Genomalicious tutorial 2: Quality control and filtering

Joshua A. Thia

25 February 2024

Preamble

Genetic data obtained from population genomic analyses is rarely perfect (although we often wish it was!). Missing data can arise from a failure to recover reads from a particular SNP position, or poor read coverage at a locus which reduces our confidence in the SNP calls at that position. We may want to remove SNPs that are below a certain depth threshold or minor allele frequency, or reduce the non-independence among loci by filtering loci from the same genomic region.

In this tutorial, you will:

1. Use *genomalicious* to investigate the distribution of missing data.
2. Use *genomalicious* to filter for samples and loci for missing data and replace missing genotypes using a "most common" genotype approach.
3. Use *genomalicious* to filter loci for minor allele frequency, read depth, and independence within genomic regions.

Load *genomalicious*

```
library(genomalicious)
```

Missing data

Population genomic datasets typically come with some amount of missingness. Some programs have built in methods that account for missing data. Others require complete datasets without missing genotypes. In the second case, we want to reduce the missingness in our dataset:

1. Loci with too much missing data can be removed.
2. Individuals with too much missing ddata can be removed.
3. Missing data can be imputed.

The choice to remove loci and/or inidivduals versus impute genotypes will depend on your dataset and what seems reasonable. Ultimately, you want to aim for a dateset with as many (independent) loci as possible, with high confidence genotypes, represented in most individuals and populations. When missingness is biased to certain loci, samples, or populations, this can be problematic. Decisions on how to deal with missingness are, however, not the subject of this lesson.

Instead, this lesson will walk you through the functions available in `genomalicious` to help you visualise and understand the distribution of missingness in your data.

We will work with a simulated data set. This data set was simulated with no missingness, so we will need to make missingness with some simple code. Our data set comprises four populations, 30 individuals per populations, and 200 independent SNP loci.

```
# Import genotype data
data(data_Genos)

# Number of samples per population
data_Genos[, length(unique(SAMPLE)), by=POP]
```

```
##     POP V1
## 1: Pop1 25
## 2: Pop2 25
## 3: Pop3 25
## 4: Pop4 25
```

```
# Number of unique loci
data_Genos$LOCUS %>% unique %>% length
```

```
## [1] 200
```

```
# Working genotype dataset
missGenos <- data_Genos %>% copy

# Add missing values.
missGenos <- do.call(
  'rbind',
  # Split data table by sample, and iterate through samples, X
  split(missGenos, by='POP') %>%
    lapply(., function(Dpop){
      pop <- Dpop$POP[1]

      if(pop=='Pop1'){
        pr <- 0.1
      } else if(pop=='Pop2'){
        pr <- 0.2
      } else if(pop %in% c('Pop3','Pop4')){
        pr <- 0.05
      }

      # Numbers and unique loci and samples
      num.loc <- Dpop$LOCUS %>% unique %>% length
      uniq.loc <- Dpop$LOCUS %>% unique
      num.samp <- Dpop$SAMPLE %>% unique %>% length
      uniq.samp <- Dpop$SAMPLE %>% unique

      # Vector of missingness
      num.miss <- rbinom(n=num.samp, size=num.loc, prob=pr)

      # Iterate through samples and add unique loci
      for(i in 1:num.samp){
        locs <- sample(uniq.loc, size=num.miss[i], replace=FALSE)
        Dpop[SAMPLE==uniq.samp[i] & LOCUS%in%locs, GT:=NA]
```

```
      }

      # Return
      return(Dpop)
      }
    )
  )
```

Note in the above code the use of the `%>%`. This is called a **pipe**, and is from the `magrittr` package. Pipes are incredibly useful for moving data from function to function without having to save intermediate objects.

Anyway, we have now created a `data.table` classed object called `missGenos`, which will be our working data set. We have simulated 10% missingness from Pop1, 20% missingness in Pop2, and 5% missingness from Pop3 and Pop4, per sample.

Note, your results may be a little different to those simulated here because of the randomisation in my code.

Let's take a look:

```
# Per sample missingness in each population
missGenos[, .(MISS.PROP=sum(is.na(GT))/length(GT)), by=c('POP','SAMPLE')] %>%
  .[, .(MISS.PROP.MEAN=mean(MISS.PROP)), by=POP]
```

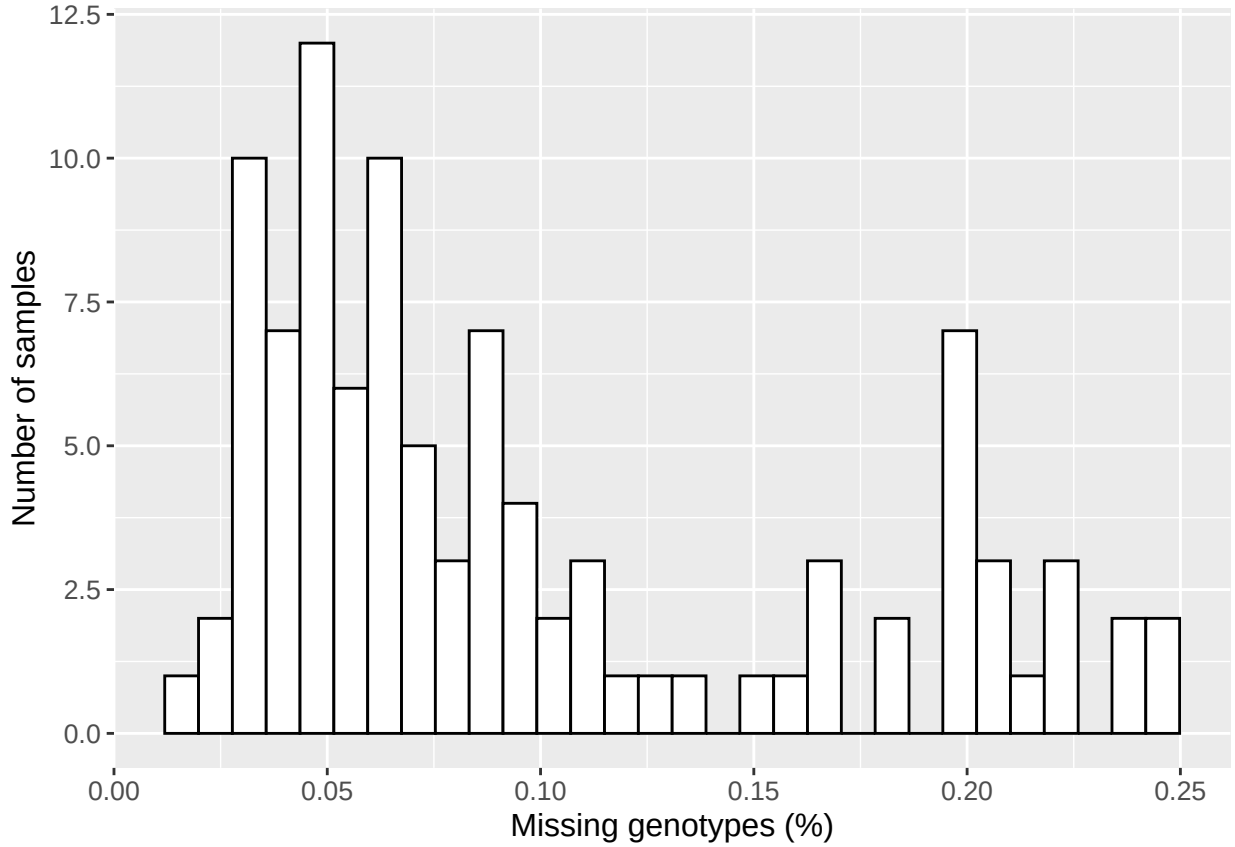
```
##     POP MISS.PROP.MEAN
## 1: Pop1         0.0952
## 2: Pop2         0.2002
## 3: Pop3         0.0512
## 4: Pop4         0.0442
```

```
# Per locus missingness in each population
missGenos[, .(MISS.PROP=sum(is.na(GT))/length(GT)), by=c('POP','LOCUS')] %>%
  .[, .(MISS.PROP.MEAN=mean(MISS.PROP)), by=POP]
```

