Analysing genome-wide SNP data using *adegenet* 1.4-1

Thibaut Jombart and Caitlin Collins *

Imperial College London

MRC Centre for Outbreak Analysis and Modelling

March 24, 2014

Abstract

Genome-wide SNP data can quickly be challenging to analyse using standard computer. The package *adegenet* [1] for the R software [2] implements representation of these data with unprecedented efficiency using the classes `SNPbin` and `genlight`, which can require up to 60 times less RAM than usual representation using allele frequencies. This vignette introduces these classes and illustrates how these objects can be handled and analyzed in R.

*`tjombart@imperial.ac.uk`, `caitlin.collins12@imperial.ac.uk`

Contents

1 Introduction

Modern sequencing technologies now make complete genomes more widely accessible. The subsequent amounts of genetic data pose challenges in terms of storing and handling the data, making former tools developed for classical genetic markers such as microsatellite impracticable using standard computers. Adegenet has developed new object classes dedicated to handling genome-wide polymorphism (SNPs) with minimum random access memory (RAM) requirements.

Two new formal classes have been implemented: `SNPbin`, used to store genome-wide SNPs for one individual, and `genlight`, which stored the same information for multiple individuals. Information represented this way is binary: only biallelic SNPs can be stored and analyzed using these classes. However, these objects are otherwise very flexible, and can incorporate different levels of ploidy across individuals within a single dataset. In this vignette, we present these object classes and show how their content can be further handled and content analyzed.

2 Classes of objects

2.1 `SNPbin`: storage of single genomes

The class `SNPbin` is the core representation of biallelic SNPs which allows to represent data with unprecedented efficiency. The essential idea is to code binary SNPs not as integers, but as bits. This operation is tricky in R as there is no handling of bits, only bytes – series of 8 bits. However, the class `SNPbin` handles this transparently using sub-rountines in C language. Considerable efforts have been made so that the user does not have to dig into the complex internal structure of the objects, and can handle `SNPbin` objects as easily as possible.

Like `genind` and `genpop` objects, `SNPbin` is a formal "S4" class. The structure of these objects is detailed in the dedicated manpage (`?SNPbin`). As all S4 objects, instances of the class `SNPbin` are composed of slots accessible using the `@` operator. This content is generic (it is the same for all instances of the class), and returned by:

```
library(adegenet)
getClassDef("SNPbin")

## Class "SNPbin" [package "adegenet"]
##
## Slots:
##
## Name:         snp      n.loc    NA.posi      label     ploidy
## Class:       list    integer    integer charOrNULL    integer
```

The slots respectively contain:

- `snp`: SNP data with specific internal coding.

- `n.loc`: the number of SNPs stored in the object.
- `NA.posi`: position of the missing data (NAs).
- `label`: an optional label for the individual.
- `ploidy`: the ploidy level of the genome.

New objects are created using `new`, with these slots as arguments. If no argument is provided, an empty object is created:

```
new("SNPbin")
```

```
##  === S4 class SNPbin ===
##  0 SNPs coded as bits
##  Ploidy: 1
##  0 (NaN %) missing data
```

In practice, only the `snp` information and possibly the ploidy has to be provided; various formats are accepted for the `snp` component, but the simplest is a vector of integers (or numeric) indicating the number of second allele at each locus. The argument `snp`, if provided alone, does not have to be named:

```
x <- new("SNPbin", c(0,1,1,2,0,0,1))
x
```

```
##  === S4 class SNPbin ===
##  7 SNPs coded as bits
##  Ploidy: 2
##  0 (0 %) missing data
```

If not provided, the ploidy is detected from the data and determined as the largest number in the input vector. Obviously, in many cases this will not be adequate, but ploidy can always be rectified afterwards; for instance:

```
x
```

```
##  === S4 class SNPbin ===
##  7 SNPs coded as bits
##  Ploidy: 2
##  0 (0 %) missing data
```

```
ploidy(x) <- 3
x
```

```
##  === S4 class SNPbin ===
##  7 SNPs coded as bits
##  Ploidy: 3
##  0 (0 %) missing data
```

The internal coding of the objects is cryptic, and not meant to be accessed directly:

```
x@snp

## [[1]]
## [1] 08
##
## [[2]]
## [1] 4e
```

Fortunately, data are easily converted back into integers:

```
as.integer(x)

## [1] 0 1 1 2 0 0 1
```

The main interest of this representation is its efficiency in terms of storage. For instance:

```
dat <- sample(0:1, 1e6, replace=TRUE)
print(object.size(dat),unit="auto")

## 3.8 Mb
```

```
x <- new("SNPbin", dat, parallel=FALSE)
```

```
print(object.size(x),unit="auto")

## 123.4 Kb
```

here, we converted a million SNPs into a `SNPbin` object, which turns out to be 32 smaller than the original data. However, the information in `dat` and `x` is strictly identical:

```
identical(as.integer(x),dat)

## [1] FALSE
```

The advantage of this storage is therefore being extremely compact, and allowing to analyse big datasets using standard computers.

While `SNPbin` objects are the very mean by which we store data efficiently, in practice we need to analyze several genomes at a time. This is made possible by the class `genlight`, which relies on `SNPbin` but allows for storing data from several genomes at a time.

2.2 genlight: storage of multiple genomes

Like SNPbin, genlight is a formal S4 class. The slots of instances of this class are described by:

```
getClassDef("genlight")

## Class "genlight" [package "adegenet"]
##
## Slots:
##
## Name:           gen        n.loc    ind.names    loc.names      loc.all
## Class:         list      integer   charOrNULL   charOrNULL   charOrNULL
##
## Name:    chromosome     position       ploidy          pop        other
## Class: factorOrNULL    intOrNULL    intOrNULL factorOrNULL         list
```

As it can be seen, these objects allow for storing more information in addition to vectors of SNP frequencies. More precisely, their content is (see ?genlight for more details):

- gen: SNP data for different individuals, each stored as a SNPbin; loci have to be identical across all individuals.
- n.loc: the number of SNPs stored in the object.
- ind.names: (optional) labels for the individuals.
- loc.names: (optional) labels for the loci.
- loc.all: (optional) alleles of the loci separated by '/' (e.g. 'a/t', 'g/c', etc.).
- chromosome: (optional) a factor indicating the chromosome to which the SNPs belong.
- position: (optional) the position of each SNPs in their chromosome.
- ploidy: (optional) the ploidy of each individual.
- pop: (optional) a factor grouping individuals into 'populations'.
- other: (optional) a list containing any supplementary information to be stored with the data.

Like SNPbin object, genlight object are created using the constructor new, providing content for the slots above as arguments. When none is provided, an empty object is created:

```
new("genlight")
```

```
##  === S4 class genlight ===
##  0 genotypes,  0 binary SNPs
```

The most important information to provide is obviously the genotypes (argument `gen`); these can be provided as:

- a `list` of integer vectors representing the number of second allele at each locus.
- a `matrix` / `data.frame` of integers, with individuals in rows and SNPs in columns.
- a list of `SNPbin` objects.

Ploidy has to be consistent across loci for a given individual, but individuals do not have to have the same ploidy, so that it is possible to have hapoid, diploid, and tetraploid individuals in the same dataset; for instance:

```
x <- new("genlight", list(indiv1=c(1,1,0,1,1,0), indiv2=c(2,1,1,0,0,0),
                          toto=c(2,2,0,0,4,4)))
```

```
x

##  === S4 class genlight ===
##  3 genotypes,  6 binary SNPs
##  Ploidy statistics (min/median/max): 1 / 2 / 4
##  0 (0 %) missing data

ploidy(x)

## indiv1 indiv2   toto
##      1      2      4
```

As for `SNPbin`, `genlight` objects can be converted back to integers vectors, stored as matrices or lists:

```
as.list(x)

## $indiv1
## [1] 1 1 0 1 1 0
##
## $indiv2
## [1] 2 1 1 0 0 0
##
## $toto
## [1] 2 2 0 0 4 4

as.matrix(x)
```

```
##        [,1] [,2] [,3] [,4] [,5] [,6]
## indiv1    1    1    0    1    1    0
## indiv2    2    1    1    0    0    0
## toto      2    2    0    0    4    4
```

In practice, **genlight** objects can be handled as if they were matrices of integers as the one above returned by **as.matrix**. However, they offer the advantage of efficient storage of the information; for instance, we can simulate 50 individuals typed for 100,000 SNPs each (including occasional NAs):

```
dat <- lapply(1:50, function(i) sample(c(0,1,NA), 1e5, prob=c(.5, .499, .001),
                                       replace=TRUE))
names(dat) <- paste("indiv", 1:length(dat))
print(object.size(dat),unit="auto")

## 38.2 Mb
```

```
x <- new("genlight", dat)
```

```
print(object.size(x),unit="auto")

## 699.7 Kb

object.size(dat)/object.size(x)

## 55.8327378692808 bytes
```

here again, the storage if the data is much more efficient in **genlight** than using integers: converted data occupy 56 times less memory than the original data.

The advantage of this storage is therefore being extremely compact, and allowing to analyse very large datasets using standard computers. Obviously, usual computations demand data to be at one moment coded as numeric values (as opposed to bits). However, most usual computations can be achieved by only converting one or two genomes back to numeric values at a time, therefore keeping RAM requirements low, albeit at a possible cost of increased computational time. This however is minimized by three ways:

1. conversion routines are optimized for speed using C code.
2. using parallel computation where multicore architectures are available.
3. handling smaller objects, thereby decreasing the possibly high computational time taken by memory allocation.

While this makes implementing methods more complicated. In practice, routines are implemented so as to minimize the amount of data converted back to integers, use C code where possible, and use multiple cores if the package *parallel* is installed an multiple cores are available. Fortunately, these underlying technical issues are oblivious to the user, and one merely needs to know how to manipulate `genlight` objects using a few key functions to be able to analyze data.

3 Data handling using `genlight` objects

3.1 Using accessors

In the following, we demonstrate how to manipulate and analyse `genlight` objects. The phylosophy underlying formal (S4) classes in general, and `genlight` objects in particular, is that internal representation of the information can be complex as long as accessing this information is simple. This is made possible by decoupling storage and accession: the user is not meant to access the content of the object directly, but has to use `accessors` to retrieve or modify information.

Available accessors are documented in `?genlight`. Most of them are identical to accessors for `genind` and `genpop` objects, such as:

- `nInd`: returns the number of individuals in the object.
- `nLoc`: returns the number of loci (SNPs).
- `indNames`†: returns/sets labels for individuals.
- `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.
- `other`†: returns/sets misc information stored as a list.

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

```
dat <- lapply(1:3, function(i) sample(0:2, 10, replace=TRUE))
dat

## [[1]]
##  [1] 2 2 0 1 2 0 1 1 0 0
##
## [[2]]
##  [1] 1 1 0 0 2 0 2 1 2 0
```

```
##
## [[3]]
##  [1] 2 2 2 0 1 1 1 2 1 1

x <- new("genlight", dat, parallel=FALSE)
x

##  === S4 class genlight ===
##  3 genotypes,  10 binary SNPs
##  Ploidy: 2
##  0 (0 %) missing data

indNames(x)

## NULL

indNames(x) <- paste("individual", 1:3)
indNames(x)

## [1] "individual 1" "individual 2" "individual 3"

locNames(x)
locNames(x) <- paste("SNP",1:nLoc(x),sep=".")
as.matrix(x)

##              SNP.1 SNP.2 SNP.3 SNP.4 SNP.5 SNP.6 SNP.7 SNP.8 SNP.9 SNP.10
## individual 1     2     2     0     1     2     0     1     1     0      0
## individual 2     1     1     0     0     2     0     2     1     2      0
## individual 3     2     2     2     0     1     1     1     2     1      1
```

In addition, some specific accessors are available for `genlight` objects:

- `NA.posi`: returns the position of missing values in each individual.
- `chromosome`†: returns/sets the chromosome of each SNP.
- `chr`†: same as `chromosome` — used as a shortcut.
- `position`†: returns/sets the position of each SNP.

