Tutorial using the R software

A tutorial for the R package `adegenet_1.2-7`

T. JOMBART

Looking for information?

More information is to be found from adegenet website: `http://adegenet.r-forge.r-project.org/`. Questions can be asked on the *adegenet forum* (`adegenet-forum@lists.r-forge.r-project.org`), a public mailing list whose archives are browsable and searchable. Please don't hesitate to use it! You will find more information about this forum in the section 'contact' of the adegenet website.

Comments and contributions on this tutorial are very welcome; please email me directly at: `t.jombart@imperial.ac.uk`.

Contents

1 Introduction

This tutorial proposes a short visit through functionalities of the adegenet package for R (Ihaka & Gentleman, 1996; R Development Core Team, 2009). The purpose of this package is to facilitate the multivariate analysis of molecular marker data, especially using the ade4 package (Chessel *et al.*, 2004). Data can be imported from a wide range of formats, including those of popular software (GENETIX, STRUCTURE, Fstat, Genepop), or from simple data frame of genotypes. adegenet also aims at providing a platform from which to use easily methods provided by other R packages (e.g., Goudet, 2005). Indeed, if it is possible to perform various genetic data analyses using R, data formats often differ from one package to another, and conversions are sometimes far from easy and straightforward.

In this tutorial, I first present the two object classes used in adegenet, namely genind (genotypes of individuals) and genpop (genotypes grouped by populations). Then, several topics will be tackled using reproductible examples.

2 First steps

2.1 Installing the package

Current version of the package is 1.2-3, and is compatible with R 2.8.1. Please make sure to be using at least R 2.8.1 and adegenet 1.2-3 before sending question about missing functions.

Here the adegenet package is installed along with other recommended packages.

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

Then the first step is to load the package:

```
> library(adegenet)
```

2.2 Object classes

Two classes of objects are defined, depending on the level at which the genetic information is stored: 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.

2.2.1 genind objects

These objects can be obtained by reading data files from other software, from a data.frame of genotypes, by conversion from a table of allelic frequencies, or even from aligned DNA sequences (see 'importing data').

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

```
[1] TRUE
```

```
> nancycats
```

```
   #####################
   ### Genind object ###
   #####################
- genotypes of individuals -

S4 class:  genind
@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
@type:  codom

Optionnal contents: 
@pop:  factor giving the population of each individual
@pop.names:  factor giving the population of each individual

@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 '$' was also implemented for adegenet objects, so that slots can be accessed as if they were components of a list. The main slot in genind is a table of allelic frequencies of individuals (in rows) for every alleles in every loci. Being frequencies, data sum to one per locus, giving the score of 1 for an homozygote and 0.5 for an heterozygote. The particular case of presence/absence data will is described in an ad-hoc section (see 'Handling presence/absence data'). For instance:

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

```
    L1.01 L1.02 L1.03 L1.04 L1.05 L1.06 L1.07 L1.08 L1.09 L1.10
010     0     0     0     0     0   0.0   0.0   0.0   1.0   0.0
011     0     0     0     0     0   0.0   0.0   0.0   0.0   0.5
012     0     0     0     0     0   0.5   0.0   0.5   0.0   0.0
013     0     0     0     0     0   0.5   0.0   0.5   0.0   0.0
014     0     0     0     0     0   0.0   0.0   1.0   0.0   0.0
015     0     0     0     0     0   0.0   0.5   0.0   0.5   0.0
016     0     0     0     0     0   0.5   0.0   0.0   0.5   0.0
017     0     0     0     0     0   0.5   0.0   0.5   0.0   0.0
018     0     0     0     0     0   0.5   0.0   0.0   0.5   0.0
```

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. Alternatively, one can use the accessor `locNames`:

```
> locNames(nancycats)
```

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

```
> head(locNames(nancycats, withAlleles = TRUE), 10)
```

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

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 components are also allowed. The slot @other is a list that can include any additionnal information. The optional slot @pop (a factor giving a grouping of individuals) is particular in that the behaviour of many functions will check automatically for it 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:

```
> table(nancycats$pop)
```

```
P01 P02 P03 P04 P05 P06 P07 P08 P09 P10 P11 P12 P13 P14 P15 P16 P17
 10  22  12  23  15  11  14  10   9  11  20  14  13  17  11  12  13
```

```
> catpop <- genind2genpop(nancycats)
```

```
 Converting data from a genind to a genpop object...

...done.
```

```
> catpop
```

```
       #####################
       ### Genpop object ###
       #####################
- Alleles counts for populations -

S4 class:  genpop
@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) but will not be passed during any conversion (catpop has no $other$xy).
Note that the slot 'pop' can be retrieved and set using the pop function:

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

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

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

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

Finally, a genind object generally contains its matched call, *i.e.* the instruction that created it. This is not the case, however, for objects loaded using data. 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
```

2.2.2 genpop objects

We use the previously built genpop object:

```
> catpop
```

```
       ######################
       ### Genpop object ###
       ######################
- Alleles counts for populations -

S4 class:  genpop
@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]
```

```
   L1.01 L1.02 L1.03 L1.04 L1.05 L1.06 L1.07 L1.08 L1.09 L1.10
01     0     0     0     0     0     0     0     2     9     1
02     0     0     0     0     0    10     9     8    14     2
03     0     0     0     4     0     0     0     0     1    10
04     0     0     0     3     0     0     0     1     7    17
05     0     0     0     1     0     0     0     0     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.

3 Various topics

3.1 Importing data

3.1.1 From GENETIX, STRUCTURE, FSTAT, Genepop

Data can be read from the software GENETIX (.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 character) 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 2: 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).

3.1.2 From other software

Genetic markers data can most of the time be stored as a table 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. As a consequence, SNP data should consist of the raw nucleotides. 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 any 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, I provide an example using a data set from the library hierfstat.

```
> library(hierfstat)
> toto <- read.fstat.data(paste(.path.package("hierfstat"), "/data/diploid.dat",
+     sep = "", collapse = ""), nloc = 5)
> head(toto)
```

```
  Pop loc-1 loc-2 loc-3 loc-4 loc-5
1   1    44    43    43    33    44
2   1    44    44    43    33    44
3   1    44    44    43    43    44
4   1    44    44    NA    33    44
5   1    44    44    24    34    44
6   1    44    44    NA    43    44
```

toto is a data frame containing genotypes and a population factor.

```
> obj <- df2genind(X = toto[, -1], pop = toto[, 1])
> obj
```

```
   #####################
   ### Genind object ###
   #####################
- genotypes of individuals -

S4 class:  genind
@call: df2genind(X = toto[, -1], pop = toto[, 1])

@tab:  44 x 11 matrix of genotypes

@ind.names: vector of  44 individual names
@loc.names: vector of  5 locus names
@loc.nall: number of alleles per locus
@loc.fac: locus factor for the  11 columns of @tab
@all.names: list of  5 components yielding allele names for each locus
@ploidy:  2
@type:  codom

Optionnal contents:
@pop:  factor giving the population of each individual
@pop.names:  factor giving the population of each individual

@other: - empty -
```

obj is a genind containing the same information, but recoded as a matrix of allele frequencies ($tab slot).

3.1.3 SNPs data

In adegenet, SNP data are handled as other codominant markers such as microsatellites. 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 "g" "t" "c" "g" "t"
genot. 2 "g" "a" "t" "c" "a"
genot. 3 "t" "g" "c" "a" "g"
```

```
> obj <- df2genind(dat, ploidy = 1)
> truenames(obj)
```

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

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

3.1.4 DNA sequences

DNA sequences can be read into R using the ape package (Paradis *et al.*, 2004; Paradis, 2006), and imported into adegenet using DNAbin2genind. There are several ways ape can be used to read in DNA sequences. The easiest one is reading data from a usual format such as FASTA or Clustal using read.dna. 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"


> summary(myDNA)


       Length Class  Mode
U15717 1045   DNAbin raw
U15718 1045   DNAbin raw
U15719 1045   DNAbin raw
U15720 1045   DNAbin raw
U15721 1045   DNAbin raw
U15722 1045   DNAbin raw
U15723 1045   DNAbin raw
U15724 1045   DNAbin raw
```

In adegenet, only polymorphic loci are conserved; importing data from a DNA sequence to adegenet therefore consist in extracting SNPs from the aligned sequences. 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 ###
   #####################
- genotypes of individuals -

S4 class:  genind
@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
@type:  codom

Optionnal contents:
@pop:  - empty -
@pop.names:  - empty -

@other: - empty -
```

Here, out of the 1045 nucleotides of the sequences, 318 SNPs where extracted and stored as a genind object.

3.1.5 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 (Charif & Lobry, 2007). 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] "-kifercelartlkklgldgykgvslanwvclakwesgynteatnynpgdestdygifqinsrywcnngkpgavdachiscsallqnniada

$seq[[2]]
[1] "-kifercelartlkrlgldgyrgislanwvclakwesdyntqatnynpgdqstdygifqinshywcndgkpgavnachiscnallqdnitda

$seq[[3]]
[1] "-kvfercelartlkrlgmdgyrgislanwmclakwesgyntratnynagdrstdygifqinsrywcndgkpgavnachlscsallqdniada

$seq[[4]]
[1] "-ktyercefartlkrngmsgyygvsladwvclaqhesnyntqarnydpgdqstdygifqinsrywcndgkpraknacgipcsallqdditqa
```

```
$seq[[5]]
[1] "-kvfercelartlkklgldgykgvslanwlcltkwessyntkatnynpssestdygifqinskwwcndgkpnavdgchvscselmendiaka

$seq[[6]]
[1] "-kvfskcelahklkaqemdgfggyslanwvcmaeyesnfntrafngknangssdyglfqlnnkwwckdnkrsssnacnimcsklldeniddd


$com
[1] ";empty description\n" ";\n"                   ";\n"
[4] ";\n"                   ";\n"                   ";\n"

attr(,"class")
[1] "alignment"
```

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

```
   #####################
   ### Genind object ###
   #####################
- genotypes of individuals -

S4 class:  genind
@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
@type:  codom

Optionnal contents:
@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 for 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:

