Analysing genomic-wide SNP data using adegenet

Thibaut Jombart

May 31, 2011

Abstract

Genome-wide SNP data can quickly be challenging to analyse using standard computer. *adegenet* 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. It also introduces more advanced features of an API in C language which may be useful to develop new method based on these objects.

Contents

1 Introduction **3**

2 Classes of objects **3**
2.1 `SNPbin`: storage of single genomes 3
2.2 `genlight`: storage of multiple genomes 5

3 In practice **8**
3.1 Using accessors 8
3.2 Data conversions 8
3.3 Principal Component Analysis (PCA) 8
3.4 Discriminant Analysis of Principal Components (DAPC) 8

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 rapid 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: NA
 2 (Inf %) 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, 1e+06, replace = TRUE)
> print(object.size(dat), unit = "auto")


3.8 Mb


> x <- new("SNPbin", dat)
> print(object.size(x), unit = "auto")


122.8 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] TRUE
```

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 `SNbin` 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 1,00,000 SNPs each (including occasional NAs):

```
> dat <- lapply(1:50, function(i) sample(c(0, 1, NA), 1e+06, prob = c(0.5,
+     0.499, 0.001), replace = TRUE))
> names(dat) <- paste("indiv", 1:length(dat))
> print(object.size(dat), unit = "auto")


381.5 Mb


> x <- new("genlight", dat)
> print(object.size(x), unit = "auto")


6.2 Mb
```

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

```
61.6340315378476 bytes
```

here again, the storage if the data is much more efficient in `genlight` than using integers: converted data occupy 62 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, considerable efforts have been devoted to making these issues oblivious to the user. 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 *multicore* is installed an multiple cores are available.

3 In practice

3.1 Using accessors

3.2 Data conversions

3.3 Principal Component Analysis (PCA)

3.4 Discriminant Analysis of Principal Components (DAPC)