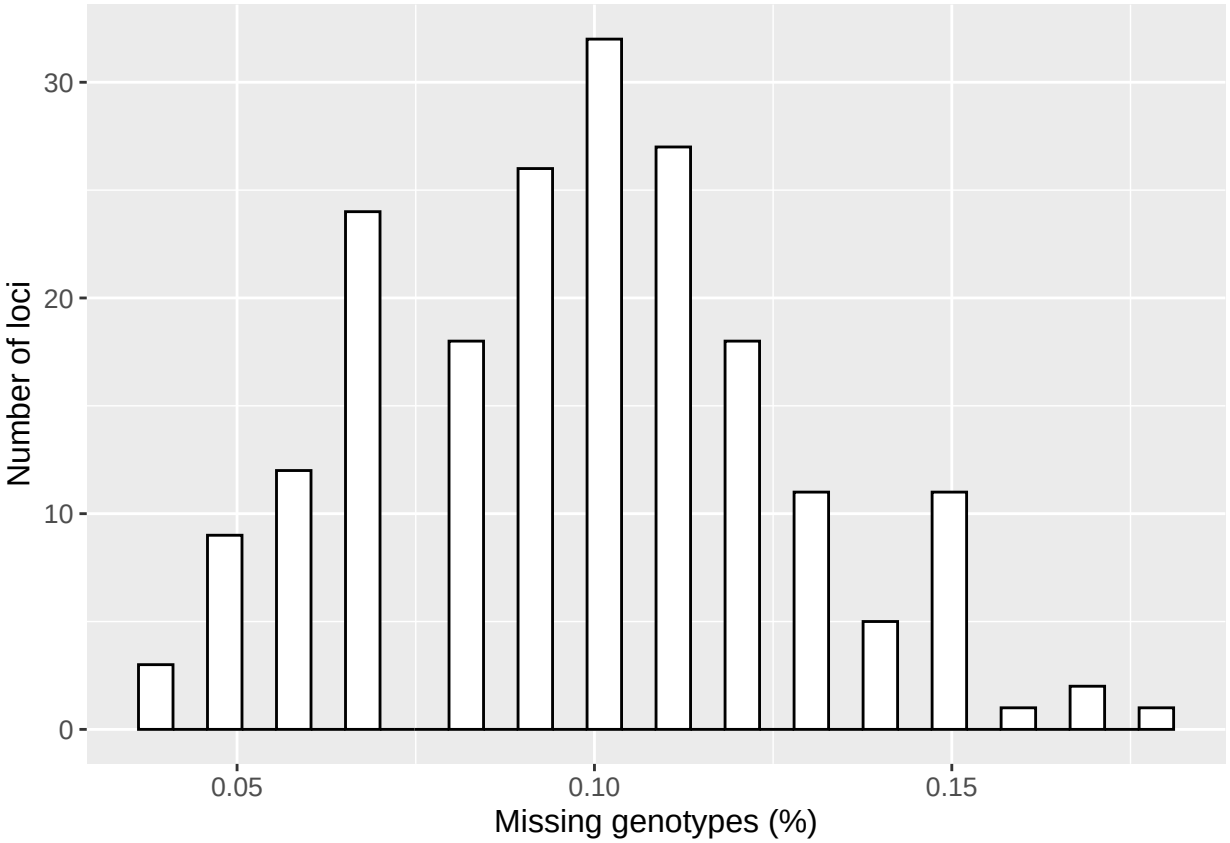
```
##     POP MISS.PROP.MEAN
## 1: Pop1         0.0952
## 2: Pop2         0.2002
## 3: Pop3         0.0512
## 4: Pop4         0.0442
```

Let's first create a histogram of the distribution of missingness using the `missHist` function. This function has a number of ways slice the data and visualise the distribution of missingness. It also has features that allow us to customise the output to suit our aesthetic tastes.

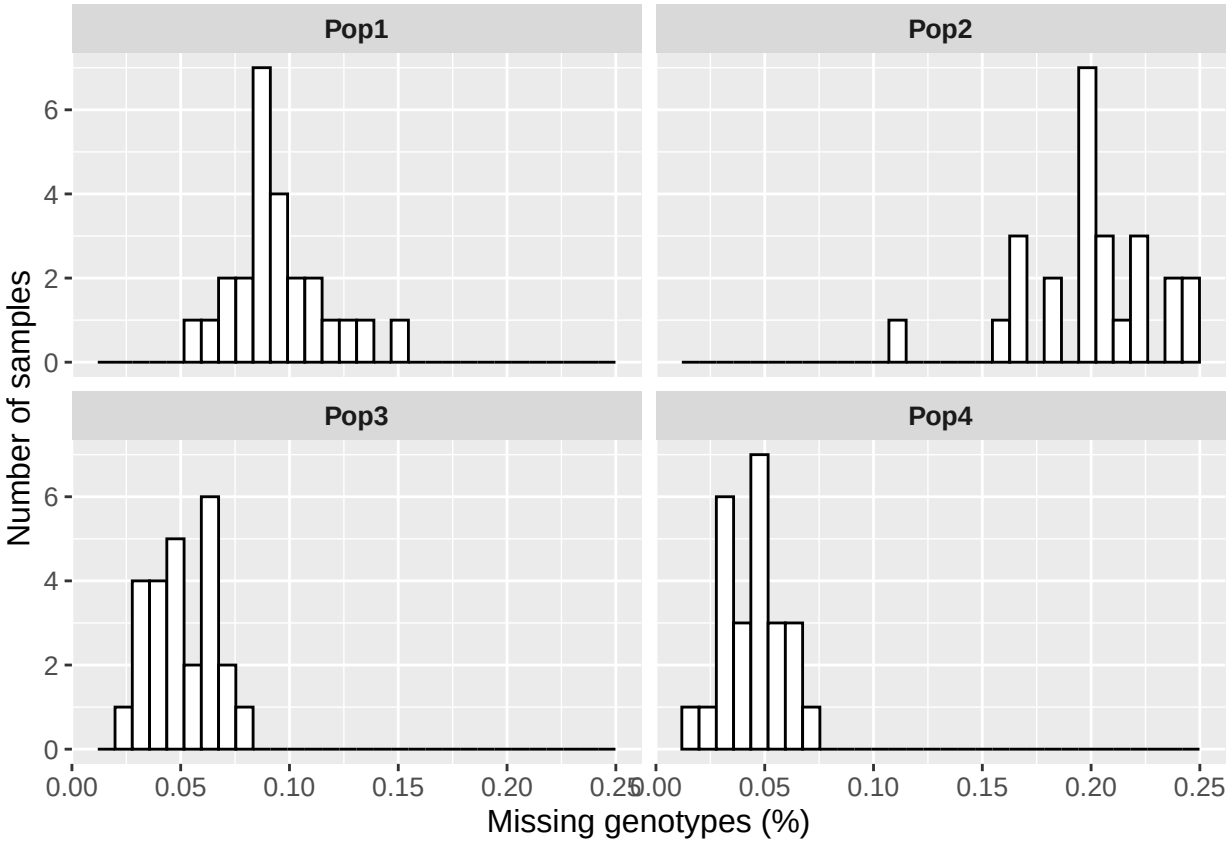
```
# We need to specify whether we want the histogram to summarise missingness
# across samples or across loci.
plot_miss_hist_samps <- missHist(missGenos, plotBy='samples')
plot_miss_hist_samps
```



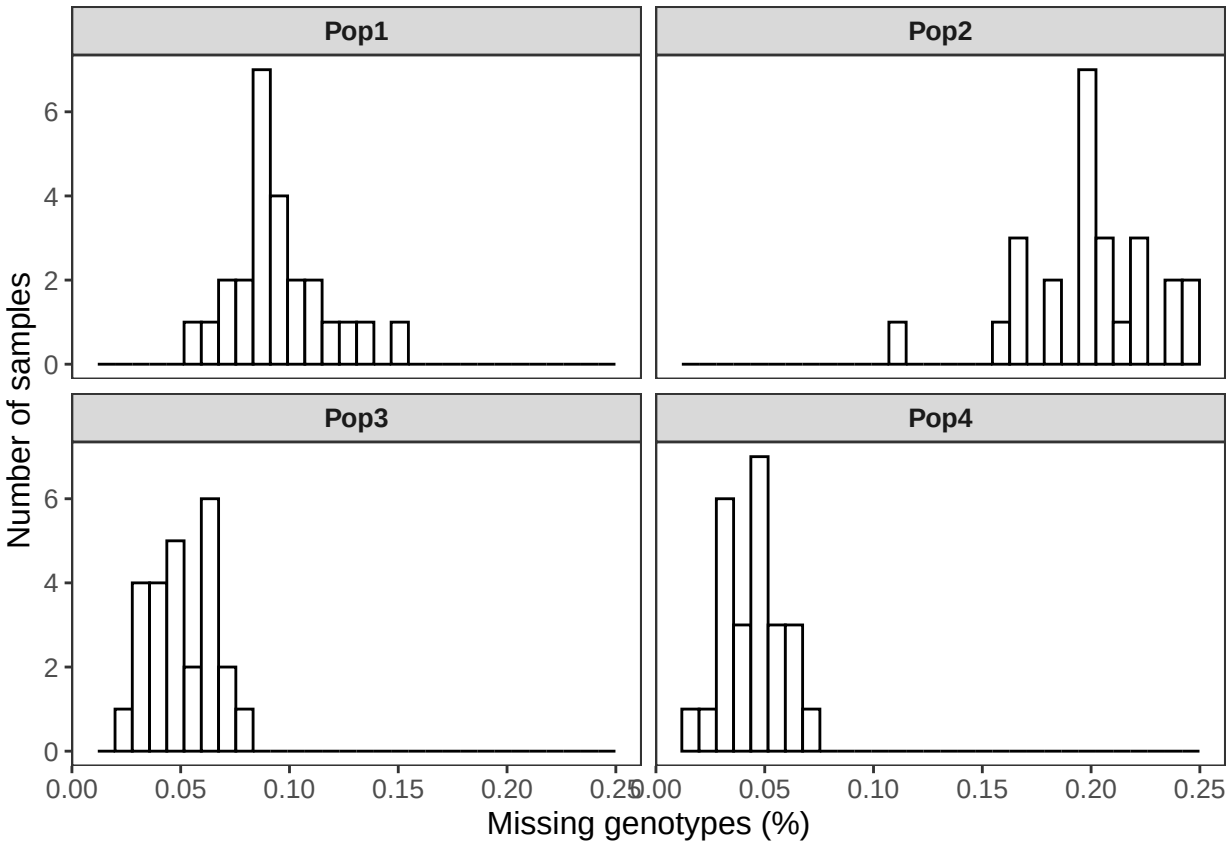

```
plot_miss_hist_loci <- missHist(missGenos, plotBy='loci')
plot_miss_hist_loci
```