Accessors are meant to be clever about replacement, meaning that they try hard to prevent replacement with inconsistent values. For instance, in object `x`:

```
x

##  === S4 class genlight ===
##  3 genotypes,  10 binary SNPs
```

```
##  Ploidy: 2
##  0 (0 %) missing data
##  @loc.names: labels of the SNPs
```

if we try to set information about the chromosomes of the SNPs, the instruction:

```
chr(x) <- rep("chr-1", 7)
```

will generate an error because the provided factor does not match the number of loci (10), while:

```
chr(x) <- rep("chr-1", 10)
x

##  === S4 class genlight ===
##  3 genotypes,  10 binary SNPs
##  Ploidy: 2
##  0 (0 %) missing data
##  @chromosome: chromosome of the SNPs
##  @loc.names: labels of the SNPs

chr(x)

##  [1] chr-1 chr-1 chr-1 chr-1 chr-1 chr-1 chr-1 chr-1 chr-1 chr-1
## Levels: chr-1
```

is a valid replacement.

3.2 Subsetting the data

`genlight` objects are meant to be handled as if they were matrices of allele numbers, as returned by `as.matrix`. Therefore, subsetting can be achieved using [`idx.row , idx.col`] where `idx.row` and `idx.col` are indices for rows (individuals) and columns (SNPs). For instance, using the previous toy dataset, we try a few classical subsetting of rows and columns:

```
x

##  === S4 class genlight ===
##  3 genotypes,  10 binary SNPs
##  Ploidy: 2
##  0 (0 %) missing data
##  @chromosome: chromosome of the SNPs
##  @loc.names: labels of the SNPs

as.matrix(x)
```

```
##              SNP.1 SNP.2 SNP.3 SNP.4 SNP.5 SNP.6 SNP.7 SNP.8 SNP.9 SNP.10
## individual 1     2     2     0     1     2     0     1     1     0      0
## individual 2     1     1     0     0     2     0     2     1     2      0
## individual 3     2     2     2     0     1     1     1     2     1      1

as.matrix(x[c(1,3),])

##              SNP.1 SNP.2 SNP.3 SNP.4 SNP.5 SNP.6 SNP.7 SNP.8 SNP.9 SNP.10
## individual 1     2     2     0     1     2     0     1     1     0      0
## individual 3     2     2     2     0     1     1     1     2     1      1

as.matrix(x[, c(TRUE,FALSE)])

##              SNP.1 SNP.3 SNP.5 SNP.7 SNP.9
## individual 1     2     0     2     1     0
## individual 2     1     0     2     2     2
## individual 3     2     2     1     1     1

as.matrix(x[1:2, c(1,1,1,2,2,2,3,3,3)])

##              SNP.1 SNP.1 SNP.1 SNP.2 SNP.2 SNP.2 SNP.3 SNP.3 SNP.3
## individual 1     2     2     2     2     2     2     0     0     0
## individual 2     1     1     1     1     1     1     0     0     0
```

Moreover, one can split data into blocks of SNPs using `seploc`. This can be achieved by specifying either a number of blocks (argument `n.block`) or the size of the blocks (argument `block.size`). The function also allows for randomizing the distribution of the SNPs in the blocks (argument `random=TRUE`), which is especially useful to replace computations that cannot be achieved on the whole dataset with parallelized computations performed on random blocks (for parallelization, remove the argument `parallel=FALSE`). For instance:

```
x

##  === S4 class genlight ===
##  3 genotypes,  10 binary SNPs
##  Ploidy: 2
##  0 (0 %) missing data
##  @chromosome: chromosome of the SNPs
##  @loc.names: labels of the SNPs

as.matrix(x)

##              SNP.1 SNP.2 SNP.3 SNP.4 SNP.5 SNP.6 SNP.7 SNP.8 SNP.9 SNP.10
## individual 1     2     2     0     1     2     0     1     1     0      0
## individual 2     1     1     0     0     2     0     2     1     2      0
## individual 3     2     2     2     0     1     1     1     2     1      1
```

```
seploc(x, n.block=2, parallel=FALSE)

## $block.1
##  === S4 class genlight ===
##  3 genotypes,  5 binary SNPs
##  Ploidy: 2
##  0 (0 %) missing data
##  @chromosome: chromosome of the SNPs
##  @loc.names: labels of the SNPs
##
## $block.2
##  === S4 class genlight ===
##  3 genotypes,  5 binary SNPs
##  Ploidy: 2
##  0 (0 %) missing data
##  @chromosome: chromosome of the SNPs
##  @loc.names: labels of the SNPs

lapply(seploc(x, n.block=2, parallel=FALSE),as.matrix)

## $block.1
##              SNP.1 SNP.2 SNP.3 SNP.4 SNP.5
## individual 1     2     2     0     1     2
## individual 2     1     1     0     0     2
## individual 3     2     2     2     0     1
##
## $block.2
##              SNP.6 SNP.7 SNP.8 SNP.9 SNP.10
## individual 1     0     1     1     0      0
## individual 2     0     2     1     2      0
## individual 3     1     1     2     1      1
```

splits the data into two blocks of contiguous SNPs, while:

```
lapply(seploc(x, n.block=2, random=TRUE, parallel=FALSE),as.matrix)

## $block.1
##              SNP.5 SNP.2 SNP.4 SNP.6 SNP.7
## individual 1     2     2     1     0     1
## individual 2     2     1     0     0     2
## individual 3     1     2     0     1     1
##
## $block.2
##              SNP.3 SNP.10 SNP.8 SNP.9 SNP.1
## individual 1     0      0     1     0     2
```

```
## individual 2     0      0     1     2     1
## individual 3     2      1     2     1     2
```

generates blocks of randomly selected SNPs.

3.3 Data conversions

3.3.1 The `.snp` format

adegenet has defined its own format for storing biallelic SNP data in text files with extension `.snp`. This format has several advantages: it is fairly compact (more so than usual non-compressed formats), allows for any information about individuals or loci to be stored, allows for comments, and is easily parsed — in particular, not all information has to be read at a time, again minimizing RAM requirements for import procedures.

An example file of this format is distributed with adegenet. Once the package has been installed, the file can be accessed by typing:

```
file.show(system.file("files/exampleSnpDat.snp",package="adegenet"))
```

Otherwise, this file is also accessible from the *adegenet* website (section 'Documents'). A complete description of the `.snp` format is provided in the comment section of the file.

The structure of a `.snp` file can be summarized as follows:

- a (possibly empty) `comment section`
- `meta-information`, i.e. information about loci or individuals, stored as named vectors
- `genotypes`, stored as named vectors

The *comment section* can starts with the line:

```
>>>> begin comments - do not remove this line <<<<
```

and ends with the line:

```
>>>> end comments - do not remove this line <<<<}
```

While this section can be left empty, these two lines have to be present for the format to be valid. Each *meta-information* is stored using two lines, the first starting as:

```
>> name-of-the-information
```

and the second containing the information itself, each item separated by a single space. Any label can be used, but some specific names will be recognized and interpreted by the parser:

- `position`: the following line contains integers giving the position of the SNPs on the sequence

- `allele`: character strings representing the two alleles of each loci separated by "/"
- `population`: character strings indicating a group memberships of the individuals
- `ploidy`: integers indicating the ploidy of each individual; alternatively, one single integer if all individuals have the same ploidy
- `chromosome`: character strings indicating the chromosome on which the SNP are located

Each *genotype* is stored using two lines, the first being:

```
> label-of-the-individual
```

and the second being integers corresponding to the number of second allele for each loci, without separators; missing data are coded as '`-`'.

`.snp` files can be read in R using `read.snp`, which converts data into `genlight` objects. The function reads data by chunks of a several individuals (minimum 1, no maximum besides RAM constraints) at a time, which allows one to read massive datasets with negligible RAM requirements (albeit at a cost of computational time). The argument `chunkSize` indicates the number of genomes read at a time; larger values mean reading data faster but require more RAM. We can illustrate `read.snp` using the example file mentioned above. The non-comment part of the file reads:

```
[...]
>> position
1 8 11 43
>> allele
a/t g/c a/c t/a
>> population
Brit Brit Fren monster NA
>> ploidy
2
> foo
1020
> bar
0012
> toto
10-0
> Nyarlathotep
0120
> an even longer label but OK since on a single line
1100
```

We read the file in using:

```
obj <- read.snp(system.file("files/exampleSnpDat.snp",package="adegenet"),
                chunk=2, parallel=FALSE)

##
##  Reading biallelic SNP data file into a genlight object...
##
##
##  Reading comments...
##
##  Reading general information...
##
##  Reading 5 genotypes...
## ...
##  Checking consistency...
##
##  Building final object...
##
## ...done.

obj

##  === S4 class genlight ===
##  5 genotypes,  4 binary SNPs
##  Ploidy: 2
##  1 (0.05 %) missing data
##  @pop: individual membership for 4 populations
##  @position: position of the SNPs
##  @alleles: alleles of the SNPs

as.matrix(obj, parallel=FALSE)

##                                                      1.a/t 8.g/c 11.a/c
## foo                                                      1     0      2
## bar                                                      0     0      1
## toto                                                     1     0     NA
## Nyarlathotep                                             0     1      2
## an even longer label but OK since on a single line       1     1      0
##                                                      43.t/a
## foo                                                       0
## bar                                                       2
## toto                                                      0
## Nyarlathotep                                              0
## an even longer label but OK since on a single line        0

alleles(obj)
```

```
## [1] "a/t" "g/c" "a/c" "t/a"

obj@pop

## [1] Brit    Brit    Fren    monster NA
## Levels: Brit Fren monster NA

indNames(obj)

## [1] "foo"
## [2] "bar"
## [3] "toto"
## [4] "Nyarlathotep"
## [5] "an even longer label but OK since on a single line"
```

Note that `system.file` is generally useless: it is only used in this example to access a file installed alongside the package. Usual calls to `read.snp` will ressemble:

```
obj <- read.snp("path-to-my-file.snp")
```

3.3.2 Importing data from PLINK

Genome-wide SNP data of diploid organisms are frequently analyzed using PLINK, whose format is therefore becoming a standard. Data with PLINK format (`.raw`) can be imported into `genlight` objects using `read.PLINK`. This function requires the data to be saved in PLINK using the '*-recodeA*' option (see details section in `?read.PLINK`). More information on exporting from PLINK can be found at http://pngu.mgh.harvard.edu/~purcell/plink/dataman.shtml#recode.

Like `read.snp`, `read.PLINK` has the advantage of reading data by chunks of a few individuals (down to a single one at a time, no upper limits), which minimizes the amount of memory needed to read information before its conversion to `genlight`; however, using more chunks also means more computational time, since the procedure has to re-read the same file several time. Note that meta information about the loci also known as `.map` can also be read alongside a `.raw` file using the argument `map.file`. Alternatively, such information can be added to a `genlight` object afterwards using `extract.PLINKmap`.

3.3.3 Extracting SNPs from alignments

In many cases, raw genomic data are available as aligned sequences, in which case extracting polymorphic sites can be non-trivial. The biggest issue is again memory: most software extracting SNPs from aligned sequences require all the sequences to be stored in memory at a time, a duty that most common computers cannot undertake. *adegenet* implements a more parsimonious alternative which allows for extracting SNPs from alignment while

processing a reduced number of sequences (down to a single) at a time.

The function `fasta2genlight` extracts SNPs from alignments with *fasta* format (file extensions '`.fasta`', '`.fas`', or '`.fa`'). Like `read.snp` and `read.PLINK`, `fasta2genlight` processes data by chunks of individuals so as to save memory requirements. It first scans the whole file for polymorphic positions, and then extracts all biallelic SNPs from the alignment.