```
> locNames(x)
```

```
  L01   L02   L03   L04   L05   L06   L07   L08   L09   L10   L11   L12   L13
  "3"   "4"   "5"   "6"   "9"  "11"  "12"  "15"  "16"  "17"  "18"  "19"  "21"
  L14   L15   L16   L17   L18   L19   L20   L21   L22   L23   L24   L25   L26
 "22"  "24"  "28"  "30"  "32"  "33"  "34"  "35"  "38"  "39"  "42"  "44"  "46"
  L27   L28   L29   L30   L31   L32   L33   L34   L35   L36   L37   L38   L39
 "47"  "48"  "49"  "50"  "51"  "53"  "57"  "60"  "62"  "63"  "64"  "67"  "68"
  L40   L41   L42   L43   L44   L45   L46   L47   L48   L49   L50   L51   L52
 "69"  "71"  "72"  "73"  "74"  "75"  "76"  "78"  "79"  "80"  "82"  "83"  "85"
  L53   L54   L55   L56   L57   L58   L59   L60   L61   L62   L63   L64   L65
 "86"  "87"  "88"  "90"  "91"  "92"  "93"  "94"  "98"  "99" "101" "102" "103"
  L66   L67   L68   L69   L70   L71   L72   L73   L74   L75   L76   L77   L78
"105" "106" "109" "112" "113" "114" "116" "117" "118" "120" "121" "122" "124"
  L79   L80   L81   L82
"125" "126" "128" "129"
```

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
```

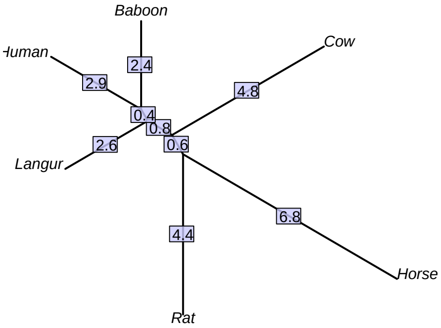
Now that polymorphic sites have been converted into a genind object, simple distances can be computed between the sequences. Note that adegenet does not implement specific distances for protein sequences, we only use the simple Euclidean distance. Fancier protein distances are implemented in R; see for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

```
> D <- dist(truenames(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
```

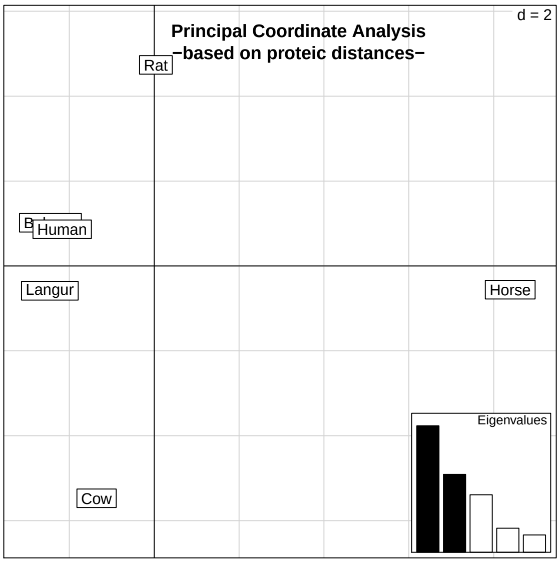
This matrix of distances is small enough for one to interprete the raw numbers. However, it is also very straightforward to represent these distances as a tree or in a reduced space. We first build a Neighbor-Joining tree using the *ape* package:

```
> library(ape)
> tre <- nj(D)
> plot(tre, type = "unrooted", edge.w = 2)
> edgelabels(tex = round(tre$edge.length, 1), bg = rgb(0.8, 0.8,
+     1, 0.8))
```



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-")
```



3.1.6 Using genind/genpop constructors

Lastly, 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, by leading to meaningless subsequent analysis.

Therefore, one should 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:

```
> library(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)
> toto


       #####################
       ### Genpop object ###
       #####################
- Alleles counts for populations -

S4 class:  genpop
@call: genpop(tab = microsatt$tab)

@tab:  18 x 112 matrix of alleles counts

@pop.names: vector of  18 population names
@loc.names: vector of  9 locus names
@loc.nall: number of alleles per locus
@loc.fac: locus factor for the  112 columns of @tab
@all.names: list of  9 components yielding allele names for each locus
@ploidy:  2
@type:  codom

@other: - empty -


> summary(toto)


 # Number of populations:  18

 # Number of alleles per locus:
L1 L2 L3 L4 L5 L6 L7 L8 L9
 8 15 11 10 17 10 14 15 12

 # Number of alleles per population:
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18
39 69 51 59 52 41 34 48 46 47 43 56 57 52 49 64 56 67

 # Percentage of missing data:
[1] 0
```

3.2 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. Here are examples of how to use these functions:

```
> obj <- genind2genotype(nancycats)
> class(obj)

[1] "data.frame"

> obj[1:4, 1:5]

        fca8   fca23   fca43   fca45   fca77
N215    <NA> 136/146 139/139 120/116 156/156
N216    <NA> 146/146 139/145 126/120 156/156
N217 135/143 136/146 141/141 116/116 156/152
N218 135/133 138/138 139/141 126/116 150/150

> class(obj$fca8)

[1] "genotype" "factor"

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

[1] "data.frame"

> obj[1:4, 1:5]

     pop   fca8  fca23  fca43  fca45
N215   1     NA 136146 139139 116120
N216   1     NA 146146 139145 120126
N217   1 135143 136146 141141 116116
N218   1 133135 138138 139141 116126
```

Now we can use the function varcomp.glob from *hierfstat* to compute 'variance' components:

```
> varcomp.glob(obj$pop, obj[, -1])

$loc
            [,1]         [,2]      [,3]
fca8  0.08867161  0.116693199 0.6682028
fca23 0.05384247  0.077539920 0.6666667
fca43 0.05518935  0.066055996 0.6793249
fca45 0.05861271 -0.001026783 0.7083333
fca77 0.08810966  0.156863586 0.6329114
fca78 0.04869695  0.079006911 0.5654008
fca90 0.07540329  0.097194716 0.6497890
fca96 0.07538325 -0.005902071 0.7543860
fca37 0.04264094  0.116318729 0.4514768

$overall
      Pop       Ind     Error
0.5865502 0.7027442 5.7764917

$F
             Pop       Ind
Total 0.08301274 0.1824701
Pop   0.00000000 0.1084610
```

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
N215   1   <NA> 136146 139139 116120
N216   1   <NA> 146146 139145 120126
N217   1 135143 136146 141141 116116
N218   1 133135 138138 139141 116126
N219   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
N215   1    <NA> 136|146 139|139 116|120
N216   1    <NA> 146|146 139|145 120|126
N217   1 135|143 136|146 141|141 116|116
N218   1 133|135 138|138 139|141 116|126
N219   1 133|135 140|146 141|145 126|126
```

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

3.3 Manipulating data

Data manipulation is meant to be easy in adegenet (if it is not, complain!). 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:

```
> titi <- toto[1:3, ]
> toto$pop.names
```

```
          01           02           03           04           05           06
    "Baoule"     "Borgou"        "BPN"  "Charolais"   "Holstein"     "Jersey"
          07           08           09           10           11           12
 "Lagunaire"   "Limousin" "MaineAnjou"  "Mtbeliard"      "NDama"    "Normand"
          13           14           15           16           17           18
"Parthenais"      "Somba"  "Vosgienne"      "ZChoa"   "ZMbororo"      "Zpeul"
```

```
> titi
```

```
       #####################
       ### Genpop object ###
       #####################
- Alleles counts for populations -

S4 class:  genpop
@call: .local(x = x, i = i, j = j, drop = drop)

@tab:  3 x 112 matrix of alleles counts

@pop.names: vector of  3 population names
@loc.names: vector of  9 locus names
@loc.nall: number of alleles per locus
@loc.fac: locus factor for the  112 columns of @tab
@all.names: list of  9 components yielding allele names for each locus
@ploidy:  2
@type:  codom

@other: a list containing: elements without names


> titi$pop.names


       1        2        3
"Baoule" "Borgou"    "BPN"
```

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. For instance, we can subset titi to keep only the third marker:

```
> titi <- titi[, titi$loc.fac == "L3"]
> titi

       #####################
       ### Genpop object ###
       #####################
- Alleles counts for populations -

S4 class:  genpop
@call: .local(x = x, i = i, j = j, drop = drop)

@tab:  3 x 11 matrix of alleles counts

@pop.names: vector of  3 population names
@loc.names: vector of  1 locus names
@loc.nall: number of alleles per locus
@loc.fac: locus factor for the  11 columns of @tab
@all.names: list of  1 components yielding allele names for each locus
@ploidy:  2
@type:  codom

@other: a list containing: elements without names
```

Now, titi only contains the 11 alleles of the third marker of toto.

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:

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

```
[1] "list"
```

```
> names(sepCats)
```

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

```
> sepCats$fca45
```

```
   ######################
   ### Genind object ###
   ######################
- genotypes of individuals -

S4 class:  genind
@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
@type:  codom

Optionnal contents:
@pop:  factor giving the population of each individual
@pop.names:  factor giving the population of each individual

@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"      "NDama"
 [5] "Somba"          "Aubrac"         "Bazadais"       "BlondeAquitaine"
 [9] "BretPieNoire"   "Charolais"      "Gascon"         "Limousin"
[13] "MaineAnjou"     "Montbeliard"    "Salers"
```

```
> obj$Borgou
```

```
   #####################
   ### Genind object ###
   #####################
- genotypes of individuals -

S4 class:  genind
@call: .local(x = x, i = i, j = j, treatOther = ..1, 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
@type:  codom

Optionnal contents:
@pop:  factor giving the population of each individual
@pop.names:  factor giving the population of each individual

@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"      "NDama"
 [5] "Somba"          "Aubrac"         "Bazadais"       "BlondeAquitaine"
 [9] "BretPieNoire"   "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 ### 
   ######################
- genotypes of individuals - 

S4 class:  genind
@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
@type:  codom

Optionnal contents: 
@pop:  factor giving the population of each individual
@pop.names:  factor giving the population of each individual

@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"       "NDama"          
 [5] "Somba"           "Aubrac"          "Bazadais"        "BlondeAquitaine"
 [9] "BretPieNoire"    "Charolais"       "Gascon"          "Limousin"       
[13] "MaineAnjou"      "Montbeliard"     "Salers"         
```

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


   #####################
   ### Genind object ###
   #####################
- genotypes of individuals -

S4 class:  genind
@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
@type:  codom

Optionnal contents:
@pop:  factor giving the population of each individual
@pop.names:  factor giving the population of each individual

