An introduction to *adegenet* 2.0.0

Thibaut Jombart *

Imperial College London

MRC Centre for Outbreak Analysis and Modelling

June 18, 2015

Abstract

This vignette provides an introductory tutorial to the *adegenet* package [1] for the R software [2]. This package implements tools to handle, analyse and simulate genetic data. Originally developped for multiallelic, codominant markers such as microsatellites, *adegenet* now also handles dominant markers, allows for any ploidy in the data, handles SNPs and sequence data, and implements a memory-efficient storage for genome-wide SNP data. This tutorial provides an overview of *adegenet*'s basic functionalities. Since *adegenet* 1.4-0, this tutorial is no longer distributed as a package vignette.

*tjombart@imperial.ac.uk

Contents

1 Introduction

This tutorial introduces some basic functionalities of the *adegenet* package for R [2]. The purpose of this package is to provide tools for handling, analysing and simulating genetic data, with an emphasis on multivariate approaches and exploratory methods. Standard multivariate analyses are implemented in the *ade4* package [3], of which *adegenet* was originally an extension. However, the package has since grown methods of its own such as the Discriminant Analysis of Principal Components (DAPC, [4]), the spatial Principal Components Analysis (sPCA, [5]), or the *SeqTrack* algorithm [6]. In this tutorial, we introduce the main data structures, show how to import data into *adegenet*, and cover some basic population genetics and multivariate analysis.

Other tutorials are available via the command `adegenetTutorial`:

- `adegenetTutorial("spca")`: tutorial on the sPCA
- `adegenetTutorial("dapc")`: tutorial on the DAPC
- `adegenetTutorial("genomics")`: tutorial on handling large SNP datasets using `genlight` objects

2 Getting started

2.1 Installing the package

Before going further, we shall make sure that *adegenet* is well installed on the computer. The current version of the package is 2.0.0. Make sure you have a recent version of R ($\geq$ 3.0.3) by typing:

```
R.version.string

## [1] "R Under development (unstable) (2015-04-13 r68171)"
```

Then, install *adegenet* with dependencies using:

```
install.packages("adegenet", dep=TRUE)
```

We can now load the package alongside other useful packages:

```
library("ape")
library("pegas")

## Loading required package:  adegenet
## Loading required package:  ade4
##    ============================
##    adegenet 2.0.0 is loaded
##    ============================
##
##  - to start:  type '?adegenet'
##  - to browse the adegenet website:  type 'adegenetWeb()'
##  - to post questions/comments:  adegenet-forum@lists.r-forge.r-project.org
##  - to report bugs, request features, contribute:  http://goo.gl/dZuu5X
##
##
##
## Attaching package:  'pegas'
##
## The following object is masked from 'package:ade4':
##
##    amova
##
## The following object is masked from 'package:ape':
##
##    mst

library("seqinr")
```

```
##
## Attaching package:  'seqinr'
##
## The following objects are masked from 'package:ape':
##
##    as.alignment, consensus
```

```
library("ggplot2")
library("adegenet")
```

You can make sure that the right version of the package is installed using:

```
packageDescription("adegenet", fields = "Version")
```

```
## [1] "2.0.0"
```

adegenet version should read 2.0.0.

2.2 Getting help

There are several ways of getting information about R in general, and about *adegenet* in particular. The function `help.search` is used to look for help on a given topic. For instance:

```
help.search("Hardy-Weinberg")
```

replies that there is a function `HWE.test.genind` in the *adegenet* package, and other similar functions in *genetics* and *pegas*. To get help for a given function, use `?foo` where `foo` is the function of interest. For instance (quotes and parentheses can be removed):

```
?spca
```

will open up the manpage of the spatial principal component analysis [5]. At the end of a manpage, an 'example' section often shows how to use a function. This can be copied and pasted to the console, or directly executed from the console using `example`. For further questions concerning R, the function `RSiteSearch` is a powerful tool for making online researches using keywords in R's archives (mailing lists and manpages).

adegenet has a few extra documentation sources. Information can be found from the website (http://adegenet.r-forge.r-project.org/), in the 'documents' section, including several tutorials and a manual which compiles all manpages of the package, and a dedicated mailing list with searchable archives. To open the website from R, use:

```
adegenetWeb()
```

The same can be done for tutorials, using `adegenetTutorial` (see manpage to choose the tutorial to open). You will also find an overview of the main functionalities of the package typing:

```
?adegenet
```

Note that you can also browse help pages as html pages, using:

```
help.start()
```

To go to the *adegenet* page, click 'packages', 'adegenet', and 'adegenet-package'.

Lastly, several mailing lists are available to find different kinds of information on R; to name a few:

- *adegenet forum*: adegenet and multivariate analysis of genetic markers.
  https://lists.r-forge.r-project.org/cgi-bin/mailman/listinfo/adegenet-forum
- *R-help*: general questions about R.
  https://stat.ethz.ch/mailman/listinfo/r-help
- *R-sig-genetics*: genetics in R.
  https://stat.ethz.ch/mailman/listinfo/r-sig-genetics
- *R-sig-phylo*: phylogenetics in R.
  https://stat.ethz.ch/mailman/listinfo/r-sig-phylo

3 Object classes

Two main classes of objects are used for storing genetic marker data, depending on the level at which the genetic information is considered: `genind` is used for individual genotypes, whereas `genpop` is used for alleles numbers counted by populations. Note that the term 'population', here and later, is employed in a broad sense: it simply refers to any grouping of individuals. The specific class `genlight` is used for storing large genome-wide SNPs data. See the *genomics* tutorial for more information on this topic.

3.1 genind objects

These objects store **gen**etic data at an **ind**ividual level, plus various meta-data (e.g. group membership). `genind` objects can be obtained by reading data files from other software, from a `data.frame` of genotypes, by conversion from a table of allele counts, or even from aligned DNA or proteic sequences (see 'importing data'). Here, we introduce this class using the dataset `nancycats`, which is already stored as a `genind` object:

```
data(nancycats)
is.genind(nancycats)

## [1] TRUE

nancycats

##  /// genind object \\\
## @call: genind(tab = truenames(nancycats)$tab, pop = truenames(nancycats)$pop)
##
## @tab:  237 x 108 matrix of genotypes
##
## @ind.names: vector of  237 individual names
## @loc.names: vector of  9 locus names
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  108 columns of @tab
## @all.names: list of  9 components yielding allele names for each locus
## @ploidy:  2 2 2 2 2 2
## @type:  codom
##
## Optional contents:
## @strata: - empty -
## @hierarchy:  - empty -
## @pop:  factor giving the population of each individual
## @pop.names:  character giving the name of each population
##
## @other: a list containing: xy
```

A `genind` object is formal S4 object with several slots, accessed using the '`@`' operator (see `class?genind`). Note that the '`$`' is also implemented for adegenet objects, so that slots can be accessed as if they were components of a list.

`genind` objects possess the following slots:

- `tab`: a matrix of alleles counts (individuals in rows, alleles in columns).
- `loc.names`: a vector of alternative labels for the loci.
- `loc.fac`: a factor indicating which columns in `@tab` correspond to which marker.
- `loc.nall`: the number of alleles in each marker.
- `all.names`: a vector of alternative labels for the alleles.
- `ind.names`: a vector of alternative labels for the individuals.
- `pop`: a factor storing group membership of the individuals.
- `pop.names`: a vector of alternative labels for populations.
- `ploidy`: a vector of integer indicating the ploidy of each individual.
- `type`: a character string indicating whether the marker is codominant (`codom`) or presence/absence (`PA`).
- `other`: a list storing optional information.
- `strata`: an optional data.frame storing hierarchical structure of individuals.
- `hierarchy`: an optional formula defining the hierarchical structure of individuals.
- `call`: the matched call, i.e. command used to create the object.

Slots can be accessed using '`@`' or '`$`', although in some cases it is more convenient to use accessors (i.e. functions which return specific contents of the object) than accessing the slot directly (see section 'Using accessors').

The main slot of a `genind` is the table of allelic counts `@tab`, with individuals in rows and alleles in columns. Note that as of *adegenet* 2.0.0, this table is no storing data as relative frequencies. These can still be obtained using the accessor '`tab`'.

The particular case of presence/absence data is described in a dedicated section (see 'Handling presence/absence data'). For instance:

```
nancycats$tab[10:18,1:10]
```

```
##      fca8.117 fca8.119 fca8.121 fca8.123 fca8.127 fca8.129 fca8.131
## N224        0        0        0        0        0        0        0
## N7          0        0        0        0        0        0        0
## N141        0        0        0        0        0        1        0
## N142        0        0        0        0        0        1        0
## N143        0        0        0        0        0        0        0
## N144        0        0        0        0        0        0        1
## N145        0        0        0        0        0        1        0
## N146        0        0        0        0        0        1        0
## N147        0        0        0        0        0        1        0
##      fca8.133 fca8.135 fca8.137
## N224        0        2        0
## N7          0        0        1
## N141        1        0        0
## N142        1        0        0
## N143        2        0        0
## N144        0        1        0
## N145        0        1        0
## N146        1        0        0
## N147        0        1        0
```

Individual '010' is an homozygote for the allele 09 at locus 1, while '018' is an heterozygote with alleles 06 and 09. As user-defined labels are not always valid (for instance, they can be duplicated), generic labels are used for individuals, markers, alleles and eventually population. The true names are stored in the object (components `$[...].names` where `$[...]` can be `ind`, `loc`, `all` or `pop`). For instance :

```
nancycats$loc.names

##      L1      L2      L3      L4      L5      L6      L7      L8      L9
##  "fca8" "fca23" "fca43" "fca45" "fca77" "fca78" "fca90" "fca96" "fca37"
```

gives the true marker names, and

```
nancycats$all.names[[3]]

##    01    02    03    04    05    06    07    08    09    10
## "133" "135" "137" "139" "141" "143" "145" "147" "149" "157"
```

gives the allele names for marker 3.

The slot 'ploidy' is an integer giving the level of ploidy of the considered organisms (defaults to 2). This parameter is essential, in particular when switching from individual frequencies (`genind` object) to allele counts per populations (`genpop`).
The slot 'type' describes the type of marker used: codominant (`codom`, e.g. microsatellites) or presence/absence (`PA`, e.g. AFLP). By default, adegenet considers that markers are

codominant. Note that actual handling of presence/absence markers has been made available since version 1.2-3. See the dedicated section for more information about presence/absence markers.

Optional content can are also be stored within the object. The slot `@other` is a list that can include any additional information. The optional slot `@pop` (a factor giving a grouping of individuals) is particular in that the behaviour of many functions will check automatically its content and behave accordingly. In fact, each time an argument 'pop' is required by a function, it is first seeked in `@pop`. For instance, using the function `genind2genpop` to convert `nancycats` to a `genpop` object, there is no need to give a 'pop' argument as it exists in the `genind` object:

```
head(pop(nancycats))

## [1] 1 1 1 1 1 1
## Levels: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

catpop <- genind2genpop(nancycats)

##
##  Converting data from a genind to a genpop object...
##
## ...done.

catpop

##  /// genpop object \\\
## @call: genind2genpop(x = nancycats)
##
## @tab:  17 x 108 matrix of alleles counts
##
## @pop.names: vector of  17 population names
## @loc.names: vector of  9 locus names
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  108 columns of @tab
## @all.names: list of  9 components yielding allele names for each locus
## @ploidy:  2
## @type:  codom
##
## @other: a list containing: xy
```

Other additional components can be stored (like here, spatial coordinates of populations in $xy) and processed during the conversion if the argument `process.other` is set to `TRUE`. In this case, numeric vectors with a length corresponding to the number of individuals will we averaged per groups; note that any other function than `mean` can be used by providing any function to the argument `other.action`. Matrices with a number of rows corresponding

to the number of individuals are processed similarly.

Finally, a **genind** object generally contains its matched call, *i.e.* the instruction that created it. When call is available, it can be used to regenerate an object.

```
obj <- read.genetix(system.file("files/nancycats.gtx",package="adegenet"))

##
##  Converting data from GENETIX to a genind object...
##
## ...done.

obj$call

## read.genetix(file = system.file("files/nancycats.gtx", package = "adegenet"))

toto <- eval(obj$call)

##
##  Converting data from GENETIX to a genind object...
##
## ...done.

identical(obj,toto)

## [1] TRUE
```

3.2 genpop objects

We use the previously built **genpop** object:

```
catpop

##  /// genpop object \\\
## @call: genind2genpop(x = nancycats)
##
## @tab:  17 x 108 matrix of alleles counts
##
## @pop.names: vector of  17 population names
## @loc.names: vector of  9 locus names
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  108 columns of @tab
## @all.names: list of  9 components yielding allele names for each locus
## @ploidy:  2
## @type:  codom
```

```
##
## @other: a list containing: xy

is.genpop(catpop)

## [1] TRUE

catpop$tab[1:5,1:10]

##   fca8.117 fca8.119 fca8.121 fca8.123 fca8.127 fca8.129 fca8.131 fca8.133
## 1        0        0        0        0        0        0        0        2
## 2        0        0        0        0        0       10        9        8
## 3        0        0        0        4        0        0        0        0
## 4        0        0        0        3        0        0        0        1
## 5        0        0        0        1        0        0        0        0
##   fca8.135 fca8.137
## 1        9        1
## 2       14        2
## 3        1       10
## 4        7       17
## 5        7       10
```

The matrix $tab contains alleles counts per population (here, cat colonies). These objects are otherwise very similar to `genind` in their structure, and possess generic names, true names, the matched call and an `@other` slot:

```
getClassDef("genpop")

## Class "genpop" [package "adegenet"]
##
## Slots:
##
## Name:           tab    loc.names      loc.fac     loc.nall    all.names
## Class:       matrix    character factorOrNULL     intOrNum   listOrNULL
##
## Name:          call    pop.names       ploidy         type        other
## Class:   callOrNULL    character      integer    character   listOrNULL
##
## Extends: "gen", "popInfo"
```

3.3 Using accessors

One advantage of formal (S4) classes is that they allow for interacting simply with possibly complex objects. This is made possible by using accessors, i.e. functions that extract

information from an object, rather than accessing the slots directly. Another advantage of this approach is that as long as accessors remain identical on the user's side, the internal structure of an object may change from one release to another without generating errors in old scripts. Although `genind` and `genpop` objects are fairly simple, we recommend using accessors whenever possible to access their content.

Available accessors are:

- `nInd`: returns the number of individuals in the object; only for `genind`.
- `nLoc`: returns the number of loci (SNPs).
- `indNames`†: returns/sets labels for individuals; only for `genind`.
- `locNames`†: returns/sets labels for loci (SNPs).
- `alleles`†: returns/sets alleles.
- `ploidy`†: returns/sets ploidy of the individuals.
- `pop`†: returns/sets a factor grouping individuals; only for `genind`.
- `other`†: returns/sets misc information stored as a list.

where † indicates that a replacement method is available using `<-`; for instance:

```
head(indNames(nancycats),10)

##    001    002    003    004    005    006    007    008    009    010
## "N215" "N216" "N217" "N218" "N219" "N220" "N221" "N222" "N223" "N224"

indNames(nancycats) <- paste("cat", 1:nInd(nancycats),sep=".")
head(indNames(nancycats),10)

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

Some accessors such as `locNames` may have specific options; for instance:

```
locNames(nancycats)

##      L1      L2      L3      L4      L5      L6      L7      L8      L9
##  "fca8" "fca23" "fca43" "fca45" "fca77" "fca78" "fca90" "fca96" "fca37"
```

returns the names of the loci, while:

```
temp <- locNames(nancycats, withAlleles=TRUE)
head(temp, 10)

##  [1] "fca8.117" "fca8.119" "fca8.121" "fca8.123" "fca8.127" "fca8.129"
##  [7] "fca8.131" "fca8.133" "fca8.135" "fca8.137"
```

returns the names of the alleles in the form 'loci.allele'.

The slot 'pop' can be retrieved and set using `pop`:

```
obj <- nancycats[sample(1:50,10)]
pop(obj)

##  [1] 3 1 2 4 2 3 4 1 3 4
## Levels: 1 2 3 4

pop(obj) <- rep("newPop",10)
pop(obj)

##  [1] newPop newPop newPop newPop newPop newPop newPop newPop newPop newPop
## Levels: newPop
```

An additional advantage of using accessors is they are most of the time safer to use. For instance, `pop<-` will check the length of the new group membership vector against the data, and complain if there is a mismatch. It also converts the provided replacement to a factor, while the command:

```
obj@pop <- rep("newPop",10)

## Error in checkAtAssignment(structure("genind", package = "adegenet"), :
assignment of an object of class "character" is not valid for @'pop' in an
object of class "genind"; is(value, "factorOrNULL") is not TRUE
```

generates an error (since replacement is not a factor).

4 Importing/exporting data

4.1 Importing data from GENETIX, STRUCTURE, FSTAT, Genepop

Data can be read from the software GENETIX (extension .gtx), STRUCTURE (.str or .stru), FSTAT (.dat) and Genepop (.gen) files, using the corresponding `read` function: `read.genetix`, `read.structure`, `read.fstat`, and `read.genepop`. These functions take as main argument the path (as a string of characters) to an input file, and produce a `genind` object. Alternatively, one can use the function `import2genind` which detects a file format from its extension and uses the appropriate routine. For instance:

```
obj1 <- read.genetix(system.file("files/nancycats.gtx",package="adegenet"))

##
##  Converting data from GENETIX to a genind object...
##
## ...done.

obj2 <- import2genind(system.file("files/nancycats.gtx", package="adegenet"))

##
##  Converting data from GENETIX to a genind object...
##
## ...done.

all.equal(obj1,obj2)

## [1] "Attributes: < Component \"call\": target, current do not match when deparsed >"
```

The only difference between `obj1` and `obj2` is their call (which is normal as they were obtained from different command lines).

4.2 Importing data from other software

Raw genetic markers data are often stored as tables with individuals in row and markers in column, where each entry is a character string coding the alleles possessed at one locus. Such data are easily imported into R as a `data.frame`, using for instance `read.table` for text files or `read.csv` for comma-separated text files. Then, the obtained `data.frame` can be converted into a `genind` object using `df2genind`.

There are only a few pre-requisite the data should meet for this conversion to be possible. The easiest and clearest way of coding data is using a separator between alleles. For instance, "80/78", "80—78", or "80,78" are different ways of coding a genotype at a microsatellite locus with alleles '80' and 78". Note that for haploid data, no separator shall be used. The only contraint when using a separator is that the same separator is used in all

the dataset. There are no contraints as to i) the type of separator used or ii) the ploidy of the data. These parameters can be set in `df2genind` through arguments `sep` and `ploidy`, respectively.

Alternatively, no separator may be used provided a fixed number of characters is used to code each allele. For instance, in a diploid organism, "0101" is an homozygote 1/1 while "1209" is a heterozygote 12/09 in a two-character per allele coding scheme. In a tetraploid system with one character per allele, "1209" will be understood as 1/2/0/9.

Here, we provide an example using randomly generated tetraploid data and no separator.