`fasta2genlight` is illustrated like `read.snp` using a toy dataset distributed alongside the package. The file is first located using `system.file`, and then processed using `fasta2genlight`:

```
myPath <- system.file("files/usflu.fasta",package="adegenet")
flu <- fasta2genlight(myPath, chunk=10, parallel=FALSE)

##
##  Converting FASTA alignment into a genlight object...
##
##
##  Looking for polymorphic positions...
## ..........................................................................
##  Extracting SNPs from the alignment...
## ..........................................................................
##  Building final object...
##
## ...done.

flu

##  === S4 class genlight ===
##  80 genotypes,  274 binary SNPs
##  Ploidy: 1
##  26 (0 %) missing data
##  @position: position of the SNPs
##  @alleles: alleles of the SNPs
```

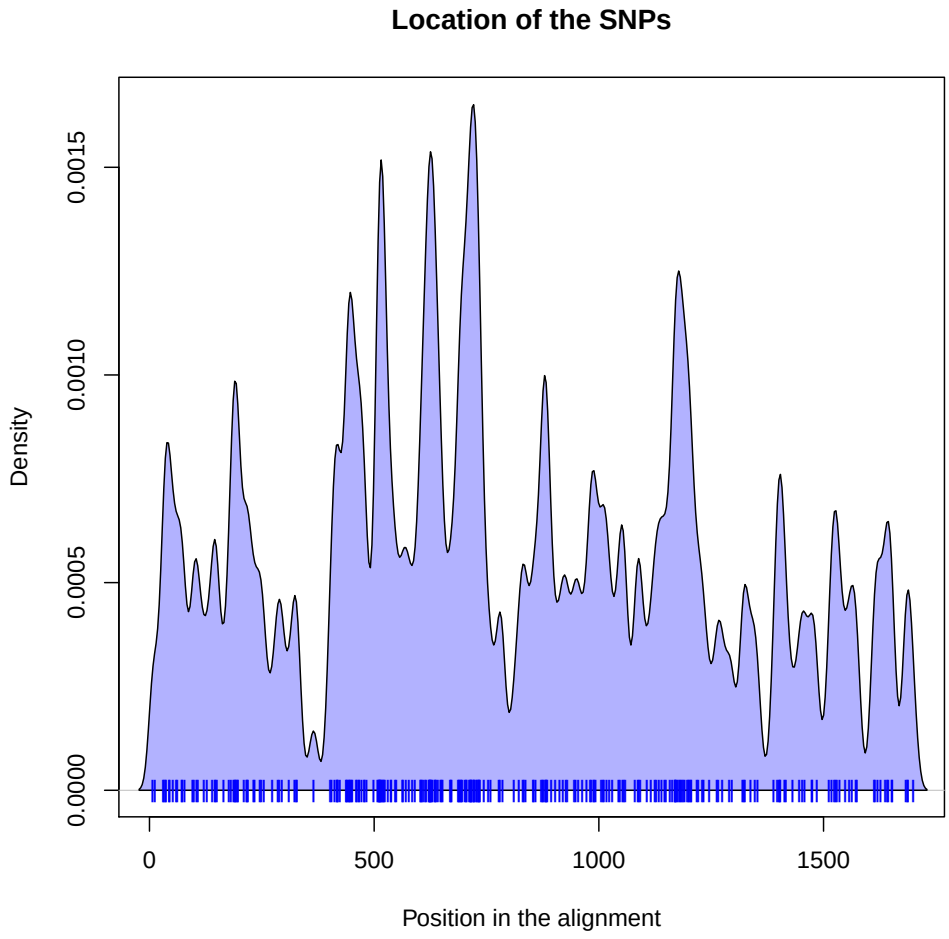
`flu` is a `genlight` object containing SNPs of 80 isolates of seasonal influenza (H3N2) sampled within the US over the last two decades; sequences correspond to the hemagglutinin (HA) segment. Besides genotypes, `flu` contains the positions of the SNPs and the alleles at each retained loci. Names of the loci are constructed as the combination of both:

```
head(position(flu), 20)

##  [1]   7  12  31  32  36  37  44  45  52  60  62  72  73  78  96  99 105
## [18] 108 121 128

head(alleles(flu), 20)
```

```
##  [1] "a/g" "c/t" "t/c" "t/c" "t/c" "c/a" "t/c" "c/t" "a/g" "c/t" "g/t"
## [12] "c/a" "a/g" "a/g" "a/g" "c/t" "a/g" "g/a" "c/a" "a/g"

head(locNames(flu), 20)

##  [1] "7.a/g"   "12.c/t"  "31.t/c"  "32.t/c"  "36.t/c"  "37.c/a"  "44.t/c"
##  [8] "45.c/t"  "52.a/g"  "60.c/t"  "62.g/t"  "72.c/a"  "73.a/g"  "78.a/g"
## [15] "96.a/g"  "99.c/t"  "105.a/g" "108.g/a" "121.c/a" "128.a/g"
```

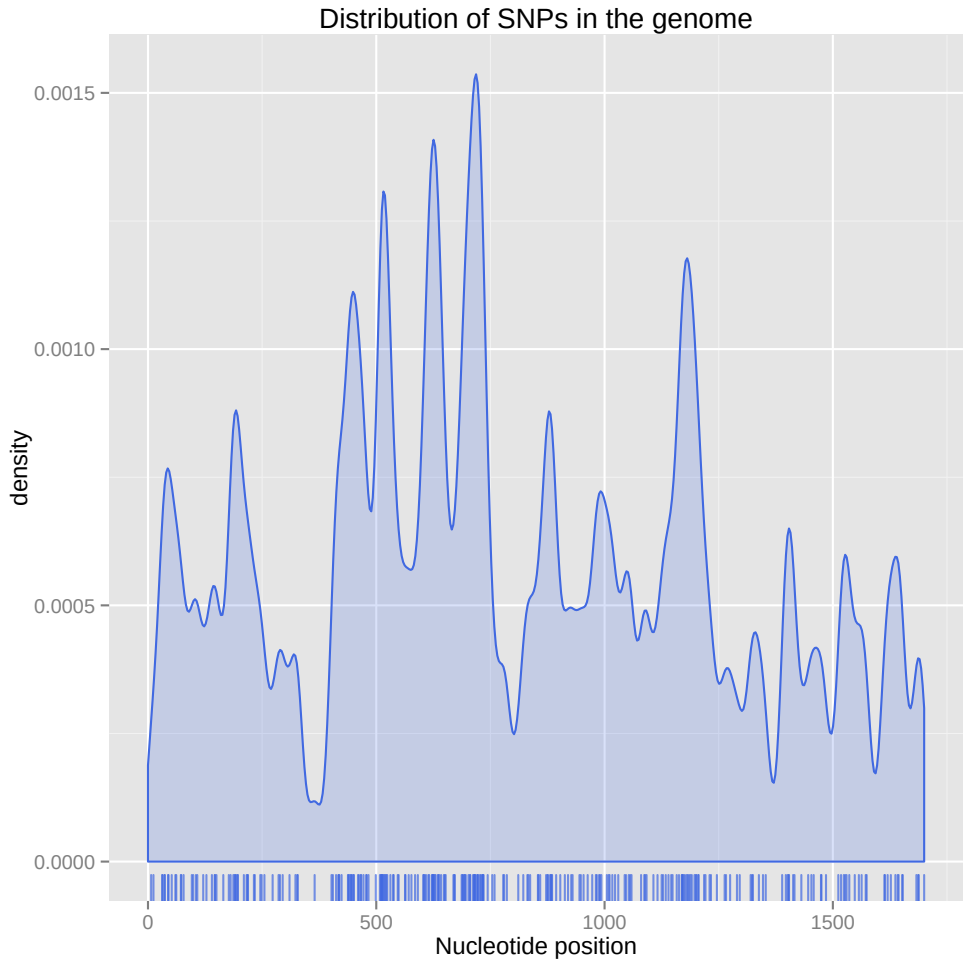
It is usually informative to assess the position of the polymorphic sites within the genome; this is very easily done in R, using `density` with an appropriate bandewidth:

```
temp <- density(position(flu), bw=10)
plot(temp, type="n", xlab="Position in the alignment",
     main="Location of the SNPs", xlim=c(0,1701))
polygon(c(temp$x,rev(temp$x)), c(temp$y, rep(0,length(temp$x))),
        col=transp("blue",.3))
points(position(flu), rep(0, nLoc(flu)), pch="|", col="blue")
```

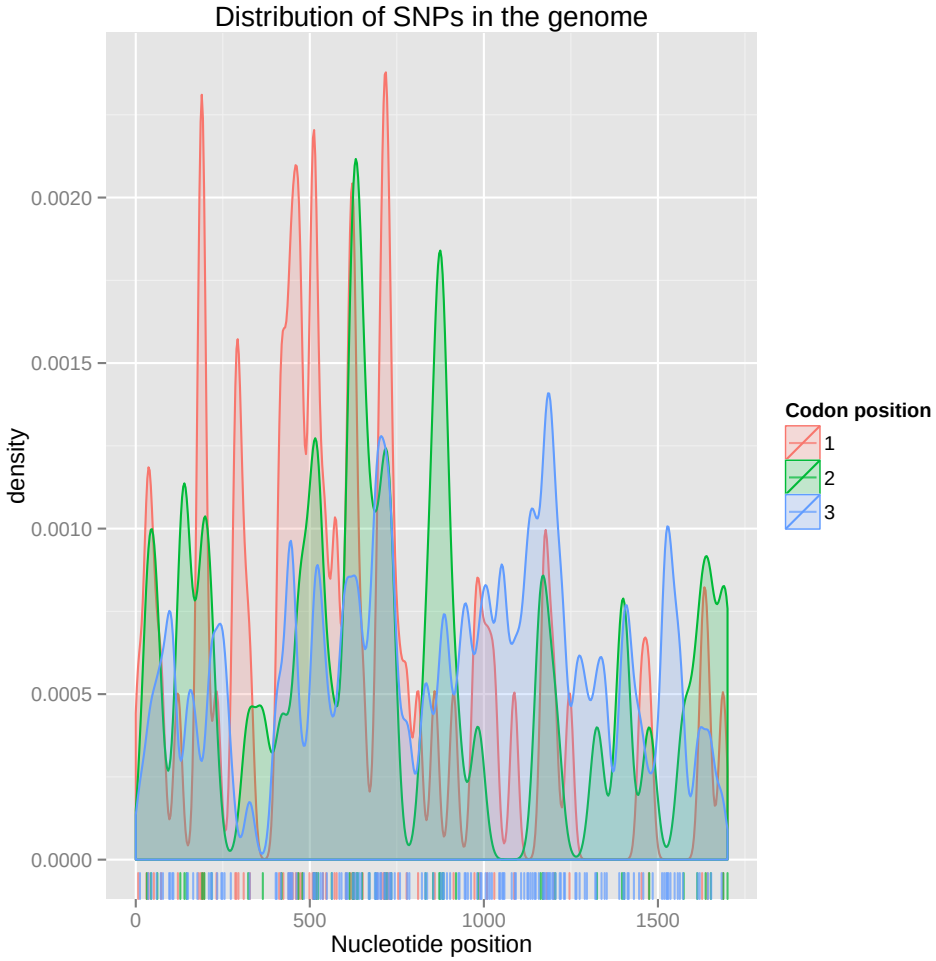


As of *adegenet* 1.4-0, this approach is available in `snpposi.plot`, which also allow, as an option, to represent density by codon position:

```
snpposi.plot(position(flu), genome.size=1700, codon=FALSE)
```




```
snpposi.plot(position(flu), genome.size=1700)
```



In this case, SNPs seem to be distributed fairly homogeneously across the HA segment, with a few possible hotspots of polymorphism within positions 400—700. This can be tested by `snpposi.test`:

```
snpposi.test(position(flu), genome.size=1700)

## Monte-Carlo test
## Call: as.randtest(sim = sim, obs = obs, alter = "less")
##
## Observation: 2
##
## Based on 999 replicates
## Simulated p-value: 0.492
## Alternative hypothesis: less
##
##     Std.Obs Expectation    Variance
##     -0.9984      2.4900      0.2409
```

Note that retaining only biallelic sites may cause minor loss of information, as sites with

more than 2 alleles are discarded from the data. It is however possible to ask `fasta2genlight` to keep track of the number of alleles for each site of the original alignment, by specifying:

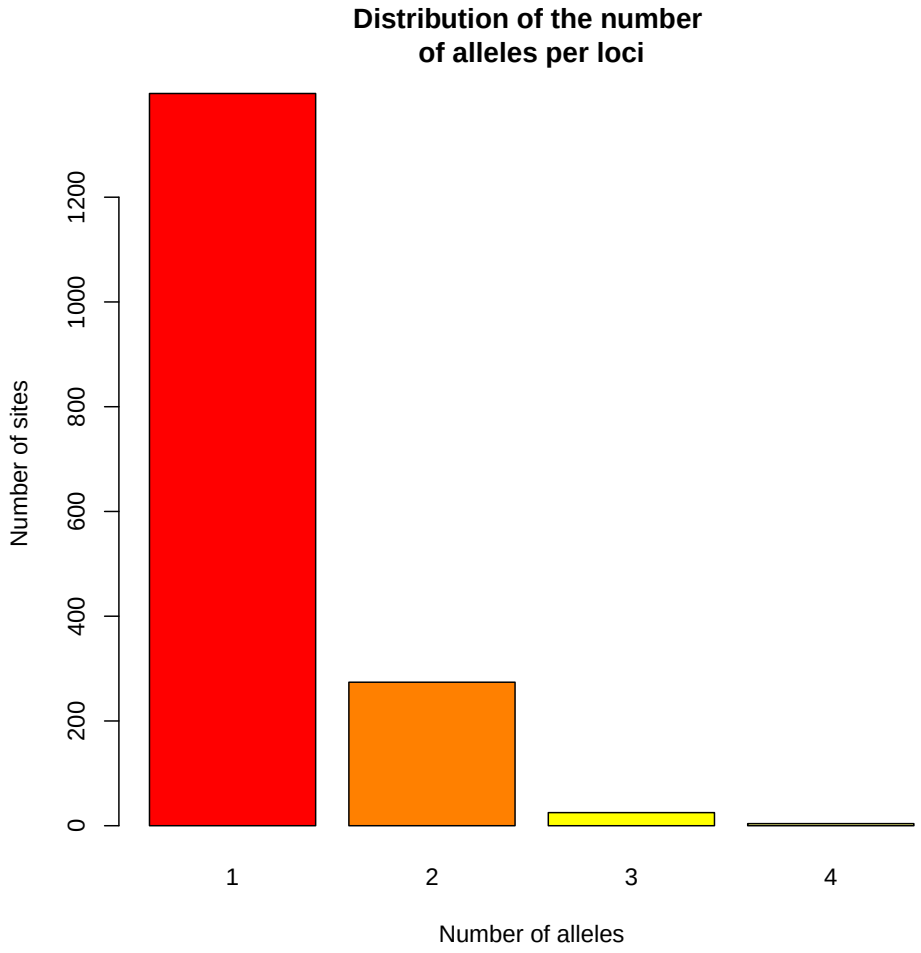
```
flu <- fasta2genlight(myPath, chunk=10,saveNbAlleles=TRUE, quiet=TRUE,
                      parallel=FALSE)
flu

##  === S4 class genlight ===
##  80 genotypes,  274 binary SNPs
##  Ploidy: 1
##  26 (0 %) missing data
##  @position: position of the SNPs
##  @alleles: alleles of the SNPs
##  @other: a list containing: nb.all.per.loc
```

The output object `flu` now contains the number of alleles of each position, stored in the `other` slot:

```
head(other(flu)$nb.all.per.loc, 20)

##  [1] 1 1 1 1 1 1 2 1 1 1 1 2 1 1 1 1 1 1 1 1

100*mean(unlist(other(flu))>1)

## [1] 17.81
```

About 18% of the sites are polymorphic, which is fairly high. This is not entirely surprising, given that the HA segment of influenza is known for its high mutation rate. What is the nature of this polymorphism?

```
temp <- table(unlist(other(flu)))
barplot(temp, main="Distribution of the number \nof alleles per loci",
        xlab="Number of alleles", ylab="Number of sites", col=heat.colors(4))
```



Most polymorphic loci are biallelic, but a few loci with 3 or 4 alleles were lost. We can estimate the loss of information very simply:

```
temp <- temp[-1]
temp <- 100*temp/sum(temp)
round(temp,1)

##
##    2    3    4
## 90.4  8.3  1.3
```

In this case, 90.4% of the polymorphic sites were biallelic, the others being essentially triallelic. This is probably a fairly exceptional situation due to the high mutation rate of the HA segment.

4 Data analysis using `genlight` objects

In the following, we illustrate some methods for the analysis of `genlight` objects, ranging from simple tools for diagnosing allele frequencies or missing data to recently developed

multivariate approaches. Some examples below are illustrated using toy datasets generated using the function `glSim`.

4.1 Simulating genlight objects

`glSim` is a simple tool for simulating SNPs datasets. As `glSim` returns simulated data in the form of `genlight` objects, it allows the user to generate large SNPs matrices in a compact way, and provides customisable 'slots' to which the user can append additional information. The arguments used by textttglSim are as follows:

```
args(glSim)

## function (n.ind, n.snp.nonstruc, n.snp.struc = 0, grp.size = c(0.5,
##     0.5), k = NULL, pop.freq = NULL, ploidy = 1, alpha = 0, parallel = FALSE,
##     LD = TRUE, block.minsize = 10, block.maxsize = 1000, theta = NULL,
##     sort.pop = FALSE, ...)
## NULL
```

The new version of `glSim` contains new optional arguments to allow for greater complexity in the simulated data.

Contrasting structures between two groups of individuals can be generated by specifying the parameters `n.snp.struc`, `grp.size`, and `alpha`, an asymmetry parameter enforcing differences between groups (which are strongest when alpha = 0.5 and weakest when alpha = 0). The resultant factorisation of individuals into these two groups will occupy the slot `pop` of the `genlight` object created.

In addition, the user has the option to simulate background group structure between `k` ancestral populations of relative size `pop.freq`, generated by altering allele frequencies in the 'non-structural' SNPs. The factorisation of individuals into these `k` groups will then occupy the slot `@other$ancestral.pops`. If the user wishes to sort the genotypes according to the ancestral populations (rather than the dichotomous structural groups), this can be accomplished by setting `sort.pop = TRUE`.

The non-structural SNPs, with or without ancestral population structure, can also be generated with or without linkage disequilibrium (LD). The arguments `LD = TRUE`, `block.minsize`, `block.maxsize`, and `theta`, a dilution parameter (set to 0 for strongest LD and 0.5 for weakest-but-present LD).

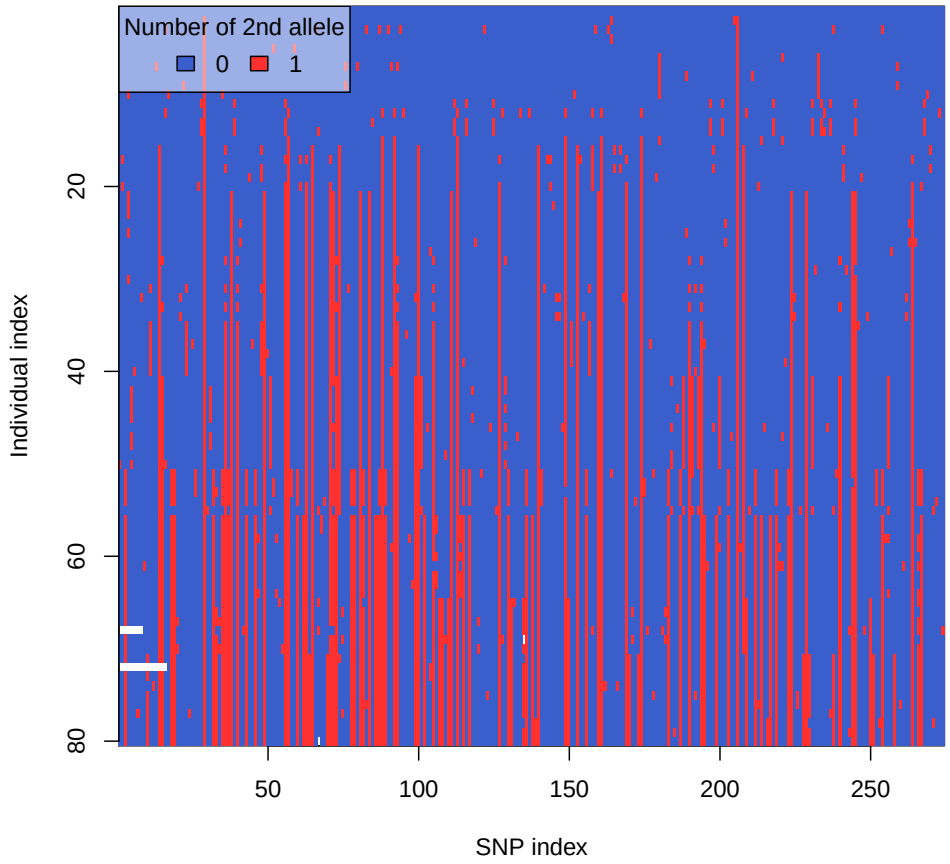
See `?glSim` for more details.

4.2 Basic analyses

4.2.1 Plotting `genlight` objects

Basic features of the data may also be inferred by simply looking at the data. `genlight` objects can be plotted using `glPlot`, or simply `plot` (both names actually correspond to the same function). This function displays the data as images, representing numbers of second alleles using colours. For instance, we can have a feel for the amount and location of missing data in the influenza dataset (see previous section) fairly easily:

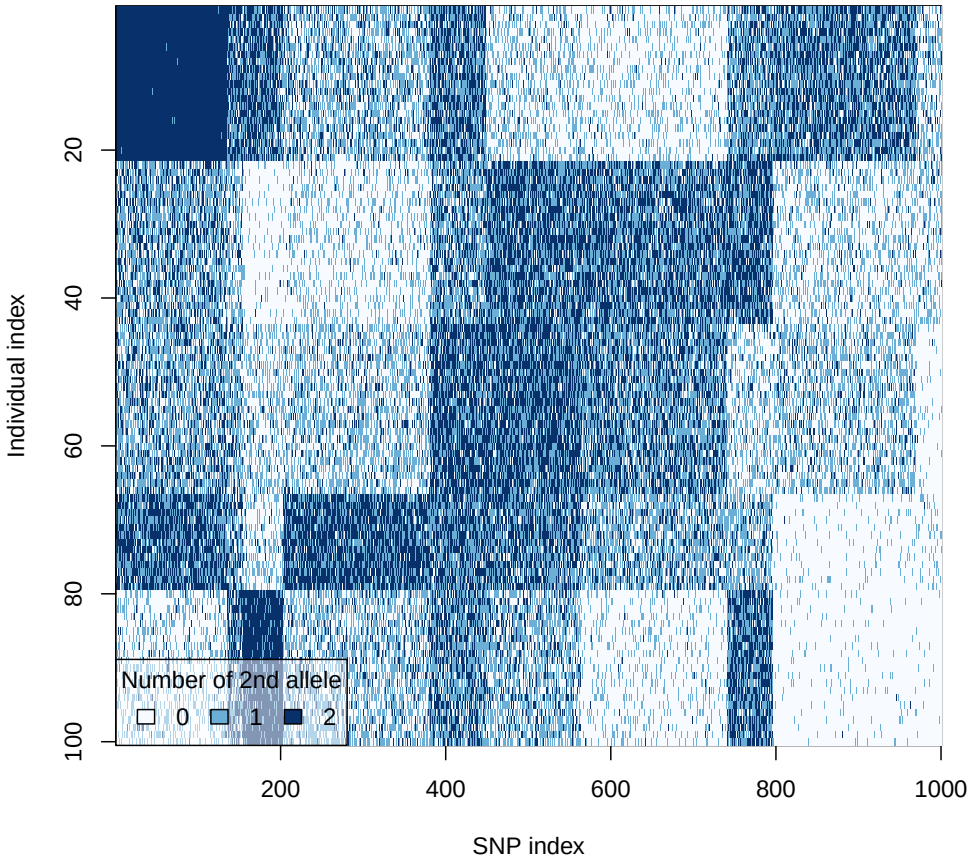
```
glPlot(flu, posi="topleft")
```



The white streches in the first 30 SNPs observed around individual 70 indicate missing data. There are only a few missing data, and they only concern a couple of individuals.

In some simple cases, some biological structures might also be apparent in such plot. For instance, we can generate data for 100 diploid individuals belonging to 5 separate populations (i.e. with independent allele frequencies):

```
x <- glSim(100, 1000, k=5, block.maxsize=200, ploidy=2,
           sort.pop=TRUE)
glPlot(x, col=bluepal(3))
```



Note that both population structures (vertical blocks) and patterns of LD between contiguous sites (horizontal blocks) are easy to spot on the above figure. Of course, data visualization merely is a preliminary approach to the data. More detailed analysis can be achieved using both standard and *ad hoc* procedures as detailed below.