@other: - empty -


> newObj$pop.names


         P1          P2
   "Borgou" "Charolais"
```

Done !

3.4 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.

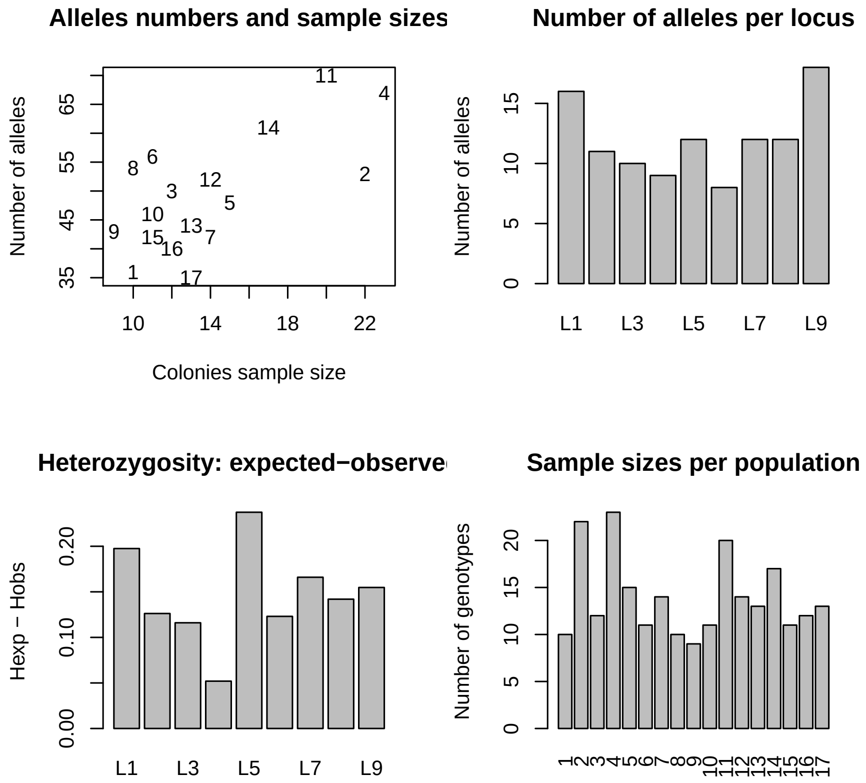
```
> toto <- summary(nancycats)


 # 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:
01 02 03 04 05 06 07 08 09 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:
       L1        L2        L3        L4        L5        L6        L7        L8
0.6682028 0.6666667 0.6793249 0.7083333 0.6329114 0.5654008 0.6497890 0.6184211
       L9
0.4514768

 # Expected heterozygosity:
       L1        L2        L3        L4        L5        L6        L7        L8
0.8657224 0.7928751 0.7953319 0.7603095 0.8702576 0.6884669 0.8157881 0.7603493
       L9
0.6062686
```

```
> names(toto)
```

```
[1] "N"        "pop.eff"  "loc.nall" "pop.nall" "NA.perc"  "Hobs"     "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 = 0.047, 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.

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

Population structure is traditionally measured and tested using F statistics, in particular Fst. *adegenet* proposes different tools in this respect: general F statistics (fstat), a test of overall population structure (gstat.randtest), and pairwise Fst between all pairs of populations in a dataset (pairwise.fst). The first two are wrappers for functions implemented in the *hierfstat* package; pairwise Fst is implemented in *adegenet*.

We illustrate their use using the dataset of microsatellite of cats from Nancy:

```
> library(hierfstat)
> data(nancycats)
> fstat(nancycats)
```

```
             pop       Ind
Total 0.08301274 0.1824701
pop   0.00000000 0.1084610
```

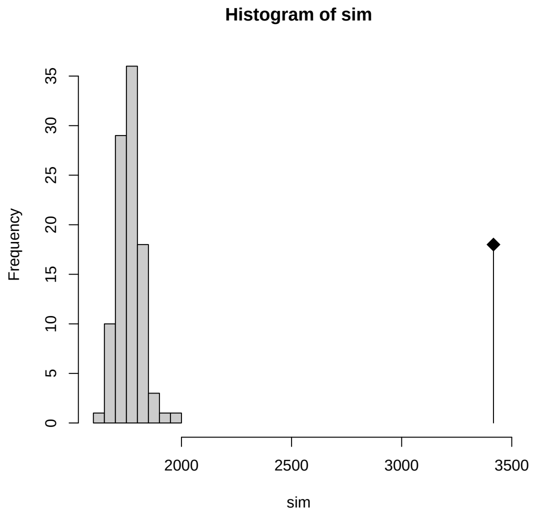
This table provides the three F statistics Fst (pop/total), Fit (Ind/total), and Fis (ind/pop). These are overall measures which take into account all genotypes and all loci.

Is the structure between populations significant? This question can be addressed using the G-statistic test (Goudet *et al.*, 1996); it is implemented for genind objects and produces a randtest object (package ade4).

```
> library(ade4)
> toto <- gstat.randtest(nancycats, nsim = 99)
> toto
```

```
Monte-Carlo test
Call: gstat.randtest(x = nancycats, nsim = 99)

Observation: 3416.974

Based on 99 replicates
Simulated p-value: 0.01
Alternative hypothesis: greater

    Std.Obs Expectation    Variance
   29.69807  1776.07127  3052.87498
```

```
> plot(toto)
```



Yes, it is (the observed value is indicated on the right, while histograms correspond to the permuted values). Note that *hierfstat* allows for more ellaborated tests, in particular when different levels of hierarchical clustering are available. Such tests are better done directly in *hierfstat*; for this, genind objects can be converted to the adequat format using genind2hierfstat. For instance:

```
> toto <- genind2hierfstat(nancycats)
> head(toto)
```

```
     pop   fca8  fca23  fca43  fca45  fca77  fca78  fca90  fca96  fca37
N215   1     NA 136146 139139 116120 156156 142148 199199 113113 208208
N216   1     NA 146146 139145 120126 156156 142148 185199 113113 208208
N217   1 135143 136146 141141 116116 152156 142142 197197 113113 210210
N218   1 133135 138138 139141 116126 150150 142148 199199  91105 208208
N219   1 133135 140146 141145 126126 152152 142148 193199 113113 208208
N220   1 135143 136146 145149 120126 150156 148148 193195  91113 208208
```

```
> varcomp.glob(toto$pop, toto[, -1])
```

```
$loc
            [,1]         [,2]      [,3]
fca8  0.08867161  0.116693199 0.6682028
fca23 0.05384247  0.077539920 0.6666667
fca43 0.05518935  0.066055996 0.6793249
fca45 0.05861271 -0.001026783 0.7083333
fca77 0.08810966  0.156863586 0.6329114
fca78 0.04869695  0.079006911 0.5654008
fca90 0.07540329  0.097194716 0.6497890
fca96 0.07538325 -0.005902071 0.7543860
fca37 0.04264094  0.116318729 0.4514768

$overall
      Pop       Ind      Error
```

```
0.5865502 0.7027442 5.7764917

$F
             Pop       Ind
Total 0.08301274 0.1824701
Pop   0.00000000 0.1084610
```

F statistics are provided in $F; for instance, here, F_{st} is 0.083.

Lastly, pairwise Fst is frequently used as a measure of distance between populations. The function pairwise.fst computes Nei's estimator (Nei, 1973) of pairwise Fst, computed 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):

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

3.6 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")
> dim(toto)
```

```
[1] 17  9
```

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)
```

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

```
> which(toto < 1e-04, TRUE)
```

```
    row col
P14  14   2
P02   2   7
P02   2   8
P05   5   9
```

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")
> toto$fca23$P06
```

```
        Pearson's Chi-squared test

data:  tab
X-squared = 19.25, df = 10, p-value = 0.0372
```

```
> toto$fca90$P10
```

```
        Pearson's Chi-squared test

data:  tab
X-squared = 19.25, df = 10, p-value = 0.0372
```

```
> toto$fca96$P10


        Pearson's Chi-squared test

data:  tab
X-squared = 4.8889, df = 10, p-value = 0.8985


> toto$fca37$P13


        Pearson's Chi-squared test

data:  tab
X-squared = 14.8281, df = 10, p-value = 0.1385
```

3.7 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 typology of individuals. Such analysis is straightforward using *adegenet* to prepare data and *ade4* for the analysis *per se*. One has first to replace missing data. Putting each missing observation at the mean of the concerned allele frequency seems the best choice (NA will be stuck at the origin).

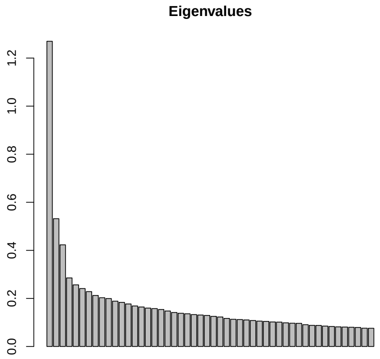
```
> data(microbov)
> any(is.na(microbov$tab))


[1] TRUE


> sum(is.na(microbov$tab))


[1] 6325
```

There are 6325 missing data. Assuming that these are evenly distributed (for illustration purpose only!), we replace them using na.replace. As we intend to use a PCA, the appropriate replacement method is to put each NA at the mean of the corresponding allele (argument ’method’ set to ’mean’).

```
> obj <- na.replace(microbov, method = "mean")


 Replaced 6325 missing values
```

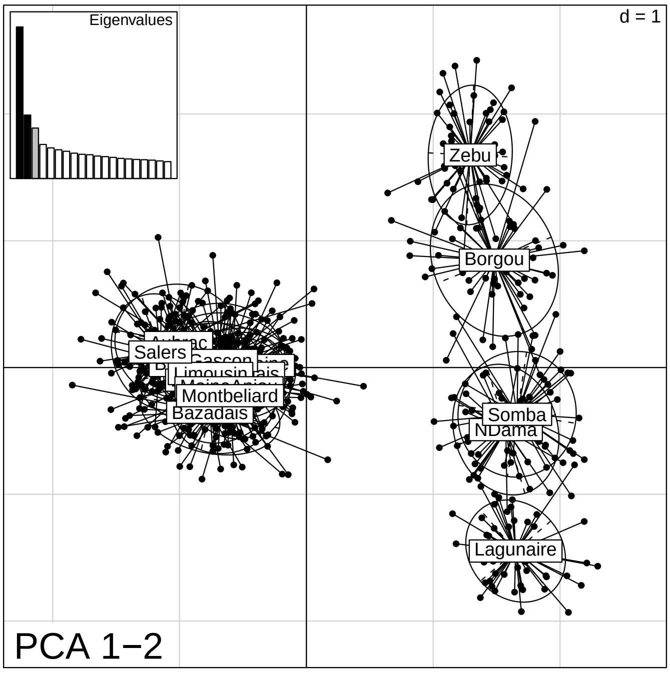
Done. Now, the analysis can be performed. Data are centred but not scaled as ’units’ are the same.

