Introduction to the `argyle` package

Andrew P Morgan

Updated 17 May 2015

Contents

Introduction

High-throughput genotyping of tens of thousands of SNPs using microarrays is common practice in both laboratory and population genetics. Ggenotypes at a dense panel of biallelic markers with a low rate of missing data are a valuable resource for breeding, marker-assisted selection, genetic mapping and analyses of population structure. The Illumina Infinium system is one popular and cost-effective ($\sim$ 100/sample) platform. Custom Illumina arrays are available for many organisms of research, agricultural or ecological interest including mouse (CCC (2012)), dog, chicken, cow, pig, horse, sheep, salmon (http://www.biomedcentral.com/1471-2164/14/439) and cotton (http://g3journal.org/content/early/2015/04/22/g3.115.018416).

The `argyle` package provides basic functionality for loading biallelic SNP datasets directly from (1) the output of Illumina BeadStudio; or (2) `PLINK` binary filesets. For genotypes from Illumina arrays, hybridization intensity data is stored in parallel with genotype calls for use in quality control and downstream analyses. Several functions for quality control at the level of both hybridization intensity (where available) and genotype calls are provided. Data can be exported from `argyle` to `PLINK` binary format; or to formats compatible with the `R/qtl` package for genetic mapping in experimental crosses (Broman *et al.* (2003)), or the `DOQTL` package (Gatti *et al.* (2014)) for mapping in multi-founder outbred populations.

The design of `argyle` is inspired by the `PLINK` (Purcell *et al.* (2007)) software. A `PLINK` fileset has three parts: a genotype matrix, a marker map and a "pedigree" (sample and family metadata) file. Likewise the central data structure in this package (the `genotypes` object) stores a genotype matrix in parallel with a marker map and sample metadata.

> **Note**: `argyle` was designed for mouse genetics, but should be applicable to any diploid organism with an X-Y sex chromosome system (with males XY and females XX.)

Caveats and other software

We created `argyle` to fill a specific niche: quality control and basic exploratory analysis of genotype data obtained from the Mouse Universal Genotyping Array series (MUGA, MegaMUGA and GigaMUGA). These are custom Illumina Infinium arrays processed by Neogen Inc (Lincoln, NE). The design and content of the MUGA arrays are described elsewhere (Morgan & Welsh (2015)).

This package explicitly favors *simplicity* and *readability* of code over raw efficiency. It is appropriate for the "medium-sized" data – say, tens of thousands of markers and hundreds of individuals – regularly encountered in experimental genetics. Users with larger datasets routinely collected in human genetics – say, millions of markers and thousands of individuals – which do not fit comfortably in memory as plain-vanilla `R` objects probably want to explore more sophisticated `R` packages (such as the `GenABEL` suite).

A zoo of `R` packages already exists for normalization of raw intensity data from Illumina platforms: see `lumi` (Du *et al.* (2008)) and `beadarray` (Dunning *et al.* (2007)) among others. Users interested in exploiting the copy-number signal from Illumina arrays should consult these packages and related references. Gross copy-number aberrations affecting hundreds or thousands of markers, such as (partial) aneuploidy, *can* be identified in `argyle`; see **Intensity-based analyses** for more.

Although `argyle` implements some basic frequency calculations for use in genotyping quality control, it makes no effort to duplicate or replace the functionality of existing `R` packages for statistical and population genetics. More serious calculations – marker LD, Hardy-Weinbery equilibrium tests, association tests – can be accessed through the pacakge's `PLINK` wrappers. (These calls don't require holding the genotypes themselves in memory in the `R` session, so they can be applied to quite large datasets.)

The `genotypes` object

The central datastructure in the `argyle` package is the `genotypes` object. It is simply a matrix of genotypes (markers in rows, samples in columns) with a marker map and sample metadata stored as attributes. Row names (marker names) and column names are strictly required in order to keep the various pieces of the object unambiguously in sync. Because a `genotypes` is a matrix (ie. `is.matrix(x) == TRUE`), any function which accepts a matrix will also accept a `genotypes`. Standard attributes of the `genotypes` object are as follows.

- `attr(,"map")` – the marker map, in `PLINK` format. Required columns are
  - `chr` – chromosome identifier (anything containing "X" assumed to be chrX; anything with a "Y", chrY); `NA` for missing
  - `marker` – globally-unique marker identifier without whitespace
  - `cM` – genetic position in centimorgans; 0 for missing
  - `pos` – physical position in basepairs; 0 for missing
  - `A1` – allele 1, arbitrarily defined (but we will label it the REF allele)
  - `A2` – allele 2, arbitrarily defined (but we will label it ALT)
- `attr(,"ped")` – sample information, in `PLINK` format. Required columns are
  - `fid` – the "family" or group (or batch, or strain, . . .) label
  - `iid` – globally-unnique individual identifier without whitespace

    - `mom` – individual ID of this sample's mother; 0 for missing
    - `dad` – individual ID of this sample's mother; 0 for missing
    - `sex` – sex; 1 = male, 2 = female, 0 = missing
    - `pheno` – phenotype; 0 or -9 = missing, 1 = "control", 2 = "case"; or any floating-point value for a quantitative trait

- `attr(,"alleles")` – allele encoding (see **Allele encoding schemes** later in this vignette), one of `"native"`, `"01"`, `"relative"` or `"parent"`.
- `attr(,"filter.sites")` – logical vector of length equal to number of *rows* of genotype matrix, with `TRUE` values flagging suspicious markers
- `attr(,"filter.samples")` – logical vector of length equal to number of *columns* of genotype matrix, with `TRUE` values flagging suspicious samples

When hybridization intensity data are included, the object has several additional attributes.

- `attr(,"intensity")` – a named list of length 2, whose elements `$x` and `$y` are matrices of x and y hybridization intensities. These matrices have the same shape, and the same row and column names, as the main genotypes matrix.
- `attr(,"normalized")` – logical scalar indicating of any normalization has been applied to the intensity matrices

If the thresholded quantile-normalization procedure (tQN; see section **Particulars for Illumina arrays** later in this section) has been performed, the result will be stored in two additional attributes. * `attr(,"baf")` – matrix of B-allele frequency (BAF) values * `attr(,"lrr")` – matrix of log2-intensity ratio (LRR) values

Several `R` generics are implemented for working `genotypes` objects. To demonstrate them, we will use an example dataset containing genotypes from 116 mouse samples at 14319 markers across three chromosomes (chr17, chr18, chr19, chrY).

```
data(ex)
```

The `summary()` method returns a brief overview of the contents of a `genotypes`: count of samples and markers; how alleles are encoded; whether underlying hybridization-intensity is also included; and any quality filters which have been set.

```
summary(ex)
```

```
## --- ex ---
## A genotypes object with 14319 sites x 116 samples
## Allele encoding: native
## Intensity data: yes (raw)
## Sample metadata: yes ( 67 male / 49 female / 0 unknown )
## Filters set: 0 sites / 0 samples
```

The `print()` method returns the same information as `summary()`, plus a count of markers by chromosome. It is included only to prevent accidentally flooding an interactive terminal with the contents of a big object.

```
print(ex) # or just type 'ex'
```

```
## --- gty ---
## A genotypes object with 14319 sites x 116 samples
## Allele encoding: native
## Intensity data: yes (raw)
## Sample metadata: yes ( 67 male / 49 female / 0 unknown )
## Filters set: 0 sites / 0 samples
##
## Counts of markers by chromosome:
## chr17 chr18 chr19  chrY
##  5548  4925  3763    83
```

Use `head()` to peek at the first k markers in a `genotypes`, and to see underlying marker and sample

```
head(ex, n = 6, nsamples = 6)
```

```
## Genotypes matrix:
##                  AA037 AA037 AA041 AA041 AA376 AB009
##     UNCHS043509     T     T     T     T     T     T
##     UNCHS043511     T     T     T     T     T     T
##    UNCJPD006513     A     A     A     A     A     A
##     UNCHS043510     G     G     G     G     G     G
##     UNCHS043512     G     G     G     G     G     G
##     JAX00429559     T     T     T     T     T     T
##
## Marker map:
##    chr       marker          cM     pos A1 A2
##  chr17  UNCHS043509 0.006630976 3075928  T  C
##  chr17  UNCHS043511 0.006630976 3075928  T  C
##  chr17 UNCJPD006513 0.007102397 3081326  A  G
##  chr17  UNCHS043510 0.007365267 3084336  T  G
##  chr17  UNCHS043512 0.007365267 3084336  T  G
##  chr17  JAX00429559 0.007628051 3087345  T  C
##
## Sample info:
##  fid                           iid mom dad sex pheno
##   AA   AA_0374_F_10004312002_R01C01   0   0   2    -9
##   AA   AA_0374_F_10004312002_R11C01   0   0   2    -9
##   AA   AA_0417_M_10004312002_R02C01   0   0   1    -9
##   AA   AA_0417_M_10004312002_R03C01   0   0   1    -9
##   AA  AA_37621_M_10004312002_R04C01   0   0   1    -9
##   AB   AB_0095_F_10004312006_R04C02   0   0   2    -9
```

Several accessor methods provide convenient shortcuts to obtain marker map, sample metadata, etc. without need for the awkward `attr(x, "attribute")` syntax. These methods offer *read-only* access to object attributes.

