Error correction of GBS data using GBScleanR

Tomoyuki Furuta

March 25, 2021

Contents

Introduction

The GBScleanR package has been mainly developed to conduct error correction on genotype data obtained via NGS-base genotyping methods such as RAD-seq and GBS. Nevertheless, several quality check procedure and data filtering are highly encouraged to improve correction acculacy. Therefore, this package also provide the functions for data quality check and filtering with some data visualization functions to help filtering procedure. In this document, we walk through an error correction procedure for GBS data of a biparental population. Introduction of basic utility functions can be found in another vignette.

Prerequisites

This package internally uses the following packages.
- ggplot2
- dplyr
- tidyr
- GWASTools
- SNPRelate
- SeqArray

To install them all, run the codes below.

```
if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")

BiocManager::install("GWASTools")
BiocManager::install("SNPRelate")
BiocManager::install("SeqArray")
install.packages("ggplot2")
install.packages("dplyr")
install.packages("tidyr")
```

You can install GBScleanR from the local source file with the following code.

```
install.packages("path/to/source/GBScleanR.tar.gz", repos = NULL, type = "source")
```

The code below let you install the package from the github repository.

```
if (!requireNamespace("devtools", quietly = TRUE))
    install.packages("devtools")
devtools::install_github("")
```

To load the package.

```
library("GBScleanR")
```

Data format conversion and object instantiation

The main class of the `GBScleanR` package is `gbsrGenotypData` which inherits the `GenotypeData` class in the `GWASTools` package. The `gbsrGenotypeData` class object has three slots: `data`, `snpAnnot`, and `scanAnnot`. The `data` slot holds genotype data as a `gds.class` object which is defined in the `gdsfmt` package while `snpAnnot` and `scanAnnot` contain objects storing annotation information of SNPs and samples, which are the `SnpAnnotationDataFrame` and `ScanAnnotationDataFrame` objects defined in the `GWASTools` package. See the vignette of `GWASTools` for more detail. `GBScleanR` follows the way of `GWASTools` in which a unique genotyping instance (genotyped sample) is called "scan".

As mentioned above, the `gbsrGenotypeData` class requires genotype data in the `gds.class` object which enable us quick access to the genotype data without loading the whole data on RAM. At the beginning of the processing, we need to convert data format of our genotype data from VCF to GDS. This conversion can be achi eved using `gbsrVCF2GDS` as shown below.

```
gbsrVCF2GDS(vcf_fn = "./data/gbs_nbolf2.vcf.gz", # Path to the input VCF file.
            out_fn = "./data/gbs_nbolf2.gds") # Path to the output GDS file.
```

Once we created the GDS, we can create the `gbsrGenotypeData` instance for our data.

```
gdata <- loadGDS("./data/gbs_nbolf2.gds")
```

Check the number of SNPs and samples.

```
nSnp(gdata)
```

```
## [1] 20224
```

```
nScan(gdata)
```

```
## [1] 816
```

This sample data contains 813 of F2 individuals derived from the cross between Oryza sativa cv. Nipponbare and O. longistaminata, which is a cross between distant relatives. The parental lines and one F1 individual are also included in the data.
We can omit the F1 individual which will not be used in the downstream genetic analysis.

```
f1 <- grep("F1", getScanID(gdata), value = TRUE)
f1
```

```
## [1] "03_F1"
```

```
gdata <- setScanFilter(gdata, id = f1)
nScan(gdata)
```

```
## [1] 815
```

Set the parental samples

In the case of genotype data in a biparental population, peaple usually filter out SNPs which are not monomorphic in each parental sample and not biallelic between parents. `setParents()` automatically do this filtering.

```
p1 <- grep("NB", getScanID(gdata), value = TRUE)
p2 <- grep("OL", getScanID(gdata), value = TRUE)
gdata <- setParents(gdata, parents = c(p1, p2))
nSnp(gdata)
```

```
## [1] 12853
```

As you can see in the message from the function, this function also sorts the genotype data to make the allele of the first parent being the reference allele. Therefore, the order of sample names given to the `parents` argument is important. In this example, all the alleles found in “NB” are set as the reference alleles.

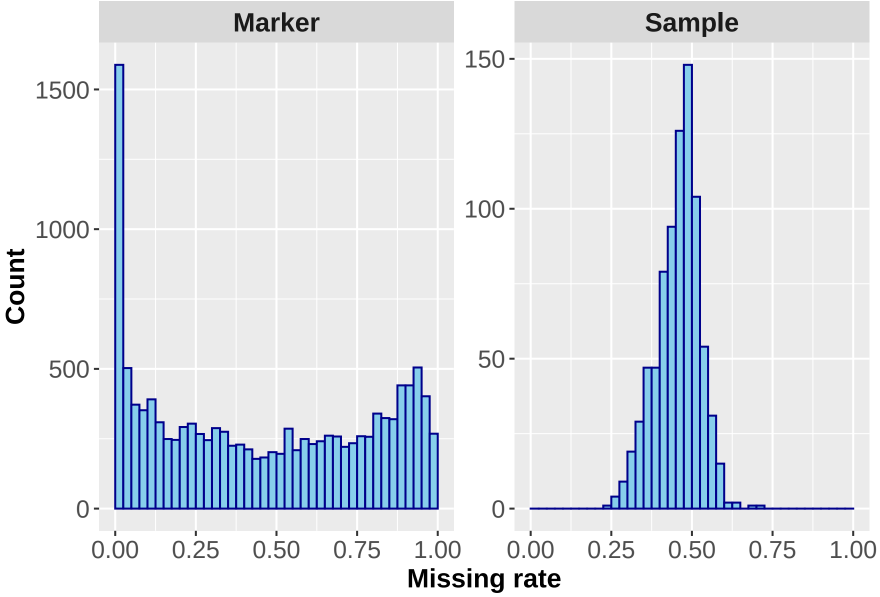
Check basic statistics of the given data

To calculate several basic statistics including missing rate and heterozygosity, first we need to run `countGenotype()`.

