Package 'tidypopgen'

May 8, 2026

Title Tidy Population Genetics

Version 0.4.3

Description We provide a tidy grammar of population genetics, facilitating the manipulation and analysis of data on biallelic single nucleotide polymorphisms (SNPs). 'tidypopgen' scales to very large genetic datasets by storing genotypes on disk, and performing operations on them in chunks, without ever loading all data in memory. The full functionalities of the package are described in Carter et al. (2025) <doi:10.1111/2041-210x.70204>.

License GPL (>= 3)

Encoding UTF-8

Language en-GB

URL https://github.com/EvolEcolGroup/tidypopgen, https://evolecolgroup.github.io/tidypopgen/

BugReports https://github.com/EvolEcolGroup/tidypopgen/issues

RoxygenNote 7.3.3

Depends R (>= 3.5.0), dplyr, tibble

Imports bigparallelr, bigsnpr, bigstatsr, foreach, generics, ggplot2, methods, MASS, patchwork, runner, rlang, sf, stats, tidyselect, tidyr, utils, Rcpp, UpSetR, vctrs

Suggests adegenet, admixtools, broom, data.table, hierfstat, knitr, detectRUNS, LEA, RhpcBLASctl, rmarkdown, rnaturalearth, rnaturalearthdata, readr, reticulate, testthat (>= 3.0.0), vcfR, xgboost, spelling

Additional_repositories https://evolecolgroup.r-universe.dev/

VignetteBuilder knitr

Config/testthat/edition 3

LinkingTo Rcpp, RcppArmadillo (>= 0.9.600), bigstatsr, rmio

LazyData true

Config/Needs/website rmarkdown

NeedsCompilation yes

Author Evie Carter [aut],
Eirlys Tysall [aut],
Andrea Manica [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0003-1895-450X>),
Chang Christopher [ctb] (Author of Hardy-Weinberg Equilibrium algorithm in PLINK 1.90, used in loci_hwe()),
Shaun Purcell [ctb] (Author of Hardy-Weinberg Equilibrium algorithm in PLINK 1.90, used in loci_hwe()),
Bengtsson Henrik [ctb] (Author of countLines in R.utils, modified for .vcf in count_vcf_variants())

Maintainer Andrea Manica <am315@cam.ac.uk>

Repository CRAN

Date/Publication 2026-01-24 00:30:02 UTC

Contents

`arrange.gen_tbl` *An arrange method for* `gen_tibble` *objects*

Description

An arrange method for `gen_tibble` objects

Usage

```
## S3 method for class 'gen_tbl'
arrange(..., deparse.level = 1)
```

Arguments

`...`	a gen_tibble and a data.frame or tibble
`deparse.level`	an integer controlling the construction of column names.

Value

a `gen_tibble`

Examples

```
test_gt <- load_example_gt("gen_tbl")
test_gt %>% arrange(id)
```

`arrange.grouped_gen_tbl` *An arrange method for grouped* `gen_tibble` *objects*

Description

An arrange method for grouped `gen_tibble` objects

Usage

```
## S3 method for class 'grouped_gen_tbl'
arrange(..., deparse.level = 1)
```

Arguments

`...`	a gen_tibble and a data.frame or tibble
`deparse.level`	an integer controlling the construction of column names.

Value

a grouped `gen_tibble`

Examples

```
test_gt <- load_example_gt("grouped_gen_tbl")
test_gt %>% arrange(id)
test_gt <- load_example_gt("grouped_gen_tbl_sf")
test_gt %>% arrange(id)
```

augment.gt_dapc *Augment data with information from a gt_dapc object*

Description

Augment for gt_dapc accepts a model object and a dataset and adds scores to each observation in the dataset. Scores for each component are stored in a separate column, which is given name with the pattern ".fittedLD1", ".fittedLD2", etc. For consistency with broom::augment.prcomp, a column ".rownames" is also returned; it is a copy of 'id', but it ensures that any scripts written for data augmented with broom::augment.prcomp will work out of the box (this is especially helpful when adapting plotting scripts).

Usage

```
## S3 method for class 'gt_dapc'
augment(x, data = NULL, k = NULL, ...)
```

Arguments

x	A gt_dapc object returned by gt_dapc().
data	the gen_tibble used to run the PCA.
k	the number of components to add
...	Not used. Needed to match generic signature only.

Value

A gen_tibble containing the original data along with additional columns containing each observation's projection into PCA space.