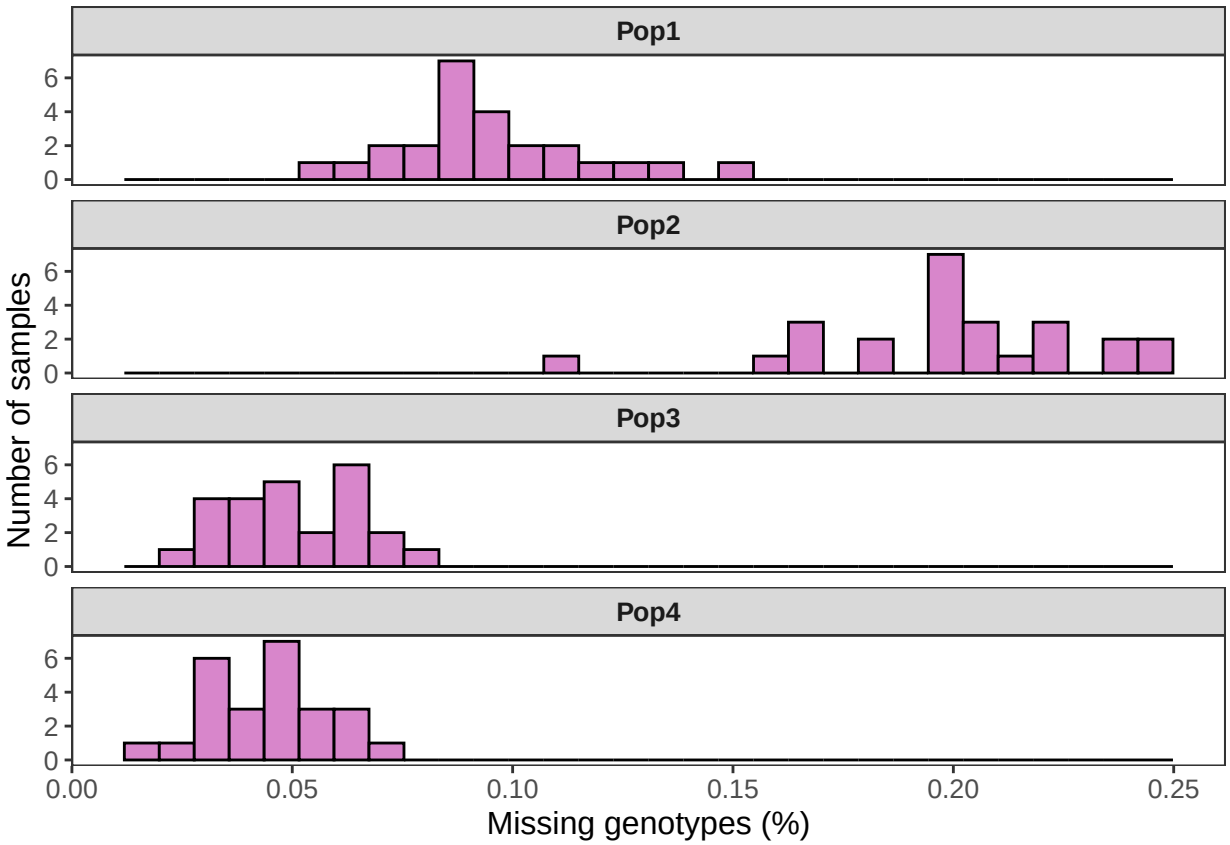

```
# By default, the arguments `sampCol`=='SAMPLE', `locusCol`=='LOCUS' and
# `genoCol`=='GT'. The argument `popCol` is NA by default.
# But if we specify a population # column, then we can get our results plotted
# with respect to population.
plot_miss_hist_pop <- missHist(missGenos, plotBy='samples', popCol='POP')
plot_miss_hist_pop
```




```
# By default, missHist produces a ggplot style plot, but we can produce a classic
# R style plot by specifying the `look` argument.
plot_miss_hist_pop_classic <- missHist(
  missGenos, plotBy='samples', popCol='POP', look='classic'
  )
plot_miss_hist_pop_classic
```

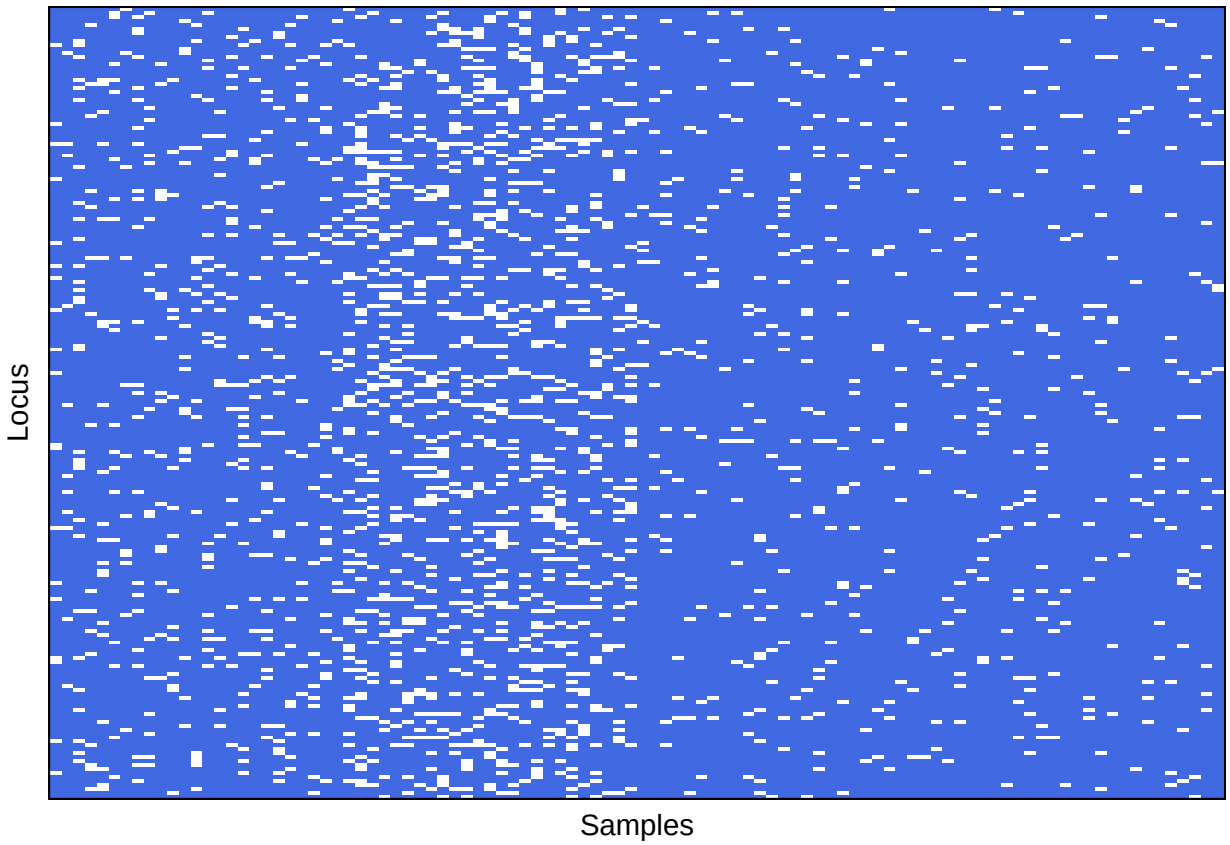



```
# We also have options to manipulate the colours of bars (`plotColours`) in
# the histogram and the grid number of columns in the grid (`plotNCol`).
plot_miss_hist_pop_colour <- missHist(
  missGenos, plotBy='samples', popCol='POP', look='classic',
  plotColours='#D886CB', plotNCol=1
  )
plot_miss_hist_pop_colour
```