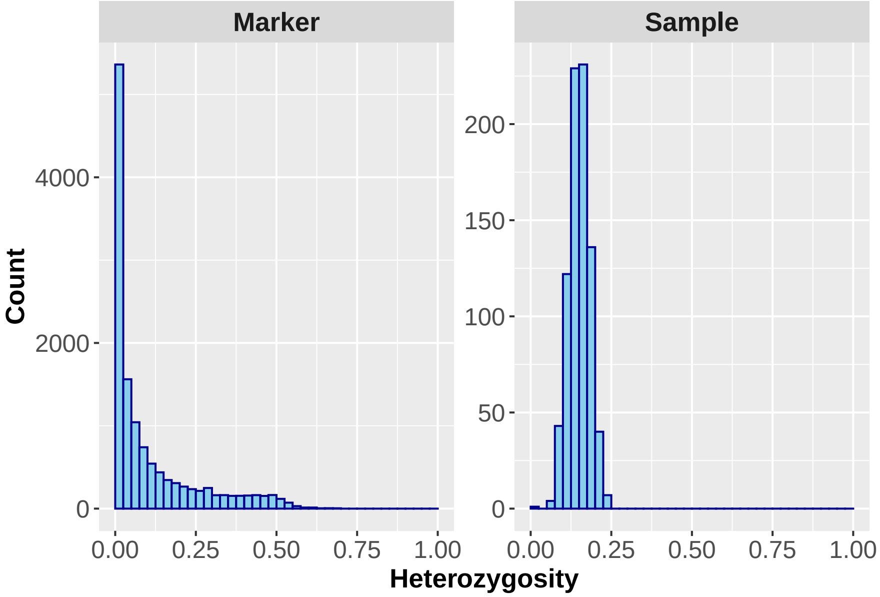
```
gdata <- countGenotype(gdata)
```

Then, get histograms using `hist()`.

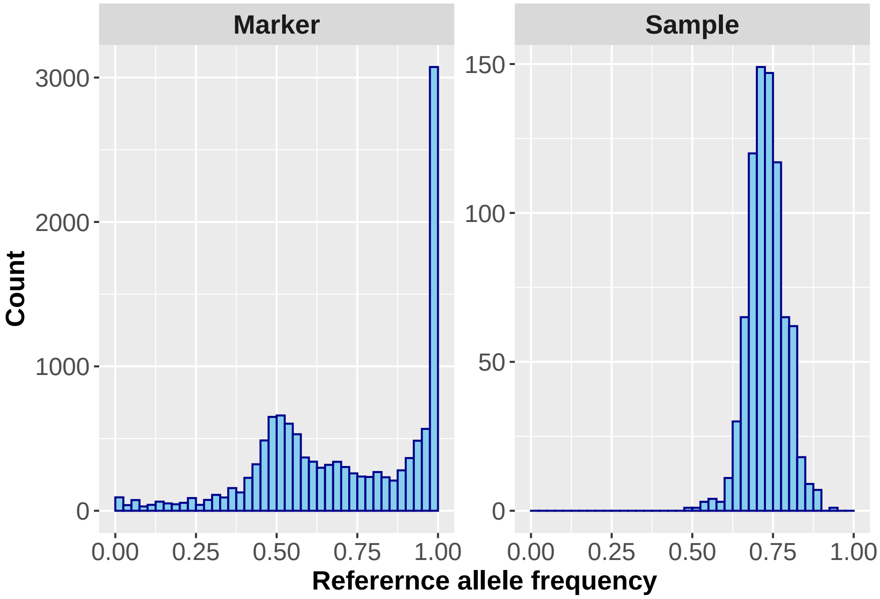
```
hist(gdata, stats = "missing")
```



```
hist(gdata, stats = "het")
```



```
hist(gdata, stats = "raf")
```



As the plots showed, the data contains a lot of missing genotype calls with unreasonable heterozygosity in a F2 population. Reference allele frequency shows a huge bias to reference allele. If you can say your population has no strong segregation distortion in any positions of the genome, you can filter out the markers having too high or too low reference allele frequency.

```
# filter out markers with reference allele frequency
# less than 5% or more than 95%.
gdata <- setSnpFilter(gdata, maf = 0.05)
```

However, sometimes filtering based on allele frequency per marker removes all markers from regions truly showing segregation distortion. Although heterozygosity also can be a criterion to filter out markers, this will removes too many markers which even contains useful information for genotyping.

If we found poor quality samples in you dataset based on missing rate, heterozygosity, and reference allele frequency, we can omit those samples with `setScanFilter()`.

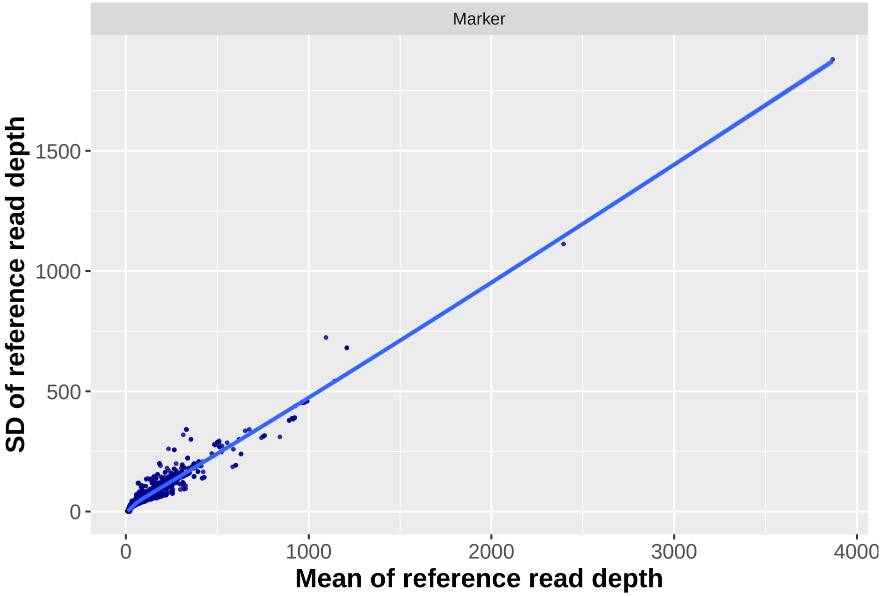
```
# Filter out samples with more than 90% missing genotype calls,
# less than 5% heterozygosity, and less than 5% minor allele frequency.
gdata <- setScanFilter(gdata, missing = 0.9, het = 0.05, maf = 0.05)
```

As the first step of marker filtering, we should filter out poor quality markers. There are many criterion to filter out markers, e.g. missing rate, allele frequency, and read counts. Here we use degree of dispersion of read counts. As a measure of dispersion, we use the ratio of mean and SD per marker. `calcReadStats()` gives us the mean and SD of normalized read counts per marker. This function first calculate a normalized read count of each marker of each sample by dividing each read count by total reads per sample followed by multiplication by 1,000,000 to obtain read count per million. Then, the mean and SD of normalized read counts per sample and per marker are calculated only for non-zero values. In other words, genotype calls with no read for reference allele are omitted from the calculation of mean and SD of normalized reference allele reads, and also do the same for alternative allele. With the normalization, we can compare mean read depth per marker without concern about absolute differences in total read depth per marker. `calcReadStats()` takes a longer running time than `countGenotype` and `countRead`. Please wait for a while with a cup of coffee, if your data has a many markers and samples.