> **Technical note**: Accessor functions are read-only for two reasons. The first is to raise the barrier to accidental overwriting of data. The second is `attr<-` is an `R` primitive, so that setting attributes can be performed without making a new copy of an object. However, user-defined functions for setting object attributes would have to pass the object by value rather than by reference, making at least one new copy in the process. This performance hit becomes significant when handling objects containing hundreds of samples genotyped at tens of thousands of markers.

To pull out the marker map, use

```
## see the marker map
map <- markers(ex)
head(map)
```

```
##                 chr       marker          cM     pos A1 A2           type
## UNCHS043509  chr17  UNCHS043509 0.006630976 3075928  T  C recomb_hotspot
## UNCHS043511  chr17  UNCHS043511 0.006630976 3075928  T  C recomb_hotspot
## UNCJPD006513 chr17 UNCJPD006513 0.007102397 3081326  A  G     wild_novel
## UNCHS043510  chr17  UNCHS043510 0.007365267 3084336  T  G recomb_hotspot
## UNCHS043512  chr17  UNCHS043512 0.007365267 3084336  T  G recomb_hotspot
## JAX00429559  chr17  JAX00429559 0.007628051 3087345  T  C      MDA_other
##              is.MM unique is.biallelic
## UNCHS043509  FALSE   TRUE         TRUE
## UNCHS043511  FALSE   TRUE         TRUE
## UNCJPD006513 FALSE   TRUE         TRUE
## UNCHS043510  FALSE   TRUE         TRUE
## UNCHS043512  FALSE   TRUE         TRUE
## JAX00429559  FALSE   TRUE         TRUE
```

The result is a dataframe whose rows are parallel to those in the genotypes matrix. The map must have rownames which match the rownames of the parent object. Columns 1 through 6 are the required columns for a `PLINK` marker file (`*.bim`); remaining columns can store any extra marker metadata of interest.

To pull out sample information, use

```
## see sample metadata
mice <- samples(ex)
head(mice)
```

```
##                                  fid                           iid mom dad
## AA_0374_F_10004312002_R01C01   AA  AA_0374_F_10004312002_R01C01   0   0
## AA_0374_F_10004312002_R11C01   AA  AA_0374_F_10004312002_R11C01   0   0
## AA_0417_M_10004312002_R02C01   AA  AA_0417_M_10004312002_R02C01   0   0
## AA_0417_M_10004312002_R03C01   AA  AA_0417_M_10004312002_R03C01   0   0
## AA_37621_M_10004312002_R04C01  AA AA_37621_M_10004312002_R04C01   0   0
## AB_0095_F_10004312006_R04C02   AB  AB_0095_F_10004312006_R04C02   0   0
##                                sex pheno
## AA_0374_F_10004312002_R01C01    2    -9
## AA_0374_F_10004312002_R11C01    2    -9
## AA_0417_M_10004312002_R02C01    1    -9
## AA_0417_M_10004312002_R03C01    1    -9
## AA_37621_M_10004312002_R04C01   1    -9
## AB_0095_F_10004312006_R04C02    2    -9
```

The result is a dataframe whose rows are parallel to the *columns* of the genotype matrix, with matching names. The first 6 columns are the required columns for a `PLINK` "family" file (`*.fam`, `*.tfam`): group ID, individual ID, mother ID, father ID, sex (1 = male, 2 = female), and phenotype (0/-9 = missing, 1 = control, 2 = case; or any floating-point number for a quantitative trait). For downstream compatibility with `PLINK`, missing values are encoded as 0 rather than `NA`. Note that the rownames of the sample metadata must match the column `iid` (unique individual ID), and the individual IDs cannot contain whitespace characters.

Subsets of a `genotypes` object can be obtained in two ways. First, the generic `[` indexing operator has been implemented for `genotypes` to allow for taking "slices" of the genotypes matrix and to have that slicing propagated to all the object's attributes. As usual, the index argument(s) to `[` can be logical vectors (`TRUE` to include a row or column in the result), character vectors (row or column names to include), or integer vectors (row or column indices to include). For example, we can extract the first 1000 markers in the example dataset and call `summary()` to check the result.

```
first1k <- ex[ 1:1000, ]
summary(first1k)
```

```
## --- first1k ---
## A genotypes object with 1000 sites x 116 samples
## Allele encoding: native
## Intensity data: yes (raw)
## Sample metadata: yes ( 67 male / 49 female / 0 unknown )
## Filters set: 0 sites / 0 samples
```

We could also subset by samples (columns); here we pick a random 10 columns.

```
summary( ex[ ,sample.int(ncol(ex), 10) ] )
```

```
## --- ex[, sample.int(ncol(ex), 10)] ---
## A genotypes object with 14319 sites x 10 samples
## Allele encoding: native
## Intensity data: yes (raw)
## Sample metadata: yes ( 5 male / 5 female / 0 unknown )
## Filters set: 0 sites / 0 samples
```

A common task is to extract markers or samples by groups defined in the marker map (eg. all markers in some genomic region) or sample metadata (eg. only female samples). The generic `subset()` has been implemented for `genotypes` to make the syntax for such operations simpler. Similar to the `subset(x, ...)` method for dataframes, the indexing expression in `...` is evaluated in the scope of the marker map (the default, `by = "markers"`) or sample metadata (`by = "samples"`). For example, the following pairs of code fragments produce the same result.

```
## get only markers on chrY
x <- subset(ex, chr == "chrY")
y <- ex[ (markers(ex)$chr == "chrY"), ]
identical(x,y)
```

```
## [1] TRUE
```

```
## get only the female samples
x <- subset(ex, sex == 2, by = "samples")
y <- ex[ ,(samples(ex)$sex == 2) ]
identical(x,y)
```

```
## [1] TRUE
```

Technical note: If using the `subset()` syntax, unexpected results can occur if a variable defined in the current environment shares a name with a column in the relevant dataframe. This is due to underlying use of `R`'s "non-standard evaluation" in `subset()`. Just be careful.

> **Technical note:** A side effect of overloading the `[` operator for `genotypes` is that any operation which repeatedly slices a matrix (including functions in the `apply` family) will incur the extra cost of slicing the marker map, sample metadata, intensity matrices, etc. on every pass. If this is a problem, use the unexported `argyle:::.copy.matrix.noattr()` function to make a copy of the genotypes matrix which preseves row and column names but dumps all the other attributes.

The `argyle` package borrows its notion of a "filter" from the VCF format: null (in our case `FALSE`) unless there is evidence for low quality. Unlike VCF, which allows filters for sites (in our case markers) only, `argyle` defines filters for both sites and samples. A marker (site) or sample can be marked as suspicious by setting its entry to `TRUE` in `attr(,"filter.sites")` or `attr(,"filter.samples")` respectively. Let us check how many filters are set in the example dataset.

```
## see quality filters (like VCF's 'FILTER' field)
fl <- filters(ex)
sapply(fl, sum)
```

```
##   sites samples
##       0       0
```

Set filters for the first 10 sites and check them.

```
attr(ex, "filter.sites")[1:10] <- TRUE
sapply(filters(ex), sum)
```

```
##   sites samples
##      10       0
```

To drop all filtered sites and/or samples, use `apply.filters()`. The default behavior is to apply filters to both sites and samples (`apply.to = "both"`).

```
fl <- apply.filters(ex, "both")
```

```
## Dropping 10 markers and 0 samples...
```

```
summary(fl)
```

```
## --- fl ---
## A genotypes object with 14309 sites x 116 samples
## Allele encoding: native
## Intensity data: yes (raw)
## Sample metadata: yes ( 67 male / 49 female / 0 unknown )
## Filters set: 0 sites / 0 samples
```

After application of filters, all filters in the resulting `genotypes` are (trivially) set to `FALSE`.

Allele encoding

The `argyle` package supports both character and numeric representation of genotypes. Character genotypes (the `"native"` encoding) are expected to take the values `ACGTHN`, where `H` represents a heterozygous call and `N` a no-call (ie. missing data). Since all markers are expected to be biallelic SNPs, character genotypes can be converted to a computationally more convenient numeric encoding in one of two ways.