```
temp <- lapply(1:30, function(i) sample(1:9, 4, replace=TRUE))
temp <- sapply(temp, paste, collapse="")
temp <- matrix(temp, nrow=10, dimnames=list(paste("ind",1:10), paste("loc",1:3)))
temp

##        loc 1  loc 2  loc 3
## ind 1  "6828" "4692" "1974"
## ind 2  "9158" "8828" "6744"
## ind 3  "5152" "3349" "1323"
## ind 4  "3174" "2393" "5959"
## ind 5  "1552" "8266" "9299"
## ind 6  "8382" "5216" "8346"
## ind 7  "9157" "4523" "9913"
## ind 8  "8832" "7499" "8441"
## ind 9  "6566" "6246" "4186"
## ind 10 "6914" "4627" "8448"

obj <- df2genind(temp, ploidy=4, sep="")
obj

##  /// genind object \\\
## @call: df2genind(X = temp, sep = "", ploidy = 4)
##
## @tab:  10 x 27 matrix of genotypes
##
## @ind.names: vector of  10 individual names
## @loc.names: vector of  3 locus names
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  27 columns of @tab
## @all.names: list of  3 components yielding allele names for each locus
## @ploidy:  4 4 4 4 4 4
## @type:  codom
##
## Optional contents:
```

```
## @strata: - empty -
## @hierarchy:  - empty -
## @pop:  - empty -
## @pop.names:  - empty -
##
## @other: - empty -
```

`obj` is a `genind` containing the same information, but recoded as a matrix of allele frequencies (`$tab` slot). We can check that the conversion was exact by converting back the object into a table of character strings (function `genind2df`):

```
genind2df(obj, sep="|")
```

```
##          loc 1   loc 2   loc 3
## ind 1  6|8|8|2 4|6|9|2 1|9|7|4
## ind 2  8|9|1|5 2|8|8|8 7|4|4|6
## ind 3  2|1|5|5 4|9|3|3 1|3|3|2
## ind 4  1|3|7|4 9|2|3|3 9|9|5|5
## ind 5  2|1|5|5 6|6|2|8 9|9|9|2
## ind 6  8|8|2|3 6|2|5|1 4|6|3|8
## ind 7  9|1|5|7 4|2|3|5 1|9|9|3
## ind 8  8|8|2|3 4|9|9|7 1|4|4|8
## ind 9  6|6|6|5 4|6|6|2 1|4|6|8
## ind 10 6|9|1|4 4|6|2|7 4|4|8|8
```

4.3 Handling presence/absence data

adegenet was primarly designed to handle codominant, multiallelic markers like microsatellites. However, dominant markers like AFLP can be used as well. In such a case, only presence/absence of alleles can be deduced accurately from the genotypes. This has several consequences, like the unability to compute allele frequencies. Hence, some functionalities in *adegenet* won't be available for dominant markers.

From version 1.2-3 of *adegenet*, the distinction between both types of markers is made by the slot `@type` of genind or genpop objects, which equals `codom` for codominant markers, and `PA` for presence/absence data. In the latter case, the 'tab' slot of a genind object no longer contains allele frequencies, but only presence/absence of alleles in a genotype. Similarly, the `tab` slot of a genpop object not longer contains counts of alleles in the populations; instead, it contains the number of genotypes in each population possessing at least one copy of the concerned alleles. Moreover, in the case of presence/absence, the slots 'loc.nall', 'loc.fac', and 'all.names' become useless, and are thus all set to NULL.

Objects of type 'PA' are otherwise handled like usual (type 'codom') objects. Operations that are not available for PA type will issue an appropriate error message.

Here is an example using a toy dataset 'AFLP.txt' that can be downloaded from the adegenet website, section 'Documentation':

```
dat <- read.table(system.file("files/AFLP.txt",package="adegenet"), header=TRUE)
dat

##      loc1 loc2 loc3 loc4
## indA    1    0    1    1
## indB    0    1    1    1
## indC    1    1    0    1
## indD    0   NA    1   NA
## indE    1    1    0    0
## indF    1    0    1    1
## indG    0    1    1    0
```

The function `df2genind` is used to obtain a genind object:

```
obj <- genind(dat, ploidy=1, type="PA")

## Error in .local(.Object, ...):  (list) object cannot be coerced to type
'integer'

obj

##  /// genind object \\\
## @call: df2genind(X = temp, sep = "", ploidy = 4)
##
## @tab:  10 x 27 matrix of genotypes
##
## @ind.names: vector of  10 individual names
## @loc.names: vector of  3 locus names
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  27 columns of @tab
## @all.names: list of  3 components yielding allele names for each locus
## @ploidy:  4 4 4 4 4 4
## @type:  codom
##
## Optional contents:
## @strata: - empty -
## @hierarchy:  - empty -
## @pop:  - empty -
## @pop.names:  - empty -
##
## @other: - empty -

tab(obj)

##        loc 1.6 loc 1.8 loc 1.2 loc 1.9 loc 1.1 loc 1.5 loc 1.3 loc 1.7
## ind 1        1       2       1       0       0       0       0       0
```

```
## ind 2        0       1       0       1       1       1       0       0
## ind 3        0       0       1       0       1       2       0       0
## ind 4        0       0       0       0       1       0       1       1
## ind 5        0       0       1       0       1       2       0       0
## ind 6        0       2       1       0       0       0       1       0
## ind 7        0       0       0       1       1       1       0       1
## ind 8        0       2       1       0       0       0       1       0
## ind 9        3       0       0       0       0       1       0       0
## ind 10       1       0       0       1       1       0       0       0
##        loc 1.4 loc 2.4 loc 2.6 loc 2.9 loc 2.2 loc 2.8 loc 2.3 loc 2.5
## ind 1        0       1       1       1       1       0       0       0
## ind 2        0       0       0       0       1       3       0       0
## ind 3        0       1       0       1       0       0       2       0
## ind 4        1       0       0       1       1       0       2       0
## ind 5        0       0       2       0       1       1       0       0
## ind 6        0       0       1       0       1       0       0       1
## ind 7        0       1       0       0       1       0       1       1
## ind 8        0       1       0       2       0       0       0       0
## ind 9        0       1       2       0       1       0       0       0
## ind 10       1       1       1       0       1       0       0       0
##        loc 2.1 loc 2.7 loc 3.1 loc 3.9 loc 3.7 loc 3.4 loc 3.6 loc 3.3
## ind 1        0       0       1       1       1       1       0       0
## ind 2        0       0       0       0       1       2       1       0
## ind 3        0       0       1       0       0       0       0       2
## ind 4        0       0       0       2       0       0       0       0
## ind 5        0       0       0       3       0       0       0       0
## ind 6        1       0       0       0       0       1       1       1
## ind 7        0       0       1       2       0       0       0       1
## ind 8        0       1       1       0       0       2       0       0
## ind 9        0       0       1       0       0       1       1       0
## ind 10       0       1       0       0       0       2       0       0
##        loc 3.2 loc 3.5 loc 3.8
## ind 1        0       0       0
## ind 2        0       0       0
## ind 3        1       0       0
## ind 4        0       2       0
## ind 5        1       0       0
## ind 6        0       0       1
## ind 7        0       0       0
## ind 8        0       0       1
## ind 9        0       0       1
## ind 10       0       0       2
```

One can see that for instance, the summary of this object is more simple (no numbers of alleles per locus, no heterozygosity):

```
pop(obj) <- rep(c('a','b'),4:3)

## Error in `pop<-`(`*tmp*`, value = c("a", "a", "a", "a", "b", "b", "b")):
wrong length for population factor

summary(obj)

##
##  # Total number of genotypes:  10
##
##  # Population sample sizes:
##
## 10
##
##  # Number of alleles per locus:
## loc 1 loc 2 loc 3
##     9     9     9
##
##  # Number of alleles per population:
##
## 27
##
##  # Percentage of missing data:
## [1] 0
##
##  # Observed heterozygosity:
## loc 1 loc 2 loc 3
##     1     1     1
##
##  # Expected heterozygosity:
##   loc 1   loc 2   loc 3
## 0.86875 0.86000 0.85500
```

But we can still perform basic manipulation, like converting our object into a genpop:

```
obj2 <- genind2genpop(obj)

##
##  Converting data from a genind to a genpop object...
##
## ...done.

obj2

##  /// genpop object \\\
## @call: genind2genpop(x = obj)
```

```
##
## @tab:  1 x 27 matrix of alleles counts
##
## @pop.names: vector of  1 population names
## @loc.names: vector of  3 locus names
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  27 columns of @tab
## @all.names: list of  3 components yielding allele names for each locus
## @ploidy:  4
## @type:  codom
##
## @other: - empty -
```

```
tab(obj2)
```

```
##   loc 1.6 loc 1.8 loc 1.2 loc 1.9 loc 1.1 loc 1.5 loc 1.3 loc 1.7 loc 1.4
## 1       5       7       5       3       6       7       3       2       2
##   loc 2.4 loc 2.6 loc 2.9 loc 2.2 loc 2.8 loc 2.3 loc 2.5 loc 2.1 loc 2.7
## 1       6       7       5       8       4       5       2       1       2
##   loc 3.1 loc 3.9 loc 3.7 loc 3.4 loc 3.6 loc 3.3 loc 3.2 loc 3.5 loc 3.8
## 1       5       8       2       9       3       4       2       2       5
```

To continue with the toy example, we can proceed to a simple PCA. NAs are first replaced:

```
objNoNa <- na.replace(obj,met=0)
```

```
## Error in eval(expr, envir, enclos):  could not find function "na.replace"
```

```
objNoNa@tab
```

```
## Error in eval(expr, envir, enclos):  object 'objNoNa' not found
```

Now the PCA is performed and plotted:

```
pca1 <- dudi.pca(objNoNa,scannf=FALSE,scale=FALSE)
```

```
## Error in as.data.frame(df):  object 'objNoNa' not found
```

```
temp <- as.integer(pop(objNoNa))
```

```
## Error in pop(objNoNa):  error in evaluating the argument 'x' in selecting
a method for function 'pop':  Error:  object 'objNoNa' not found
```

```
myCol <- transp(c("blue","red"),.7)[temp]
myPch <- c(15,17)[temp]
plot(pca1$li, col=myCol, cex=3, pch=myPch)
```

```
## Error in plot(pca1$li, col = myCol, cex = 3, pch = myPch):  error in
evaluating the argument 'x' in selecting a method for function 'plot':
Error:  object 'pca1' not found

abline(h=0,v=0,col="grey",lty=2)

## Error in int_abline(a = a, b = b, h = h, v = v, untf = untf, ...):
plot.new has not been called yet

s.arrow(pca1$c1, add.plot=TRUE)

## Error in data.frame(dfxy):  object 'pca1' not found

legend("topright", pch=c(15,17), col=transp(c("blue","red"),.7),
       leg=c("Group A","Group B"), pt.cex=2)

## Error in strwidth(legend, units = "user", cex = cex, font = text.font):
plot.new has not been called yet
```

More generally, multivariate analyses from ade4, sPCA (`spca`), DAPC (`dapc`), the global and local tests (`global.rtest`, `local.rtest`), or the Monmonier's algorithm (`monmonier`) will work just fine with presence/absence data. However, it is clear that the usual Euclidean distance (used in PCA and sPCA), as well as many other distances, is not as accurate to measure genetic dissimilarity using presence/absence data as it is when using allele frequencies. The reason for this is that in presence/absence data, a part of the information is simply hidden. For instance, two individuals possessing the same allele will be considered at the same distance, whether they possess one or more copies of the allele. This might be especially problematic in organisms having a high degree of ploidy.

4.4 SNPs data

In *adegenet*, SNP data can be handled in two different ways. For relatively small datasets (up to a few thousand SNPs) SNPs can be handled as usual codominant markers such as microsatellites using `genind` objects. In the case of genome-wide SNP data (from hundreds of thousands to millions of SNPs), `genind` objects are no longer efficient representation of the data. In this case, we use `genlight` objects to store and handle information with maximum efficiency and minimum memory requirements. See the tutorial *genomics* for more information. Below, we introduce only the case of SNPs handled using `genind` objects.

The most convenient way to convert SNPs into a `genind` is using `df2genind`, which is described in the previous section. Let `dat` be an input matrix, as can be read into R using `read.table` or `read.csv`, with genotypes in row and SNP loci in columns.

```
dat <- matrix(sample(c("a","t","g","c"), 15, replace=TRUE),nrow=3)
rownames(dat) <- paste("genot.", 1:3)
colnames(dat) <- 1:5
```

```
dat

##          1   2   3   4   5
## genot. 1 "t" "c" "a" "c" "c"
## genot. 2 "t" "g" "t" "t" "a"
## genot. 3 "a" "g" "c" "c" "t"

obj <- df2genind(dat, ploidy=1)
tab(obj)

##          1.t 1.a 2.c 2.g 3.a 3.t 3.c 4.c 4.t 5.c 5.a 5.t
## genot. 1   1   0   1   0   1   0   0   1   0   1   0   0
## genot. 2   1   0   0   1   0   1   0   0   1   0   1   0
## genot. 3   0   1   0   1   0   0   1   1   0   0   0   1
```

`obj` is a `genind` containing the SNPs information, which can be used for further analysis in adegenet.

4.5 Extracting polymorphism from DNA sequences

This section only covers the cases of relatively small datasets which can be handled efficiently using `genind` objects. For bigger (near full-genomes) datasets, SNPs can be extracted from *fasta* files into a `genlight` object using `fasta2genlight`. See the tutorial on *genomics* for more information.

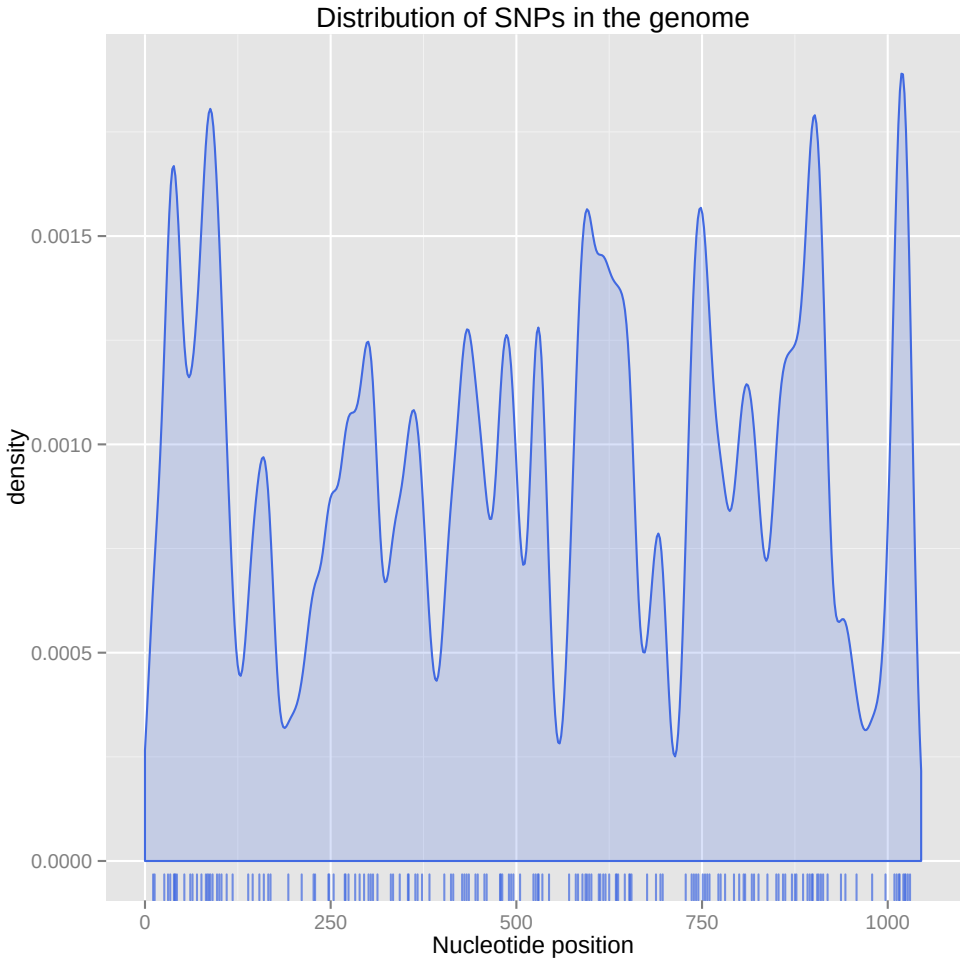
DNA sequences can be read into R using *ape*'s `read.dna` (fasta and clustal formats), or *adegenet*'s `fasta2DNAbin` for a RAM-greedy implementation (fasta alignments only). Other options include reading data directly from GenBank using `read.GenBank`, or from other public databases using the *seqinr* package and transforming the `alignment` object into a `DNAbin` using `as.DNAbin`. Here, we illustrate this approach by re-using the example of `read.GenBank`. A connection to the internet is required, as sequences are read directly from a distant database.

