Error correction of GBS data using GBScleanR

Tomoyuki Furuta

September 30, 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 "BasicUsageOfGBSR.pdf".

Prerequisites

This package internally uses the following packages.
- ggplot2
- dplyr
- tidyr - expm - gdsfmt - biobase - GWASTools
- SeqArray

You can install GBScleanR from the Bioconductor repository with the following code.

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

BiocManager::install("GBScleanR")
```

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".

GBScleanR only support a VCF file as input. As an example data, we use simulated genotype data for a simulated biparental F2 population derived from inbred founders.

```
vcf_fn <- system.file("extdata", "simpop.vcf", package = "GBScleanR")
gds_fn <- system.file("extdata", "simpop.gds", package = "GBScleanR")
```

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 achieved using `gbsrVCF2GDS` as shown below. A compressed VCF file (.vcf.gz) also can be the input. Our sample dataset contains genotype information of 100 samples with 1000 markeres on only one chromosome.

```
gbsrVCF2GDS(vcf_fn = vcf_fn, # Path to the input VCF file.
            out_fn = gds_fn) # Path to the output GDS file.
```

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

```
gdata <- loadGDS(gds_fn)
```

Check the number of SNPs and samples.

```
nsnp(gdata)
```

```
## [1] 100
```

```
nscan(gdata)
```

```
## [1] 102
```

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("Founder1", getScanID(gdata), value = TRUE)
p2 <- grep("Founder2", getScanID(gdata), value = TRUE)
gdata <- setParents(gdata, parents = c(p1, p2))
nsnp(gdata)
```

```
## [1] 87
```

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.

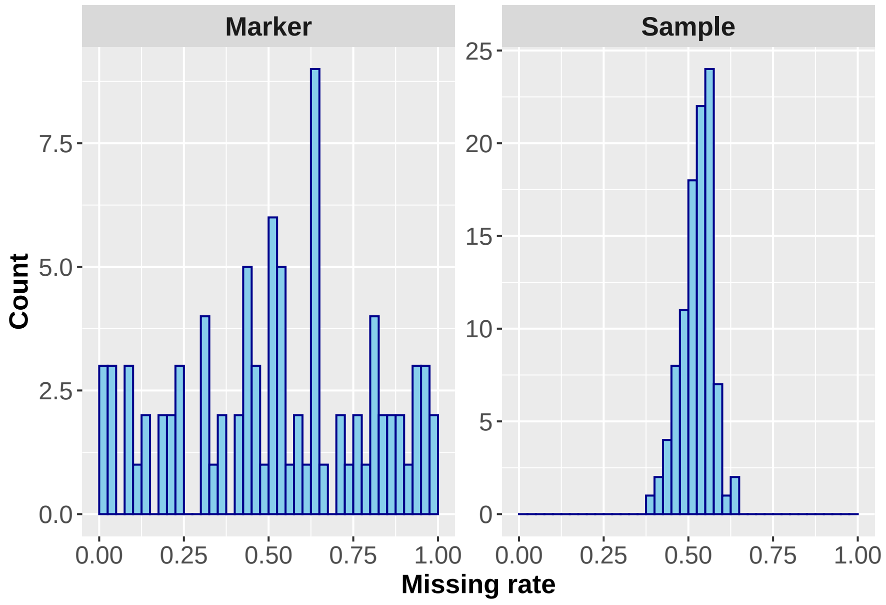
Check basic statistics of the given data