- **By reference alleles** ("01"). Genotypes are encoded 0 if they match column A1 in the marker map, 1 if heterozygous (H), 2 if they match column A2, and missing (NA) otherwise. This encoding is most useful for analyses of experimental crosses.
- **By frequency** ("relative"). Genotypes are encoded 0 if they match the majority homozygous call (the major allele) at this marker within *this* genotypes object, 1 if heterozygous, 2 if they match the other homozygous call (the minor allele), and NA if missing. This encoding is most useful for population-based analyses.

All genetic analyses in argyle require that genotypes be first converted to a numeric encoding. Since both allele encoding and reference alleles (A1 and A2) are both required fields in a genotypes object, inter-conversion between the "01" and "native" encodings entails no loss of information. Conversion from "relative" to "native" is not supported, mostly to avoid the ambiguity that results when minor alleles are defined against a genotype matrix which is then subsetted (possibly destroying the definition of the "minor allele.")

To convert between encodings, use recode():

```
## convert from character ('native') to numeric ('01')
ex.recode <- recode(ex, "01")
```

```
## Recoding to 0/1/2 using reference alleles.
```

```
summary(ex.recode)
```

```
## --- ex.recode ---
## A genotypes object with 14319 sites x 116 samples
## Allele encoding: 01
## Intensity data: yes (raw)
## Sample metadata: yes ( 67 male / 49 female / 0 unknown )
## Filters set: 10 sites / 0 samples
```

And now convert again, back to the original encoding, and prove that no information was lost.

```
## convert back
ex.rerecode <- recode(ex.recode, "native")
```

```
## Recoding to character using reference alleles.
```

```
identical(ex, ex.rerecode)
```

```
## [1] TRUE
```

Particulars for Illumina arrays

The technical aspects of the Illumina Infinium genotyping array platform are described at length in Steemers *et al.* (2006). Briefly, oligonucleotide probes are designed to target k invariant base pairs adjacent to a biallelic SNP. These oligos are conjugated to silica beads which are addressed into a microarray. Sample DNA is hyridized to the array, and a single-base extension reaction is performed at the target SNP using fluorescently-labelled nucleotides. Two-channel fluorescence intensity signals are captured by a scanner.

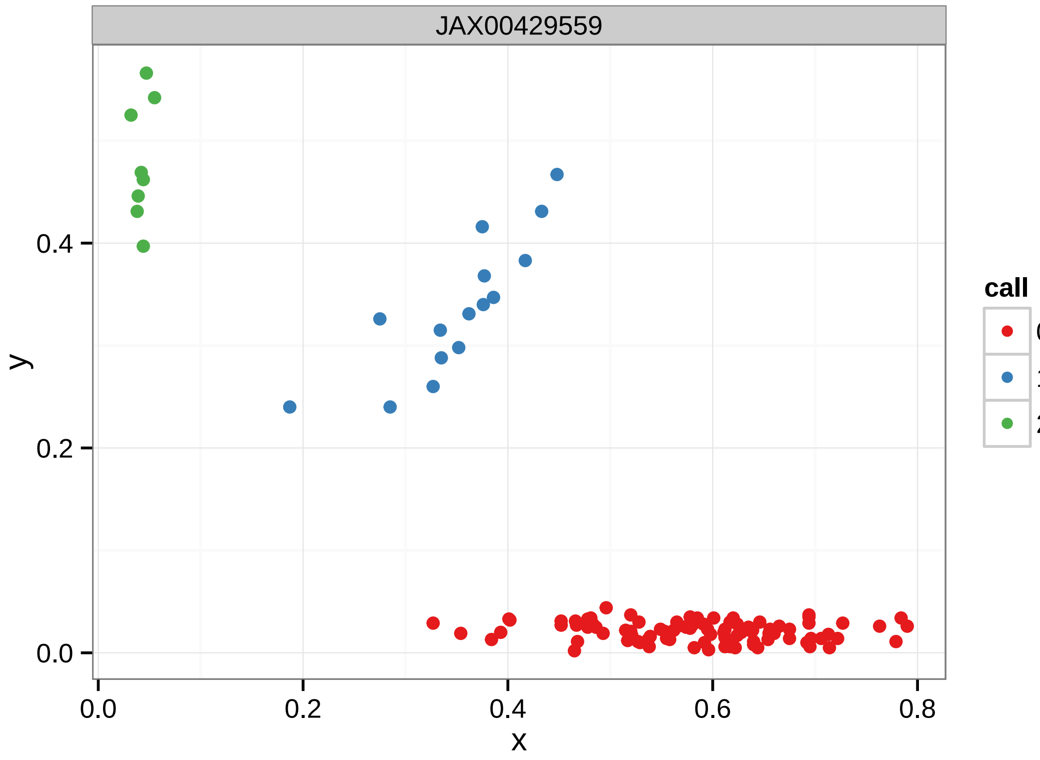
Raw fluorescence signals are post-processed and normalized prior to genotype-calling. Many normalization algorithms have been proposed that take advantage of specific properties of the Illumina platform to estimate and remove technical artefacts; see Ritchie *et al.* (2009) for discussion. Illumina's own BeadStudio software use a procedure described in Peiffer *et al.* (2006) that attempts to place samples into three clusters: two (presumably) homozygous clusters, one near the x- and another near the y-axes, and a (presumably) heterozygous cluster along an arc between them.

Hybridization-intensity signals have carry information about both genotype and copy number. Discrete genotypes for a group of samples can be inferred by first clustering samples in the x, y plane at each marker; then identifying the three canonical clusters (corresponding to genotypes AA, AB and BB at the target SNP); and finally assigning a most probably cluster membership to each sample. The total hybridization intensity along both x and y dimensions – and, at heterozygous markers, the relative magnitude of x and y signals – is informative for copy number. "Total intensity" is defined by `argyle` not as $x + y$ but as $d = \sqrt{x^2 + y^2}$, the total distance from the origin. This is ancedotally superior to the simple sum if intensities are saturated along either dimension.

Note The few intensity-based analyses in `argyle` have BeadStudio output in mind. They are appropriate for identifying failed or contaminated samples, but not much more. The package was designed to complement a custom array for mouse (the Mouse Universal Genotyping Array (MUGA) series) supplied by Neogen Inc. That vendor supplies BeadStudio output to its customers. Users with deeper interest in normalization and copy-number estimation should consult the Bioconductor "Microarray analysis" task view for more information.

For well-performing markers, genotype is obvious in the x, y coordinate system, as demonstrated by the plot below (which we will call a "cluster plot"). The function `plot.clusters()` takes a `genotypes` and a list of markers (or any valid row-indexing vector for the genotypes matrix) and generates a plot with one panel per marker.

```r
plot.clusters(ex, "JAX00429559")
```



We can check that the example dataset has intensity matrices attached as follows.

```
has.intensity(ex)
```

```
## [1] TRUE
```

To extract the full hybridization intensity matrices from a `genotypes`, use `intensity()`. The result is a named list of length 2 with elements `$x` and `$y`. Here we just check the dimensions of the result, to see that they match each other and those of the parent genotypes matrix.

```
## get intensity matrices (x and y)
intens <- intensity(ex)
lapply(intens, dim)
```

```
## $x
## [1] 14319   116
##
## $y
## [1] 14319   116
```

Use of intensity data for genotyping quality control is discussed in **Quality control** below.

Data import

Genotypes can be bundled into to a `genotypes` object three ways: manually from an `R` matrix using the `genotypes()` constructor; from Illumina BeadStudio output; or from a `PLINK` binary fileset.