```
> pca1 <- dudi.pca(obj$tab, cent = TRUE, scale = FALSE, scannf = FALSE,
+     nf = 3)
> barplot(pca1$eig[1:50], main = "Eigenvalues")
```



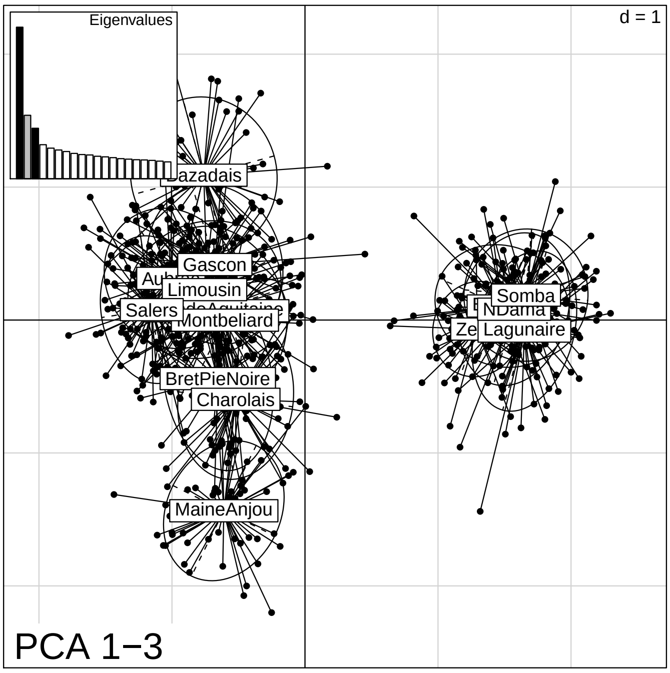
Here we represent the genotypes and 95% inertia ellipses for populations.

```
> s.class(pca1$li, obj$pop, lab = obj$pop.names, sub = "PCA 1-2",
+     csub = 2)
> add.scatter.eig(pca1$eig[1:20], nf = 3, xax = 1, yax = 2, posi = "top")
```



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: Overall, all breeds seem well differentiated.

```
> s.class(pca1$li, obj$pop, xax = 1, yax = 3, lab = obj$pop.names,
+     sub = "PCA 1-3", csub = 2)
> add.scatter.eig(pca1$eig[1:20], nf = 3, xax = 1, yax = 3, posi = "top")
```



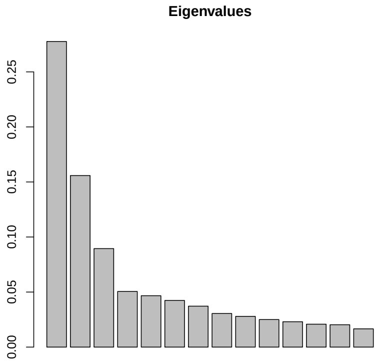
3.8 Performing a Correspondance Analysis on genpop objects

Being contingency tables, the @tab 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")
```

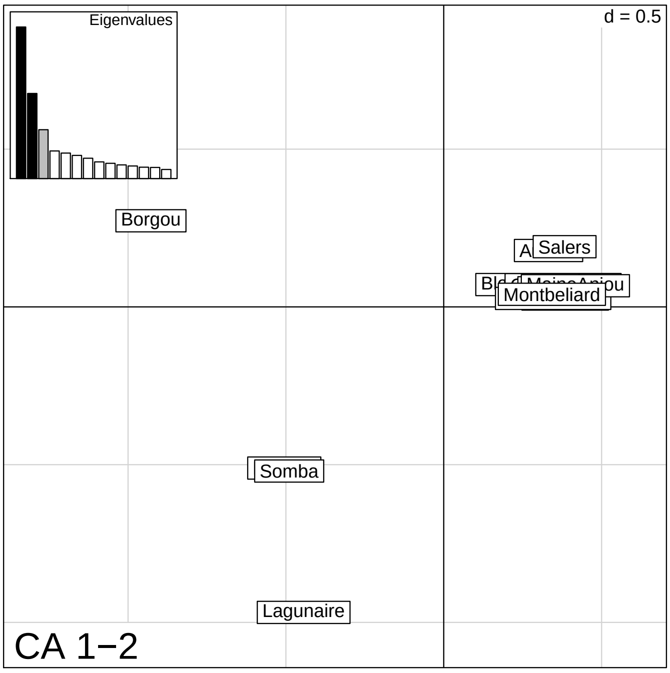
```
 Converting data from a genind to a genpop object...

 Replaced 0 missing values

...done.
```

```
> ca1 <- dudi.coa(as.data.frame(obj$tab), scannf = FALSE, nf = 3)
> barplot(ca1$eig, main = "Eigenvalues")
```

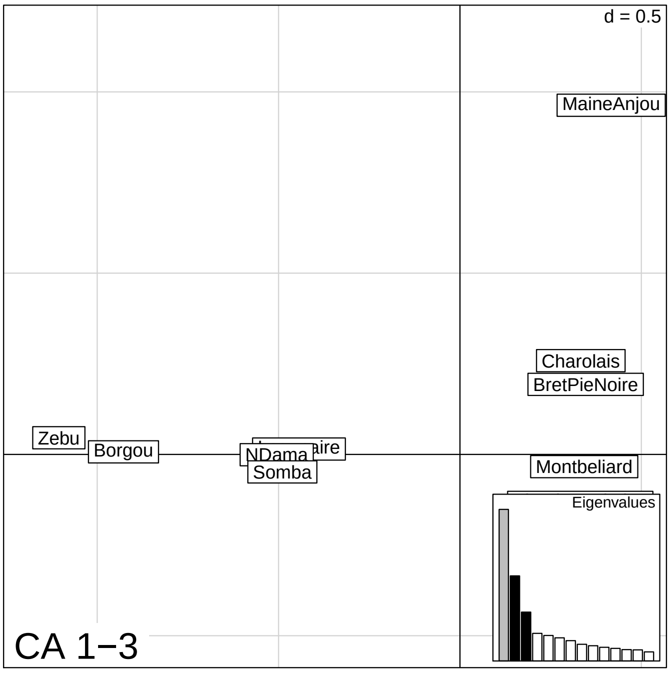


Now we display the resulting typologies:

```
> s.label(ca1$li, lab = obj$pop.names, sub = "CA 1-2", csub = 2)
> add.scatter.eig(ca1$eig, nf = 3, xax = 1, yax = 2, posi = "top")
```



```
> s.label(ca1$li, xax = 1, yax = 3, lab = obj$pop.names, sub = "CA 1-3",
+     csub = 2)
> add.scatter.eig(ca1$eig, nf = 3, xax = 2, yax = 3, posi = "bottomright")
```



Once again, axes are to be interpreted separately in terms of continental differentiation, a among-breed diversities.

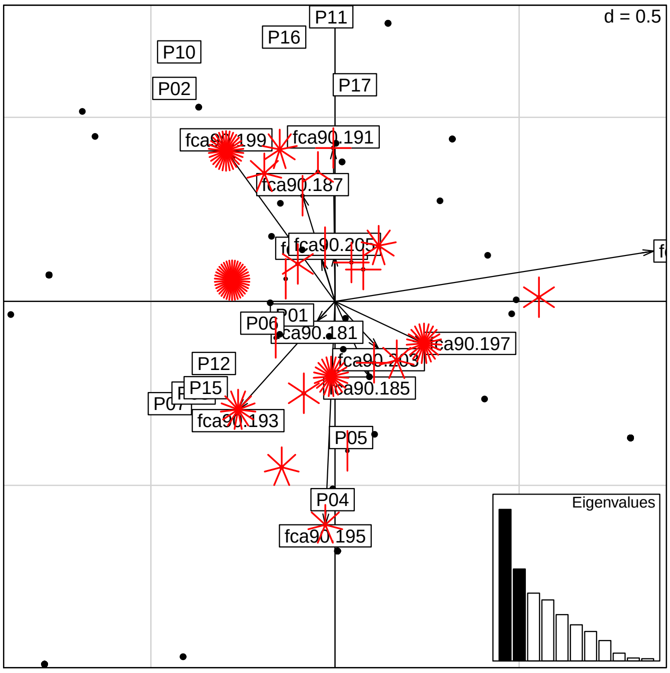
3.9 Analyzing a single locus

Here the emphasis is put on analyzing a single locus using different methods. Any marker can be isolated using the seploc instruction.

```
> data(nancycats)
> toto <- seploc(nancycats, truenames = TRUE, res.type = "matrix")
> X <- toto$fca90
```

fca90.ind is a matrix containing only genotypes for the marker fca90. It can be analyzed, for instance, using an inter-class PCA. This analyzis provides a typology of individuals having maximal inter-colonies variance.

```
> library(ade4)
> pcaX <- dudi.pca(X, cent = T, scale = F, scannf = FALSE)
> pcabetX <- between(pcaX, nancycats$pop, scannf = FALSE)
> s.arrow(pcabetX$c1, xlim = c(-0.9, 0.9))
> s.class(pcabetX$ls, nancycats$pop, cell = 0, cstar = 0, add.p = T)
> sunflowerplot(X %*% as.matrix(pcabetX$c1), add = T)
> add.scatter.eig(pcabetX$eig, xax = 1, yax = 2, posi = "bottomright")
```



Here the differences between individuals are mainly expressed by three alleles: 199, 197 and 193. However, there is no clear structuration to be seen at an individual level. Is F_{st} significant taking only this marker into account? We perform the G-statistic test and enventually compute the corresponding F statistics. Note that we use the constructor genind to generate an object of this class from X:

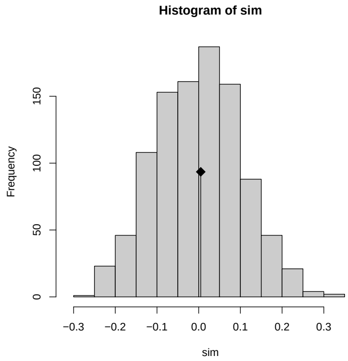
```
> fca90.ind <- genind(X, pop = nancycats$pop)
> gstat.randtest(fca90.ind, nsim = 999)


Monte-Carlo test
Call: gstat.randtest(x = fca90.ind, nsim = 999)

Observation: 437.135

Based on 999 replicates
Simulated p-value: 0.001
Alternative hypothesis: greater

    Std.Obs Expectation    Variance
   14.95085   188.47771   276.61161


> F <- varcomp(genind2hierfstat(fca90.ind))$F
> rownames(F) <- c("tot", "pop")
> colnames(F) <- c("pop", "ind")
> F


           pop       ind
tot 0.09168833 0.2098744
pop 0.00000000 0.1301162
```

In this case the information is best summarized by F statistics than by an ordination method. It is likely because all colonies are differentiated but none forming clusters of related colonies.