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

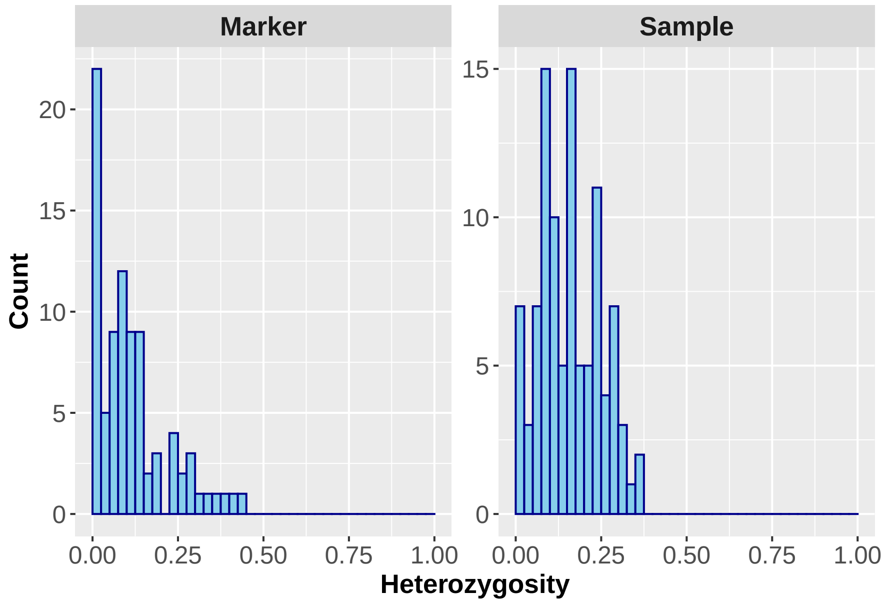
```
gdata <- countGenotype(gdata)
```

Then, get histograms using `hist()`.

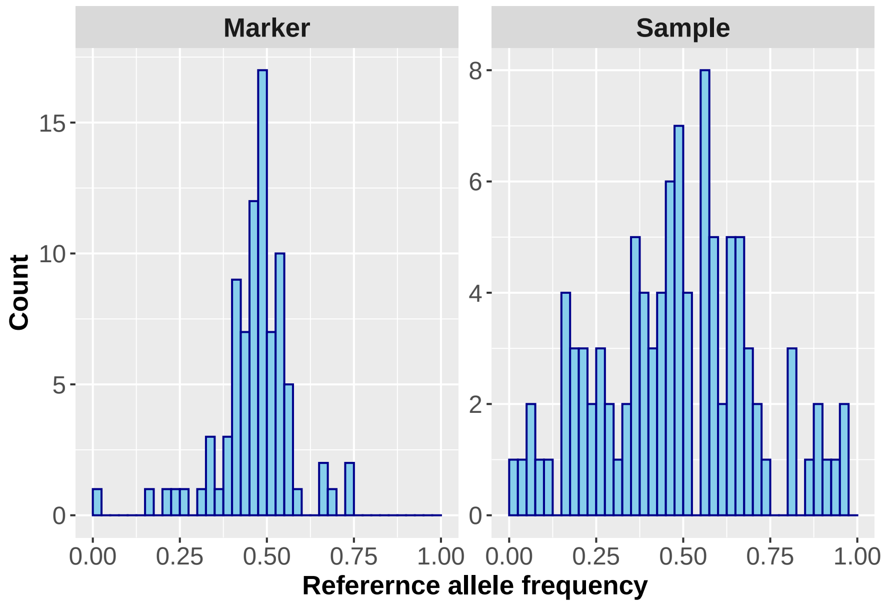
```
histGBSR(gdata, stats = "missing")
```



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



```
histGBSR(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.

```r
# 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()`.

```r
# 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)
```