First we demonstrate making a `genotypes` from scratch. The `genotypes()` constructor requires a genotypes matrix with appropriate row (marker) and column (sample) names; a properly-formatted marker map; and an *explicitly-specified allele encoding*. (This is important.) If the dataframe of sample metadata is absent, a mostly-uninformative one will be automatically generated from the column names of the genotypes matrix. Alternatively, we use the `make.fam()` function ourselves to generate an appropriately-formatted dataframe of sample metadata from a vector of sample IDs. (See `?make.fam` for details.)

```
sm <- c("id1","id2","id3")
mk <- c("snp1","snp2","snp3","snp4")
map <- data.frame(chr = "chr1", marker = mk,
                  cM = c(0.5, 1.0, 1.5, 2.0),
                  pos = c(0.5, 1.0, 1.5, 2.0)*1e6,
                  A1 = c("A","G","A","C"),
                  A2 = c("C","T","G","G"))
rownames(map) <- mk

fam <- make.fam(sm, fid = "testers", sex = c("f","f","m"))

G <- matrix( c("A","C","A",
               "T","G","T",
               "A","G","A",
               "C","G","C"),
             byrow = TRUE, nrow = 4, ncol = 3,
             dimnames = list(mk, sm) )

geno <- genotypes(G, map = map, ped = fam, alleles = "native")
```

To confirm that the object was created as expected, check the summary:

```r
summary(geno)
```

```
## --- geno ---
## A genotypes object with 4 sites x 3 samples
## Allele encoding: native
## Intensity data: no
## Sample metadata: yes ( 1 male / 2 female / 0 unknown )
## Filters set: 0 sites / 0 samples
```

And check the contents:

```r
head(geno)
```

```
## Genotypes matrix:
##                     id1   id2   id3
##              snp1     A     C     A
##              snp2     T     G     T
##              snp3     A     G     A
##              snp4     C     G     C
##
## Marker map:
##    chr marker  cM     pos A1 A2
##   chr1   snp1 0.5  500000  A  C
##   chr1   snp2 1.0 1000000  G  T
##   chr1   snp3 1.5 1500000  A  G
##   chr1   snp4 2.0 2000000  C  G
##
## Sample info:
##       fid iid mom dad sex pheno
##   testers id1   0   0   2    -9
##   testers id2   0   0   2    -9
##   testers id3   0   0   1    -9
```

From Illumina BeadStudio

Import of genotypes and hybridization intensities from BeadStudio reports is achieved with the `read.beadstudio()` function. Although the format of the human-readable output from BeadStudio can be customized by the user, `argyle` assumes that it has the format supplied to customers of Neogen Inc. That format involves two files:

- `Sample_Map.zip` – a sample manifest, with column headers (at least) `Name` and `Gender`.
- `{prefix_}FinalReport.zip` – the genotyping results. This file has a 9-line header section followed by a line of column headings (expected to have the order below), and then the data itself. The first 6 columns should be:
  - `SNP Name` – marker identifier, globally-unique
  - `Sample ID` – sample identifier matching contents of column `Name` in `Sample_Map`
  - `X` – transformed *x*-intensity
  - `Y` – transformed *y*-intensity

  - `Allele1 - Forward` – genotype call for allele 1 (one of `ACTG-`)
  - `Allele2 - Forward` – genotype call for allele 2 (one of `ACTG-`)

The `-` character is assumed to represent a missing genotype, and rows with `-` in either `Allele1` or `Allele2` will be marked as missing.

The `data.table` package, which provides memory-efficient re-implementation of the base-`R` dataframe, is used to read from these files. It can comfortably handle several million rows at reasonable speed on a modern laptop.

Decompressing the files is not necesssary if command-line `zip` is available – `argyle` will decompress on-the-fly. For platforms without command-line `zip` (eg. Windows), `FinalReport.zip` must be unzipped but `Sample_Map` should not be. (This is due to a quirk of `data.table`.)

In addition to the files from BeadStudio, a dataframe containing a valid marker map (as discussed in the **Introduction**) is *required* to perform the import. Pre-computed ones for the MUGA series of arrays are available from [**URL**]. Users of other arrays will have to prepare their own.

The running time of `read.beadstudio()` for a realistic dataset (say $80,000$ markers and 96 samples) is about ~ 1 minute on my 2014 MacBook Air. That's inconveniently long for this vignette but quite reasonable in practice. The code below demonstrates the command, but is not actually run. The parmater `prefix` corresponds to the `*` in the filename of `*FinalReport.zip`.

```
data(snps) # the marker map
geno <- read.beadstudio(prefix = "", snps, in.path = "path/to/folder")
```

From `PLINK` a fileset

Reading a `PLINK` binary fileset with `argyle` is simple with `read.plink()`. As an example, we can load genotypes of 28 wild mice from the Mouse Diversity Array (Yang *et al.* (2011)), provided in this package's "data/" directory. Genotypes from `PLINK` filesets are always read directly to the `"01"` encoding.

```
infile <- system.file("data/wild.chr19.bed", package = "argyle")
wild <- read.plink(infile)
```

```
## Reading family info from: </Users/apm/Dropbox/pmdvlab/argyle/data/wild.chr19.fam>
## Reading marker info from: </Users/apm/Dropbox/pmdvlab/argyle/data/wild.chr19.bim>
## Reading binary genotypes from: </Users/apm/Dropbox/pmdvlab/argyle/data/wild.chr19.bed>
```

```
print(wild)
```

```
## --- gty ---
## A genotypes object with 14306 sites x 28 samples
## Allele encoding: 01
## Intensity data: no
## Sample metadata: yes ( 13 male / 15 female / 0 unknown )
## Filters set: 0 sites / 0 samples
##
## Counts of markers by chromosome:
## chr19
## 14306
```

Check the contents to see that import worked:

```r
head(wild, n = 6, nsamples = 6)
```

```
## Genotypes matrix:
##                 IN13  IN17  IN25  IN34  IN38  IN40
##     JAX00468897    2     0     2     2     2     2
##     JAX00086300    2     1     2     2     2     2
##     JAX00086302    1     1    NA    NA     1     1
##     JAX00468901    2    NA     2     2     2     2
##     JAX00468902    1     2    NA    NA     1     1
##     JAX00086303    2    NA     2     2     2     2
##
## Marker map:
##    chr      marker    cM     pos A1 A2
##  chr19 JAX00468897 0.105 3125547  G  A
##  chr19 JAX00086300 0.121 3145514  T  C
##  chr19 JAX00086302 0.133 3159638  G  T
##  chr19 JAX00468901 0.133 3159902  G  A
##  chr19 JAX00468902 0.134 3160138  C  A
##  chr19 JAX00086303 0.134 3160277  A  G
##
## Sample info:
##  fid  iid mom dad sex pheno
##  cas IN13   0   0   1    NA
##  cas IN17   0   0   1    NA
##  cas IN25   0   0   1    NA
##  cas IN34   0   0   1    NA
##  cas IN38   0   0   2    NA
##  cas IN40   0   0   2    NA
```

From your own database

Users who routinely genotype hundreds or thousands of samples will probably want to store genotypes an a relational database. No (direct) database interface has been implemented in `argyle` yet. However, once a matrix of genotypes is available, a `genotypes` can be constructed using the "from scracth" method outlined at the top of this section.

Quality control

Careful quality control and sanity checking of microarray genotypes is an essential prerequisite to further analysis. QC can (and should) be performed both marker-wise and sample-wise. When hybridization-intensity data is available, QC should be performed on both intensities – they are the primary data – and the genotype calls. The `argyle` package provides a suite of functions implementing best practices developed in our group for QC of genotypes from the MUGA arrays.

For sample-wise QC, we recommend checking (at least) the following three metrics.

- **Call rate.** Arrays which fail due to poor input DNA quality, mixed or contaminated samples, or technical problems at the hybridization step have an excess of heterozygous (H) and missing (N) calls relative to expectations. Of course those expectations are key: highly inbred samples should have relatively few H calls, while samples from outbred or natural populations should have many Hs.

- **Intensity distribution.** Failed arrays have lower mean intensity and greater variance in intensity across probes than successful arrays. Comparing the distribution of total intensity ($d = \sqrt{x^2 + y^2}$; see **Introduction**) by sample within a batch, and between batches when multiple batches are available, can reveal failed samples.
- **Sex-chromosome concordance.** Nominally male samples with an excess of heterozygous calls on the X-chromosome, or female samples with nonmissing calls on the Y-chromosome, represent potential failures or sample swaps.