4.2.2 `genlight`-optimized routines

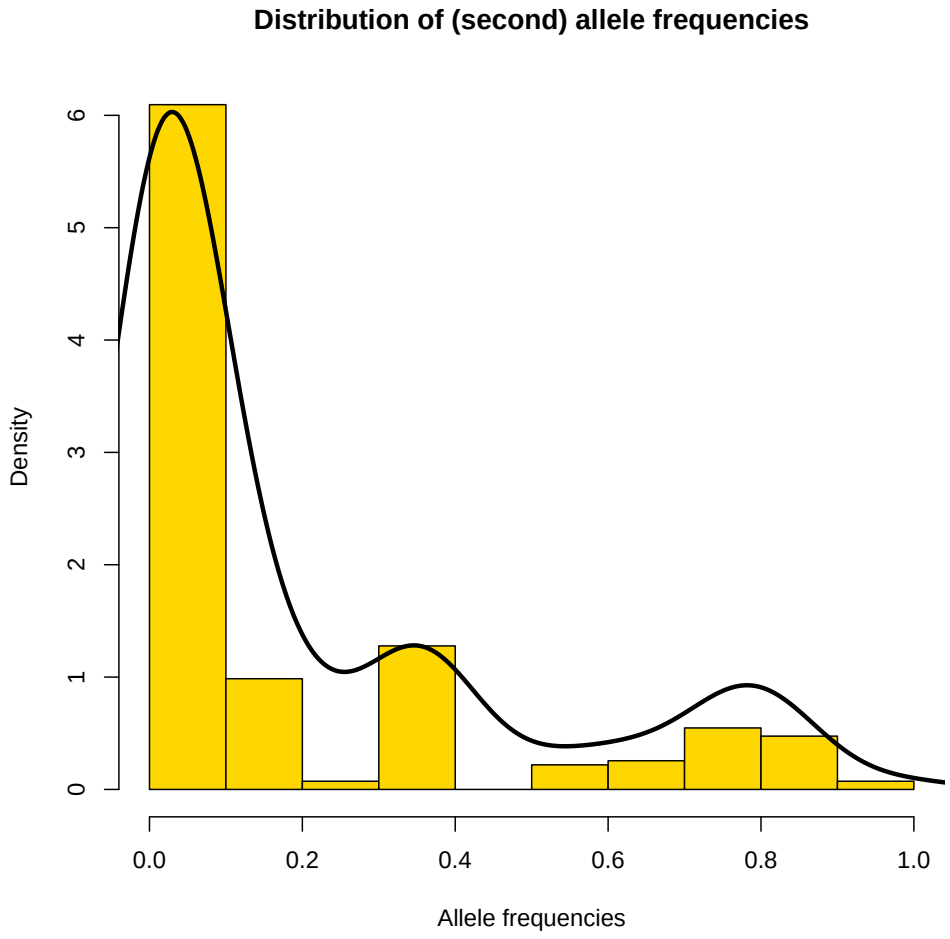
Some simple operations such as computing allele frequencies or diagnosing missing values can be problematic when the data matrix cannot be represented in memory. *adegenet* implements a few basic procedures which perform such basic tasks on `genlight` objects processing one individual at a time, thereby minimizing memory requirements. The most computer-intensive of these procedures can also use compiled C code and/or multicore capabilities (when available) to speed up computations.

All these procedures are named using the prefix `gl` (for `genlight`), and can therefore be listed by typing `gl` and pressing the TAB key twice. They are (see `?glMean`):

- `glSum`: computes the sum of second alleles for each SNP.
- `glNA`: computes the number of missing values in each locus.
- `glMean`: computes the mean of second alleles, i.e. second allele frequencies for each SNP.
- `glVar`: computes the variance of the second allele frequency for each SNP.
- `glDotProd`: computes the dot products between all pairs of individuals, with possible centring and scaling.

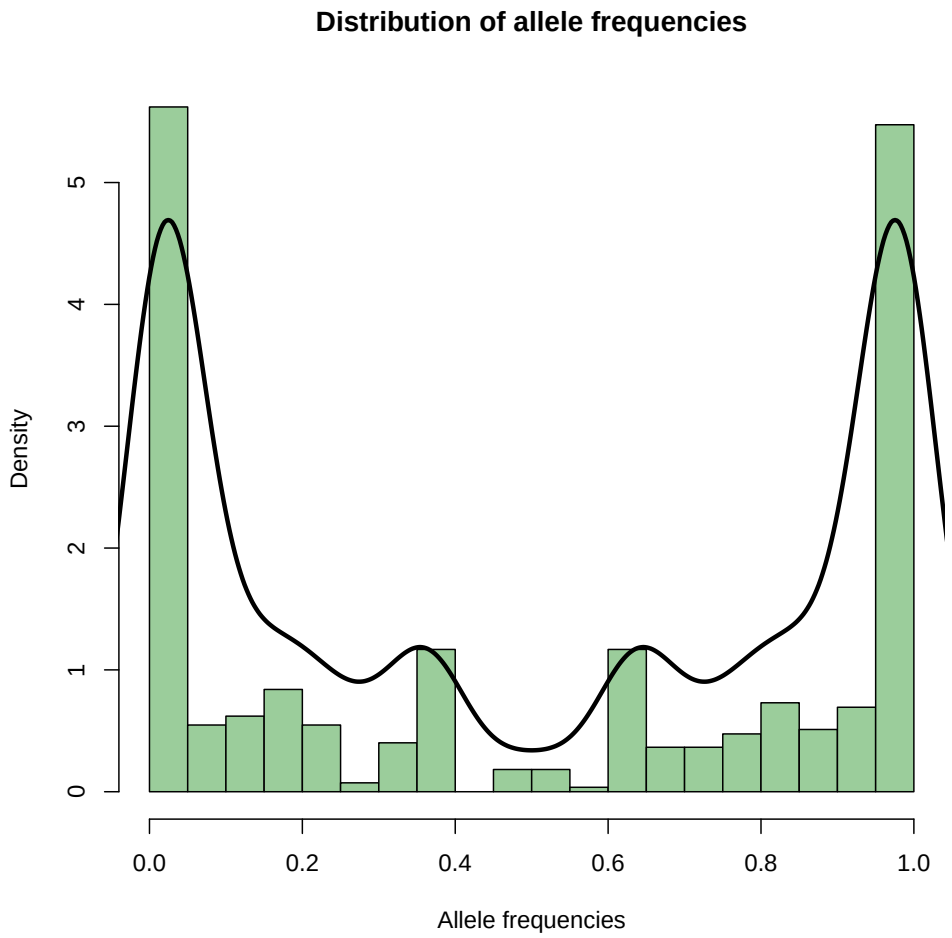
For instance, one can easily derive the distributiong of allele frequencies using:

```
myFreq <- glMean(flu)
hist(myFreq, proba=TRUE, col="gold", xlab="Allele frequencies",
     main="Distribution of (second) allele frequencies")
temp <- density(myFreq)
lines(temp$x, temp$y*1.8,lwd=3)
```



In biallelic loci, one allele is always entirely redundant with the other, so it is generally sufficient to analyse a single allele per loci. However, the distribution of allele frequencies may be more interpretable by restoring its native symmetry:

```
myFreq <- glMean(flu)
myFreq <- c(myFreq, 1-myFreq)
hist(myFreq, proba=TRUE, col="darkseagreen3", xlab="Allele frequencies",
     main="Distribution of allele frequencies", nclass=20)
temp <- density(myFreq, bw=.05)
lines(temp$x, temp$y*2,lwd=3)
```

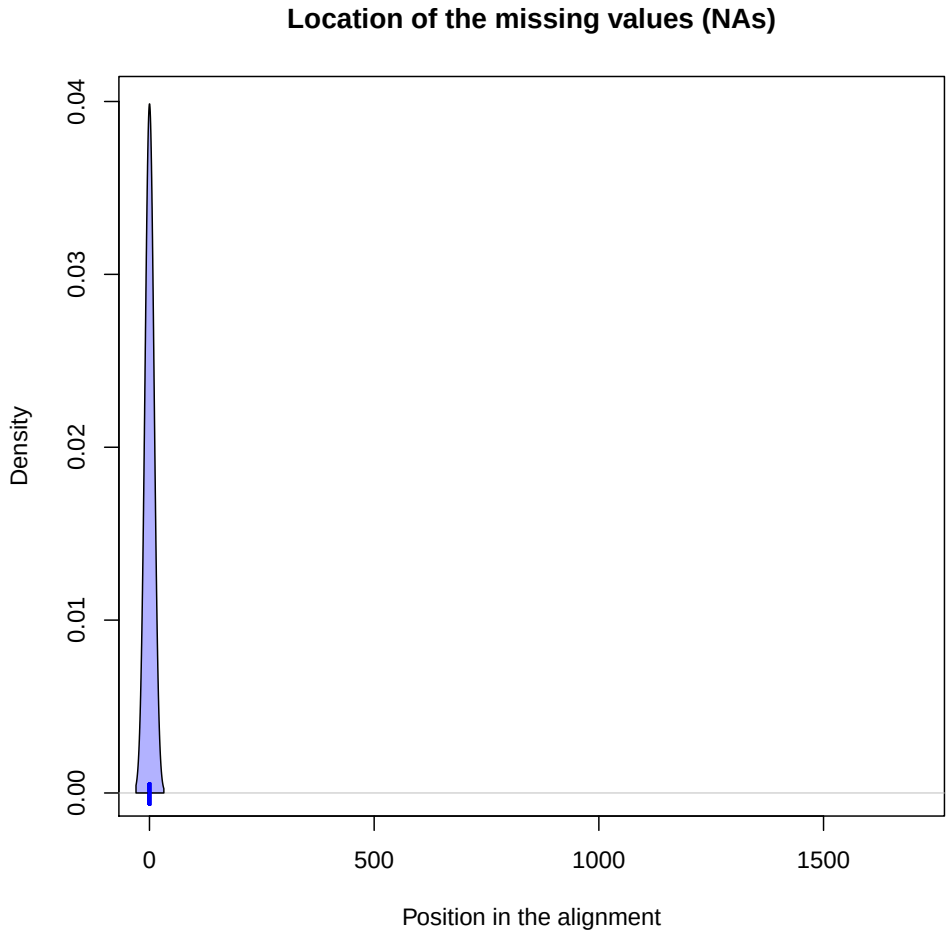


While a large number of loci are nearly fixed (frequencies close to 0 or 1), there is an appreciable number of alleles with intermediate frequencies and therefore susceptible to contain interesting biological signal. More generally and perhaps more importantly, this figure may also cast light on a well-known social phenomenon occuring mainly in young people attending noisy kinds of conferences:

We can indeed wonder whether the gesture usually referred to as the '*devil sign*' is not actually a reference to the usual shape of SNPs frequency distributions. It is still unclear, however, how many geneticists do attend metal gigs, although recent observations suggest they would be more frequent in grindcore events than in classical heavy metal shows.

Besides these considerations, we can also map missing data across loci as we have done for SNP positions in the US influenza dataset (see previous section) using `glNA` and `density`:

```
head(glNA(flu),20)

##   7.a/g  12.c/t  31.t/c  32.t/c  36.t/c  37.c/a  44.t/c  45.c/t  52.a/g
##       2       2       2       2       2       2       2       2       1
##  60.c/t  62.g/t  72.c/a  73.a/g  78.a/g  96.a/g  99.c/t 105.a/g 108.g/a
##       1       1       1       1       1       1       1       0       0
## 121.c/a 128.a/g
##       0       0

temp <- density(glNA(flu), bw=10)
plot(temp, type="n", xlab="Position in the alignment", main="Location of the missing val
     xlim=c(0,1701))
polygon(c(temp$x,rev(temp$x)), c(temp$y, rep(0,length(temp$x))), col=transp("blue",.3))
points(glNA(flu), rep(0, nLoc(flu)), pch="|", col="blue")
```



Here, the few missing values are all located at the beginning at the alignment, probably reflecting heterogeneity in DNA amplification during the sequencing process. In larger datasets, such simple investigation can give crucial insights about the quality of the data and the existence of possible sequencing biases.

4.2.3 Analysing data per block

Some operations such as computations of distances between individuals can also be useful, and have yet to be implemented for `genlight` objects. These operations are easy to carry out by converting data to alleles counts (using `as.matrix`), but this conversion itself can be problematic because of memory limitations. One easy workaround consists in parallelizing computations across blocks of loci. `seploc` is first used to create a list of smaller `genlight` objects, each of which can individually be converted to absolute allele frequencies using `as.matrix`. Then, computations are carried on the list of object, without ever having to convert the entire dataset, and results are finally reunited.