3.10 Testing for 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. We use Edwards' distance *versus* Euclidean distances between colonies.

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

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

 Replaced 9 missing values

...done.

> Dgen <- dist.genpop(toto, method = 2)
> Dgeo <- dist(nancycats$other$xy)
> library(ade4)
> ibd <- mantel.randtest(Dgen, Dgeo)
> ibd

Monte-Carlo test
Call: mantel.randtest(m1 = Dgen, m2 = Dgeo)

Observation: 0.00492068

Based on 999 replicates
Simulated p-value: 0.487
Alternative hypothesis: greater

     Std.Obs  Expectation     Variance
0.0003337896 0.0048859509 0.0108255438

> plot(ibd)
```



Isolation by distance is clearly not significant.

3.11 Using Monmonier's algorithm to define genetic boundaries

Monmonier's algorithm (Monmonier, 1973) 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, Manni *et al.* (2004) 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 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 defined spatial connectivity among 'points' (genotypes or populations), any couple of points 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.

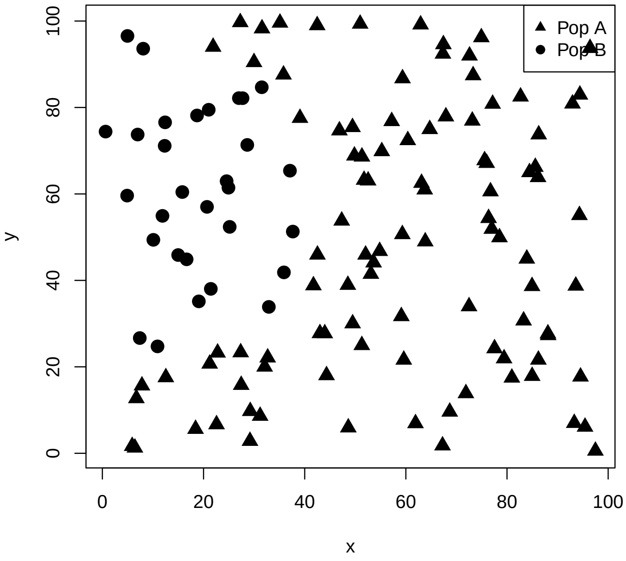
Let's take the example from the function's manpage and detail it. The dataset used is sim2pop.

```
> data(sim2pop)
> sim2pop


   #####################
   ### Genind object ###
   #####################
- genotypes of individuals -

S4 class:  genind
@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
@type:  codom

Optionnal contents:
@pop:  factor giving the population of each individual
@pop.names:  factor giving the population of each individual

@other: a list containing: xy
```

```
> summary(sim2pop$pop)
```

```
P01 P02
100  30
```

```
> temp <- sim2pop$pop
> levels(temp) <- c(17, 19)
> temp <- as.numeric(as.character(temp))
> plot(sim2pop$other$xy, pch = temp, cex = 1.5, xlab = "x", ylab = "y")
> legend("topright", leg = c("Pop A", "Pop B"), pch = c(17, 19))
```



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). So, what do monmonier ask for?

```
> args(monmonier)
```

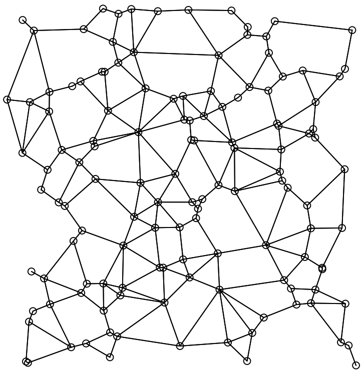
```
function (xy, dist, cn, threshold = NULL, bd.length = NULL, nrun = 1,
    skip.local.diff = rep(0, nrun), scanthres = is.null(threshold),
    allowLoop = TRUE)
NULL
```

The first argument (xy) is a matrix of geographic coordinates, already stored in sim2pop. Next argument is an object of class dist, which is basically a distance matrix cut in half. For now, we will use the classical Euclidean distance among alleles frequencies of genotypes. This is obtained by:

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

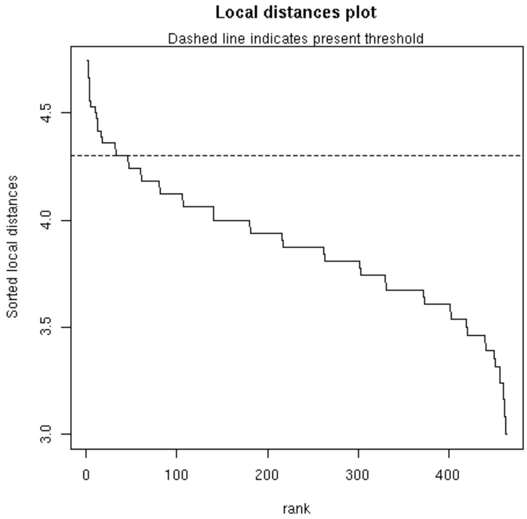
The next argument (cn) is a connection network. As existing routines to build such networks are spread over several packages, the function chooseCN will help you choose one. This is an interactive function, so difficult to demonstrate here (see ?chooseCN). Here we ask the function not to ask for a choice (ask=FALSE) and select the second type of graph which is the one 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. Here, no boundary is visible: we stop.

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?

```
> library(hierfstat)
> temp <- genind2hierfstat(sim2pop)
> varcomp.glob(temp[, 1], temp[, -1])$F
```

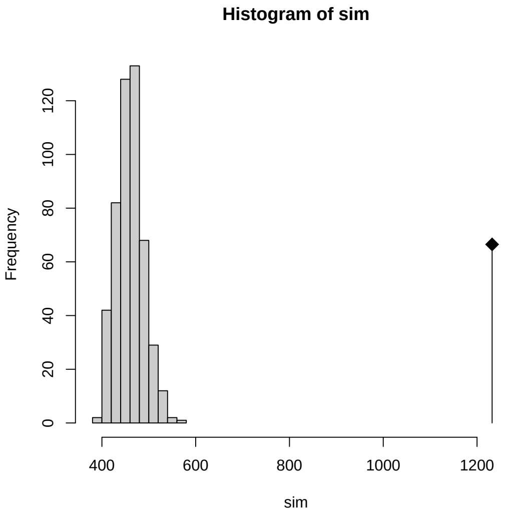
```
             Pop         Ind
Total 0.03824374 -0.07541793
Pop   0.00000000 -0.11818137
```

This value is somewhat moderate ($F_{st} = 0.038$). Is it significant?

```
> gtest <- gstat.randtest(sim2pop)
> gtest
```

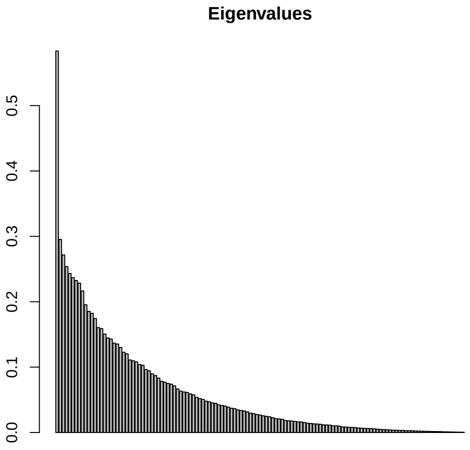
```
Monte-Carlo test
Call: gstat.randtest(x = sim2pop)

Observation: 1232.192

Based on 499 replicates
Simulated p-value: 0.002
Alternative hypothesis: greater

    Std.Obs Expectation    Variance
   27.65661   459.81364   779.94176
```

```
> plot(gtest)
```



Yes, it is very significant. The two samples are indeed genetically differenciated. So, 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 simple ordination method like Principal Coordinates Analysis.

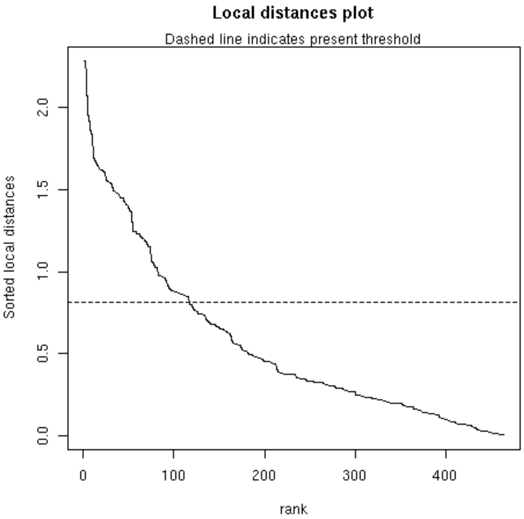
```
> library(ade4)
> pco1 <- dudi.pco(D, scannf = FALSE, nf = 1)
> barplot(pco1$eig, main = "Eigenvalues")
```



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

```
> 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):  0.8154
$xy: spatial coordinates
$cn: connection network

      # Runs content #
# Run 1
# First direction
Class:  list
$path:
               x        y
Point_1 14.98299 93.81162

$values:
 2.281778
# 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:
 2.281778 1.617905 1.953220 ...
```

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] 2.281778
```