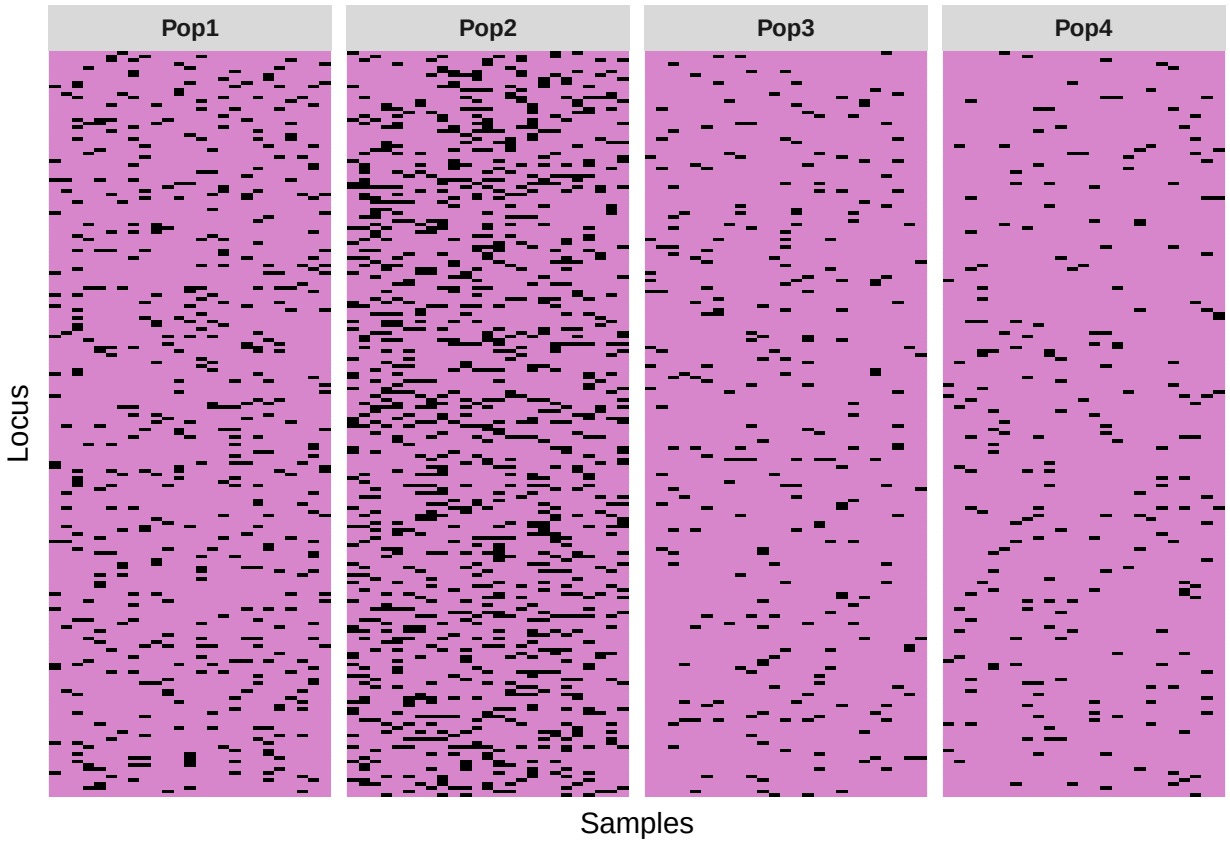


We may be interested in visualising the patterns of missingness among loci, samples, and populations. We can do this using the `missHeatmap` function. This function also has several features to customise the style, colours and grid structure to suit your aesthetic.

```
# The default output, with default arguments sampCol`=='SAMPLE', `locusCol`=='LOCUS'
# and `genoCol`=='GT'.
plot_heat <- missHeatmap(missGenos)
plot_heat
```




```
# Now, split by population. Make missing values black and non-missing values
# pink. Use 4 columns in the grid strucutre.
plot_heat_colours <- missHeatmap(
  missGenos, popCol='POP', plotColours=c('black','#D886CB'),
  plotNCol=4
  )
plot_heat_colours
```



Filtering missing data

There are two ways to filter missing data using *genomalicious*. The filtering functions return lists of "good loci", those that pass the function's missingness threshold crtieria.

Filtering by loci is achieved using the `filter_missing_loci` function. This function calculates missingness for each locus and removes loci based on a missingness threshold. This threshold is set with the `missingness` argument, which takes a value betwee 0 and 1.

`filter_missing_loci` can filter either a long-format `data.table` object of individual genotypes or population allele frequencies. We have to specify the type of data using the `type` argument, e.g., `type=="genos"` or `type=="freqs"`, respectively.

For individually sequenced genotypes, we have two ways of filtering loci using the `method` argument. When `method=="samples"`, missingness at each locus is calculated across all samples, and if this value is above the threshold, the locus will be removed. When `method=="pop"`, missingness at each locus is calculated for each population, and if any population has missingness above the threshold, the locus will be removed.

Let's start with filtering loci for individually genotypes samples.

```
# Filter across all samples, irrespective of population
good_loci_genos_samps <- filter_missing_loci(
  missGenos, missing=0.1, type="genos", method="samples",
  sampCol='SAMPLE', locusCol='LOCUS', genoCol='GT'
  )

# Filter if any population does not meet the threshold
```

```
good_loci_genos_pops <- filter_missing_loci(
  missGenos, missing=0.1, type="genos", method="pops",
  sampCol='SAMPLE', locusCol='LOCUS', genoCol='GT', popCol='POP'
  )

# Take a look at the number of loci and the overlapping (intersecting) loci
length(good_loci_genos_samps)
```

```
## [1] 124
```

```
length(good_loci_genos_pops)
```

```
## [1] 8
```

```
intersect(good_loci_genos_samps, good_loci_genos_pops)
```

```
## [1] "Contig795_251"  "Contig3113_10"  "Contig3846_496" "Contig4649_247"
## [5] "Contig4714_354" "Contig4826_136" "Contig4909_463" "Contig6831_64"
```

```
# Subset the genotypes based on the vector of good loci
missGenos[LOCUS %in% good_loci_genos_samps]
```

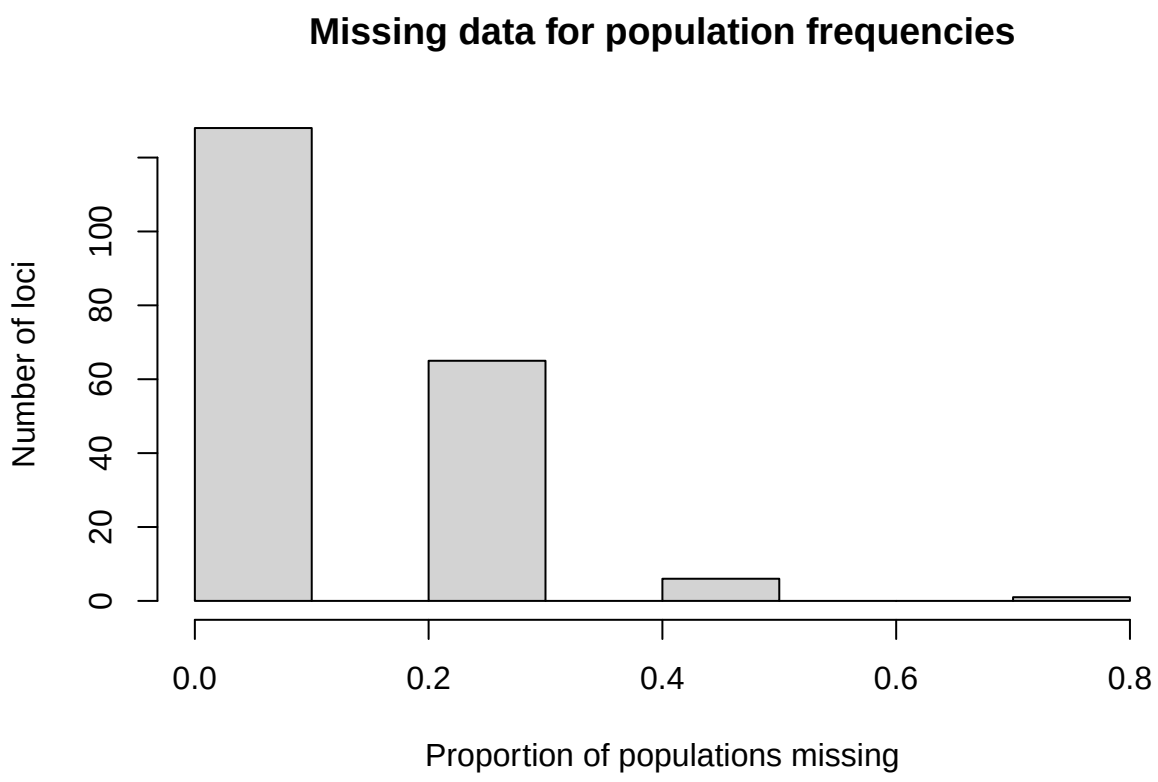
```
##             CHROM POS          LOCUS  POP  SAMPLE GT DP AO RO ALT REF
##     1:    Contig1 437    Contig1_437 Pop1  Ind1_1  1 27 11 16   A   T
##     2:   Contig38 427   Contig38_427 Pop1  Ind1_1  0 46  0 46   A   G
##     3:  Contig183 250  Contig183_250 Pop1  Ind1_1  2 44 44  0   C   A
##     4:  Contig263 267  Contig263_267 Pop1  Ind1_1  2 24 24  0   T   A
##     5:  Contig346 278  Contig346_278 Pop1  Ind1_1  1 47 21 26   T   G
##    ---
## 12396: Contig7239 219 Contig7239_219 Pop4 Ind4_25  0 27  0 27   C   G
## 12397: Contig7274 197 Contig7274_197 Pop4 Ind4_25  1 52 24 28   G   C
## 12398: Contig7291  92  Contig7291_92 Pop4 Ind4_25  2 39 39  0   G   T
## 12399: Contig7293 124 Contig7293_124 Pop4 Ind4_25  2 30 30  0   A   T
## 12400: Contig7299 279 Contig7299_279 Pop4 Ind4_25  1 49 15 34   G   T
```

```
missGenos[LOCUS %in% good_loci_genos_pops]
```

```
##           CHROM POS          LOCUS  POP  SAMPLE GT DP AO RO ALT REF
##   1:  Contig795 251  Contig795_251 Pop1  Ind1_1  1 42 25 17   C   G
##   2: Contig3113  10  Contig3113_10 Pop1  Ind1_1  1 25 11 14   C   T
##   3: Contig3846 496 Contig3846_496 Pop1  Ind1_1  0 42  0 42   A   T
##   4: Contig4649 247 Contig4649_247 Pop1  Ind1_1  0 26  0 26   A   G
##   5: Contig4714 354 Contig4714_354 Pop1  Ind1_1  2 41 41  0   T   A
##  ---
## 796: Contig4649 247 Contig4649_247 Pop4 Ind4_25  0 43  0 43   A   G
## 797: Contig4714 354 Contig4714_354 Pop4 Ind4_25  1 51 27 24   T   A
## 798: Contig4826 136 Contig4826_136 Pop4 Ind4_25  0 27  0 27   A   G
## 799: Contig4909 463 Contig4909_463 Pop4 Ind4_25  0 27  0 27   G   T
## 800: Contig6831  64  Contig6831_64 Pop4 Ind4_25  2 25 25  0   A   G
```