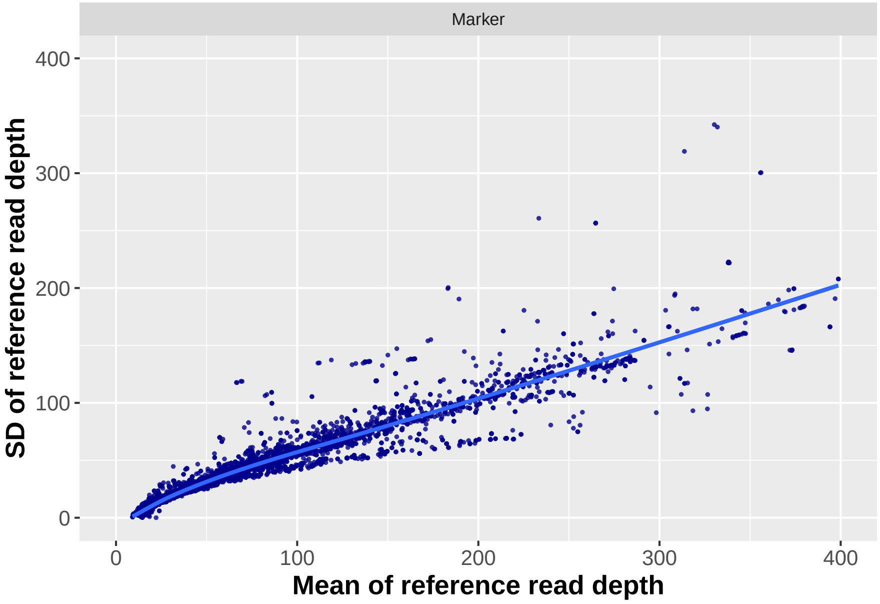
```
gdata <- calcReadStats(gdata)
```

The `pairs()` function allow us to make a two dimensional scatter plot to visualize mean vs SD of each marker.

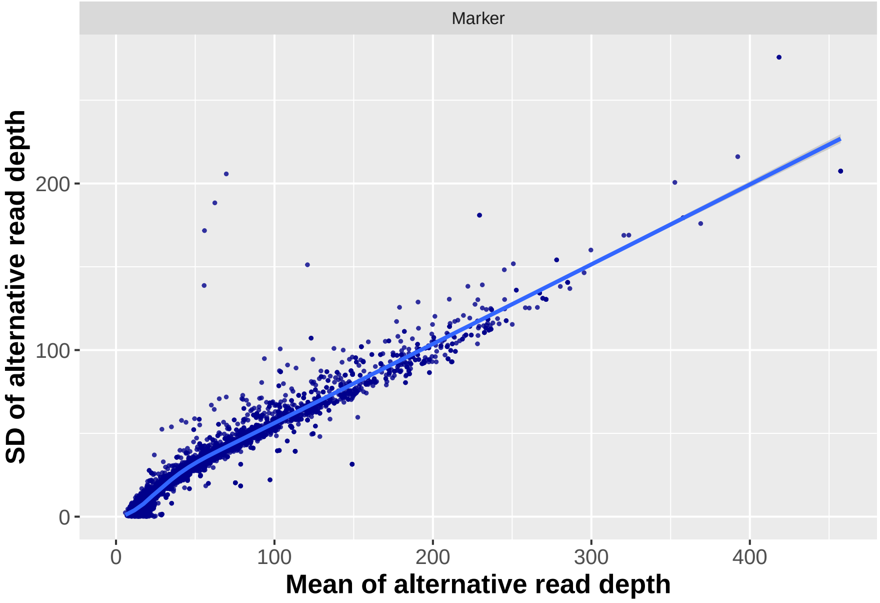
```
pairs(gdata, stats1 = "mean_ref", stats2 = "sd_ref", target = "snp", smooth = TRUE)
```



```
pairs(gdata, stats1 = "mean_ref", stats2 = "sd_ref", target = "snp", smooth = TRUE,
      ggargs = "xlim(c(0, 400)) + ylim(c(0, 400))")
```



```
pairs(gdata, stats1 = "mean_alt", stats2 = "sd_alt", target = "snp", smooth = TRUE)
```



`smooth = TRUE` puts a smoothed line in the scatter plot and allow us to visualize a trend of mean/SD ratios. We can see some markers plot far away from the trend line and interpret it as that observations of reads supporting them have completely different characteristics from the majority which follows the trend line. The trend lines shown in the plots were obtained via the method `gam` with the formula `y ~ s(x, bs = "cs")`. This model can be done with the `mgcv` package.

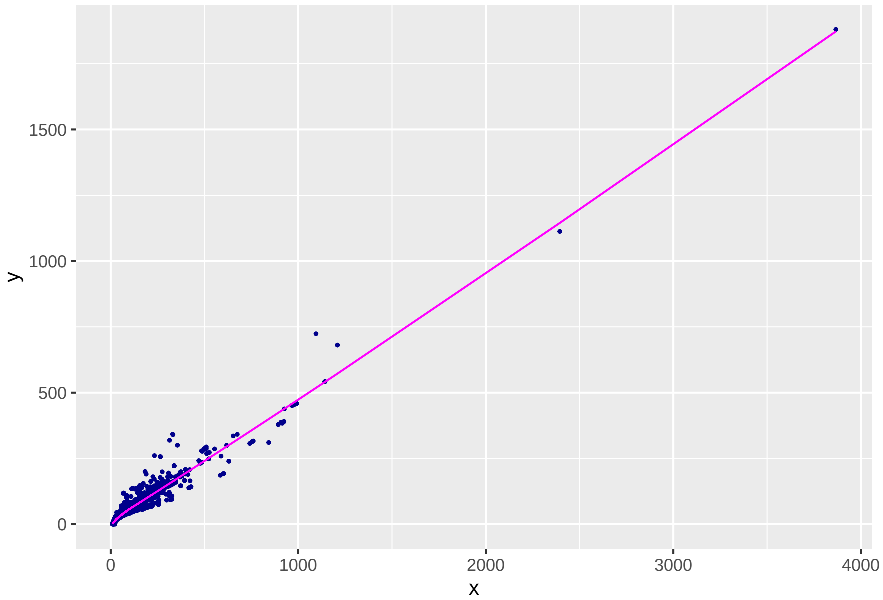
```
library(mgcv)
```

```
x <- getMeanReadRef(gdata, target = "snp")
y <- getSDReadRef(gdata, target = "snp")
df <- data.frame(x, y)
df <- subset(df, subset = !is.na(x) & !is.na(y))
gam_fit <- gam(formula = y ~ s(x, bs = "cs"), data = df)
```