Let us illustrate this procedure using 40 simulated individuals with 10,000 SNPs each:

```
x <- glSim(40, 1e4, LD=FALSE, parallel=FALSE)
x

##  === S4 class genlight ===
##  40 genotypes,  10000 binary SNPs
##  Ploidy: 1
##  0 (0 %) missing data
##  @other: a list containing: ancestral.pops
```

`seploc` is used to create a list of smaller objects (here, 10 blocks of 10,000 SNPs):

```
x <- seploc(x, n.block=10, parallel=FALSE)
class(x)

## [1] "list"

names(x)

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

x[1:2]

## $block.1
##  === S4 class genlight ===
##  40 genotypes,  1000 binary SNPs
##  Ploidy: 1
##  0 (0 %) missing data
##  @other: a list containing: ancestral.pops
##
##
## $block.2
##  === S4 class genlight ===
##  40 genotypes,  1000 binary SNPs
##  Ploidy: 1
##  0 (0 %) missing data
##  @other: a list containing: ancestral.pops
```

`dist` is used within a `lapply` loop to compute pairwise distances between individuals for each block:

```
lD <- lapply(x, function(e) dist(as.matrix(e)))
class(lD)

## [1] "list"
```

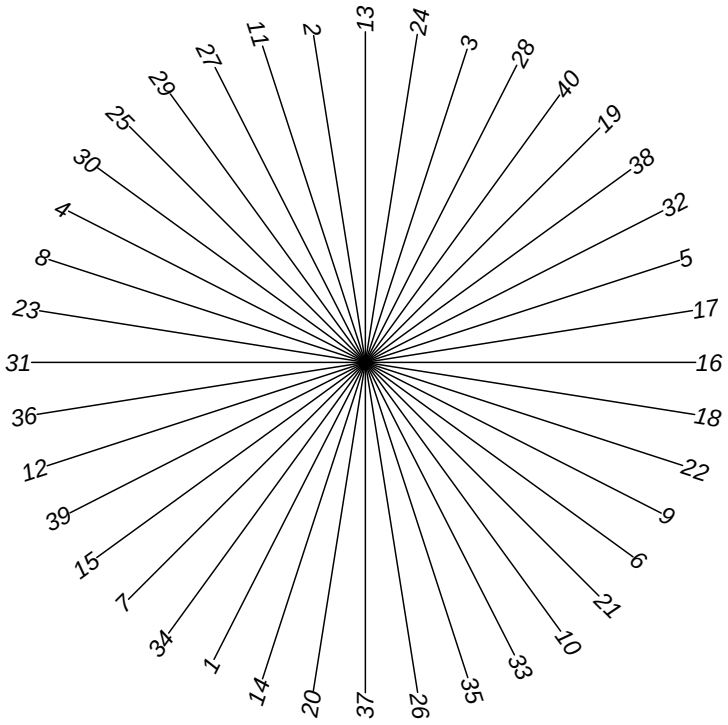
```
names(lD)

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

class(lD[[1]])

## [1] "dist"
```

`lD` is a list of distances matrices (`dist` objects) between pairs of individuals. The general distance matrix is obtained by summing these:

```
D <- Reduce("+", lD)
```

And we could now carry on further analyses, such as a neighbor-joining tree using the *ape* package:

```
library(ape)
plot(nj(D), type="fan")
title("A simple NJ tree of simulated genlight data")
```



4.2.4 What is the unit of observation?

Whenever ploidy varies across individuals, an issue arises as to what is defined as the *unit of observation*. Technically speaking, the unit of observation is the entity on which the observation is made. When working with allelic data, it is not always clear what the unit of observation is. The unit of observation may be:

- *individuals*: in this case each individual is represented by a vector of allele frequencies
- *alleles*: in this case we consider that each individual represents a sample of alleles, with a sample size equalling the ploidy for each locus

This distinction is most of the time overlooked when analysing genetic data. As a matter of fact, it does not matter when all individuals have the same ploidy. For instance, if we take the following data:

```
x <- new("genlight", list(a=c(0,0,1,1), b=c(1,1,0,0), c=c(1,1,1,1)),
         parallel=FALSE)
locNames(x) <- 1:4
```

```
x

##  === S4 class genlight ===
##  3 genotypes,  4 binary SNPs
##  Ploidy: 1
##  0 (0 %) missing data
##  @loc.names: labels of the SNPs

as.matrix(x)

##   1 2 3 4
## a 0 0 1 1
## b 1 1 0 0
## c 1 1 1 1
```

and assume that all individuals are haploid, then computing e.g. the allele frequencies is straightforward (they all equal 2/3):

```
glMean(x)

##      1      2      3      4
## 0.6667 0.6667 0.6667 0.6667
```

Let us no consider a sightly different case:

```
x <- new("genlight", list(a=c(0,0,2,2), b=c(1,1,0,0), c=c(1,1,1,1)),
         parallel=FALSE)
locNames(x) <- 1:4
x

##  === S4 class genlight ===
##  3 genotypes,  4 binary SNPs
##  Ploidy statistics (min/median/max): 1 / 1 / 2
##  0 (0 %) missing data
##  @loc.names: labels of the SNPs

as.matrix(x)

##   1 2 3 4
## a 0 0 2 2
## b 1 1 0 0
## c 1 1 1 1

ploidy(x)

## a b c
## 2 1 1
```

What are the allele frequencies in this case? Well, it depends on what we mean by '*allele frequency*'.

Is it the frequency of the alleles in the population? In this case, the unit of observation is the allele. We have a total of 4 samples for each loci, (since 'a' is diploid, it represents actually two samples) and the frequencies are 1/2, 1/2, 3/4, 3/4. Note, however, that this assumes that alleles are randomly associated within individuals (pangamy).

Or is it the frequency of the alleles within the individuals? In this case, the unit of observation is the individual, and the vector of allele frequencies represents the 'average individual'. We first need to convert each individual vector into relative frequencies (i.e., divide by their respective ploidy), and then compute the average frequency across individuals, which ends up with 2/3 for each locus:

```
M <- as.matrix(x)/ ploidy(x)
apply(M,2,mean)

##      1      2      3      4
## 0.6667 0.6667 0.6667 0.6667
```

The procedures designed for `genlight` objects seen above (`glMean`, `glNA`, etc.) allow for this distinction to be made. The option `alleleAsUnit` is a logical indicating whether the observation unit is the allele (`TRUE`, default) or the individual (`FALSE`). For instance:

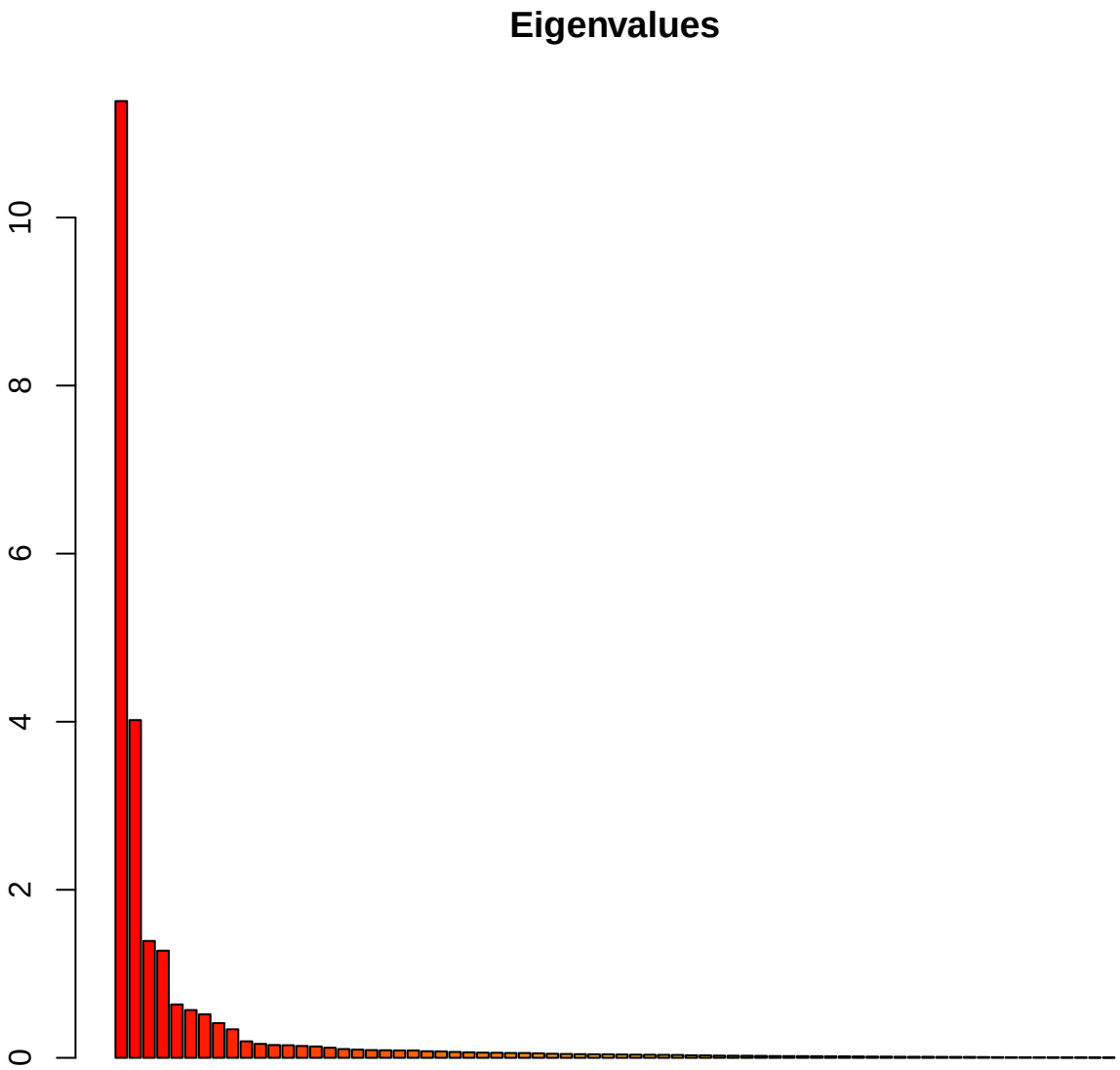
```
as.matrix(x)

##   1 2 3 4
## a 0 0 2 2
## b 1 1 0 0
## c 1 1 1 1

glMean(x, alleleAsUnit=TRUE)

##    1    2    3    4
## 0.50 0.50 0.75 0.75

glMean(x, alleleAsUnit=FALSE)

##      1      2      3      4
## 0.6667 0.6667 0.6667 0.6667
```

4.3 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is implemented for `genlight` objects by the function `glPca`. This function can accommodate any level of ploidy in the data (including varying ploidy across individuals). More importantly, it performs computations without

ever processing more than a couple of genomes at a time, thereby minimizing memory requirements. It also uses compiled C code and possibly multicore ressources if available to speed up computations. We illustrate the method on the previously introduced influenza dataset (object `flu`):

```
pca1 <- glPca(flu)
```



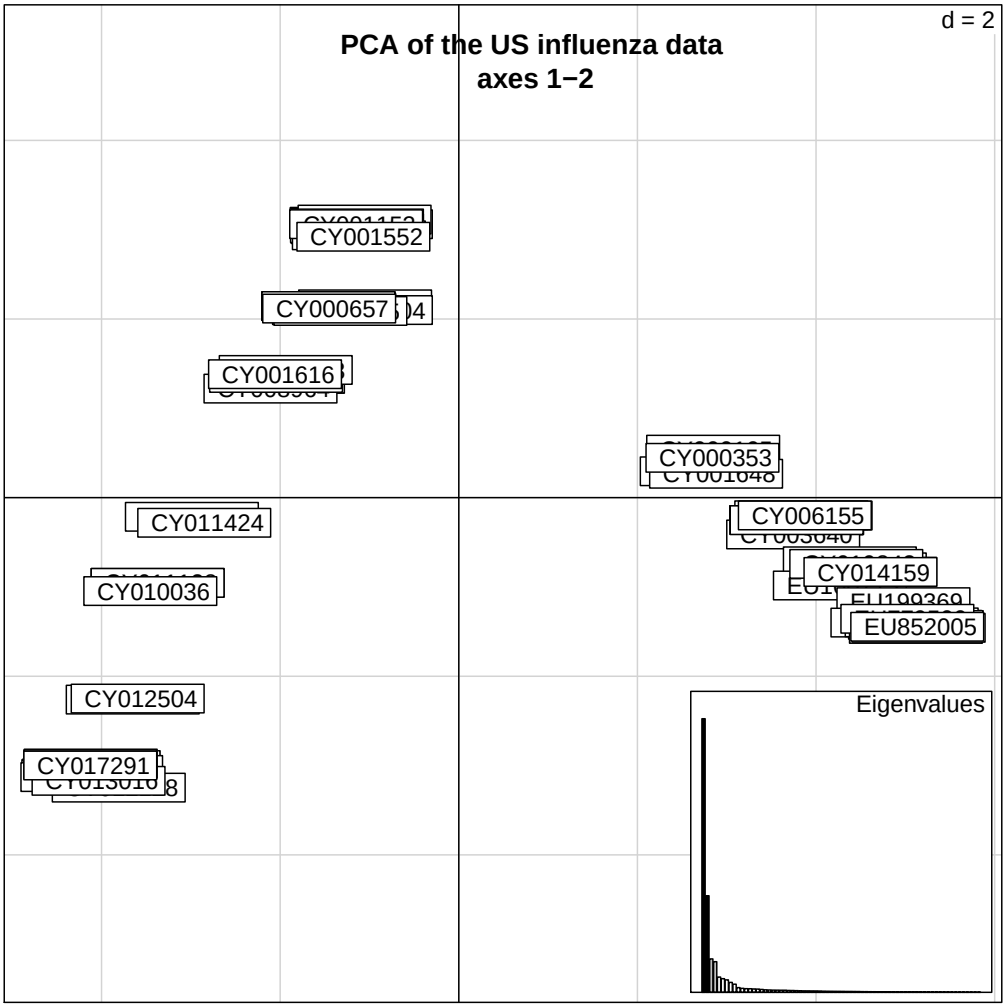
When `nf` (number of retained factors) is not specified, the function displays the barplot of eigenvalues of the analysis and asks the user for a number of retained principal components.