Filtering populations allele frequency data is a bit different. In this case, the samples are populations (not individuals), and missingness is calculated as the proportion of populations for which allele frequencies are available. Let's take a look.

```
# Make some missing population frequency data from population pool-seq data.
data("data_PoolFreqs")

missFreqs <- data_PoolFreqs %>% copy
missFreqs$FREQ[sample(1:nrow(missFreqs), size=0.1*nrow(missFreqs), replace=FALSE)] <- NA

# Take a look at the missing data (note, this may look different on your system)
missFreqs.props <- missFreqs[, .(MISS.PROP=sum(is.na(FREQ))/length(FREQ)), by=c('LOCUS')]

missFreqs.props$MISS.PROP %>%
  hist(
    .,
    main='Missing data for population frequencies',
    xlab='Proportion of populations missing',
    ylab='Number of loci'
    )
```




```
# Filter
good_loci_freqs <- filter_missing_loci(
  missFreqs, missing=0.1, type='freqs',
  locusCol='LOCUS', freqCol='FREQ', popCol='POOL'
  )
```

```r
length(good_loci_freqs)
```

```
## [1] 128
```

```r
# Subset for the good loci
missFreqs[LOCUS %in% good_loci_freqs]
```

```
##           CHROM POS          LOCUS ALT REF  POP       FREQ DP AO RO POOL
##   1:    Contig1 437    Contig1_437   A   T Pop1 0.24193548 62 15 47    1
##   2:  Contig170  36   Contig170_36   G   A Pop1 0.69879518 83 58 25    1
##   3:  Contig183 250  Contig183_250   C   A Pop1 0.93023256 43 40  3    1
##   4:  Contig263 267  Contig263_267   T   A Pop1 0.31168831 77 24 53    1
##   5:  Contig425 300  Contig425_300   G   C Pop1 0.59493671 79 47 32    1
##  ---
## 508: Contig7220 287 Contig7220_287   T   C Pop4 0.23880597 67 16 51    4
## 509: Contig7239 219 Contig7239_219   C   G Pop4 0.32432432 74 24 50    4
## 510: Contig7287  10  Contig7287_10   T   C Pop4 0.02127660 47  1 46    4
## 511: Contig7291  92  Contig7291_92   G   T Pop4 0.05882353 51  3 48    4
## 512: Contig7293 124 Contig7293_124   A   T Pop4 0.82089552 67 55 12    4
```

We can also filter missing data by sample, removing samples with too many loci missing genotypes or population allele frequencies. The necessary function is `filter_missing_units`. In this case, the "units" refers to the sampled unit, individuals or populations. The parameterisation of `filter_missing_units` is basically identical to that of `filter_missing_loci` that we explored above. Except this time, missing data is calculated across loci (instead of samples).

For `filter_missing_units`, if `type=="genos"`, there are again two ways to parameterise the argument `method`. If `method=="samples"`, then missingness is calculated across loci for each sample, irrespective of its populations, and any sample with missingness above the threshold is removed. If `method=="pops"`, then the mean missingness is calculated per population, and any population with missingness above the threshold is removed.

```r
# Filter by samples, irrespective of populations. Returns vector of individuals.
good_units_samps <- filter_missing_units(
  missGenos, missing=0.1, type='genos', method='samples',
  sampCol='SAMPLE', locusCol='LOCUS', genoCol='GT'
)

# Filter by populations means. Returns vector of populations.
good_units_pops <- filter_missing_units(
  missGenos, missing=0.1, type='genos', method='pops',
  sampCol='SAMPLE', locusCol='LOCUS', genoCol='GT', popCol='POP'
)

# Subset samples
missGenos[SAMPLE %in% good_units_samps]
```

```
##             CHROM POS          LOCUS  POP  SAMPLE GT DP AO RO ALT REF
##     1:    Contig1 437    Contig1_437 Pop1  Ind1_1  1 27 11 16   A   T
##     2:   Contig34 213   Contig34_213 Pop1  Ind1_1  1 22 17  5   G   A
##     3:   Contig38 427   Contig38_427 Pop1  Ind1_1  0 46  0 46   A   G
##     4:  Contig152 389  Contig152_389 Pop1  Ind1_1  0 19  0 19   T   A
```

```
##     5:  Contig170  36   Contig170_36 Pop1  Ind1_1  1 38 20 18   G   A
##    ---
## 13396: Contig7274 197 Contig7274_197 Pop4 Ind4_25  1 52 24 28   G   C
## 13397: Contig7287  10  Contig7287_10 Pop4 Ind4_25  0 59  0 59   T   C
## 13398: Contig7291  92  Contig7291_92 Pop4 Ind4_25  2 39 39  0   G   T
## 13399: Contig7293 124 Contig7293_124 Pop4 Ind4_25  2 30 30  0   A   T
## 13400: Contig7299 279 Contig7299_279 Pop4 Ind4_25  1 49 15 34   G   T
```

```
# Subset populations
missGenos[POP %in% good_units_pops]
```

```
##             CHROM POS          LOCUS  POP  SAMPLE GT DP AO RO ALT REF
##     1:    Contig1 437    Contig1_437 Pop1  Ind1_1  1 27 11 16   A   T
##     2:   Contig34 213   Contig34_213 Pop1  Ind1_1  1 22 17  5   G   A
##     3:   Contig38 427   Contig38_427 Pop1  Ind1_1  0 46  0 46   A   G
##     4:  Contig152 389  Contig152_389 Pop1  Ind1_1  0 19  0 19   T   A
##     5:  Contig170  36   Contig170_36 Pop1  Ind1_1  1 38 20 18   G   A
##    ---
## 14996: Contig7274 197 Contig7274_197 Pop4 Ind4_25  1 52 24 28   G   C
## 14997: Contig7287  10  Contig7287_10 Pop4 Ind4_25  0 59  0 59   T   C
## 14998: Contig7291  92  Contig7291_92 Pop4 Ind4_25  2 39 39  0   G   T
## 14999: Contig7293 124 Contig7293_124 Pop4 Ind4_25  2 30 30  0   A   T
## 15000: Contig7299 279 Contig7299_279 Pop4 Ind4_25  1 49 15 34   G   T
```

And similarly, we can use `type=="freqs"` to remove populations from a data table of allele frequencies.

```
# Filter populations with too many missing loci
good_units_pops_freqs <- filter_missing_units(
  missFreqs, missing=0.1, type='freqs',
  locusCol='LOCUS', freqCol='FREQ', popCol='POOL'
)

good_units_pops_freqs
```

```
## [1] "Pop3" "Pop4"
```

```
# Subset good populations
missFreqs[POOL %in% good_units_pops_freqs]
```

```
## Empty data.table (0 rows and 11 cols): CHROM,POS,LOCUS,ALT,REF,POP...
```