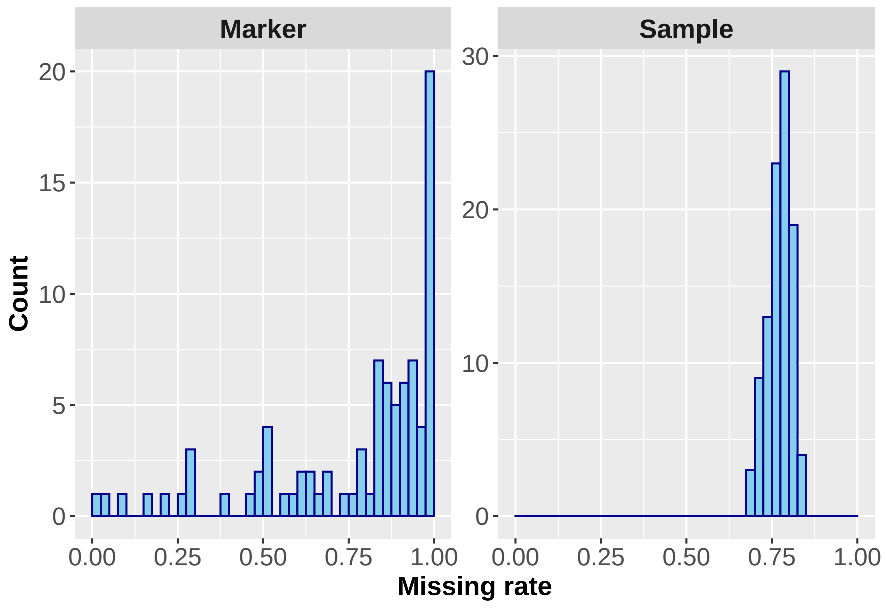
As the next step of marker filtering, we can conduct filtering on each genotype call based on read depth. The error correction via `GBScleanR` is robust against low coverage calls, while genotype calls messed up by mismapping might lead less reliable error correction. Therefore, filtering for low coverage 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.

```r
# 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.

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