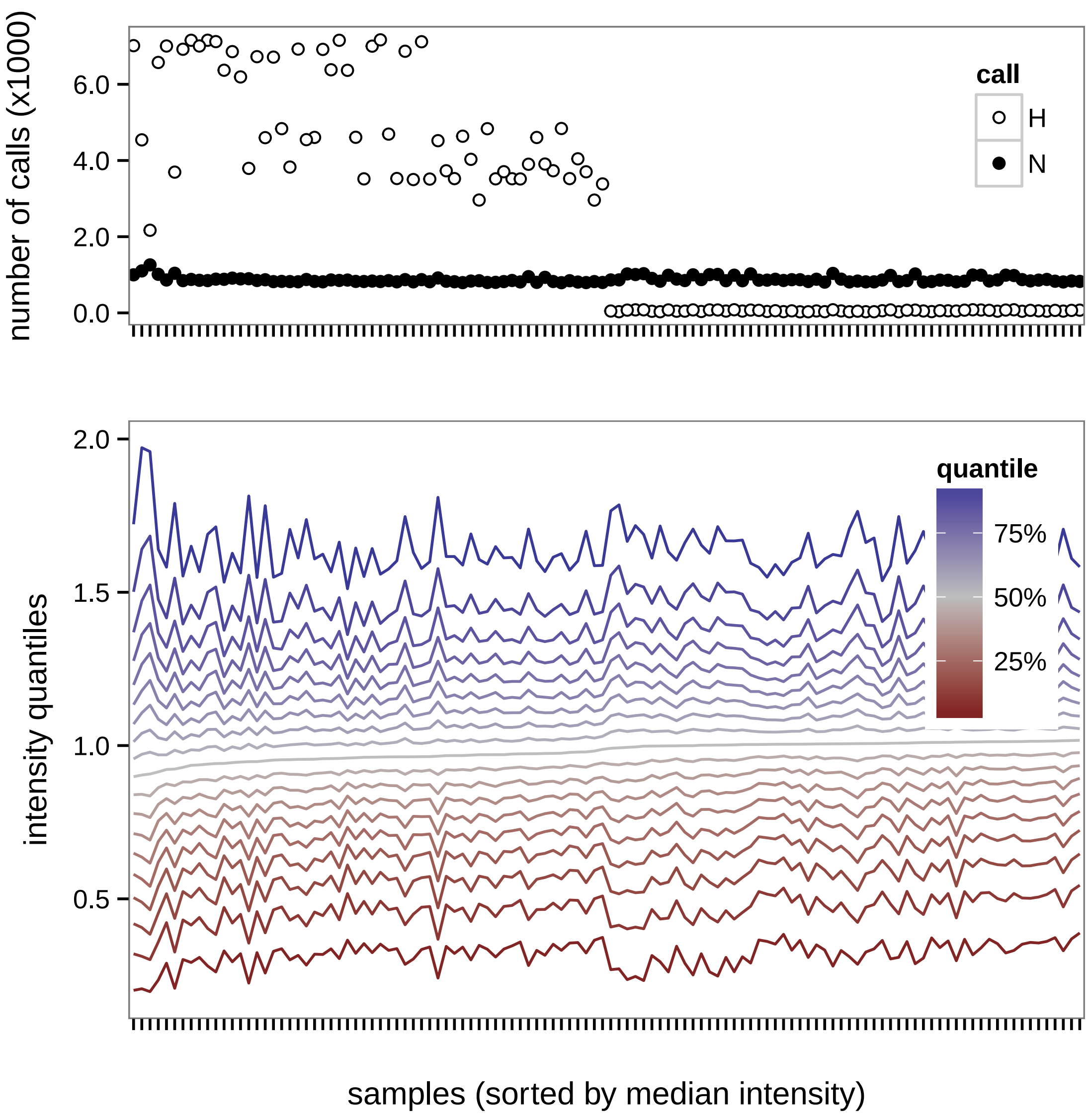
We will demonstrate these procedures on the example dataset in `ex`, which includes intensity data. To run sample-wise QC checks, use `run.qc.checks()`, which returns a copy of the input with a new attribute `attr(,"qc")` containing the results.

```
data(ex)
ex <- run.qc.checks(ex)
```

```
## Performing QC checks on genotype calls...
## Recoding to 0/1/2 using reference alleles.
## Performing QC checks on hybridization intensities...
## 0 markers and 0 samples now flagged as low-quality.
```

Use `qcplot()` to generate the default QC plots. (If `run.qc.checks()` had not been run already, it would be automatically called here.)

```
qcplot(ex)
```



The upper panel shows the count of H and N calls by sample. The lower plot shows intensity percentiles (contours) across all samples. Samples are sorted by their median intensity in both panels. In this example dataset, there are no outlying intensity profiles, but the samples in the left half of the plot have many more H calls than those in the right half despite having similar number of no-calls. This is expected: about half of the samples in the example dataset are inbred mouse strains, and the other half are F1s between them.

We can also check for concordance between the nominal sex of the samples and their sex as predicted by genotype. The current version of `argyle` uses only a crude threshold for number of non-missing Y-chromosome calls to predict sex. The defaults are calibrated for the GigaMUGA array (from which the example dataset is taken). More sophisticated sex-chromosome predictions based on hybridization intensity will be implemented in the future.

```
sexing <- predict.sex(ex)
```

```
## Predicting sex using count of good calls on chrY...
## Recoding to 0/1/2 using reference alleles.
```

```r
xtabs(~ predicted + nominal, data = sexing)
```

```
##          nominal
## predicted  1  2
##         1 67  0
##         2  0 49
```

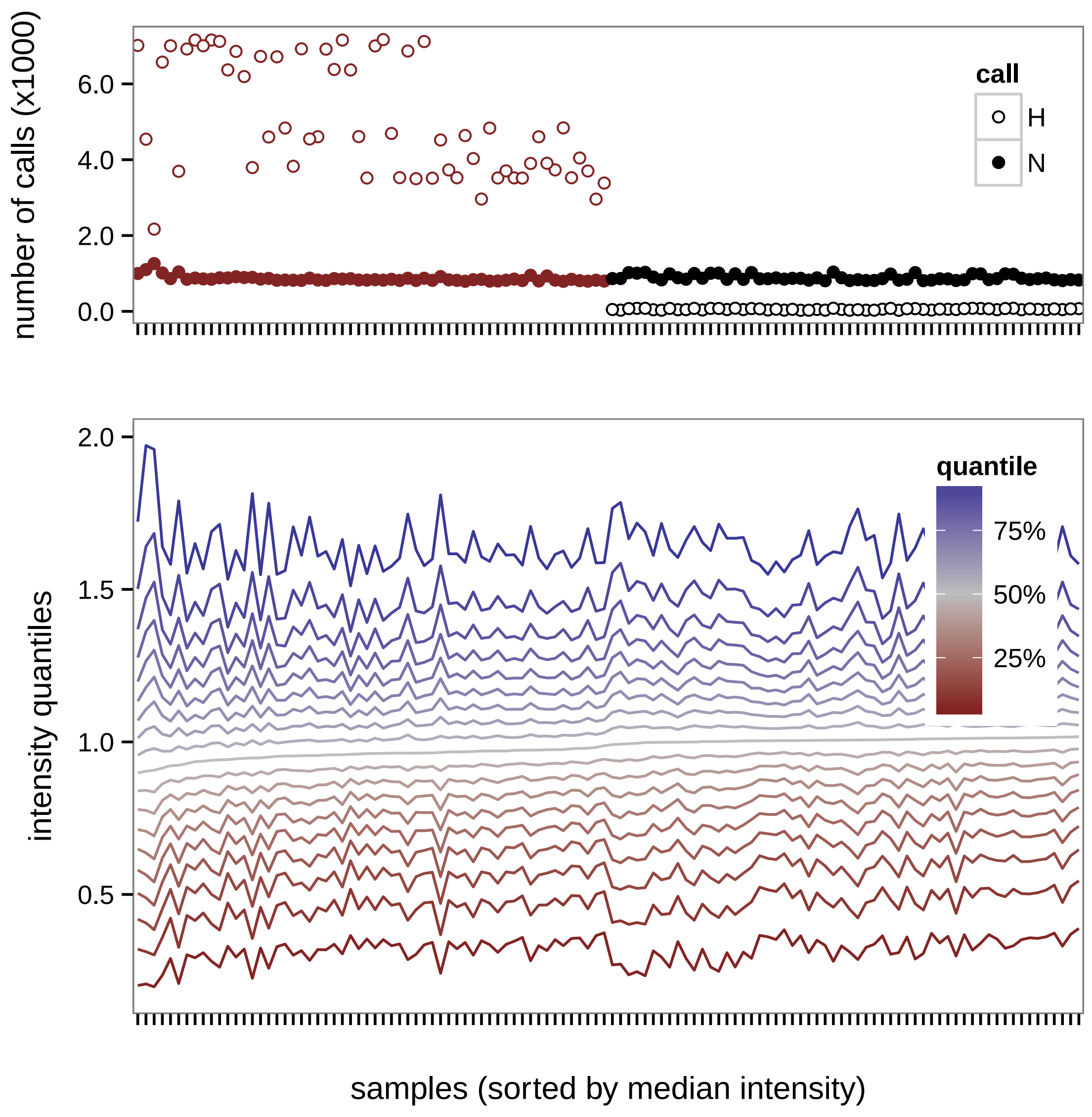
There is perfect concordance between the nominal and inferred sex of these samples.