See Also

gt_dapc() gt_dapc_tidiers

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

# Create PCA and run DAPC
```

```
pca <- gt_pca_partialSVD(lobsters)
populations <- as.factor(lobsters$population)
dapc_res <- gt_dapc(pca, n_pca = 6, n_da = 2, pop = populations)

# Augment the gen_tibble with the DAPC scores
augment(dapc_res, data = lobsters, k = 2)
```

`augment_gt_pca` *Augment data with information from a gt_pca object*

Description

Augment for `gt_pca` accepts a model object and a dataset and adds scores to each observation in the dataset. Scores for each component are stored in a separate column, which is given name with the pattern ".fittedPC1", ".fittedPC2", etc. For consistency with broom::augment.prcomp, a column ".rownames" is also returned; it is a copy of 'id', but it ensures that any scripts written for data augmented with broom::augment.prcomp will work out of the box (this is especially helpful when adapting plotting scripts).

Usage

```
## S3 method for class 'gt_pca'
augment(x, data = NULL, k = NULL, ...)
```

Arguments

`x`	A `gt_pca` object returned by one of the `gt_pca_*` functions.
`data`	the `gen_tibble` used to run the PCA.
`k`	the number of components to add
`...`	Not used. Needed to match generic signature only.

Value

A gen_tibble containing the original data along with additional columns containing each observation's projection into PCA space.

See Also

`gt_pca_autoSVD()` gt_pca_tidiers

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

# Create PCA object
pca <- gt_pca_partialSVD(lobsters)

# Augment the gen_tibble with PCA scores
augment(pca, data = lobsters)

# Adjust the number of components to add
augment(pca, data = lobsters, k = 2)
```

---

augment_loci *Augment the loci table with information from a analysis object*

Description

`augment_loci` add columns to the loci table of a `gen_tibble` related to information from a given analysis.

Usage

```
augment_loci(x, data, ...)
```

Arguments

`x`	An object returned by one of the `gt_` functions (e.g. `gt_pca()`).
`data`	the `gen_tibble` used to run the PCA.
`...`	Additional parameters passed to the individual methods.

Value

A loci tibble with additional columns. If `data` is missing, a tibble of the information, with a column `.rownames` giving the loci names.

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

# Create PCA
pca <- gt_pca_partialSVD(lobsters)

# Augment the gen_tibble with the PCA scores
augment_loci(pca, data = lobsters)
```

`augment_loci_gt_pca` *Augment the loci table with information from a gt_pca object*

Description

Augment for `gt_pca` accepts a model object and a `gen_tibble` and adds loadings for each locus to the loci table. Loadings for each component are stored in a separate column, which is given name with the pattern ".loadingPC1", ".loadingPC2", etc. If `data` is missing, then a tibble with the loadings is returned.

Usage

```
## S3 method for class 'gt_pca'
augment_loci(x, data = NULL, k = NULL, ...)
```

Arguments

`x`	A `gt_pca` object returned by one of the `gt_pca_*` functions.
`data`	the `gen_tibble` used to run the PCA.
`k`	the number of components to add
`...`	Not used. Needed to match generic signature only.

Value

A gen_tibble with a loadings added to the loci tibble (accessible with `show_loci()`. If `data` is missing, a tibble of loadings.

See Also

gt_pca_autoSVD() gt_pca_tidiers

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

# Create PCA
pca <- gt_pca_partialSVD(lobsters)

# Augment the gen_tibble with the PCA scores
augment_loci(pca, data = lobsters)
```

---

augment_q_matrix *Augment data with information from a q_matrix object*

Description

Augment for q_matrix accepts a model object and a dataset and adds Q values to each observation in the dataset. Q values are stored in separate columns, which is given name with the pattern ".Q1",".Q2", etc. For consistency with broom::augment.prcomp, a column ".rownames" is also returned; it is a copy of 'id', but it ensures that any scripts written for data augmented with broom::augment.prcomp will work out of the box (this is especially helpful when adapting plotting scripts).

Usage

```
## S3 method for class 'q_matrix'
augment(x, data = NULL, ...)
```

Arguments

x	A q_matrix object
data	the gen_tibble used to run the clustering algorithm
...	Not used. Needed to match generic signature only.

Value

A gen_tibble containing the original data along with additional columns containing each observation's Q values.

Examples

```
# run the example only if we have the package installed
if (requireNamespace("LEA", quietly = TRUE)) {
  example_gt <- load_example_gt("gen_tbl")

  # Create a gt_admix object
  admix_obj <- example_gt %>% gt_snmf(k = 1:3, project = "force")

  # Extract a Q matrix
  q_mat_k3 <- get_q_matrix(admix_obj, k = 3, run = 1)

  # Augment the gen_tibble with Q values
  augment(q_mat_k3, data = example_gt)
}
```

---

autoplot.gt_cluster_pca

Autoplots for gt_cluster_pca *objects*

Description

For gt_cluster_pca, autoplot produces a plot of a metric of choice ('BIC', 'AIC' or 'WSS') against the number of clusters (*k*). This plot is can be used to infer the best value of *k*, which corresponds to the smallest value of the metric (the minimum in an 'elbow' shaped curve). In some cases, there is not 'elbow' and the metric keeps decreasing with increasing *k*; in such cases, it is customary to choose the value of *k* at which the decrease in the metric reaches as plateau. For a programmatic way of choosing *k*, use gt_cluster_pca_best_k().