`glPca` returns a list with the class `glPca` containing the eigenvalues, principal components and loadings of the analysis:

```
pca1

##  === PCA of genlight object ===
## Class: list of type glPca
## Call ($call):glPca(x = flu)
##
## Eigenvalues ($eig):
##  11.38 4.019 1.391 1.275 0.636 0.569 ...
##
## Principal components ($scores):
##  matrix with 80 rows (individuals) and 4 columns (axes)
##
## Principal axes ($loadings):
##  matrix with 274 rows (SNPs) and 4 columns (axes)
```

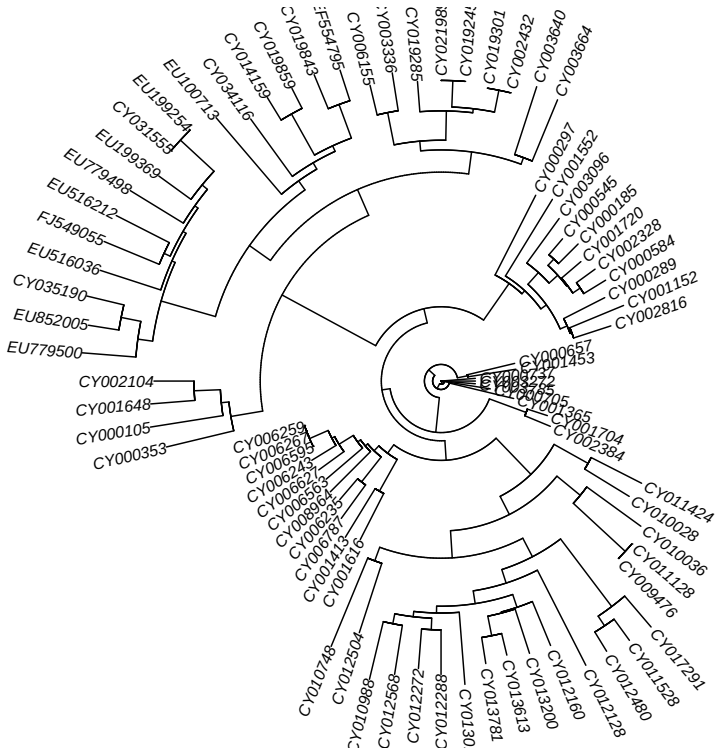
In addition to usual graphics, `glPca` object can displayed using `scatter` (produces a scatterplot of the principal components (PCs)) and `loadingplot` (plots the allele contributions, i.e. squared loadings). The scatterplot is obtained by:

```
scatter(pca1, posi="bottomright")
title("PCA of the US influenza data\n axes 1-2")
```



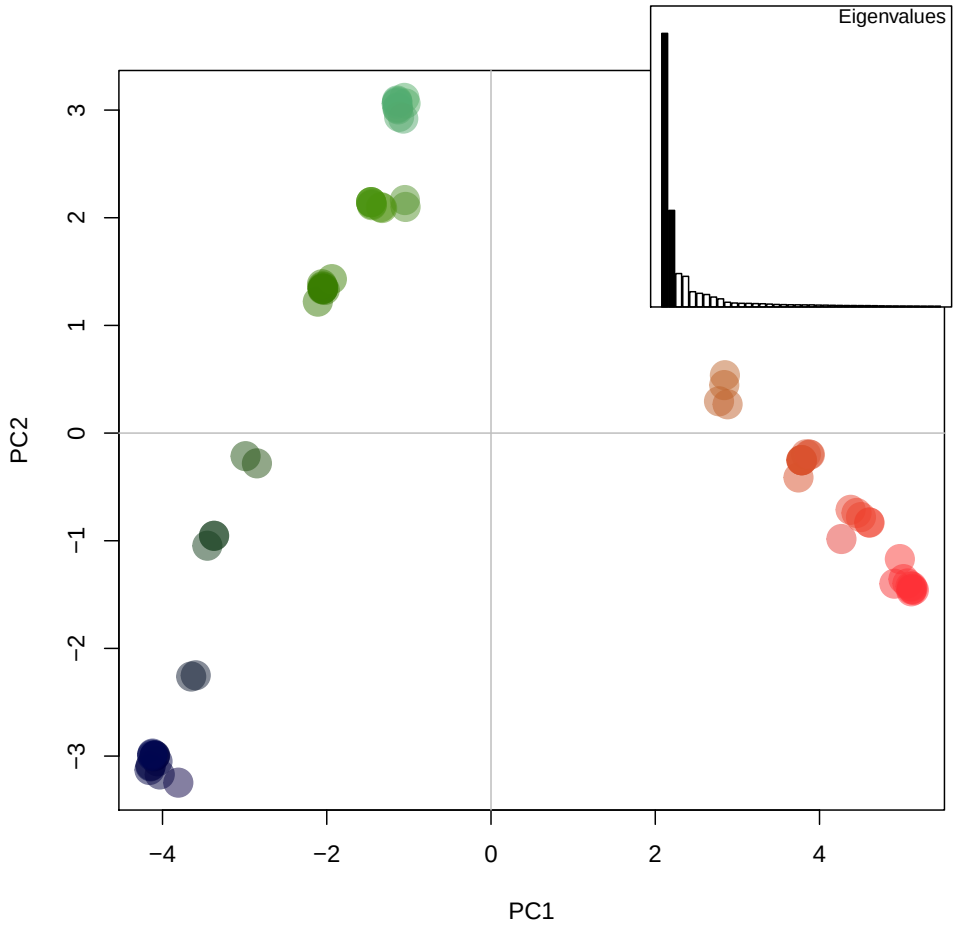
The first PC suggests the existence of two clades in the data, while the second one shows groups of closely related isolates arranged along a cline of genetic differentiation. This structure is confirmed by a simple neighbour-joining (NJ) tree:

```
library(ape)
tre <- nj(dist(as.matrix(flu)))
tre

##
## Phylogenetic tree with 80 tips and 78 internal nodes.
##
## Tip labels:
##   CY013200,  CY013781,  CY012128,  CY013613,  CY012160,  CY012272, ...
##
## Unrooted; includes branch lengths.

plot(tre, typ="fan", cex=0.7)
title("NJ tree of the US influenza data")
```

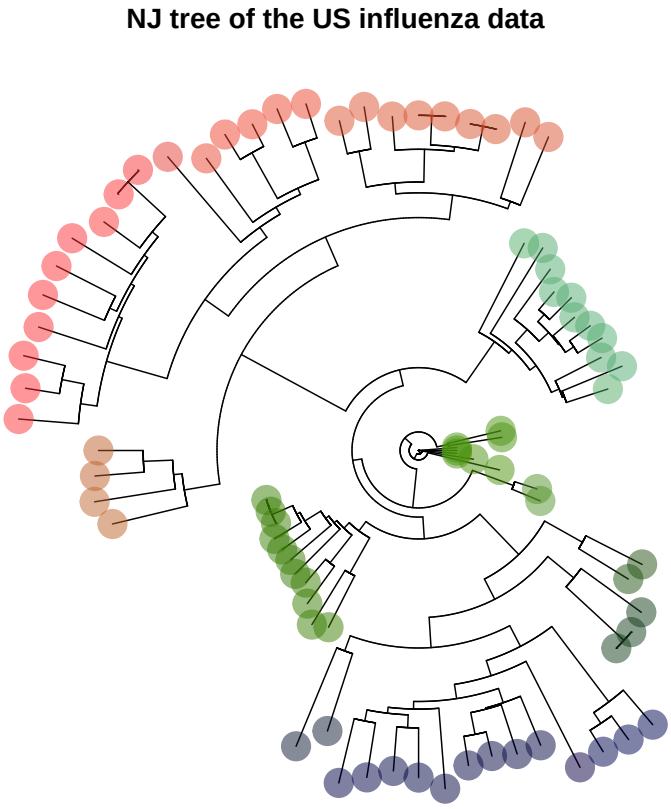


The correspondance between both analyses can be better assessed using colors based on PCs; this is achieved by `colorplot`:

```
myCol <- colorplot(pca1$scores,pca1$scores, transp=TRUE, cex=4)
abline(h=0,v=0, col="grey")
add.scatter.eig(pca1$eig[1:40],2,1,2, posi="topright", inset=.05, ratio=.3)
```




```
plot(tre, typ="fan", show.tip=FALSE)
tiplabels(pch=20, col=myCol, cex=4)
title("NJ tree of the US influenza data")
```



As expected, both approaches give congruent results, but both are complementary: NJ is better at showing bunches of related isolates, but the cline of genetic differentiation is much clearer in PCA.

4.4 Discriminant Analysis of Principal Components (DAPC)

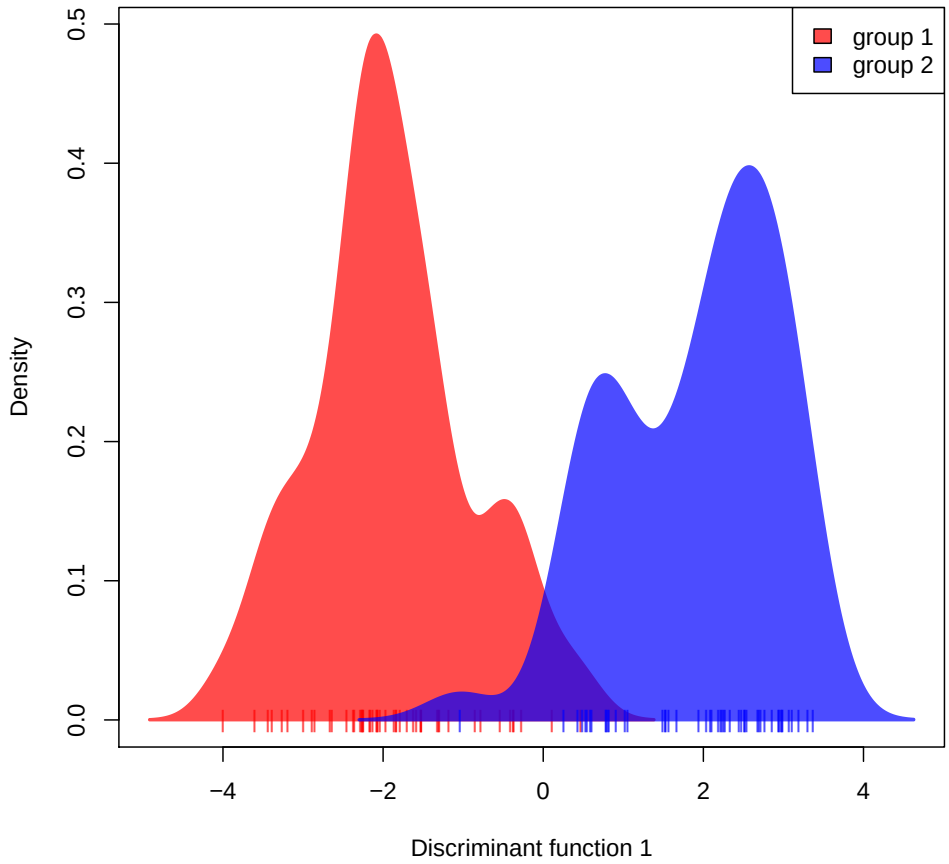
Discriminant analysis of Principal Components (DAPC) is implemented for `genlight` objects by an appropriate method for the `find.clusters` and `dapc` generics. To put it simply, you can run `find.clusters` and `dapc` on `genlight` objects and the appropriate functions will be used. As in `glPca`, these methods never require more than a couple of genomes to be translated into allele frequencies at a time, thereby minimizing RAM requirements.

