions were aliased because they were redundant
##
## Eigenvalues for constrained axes:
##   RDA1   RDA2   RDA3   RDA4   RDA5   RDA6   RDA7   RDA8   RDA9  RDA10
## 0.6292 0.2706 0.2146 0.1414 0.1123 0.0467 0.0344 0.0270 0.0064 0.0003
##
## Eigenvalues for unconstrained axes:
##    PC1    PC2    PC3    PC4    PC5    PC6    PC7    PC8    PC9   PC10
## 2.1635 0.6080 0.5605 0.5166 0.3749 0.2212 0.2052 0.1588 0.1477 0.0040
```

Crayfish — Reach scale

```r
set.seed(1)
ctrl <- how(nperm = 499, within = Within(type = "none"),
            plots = with(design, Plots(strata = Run, type = "free")),
            blocks = with(design, Stream))
(sig.re <- anova(m.re, permutations = ctrl))
```

```
## Permutation test for rda under reduced model
## Blocks:  with(design, Stream)
## Plots: Run, plot permutation: free
## Permutation: none
## Number of permutations: 499
##
## Model: rda(formula = crayfish ~ Reach + Condition(Stream), data = design)
##           Df Variance      F Pr(>F)
## Model     42   1.4829 3.5875  0.002 **
## Residual 504   4.9603
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Crayfish — Run scale

```
(m.run ← rda(crayfish ~ Run + Condition(Reach), data = design))
```

```
## Call: rda(formula = crayfish ~ Run + Condition(Reach), data = design)
##
##               Inertia Proportion Rank
## Total          9.3580     1.0000
## Conditional    4.3977     0.4699   62
## Constrained    1.8225     0.1948   10
## Unconstrained  3.1378     0.3353   10
## Inertia is variance
## Some constraints or conditions were aliased because they were redundant
##
## Eigenvalues for constrained axes:
##   RDA1   RDA2   RDA3   RDA4   RDA5   RDA6   RDA7   RDA8   RDA9  RDA10
## 0.8541 0.3141 0.1679 0.1393 0.1328 0.0835 0.0474 0.0429 0.0390 0.0016
##
## Eigenvalues for unconstrained axes:
##    PC1    PC2    PC3    PC4    PC5    PC6    PC7    PC8    PC9   PC10
## 1.3137 0.4165 0.3832 0.2759 0.2378 0.1725 0.1215 0.1130 0.1016 0.0021
```

Crayfish — Run scale

```r
set.seed(1)
ctrl <- how(nperm = 499, within = Within(type = "free"),
            blocks = with(design, Reach))
(sig.run <- anova(m.run, permutations = ctrl))
```

```
## Permutation test for rda under reduced model
## Blocks:  with(design, Reach)
## Permutation: free
## Number of permutations: 499
##
## Model: rda(formula = crayfish ~ Run + Condition(Reach), data = design)
##           Df Variance      F Pr(>F)
## Model    126   1.8225 1.7425  0.002 **
## Residual 378   3.1378
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```