Now we got the model of a smoothed line which should be same with that in the plot shown above. Let's check the fit of the line on the scatter plot for mean vs SD of reference allele read depth.

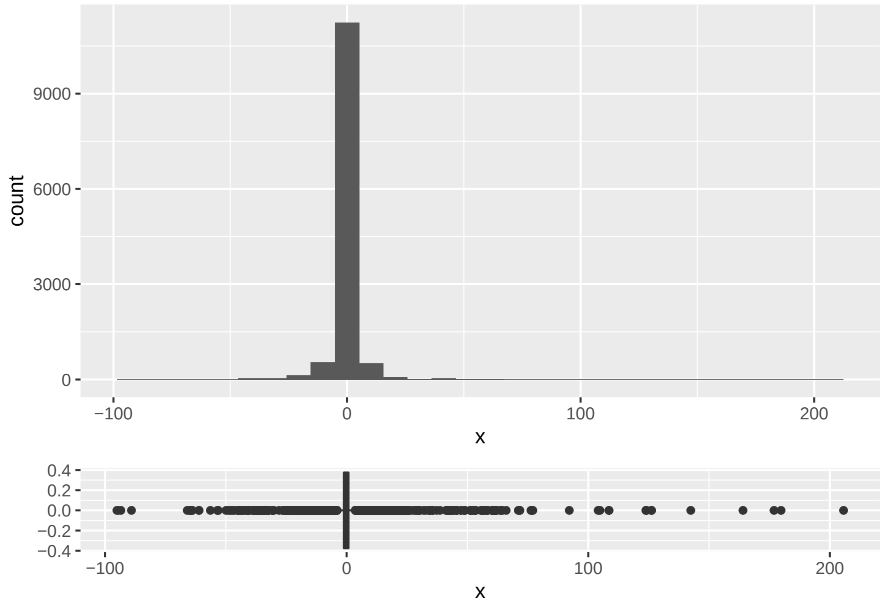
```
library(ggplot2)
```

```
ggplot(df, aes(x = x, y = y)) + geom_point(size = 0.5, color = "darkblue") +
  geom_line(data = data.frame(x = gam_fit$model$x, y = gam_fit$fitted.values),
            mapping = aes(x = x, y = y, group = 1), color = "magenta")
```



We can also visualize how much each data point diverges from the trend line.

```
p1 <- ggplot(data.frame(x = gam_fit$residuals), aes(x = x)) + geom_histogram()
p2 <- ggplot(data.frame(x = gam_fit$residuals), aes(x = x)) + geom_boxplot()
cowplot::plot_grid(p1, p2, ncol = 1, rel_heights = c(3, 1), align = "v", axis = "lr")
```



The boxplot shown above showed our data have a lot of markers seem to have over dispersion. We can filter out these outliers.

```
retain_ref <- rep(FALSE, length(x))
b <- boxplot(gam_fit$residuals, plot = FALSE)
retain_ref[!is.na(x) & !is.na(y)]  <- gam_fit$residuals >= b$stats[1, 1] & gam_fit$residuals <= b$stats
```

Do the same for alternative allele read.

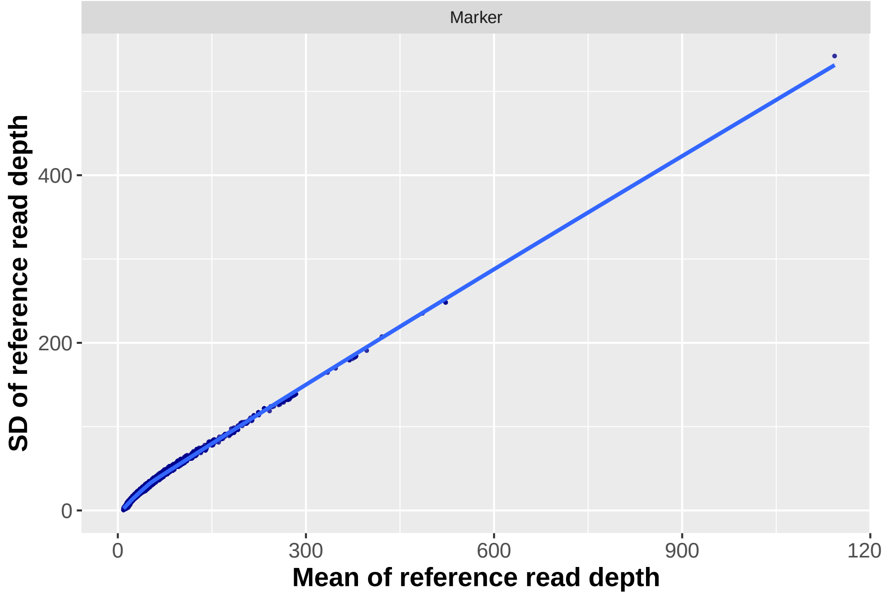
```
x <- getMeanReadAlt(gdata, target = "snp")
y <- getSDReadAlt(gdata, target = "snp")
```

```r
df <- data.frame(x, y)
df <- subset(df, subset = !is.na(x) & !is.na(y))
gam_fit <- gam(formula = y ~ s(x, bs = "cs"), data = df)
retain_alt <- rep(FALSE, length(x))
b <- boxplot(gam_fit$residuals, plot = FALSE)
retain_alt[!is.na(x) & !is.na(y)]  <- gam_fit$residuals >= b$stats[1, 1] & gam_fit$residuals <= b$stats
gdata <- setValidSnp(gdata, update = retain_ref & retain_alt)
nSnp(gdata)
```

```
## [1] 9159
```