```
library(ape)
ref <- c("U15717", "U15718", "U15719", "U15720",
         "U15721", "U15722", "U15723", "U15724")
myDNA <- read.GenBank(ref)
myDNA

## 8 DNA sequences in binary format stored in a list.
##
## All sequences of same length: 1045
##
## Labels: U15717 U15718 U15719 U15720 U15721 U15722 ...
##
```

```
## Base composition:
##     a     c     g     t
## 0.267 0.351 0.134 0.247

class(myDNA)

## [1] "DNAbin"

myDNA <- as.matrix(myDNA)
```

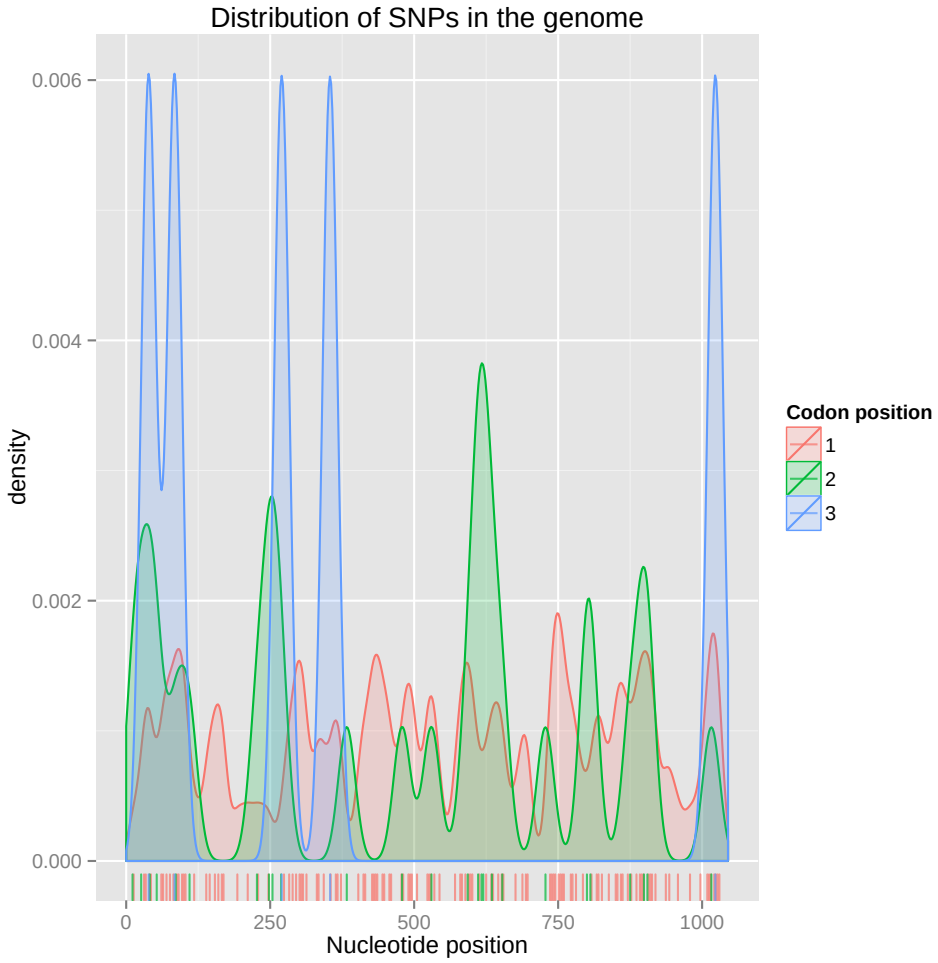
Polymorphism can be characterized using `snpposi.plot` and `snpposi.test`: the first plots SNP density along the alignment, the second tests whether these SNPs are randomly distributed. For instance:

```
snpposi.plot(myDNA,codon=FALSE)
```



By default, the function differentiates nucleotide positions:

```
snpposi.plot(myDNA)
```



In *adegenet*, only polymorphic loci are conserved to form a *genind* object. This conversion is achieved by `DNAbin2genind`. This function allows one to specify a threshold for polymorphism; for instance, one could retain only SNPs for which the second largest allele frequency is greater than 1% (using the `polyThres` argument). This is achieved using:

```
obj <- DNAbin2genind(myDNA, polyThres=0.01)
obj

##  /// genind object \\\
## @call: DNAbin2genind(x = myDNA, polyThres = 0.01)
##
## @tab:  8 x 318 matrix of genotypes
##
## @ind.names: vector of  8 individual names
## @loc.names: vector of  155 locus names
## @loc.nall: number of alleles per locus
```

```
## @loc.fac: locus factor for the  318 columns of @tab
## @all.names: list of  155 components yielding allele names for each locus
## @ploidy:  1 1 1 1 1 1
## @type:  codom
##
## Optional contents:
## @strata: - empty -
## @hierarchy:  - empty -
## @pop:  - empty -
## @pop.names:  - empty -
##
## @other: - empty -
```

Here, out of the 1,045 nucleotides of the sequences, 318 SNPs where extracted and stored as a `genind` object. Positions of the SNPs are stored as names of the loci:

```
head(locNames(obj))

## [1] "11" "13" "26" "31" "34" "39"
```

4.6 Extracting polymorphism from proteic sequences

Alignments of proteic sequences can be exploited in *adegenet* in the same way as DNA sequences (see section above). Alignments are scanned for polymorphic sites, and only those are retained to form a `genind` object. Loci correspond to the position of the residue in the alignment, and alleles correspond to the different amino-acids (AA). Aligned proteic sequences are stored as objects of class `alignment` in the *seqinr* package [9]. See `?as.alignment` for a description of this class. The function extracting polymorphic sites from `alignment` objects is `alignment2genind`.

Its use is fairly simple. It is here illustrated using a small dataset of aligned proteic sequences:

```
library(seqinr)
mase.res   <- read.alignment(file=system.file("sequences/test.mase",
                             package="seqinr"), format = "mase")
mase.res

## $nb
## [1] 6
##
## $nam
## [1] "Langur" "Baboon" "Human"  "Rat"    "Cow"    "Horse"
##
## $seq
```

```
## $seq[[1]]
## [1] "-kifercelartlkklgldgykgvslanwvclakwesgynteatnynpgdestdygifqinsrywcnngkpgavdachis
##
## $seq[[2]]
## [1] "-kifercelartlkrlgldgyrgislanwvclakwesdyntqatnynpgdqstdygifqinshywcndgkpgavnachis
##
## $seq[[3]]
## [1] "-kvfercelartlkrlgmdgyrgislanwmclakwesgyntratnynagdrstdygifqinsrywcndgkpgavnachls
##
## $seq[[4]]
## [1] "-ktyercefartlkrngmsgyygvsladwvclaqhesnyntqarnydpgdqstdygifqinsrywcndgkpraknacgip
##
## $seq[[5]]
## [1] "-kvfercelartlkklgldgykgvslanwlcltkwessyntkatnynpssestdygifqinskwwcndgkpnavdgchvs
##
## $seq[[6]]
## [1] "-kvfskcelahklkaqemdgfggyslanwvcmaeyesnfntrafngknangssdyglfqlnnkwwckdnkrsssnacnim
##
##
## $com
## [1] ";empty description\n" ";\n"                  ";\n"
## [4] ";\n"                  ";\n"                  ";\n"
##
## attr(,"class")
## [1] "alignment"
```

```
x <- alignment2genind(mase.res)
x
```

```
##  /// genind object \\\
## @call: alignment2genind(x = mase.res)
##
## @tab:  6 x 212 matrix of genotypes
##
## @ind.names: vector of  6 individual names
## @loc.names: vector of  82 locus names
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  212 columns of @tab
## @all.names: list of  82 components yielding allele names for each locus
## @ploidy:  1 1 1 1 1 1
## @type:  codom
##
## Optional contents:
## @strata: - empty -
## @hierarchy:  - empty -
```

```
## @pop:  - empty -
## @pop.names:  - empty -
##
## @other: a list containing: com
```

The six aligned protein sequences (`mase.res`) have been scanned for polymorphic sites, and these have been extracted to form the `genind` object `x`. Note that several settings such as the characters corresponding to missing values (i.e., gaps) and the polymorphism threshold for a site to be retained can be specified through the function's arguments (see `?alignment2genind`).

The names of the loci directly provides the indices of polymorphic sites:

```
head(locNames(x))

## [1] "3"  "4"  "5"  "6"  "9"  "11"
```

The table of polymorphic sites can be reconstructed easily by:

```
tabAA <- genind2df(x)
dim(tabAA)

## [1]  6 82

tabAA[, 1:20]

##        3 4 5 6 9 11 12 15 16 17 18 19 21 22 24 28 30 32 33 34
## Langur i f e r l  r  t  k  l  g  l  d  y  k  v  n  v  l  a  k
## Baboon i f e r l  r  t  r  l  g  l  d  y  r  i  n  v  l  a  k
## Human  v f e r l  r  t  r  l  g  m  d  y  r  i  n  m  l  a  k
## Rat    t y e r f  r  t  r  n  g  m  s  y  y  v  d  v  l  a  q
## Cow    v f e r l  r  t  k  l  g  l  d  y  k  v  n  l  l  t  k
## Horse  v f s k l  h  k  a  q  e  m  d  f  g  y  n  v  m  a  e
```

The global AA composition of the polymorphic sites is given by:

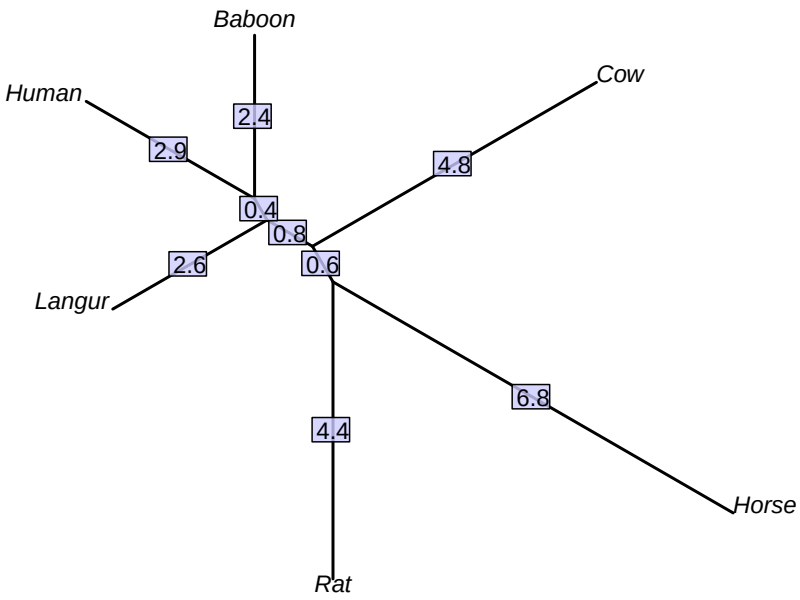
```
table(unlist(tabAA))

##
##  a  d  e  f  g  h  i  k  l  m  n  p  q  r  s  t  v  w  y
## 35 38 16  9 33 13 27 28 31  8 44 10 26 47 36 20 42  6 23
```

Now that polymorphic sites have been converted into a `genind` object, simple distances can be computed between the sequences. Note that *adegenet* does not implement specific distances for protein sequences, we only use the simple Euclidean distance. Fancier protein distances are implemented in R; see for instance `dist.alignment` in the *seqinr* package, and `dist.ml` in the *phangorn* package.

```
D <- dist(tab(x))
D

##           Langur    Baboon     Human       Rat       Cow
## Baboon  5.291503
## Human   6.000000  5.291503
## Rat     8.717798  8.124038  8.602325
## Cow     7.874008  8.717798  8.944272 10.392305
## Horse  11.313708 11.313708 11.224972 11.224972 11.747340
```

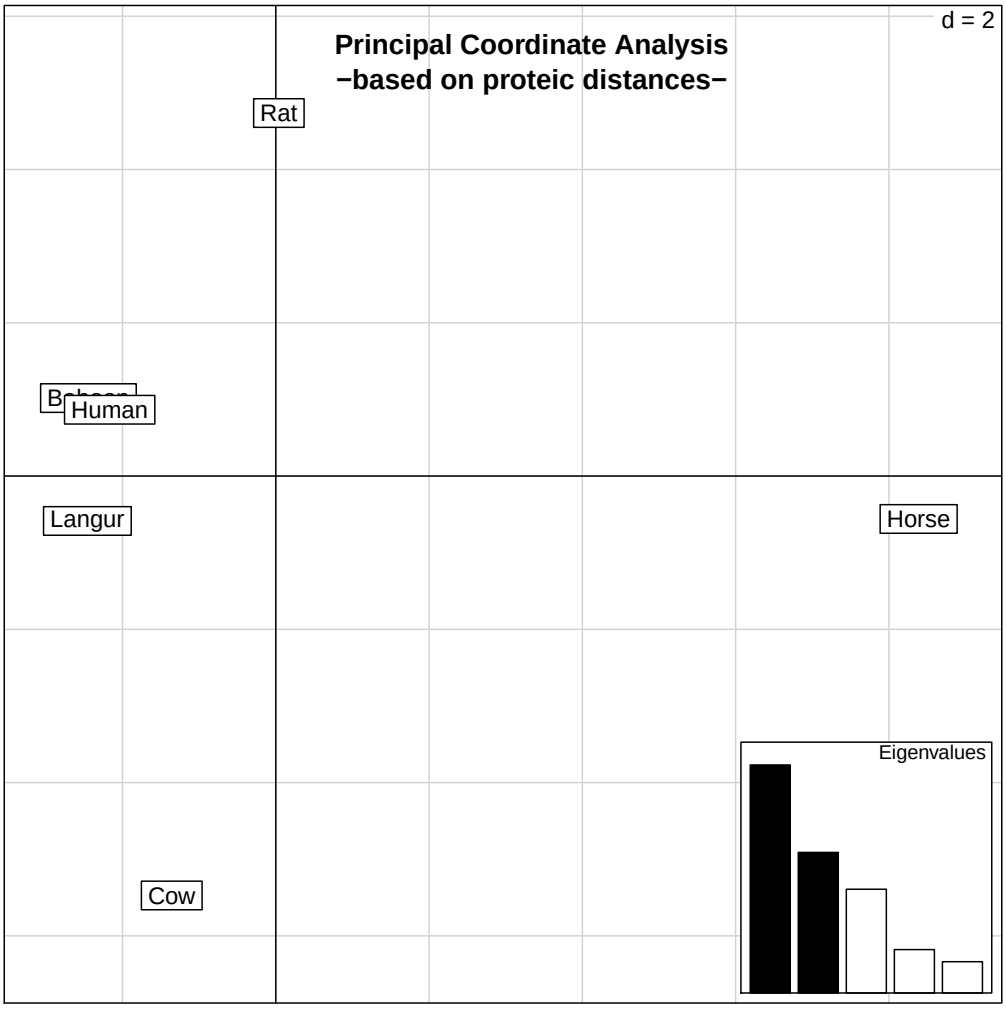
This matrix of distances is small enough for one to interprete the raw numbers. However, it is also very straightforward to represent these distances as a tree or in a reduced space. We first build a Neighbor-Joining tree using the *ape* package:

```
library(ape)
tre <- nj(D)
par(xpd=TRUE)
plot(tre, type="unrooted", edge.w=2)
edgelabels(tex=round(tre$edge.length,1), bg=rgb(.8,.8,1,.8))
```



The best possible planar representation of these Euclidean distances is achieved by Principal Coordinate Analyses (PCoA), which in this case will give identical results to PCA of the original (centred, non-scaled) data:

```
pco1 <- dudi.pco(D, scannf=FALSE,nf=2)
scatter(pco1, posi="bottomright")
title("Principal Coordinate Analysis\n-based on proteic distances-")
```



4.7 Using genind/genpop constructors

`genind` or `genpop` objects can be constructed from data matrices similar to the `$tab` component (respectively, alleles frequencies and alleles counts). This is achieved by the constructors `genind` (or `as.genind`) and `genpop` (or `as.genpop`). However, these low-level functions are first meant for internal use, and are called for instance by functions such as `read.genetix`. Consequently, there is much less control on the arguments and improper specification can lead to creating improper `genind`/`genpop` objects without issuing a warning or an error. One should therefore use these functions with additional care as to how information is coded. The table passed as argument to these constructors must have

correct names: unique rownames identifying genotypes/populations, and unique colnames having the form '[marker].[allele]'.

Here is an example for **genpop** using a dataset from *ade4*:

```
data(microsatt)
microsatt$tab[10:15,12:15]

##            INRA32.168 INRA32.170 INRA32.174 INRA32.176
## Mtbeliard           0          0          0          1
## NDama               0          0          0         12
## Normand             1          0          0          2
## Parthenais          8          5          0          3
## Somba               0          0          0         20
## Vosgienne           2          0          0          0
```

`microsatt$tab` contains alleles counts per populations, and can therefore be used to make a **genpop** object. Moreover, column names are set as required, and row names are unique. It is therefore safe to convert these data into a **genpop** using the constructor:

```
toto <- genpop(microsatt$tab)

## Error in .local(.Object, ...):  (list) object cannot be coerced to type
'integer'

toto

##  /// genind object \\\
## @call: read.genetix(file = system.file("files/nancycats.gtx", package = "adegenet"))
##
## @tab:  237 x 108 matrix of genotypes
##
## @ind.names: vector of  237 individual names
## @loc.names: vector of  9 locus names
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  108 columns of @tab
## @all.names: list of  9 components yielding allele names for each locus
## @ploidy:  2 2 2 2 2 2
## @type:  codom
##
## Optional contents:
## @strata: - empty -
## @hierarchy:  - empty -
## @pop:  factor giving the population of each individual
## @pop.names:  character giving the name of each population
##
## @other: - empty -
```

```
summary(toto)

##
##  # Total number of genotypes:  237
##
##  # Population sample sizes:
##  1 10 11 12 13 14 15 16 17  2  3  4  5  6  7  8  9
## 10 11 20 14 13 17 11 12 13 22 12 23 15 11 14 10  9
##
##  # Number of alleles per locus:
##  fca8 fca23 fca43 fca45 fca77 fca78 fca90 fca96 fca37
##    16    11    10     9    12     8    12    12    18
##
##  # Number of alleles per population:
##  1 10 11 12 13 14 15 16 17  2  3  4  5  6  7  8  9
## 36 46 70 52 44 61 42 40 35 53 50 67 48 56 42 54 43
##
##  # Percentage of missing data:
## [1] 2.344116
##
##  # Observed heterozygosity:
##      fca8     fca23     fca43     fca45     fca77     fca78     fca90
## 0.6682028 0.6666667 0.6793249 0.7083333 0.6329114 0.5654008 0.6497890
##     fca96     fca37
## 0.6184211 0.4514768
##
##  # Expected heterozygosity:
##      fca8     fca23     fca43     fca45     fca77     fca78     fca90
## 0.8657224 0.7928751 0.7953319 0.7603095 0.8702576 0.6884669 0.8157881
##     fca96     fca37
## 0.7603493 0.6062686
```

4.8 Exporting data

Genotypes in `genind` format can be exported to the R packages *genetics* (using `genind2genotype`) and *hierfstat* (using `genind2hierfstat`). The package *genetics* is now deprecated, but the implemented class `genotype` is still used in various packages. The package *hierfstat* does not define a class, but requires data to be formated in a particular way. It has been removed from CRAN as of R version 2.13.0 for maintainance issues, but is supposed to be back eventually.

Here are examples of how to use these functions:

```
obj <- genind2genotype(nancycats)

## Error in eval(expr, envir, enclos):  could not find function
"genind2genotype"

class(obj)

## [1] "genind"
## attr(,"package")
## [1] "adegenet"

obj[1:4,1:5]

##  /// genind object \\\
## @call: .local(x = x, i = i, j = j, drop = drop)
##
## @tab:  4 x 5 matrix of genotypes
##
## @ind.names: vector of  4 individual names
## @loc.names: vector of  3 locus names
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  5 columns of @tab
## @all.names: list of  3 components yielding allele names for each locus
## @ploidy:  1 1 1 1
## @type:  codom
##
## Optional contents:
## @strata: - empty -
## @hierarchy:  - empty -
## @pop:  - empty -
## @pop.names:  - empty -
##
## @other: a list containing: elements without names

class(obj$fca8)

## Error in slot(x, name):  no slot of name "fca8" for this object of class
"genind"
```

```
obj <- genind2hierfstat(nancycats)
class(obj)

## [1] "data.frame"

obj[1:4,1:5]
```

```
##       pop fca8 fca23 fca43 fca45
## cat.1   1   NA   409   404   103
## cat.2   1   NA   909   407   305
## cat.3   1  913   409   505   101
## cat.4   1  809   505   405   105
```

A more generic way to export data is to produce a `data.frame` of genotypes coded by character strings. This is done by `genind2df`:

```
obj <- genind2df(nancycats)
obj[1:5,1:5]

##       pop   fca8  fca23  fca43  fca45
## cat.1   1   <NA> 136146 139139 116120
## cat.2   1   <NA> 146146 139145 120126
## cat.3   1 135143 136146 141141 116116
## cat.4   1 133135 138138 139141 116126
## cat.5   1 133135 140146 141145 126126
```

However, some software will require alleles to be separated. The argument `sep` allows one to specify any separator. For instance:

```
genind2df(nancycats,sep="|")[1:5,1:5]

##       pop    fca8   fca23   fca43   fca45
## cat.1   1    <NA> 136|146 139|139 116|120
## cat.2   1    <NA> 146|146 139|145 120|126
## cat.3   1 135|143 136|146 141|141 116|116
## cat.4   1 133|135 138|138 139|141 116|126
## cat.5   1 133|135 140|146 141|145 126|126
```

Note that tabulations can be obtained as follows using '\t' character.

5 Basics of data analysis

5.1 Manipulating the data

Data manipulation is meant to be particularly flexible in *adegenet*. First, as `genind` and `genpop` objects are basically formed by a data matrix (the `@tab` slot), it is natural to subset these objects like it is done with a matrix. The [operator does this, forming a new object with the retained genotypes/populations and alleles:

```
data(microbov)
toto <- genind2genpop(microbov)

##
##  Converting data from a genind to a genpop object...
##
## ...done.

toto

##  /// genpop object \\\
## @call: genind2genpop(x = microbov)
##
## @tab:  15 x 373 matrix of alleles counts
##
## @pop.names: vector of  15 population names
## @loc.names: vector of  30 locus names
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  373 columns of @tab
## @all.names: list of  30 components yielding allele names for each locus
## @ploidy:  2
## @type:  codom
##
## @other: a list containing: coun  breed  spe

toto@pop.names

##  [1] "Borgou"          "Zebu"            "Lagunaire"
##  [4] "NDama"           "Somba"           "Aubrac"
##  [7] "Bazadais"        "BlondeAquitaine" "BretPieNoire"
## [10] "Charolais"       "Gascon"          "Limousin"
## [13] "MaineAnjou"      "Montbeliard"     "Salers"

titi <- toto[1:3,]
titi@pop.names

## [1] "Borgou"    "Zebu"      "Lagunaire"
```

The object `toto` has been subsetted, keeping only the first three populations. Of course, any subsetting available for a matrix can be used with `genind` and `genpop` objects. In addition, we can subset loci directly using the generic marker names:

```
tata <- titi[,loc="L03"]
tata

##  /// genpop object \\\
## @call: .local(x = x, i = i, j = j, loc = "L03", drop = drop)
##
## @tab:  3 x 0 matrix of alleles counts
##
## @pop.names: vector of  3 population names
## @loc.names: vector of  0 locus names
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  0 columns of @tab
## @all.names: list of  0 components yielding allele names for each locus
## @ploidy:  2
## @type:  codom
##
## @other: a list containing: coun  breed  spe
```

Now, `tata` only contains the 12 alleles of the third marker of `titi`.

To simplify the task of separating data by marker, the function `seploc` can be used. It returns a list of objects (optionnaly, of data matrices), each corresponding to a marker:

```
data(nancycats)
sepCats <- seploc(nancycats)
class(sepCats)

## [1] "list"

names(sepCats)

## [1] "fca8"  "fca23" "fca43" "fca45" "fca77" "fca78" "fca90" "fca96" "fca37"

sepCats$fca45

##  /// genind object \\\
## @call: .local(x = x)
##
## @tab:  237 x 9 matrix of genotypes
##
## @ind.names: vector of  237 individual names
## @loc.names: vector of  1 locus names
```

```
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  9 columns of @tab
## @all.names: list of  1 components yielding allele names for each locus
## @ploidy:  2 2 2 2 2 2
## @type:  codom
##
## Optional contents:
## @strata: - empty -
## @hierarchy:  - empty -
## @pop:  factor giving the population of each individual
## @pop.names:  character giving the name of each population
##
## @other: a list containing: xy
```

The object `sepCats$fca45` only contains data of the marker fca45.

Following the same idea, `seppop` allows one to separate genotypes in a `genind` object by population. For instance, we can separate genotype of cattles in the dataset `microbov` by breed:

```
data(microbov)
obj <- seppop(microbov)
class(obj)

## [1] "list"

names(obj)

##  [1] "Borgou"          "Zebu"            "Lagunaire"
##  [4] "NDama"           "Somba"           "Aubrac"
##  [7] "Bazadais"        "BlondeAquitaine" "BretPieNoire"
## [10] "Charolais"       "Gascon"          "Limousin"
## [13] "MaineAnjou"      "Montbeliard"     "Salers"

obj$Borgou

##  /// genind object \\\
## @call: .local(x = x, i = i, j = j, treatOther = ..1, quiet = ..2, drop = drop)
##
## @tab:  50 x 373 matrix of genotypes
##
## @ind.names: vector of  50 individual names
## @loc.names: vector of  30 locus names
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  373 columns of @tab
```

```
## @all.names: list of  30 components yielding allele names for each locus
## @ploidy:  2 2 2 2 2 2
## @type:  codom
##
## Optional contents:
## @strata: - empty -
## @hierarchy:  - empty -
## @pop:  factor giving the population of each individual
## @pop.names:  character giving the name of each population
##
## @other: a list containing: coun  breed  spe
```

The returned object `obj` is a list of `genind` objects each containing genotypes of a given breed.

A last, rather vicious trick is to separate data by population and by marker. This is easy using `lapply`; one can first separate population then markers, or the contrary. Here, we separate markers inside each breed in `obj`:

```
obj <- lapply(obj,seploc)
names(obj)

##  [1] "Borgou"          "Zebu"            "Lagunaire"
##  [4] "NDama"           "Somba"           "Aubrac"
##  [7] "Bazadais"        "BlondeAquitaine" "BretPieNoire"
## [10] "Charolais"       "Gascon"          "Limousin"
## [13] "MaineAnjou"      "Montbeliard"     "Salers"

class(obj$Borgou)

## [1] "list"

names(obj$Borgou)

##  [1] "INRA63"  "INRA5"   "ETH225"  "ILSTS5"  "HEL5"    "HEL1"    "INRA35"
##  [8] "ETH152"  "INRA23"  "ETH10"   "HEL9"    "CSSM66"  "INRA32"  "ETH3"
## [15] "BM2113"  "BM1824"  "HEL13"   "INRA37"  "BM1818"  "ILSTS6"  "MM12"
## [22] "CSRM60"  "ETH185"  "HAUT24"  "HAUT27"  "TGLA227" "TGLA126" "TGLA122"
## [29] "TGLA53"  "SPS115"

obj$Borgou$INRA63

##  /// genind object \\\
## @call: .local(x = x)
##
## @tab:  50 x 9 matrix of genotypes
##
```

```
## @ind.names: vector of  50 individual names
## @loc.names: vector of  1 locus names
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  9 columns of @tab
## @all.names: list of  1 components yielding allele names for each locus
## @ploidy:  2 2 2 2 2 2
## @type:  codom
##
## Optional contents:
## @strata: - empty -
## @hierarchy:  - empty -
## @pop:  factor giving the population of each individual
## @pop.names:  character giving the name of each population
##
## @other: a list containing: coun  breed  spe
```

For instance, `obj$Borgou$INRA63` contains genotypes of the breed Borgou for the marker INRA63.

Lastly, one may want to pool genotypes in different datasets, but having the same markers, into a single dataset. This is more than just merging the `@tab` components of all datasets, because alleles can differ (they almost always do) and markers are not necessarily sorted the same way. The function `repool` is designed to avoid these problems. It can merge any `genind` provided as arguments as soon as the same markers are used. For instance, it can be used after a `seppop` to retain only some populations:

```
obj <- seppop(microbov)
names(obj)

##  [1] "Borgou"          "Zebu"            "Lagunaire"
##  [4] "NDama"           "Somba"           "Aubrac"
##  [7] "Bazadais"        "BlondeAquitaine" "BretPieNoire"
## [10] "Charolais"       "Gascon"          "Limousin"
## [13] "MaineAnjou"      "Montbeliard"     "Salers"

newObj <- repool(obj$Borgou, obj$Charolais)
newObj

##  /// genind object \\\
## @call: repool(obj$Borgou, obj$Charolais)
##
## @tab:  105 x 295 matrix of genotypes
##
## @ind.names: vector of  105 individual names
## @loc.names: vector of  30 locus names
```

```
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  295 columns of @tab
## @all.names: list of  30 components yielding allele names for each locus
## @ploidy:  2 2 2 2 2 2
## @type:  codom
##
## Optional contents:
## @strata: - empty -
## @hierarchy:  - empty -
## @pop:  factor giving the population of each individual
## @pop.names:  character giving the name of each population
##
## @other: - empty -

newObj$pop.names

## [1] "Borgou"    "Charolais"
```

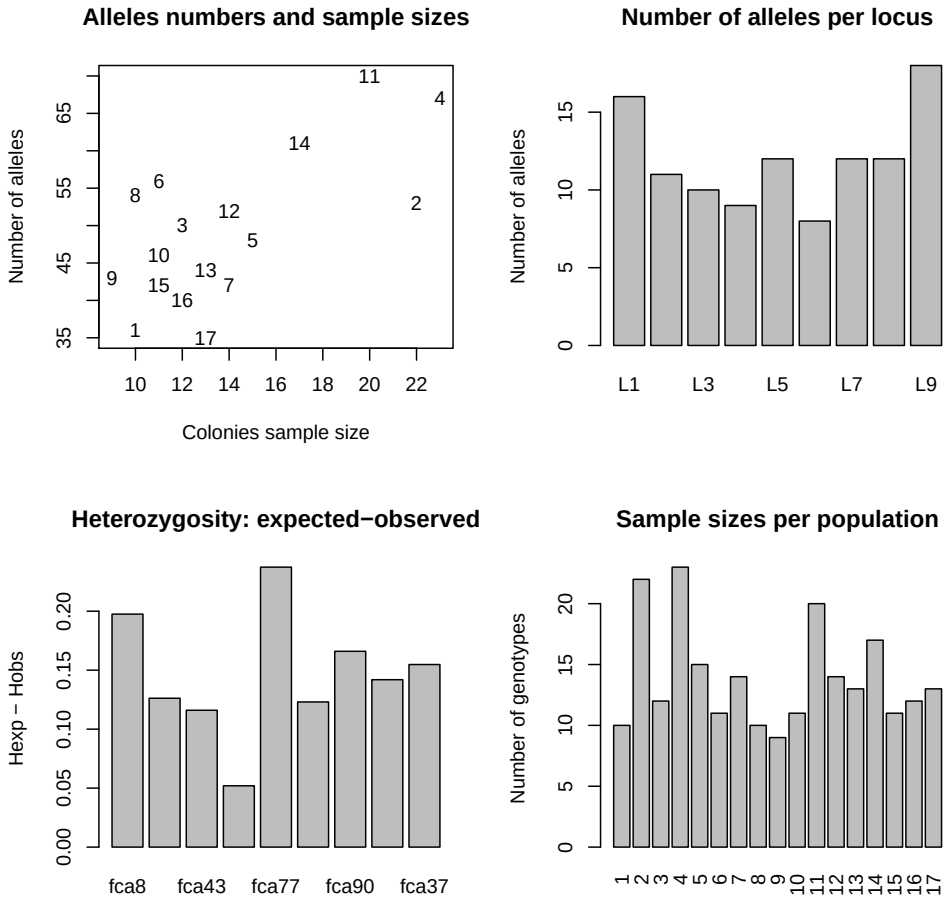
Done !

5.2 Using summaries

Both `genind` and `genpop` objects have a summary providing basic information about data. Informations are both printed and invisibly returned as a list.