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

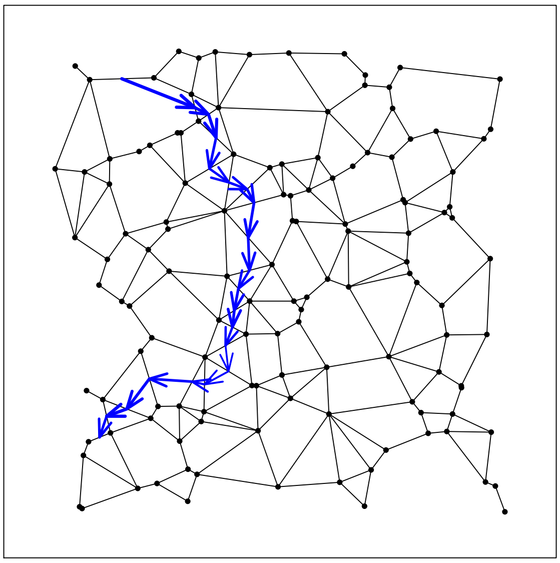
```
> coords.monmonier(mon1)

$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.

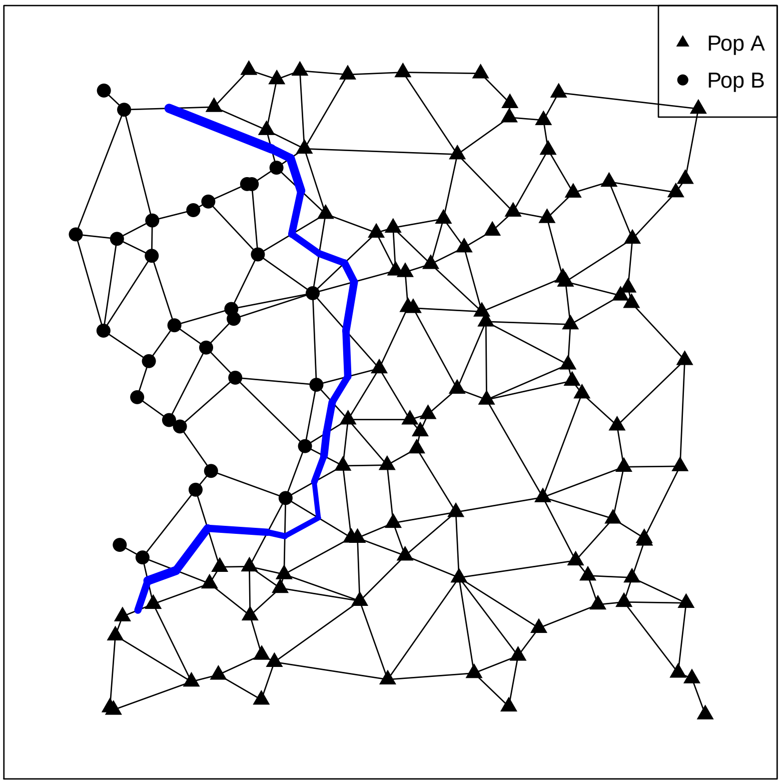
Finally, you can plot very simply the obtained boundary using the method plot:

```
> 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 = 8)
> temp <- sim2pop$pop
> levels(temp) <- c(17, 19)
> temp <- as.numeric(as.character(temp))
> points(sim2pop$other$xy, pch = temp, cex = 1.3)
> legend("topright", leg = c("Pop A", "Pop B"), pch = c(17, 19))
```



Not too bad...

3.12 How to simulate 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"     "NDama"
 [5] "Somba"          "Aubrac"        "Bazadais"      "BlondeAquitaine"
 [9] "BretPieNoire"   "Charolais"     "Gascon"        "Limousin"
[13] "MaineAnjou"     "Montbeliard"   "Salers"
```

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

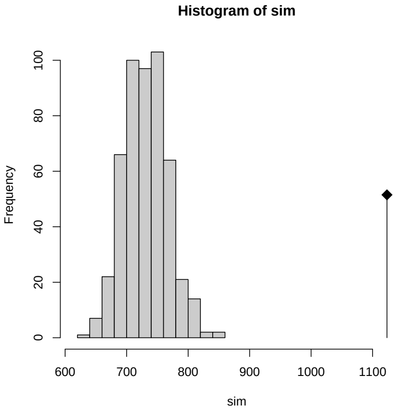
A first generation (F1) of hybrids 'zebler' is obtained. Is it possible to perform a backcross, say, with 'salers' population? Yes, here it is:

```
> F2 <- hybridize(salers, zebler, n = 40)
> F3 <- hybridize(salers, F2, n = 40)
> F4 <- hybridize(salers, F3, n = 40)
```

and so on... Are these hybrids still genetically distinct? Let's merge all hybrids in a single dataset and test for genetic differentiation:

```
> dat <- repool(zebler, F2, F3, F4)
> test <- gstat.randtest(dat)
> plot(test)
> temp <- genind2hierfstat(dat)
> varcomp.glob(temp[, 1], temp[, -1])$F
```

```
             Pop         Ind
Total 0.01384859 -0.03399172
Pop   0.00000000 -0.04851213
```



The F_{st} is not very strong (0.013) but still very significant: hybrids are still pretty well differentiated.

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!

3.13 Handling presence/absence data

Adegenet was primarily suited to handle codominant, multiallelic markers like microsatellites. However, dominant binary 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("http://adegenet.r-forge.r-project.org/files/AFLP.txt",
+     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")
> obj
```

```
   #####################
   ### Genind object ###
   #####################
- genotypes of individuals -

S4 class:  genind
```

```
@call: genind(tab = dat, ploidy = 1, type = "PA")

@tab:  7 x 4 matrix of genotypes

@ind.names: vector of  7 individual names
@loc.names: vector of  4 locus names
@loc.nall: NULL
@loc.fac: NULL
@all.names: NULL
@ploidy:  1
@type:  PA

Optionnal contents:
@pop:  - empty -
@pop.names:  - empty -

@other: - empty -


> truenames(obj)


     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
```

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)
> summary(obj)


 # Total number of genotypes:  7

 # Population sample sizes:
a b
4 3

 # Percentage of missing data:
[1] 7.142857
```

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 ###
       #####################
- Alleles counts for populations -

S4 class:  genpop
@call: genind2genpop(x = obj)

@tab:  2 x 4 matrix of alleles counts

@pop.names: vector of  2 population names
@loc.names: vector of  4 locus names
@loc.nall: NULL
@loc.fac: NULL
@all.names: NULL
@ploidy:  1
@type:  PA

@other: - empty -
```

```
> obj2@tab
```

```
  L1 L2 L3 L4
1  2  2  3  3
2  2  2  2  1
```

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

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

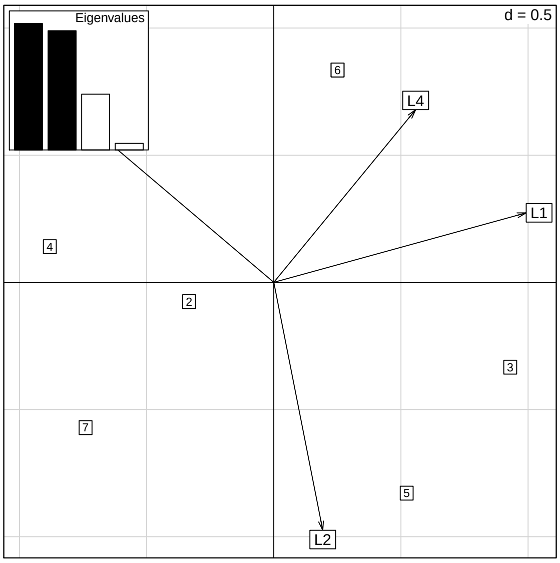
```
 Replaced 2 missing values
```

```
> objNoNa@tab
```

```
  L1 L2 L3 L4
1  1  0  1  1
2  0  1  1  1
3  1  1  0  1
4  0  0  1  0
5  1  1  0  0
6  1  0  1  1
7  0  1  1  0
```

Now the PCA is performed:

```
> library(ade4)
> pca1 <- dudi.pca(objNoNa, scannf = FALSE, scale = FALSE)
> scatter(pca1)
```



More generally, multivariate analyses from ade4, the sPCA (spca), 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.

3.14 Assigning genotypes to clusters using Discriminant Analysis

The approach described below led to the development of a a new methodological approach for studying the genetic diversity of biological populations, called the Discriminant Analysis of Principal Components (DAPC, Jombart et al. submitted). This method has been implemented by the functions find.clusters and dapc but is still considered under development. It will be documented along with this section pending the publication of the corresponding paper.

3.14.1 Defining clusters

Bayesian clustering methods are not the only approaches for assigning genotypes to groups of genotypes. Discriminant analysis (DA; for a general presentation, see Lachenbruch & Goldstein, 1979) is a multivariate method that has been used for the exact same purpose (Beharav & Nevo, 2003). It can be applied whenever pre-defined groups exist, to assign genotypes to and assess the robustness of these groups. New genotypes with unknown group can also be assigned to existing clusters. Although a few precautions have to be taken when applying DA (see Jombart *et al.* (2009) for a short overview), this is a useful and straightforward approach. It is here illustrated using cat colonies of Nancy, France (nancycats dataset).

```
> data(nancycats)
> nancycats
```

```
   ######################
   ### Genind object ###
   ######################
- genotypes of individuals -

S4 class:  genind
@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
@type:  codom

Optionnal contents:
@pop:  factor giving the population of each individual
@pop.names:  factor giving the population of each individual

@other: a list containing: xy
```

```
> unique(pop(nancycats))
```

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

This dataset contains 237 genotypes of cats sampled over 17 colonies. A usual PCA on the allele frequencies of the populations would not show any structure, but colonies seem nonetheless mildly differentiated, as confirmed by Goudet's G test (and the F_{st} value):

```
> gstat.randtest(nancycats, n = 199)
```

```
Monte-Carlo test
Call: gstat.randtest(x = nancycats, nsim = 199)

Observation: 3416.974

Based on 199 replicates
Simulated p-value: 0.005
Alternative hypothesis: greater

    Std.Obs Expectation    Variance
   29.88014  1760.48172  3073.36041
```

```
> fstat(nancycats, fstonly = TRUE)
```

```
[1] 0.08301274
```

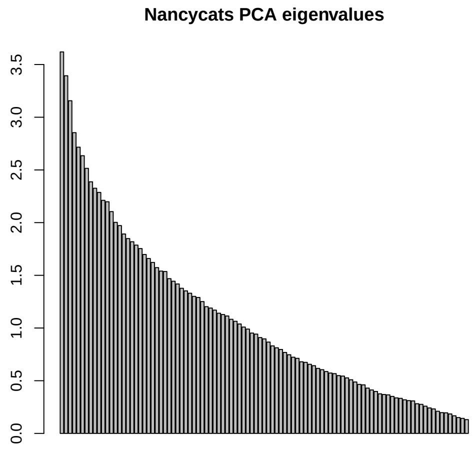
DA can be used to find the linear combinations of alleles that discriminate best the groups of genotypes (here, colonies). While a powerful method, DA is impaired by correlation between predictors, which arises for instance when linkage disequilibrium occurs between alleles. It is also impracticable when the number of alleles (p) is greater than the number of genotypes (n), and it generally requires $n >> p$ to yield reliable (numerically stable) results.

Thus, DA can seem often problematic when it comes to genetic data. One simple and efficient solution to all these issues is to transform alleles frequencies into a few independent (uncorrelated) components that summarise most of the genetic information, retaining only essential genetic features. This can be achieved by different multivariate methods; here, we shall use PCA. Genotype data are first transformed into scaled allele frequencies (using scaleGen):

```
> x <- scaleGen(nancycats, missing = "mean")
```

Then, we proceed to the PCA, retaining many principal components (PCs):

```
> x.pca <- dudi.pca(x, center = FALSE, scale = FALSE, scannf = FALSE,
+     nf = 100)
> barplot(x.pca$eig, main = "Nancycats PCA eigenvalues")
```



These eigenvalues indicate no structure, but this is no problem since here, we just use PCA as a mean of transforming genetic variables in an adequate way. PCs are stored in `x.pca$li`:

```
> head(x.pca$li[, 1:5])
```

```
           Axis1      Axis2     Axis3      Axis4       Axis5
N215  0.05847037 -0.4228593 0.4433385 -2.2204574 -1.94565586
N216 -0.30724734 -1.2376546 0.6085813 -0.8605435 -2.59138468
N217  0.12843698  1.6457391 1.4390692 -2.9874897 -0.09586744
N218 -0.93014696 -0.4694263 0.6595994 -2.0340478 -0.94786234
N219 -0.36584543  0.1139781 0.8042301 -0.9972465 -0.39733811
N220 -0.49235825 -1.2805118 0.2417573 -0.6060821 -1.63763566
```

```
> dim(x.pca$li)
```

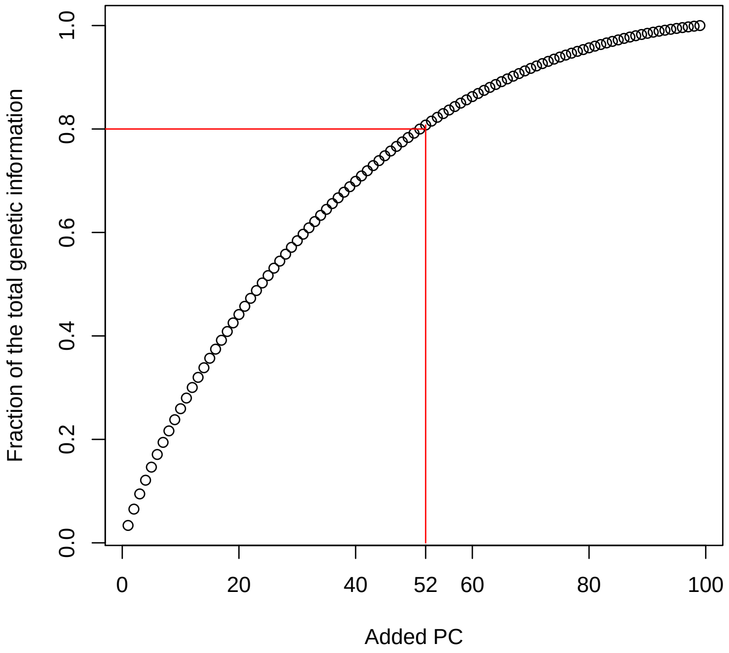
```
[1] 237  99
```

Now, the question relates to how many PCs should be retained. This choice could be based on the success of assignment using DA (looking for the optimal value), or on a given fraction of the genetic diversity we would like to retain. We here use the latter, more simple and sufficient to illustrate the method. The following graph shows the cumulative amount of genetic information brought by each added PC:

```
> temp <- cumsum(x.pca$eig)/sum(x.pca$eig)
> plot(temp, xlab = "Added PC", ylab = "Fraction of the total genetic information")
> min(which(temp > 0.8))
```

```
[1] 52
```

```
> axis(1, at = 52, lab = 52)
> segments(52, 0, 52, temp[52], col = "red")
> segments(-5, 0.8, 52, 0.8, col = "red")
```



For instance, the first 52 PCs are sufficient to express 80% of the total genetic variability (see red segments). We choose to retain these 52 PCs and use them as new predictors in a DA. While there is a discrimin function in ade4, we use the function lda from the MASS package, which allows assigning (possibly new) genotypes to clusters.

```
> x.lda <- lda(x.pca$li[, 1:52], grouping = pop(nancycats))
> names(x.lda)
```

```
[1] "prior"   "counts"  "means"   "scaling" "lev"     "svd"     "N"
[8] "call"
```

The object x.lda contains the results of the DA. For instance, coefficients of the linear combinations (*discriminant functions*) are stored in x.lda$scaling. For a further description of the content of these objects, see ?x.lda. As far as assignment is concerned, the most interesting information is provided by predict:

```
> x.pred <- predict(x.lda)
> names(x.pred)
```

```
[1] "class"     "posterior" "x"
```

```
> x.pred$class
```

```
  [1] 1  1  1  1  7  1  1  1  1  1  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2 
 [26] 2  3  2  2  2  2  2  3  3  3  4  3  3  5  3  3  3  3  3  4  4  4  4  3  4 
 [51] 4  4  4  4  4  4  4  4  4  4  4  4  8  1  4  4  4  5  5  5  5  5  5  5  5 
 [76] 5  5  5  5  5  5  5  6  6  6  6  4  6  6  6  3  6  6  7  7  7  7  7  7  7 
[101] 7  12 7  7  7  7  7  4  10 8  15 8  8  8  8  8  8  9  9  9  9  9  9  9  9 
[126] 9  10 10 10 10 10 10 6  10 10 10 9  11 11 11 11 11 11 5  5  11 11 11 11 11
[151] 11 11 11 11 3  3  11 12 12 12 12 12 12 12 13 13 13 13 13 13 13 13 13 13 13
[176] 13 13 14 14 14 14 5  14 14 14 14 14 14 4  9  14 14 14 14 15 15 15 15 15 15
[201] 15 15 12 15 15 16 16 16 16 16 16 16 16 16 16 16 16 12 12 12 12 12 16 12 17
[226] 17 17 17 17 17 17 17 17 17 17 17 17
Levels: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
```

```
> head(x.pred$posterior[, 1:5])
```

```
             1            2            3            4            5
N215 0.9789727 9.577822e-09 4.774675e-06 1.457425e-05 4.991132e-04
N216 0.8973141 1.927180e-08 2.356494e-06 2.098230e-04 5.954925e-05
N217 0.9999967 4.642204e-18 2.802290e-12 2.187224e-13 2.285345e-07
N218 0.8025053 6.453325e-12 2.245869e-12 1.445570e-12 7.946815e-10
N219 0.2111993 7.718517e-10 1.371071e-08 3.039278e-09 1.380308e-07
N220 0.9990817 9.608996e-10 1.596801e-07 1.499430e-07 4.625445e-04
```

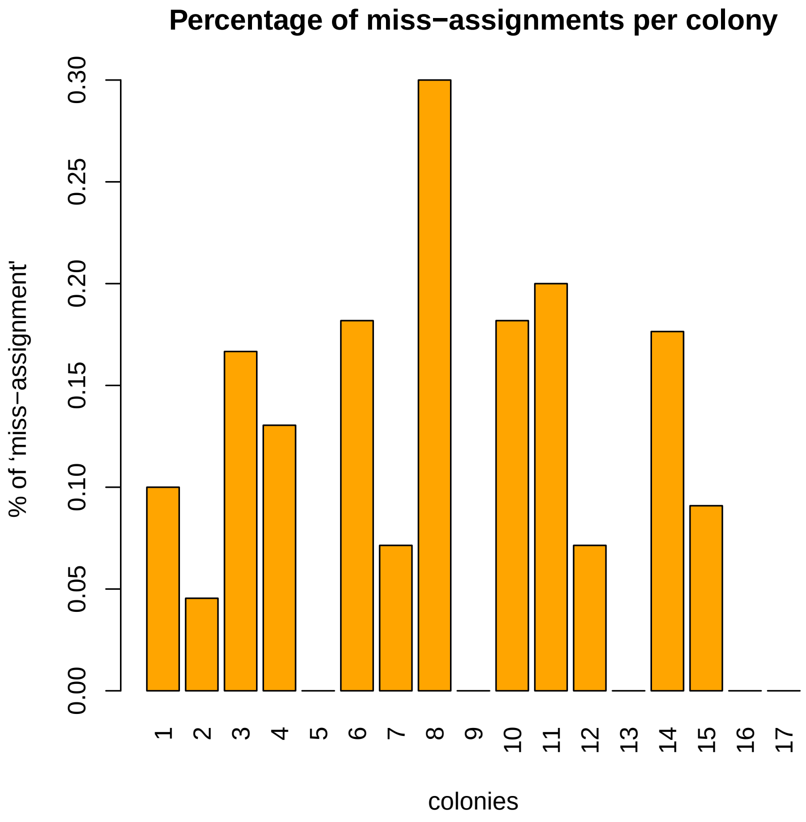
The `class` slot contains the cluster to which each genotype would be assigned with the highest probability, while `posterior` gives posterior probabilities of assignment of genotypes to clusters. The inferred groups can be compared easily to actual colonies:

```
> mean(x.pred$class == pop(nancycats))
```

```
[1] 0.8987342
```

In this case, each genotype would be assigned to the colony where it was actually found in 90% of cases. 'Miss-assigned' individuals could be hybrids or migrants, or simply reflect less clear-cut clusters. It is easy to check if some colonies have more of these:

```
> misAs <- tapply(x.pred$class != pop(nancycats), pop(nancycats),
+     mean)
> barplot(misAs, xlab = "colonies", ylab = "% of `miss-assignment'",
+     col = "orange", las = 3)
> title("Percentage of miss-assignments per colony")
```



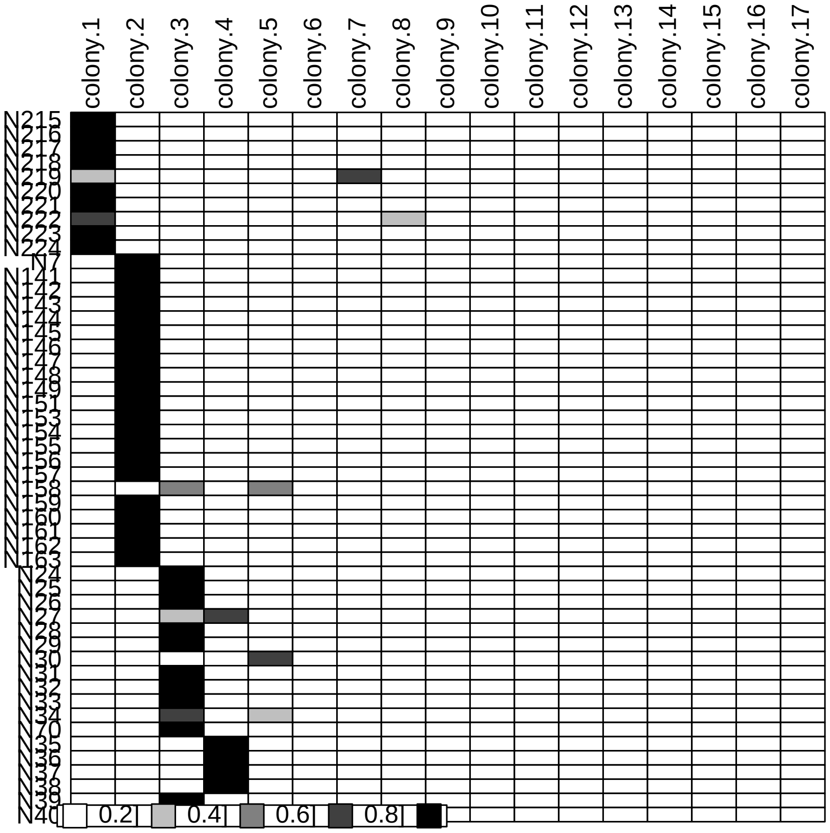
For more details about genotypes, we can have a look at the `posterior` component, which gives probabilities of belonging to each cluster for each genotype:

```
> head(x.pred$posterior[, 1:10])
```

```
             1            2            3            4            5            6
N215 0.9789727 9.577822e-09 4.774675e-06 1.457425e-05 4.991132e-04 9.618366e-09
N216 0.8973141 1.927180e-08 2.356494e-06 2.098230e-04 5.954925e-05 2.773016e-09
N217 0.9999967 4.642204e-18 2.802290e-12 2.187224e-13 2.285345e-07 3.960112e-12
N218 0.8025053 6.453325e-12 2.245869e-12 1.445570e-12 7.946815e-10 1.027150e-13
N219 0.2111993 7.718517e-10 1.371071e-08 3.039278e-09 1.380308e-07 1.470112e-09
N220 0.9990817 9.608996e-10 1.596801e-07 1.499430e-07 4.625445e-04 2.407030e-10
                7            8            9           10
N215 4.566426e-04 5.079117e-05 2.460440e-08 1.187296e-04
N216 8.503242e-04 3.125840e-03 4.100707e-09 1.303221e-05
N217 1.348738e-06 2.201991e-12 4.042410e-17 6.434196e-09
N218 1.522997e-05 7.363011e-08 4.008925e-09 1.399387e-09
N219 7.861739e-01 9.529260e-08 9.519421e-11 5.942611e-10
N220 8.340154e-08 7.668008e-05 1.783684e-10 7.353064e-06
```

This information is best perceived graphically (here, for the first 50 genotypes):

```
> table.paint(head(x.pred$posterior, 50), col.lab = paste("colony",
+     1:17, sep = "."))
```



For instance, N215 (first row) is clearly assigned to colony 1, while it is unclear whether N158 (middle) belongs to colony 3 or 5. Such graphics is really good at summarising probabilities of assignment. In particular, it can be employed even when the number of clusters is relatively high, which would not be the case with classical graphs proposed in STRUCTURE.

3.14.2 Assigning new individuals

In certain cases, we may want to assign new genotypes to a pre-existing classification, as defined by a DA. This can be the case when new samples have been made available after a pilot study, or when doing cross-validation. We will simulate these cases by drawing 30 genotypes at random, and then trying to assign them to the defined clusters.

The following code only repeats the former analyses after withdrawing the 30 genotypes:

```
> id <- sample(1:237, 30)
> newSamp <- nancycats[id]
> newObj <- nancycats[-id]
> newx <- x[-id, ]
> newx.pca <- dudi.pca(newx, center = FALSE, scale = FALSE, scannf = FALSE,
+     nf = 100)
> newx.lda <- lda(newx.pca$li[, 1:52], grouping = pop(newObj))
```

The new object x.lda now contains a DA based on only 207 individuals. Our purpose is to assign the new genotypes (newObj) to existing clusters based on the discriminant functions defined in x.lda. It is a bit tricky, because we have to make sure new data are transformed like the old data, that is, with the same centring and scaling. Fortunately, centring and scaling values are stored as attributes in x, and can be provided to a new call to scaleGen:

```
> newSamp


   #####################
   ### Genind object ###
   #####################
- genotypes of individuals -

S4 class:  genind
@call: .local(x = x, i = i, j = j, drop = drop)

@tab:  30 x 108 matrix of genotypes

@ind.names: vector of  30 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
@type:  codom

Optionnal contents:
@pop:  factor giving the population of each individual
@pop.names:  factor giving the population of each individual

@other: a list containing: xy


> newSamp.x <- scaleGen(newSamp, center = attr(x, "scaled:center"),
+     scale = attr(x, "scaled:scale"), missing = "mean")
```

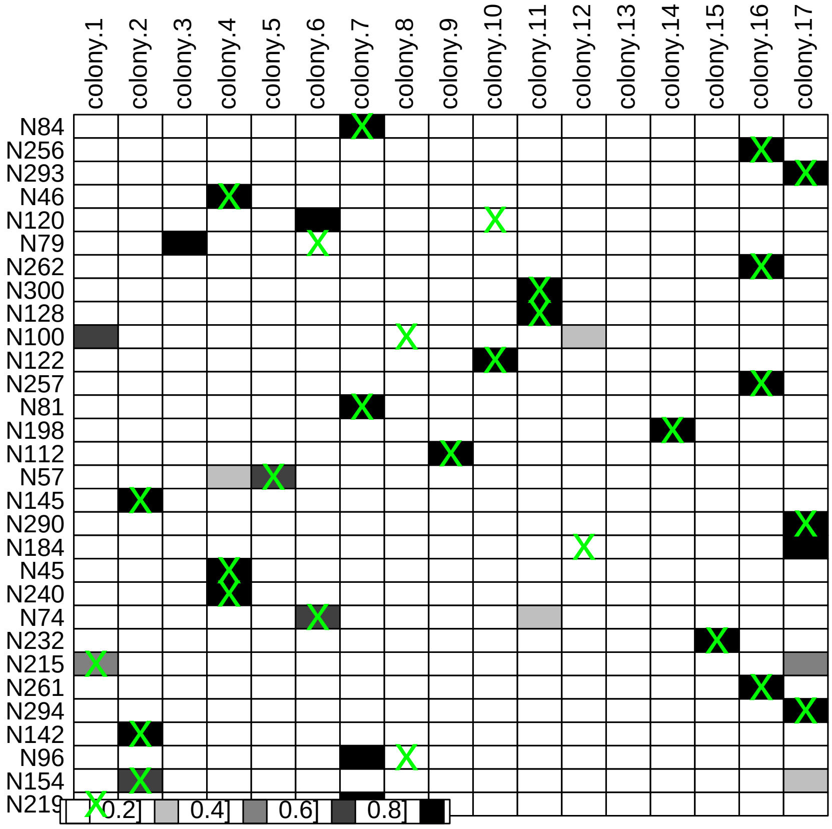
From there, the coordinates of these new genotypes onto the former PCA axes are obtained easily using suprow:

```
> newSamp.pc <- suprow(newx.pca, newSamp.x)$lisup
> dim(newSamp.pc)


[1] 30 99


> head(newSamp.pc[, 1:5])
```

```
          Axis1       Axis2      Axis3      Axis4      Axis5
N84   0.5047608  0.04247984  0.9360310 -0.4568648 -0.5826832
N256  0.3689253 -0.83837998  0.2267852  0.6999959  1.2872675
N293 -0.2335015  1.11309930  0.4451031 -0.5755287 -0.6297878
N46  -0.3714081  1.64224571 -1.0602972  0.3324720 -0.4590934
N120  0.5303501  0.24115433  0.8207165 -0.7727013  0.8559355
N79   0.7176159  0.12113290  0.5928074  1.2419890 -0.3078119
```

The object `newSamp.pc` contains the supplementary principal components for the new 30 genotypes. To find the probabilities of belonging to any cluster for each new genotype, we simply use `predict`, and plot posterior probabilities as before, adding a green cross to indicate the actual colony:

```
> newSamp.pred <- predict(newx.lda, newSamp.pc[, 1:52])

> table.paint(newSamp.pred$posterior, col.lab = paste("colony",
+     1:17, sep = "."))
> points(as.numeric(as.character(pop(newSamp))), 30:1, pch = "x",
+     col = "green", cex = 2)
> mean(as.character(newSamp.pred$class) == as.character(pop(newSamp)))

[1] 0.8
```



In this example, the new genotypes have been assigned to their actual group in 80% of cases. If our purpose was to cross-validate the classification of genotypes into groups, we would repeat this operation a large number of times, drawing a different random sample of genotypes each time.

4 Frequently Asked Questions

4.0.3 The function ... is not found. What's wrong?

You installed R, and adegenet, and all went ok. Yet, when trying to use some functions, like `read.genetix` for instance, you get an error message saying that the function is not found. The most likely explanation is that you do not have the most recent version of adegenet. This can be because you did not update your packages (see function `update.packages`). If your packages have been updated, and the problem persist, then you are likely using an outdated version of R, and though adegenet is up-to-date with respect to this R version, you are still using an outdated version of the package.

To know which version of adegenet you are using:

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

```
[1] "1.2-7"
```

And to know which version of R you are using:

```
> R.version.string
```

```
[1] "R version 2.11.1 (2010-05-31)"
```

References

Beharav, A. & Nevo, E. (2003). Predictive validity of discriminant analysis for genetic data. *Genetica* **119**, 259–267.

Charif, D. & 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* (U. Bastolla, H. R., M. Porto & Vendruscolo, M., eds.), Biological and Medical Physics, Biomedical Engineering. New York: Springer Verlag, pp. 207–232. ISBN : 978-3-540-35305-8.

Chessel, D., Dufour, A.-B. & Thioulouse, J. (2004). The ade4 package-I-one-table methods. *R News* **4**, 5–10.

Goudet, J. (2005). HIERFSTAT, a package for R to compute and test hierarchical F-statistics. *Molecular Ecology Notes* **5**, 184–186.

Goudet, J., Raymond, M., Meeüs, T. & Rousset, F. (1996). Testing differentiation in diploid populations. *Genetics* **144**, 1933–1940.

Ihaka, R. & Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational & Graphical Statistics* **5**, 299–314.

Jombart, T., Pontier, D. & Dufour, A.-B. (2009). Genetic markers in the playground of multivariate analysis. *Heredity* **102**, 330–341.

Lachenbruch, P. A. & Goldstein, M. (1979). Discriminant analysis. *Biometrics* **35**, 69–85.

Manni, F., Guérard, E. & Heyer, E. (2004). Geographic patterns of (genetic, morphologic, linguistic) variation: how barriers can be detected by "Monmonier's algorithm". *Human Biology* **76**, 173–190.

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

Nei, M. (1973). Analysis of gene diversity in subdivided populations. *Proc Natl Acad Sci U S A* **70**(12), 3321–3323.

Paradis, E. (2006). *Analysis of Phylogenetics and Evolution with R*. Springer-Verlag, Heidelberg.

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

R Development Core Team (2009). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL `http://www.R-project.org`. ISBN 3-900051-07-0.