Usage

```
## S3 method for class 'gt_cluster_pca'
autoplot(object, metric = c("BIC", "AIC", "WSS"), ...)
```

Arguments

object	an object of class gt_dapc
metric	the metric to plot on the y axis, one of 'BIC', 'AIC', or 'WSS' (with sum of squares)
...	not currently used.

Details

autoplot produces simple plots to quickly inspect an object. They are not customisable; we recommend that you use ggplot2 to produce publication ready plots.

Value

a ggplot2 object

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

# Create PCA object
pca <- gt_pca_partialSVD(lobsters)

# Run clustering on the first 10 PCs
cluster_pca <- gt_cluster_pca(
  x = pca,
  n_pca = 10,
  k_clusters = c(1, 5),
  method = "kmeans",
  n_iter = 1e5,
  n_start = 10,
  quiet = FALSE
)

# Autoplot BIC
autoplot(cluster_pca, metric = "BIC")

# # Autoplot AIC
autoplot(cluster_pca, metric = "AIC")

# # Autoplot WSS
autoplot(cluster_pca, metric = "WSS")
```

---

autoplot.gt_dapc *Autoplots for* gt_dapc *objects*

Description

For gt_dapc, the following types of plots are available:

- screeplot: a plot of the eigenvalues of the discriminant axes
- scores a scatterplot of the scores of each individual on two discriminant axes (defined by ld)

- `loadings` a plot of loadings of all loci for a discriminant axis (chosen with `ld`)
- `components` a bar plot showing the probability of assignment to each cluster

Usage

```
## S3 method for class 'gt_dapc'
autoplot(
  object,
  type = c("screeplot", "scores", "loadings", "components"),
  ld = NULL,
  group = NULL,
  n_col = 1,
  ...
)
```

Arguments

`object`	an object of class `gt_dapc`
`type`	the type of plot (one of "screeplot", "scores", "loadings", and "components")
`ld`	the principal components to be plotted: for scores, a pair of values e.g. c(1,2); for `loadings` either one or more values.
`group`	a vector of group memberships to order the individuals in "components" plot. If NULL, the clusters used for the DAPC will be used.
`n_col`	for `loadings` plots, if multiple LD axis are plotted, how many columns should be used.
`...`	not currently used.

Details

`autoplot` produces simple plots to quickly inspect an object. They are not customisable; we recommend that you use `ggplot2` to produce publication ready plots.

Value

a `ggplot2` object

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")
```

```
# Create PCA and run DAPC
pca <- gt_pca_partialSVD(lobsters)
populations <- as.factor(lobsters$population)
dapc_res <- gt_dapc(pca, n_pca = 6, n_da = 2, pop = populations)

# Screeplot
autoplot(dapc_res, type = "screeplot")

# Scores plot
autoplot(dapc_res, type = "scores", ld = c(1, 2))

# Loadings plot
autoplot(dapc_res, type = "loadings", ld = 1)

# Components plot
autoplot(dapc_res, type = "components", group = populations)
```

`autoplot.qc_report_indiv`

Autoplots for `qc_report_indiv` *objects*

Description

For `qc_report_indiv`, the following types of plots are available:

- `scatter`: a plot of missingness and observed heterozygosity within individuals.
- `relatedness`: a histogram of paired kinship coefficients
- `histogram`: for gen_tibbles containing pseudohaploid data, a histogram of missingness, split by ploidy.

Usage

```
## S3 method for class 'qc_report_indiv'
autoplot(
  object,
  type = c("scatter", "relatedness", "histogram"),
  miss_threshold = 0.05,
  kings_threshold = NULL,
  ...
)
```

Arguments

`object`	an object of class `qc_report_indiv`
`type`	the type of plot (`scatter`,`relatedness`,`histogram`)
`miss_threshold`	a threshold for the accepted rate of missingness within individuals

kings_threshold
: an optional numeric, a threshold of relatedness for the sample

...
: not currently used.

Details

autoplot produces simple plots to quickly inspect an object. They are not customisable; we recommend that you use ggplot2 to produce publication ready plots.

Value

a ggplot2 object

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
example_gt <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Create QC report for individuals
indiv_report <- example_gt %>% qc_report_indiv()

# Autoplot missingness and observed heterozygosity
autoplot(indiv_report, type = "scatter", miss_threshold = 0.1)

# Create QC report with kinship filtering
indiv_report_rel <-
  example_gt %>% qc_report_indiv(kings_threshold = "second")

# Autoplot relatedness
autoplot(indiv_report_rel, type = "relatedness", kings_threshold = "second")
```

---

autoplot.qc_report_loci

Autoplots for qc_report_loci *objects*

Description

For qc_report_loci, the following types of plots are available:

- overview: an UpSet plot, giving counts of snps over the threshold for missingness, minor allele frequency, and Hardy-Weinberg equilibrium P-value, and visualising the interaction between these

- `all`: a four panel plot, containing `missing high maf`, `missing low maf`, `hwe`, and `significant hwe` plots
- `missing`: a histogram of proportion of missing data
- `missing low maf`: a histogram of the proportion of missing data for snps with low minor allele frequency
- `missing high maf`:a histogram of the proportion of missing data for snps with high minor allele frequency
- `maf`: a histogram of minor allele frequency
- `hwe`: a histogram of HWE exact test p-values
- `significant hwe`: a histogram of significant HWE exact test p-values

Usage