Suppose we expected all samples to be mostly homozygous: we could supply the argument `max.H` to `run.qc.checks()` to flag all samples with greater than that threshold number of heterozygous calls. Flagged samples are now shown in dark red in the upper panel of the QC plot.

```r
ex <- run.qc.checks(ex, max.H = 2000)
```

```
## Performing QC checks on genotype calls...
## Recoding to 0/1/2 using reference alleles.
## Performing QC checks on hybridization intensities...
## 0 markers and 58 samples now flagged as low-quality.
```

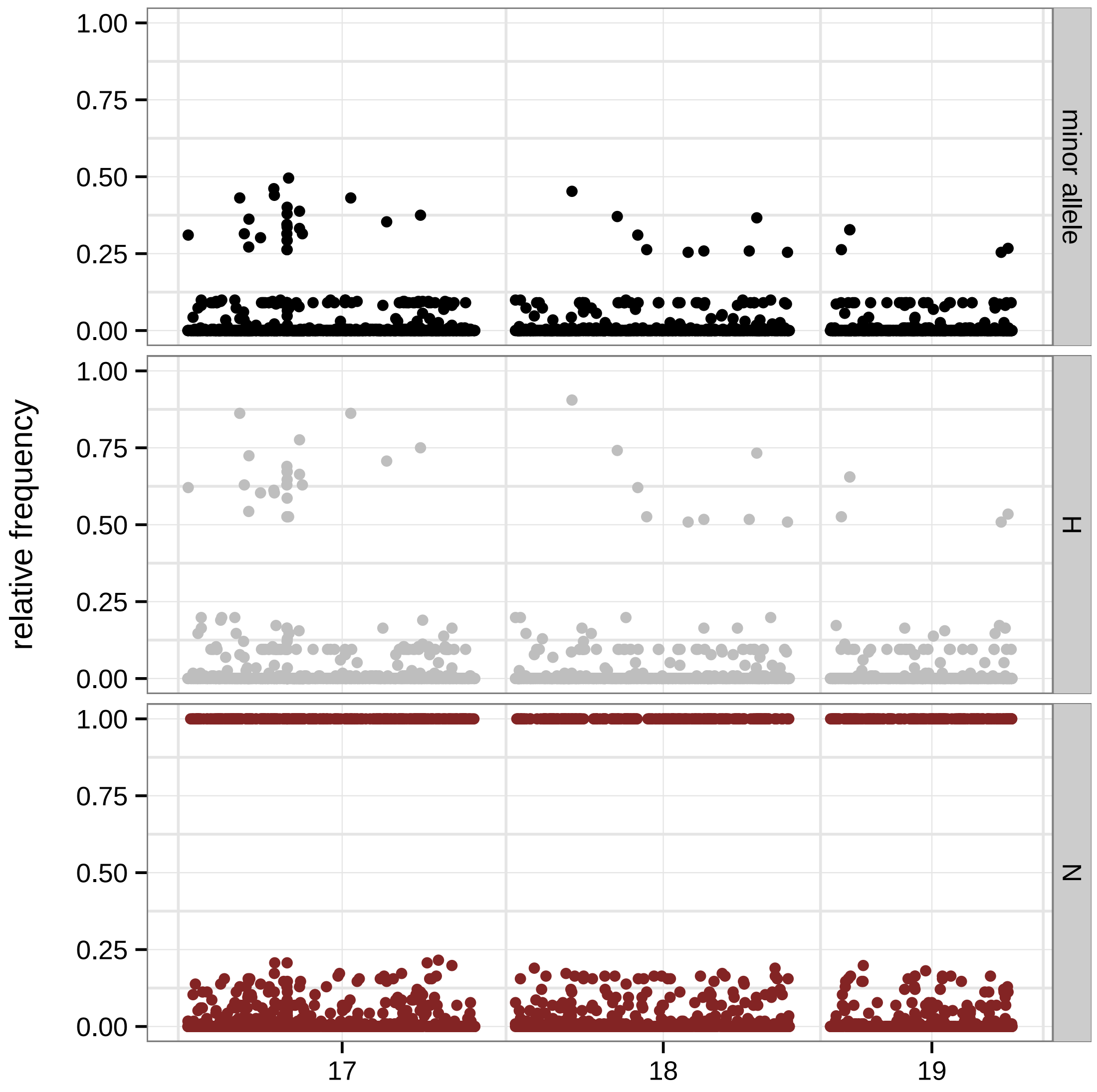
```r
qcplot(ex)
```



After removing failed samples we can identify low-performing markers. A low-performing marker is one with (1) high rate of missingness; (2) low or zero minor-allele frequency in the population of interest; or (3) higher-than-expected heterozygosity given its minor-allele frequencies. (The Illumina genotype-caller is prone to misclassifying no-calls as heterozygous calls.) The `freqplot()` function generates a graphical summary of call frequencies at markers falling below defined thresholds. A text summary is written to the console.

```
## plot low-performing markers
freqplot(subset(ex, chr != "chrY"), max.N = 0.2, max.H = 0.5, min.maf = 0.1)
```

```
## Recoding to 0/1/2 using empirical frequencies.
## Markers failing by
##  no-call rate:    768
##      het rate:     37
##           MAF:   2724
##          Total   2762
## Scale for 'colour' is already present. Adding another scale for 'colour', which will replace the exi
```



Note that we have excluded Y-linked markers from this analysis: the Y chromosome requires bespoke QC beyond the scopeof this vignette.

> **Note**: Criteria for excluding samples or markers from an analysis depend completely on the samples and the experiment at hand.

Intensity-based analyses

Hybridization intensity to Illumina arrays carries information about both genotype and copy number. Whereas the x, y intensities at a single probe are (usually) sufficient to determine genotype at the corresponding target SNP, noise in the hybridization signal means that intensities from adjacent probes must be aggregated to assess copy number. Inter-probe variability in x, y and their relative magnitude is accomodated by two transformations introduced by Peiffer *et al.* (2006): the B-allele (pseudo-)frequency (BAF) and log2-intensity ratio (LRR). Both quantities are calculated with respect to intensity values derived from a large set of reference samples.

The B-allele frequency is not, in fact, anything to do with a true allele frequency, but rather a measure of the distance of a sample from the *BB* homozygous reference cluster, scaled and truncated to fall in $[0, 1]$. At homozygous markers BAF takes values near 0 or 1; at heterozygous markers it takes values near 0.5. The LRR is $\log 2(R/R_{ref})$ where $R = x + y$ and $R_{ref} = x_{ref} + y_{ref}$, and in turn x_{ref}, y_{ref} are the centroid of the reference cluster to which the same is nearest. It is expected to have mean zero across the array, in the absence of aneuploidy or large structural variants.

Technical artefacts at the hybridization or scanning steps can introduce bias in BAF and LRR. The function `argyle::tQN()` (re-)implements a normalization procedure described in Staaf *et al.* (2008), using code borrowed from the `CLASP` package (Didion *et al.* (2014)). Briefly, quantile normalization (see Bolstad *et al.* (2003)) is applied to the array-wide x and y intensities per sample to account for systematic differences in signal intensity from the two fluorophores used in the Infinium chemistry. Relative differences between the pre- and post-normalization intensities are subject to a pre-specified threshold (by default, 1.5). Then BAF and LRR are computed using a matrix of (AA, AB, BB) reference cluster centroids.

The tQN procedure is somewhat time-consuming; for the purposes of this vignette we demonstrate it on a single sample (an F1 offspring of a cross between the inbred mouse strains A/J and NOD/ShiLtJ.) The `clusters` object contains reference cluster centroids for the GigaMUGA array.

```
ex.norm <- tQN(ex[ markers(ex)$chr != "chrY","AD_15423_F" ], clusters = clusters)
```

```
## Performing tQN normalization...
## Done.
```

Use the function `bafplot()` to see the result, with smoothed fits superposed in red.

```
bafplot(ex.norm, "AD_15423_F")
```

As expected, this sample is euploid (at least for the chromsomes shown) and mostly heterozygous. A discussion of the patterns of BAF and LRR expected due to aneuploidy or sample contamination can be found in Didion *et al.* (2014).