Below, we illustrate DAPC on a `genlight` with 100 individuals, including only 50 structured SNPs out of 10,000 non-structured SNPs:

```
x <- glSim(100, 1e4, 50)
dapc1 <- dapc(x, n.pca=10, n.da=1)
```

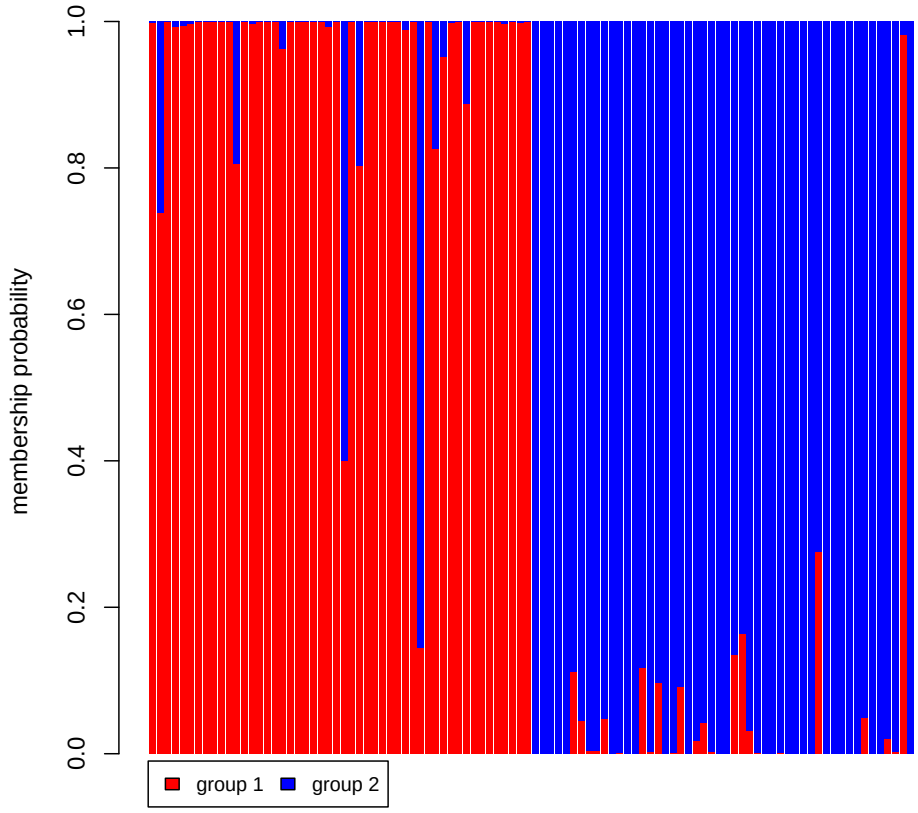
For the last 10 structured SNPs (located at the end of the alignment), the two groups of individuals have different (random) distribution of allele frequencies, while they share the same distributions in other loci. DAPC can still make some decent discrimination:

```
scatter(dapc1,scree.da=FALSE, bg="white", posi.pca="topright", legend=TRUE,
        txt.leg=paste("group", 1:2), col=c("red","blue"))
```



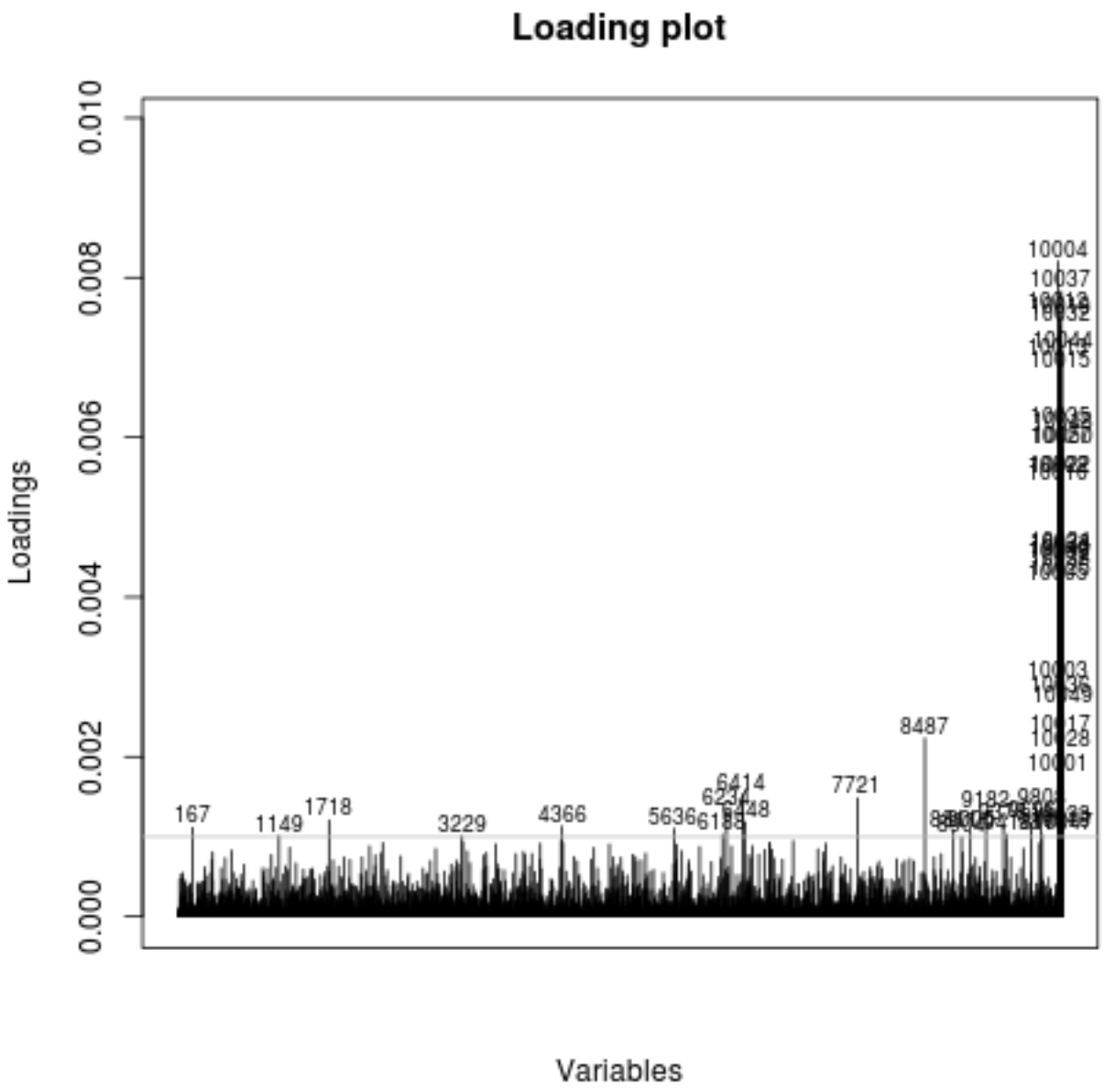
While the composition plot confirms that groups are not entirely disentangled...

```
compoplot(dapc1, col=c("red","blue"),lab="", txt.leg=paste("group", 1:2), ncol=2)
```



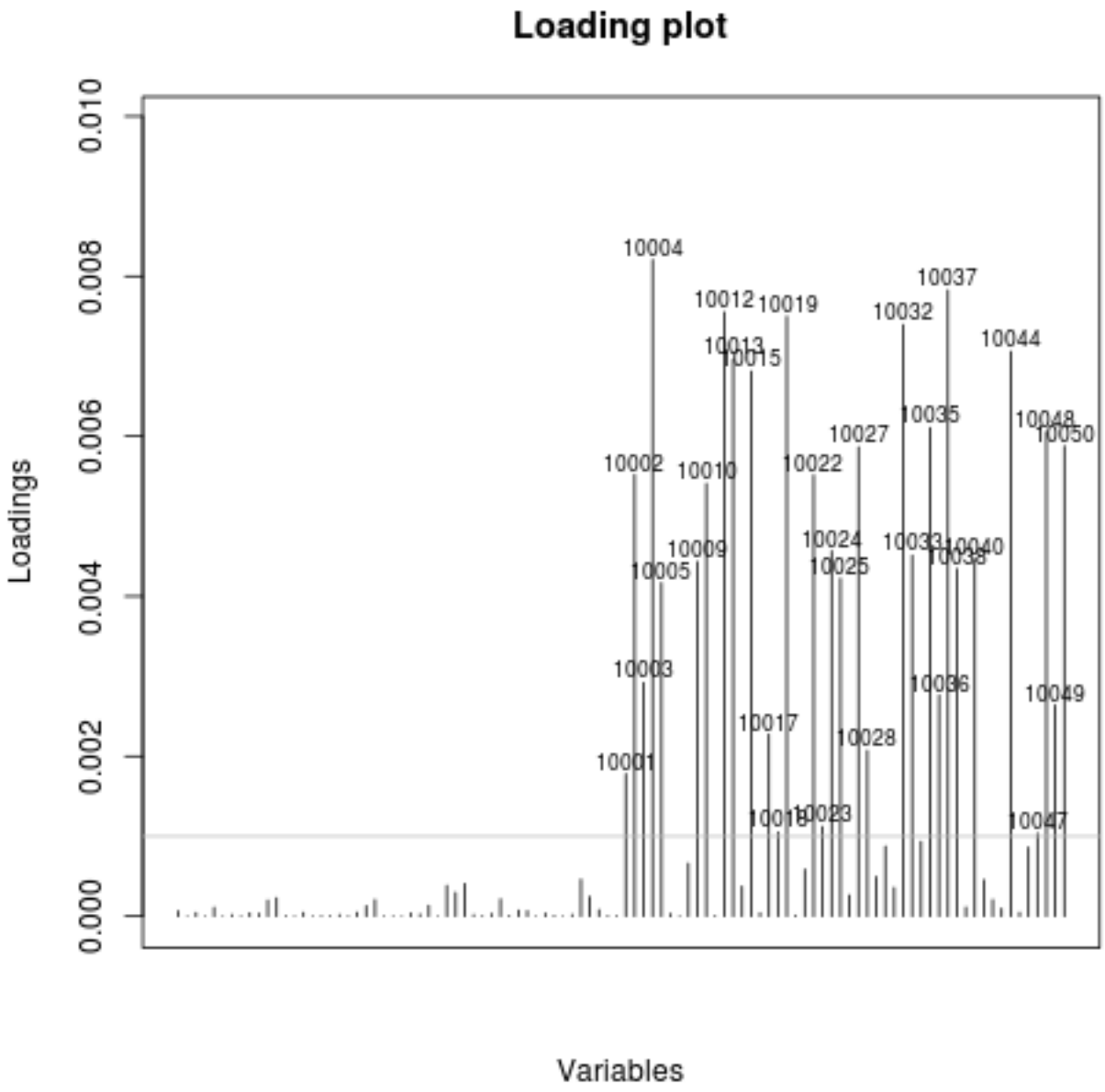
... the loading plot identifies pretty well the most discriminating alleles:

```
loadingplot(dapc1$var.contr, thres=1e-3)
```



And we can zoom in to the contributions of the last 100 SNPs to make sure that the tail indeed corresponds to the 50 last structured loci:

```
loadingplot(tail(dapc1$var.contr[,1],100), thres=1e-3)
```



Here, we indeed identified the structured region of the genome fairly well.

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 (2011). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.