```
## S3 method for class 'qc_report_loci'
autoplot(
  object,
  type = c("overview", "all", "missing", "missing low maf", "missing high maf", "maf",
    "hwe", "significant hwe"),
  maf_threshold = 0.05,
  miss_threshold = 0.01,
  hwe_p = 0.01,
  ...
)
```

Arguments

`object`	an object of class `qc_report_loci`
`type`	the type of plot (one of `overview`, `all`, `missing`, `missing low maf`, `missing high maf`, `maf`, `hwe`, and `significant hwe`)
`maf_threshold`	default 0.05, a threshold for the accepted rate of minor allele frequency of loci
`miss_threshold`	default 0.01, a threshold for the accepted rate of missingness per loci
`hwe_p`	default 0.01, a threshold of significance for Hardy-Weinberg exact p-values
`...`	not currently used.

Details

`autoplot` produces simple plots to quickly inspect an object. They are not customisable; we recommend that you use `ggplot2` to produce publication ready plots.

Value

a `ggplot2` object

Examples

```
# Create a gen_tibble
bed_file <-
  system.file("extdata", "related", "families.bed", package = "tidypopgen")
example_gt <- gen_tibble(bed_file,
  backingfile = tempfile("families"),
  quiet = TRUE,
  valid_alleles = c("1", "2")
)

loci_report <- example_gt %>% qc_report_loci()

# Plot the QC report overview
autoplot(loci_report, type = "overview")

# Plot the QC report all
autoplot(loci_report, type = "all")

# Plot missing data
autoplot(loci_report, type = "missing")

# Plot missing with low maf
autoplot(loci_report, type = "missing low maf", maf_threshold = 0.05)

# Plot missing with high maf
autoplot(loci_report, type = "missing high maf", maf_threshold = 0.05)

# Plot maf
autoplot(loci_report, type = "maf", maf_threshold = 0.05)

# Plot hwe
autoplot(loci_report, type = "hwe", hwe_p = 0.01)

# Plot significant hwe
autoplot(loci_report, type = "significant hwe", hwe_p = 0.01)
```

---

autoplot_gt_admix *Autoplots for* gt_admix *objects*

Description

For gt_admix, the following types of plots are available:

- cv: the cross-validation error for each value of k
- barplot a standard barplot of the admixture proportions

Usage

```
## S3 method for class 'gt_admix'
autoplot(object, type = c("cv", "barplot"), k = NULL, run = NULL, ...)
```

Arguments

object	an object of class gt_admixture
type	the type of plot (one of "cv", and "barplot")
k	the value of k to plot (for barplot type only) param repeat the repeat to plot (for barplot type only)
run	the run to plot (for barplot type only)
...	additional arguments to be passed to autoplot method for q_matrices autoplot_q_matrix(), used when type is barplot.

Details

autoplot produces simple plots to quickly inspect an object. They are not customisable; we recommend that you use ggplot2 to produce publication ready plots.

This autoplot will automatically rearrange individuals according to their id and any grouping variables if an associated 'data' gen_tibble is provided. To avoid any automatic re-sorting of individuals, set arrange_by_group and arrange_by_indiv to FALSE. See autoplot.q_matrix for further details.

Value

a ggplot2 object

Examples

```
# Read example gt_admix object
admix_obj <-
  readRDS(system.file("extdata", "anolis", "anole_adm_k3.rds",
    package = "tidypopgen"
  ))
# Cross-validation plot
autoplot(admix_obj, type = "cv")

# Basic barplot
autoplot(admix_obj, k = 3, run = 1, type = "barplot")

# Barplot with individuals arranged by Q proportion
# (using additional arguments, see `autoplot.q_matrix` for details)
autoplot(admix_obj,
  k = 3, run = 1, type = "barplot", annotate_group = TRUE,
  arrange_by_group = TRUE, arrange_by_indiv = TRUE,
  reorder_within_groups = TRUE
)
```

autoplot_gt_pca *Autoplots for* gt_pca *objects*

Description

For gt_pca, the following types of plots are available:

- screeplot: a plot of the eigenvalues of the principal components (currently it plots the singular value)
- scores a scatterplot of the scores of each individual on two principal components (defined by pc)
- loadings a plot of loadings of all loci for a given component (chosen with pc)

Usage

```
## S3 method for class 'gt_pca'
autoplot(object, type = c("screeplot", "scores", "loadings"), k = NULL, ...)
```

Arguments

object	an object of class gt_pca
type	the type of plot (one of "screeplot", "scores" and "loadings")
k	the principal components to be plotted: for scores, a pair of values e.g. c(1,2); for loadings either one or more values.
...	not currently used.

Details

autoplot produces simple plots to quickly inspect an object. They are not customisable; we recommend that you use ggplot2 to produce publication ready plots.

Value

a ggplot2 object

Examples