Let's check the effect of

```r
pairs(gdata, stats1 = "mean_ref", stats2 = "sd_ref", target = "snp", smooth = TRUE)
```

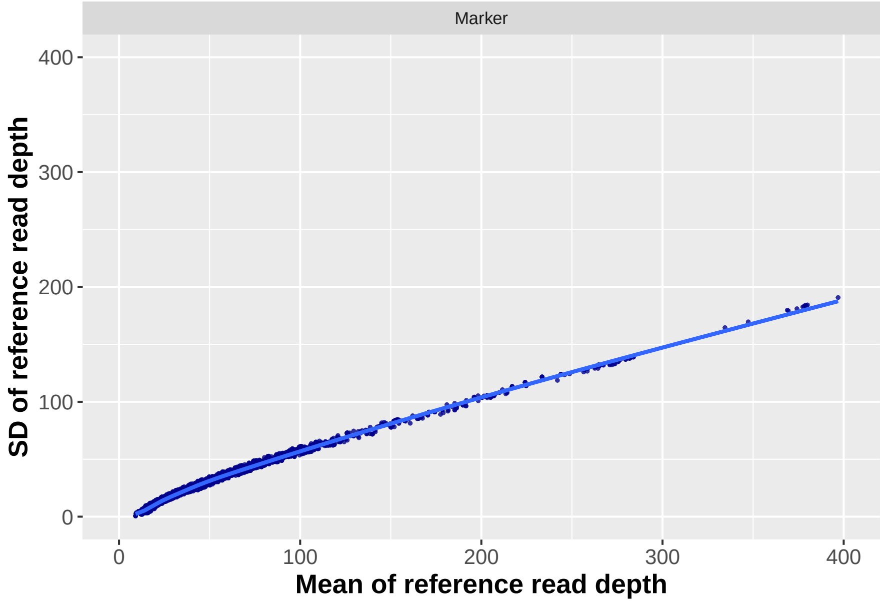


```r
pairs(gdata, stats1 = "mean_ref", stats2 = "sd_ref", target = "snp", smooth = TRUE,
      ggargs = "xlim(c(0, 400)) + ylim(c(0, 400))")
```

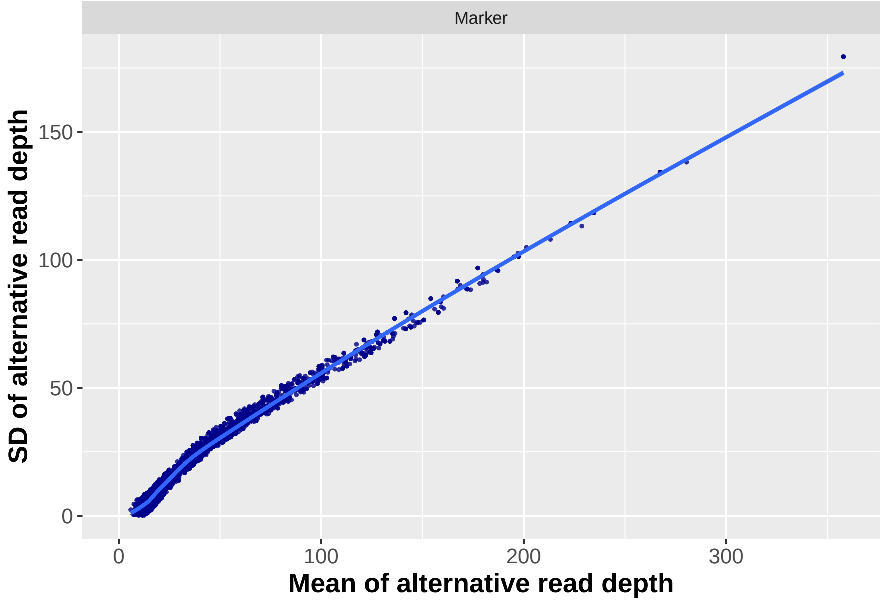
```
## Scale for 'x' is already present. Adding another scale for 'x', which will
## replace the existing scale.
```

```
## Scale for 'y' is already present. Adding another scale for 'y', which will
## replace the existing scale.
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



```
pairs(gdata, stats1 = "mean_alt", stats2 = "sd_alt", target = "snp", smooth = TRUE)
```



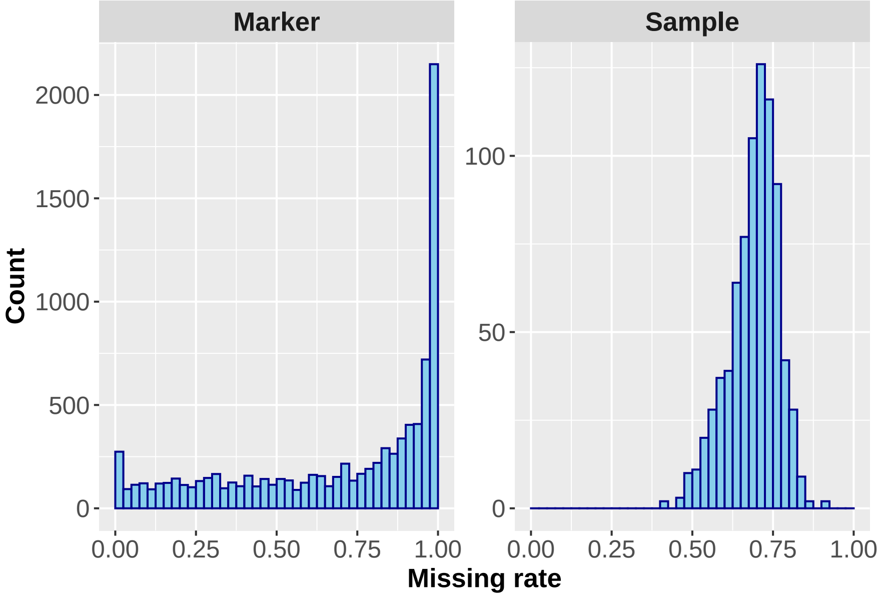
As the next step of marker filtering, we can conduct filtering on each genotype call based on read depth. The error correction via GBScleanR is robust against low coverage calls, while genotype calls messed up by mismapping might lead less reliable error correction. Therefore, filtering for low coverage calls are not necessary. However, if the given dataset is super low coverage, e.g. < 1x in average, filtering out genotype calls supported by only one read may be helpful. Heterozygote is never be able to be called as heterozygote with only read. Filtering on each genotype call takes several tens of minutes. Please wait for a while with a cup of coffee with some sweets, if your data has a many markers and samples.