Replacing missing genotypes

Sometimes you cannot filter missing data completely without compromising your dataset. In which case, missing genotypes need to be replaced with estimated values. Ideally, this would be done via imputation methods. However, if you are in a hurry and need to run some quick analyses, it might be easier to replace missing values using the most common genotype

The *genomalicious* function `replace_miss_genos` can be used to replace missing genotypes. How this function operate depends on how the argument `popCol` is parameterised. By default, `popCol` is NULL, so the function will identify the most common genotypes across all individuals for each locus. If, however, `popCol` is given a value, a population ID column, then the most common genotype for each locus will be calculated per population.

```
D <- data_Genos %>% copy

# Sites with missing data
D[sample(1:nrow(D), round(0.1*nrow(D)), FALSE), GT:=NA] %>%
 setnames(., 'GT', 'GT.MISS')

# Replace across individuals
D.rep.inds <- replace_miss_genos(
   dat=D, sampCol='SAMPLE', locusCol='LOCUS', genoCol='GT.MISS'
) %>%
   setnames(., 'GT', 'GT.INDS')

# Replace within populations
D.rep.pops <- replace_miss_genos(
   dat=D, sampCol='SAMPLE', locusCol='LOCUS', genoCol='GT.MISS', popCol='POP'
) %>%
   setnames(., 'GT', 'GT.POPS')

# Tabulate comparisons between methods
compReplace <- left_join(
   data_Genos[, c('LOCUS','SAMPLE','POP','GT')],
   D[, c('LOCUS','SAMPLE','POP','GT.MISS')]
) %>%
.[is.na(GT.MISS), !'GT.MISS'] %>%
   left_join(., D.rep.inds[,c('LOCUS','SAMPLE','POP','GT.INDS')]) %>%
   left_join(., D.rep.pops[,c('LOCUS','SAMPLE','POP','GT.POPS')])

# Number of correct matches is slightly higher when using the most
# common genotype within populations
compReplace[GT==GT.INDS] %>% nrow
```

```
## [1] 1305
```

```
compReplace[GT==GT.POPS] %>% nrow
```

```
## [1] 1345
```

Filtering for minor allele frequency, read depth, and "unlinked" loci

There are additional filtering functions available in *genomalicious*. It is important to note that the functions in this section *cannot* have handle missing data. Missing data therefore need to be removed or imputed before applying these functions.

Minor allele frequency (MAF) filtering can be performed with the function, `filter_maf`. The MAF threshold is set using the argument `maf`. This function filer a long-format `data.table` object of individual genotypes or population allele frequencies, specified with `type=="genos"` and `type=="freqs"`, respectively.

`filter_maf` filters loci via one of two methods specified with the argument `method`. When `method=="mean"`, the MAF is calculated as the mean across populations, and any locus with the mean MAF below the threshold is removed. When `method=="any_pop"`, the MAF is calcualted for each population and if any population has a MAF below the threshold, the locus will be removed.

```r
# Loci with good MAF, taking the mean across populations
good_maf_genos_mean <- filter_maf(
  dat=data_Genos, maf=0.1, type='genos', method='mean',
  sampCol='SAMPLE', locusCol='LOCUS', genoCol='GT'
  )

length(good_maf_genos_mean)
```

```
## [1] 151
```

```r
# Loci with good MAF, where all populations meet the threshold
good_maf_genos_pops <- filter_maf(
  dat=data_Genos, maf=0.1, type='genos', method='any_pop',
  sampCol='SAMPLE', locusCol='LOCUS', genoCol='GT'
  )

length(good_maf_genos_pops)
```

```
## [1] 79
```

```r
# Overlap
intersect(good_maf_genos_mean, good_maf_genos_pops)
```

```
##  [1] "Contig1_437"    "Contig34_213"   "Contig170_36"   "Contig425_300"
##  [5] "Contig428_69"   "Contig437_496"  "Contig530_173"  "Contig554_17"
##  [9] "Contig606_50"   "Contig615_414"  "Contig716_299"  "Contig893_216"
## [13] "Contig1047_30"  "Contig1219_313" "Contig1323_488" "Contig1342_118"
## [17] "Contig1375_453" "Contig1384_494" "Contig1462_463" "Contig1774_449"
## [21] "Contig1790_160" "Contig1835_85"  "Contig1845_367" "Contig1915_343"
## [25] "Contig2068_434" "Contig2074_321" "Contig2078_224" "Contig2308_122"
## [29] "Contig2463_437" "Contig2491_445" "Contig2527_135" "Contig2541_330"
## [33] "Contig2578_379" "Contig2632_494" "Contig2929_19"  "Contig3208_102"
## [37] "Contig3244_166" "Contig3329_37"  "Contig3362_334" "Contig3469_425"
## [41] "Contig3478_180" "Contig3649_5"   "Contig3658_313" "Contig3744_293"
## [45] "Contig3833_198" "Contig3913_87"  "Contig3989_152" "Contig4042_277"
## [49] "Contig4178_445" "Contig4228_158" "Contig4421_230" "Contig4443_393"
## [53] "Contig4634_357" "Contig4698_463" "Contig4714_354" "Contig4767_166"
## [57] "Contig4816_474" "Contig5055_174" "Contig5248_491" "Contig5310_284"
## [61] "Contig5339_353" "Contig5411_207" "Contig5658_55"  "Contig5678_407"
## [65] "Contig6068_55"  "Contig6091_132" "Contig6108_470" "Contig6194_91"
## [69] "Contig6213_140" "Contig6240_462" "Contig6443_129" "Contig6616_329"
## [73] "Contig6679_335" "Contig6724_475" "Contig6831_64"  "Contig7170_244"
## [77] "Contig7239_219" "Contig7274_197" "Contig7299_279"
```

```r
# Filter a data table of population allele frequencies
good_maf_freqs <- filter_maf(
  dat=data_PoolFreqs, maf=0.1, type='freqs',
  popCol='POOL', locusCol='LOCUS', freqCol='FREQ'
)

length(good_maf_freqs)
```

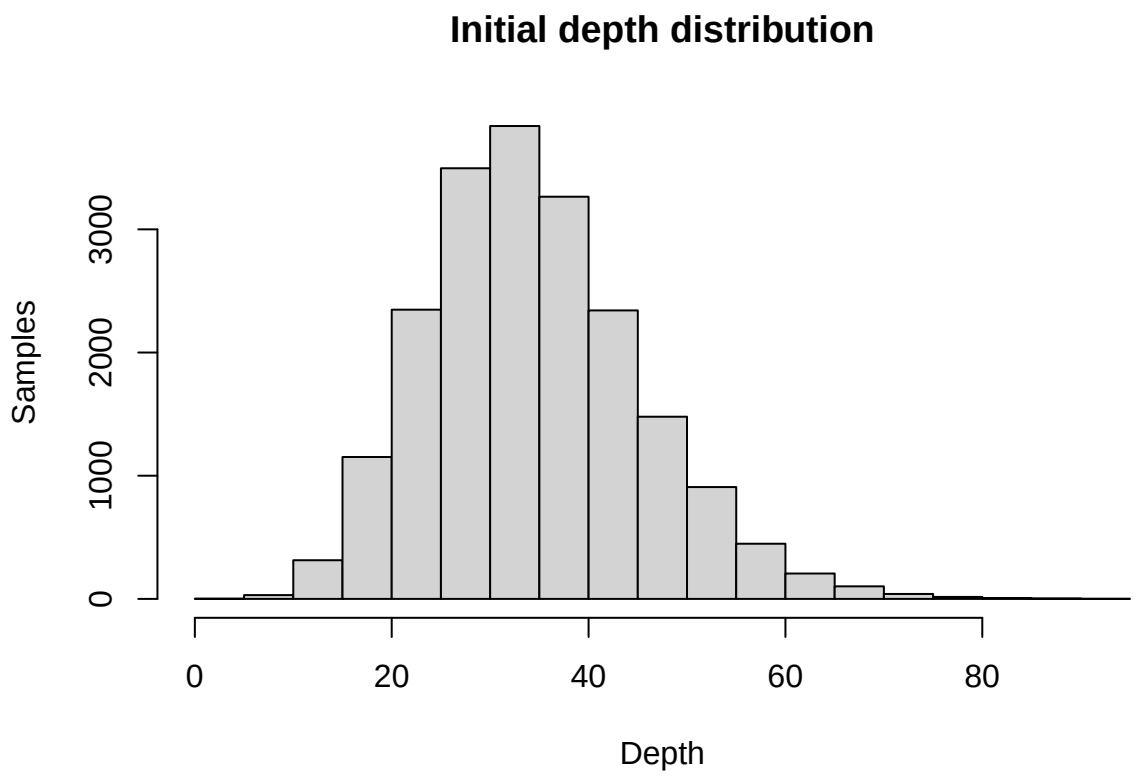
```
## [1] 144
```

Read depth filtering can be performed using the function `filter_depth`. This function is relatively straight forward in that it simply requires a `data.table` object with depth values for different loci in different samples. The key parameterisations are the minimum depth, `minDP`, and maximum depth, `maxDP`. You can set either, or both of these arguments to filter read depth.

```
# Take a look at the depth distribution before filtering
data_Genos$DP %>% summary
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    3.00   27.00   34.00   34.94   42.00   91.00
```