```
library(ggplot2)
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Remove monomorphic loci and impute
```

```
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

# Create PCA object
pca <- gt_pca_partialSVD(lobsters)

# Screeplot
autoplot(pca, type = "screeplot")

# Scores plot
autoplot(pca, type = "scores")

# Colour by population
autoplot(pca, type = "scores") + aes(colour = lobsters$population)

# Scores plot of different components
autoplot(pca, type = "scores", k = c(1, 3)) +
  aes(colour = lobsters$population)
```

`autoplot_gt_pcadapt` *Autoplots for* `gt_pcadapt` *objects*

Description

For `gt_pcadapt`, the following types of plots are available:

- `qq`: a quantile-quantile plot of the p-values from `pcadapt` (wrapping `bigsnpr::snp_qq()`)
- `manhattan` a manhattan plot of the p-values from `pcadapt` (wrapping `bigsnpr::snp_manhattan()`)

Usage

```
## S3 method for class 'gt_pcadapt'
autoplot(object, type = c("qq", "manhattan"), ...)
```

Arguments

`object`	an object of class `gt_pcadapt`
`type`	the type of plot (one of "qq", and "manhattan")
`...`	further arguments to be passed to `bigsnpr::snp_qq()` or `bigsnpr::snp_manhattan()`.

Details

`autoplot` produces simple plots to quickly inspect an object. They are not customisable; we recommend that you use `ggplot2` to produce publication ready plots.

Value

a `ggplot2` object

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

# Create PCA object
pca <- gt_pca_partialSVD(lobsters)

# Create a gt_pcadapt object
pcadapt_obj <- gt_pcadapt(lobsters, pca, k = 2)

# Plot the p-values from pcadapt
autoplot(pcadapt_obj, type = "qq")

# Plot the manhattan plot of the p-values from pcadapt
autoplot(pcadapt_obj, type = "manhattan")
```

---

autoplot_q_matrix *Autoplots for* q_matrix *objects*

Description

This autoplot will automatically rearrange individuals according to their id and any grouping variables if an associated 'data' gen_tibble is provided. To avoid any automatic re-sorting of individuals, set arrange_by_group and arrange_by_indiv to FALSE.

Usage

```
## S3 method for class 'q_matrix'
autoplot(
  object,
  data = NULL,
  annotate_group = TRUE,
  arrange_by_group = TRUE,
  arrange_by_indiv = TRUE,
  reorder_within_groups = FALSE,
  ...
)
```

Arguments

object	A Q matrix object (as returned by q_matrix()).
data	An associated tibble (e.g. a gen_tibble), with the individuals in the same order as the data used to generate the Q matrix
annotate_group	Boolean determining whether to annotate the plot with the group information
arrange_by_group	Boolean determining whether to arrange the individuals by group. If the grouping variable in the gen_tibble or the metadata of the gt_admixt object is a factor, the data will be ordered by the levels of the factor; else it will be ordered alphabetically.
arrange_by_indiv	Boolean determining whether to arrange the individuals by their individual id (if arrange_by_group is TRUE, they will be arranged by group first and then by individual id, i.e. within each group). If id in the get_tibble or the metadata of the gt_admix object is a factor, it will be ordered by the levels of the factor; else it will be ordered alphabetically.
reorder_within_groups	Boolean determining whether to reorder the individuals within each group based on their ancestry proportion (note that this is not advised if you are making multiple plots, as you would get a different order for each plot!). If TRUE, annotate_group must also be TRUE.
...	not currently used.

Value

a barplot of individuals, coloured by ancestry proportion

Examples

```
# Read example gt_admix obejct
admix_obj <-
  readRDS(system.file("extdata", "anolis", "anole_adm_k3.rds",
    package = "tidypopgen"
  ))

# Extract a Q matrix
q_mat_k3 <- get_q_matrix(admix_obj, k = 3, run = 1)

# Basic autoplot
autoplot(q_mat_k3, annotate_group = FALSE, arrange_by_group = FALSE)

# To arrange individuals by group and by Q proportion
autoplot(q_mat_k3,
  annotate_group = TRUE, arrange_by_group = TRUE,
  arrange_by_indiv = TRUE, reorder_within_groups = TRUE
)
```

c.gt_admix *Combine method for gt_admix objects*

Description

Combine method for gt_admix objects

Usage

```
## S3 method for class 'gt_admix'
c(..., match_attributes = TRUE)
```

Arguments

...	A list of gt_admix objects
match_attributes	boolean, determining whether all attributes (id, group and algorithm) of the gt_admix objects to be combined must be an exact match (TRUE, the default), or whether non-matching attributes should be ignored (FALSE)