Analysis of experimental crosses

To demonstrate the use of `argyle` for designing and analysing a laboratory experiment, we consider a hypothetical F2 cross between the inbred mouse strains A/J (coded 'A' in the example dataset) and NOD/ShiLtJ (coded 'D'). To analyze genotypes from the F2 offspring, we need to know the following:

- which markers are homozygous in the parents
- which markers are predicted to be segregating between the parents
- which markers work as predicted (homozygous in the parents, predicted to segregate, heterozygous in the F1)

First, see that 5 A/J samples (AA) and 8 NOD/ShiLtJ samples (DD), and and 2 corresponding F1 hybrids are present in the sample dataset.

```
xtabs(~ fid, samples(ex))
```

```
## fid
## AA AB AC AD AE AF AG AH BB BC BD BE BF BG BH CC CD CE CF CG CH DD DE DF DG
##  5  2  1  2  2  1  2  1  5  2  2  3  4  2  3  8  3  3  3  3  3  8  2  2  1
## DH EE EH FF FG FH GG GH HH
##  3  8  2  8  2  3  8  1  8
```

Before going any further, genotypes must be converted to numeric encoding.

```
ex <- recode(ex, "01")
```

```
## Recoding to 0/1/2 using reference alleles.
```

The `genoapply()` function is a generalization of `R`'s `apply` family for `genotypes` objects. Just as `apply(x, 1, ...)` applies a function over the rows of a matrix, `genoapply(x, margin = 1, grp, fn, ...)` applies `fn()` by samples (rows in the genotypes matrix.) Unlike `apply()`, the `genoapply()` function takes an additional grouping expression `grp` to apply a function by marker (with `margin = 1`) or sample groups (with `margin = 2`) defined by the value of `grp`. We demonstrate `genoapply()` to compute the consensus genotype of the parental strains in our example cross. For these analyses we will, of course, ignore the Y-chromosome.

```
ex <- subset(ex, chr != "Y")
parents <- subset(ex, fid == "AA" | fid == "DD", by = "samples")
cons <- genoapply(parents, 2, .(fid), consensus)
```

Note that, since the grouping expression was wrapped in `.()`, it was evaluated in the context of the sample data. This behavior is intended to make the syntax less clunky. The `consensus()` function takes a `genotypes` as input and returns the consensus genotype across all samples at each marker. (See `?consensus` for details.)

The object `cons` is a list of numeric vectors of consensus genotypes for the parent strains. Combine them into a new `genotypes` as follows:

```
cons <- genotypes( do.call(cbind, cons),
                   map = markers(parents),
                   alleles = "01" )
```

Identify markers with fixed differences between the parents using `fixed.diffs()`, and count them.

```
is.diff <- fixed.diffs(cons[ ,c("AA","DD") ])
sum(is.diff)
```

```
## [1] 2903
```

There are up to 2903 markers informative in an F2 cross between these strains, of 14319 total markers on the array.

Now predict the genotype of the F1s and check how many of those predictions match the observed genotype in the (A/JxNOD/ShiLtJ)F1 sample in dataset. Note that this prediction includes markers with the same homozygous genotype in both parents (which should be homozygous for that same genotype in the offspring), and with opposite homozygous genotypes between the parents (which should be heterozygous in the offspring.) Markers heterozygous in the parents will be marked as missing in the offspring. Really we only care about the informative markers:

```
f1s <- predict.f1(cons)
```

```
## Predicting F1 genotypes for 1 pairs of parents...
```

```
table(f1s[is.diff] == ex[ is.diff,"AD_15423_F" ], useNA = "always")
```

```
##
## FALSE  TRUE  <NA>
##     3  2898     2
```

All but a few of the 2903 informative markers perform as expected in the heterozygous state.

The default genotype encoding (either arbitrary or with respect to the major allele in some population) is not very useful for genetic analysis of an experimental cross. The `argyle` package provides `recode.to.parent()` to encode genotypes with respect to a "parent" sample (in general, an inbred strain.) This is the encoding scheme used by `R/qtl`.

```
by.mom <- recode.to.parent(ex[ ,grepl("AD", colnames(x)) ], cons[,"AA"])
summary(by.mom)
```

```
## --- by.mom ---
## A genotypes object with 14319 sites x 3 samples
## Allele encoding: parent
## Intensity data: yes (raw)
## Sample metadata: yes ( 2 male / 1 female / 0 unknown )
## Filters set: 0 sites / 1 samples
```

Once genotypes are in the `"parent"` encoding, we could convert them to an `R/qtl::cross` object to use that package for genetic mapping.

```
fake.cross <- as.rqtl(by.mom)
```

```
## Exporting genotypes at 14236 markers on 3 chromosomes.
## Converting genotypes...
## Done.
```

Population-based analyses (and PLINK)

To demonstrate the use of `argyle` on population-genetic data, we use genotypes (on chr19) from 28 wild mice genotyped on the Mouse Diversity Array (Yang *et al.* (2011)). These are provided as a `PLINK` fileset (`wild.chr19.*`) in this package's `data/` directory. The `fid` column of these samples indicates their mouse subspecies of origin.

First load the genotypes from the `PLINK` fileset. Genotypes from `PLINK` are always loaded in the `"01"` encoding.

```
ff <- system.file("data", "wild.chr19.bed", package = "argyle")
wild <- read.plink(ff)
```

```
## Reading family info from: </Users/apm/Dropbox/pmdvlab/argyle/data/wild.chr19.fam>
## Reading marker info from: </Users/apm/Dropbox/pmdvlab/argyle/data/wild.chr19.bim>
## Reading binary genotypes from: </Users/apm/Dropbox/pmdvlab/argyle/data/wild.chr19.bed>
```

```
summary(wild)
```

```
## --- wild ---
## A genotypes object with 14306 sites x 28 samples
## Allele encoding: 01
## Intensity data: no
## Sample metadata: yes ( 13 male / 15 female / 0 unknown )
## Filters set: 0 sites / 0 samples
```

Check for gross population structure using PCA on the genotypes matrix using `pca()`. The return object is a dataframe with samples projected onto the top K PCs, and has class `pca.result`.

```
pc <- pca(wild, K = 5)
```

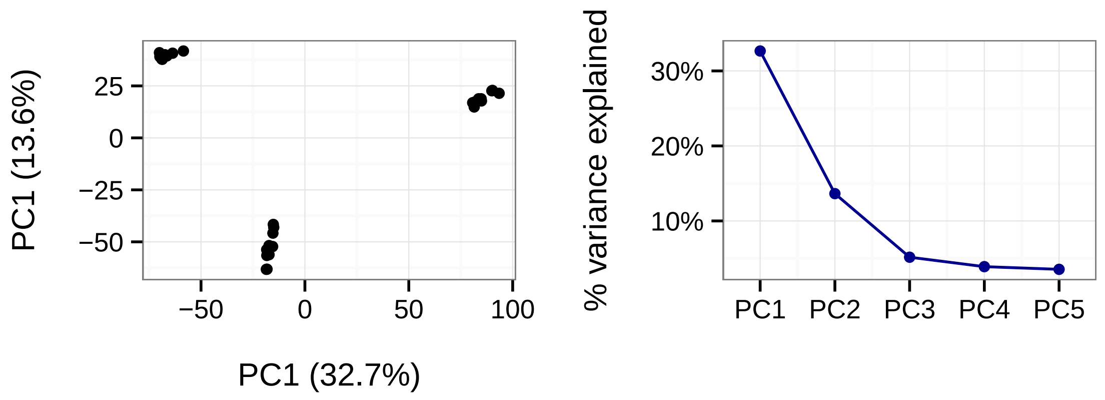
```
## Computing principal components of genotypes matrix...
##  (using base::prcomp() ...)
##  replacing missing values with minor-allele frequency...
## Done.
```

```
head(pc)
```

```
##       iid fid mom dad sex pheno       PC1       PC2        PC3         PC4
## IN13 IN13 cas   0   0   1    NA -15.21280 -41.62696  1.2044626 -0.56814144
## IN17 IN17 cas   0   0   1    NA -18.55779 -63.21966 -2.7021882  1.61250382
## IN25 IN25 cas   0   0   1    NA -15.00458 -43.05279  0.3790305 -1.95811247
## IN34 IN34 cas   0   0   1    NA -15.40547 -45.81034  0.6665433  0.09032204
## IN38 IN38 cas   0   0   2    NA -18.33270 -56.54318 -0.7055374 -2.08189524
## IN40 IN40 cas   0   0   2    NA -18.39850 -53.75009  0.2088183 -0.49792005
##            PC5
## IN13  1.9811274
## IN17 -4.7607884
## IN25  2.2605363
## IN34  0.9267065
## IN38 -2.4206857
## IN40 -1.3684164
```

The `plot()` generic is implemented for `pca.result` objects.

```
plot(pc, screeplot = TRUE)
```



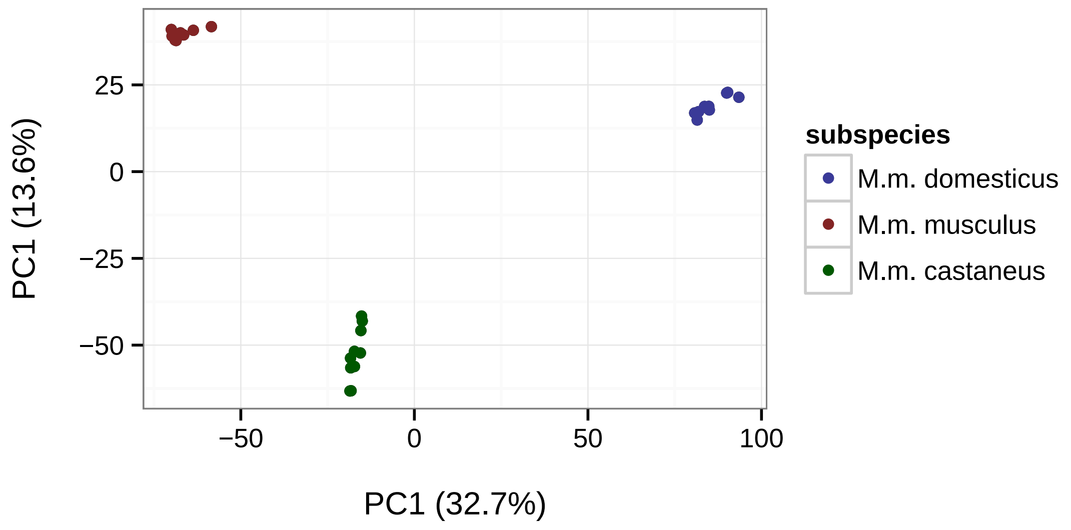
The panel at left shows the usual PCA plot, of samples projected onto principal components of the genotypes matrix. The panel at right is the "scree plot" of the relative magnitude of successive eigenvalues associated with the principal components.