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



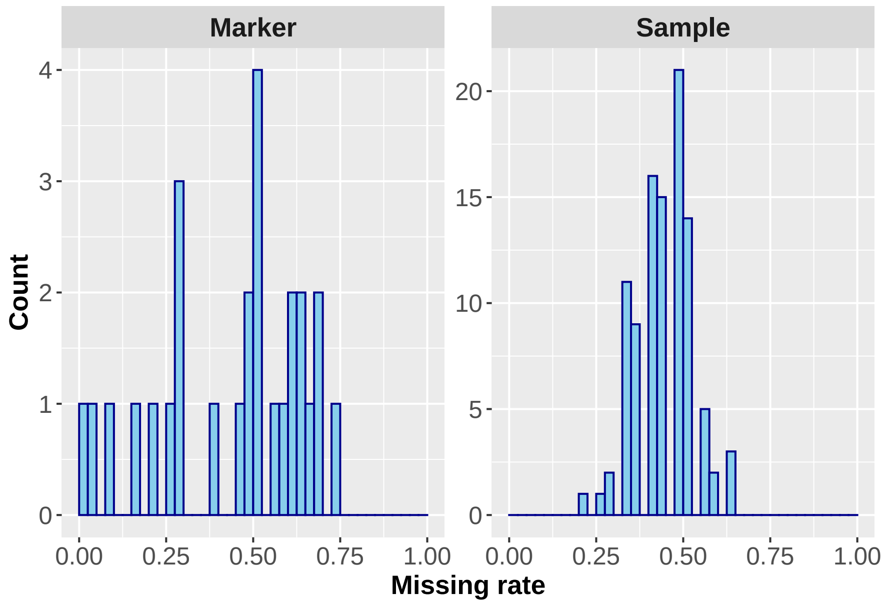
We can here remove markers based on missing genotype calls.

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

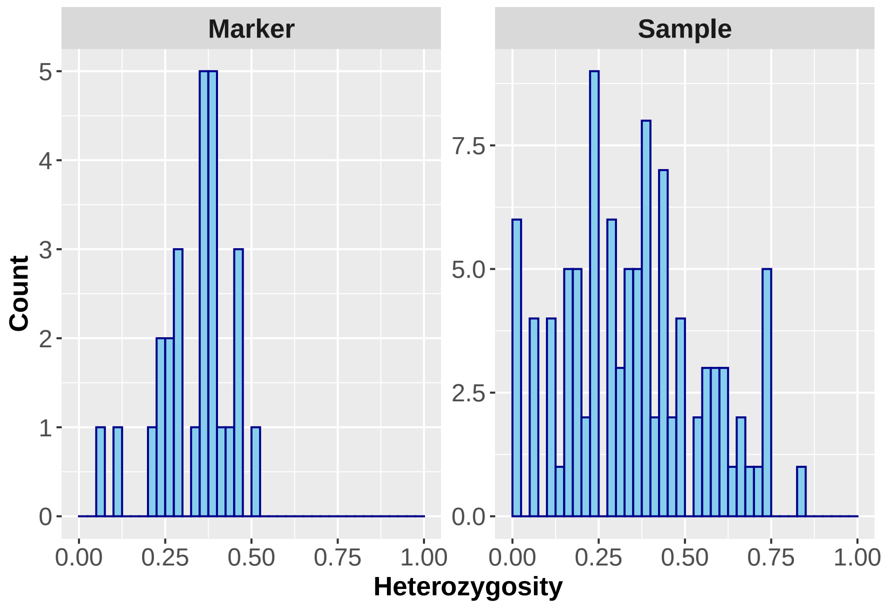
```
## [1] 27
```

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

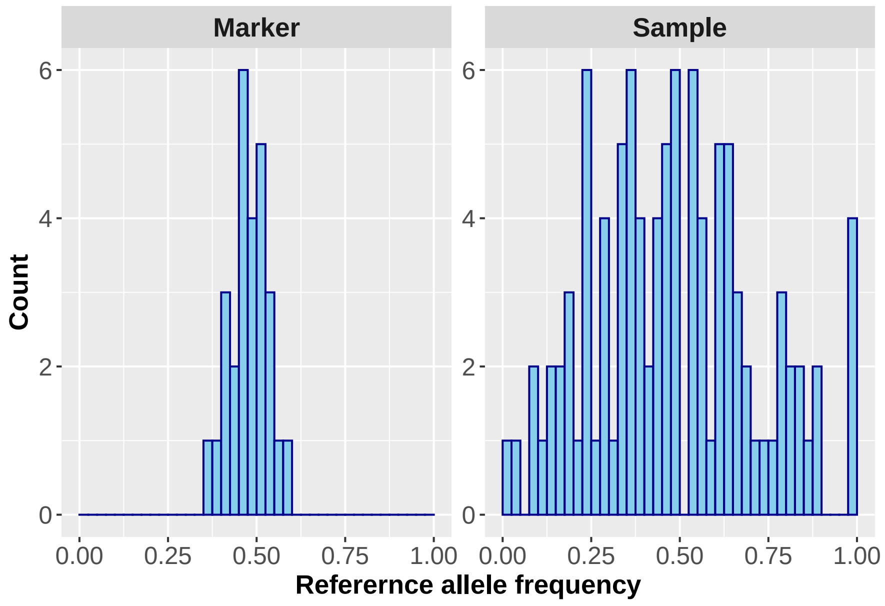
```
histGBSR(gdata, stats = "missing")
```



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

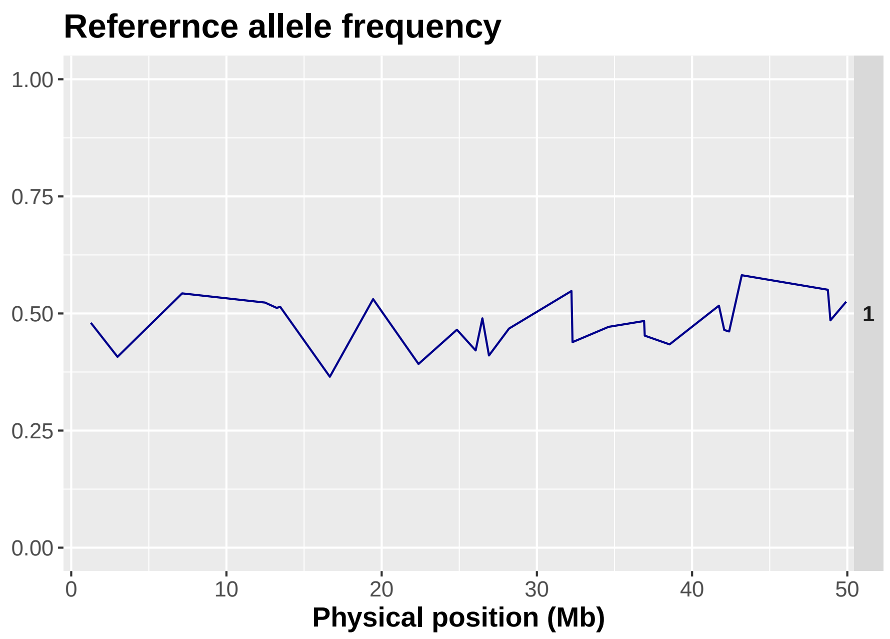