Value

A gt_admix object with the combined data

Examples

```
# run the example only if we have the package installed
if (requireNamespace("LEA", quietly = TRUE)) {
  example_gt <- load_example_gt("gen_tbl")

  # Create a gt_admix object
  admix_obj <- example_gt %>% gt_snmf(k = 1:3, project = "force")

  # Create a second gt_admix object
  admix_obj2 <- example_gt %>% gt_snmf(k = 2:4, project = "force")

  # Combine the two gt_admix objects
  new_admix_obj <- c(admix_obj, admix_obj2)
  summary(new_admix_obj)
}
```

cbind.gen_tbl *Combine a gen_tibble to a data.frame or tibble by column*

Description

A cbind() method to merge gen_tibble objects with data.frames and normal tibbles. Whilst this works, it is not ideal as it does not check the order of the tables, and we suggest that you use dplyr::left_join() instead. Note that cbind will not combine two gen_tibbles (i.e. it will NOT combine markers for the same individuals)

Usage

```
## S3 method for class 'gen_tbl'
cbind(..., deparse.level = 1)
```

Arguments

...	a gen_tibble and a data.frame or tibble
deparse.level	an integer controlling the construction of column names. See cbind for details.

Value

a gen_tibble

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Create a dataframe to combine with the gen_tibble
df <- data.frame(region = c("A", "A", "B", "B", "A", "B", "B"))

# Combine the gen_tibble with the dataframe
example_gt <- cbind(example_gt, df)
```

count_loci *Count the number of loci in a* gen_tibble

Description

Count the number of loci in gen_tibble (or directly from its genotype column).

Usage

```
count_loci(.x, ...)

## S3 method for class 'tbl_df'
count_loci(.x, ...)

## S3 method for class 'vctrs_bigSNP'
count_loci(.x, ...)
```

Arguments

.x	a gen_tibble, or a vector of class vctrs_bigSNP (usually the genotype column of a gen_tibble object).
...	currently unused.

Value

the number of loci

Examples

```
example_gt <- load_example_gt("gen_tbl")

example_gt %>% count_loci()
```

`distruct_colours` *Distruct colours*

Description

Colours in the palette used by distruct

Usage

```
distruct_colours
```

Format

A vector of 60 hex colours

filter.gen_tbl *Tidyverse methods for gt objects*

Description

A filter method for gen_tibble objects

Usage

```
## S3 method for class 'gen_tbl'
filter(..., deparse.level = 1)
```

Arguments

...	a gen_tibble and a data.frame or tibble
deparse.level	an integer controlling the construction of column names.

Value

a gen_tibble

Examples

```
test_gt <- load_example_gt("gen_tbl")
test_gt %>% filter(id %in% c("a", "c"))
```

filter.grouped_gen_tbl *A filter method for grouped* gen_tibble *objects*

Description

A filter method for grouped gen_tibble objects

Usage

```
## S3 method for class 'grouped_gen_tbl'
filter(..., deparse.level = 1)
```

Arguments

...	a gen_tibble and a data.frame or tibble
deparse.level	an integer controlling the construction of column names.

Value

a grouped gen_tibble

Examples

```
test_gt <- load_example_gt("grouped_gen_tbl")
test_gt %>% filter(id %in% c("a", "c"))
test_gt <- load_example_gt("grouped_gen_tbl_sf")
test_gt %>% filter(id %in% c("a", "c"))
```

---

filter_high_relatedness

Filter individuals based on a relationship threshold

Description

This function takes a matrix of x by y individuals containing relatedness coefficients and returns the maximum set of individuals that contains no relationships above the given threshold.

Usage

```
filter_high_relatedness(
  matrix,
  .x = NULL,
  kings_threshold = NULL,
  verbose = FALSE
)
```

Arguments

matrix	a square symmetric matrix of individuals containing relationship coefficients
.x	a gen_tibble object
kings_threshold	a threshold over which
verbose	boolean whether to report to screen

Value

a list where '1' is individual ID's to retain, '2' is individual ID's to remove, and '3' is a boolean where individuals to keep are TRUE and individuals to remove are FALSE

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Calculate relationship matrix
king_matrix <- example_gt %>% pairwise_king(as_matrix = TRUE)

# Filter individuals with threshold above 0.2
filter_high_relatedness(king_matrix, example_gt, kings_threshold = 0.2)
```

---

find_duplicated_loci *Find duplicates in the loci table*

Description

This function finds duplicated SNPs by checking the positions within each chromosome. It can return a list of duplicated SNPs or a logical value indicating whether there are any duplicated loci.

Usage

```
find_duplicated_loci(.x, error_on_false = FALSE, list_duplicates = TRUE, ...)
```

Arguments

`.x`	a vector of class `vctrs_bigSNP` (usually the `genotype` column of a `gen_tibble` object), or a `gen_tibble`.
`error_on_false`	logical, if `TRUE` an error is thrown if duplicated loci are found.
`list_duplicates`	logical, if `TRUE` returns duplicated SNP names.
`...`	other arguments passed to specific methods.