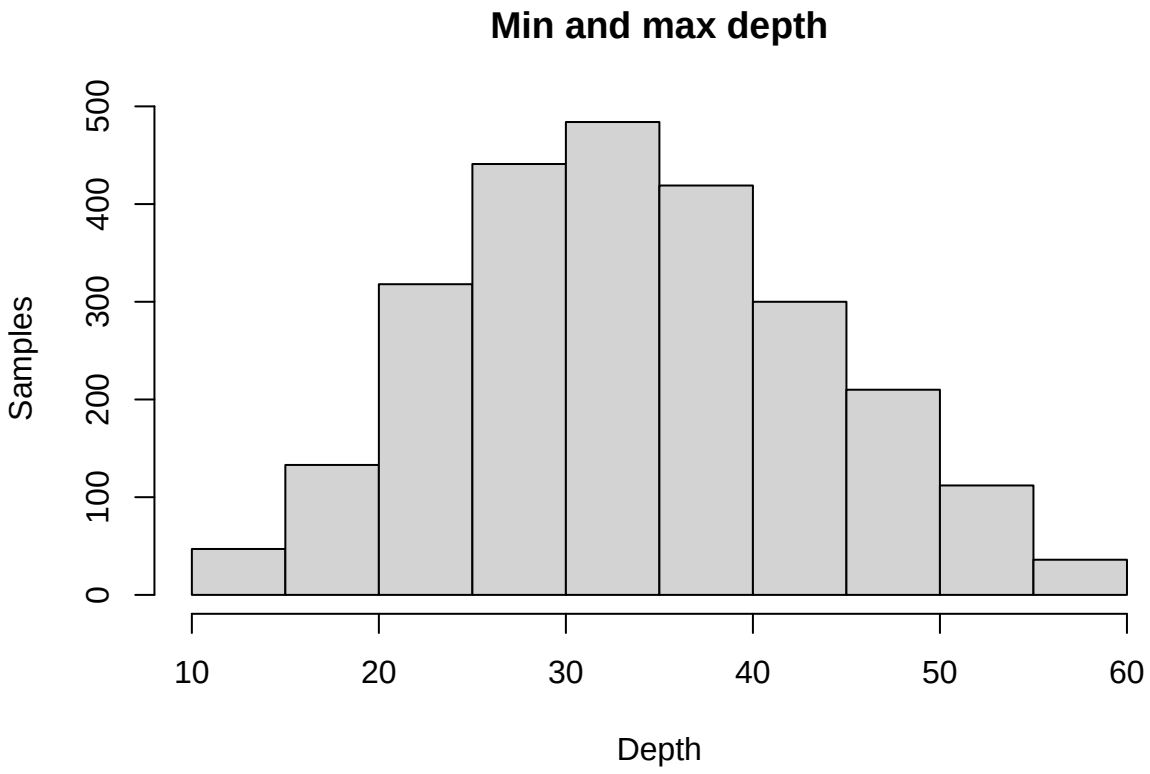
```
data_Genos$DP  %>%
  hist(., main='Initial depth distribution', xlab='Depth', ylab='Samples')
```




```
# Filtering both minimum and maximum
good_loci_dp_both <- filter_depth(
  data_Genos, minDP=10, maxDP=60,
  locusCol='LOCUS', dpCol='DP'
  )

length(good_loci_dp_both)
```

```
## [1] 25
```

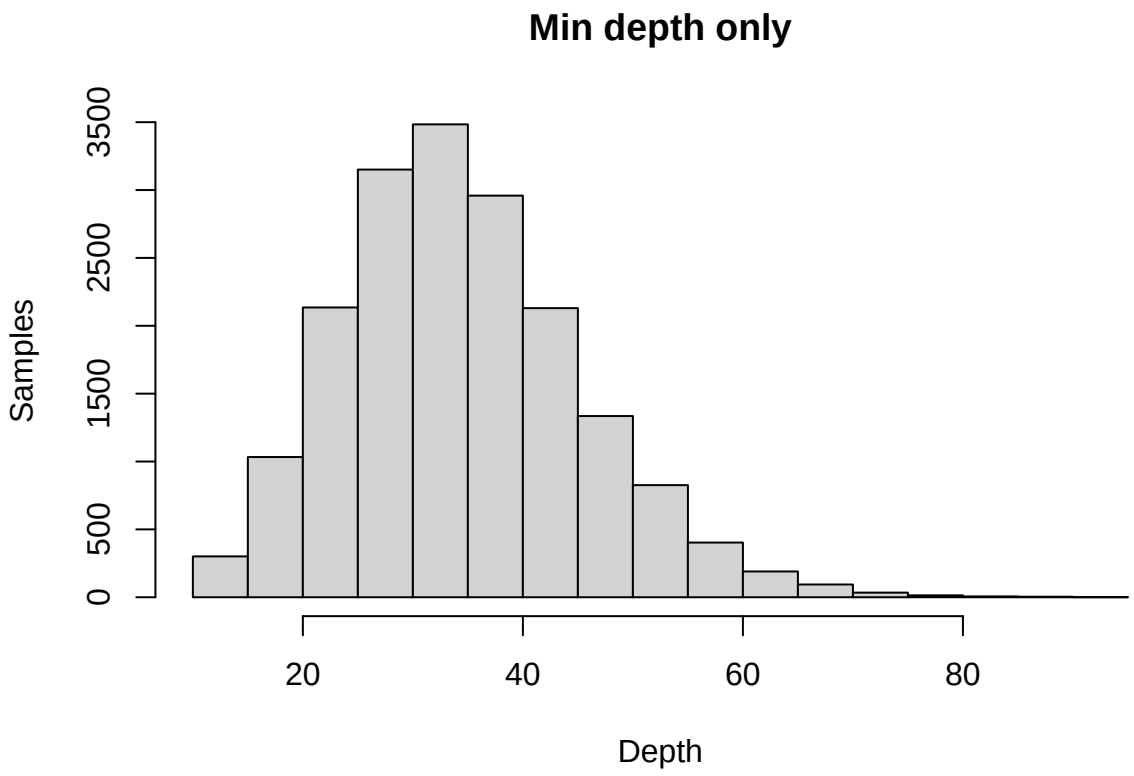
```
data_Genos[LOCUS %in% good_loci_dp_both]$DP %>%
  hist(., main='Min and max depth', xlab='Depth', ylab='Samples')
```




```
# Filtering minimum only
good_loci_dp_min <- filter_depth(
  data_Genos, minDP=10, maxDP=NULL,
  locusCol='LOCUS', dpCol='DP'
  )

length(good_loci_dp_min)
```

```
## [1] 181
```

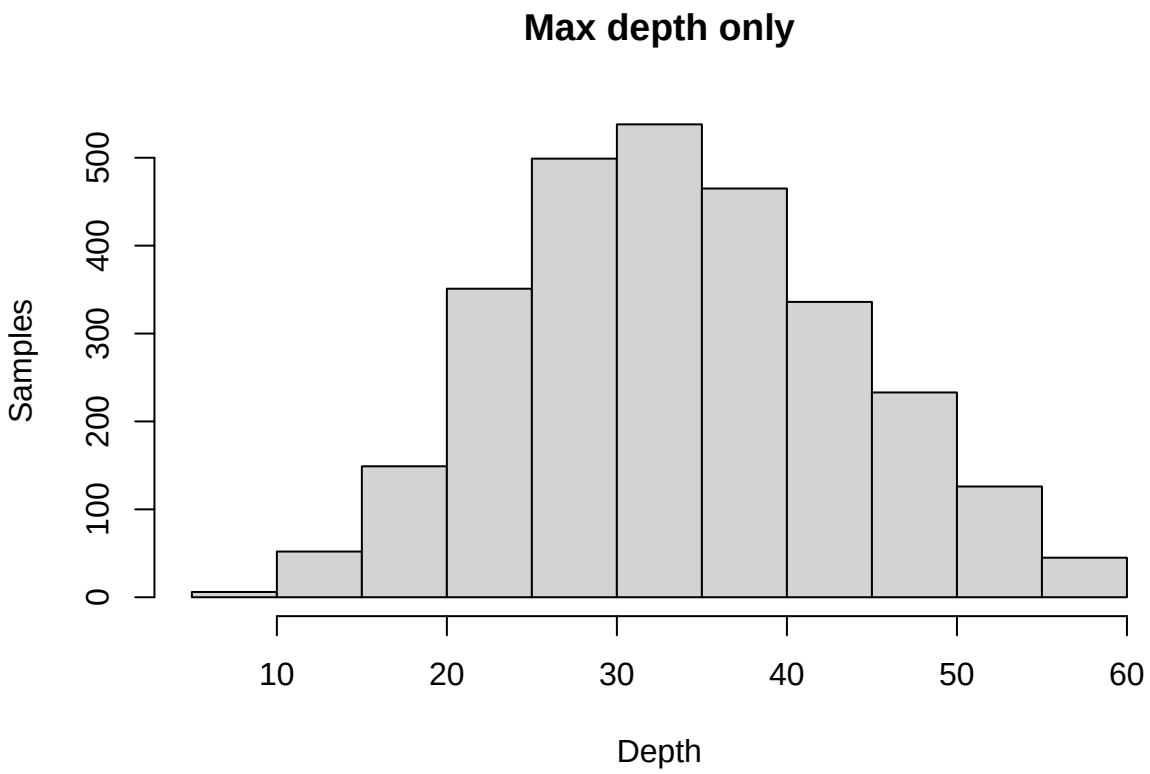
```
data_Genos[LOCUS %in% good_loci_dp_min]$DP %>%
  hist(., main='Min depth only', xlab='Depth', ylab='Samples')
```