```
# Filter out genotype calls supported by reads less than 2 reads.
gdata <- setCallFilter(gdata, dp_count = c(2, Inf))
```

Now we should check basic statistics.

```
gdata <- countGenotype(gdata)
```

```
hist(gdata, stats = "missing")
```



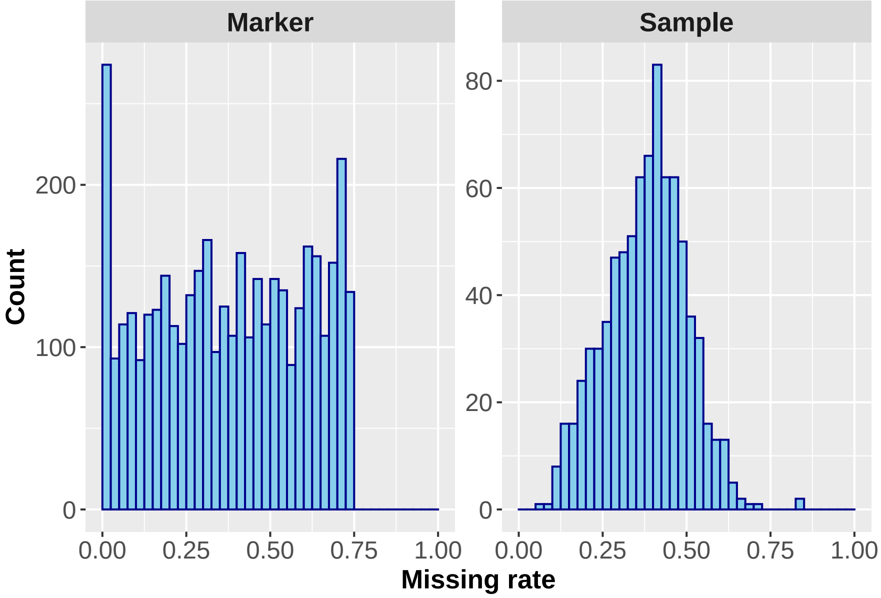
We can here remove markers based on missing genotype calls.

```
# Remove markers having more than 75% of missing genotype calls
gdata <- setSnpFilter(gdata, missing = 0.75)
nSnp(gdata)
```

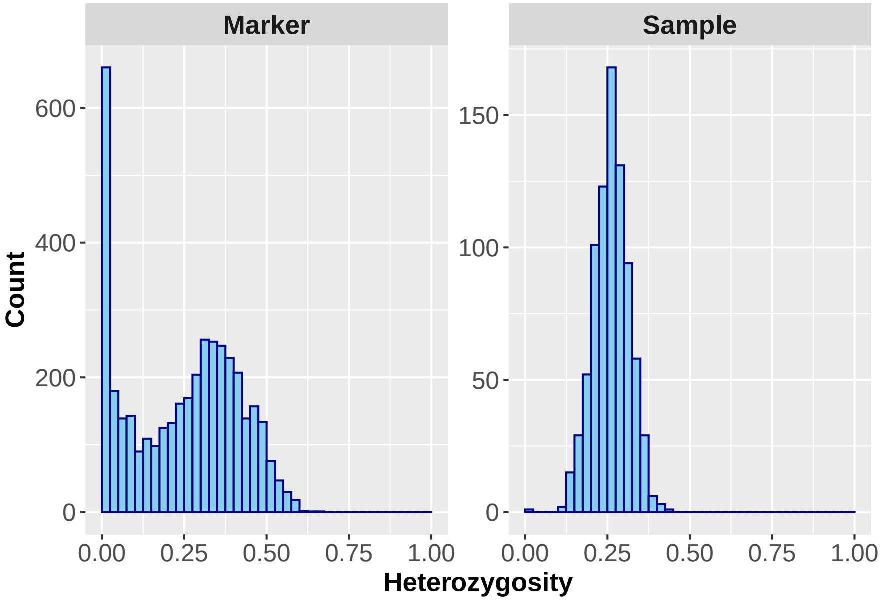
```
## [1] 4007
```

```
gdata <- countGenotype(gdata)
```

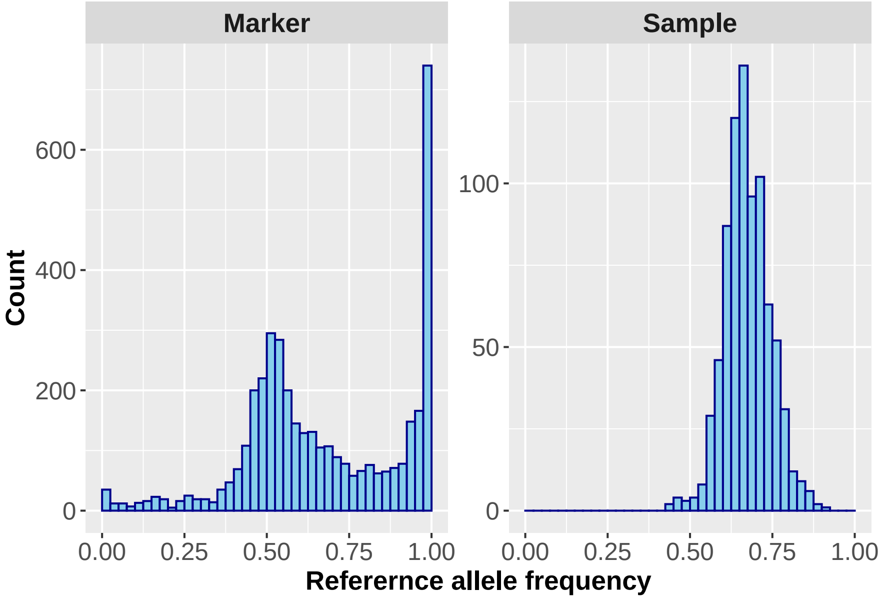
```
hist(gdata, stats = "missing")
```