Value

If `list_duplicates` is TRUE, returns a character vector of duplicated loci names (character(0) when none). If `list_duplicates` is FALSE, returns TRUE when no duplicates exist and FALSE when duplicates are present. If `error_on_false` is TRUE and duplicates exist, an error is thrown.

Examples

```
example_gt <- load_example_gt("gen_tbl")
show_loci(example_gt) <- test_loci <- data.frame(
  big_index = c(1:6),
  name = paste0("rs", 1:6),
  chromosome = paste0("chr", c(1, 1, 1, 1, 1, 1)),
  position = as.integer(c(3, 3, 5, 65, 343, 46)),
  genetic_dist = as.double(rep(0, 6)),
  allele_ref = c("A", "T", "C", "G", "C", "T"),
  allele_alt = c("T", "C", NA, "C", "G", "A")
)
```

```
show_loci(example_gt)

# Find which loci are duplicated
example_gt %>% find_duplicated_loci()
```

---

gen_tibble *Constructor for a* gen_tibble

Description

A gen_tibble stores genotypes for individuals in a tidy format. DESCRIBE here the format

Usage

```
gen_tibble(
  x,
  ...,
  valid_alleles = c("A", "T", "C", "G"),
  missing_alleles = c("0", "."),
  backingfile = NULL,
  allow_duplicates = FALSE,
  quiet = FALSE
)

## S3 method for class 'character'
gen_tibble(
  x,
  ...,
  parser = c("cpp", "vcfR"),
  n_cores = 1,
  chunk_size = NULL,
  valid_alleles = c("A", "T", "C", "G"),
  missing_alleles = c("0", "."),
  backingfile = NULL,
  allow_duplicates = FALSE,
  quiet = FALSE
)

## S3 method for class 'matrix'
gen_tibble(
  x,
  indiv_meta,
  loci,
  ...,
  ploidy = 2,
  valid_alleles = c("A", "T", "C", "G"),
```

```
  missing_alleles = c("0", "."),
  backingfile = NULL,
  allow_duplicates = FALSE,
  quiet = FALSE
)
```

Arguments

x	can be: • a string giving the path to a PLINK BED or PED file. The associated BIM and FAM files for the BED, or MAP for PED are expected to be in the same directory and have the same file name. • a string giving the path to a RDS file storing a bigSNP object from the bigsnpr package (usually created with bigsnpr::snp_readBed()) • a string giving the path to a vcf file. Only biallelic SNPs will be considered. • a string giving the path to a *packedancestry* .geno file. The associated .ind and .snp files are expected to be in the same directory and share the same file name prefix. • a genotype matrix of dosages (0, 1, 2, NA) giving the dosage of the alternate allele.
...	if x is the name of a vcf file, additional arguments passed to vcfR::read.vcfR(). Otherwise, unused.
valid_alleles	a vector of valid allele values; it defaults to 'A','T', 'C' and 'G'.
missing_alleles	a vector of values in the BIM file/loci dataframe that indicate a missing value for the allele value (e.g. when we have a monomorphic locus with only one allele). It defaults to '0' and '.' (the same as PLINK 1.9).
backingfile	the path, including the file name without extension, for backing files used to store the data (they will be given a .bk and .RDS automatically). This is not needed if x is already an .RDS file. If x is a .BED or a *VCF* file and backingfile is left NULL, the backing file will be saved in the same directory as the bed or vcf file, using the same file name but with a different file type (.bk rather than .bed or .vcf). If x is a genotype matrix and backingfile is NULL, then a temporary file will be created (but note that R will delete it at the end of the session!)
allow_duplicates	logical. If TRUE, the tibble will allow duplicated loci (those with genomic coordinate (chromosome + position) or locus name appearing more than once). If FALSE, an error will be thrown if duplicated loci are found. These validations run before backing files are saved. Default is FALSE.
quiet	provide information on the files used to store the data
parser	the name of the parser used for *VCF*, either "cpp" to use a fast C++ parser (the default), or "vcfR" to use the R package vcfR. The latter is slower but more robust; if "cpp" gives an error, try using "vcfR" in case your *VCF* has an unusual structure.
n_cores	the number of cores to use for parallel processing

chunk_size	the number of loci or individuals (depending on the format) processed at a time (currently used if x is a vcf with parser "vcfR")
indiv_meta	a list, data.frame or tibble with compulsory columns 'id' and 'population', plus any additional metadata of interest. This is only used if x is a genotype matrix. Otherwise this information is extracted directly from the files.
loci	a data.frame or tibble, with compulsory columns 'name', 'chromosome', and 'position','genetic_dist', 'allele_ref' and 'allele_alt'. This is only used if x is a genotype matrix. Otherwise this information is extracted directly from the files.
ploidy	the ploidy of the samples (either a single value, or a vector of values for mixed ploidy). Only used if creating a gen_tibble from a matrix of data; otherwise, ploidy is determined automatically from the data as they are read.