```r
histGBSR(gdata, stats = "raf")
```



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.

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



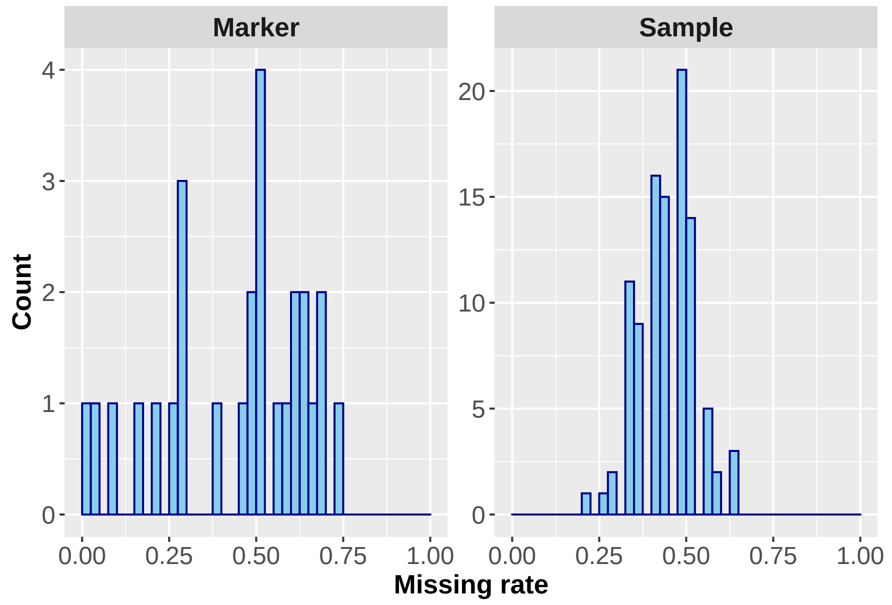
No region seem to have severe distortion. Based on the histogram of reference allele frequency, we can roughly cut off the markers with frequency more than 0.9 or less than 0.1, in other words, less than 0.1 minor allele frequency.

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

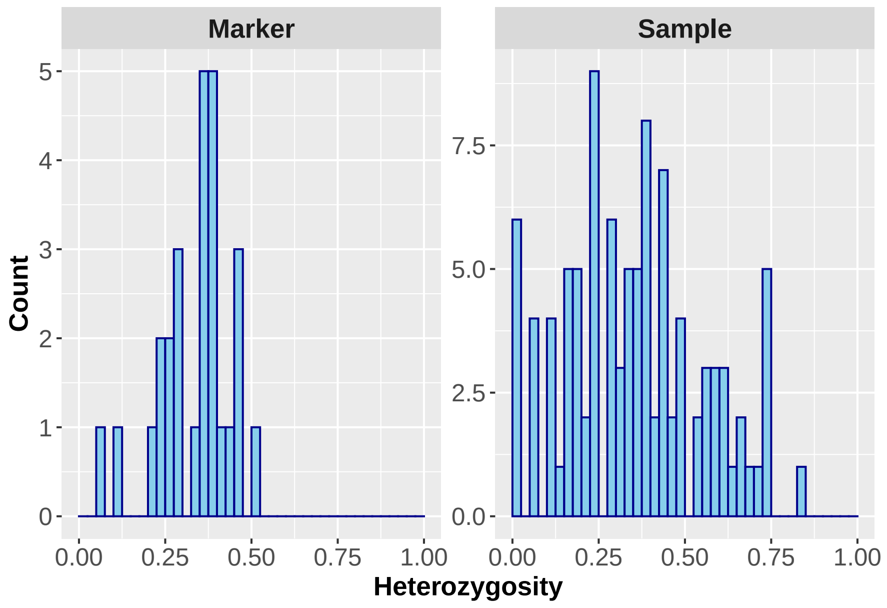
```
## [1] 27
```

Let's see the statistics again.

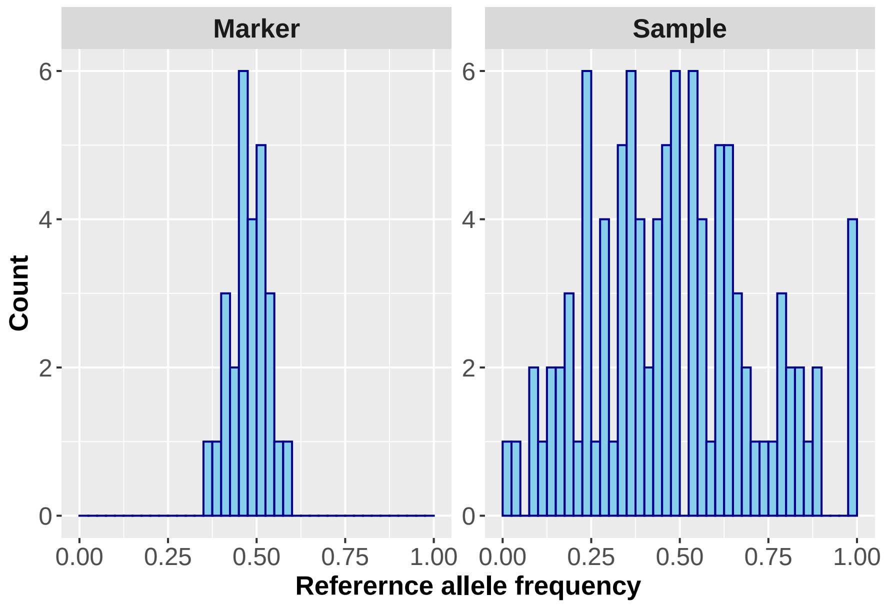
```
gdata <- countGenotype(gdata)
histGBSR(gdata, stats = "missing")
```



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

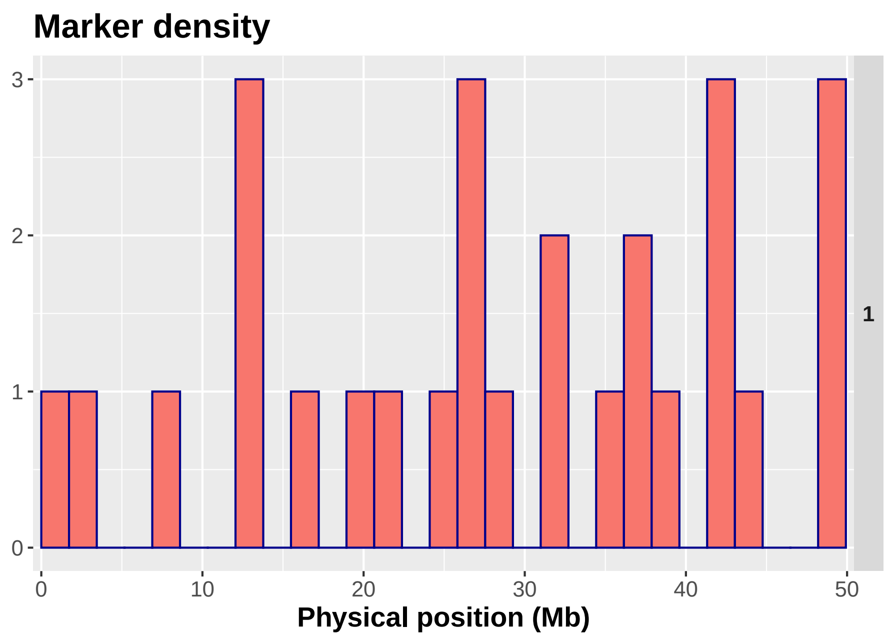