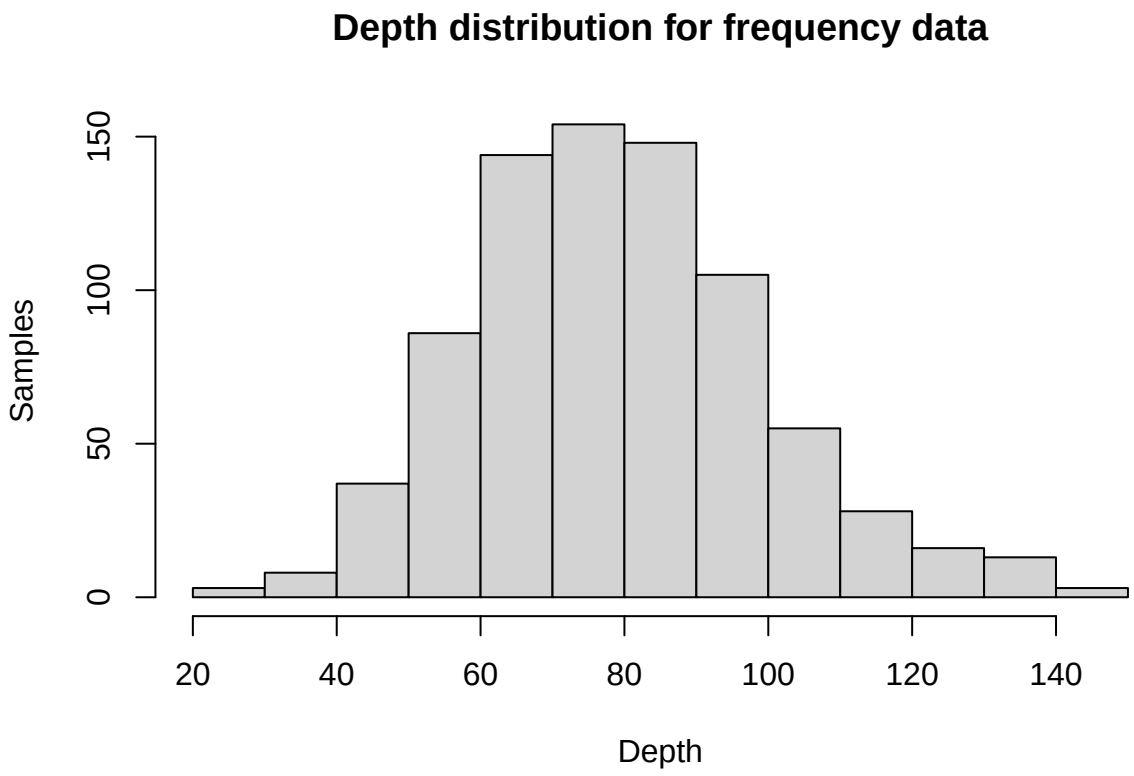

```
# Filtering maximum only
good_loci_dp_max <- filter_depth(
  data_Genos, minDP=NULL, maxDP=60,
  locusCol='LOCUS', dpCol='DP'
  )

length(good_loci_dp_max)
```

```
## [1] 28
```

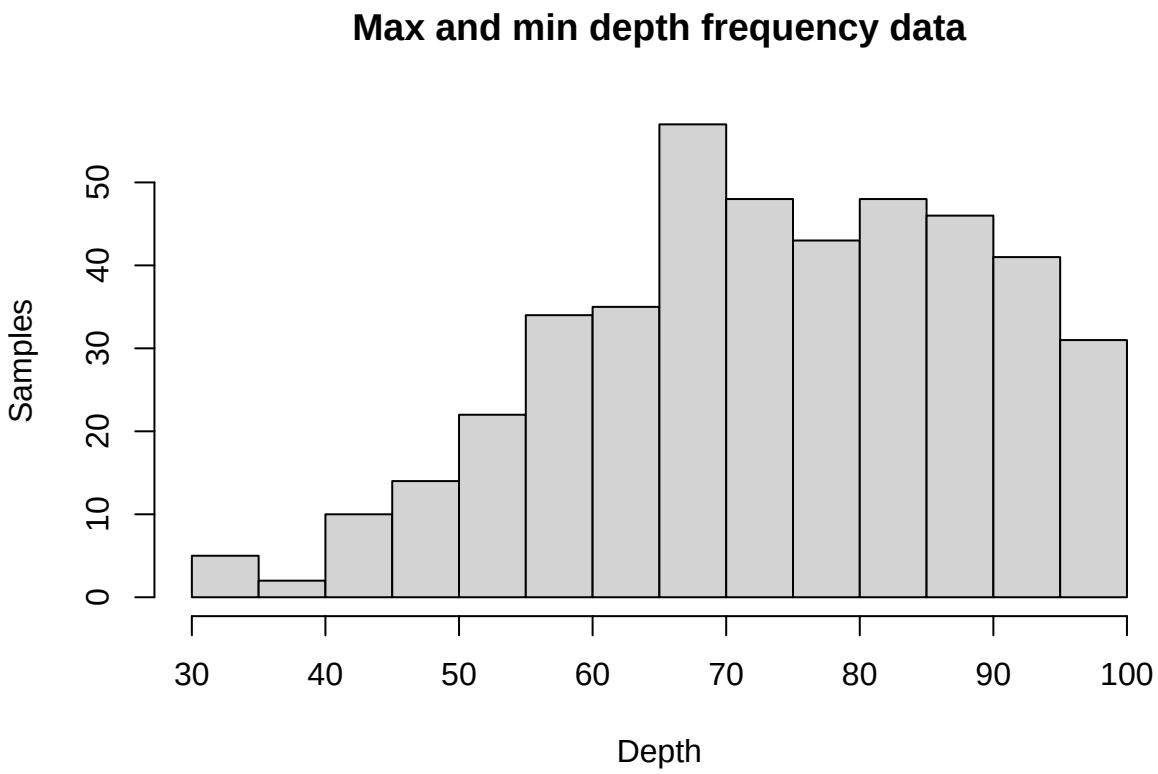
```
data_Genos[LOCUS %in% good_loci_dp_max]$DP %>%
  hist(., main='Max depth only', xlab='Depth', ylab='Samples')
```




```
# Filtering a data table of allele frequencies is the same.
data_PoolFreqs$DP %>%
  hist(., main='Depth distribution for frequency data', xlab='Depth', ylab='Samples')
```




```
good_loci_dp_freqs <- filter_depth(
  data_PoolFreqs, minDP=30, maxDP=100,
  locusCol='LOCUS', dpCol='DP'
  )

data_PoolFreqs[LOCUS %in% good_loci_dp_freqs]$DP %>%
  hist(., main='Max and min depth frequency data', xlab='Depth', ylab='Samples')
```



Finally, there is a function to "unlink" loci in reduced representation datasets (herein, "RADseq"). In these types of datasets, genotypes or allele frequencies are derived from short genomic fragments (herein, "RAD contigs"). To reduce the non-independence of loci within RAD contigs, it is standard practise to filter loci to keep just one locus per RAD contig. This function is called `filter_unlink`.

The key argument is `method`. When `method=="random"`, then the locus in the RAD contig is randomly selected. When `method=="first"`, then the first locus in the RAD contig is selected.

```
unlink_loci <- filter_unlink(
  data_Genos, chromCol='CHROM', locusCol='LOCUS', posCol='POS',
  method='random'
  )

length(unlink_loci)
```

```
## [1] 195
```

Postamble

You have now completed the 'Quality control and filtering' tutorial! You should now be familiar with the functions in *genomalicious* that help visualise patterns of missingness in your population genomic data. You should also now be able to use *genomalicious* to filter you data for missingness, minor allele frequency, read depth, and independent loci on different genomic fragments (e.g., RAD contigs in RADseq datasets). You now also know how *genomalicious* can help you replace missing genotypes with a "most common genotype" approach, although it is recommended that for real analyses you should use proper genotype imputation algorithms.