```
hist(gdata, stats = "het")
```

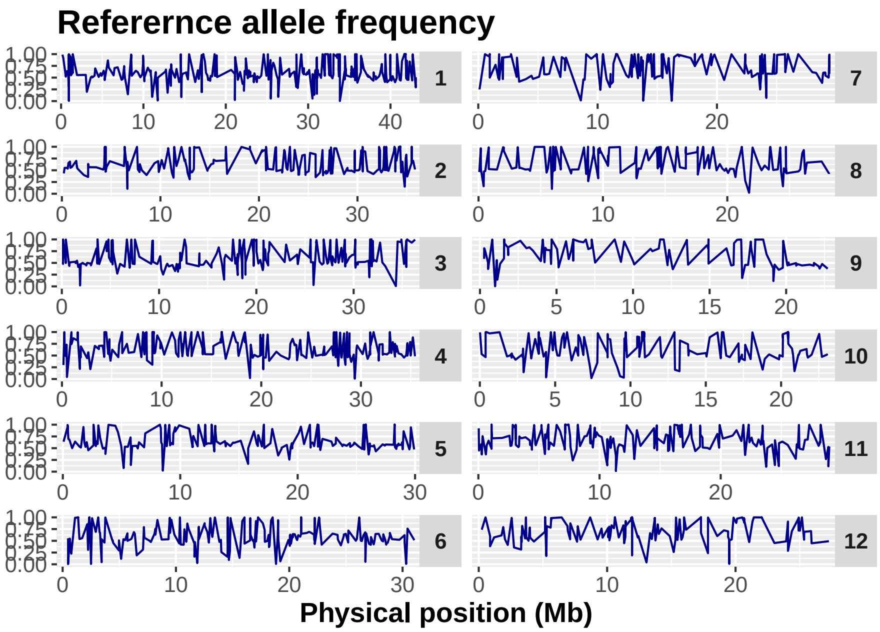


```
hist(gdata, stats = "raf")
```



We can still see the markers showing distortion in allele frequency, while the expected allele frequency is 0.5 in a F2 population. To investigate that those markers having distorted allele frequency were derived from truly distorted regions or just error prone markers, we must check if there are regions where the markers with distorted allele frequency are clustered.

```
plot(gdata, stats = "raf", coord = c(6, 2))
```



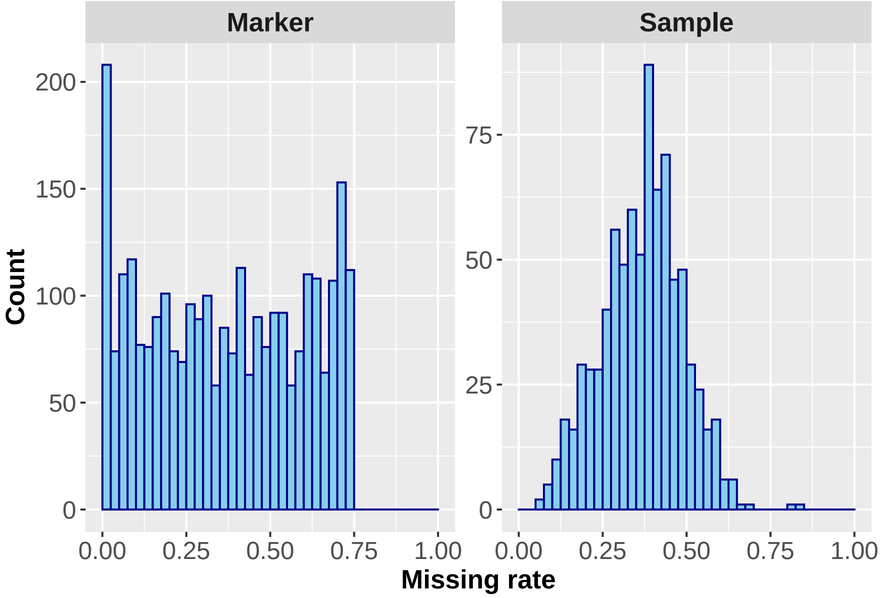
No region seem to have severe distortion. Based on the histogram of reference allele frequency, we can roughly cut off the markers with frequency more than 0.9 or less than 0.1, in other words, less than 0.1 minor allele frequency.

```
gdata <- setSnpFilter(gdata, maf = 0.1)
nSnp(gdata)
```

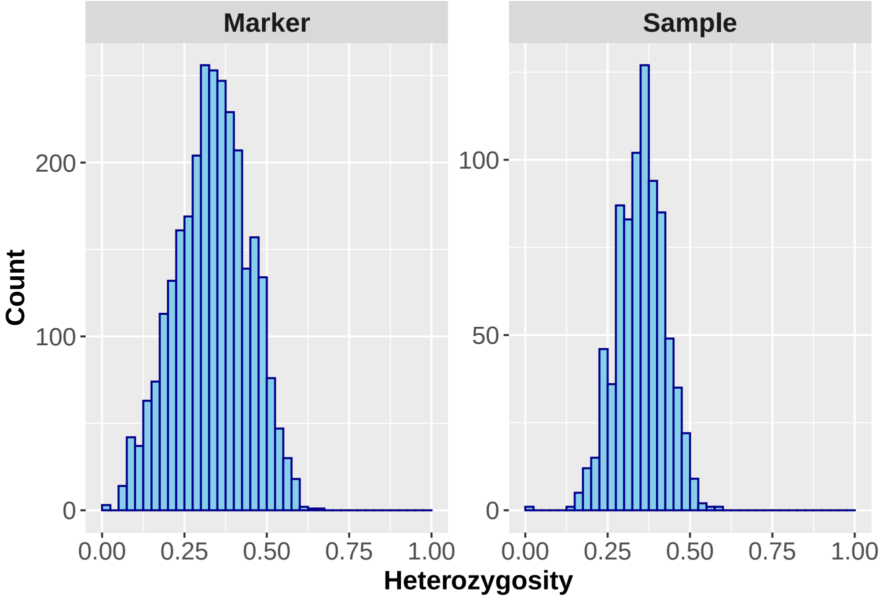
```
## [1] 2809
```

Let's see the statistics again.

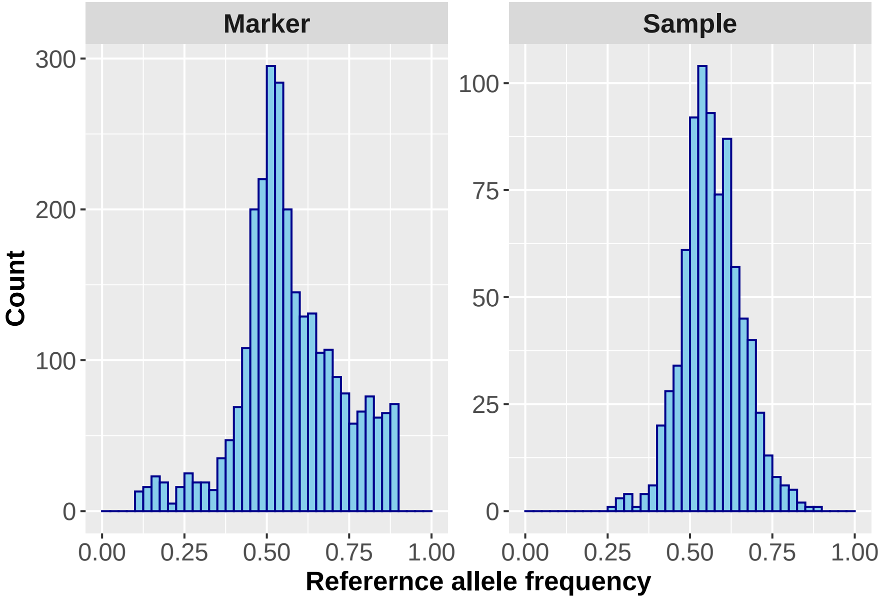
```
gdata <- countGenotype(gdata)
hist(gdata, stats = "missing")
```