```
histGBSR(gdata, stats = "raf")
```

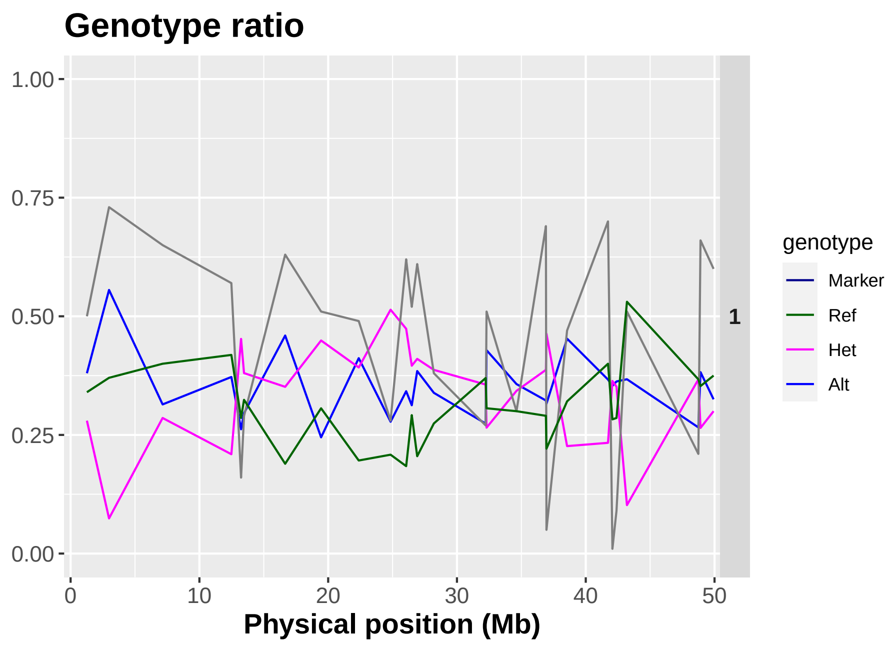


At the end of filtering, check marker density and genotype ratio per marker along chromosomes.

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