```
toto <- summary(nancycats)

##
##  # Total number of genotypes:  237
##
##  # Population sample sizes:
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
## 10 22 12 23 15 11 14 10  9 11 20 14 13 17 11 12 13
##
##  # Number of alleles per locus:
## L1 L2 L3 L4 L5 L6 L7 L8 L9
## 16 11 10  9 12  8 12 12 18
##
##  # Number of alleles per population:
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
## 36 53 50 67 48 56 42 54 43 46 70 52 44 61 42 40 35
##
##  # Percentage of missing data:
## [1] 2.344116
##
```

```
##  # Observed heterozygosity:
##      fca8     fca23     fca43     fca45     fca77     fca78     fca90
## 0.6682028 0.6666667 0.6793249 0.7083333 0.6329114 0.5654008 0.6497890
##     fca96     fca37
## 0.6184211 0.4514768
##
##  # Expected heterozygosity:
##      fca8     fca23     fca43     fca45     fca77     fca78     fca90
## 0.8657224 0.7928751 0.7953319 0.7603095 0.8702576 0.6884669 0.8157881
##     fca96     fca37
## 0.7603493 0.6062686

names(toto)

## [1] "N"        "pop.eff"  "loc.nall" "pop.nall" "NA.perc"  "Hobs"
## [7] "Hexp"

par(mfrow=c(2,2))

plot(toto$pop.eff, toto$pop.nall, xlab="Colonies sample size",
     ylab="Number of alleles",main="Alleles numbers and sample sizes",
     type="n")
text(toto$pop.eff,toto$pop.nall,lab=names(toto$pop.eff))

barplot(toto$loc.nall, ylab="Number of alleles",
        main="Number of alleles per locus")

barplot(toto$Hexp-toto$Hobs, main="Heterozygosity: expected-observed",
        ylab="Hexp - Hobs")

barplot(toto$pop.eff, main="Sample sizes per population",
        ylab="Number of genotypes",las=3)
```



Is mean observed H significantly lower than mean expected H ?

```
bartlett.test(list(toto$Hexp,toto$Hobs))

##
##  Bartlett test of homogeneity of variances
##
## data:  list(toto$Hexp, toto$Hobs)
## Bartlett's K-squared = 0.046962, df = 1, p-value = 0.8284

t.test(toto$Hexp,toto$Hobs,pair=T,var.equal=TRUE,alter="greater")

##
##  Paired t-test
##
## data:  toto$Hexp and toto$Hobs
## t = 8.3294, df = 8, p-value = 1.631e-05
## alternative hypothesis: true difference in means is greater than 0
## 95 percent confidence interval:
```

```
##  0.1134779       Inf
## sample estimates:
## mean of the differences
##               0.1460936
```

Yes, it is.

5.3 Testing for Hardy-Weinberg equilibrium

The Hardy-Weinberg equilibrium test is implemented for `genind` objects. The function to use is `HWE.test.genind`, and requires the package *genetics*. Here we first produce a matrix of p-values (`res="matrix"`) using parametric test. Monte Carlo procedure are more reliable but also more computer-intensive (use `permut=TRUE`).

```
toto <- HWE.test.genind(nancycats,res="matrix")

## As of adegenet_1.5-0, this function has been removed and is replaced by
'hw.test' in the package 'pegas'

dim(toto)

## NULL
```

One test is performed per locus and population, *i.e.* 153 tests in this case. Thus, the first question is: which tests are highly significant?

```
colnames(toto)

## NULL

idx <- which(toto<0.0001,TRUE)
idx

## integer(0)
```

Here, only 4 tests indicate departure from HW. Rows give populations, columns give markers. Now complete tests are returned, but the significant ones are already known.

```
toto <- HWE.test.genind(nancycats,res="full")

## As of adegenet_1.5-0, this function has been removed and is replaced by
'hw.test' in the package 'pegas'

mapply(function(i,j) toto[[i]][[j]], idx[,2], idx[,1], SIMPLIFY=FALSE)

## Error in idx[, 2]:  incorrect number of dimensions
```

5.4 Measuring and testing population structure (a.k.a F statistics)

Population structure is traditionally measured and tested using F statistics, in particular Fst. Since version 2.13.0 of R, the package *hierfstat*, which implemented most F statistics and related tests, has been removed from CRAN for maintenance issues. As a consequence, *adegenet* has lost a few functionalities, namely general F statistics (function `fstat`) and a test of overall population structure (`gstat.randtest`).

However, it is still possible to compute pairwise Fst using *adegenet*. Pairwise Fst is frequently used as a measure of distance between populations. The function `pairwise.fst` computes Nei's estimator [10] of pairwise Fst, defined as:

$$Fst(A,B) = \frac{H_t - (n_A H_s(A) + n_B H_s(B))/(n_A + n_B)}{Ht}$$

where A and B refer to the two populations of sample size n_A and n_B and respective expected heterozygosity $H_s(A)$ and $H_s(B)$, and H_t is the expected heterozygosity in the whole dataset. For a given locus, expected heterozygosity is computed as $1 - \sum p_i^2$, where p_i is the frequency of the ith allele, and the $\sum$ represents summation over all alleles. For multilocus data, the heterozygosity is simply averaged over all loci. These computations are achieved for all pairs of populations by the function `pairwise.fst`; we illustrate this on a subset of individuals of `nancycats` (computations for the whole dataset would take a few tens of seconds):

```
data(nancycats)
matFst <- pairwise.fst(nancycats[1:50, treatOther=FALSE])

## This function has been moved to the package hierfstat.  You can still
use it in adegenet, but this version is deprecated.

matFst

##            1          2          3
## 2 0.08018500
## 3 0.07140847 0.08200880
## 4 0.08163151 0.06512457 0.04131227
```

The resulting matrix is Euclidean when there are no missing values:

```
is.euclid(matFst)

## [1] TRUE
```

It can therefore be used in a Principal Coordinate Analysis (which requires Euclideanity), used to build trees, etc.

5.5 Estimating inbreeding

Inbreeding refers to an excess of homozygosity in a given individual due to the mating of genetically related parents. This excess of homozygosity is due to the fact that there are

non-negligible chances of inheriting two identical alleles from a recent common ancestor. Inbreeding can be associated to a loss of fitness leading to "*inbreeding depression*". Typically, loss of fitness is caused by recessive deleterious alleles which have usually low frequency in the population, but for which inbred individuals are more likely to be homozygotes.

The inbreeding coefficient F is defined as the probability that at a given locus, two identical alleles have been inherited from a common ancestor. In the absence of inbreeding, the probability of being homozygote at one loci is (for diploid individuals) simply $\sum_i p_i^2$ where i indexes the alleles and p_i is the frequency of allele i. This can be generalized incorporating F as:

$$p(\text{homozygote}) = F + (1 - F) \sum_i p_i^2$$

and even more generally, for any ploidy π:

$$p(\text{homozygote}) = F + (1 - F) \sum_i p_i^\pi$$

This therefore allows for computing the likelihood of a given state (homozygote/heterozygote) in a given genotype (log-likelihood are summed across loci for more than one marker).

This estimation is achieved by `inbreeding`. Depending on the value of the argument `res.type`, the function returns a sample from the likelihood function (`res.type='sample'`) or the likelihood function itself, as a R function (`res.type='function'`). While likelihood functions are quickly obtained and easy to display graphically, sampling from the distributions is more computer intensive but useful to derive summary statistics of the distributions. Here, we illustrate `inbreeding` using the `microbov` dataset, which contains cattle breeds genotypes for 30 microsatellites; to focus on breed Salers only, we use `seppop`:

```
data(microbov)
sal <- seppop(microbov)$Salers
sal

##  /// genind object \\\
## @call: .local(x = x, i = i, j = j, treatOther = ..1, quiet = ..2, drop = drop)
##
## @tab:  50 x 373 matrix of genotypes
##
## @ind.names: vector of  50 individual names
## @loc.names: vector of  30 locus names
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  373 columns of @tab
## @all.names: list of  30 components yielding allele names for each locus
## @ploidy:  2 2 2 2 2 2
## @type:  codom
##
```

```
## Optional contents:
## @strata: - empty -
## @hierarchy:  - empty -
## @pop:  factor giving the population of each individual
## @pop.names:  character giving the name of each population
##
## @other: a list containing: coun  breed  spe
```

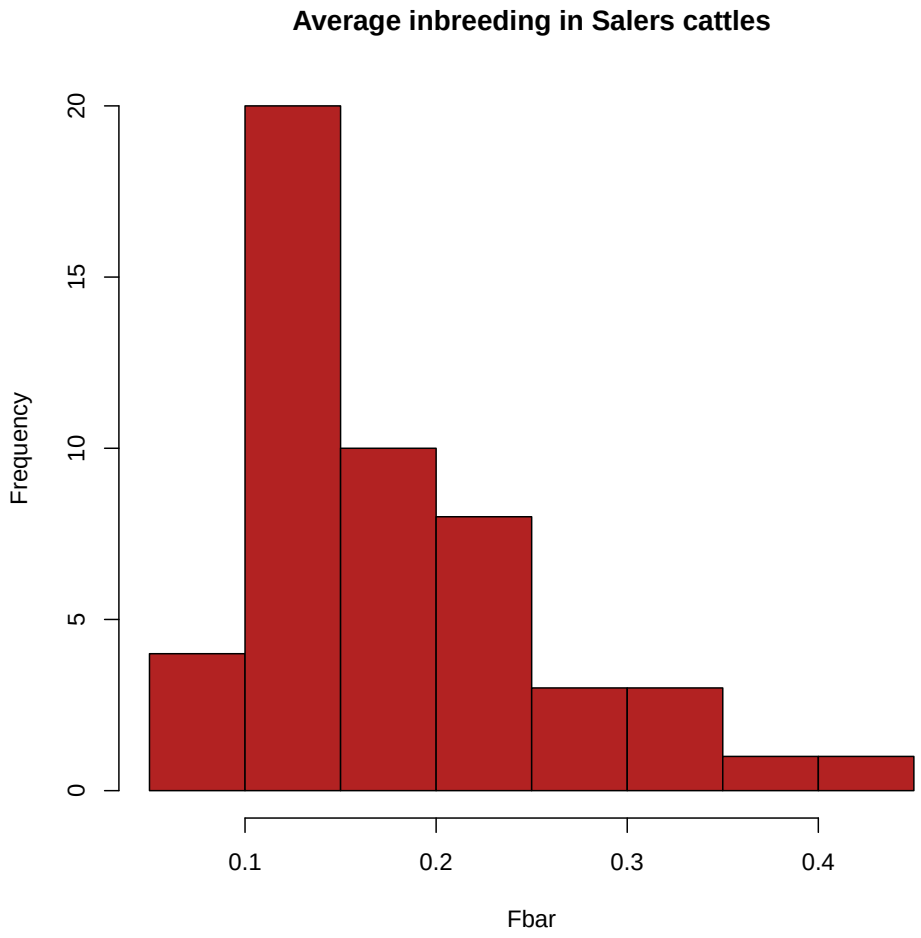
We first compute the mean inbreeding for each individual, and plot the resulting distribution:

```
temp <- inbreeding(sal, N=100)
class(temp)

## [1] "list"

head(names(temp))

## [1] "FRBTSAL9087" "FRBTSAL9088" "FRBTSAL9089" "FRBTSAL9090" "FRBTSAL9091"
## [6] "FRBTSAL9093"

head(temp[[1]],20)

##  [1] 0.102601783 0.099380929 0.057174351 0.051679721 0.071844846
##  [6] 0.225344406 0.192247657 0.165908925 0.001918136 0.032728112
## [11] 0.078722625 0.244838287 0.064105791 0.068488304 0.115756070
## [16] 0.059022418 0.161049330 0.012885339 0.417141840 0.236279445
```

`temp` is a list of values sampled from the likelihood distribution of each individual; means values are obtained for all individuals using `sapply`:

```
Fbar <- sapply(temp, mean)
```

```
hist(Fbar, col="firebrick", main="Average inbreeding in Salers cattles")
```



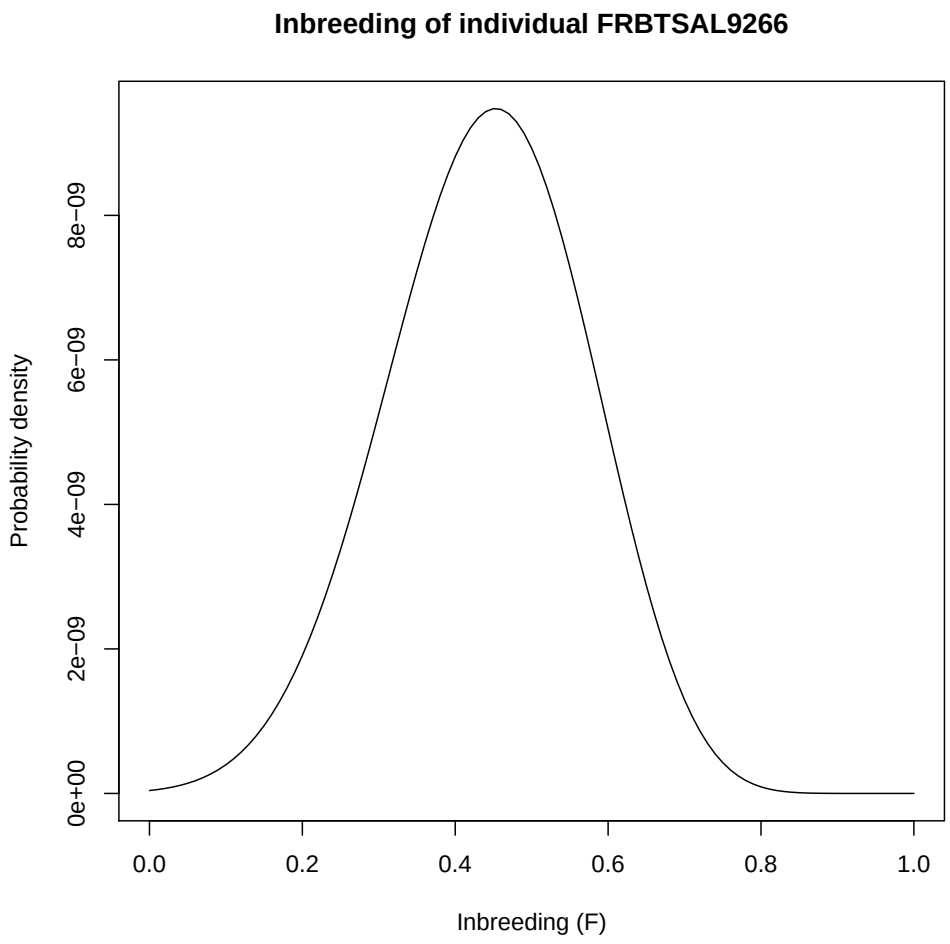
We can see that some individuals (actually, a single one) have higher inbreeding (>0.4). We can recompute inbreeding for this individual, asking for the likelihood function to be returned:

```
which(Fbar>0.4)

## FRBTSAL9266
##          37

F <- inbreeding(sal, res.type="function")[which(Fbar>0.4)]
F

## $FRBTSAL9266
## function (F)
## {
##     args <- lapply(as.list(match.call())[-1L], eval, parent.frame())
##     names <- if (is.null(names(args)))
##         character(length(args))
##     else names(args)
```

```
##     dovec <- names %in% vectorize.args
##     do.call("mapply", c(FUN = FUN, args[dovec], MoreArgs = list(args[!dovec]),
##         SIMPLIFY = SIMPLIFY, USE.NAMES = USE.NAMES))
## }
## <environment: 0xdfae1d0>
```

The output object `F` can seem a bit cryptic: it is an function embedded within a hidden environment. This does not matter, however, since it is easily represented:

```
plot(F$FRBTSAL9266, main=paste("Inbreeding of individual",names(F)),
     xlab="Inbreeding (F)", ylab="Probability density")
```



Indeed, this individual shows subsequent inbreeding, with about 50% chances of being homozygote through inheritance from a common ancestor of its parents.

6 Multivariate analysis

6.1 General overview

Multivariate analysis consists in summarising a strongly multivariate information into a few synthetic variables. In genetics, such approaches are useful to get a simplified picture of the genetic diversity obersved amongst individuals or populations. A review of multivariate analysis in population genetics can be found in [7]. Here, we aim at providing an overview of some applications using methods implemented in *ade4* and *adegenet*.

Useful functions include:

- `scaleGen` (*adegenet*): centre/scale allele frequencies and replaces missing data; useful, among other things, before running a principal component analysis (PCA).
- `dudi.pca` (*ade4*): implements PCA; can be used on transformed allele frequencies of individuals or populations.
- `dudi.ca` (*ade4*): implements Correspondance Analysis (CA); can be used on raw allele counts of populations (`@tab` slot in `genpop` objects).
- `dist.genpop` (*adegenet*): implements 5 pairwise genetic distances between populations
- `pairwise.fst` (*adegenet*): implements pairwise F_{ST}, which is also a Euclidean distance between populations.
- `dist` (*stats*): computes pairwise distances between multivariate observations; can be used on raw or transformed allele frequencies.
- `dudi.pco` (*ade4*): implements Principal Coordinates Analysis (PCoA); this methods finds synthetic variables which summarize a Euclidean distance matrix as best as possible; can be used on outputs of `dist`, `dist.genpop`, and `pairwise.fst`.
- `is.euclid` (*ade4*): tests whether a distance matrix is Euclidean, which is a pre-requisite of PCoA.
- `cailliez` (*ade4*): renders a non-Euclidean distance matrix Euclidean by adding a constant to all entries.
- `dapc` (*adegenet*): implements the Discriminant Analysis of Principal Components (DAPC [4]), a powerful method for the analysis of population genetic structures; see dedicated tutorial (*dapc*).
- `sPCA` (*adegenet*): implements the spatial Principal Component Analysis (sPCA [5]), a method for the analysis of spatial genetic structures; see dedicated tutorial (*dapc*).
- `glPca` (*adegenet*): implements PCA for genome-wide SNP data stored as `genlight` objects; see dedicated tutorial (*genomics*).

Besides the procedures themselves, graphic functions are also often of the utmost importance; these include:

- `scatter` (*ade4,adegenet*): generic function to display multivariate analyses; in practice, the most useful application for genetic data is the one implemented in `adegenet` for DAPC results.
- `s.label` (*ade4*): function used for basic display of principal components.
- `loadingplot` (*adegenet*): function used to display the loadings (i.e., contribution to a given structure) of alleles for a given principal component; annotates and returns the most contributing alleles.
- `s.class` (*ade4*): displays two quantitative variables with known groups of observations, using inertia ellipses for the groups; useful to represent principal components when groups are known.
- `s.chull` (*ade4*): same as `s.class`, except convex polygons are used rather than ellipses.
- `s.value` (*ade4*): graphical display of a quantitative variable distributed over a two-dimensional space; useful to map principal components or allele frequencies over a geographic area.
- `colorplot` (*adegenet*): graphical display of 1 to 3 quantitative variables distributed over a two-dimensional space; useful for combined representations of principal components over a geographic area. Can also be used to produce color versions of traditional scatterplots.
- `transp` (*adegenet*): auxiliary function making colors transparent.
- `num2col` (*adegenet*): auxiliary function transforming a quantitative variable into colors using a given palette.
- `assignplot` (*adegenet*): specific plot of group membership probabilities for DAPC; see dedicated tutorial (*dapc*).
- `compoplot` (*adegenet*): specific 'STRUCTURE-like' plot of group membership probabilities for DAPC; see dedicated tutorial (*dapc*).
- `add.scatter` (*ade4*): add inset plots to an existing figure.
- `add.scatter.eig` (*ade4*): specific application of `add.scatter` to add barplots of eigenvalues to an existing figure.

In the sections below, we briefly illustrate how these tools can be combined to extract information from genetic data.

6.2 Performing a Principal Component Analysis on genind objects

The tables contained in **genind** objects can be submitted to a Principal Component Analysis (PCA) to seek a summary of the genetic diversity among the sampled individuals. Such analysis is straightforward using *adegenet* to prepare data and *ade4* for the analysis *per se*. One has first to replace missing data (NAs) and transform the allele frequencies in an appropriate way. These operations are achieved by **scaleGen**. NAs are replaced by the mean allele frequency; different scaling options are available (argument **method**), but in general centring is sufficient since allele frequencies have inherently comparable variances.

```
data(microbov)
sum(is.na(microbov$tab))

## [1] 6325
```

There are 6325 missing data. They will all be replaced by **scaleGen**:

```
X <- scaleGen(microbov, missing="mean")

## Error in .local(x, ...):  unused argument (missing = "mean")

class(X)

## Error in eval(expr, envir, enclos):  object 'X' not found

dim(X)

## Error in eval(expr, envir, enclos):  object 'X' not found

X[1:5,1:5]

## Error in eval(expr, envir, enclos):  object 'X' not found
```

Note that alternatively, we could have used **na.replace** to replace missing data, and then left the centring/scaling to **dudi.pca**.

The analysis can now be performed. We disable the scaling in **dudi.pca**, which would erase the scaling choice made earlier in **scaleGen**. Note: in practice, retained axes can be chosen interactively by removing the arguments **scannf=FALSE,nf=3**.

```
pca1 <- dudi.pca(X,cent=FALSE,scale=FALSE,scannf=FALSE,nf=3)

## Error in as.data.frame(df):  object 'X' not found

barplot(pca1$eig[1:50],main="PCA eigenvalues", col=heat.colors(50))

## Error in barplot(pca1$eig[1:50], main = "PCA eigenvalues", col =
## heat.colors(50)):  object 'pca1' not found
```

```
pca1
```

```
## Error in eval(expr, envir, enclos):  object 'pca1' not found
```

The output object `pca1` is a list containing various information; of particular interest are:

- `$eig`: the eigenvalues of the analysis, indicating the amount of variance represented by each principal component (PC).
- `$li`: the principal components of the analysis; these are the synthetic variables summarizing the genetic diversity, usually visualized using scatterplots.
- `$c1`: the allele loadings, used to compute linear combinations forming the PCs; squared, they represent the contribution to each PCs.

The basic scatterplot for this analysis can be obtained by:

```
s.label(pca1$li)
```

```
## Error in data.frame(dfxy):  object 'pca1' not found
```

```
title("PCA of microbov dataset\naxes 1-2")
```

```
## Error in title("PCA of microbov dataset\naxes 1-2"):  plot.new has not
been called yet
```

```
add.scatter.eig(pca1$eig[1:20], 3,1,2)
```

```
## Error in scatterutil.eigen(w, nf = nf, wsel = c(xax, yax), sub = sub, :
object 'pca1' not found
```

However, this figure can largely be improved. First, we can use `s.class` to represent both the genotypes and inertia ellipses for populations.

```
s.class(pca1$li, pop(microbov))
```

```
## Error in data.frame(dfxy):  object 'pca1' not found
```

```
title("PCA of microbov dataset\naxes 1-2")
```

```
## Error in title("PCA of microbov dataset\naxes 1-2"):  plot.new has not
been called yet
```

```
add.scatter.eig(pca1$eig[1:20], 3,1,2)
```

```
## Error in scatterutil.eigen(w, nf = nf, wsel = c(xax, yax), sub = sub, :
object 'pca1' not found
```

This plane shows that the main structuring is between African an French breeds, the second structure reflecting genetic diversity among African breeds. The third axis reflects the diversity among French breeds:

```
s.class(pca1$li,pop(microbov),xax=1,yax=3,sub="PCA 1-3",csub=2)

## Error in data.frame(dfxy):  object 'pca1' not found

title("PCA of microbov dataset\naxes 1-3")

## Error in title("PCA of microbov dataset\naxes 1-3"):  plot.new has not
been called yet

add.scatter.eig(pca1$eig[1:20],nf=3,xax=1,yax=3)

## Error in scatterutil.eigen(w, nf = nf, wsel = c(xax, yax), sub = sub, :
object 'pca1' not found
```

Overall, all breeds seem well differentiated.

However, we can yet improve these scatterplots, which are fortunately easy to customize. For instance, we can remove the grid, choose different colors for the groups, use larger dots and transparency to better assess the density of points, and remove internal segments of the ellipses:

```
col <- funky(15)
s.class(pca1$li, pop(microbov),xax=1,yax=3, col=transp(col,.6), axesell=FALSE,
        cstar=0, cpoint=3, grid=FALSE)

## Error in data.frame(dfxy):  object 'pca1' not found
```

Let us now assume that we ignore the group memberships. We can still use color in an informative way. For instance, we can recode the principal components represented in the scatterplot on the RGB scale:

```
colorplot(pca1$li, pca1$li, transp=TRUE, cex=3, xlab="PC 1", ylab="PC 2")

## Error in colorplot(pca1$li, pca1$li, transp = TRUE, cex = 3, xlab = "PC
1", :  object 'pca1' not found

title("PCA of microbov dataset\naxes 1-2")

## Error in title("PCA of microbov dataset\naxes 1-2"):  plot.new has not
been called yet

abline(v=0,h=0,col="grey", lty=2)

## Error in int_abline(a = a, b = b, h = h, v = v, untf = untf, ...):
plot.new has not been called yet
```

Colors are based on the first three PCs of the PCA, recoded respectively on the red, green, and blue channel. In this figure, the genetic diversity is represented in two complementary ways: by the distances (further away = more genetically different), and by the colors (more different colors = more genetically different).

We can represent the diversity on the third axis similarly:

```
colorplot(pca1$li[c(1,3)], pca1$li, transp=TRUE, cex=3, xlab="PC 1", ylab="PC 3")

## Error in colorplot(pca1$li[c(1, 3)], pca1$li, transp = TRUE, cex = 3, :
object 'pca1' not found

title("PCA of microbov dataset\naxes 1-3")

## Error in title("PCA of microbov dataset\naxes 1-3"):  plot.new has not
been called yet

abline(v=0,h=0,col="grey", lty=2)

## Error in int_abline(a = a, b = b, h = h, v = v, untf = untf, ...):
plot.new has not been called yet
```

6.3 Performing a Correspondance Analysis on genpop objects

Being contingency tables, the `@tab` slot in `genpop` objects can be submitted to a Correspondance Analysis (CA) to seek a typology of populations. The approach is very similar to the previous one for PCA. Missing data are first replaced during convertion from `genind`, but one could create a `genpop` with NAs and then use `na.replace` to get rid of missing observations.

```
data(microbov)
obj <- genind2genpop(microbov,missing="chi2")

## Error in genind2genpop(microbov, missing = "chi2"):  unused argument
(missing = "chi2")

ca1 <- dudi.coa(as.data.frame(obj$tab),scannf=FALSE,nf=3)

## Error in matrix(unlist(value, recursive = FALSE, use.names = FALSE), nrow
= nr, :  'data' must be of a vector type, was 'NULL'

barplot(ca1$eig,main="Correspondance Analysis eigenvalues",
        col=heat.colors(length(ca1$eig)))

## Error in barplot(ca1$eig, main = "Correspondance Analysis eigenvalues", :
object 'ca1' not found
```

Now we display the resulting typology using a basic scatterplot:

```
s.label(ca1$li,lab=obj$pop.names,sub="CA 1-2",csub=2)

## Error in data.frame(dfxy):  object 'ca1' not found

add.scatter.eig(ca1$eig,nf=3,xax=1,yax=2,posi="bottomright")

## Error in scatterutil.eigen(w, nf = nf, wsel = c(xax, yax), sub = sub, :
object 'ca1' not found
```

```
s.label(ca1$li,xax=1,yax=3,lab=obj$pop.names,sub="CA 1-3",csub=2)

## Error in data.frame(dfxy):  object 'ca1' not found

add.scatter.eig(ca1$eig,nf=3,xax=2,yax=3,posi="topleft")

## Error in scatterutil.eigen(w, nf = nf, wsel = c(xax, yax), sub = sub, :
object 'ca1' not found
```

As in the PCA above, axes are to be interpreted separately in terms of continental differentiation, and between-breeds diversity. Importantly, as in any analysis carried out at a population level, all information about the diversity within populations is lost in this analysis. See the tutorial on DAPC for an individual-based approach which is nontheless optimal in terms of group separation (*dapc*).

7 Spatial analysis

The R software probably offers the largest collection of spatial methods among statistical software. Here, we briefly illustrate two methods commonly used in population genetics. Spatial multivariate analysis is covered in a dedicated tutorial; see *spca* tutorial for more information.

7.1 Isolation by distance

7.1.1 Testing isolation by distance

Isolation by distance (IBD) is tested using Mantel test between a matrix of genetic distances and a matrix of geographic distances. It can be tested using individuals as well as populations. This example uses cat colonies from the city of Nancy. We test the correlation between Edwards' distances and Euclidean geographic distances between colonies.

```
data(nancycats)
toto <- genind2genpop(nancycats,miss="0")

## Error in genind2genpop(nancycats, miss = "0"):  unused argument (miss =
"0")

Dgen <- dist.genpop(toto,method=2)

## Error in dist.genpop(toto, method = 2):  x is not a valid genpop object

Dgeo <- dist(nancycats$other$xy)
ibd <- mantel.randtest(Dgen,Dgeo)

## Error in inherits(m1, "dist"):  object 'Dgen' not found

ibd

## Error in eval(expr, envir, enclos):  object 'ibd' not found
```

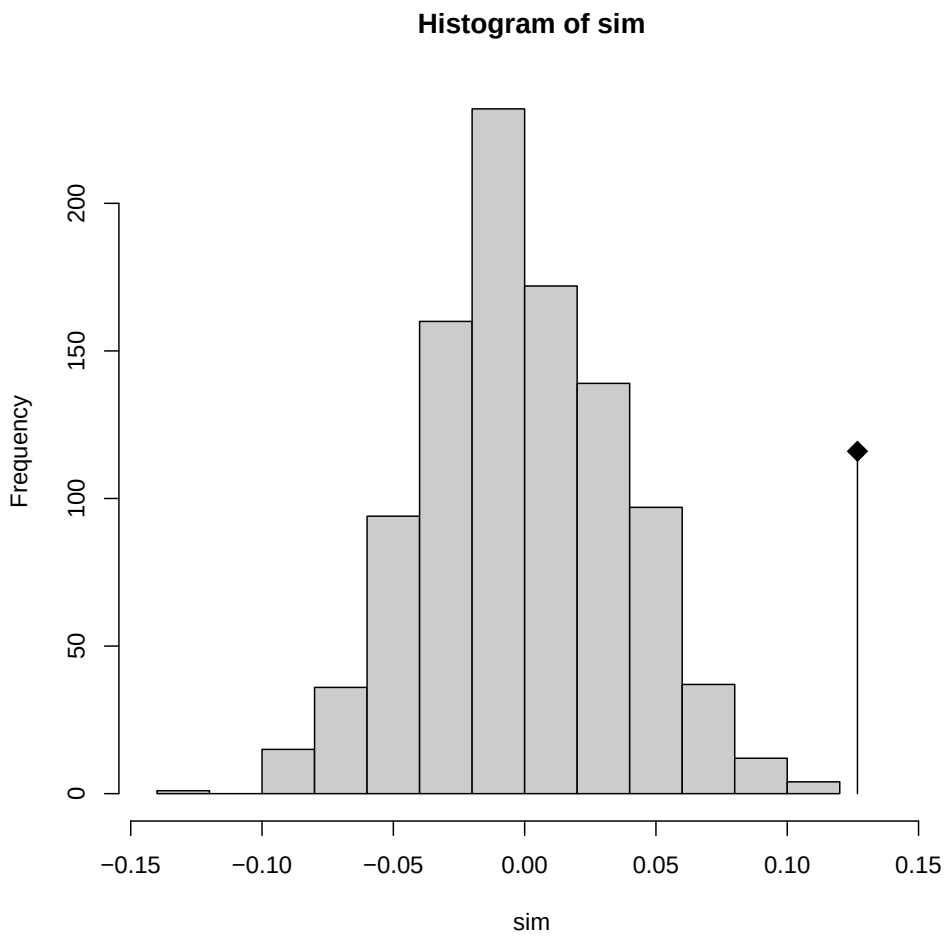
```
plot(ibd)

## Error in plot(ibd):  error in evaluating the argument 'x' in selecting a
method for function 'plot':  Error:  object 'ibd' not found
```

The original value of the correlation between the distance matrices is represented by the dot, while histograms represent permuted values (i.e., under the absence of spatial structure). Significant spatial structure would therefore result in the original value being out of the reference distribution. Here, isolation by distance is clearly not significant.

Let us provide another example using a dataset of individuals simulated under an IBD model:

```
data(spcaIllus)
x <- spcaIllus$dat2B
Dgen <- dist(x$tab)
Dgeo <- dist(other(x)$xy)
ibd <- mantel.randtest(Dgen,Dgeo)
ibd

## Monte-Carlo test
## Call: mantel.randtest(m1 = Dgen, m2 = Dgeo)
##
## Observation: 0.1267341
##
## Based on 999 replicates
## Simulated p-value: 0.001
## Alternative hypothesis: greater
##
##       Std.Obs   Expectation      Variance
##  3.3967497333 -0.0006045586  0.0014053791
```