With `screeplot = FALSE` we can suppress the "scree plot". In this case a `ggplot` object is returned, to which we can add layers to make a more informative plot.

```
library(ggplot2)

pc$fid <- factor(pc$fid, levels = c("dom","mus","cas"),
                 labels = c("M.m. domesticus","M.m. musculus","M.m. castaneus"))
fid.cols <- scales::muted(c("blue","red","green"))

plot(pc, screeplot = FALSE) +
    geom_point(aes(colour = fid)) +
    scale_colour_manual("subspecies", values = fid.cols)
```



The PCA reveals the expected differentiation between the three mouse subspecies: *Mus musculus domesticus*, *M. m. musculus* and *M. m. castaneus*.

We could also, for example, inspect the distribution of minor-allele frequencies within each subspecies by wrapping the function `maf()` in a call to `genoapply()`.

```
wild <- recode(wild, "relative")
```

```
## Recoding to 0/1/2 using empirical frequencies.
```

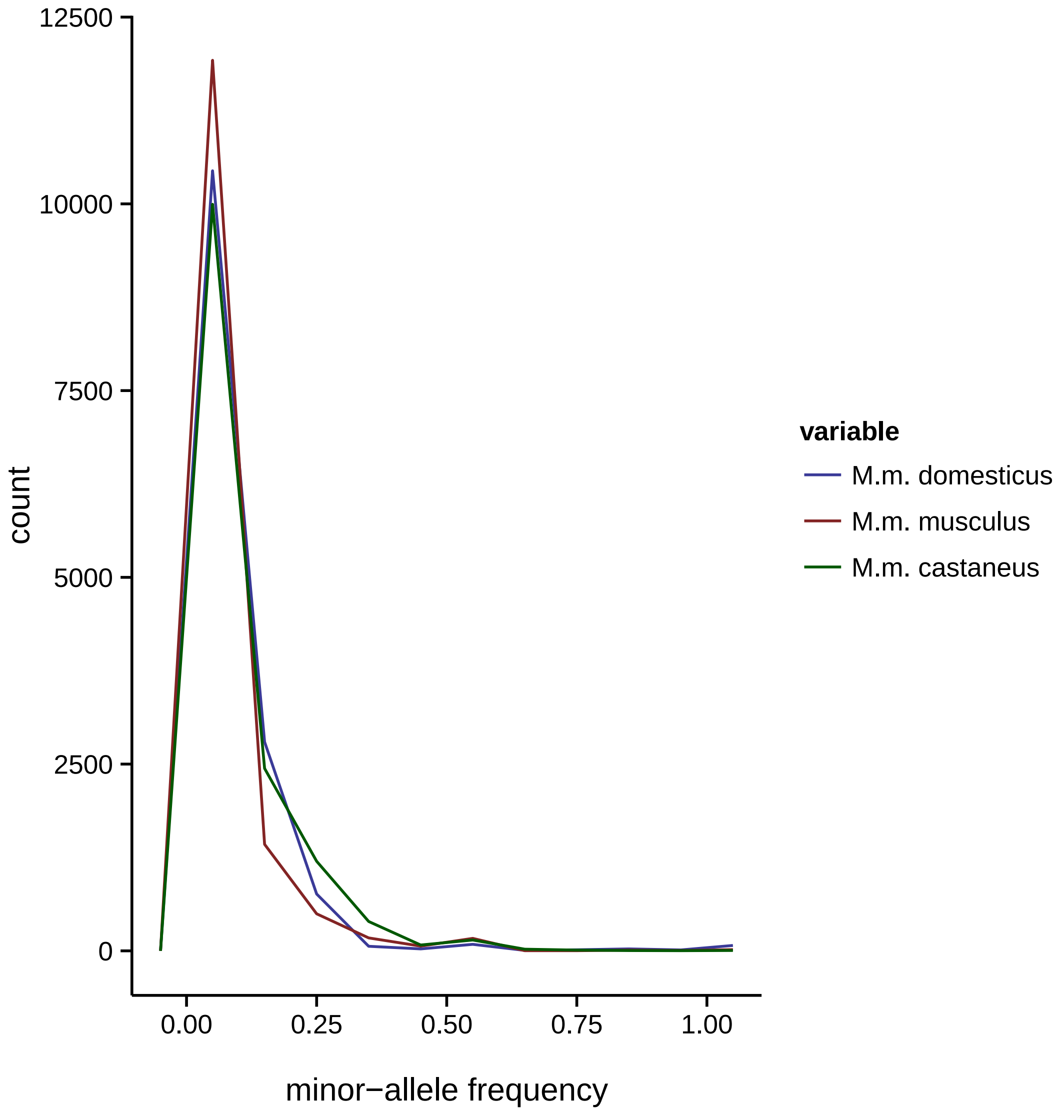
```
mafs.by.ss <- genoapply(wild, 2, .(fid), maf)

mafs <- do.call(cbind, mafs.by.ss)
mafs <- as.data.frame(mafs)
mafs$marker <- rownames(mafs)

mafs.m <- reshape2::melt(mafs, id.var = "marker")
mafs.m$variable <- factor(mafs.m$variable, levels = c("dom","mus","cas"),
                          labels = c("M.m. domesticus","M.m. musculus","M.m. castaneus"))
```

```
fid.cols <- scales::muted(c("blue","red","green"))

ggplot(mafs.m) +
    geom_freqpoly(aes(x = value, colour = variable), binwidth = 0.1) +
    scale_colour_manual(values = fid.cols) +
    theme_classic() +
    xlab("\nminor-allele frequency")
```



Wrappers for several `PLINK` commands are provided in `argyle`:

- `pca.plink(...)` – perform PCA
- `mds.plink(...)` – perform classical multidimensional scaling (MDS)
- `filter.plink(...)` – filter genotypes by missingness, minor-allele frequency, genomic position. . .
- `ld.plink(...)` – calculate pairwise LD via an EM algorithm
- `prune.plink(...)` – pruner markers by LD, to obtain a set in approximate linkage equilibrium
- `assoc.plink(...)` – perform association tests on binary or quantitative traits under a variety of models

These wrappers issue a system call to the `plink` executable (which must be in the users '$PATH' or Windows equivalent), then check that the expected output files are present. If so, they are read into the appropriate `R` object (usually dataframe or matrix); if not, `argyle` attempts to fail politely with a useful error message. Command-line output from `PLINK` is echoed to the `R` terminal.

Using the `PLINK` wrappers on an existing `PLINK` fileset **does not** require first loading the genotype matrix into the `R session.  Simply create a pointer to it using`plinkfy()':

```r
#$ recall that 'ff' is the path to the PLINK fileset of wild-mouse genotypes
rm(wild)
wild <- plinkify(ff, where = tempdir())

summary(wild)
```

```
## -- Pointer to a PLINK fileset --
## Source: /Users/apm/Dropbox/pmdvlab/argyle/data/wild.chr19.bed
## Ouput dir: /private/var/folders/_r/xn9svcws2sv9xns429lr0x_00000gp/T/RtmpCtkhyI
```

The resulting pointer has class `plink`. All the `PLINK` wrappers take such an object as their first argument. The `where` argument specifies the location to which output should be written; the default is the temporary directory corresponding to the current `R` session.

> **Safety warning**: `PLINK` will, by default, place its output in the same directory as the input fileset. This creates risk of overwriting the input if the user is not careful. Unless there is good reason to do otherwise, leave `where` set to the default to avoid clobbering important data.