```
plotGBSR(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 = "sim_pop_subset.gds")

closeGDS(gdata)
```

`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.

Once we made a new GDS file of the subset data, we restart analysis with the subset anytime.

```
gdata <- loadGDS("sim_pop_subset.gds")
```

If you have already loaded the GDS file in the current R session, the command above will return an error. In that case, please close the connection first and then load again.

```
closeGDS(subset_gdata)
```

```
library(GBScleanR)
gdata <- loadGDS(gds_fn)
```

As we can see in the information about the GDS file when we just type the `gbsrGenotypeData` object name, the file includes the `genotype` node and the `filt.genotype` node. `loadGDS()`, also `subsetGDS()`, set the `genotype` node as genotype data. If we need `filt.genotype` which stores genotype data filtered via `setCallFilter()`, we need to run the following code.

```
p1 <- grep("Founder1", getScanID(gdata), value = TRUE)
p2 <- grep("Founder2", getScanID(gdata), value = TRUE)
gdata <- setParents(gdata, parents = c(p1, p2))
nsnp(gdata)
```

```
## [1] 87
```

To execute genotyp error correction, we first need to build a scheme object. Our simulation data was a biparental F2 population. Therefore, we should run `initScheme` and `addScheme` as following.

```
gds <- initScheme(gds, crosstype = "pairing", mating = matrix(1:2, 2))
gds <- addScheme(gds, crosstype = "selfing")
```

The function `initScheme` initializes the scheme object with information about founders. You need to specify a matrix indicating combinations of 'mating', in which each column shows a pair of parental samples. For example, if you have only two parents, the 'mating' matrix should be 'mating = matrix(1:2, nrow = 1, ncol = 2)'. The indices used in the matrix should match with the IDs labeled to parental samples by [setParents()]. The created GbsrScheme object is set in the 'scheme' slot of the GbsrGenotypeData object.

The function `addScheme` adds the information about the next breeding step of your population. In the case of our example data, the second step was selfing to produce F2 individuals from the F1 obtained via the first founder crossing. If your population was derived from a 4-way or 8-way cross, you need to add more `paring` steps. See also the help of `[addScheme()](?GBScleanR::addScheme())` function.

Now we can execute genotype estimation for error correction. GBScleanR estimates error pattern via iterative optimization of parameters for genotype estimation. We could not guess the best number of iterations, but our simulation tests showed `iter = 4` usually saturates the improvement of estimation accuracy.

```
gdata <- estGeno(gdata, iter = 4)
```

If your population derived from outbred founders, please set `het_parents = TRUE`.

```
gdata <- estGeno(gdata, het_parent = TRUE, iter = 4)
```

The larger number of iterations makes running time longer. If you would like to execute no optimization, set `optim = FALSE` or `iter = 1`.

```
# Following codes do the same.
gdata <- estGeno(gdata, iter = 1)
gdata <- estGeno(gdata, optim = FALSE)
```

All of the results of estimation are stored in the gds file linked to the `GbsrGenotypeData` object. You can obtain the estimated genotype data via the `getGenotype` function with `node = "cor"`.

```
est_geno <- getGenotype(gdata, node = "cor")
```

GBScleanR also estimates phased founder genotypes and you can access it.

```
founder_geno <- getGenotype(gdata, node = "parents")
```

GBScleanR simultaneously estimates genotype and haplotype. If you need estimated haplotype data for the samples, run `getHaplotype`.