```
plot(ibd)
```

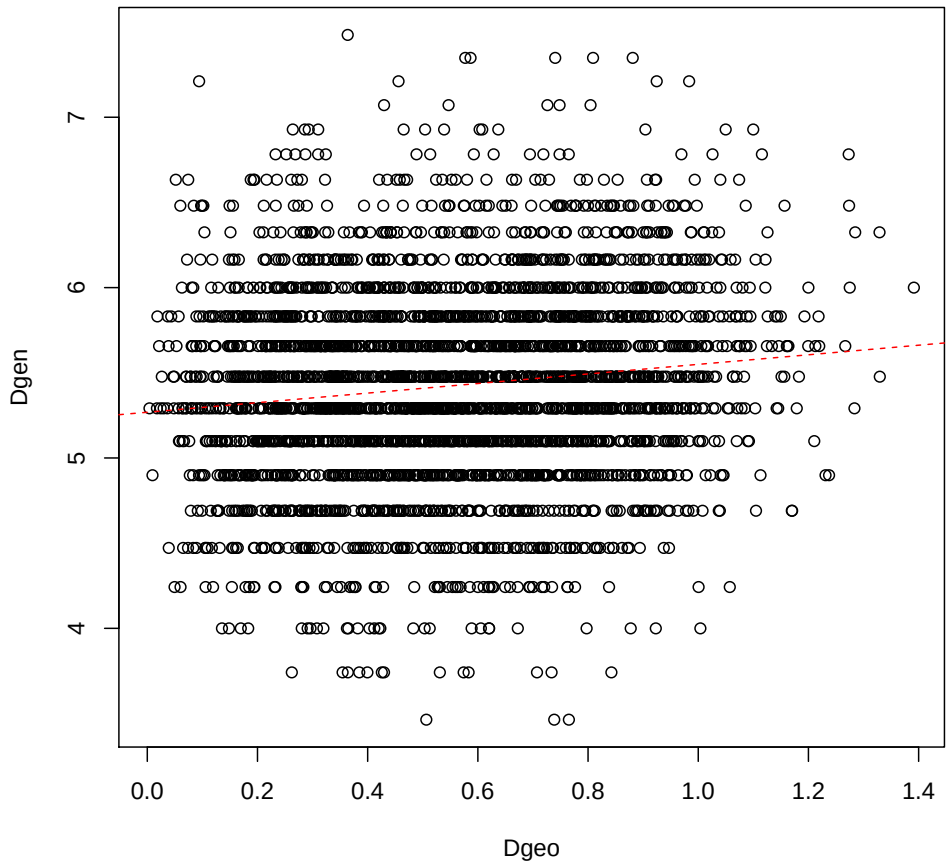


This time there is a clear isolation by distance pattern.

7.1.2 Cline or distant patches?

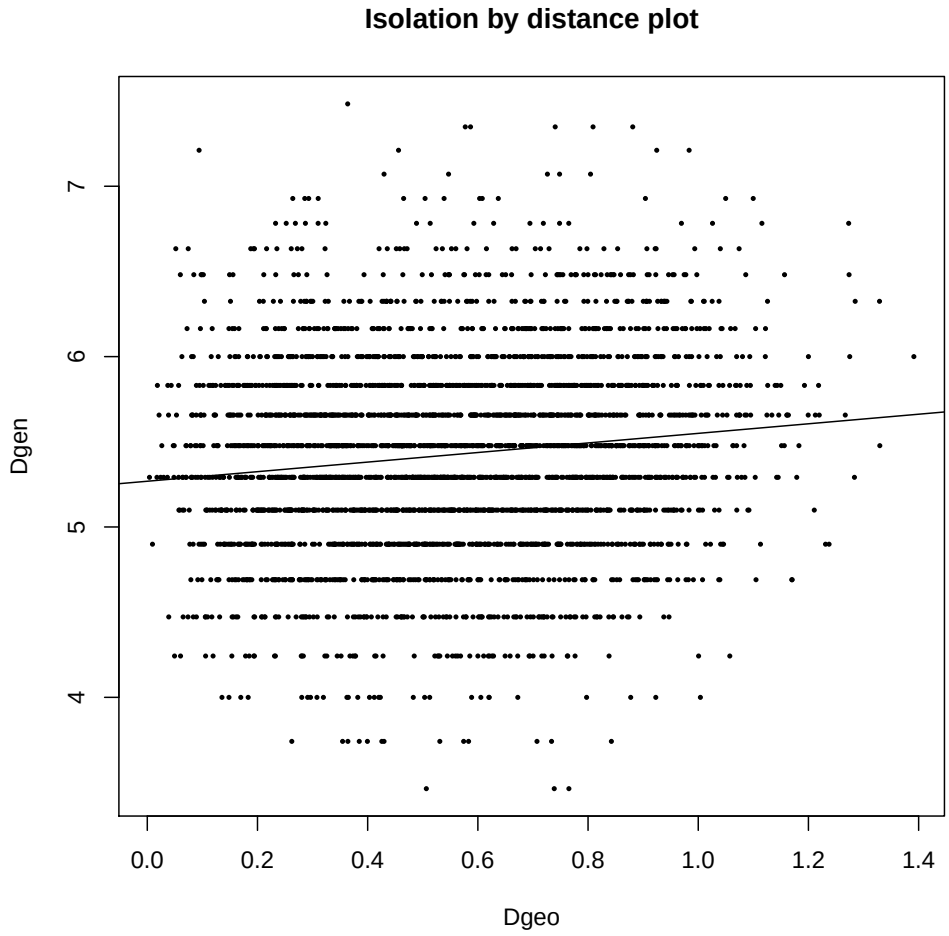
The correlation between genetic and geographic distances can occur under a range of different biological scenarios. Classical IBD would result in continuous clines of genetic differentiation and cause such correlation. However, distant and differentiated populations would also result in such a pattern. These are slightly different processes and we would like to be able to disentangle them. A very simple first approach is simply plotting both distances:

```
plot(Dgeo, Dgen)
abline(lm(Dgen~Dgeo), col="red",lty=2)
```



Most of the time, simple scatterplots fail to provide a good picture of the data as the density of points in the scatterplot is badly displayed. Colors can be used to provide better (and prettier) plots. Local density is measured using a 2-dimensional kernel density estimation (`kde2d`), and the results are displayed using `image`; `colorRampPalette` is used to generate a customized color palette:

```
dens <- kde2d(Dgeo,Dgen, n=300, lims=c(-.1, 1.5,-.5,4))

## Error in eval(expr, envir, enclos):  could not find function "kde2d"

myPal <- colorRampPalette(c("white","blue","gold", "orange", "red"))
plot(Dgeo, Dgen, pch=20,cex=.5)
image(dens, col=transp(myPal(300),.7), add=TRUE)

## Error in image(dens, col = transp(myPal(300), 0.7), add = TRUE): object
'dens' not found

abline(lm(Dgen~Dgeo))
title("Isolation by distance plot")
```



The scatterplot clearly shows one single consistent cloud of point, without discontinuities which would have indicated patches. This is reassuring, since the data were actually simulated under an IBD (continuous) model.

7.2 Using Monmonier's algorithm to define genetic boundaries

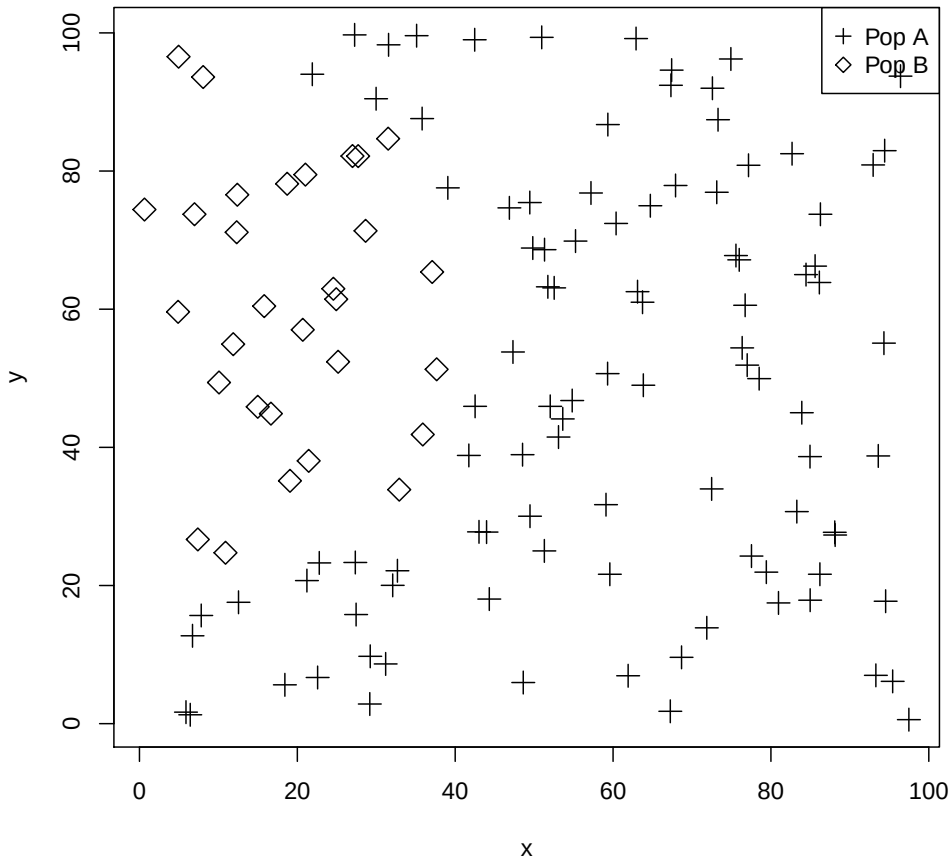
Monmonier's algorithm [11] was originally designed to find boundaries of maximum differences between contiguous polygons of a tesselation. As such, the method was basically used in geographical analysis. More recently, [12] suggested that this algorithm could be employed to detect genetic boundaries among georeferecend genotypes (or populations). This algorithm is implemented using a more general approach than the initial one in *adegenet*.

Instead of using Voronoi tesselation as in the original version, the functions `monmonier` and `optimize.monmonier` can handle various neighbouring graphs such as Delaunay triangulation, Gabriel's graph, Relative Neighbours graph, etc. These graphs define spatial connectivity among locations (of genotypes or populations), with couple of locations being neighbours (if connected) or not. Another information is given by a set of markers which define genetic distances among these 'points'. The aim of Monmonier's algorithm is to find the path through the strongest genetic distances between neighbours. A more complete description of the principle of this algorithm will be found in the documentation of

`monmonier`. Indeed, the very purpose of this tutorial is simply to show how it can be used on genetic data.

Let's take the example from the function's manpage and detail it. The dataset used is `sim2pop`.

```
data(sim2pop)
sim2pop

##  /// genind object \\\
## @call: old2new(object = sim2pop)
##
## @tab:  130 x 241 matrix of genotypes
##
## @ind.names: vector of  130 individual names
## @loc.names: vector of  20 locus names
## @loc.nall: number of alleles per locus
## @loc.fac: locus factor for the  241 columns of @tab
## @all.names: list of  20 components yielding allele names for each locus
## @ploidy:  2 2 2 2 2 2
## @type:  codom
##
## Optional contents:
## @strata: - empty -
## @hierarchy:  - empty -
## @pop:  factor giving the population of each individual
## @pop.names:  character giving the name of each population
##
## @other: a list containing: xy

summary(sim2pop$pop)

## P01 P02
## 100  30

temp <- sim2pop$pop
levels(temp) <- c(3,5)
temp <- as.numeric(as.character(temp))
plot(sim2pop$other$xy,pch=temp,cex=1.5,xlab='x',ylab='y')
legend("topright",leg=c("Pop A", "Pop B"),pch=c(3,5))
```



There are two sampled populations in this dataset, with inequal sample sizes (100 and 30). Twenty microsatellite-like loci are available for all genotypes (no missing data). `monmonier` requires several arguments to be specified:

```
args(monmonier)

## function (xy, dist, cn, threshold = NULL, bd.length = NULL, nrun = 1,
##     skip.local.diff = rep(0, nrun), scanthres = is.null(threshold),
##     allowLoop = TRUE)
## NULL
```

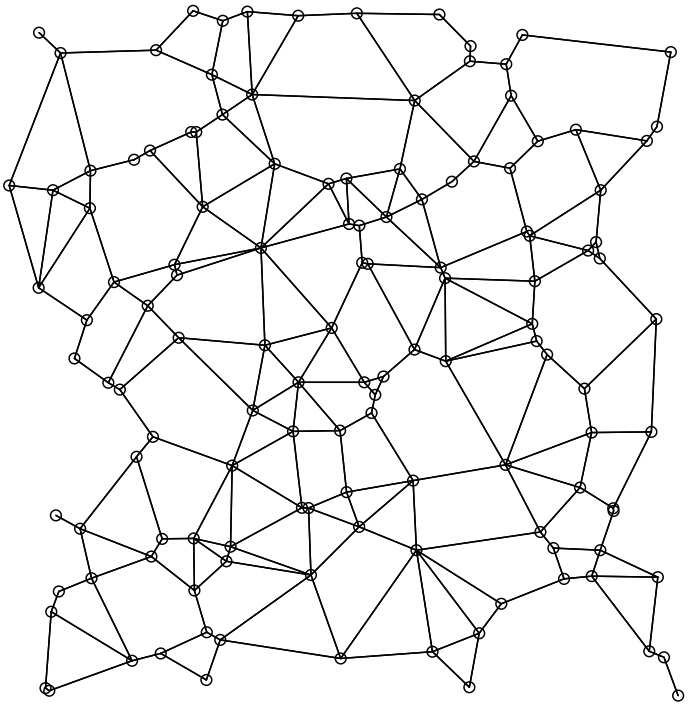
The first argument (`xy`) is a matrix of geographic coordinates, already stored in `sim2pop`. Next argument is an object of class `dist`, which is the matrix of pairwise genetic distances. For now, we will use the classical Euclidean distance between allelic profiles of the individuals. This is obtained by:

```
D <- dist(sim2pop$tab)
```

The next argument (`cn`) is a connection network. Routines for building such networks are scattered over several packages, but all made available through the function `chooseCN`. Here,

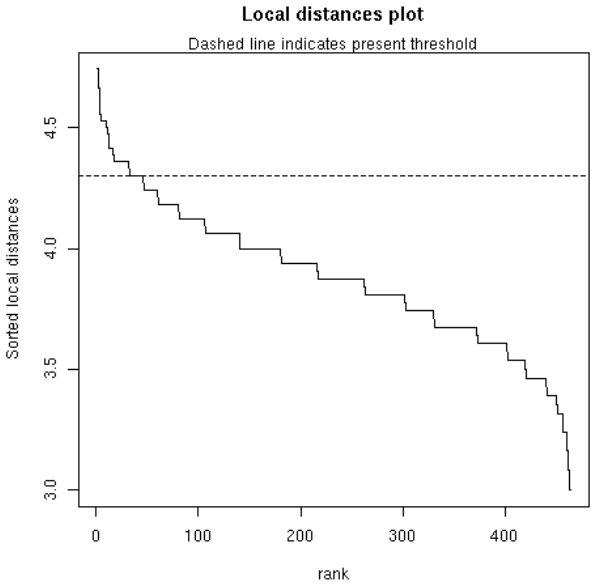
we disable the interactivity of the function (`ask=FALSE`) and select the second type of graph which is the graph of Gabriel (`type=2`).

```
gab <- chooseCN(sim2pop$other$xy,ask=FALSE,type=2)
```



The obtained network is automatically plotted by the function. It seems we are now ready to proceed to the algorithm.

```
mon1 <- monmonier(sim2pop$other$xy,D,gab)
```



This plot shows all local differences sorted in decreasing order. The idea behind this is that a significant boundary would cause local differences to decrease abruptly after the boundary. This should be used to choose the *threshold* difference for the algorithm to stop extending the boundary. Here, there is no indication af an actual boundary.

Why do the algorithm fail to find a boundary? Either because there is no genetic differentiation to be found, or because the signal differentiating both populations is too weak to overcome the random noise in genetic distances. What is the F_{st} between the two samples?

```
pairwise.fst(sim2pop)

## This function has been moved to the package hierfstat.  You can still
use it in adegenet, but this version is deprecated.