```
hist(gdata, stats = "het")
```

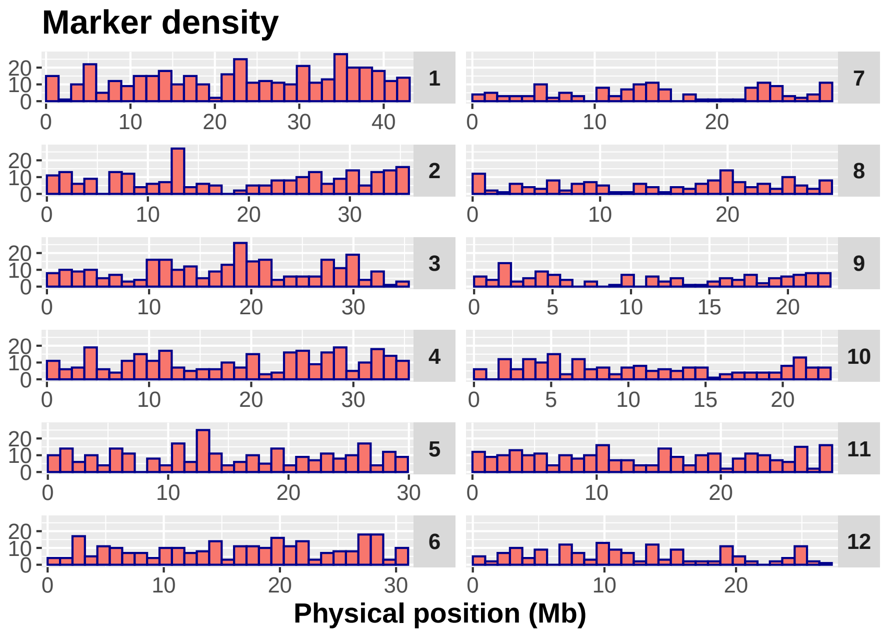


```
hist(gdata, stats = "raf")
```

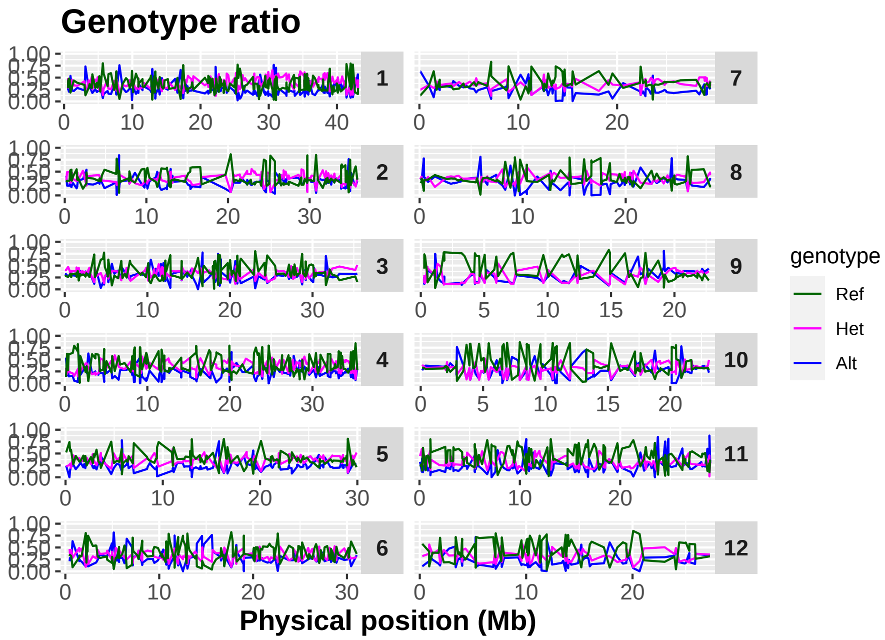


At the end of filtering, check marker density and genotype ratio per marker along chromosomes.

```
# Marker density
plot(gdata, stats = "marker", coord = c(6, 2))
```



```
plot(gdata, stats = "geno", coord = c(6, 2))
```



The `coord` argument controls the number of rows and columns of the facets in the plot.

To save the filtered data, we can create the subset GDS file containing only the retained data.

```
subset_gdata <- subsetGDS(gdata,
                          out_fn = "./data/gbs_nbolf2_subset.gds",
                          snp_incl = getValidSnp(gdata),
                          scan_incl = getValidScan(gdata))
```

```
## Clean up the fragments of GDS file:
##     open the file '/home/ftom/hdd2/gbscleanr/data/gbs_nbolf2_subset.gds' (4.3M)
##     # of fragments: 173
##     save to '/home/ftom/hdd2/gbscleanr/data/gbs_nbolf2_subset.gds.tmp'
##     rename '/home/ftom/hdd2/gbscleanr/data/gbs_nbolf2_subset.gds.tmp' (4.3M, reduced: 1.6K)
##     # of fragments: 33
```

out_fn is the file path of the output GDS file storing the subset data. Users need to specify, for snp_incl and scan_incl, a logical vector indicating which markers and samples should be included in the subset. The functions getValidSnp() and getValidScan return a logical vector indicating which markers and samples are retained by setSnpFilter() and setScanFilter(). subsetGDS returns a new gbsrGenotypeData object for the subset.