```
est_hap <- getHaplotype(gdata)
```

The function `gbsrGDS2VCF` generate a VCF file containig the estimated genotype data and haplotype information. The estimated haplotypes are indicated in the FORMAT field with the HAP tag. The founder genotypes correspond to each haplotype are indicated in the INFO field with the PGT tag. HAP shows the pair of haplotype for each marker of each sample, while PGT shows the allele of each haplotype.

```
gbsrGDS2VCF(gdata, "simpop_est.vcf.gz")
```

```
closeGDS(gdata)
```

Session information

```
sessionInfo()
```

```
## R version 4.1.1 (2021-08-10)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 20.04.3 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3
## LAPACK: /usr/lib/x86_64-linux-gnu/openblas-pthread/liblapack.so.3
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel  stats     graphics  grDevices utils     datasets  methods
## [8] base
##
## other attached packages:
## [1] GBScleanR_0.99.0    GWASTools_1.38.0    Biobase_2.52.0
## [4] BiocGenerics_0.38.0
##
## loaded via a namespace (and not attached):
##   [1] nlme_3.1-152           bitops_1.0-7           matrixStats_0.61.0
##   [4] fs_1.5.0               usethis_2.0.1          devtools_2.4.2
##   [7] bit64_4.0.5            rprojroot_2.0.2        GenomeInfoDb_1.28.4
##  [10] tools_4.1.1            backports_1.2.1        utf8_1.2.2
##  [13] R6_2.5.1             DBI_1.1.1              mgcv_1.8-37
##  [16] colorspace_2.0-2     DNAcopy_1.66.0         withr_2.4.2
##  [19] tidyselect_1.1.1     prettyunits_1.1.1      processx_3.5.2
##  [22] bit_4.0.4            compiler_4.1.1         cli_3.0.1
##  [25] quantreg_5.86        expm_0.999-6           mice_3.13.0
##  [28] SparseM_1.81         xml2_1.3.2             desc_1.4.0
##  [31] sandwich_3.0-1       labeling_0.4.2         scales_1.1.1
##  [34] lmtest_0.9-38        quantsmooth_1.58.0     callr_3.7.0
##  [37] digest_0.6.28        commonmark_1.7         stringr_1.4.0
##  [40] GWASExactHW_1.01     rmarkdown_2.11         XVector_0.32.0
##  [43] htmltools_0.5.2      pkgconfig_2.0.3        sessioninfo_1.1.1
##  [46] fastmap_1.1.0        rlang_0.4.11           rstudioapi_0.13
##  [49] RSQLite_2.2.8        farver_2.1.0           generics_0.1.0
##  [52] zoo_1.8-9            dplyr_1.0.7            RCurl_1.98-1.5
##  [55] magrittr_2.0.1       GenomeInfoDbData_1.2.6 Matrix_1.3-4
##  [58] Rcpp_1.0.7           munsell_0.5.0          S4Vectors_0.30.1
##  [61] fansi_0.5.0          lifecycle_1.0.1        yaml_2.2.1
##  [64] stringi_1.7.4        zlibbioc_1.38.0        pkgbuild_1.2.0
##  [67] grid_4.1.1           formula.tools_1.7.1    blob_1.2.2
##  [70] crayon_1.4.1         lattice_0.20-44        Biostrings_2.60.2
##  [73] splines_4.1.1        knitr_1.36             ps_1.6.0
##  [76] pillar_1.6.3         GenomicRanges_1.44.0   logistf_1.24
##  [79] gdsfmt_1.28.1        stats4_4.1.1           pkgload_1.2.2
##  [82] glue_1.4.2           evaluate_0.14          RcppParallel_5.1.4
##  [85] data.table_1.14.2    remotes_2.4.1          operator.tools_1.6.3
##  [88] vctrs_0.3.8          testthat_3.0.4         MatrixModels_0.5-0
##  [91] gtable_0.3.0         purrr_0.3.4            tidyr_1.1.4
##  [94] SeqArray_1.32.0      cachem_1.0.6           ggplot2_3.3.5
##  [97] xfun_0.26            broom_0.7.9            roxygen2_7.1.2
## [100] survival_3.2-13      tibble_3.1.5           conquer_1.0.2
## [103] memoise_2.0.0        IRanges_2.26.0         ellipsis_0.3.2
```