##            1
## 2 0.02960988
```

This value would be considered as very weak differentiation ($F_{ST} = 0.023$). Is it significant? We can easily ellaborate a permutation test of this F_{ST} value; to save computational time, we use only a small number of replicates to generate F_{ST} values in absence of population structure:

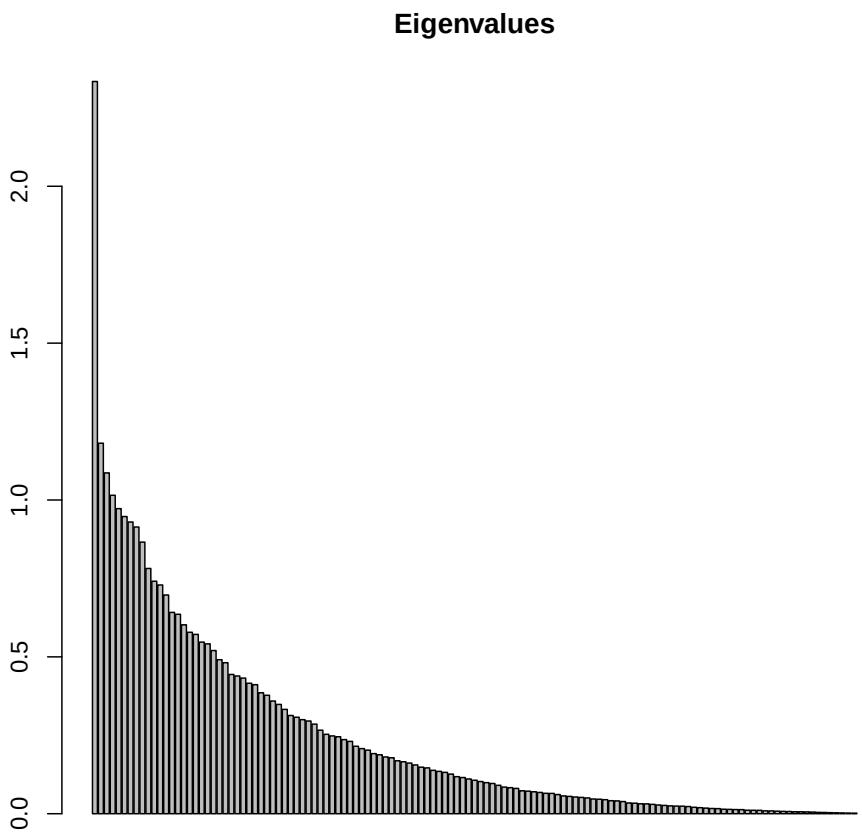
```
replicate(10, pairwise.fst(sim2pop, pop=sample(pop(sim2pop))))

## This function has been moved to the package hierfstat.  You can still
use it in adegenet, but this version is deprecated.
## This function has been moved to the package hierfstat.  You can still
```

```
use it in adegenet, but this version is deprecated.
## This function has been moved to the package hierfstat.  You can still
use it in adegenet, but this version is deprecated.
## This function has been moved to the package hierfstat.  You can still
use it in adegenet, but this version is deprecated.
## This function has been moved to the package hierfstat.  You can still
use it in adegenet, but this version is deprecated.
## This function has been moved to the package hierfstat.  You can still
use it in adegenet, but this version is deprecated.
## This function has been moved to the package hierfstat.  You can still
use it in adegenet, but this version is deprecated.
## This function has been moved to the package hierfstat.  You can still
use it in adegenet, but this version is deprecated.
## This function has been moved to the package hierfstat.  You can still
use it in adegenet, but this version is deprecated.
```

F_{ST} values in absence of population structure would be one order of magnitude lower (more replicate would give a very low p-value — just replace `10` by `200` in the above command). In fact, the two samples are indeed genetically differentiated.

Can Monmonier's algorithm find a boundary between the two populations? Yes, if we get rid of the random noise. This can be achieved using a simple ordination method such as Principal Coordinates Analysis.

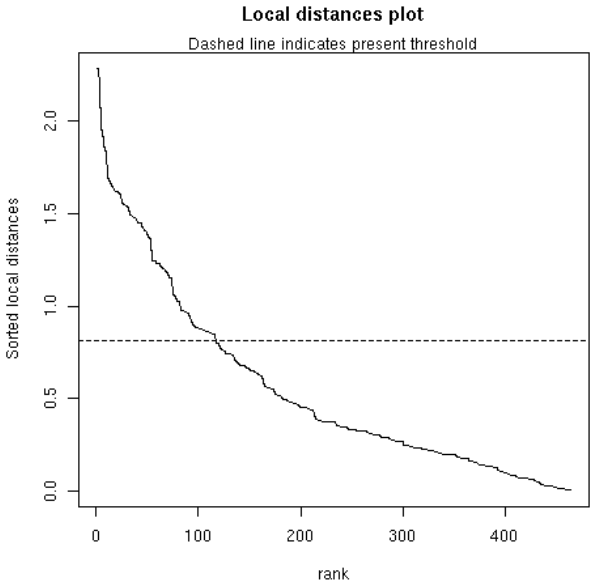
```
pco1 <- dudi.pco(D,scannf=FALSE,nf=1)
barplot(pco1$eig,main="Eigenvalues")
```



We retain only the first eigenvalue. The corresponding coordinates are used to redefine the genetic distances among genotypes. The algorithm is then re-run.

```
D <- dist(pco1$li)
```

```
mon1 <- monmonier(sim2pop$other$xy,D,gab)
```




```
##
## ################################################################
## # List of paths of maximum differences between neighbours #
## #           Using a Monmonier based algorithm             #
## ################################################################
##
## $call:monmonier(xy = sim2pop$other$xy, dist = D, cn = gab, scanthres = FALSE)
##
##       # Object content #
## Class:  monmonier
## $nrun (number of successive runs):  1
## $run1: run of the algorithm
## $threshold (minimum difference between neighbours):  1.631
## $xy: spatial coordinates
## $cn: connection network
##
##       # Runs content #
## # Run 1
## # First direction
## Class:  list
## $path:
##                x        y
## Point_1 14.98299 93.81162
##
## $values:
##  4.563555
## # Second direction
```

```
## Class:  list
## $path:
##                 x        y
## Point_1 14.98299 93.81162
## Point_2 30.74508 87.57724
## Point_3 33.66093 86.14115
## ...
##
## $values:
##  4.563555 3.23581 3.906439 ...
```

This may take some time... but never more than five minutes on an 'ordinary' personnal computer. The object `mon1` contains the whole information about the boundaries found. As several boundaries can be seeked at the same time (argument `nrun`), you have to specify about which run and which direction you want to get informations (values of differences or path coordinates). For instance:

```
names(mon1)

## [1] "run1"      "nrun"      "threshold" "xy"        "cn"        "call"

names(mon1$run1)

## [1] "dir1" "dir2"

mon1$run1$dir1

## $path
##                 x        y
## Point_1 14.98299 93.81162
##
## $values
## [1] 4.563555
```

It can also be useful to identify which points are crossed by the barrier; this can be done using `coords.monmonier`:

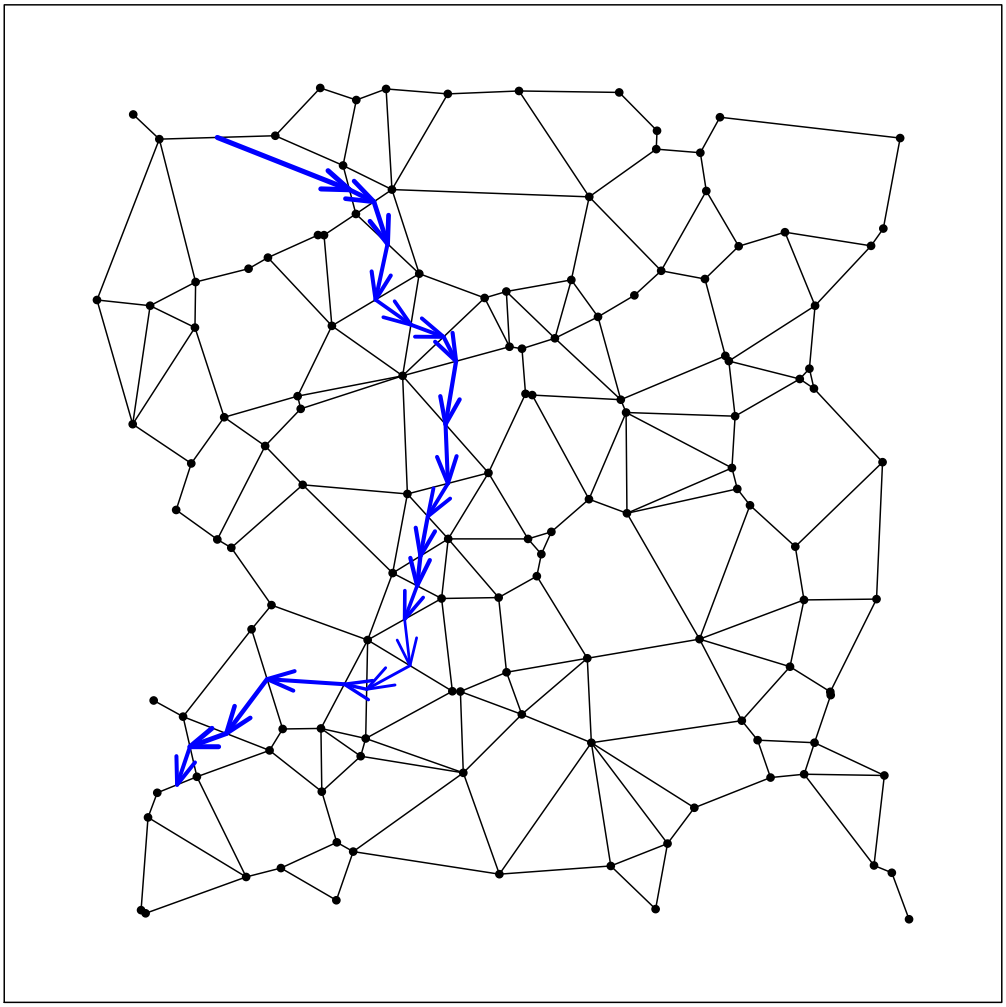
```
coords.monmonier(mon1)

## $run1
## $run1$dir1
##             x.hw     y.hw first second
## Point_1 14.98299 93.81162    11    125
##
## $run1$dir2
```

```
##              x.hw     y.hw first second
## Point_1  14.98299 93.81162    11    125
## Point_2  30.74508 87.57724    44    128
## Point_3  33.66093 86.14115    20    128
## Point_4  35.28914 81.12578    68    128
## Point_5  33.85756 74.45492    68    117
## Point_6  38.07622 71.47532    68    122
## Point_7  41.97494 70.02783    35    122
## Point_8  43.45812 67.12026    69    122
## Point_9  42.20206 59.59613    22    122
## Point_10 42.48613 52.55145    22    124
## Point_11 40.08702 48.61795    13    124
## Point_12 39.20791 43.89978    13    127
## Point_13 38.81236 40.34516    62    127
## Point_14 37.32112 36.35265    62    130
## Point_15 37.96426 30.82105    94    130
## Point_16 32.79703 28.00517    16    130
## Point_17 30.12832 28.60376    85    130
## Point_18 20.92496 29.21211    63    119
## Point_19 16.05811 22.72600    61    126
## Point_20 11.72524 21.15519    89    126
## Point_21 10.18696 16.61536    74     89
```

The returned dataframe contains, in this order, the x and y coordinates of the points of the barrier, and the identifiers of the two 'parent' points, that is, the points whose barycenter is the point of the barrier.

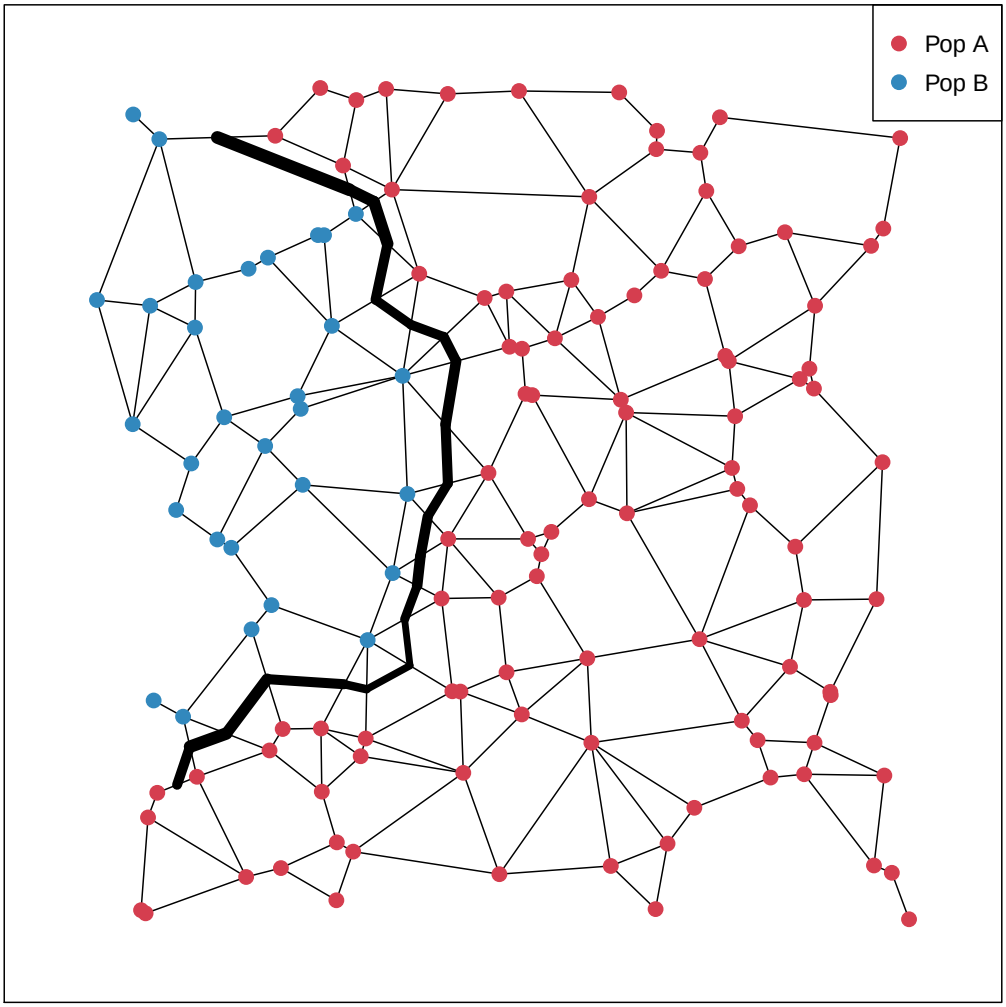
Finally, you can plot very simply the obtained boundary using the method `plot`:

```
plot(mon1)
```



see arguments in `?plot.monmonier` to customize this representation. Last, we can compare the infered boundary with the actual distribution of populations:

```
plot(mon1,add.arrows=FALSE,bwd=10,col="black")
points(sim2pop$other$xy, cex=2, pch=20,
       col=fac2col(pop(sim2pop), col.pal=spectral))
legend("topright",leg=c("Pop A", "Pop B"), pch=c(20),
       col=spectral(2), pt.cex=2)
```



Not too bad...

8 Simulating hybridization

The function `hybridize` allows to simulate hybridization between individuals from two distinct genetic pools, or more broadly between two `genind` objects. Here, we use the example from the manpage of the function, to go a little further. Please have a look at the documentation, especially at the different possible outputs (outputs for the software STRUCTURE is notably available).

```
temp <- seppop(microbov)
names(temp)
```

```
##  [1] "Borgou"          "Zebu"            "Lagunaire"
##  [4] "NDama"           "Somba"           "Aubrac"
##  [7] "Bazadais"        "BlondeAquitaine" "BretPieNoire"
## [10] "Charolais"       "Gascon"          "Limousin"
## [13] "MaineAnjou"      "Montbeliard"     "Salers"
```

```
salers <- temp$Salers
zebu <- temp$Zebu
zebler <- hybridize(salers, zebu, n=40, pop="zebler")
```

A first generation (F1) of hybrids 'zebler' is obtained. Is it possible to perform a backcross, say, with 'salers' population? Yes, here it is:

```
F2 <- hybridize(salers, zebler, n=40)

## Error in hybridize(salers, zebler, n = 40):  names of markers in x1 and
x2 do not correspond

F3 <- hybridize(salers, F2, n=40)

## Error in is(x, "genind"):  object 'F2' not found

F4 <- hybridize(salers, F3, n=40)

## Error in is(x, "genind"):  object 'F3' not found
```

Finally, note that despite this example shows hybridization between diploid organisms, `hybridize` is not retrained to this case. In fact, organisms with any even level of ploidy can be used, in which case half of the genes is taken from each reference population. Ultimately, more complex mating schemes could be implemented... suggestion or (better) contributions are welcome!

References

[1] Jombart, T. (2008) adegenet: a R package for the multivariate analysis of genetic markers. *Bioinformatics* 24: 1403-1405.

[2] R Development Core Team (2014). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

[3] Dray S and Dufour A-B (2007) The ade4 package: implementing the duality diagram for ecologists. *Journal of Statistical Software* 22: 1-20.

[4] Jombart T, Devillard S and Balloux, F (2010). Discriminant analysis of principal components: a new method for the analysis of genetically structured populations. *BMC Genetics* 11: 94.

[5] Jombart T, Devillard S, Dufour A-B and Pontier D (2008) Revealing cryptic spatial patterns in genetic variability by a new multivariate method. *Heredity* 101: 92-103.

[6] Jombart T, Eggo RM, Dodd PJ and Balloux F (2010) Reconstructing disease outbreaks from genetic data: a graph approach. *Heredity* 106: 383-390.

[7] Jombart T, Pontier D and Dufour A-B (2009) Genetic markers in the playground of multivariate analysis. *Heredity* 102: 330-341.

[8] Paradis E, Claude J, and Strimmer K (2004) APE: analyses of phylogenetics and evolution in R language. *Bioinformatics*: 20, 289-290.

[9] Charif D, and Lobry J (2007) SeqinR 1.0-2: a contributed package to the R project for statistical computing devoted to biological sequences retrieval and analysis. *in* Structural approaches to sequence evolution: Molecules, networks, populations, *Springer Verlag*, 207-232.

[10] Nei M (1973) Analysis of gene diversity in subdivided populations. *Proc Natl Acad Sci USA* 70: 3321-3323.

[11] Monmonier M (1973) Maximum-difference barriers: an alternative numerical regionalization method. *Geographical Analysis* 3: 245-261.

[12] Manni F, Guerard E and Heyer E (2004) Geographic patterns of (genetic, morphologic, linguistic) variation: how barriers can be detected by "Monmonier's algorithm". *Human Biology* 76: 173-190.