Details

- *VCF* files: the fast cpp parser is used by default. Both cpp and vcfR parsers attempt to establish ploidy from the first variant; if that variant is found in a sex chromosome (or mtDNA), the parser will fail with 'Error: a genotype has more than max_ploidy alleles...'. To successful import such a *VCF*, change the order of variants so that the first chromosome is an autosome using a tool such as vcftools. Currently, only biallelic SNPs are supported. If haploid variants (e.g. sex chromosomes) are included in the *VCF*, they are not transformed into homozygous calls. Instead, reference alleles will be coded as 0 and alternative alleles will be coded as 1.
- *packedancestry* files: When loading *packedancestry* files, missing alleles will be converted from 'X' to NA

Value

an object of the class gen_tbl.

Note

Helper functions for accessing gen_tibble object attributes and checking gen_tibble ploidy can be found in gt_helper_functions.R

Examples

```
# Create a gen_tibble from a .bed file
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Create a gen_tibble from a .vcf file
vcf_path <-
  system.file("extdata", "anolis",
    "punctatus_t70_s10_n46_filtered.recode.vcf.gz",
    package = "tidypopgen"
```

```
  )
gen_tibble(vcf_path, quiet = TRUE, backingfile = tempfile("anolis_"))

# Create a gen_tibble from a matrix of genotypes:
test_indiv_meta <- data.frame(
  id = c("a", "b", "c"),
  population = c("pop1", "pop1", "pop2")
)
test_genotypes <- rbind(
  c(1, 1, 0, 1, 1, 0),
  c(2, 1, 0, 0, 0, 0),
  c(2, 2, 0, 0, 1, 1)
)
test_loci <- data.frame(
  name = paste0("rs", 1:6),
  chromosome = paste0("chr", c(1, 1, 1, 1, 2, 2)),
  position = as.integer(c(3, 5, 65, 343, 23, 456)),
  genetic_dist = as.double(rep(0, 6)),
  allele_ref = c("A", "T", "C", "G", "C", "T"),
  allele_alt = c("T", "C", NA, "C", "G", "A")
)

gen_tibble(
  x = test_genotypes,
  loci = test_loci,
  indiv_meta = test_indiv_meta,
  valid_alleles = c("A", "T", "C", "G"),
  quiet = TRUE
)
```

`get_p_matrix` *Return a single P matrix from a* `gt_admix` *object*

Description

This function retrieves a single P matrix from a `gt_admix` object based on the specified k value and run number.

Usage

```
get_p_matrix(x, ..., k, run)
```

Arguments

`x`	A `gt_admix` object containing P matrices
`...`	Not used
`k`	The k value of the desired P matrix
`run`	The run number of the desired P matrix

Value

A single P matrix from the gt_admix object

Examples

```
# Read example gt_admix object
admix_obj <-
  readRDS(system.file("extdata", "anolis", "anole_adm_k3.rds",
    package = "tidypopgen"
  ))

# Extract a P matrix
get_p_matrix(admix_obj, k = 3, run = 1)
```

---

get_q_matrix *Return a single Q matrix from a* gt_admix *object*

Description

This function retrieves a single Q matrix from a gt_admix object based on the specified k value and run number.

Usage

```
get_q_matrix(x, ..., k, run)
```

Arguments

x	A gt_admix object containing multiple Q matrices
...	Not used
k	The k value of the desired Q matrix
run	The run number of the desired Q matrix

Value

A single Q matrix from the gt_admix object

Examples

```
# Read example gt_admix obejct
admix_obj <-
  readRDS(system.file("extdata", "anolis", "anole_adm_k3.rds",
    package = "tidypopgen"
  ))

# Extract a Q matrix
get_q_matrix(admix_obj, k = 3, run = 1)
```

gt_add_sf *Add an simple feature geometry to a* gen_tibble

Description

gt_add_sf adds an active sf geometry column to a gen_tibble object. The resulting gen_tbl inherits from sf and can be used with functions from the sf package. It is possible to either create a sf::sfc geometry column from coordinates, or to provide an existing geometry column (which will then become the active geometry for sf).

Usage

```
gt_add_sf(x, coords = NULL, crs = NULL, sfc_column = NULL)
```

Arguments

x	a gen_tibble object
coords	a vector of length 2, giving the names of the x and y columns in x (i.e. the coordinates, e.g. longitude and latitude). If coords is not provided, the geometry column must be provided.
crs	the coordinate reference system of the coordinates. If this is not set, it will be set to the default value of sf::st_crs(4326).
sfc_column	the name of an sf::sfc column to be used as the geometry

Value

a gen_tibble object with an additional geometry column (and thus belonging also to sf class).

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Add some coordinates
example_gt <- example_gt %>% mutate(
  longitude = c(0, 0, 2, 2, 0, 2, 2),
  latitude = c(51, 51, 49, 49, 51, 41, 41)
)

# Convert lat and long to sf:
example_gt <- gt_add_sf(x = example_gt, coords = c("longitude", "latitude"))

# Check class
class(example_gt)
```

gt_admixture *Run ADMIXTURE from R*

Description

This function runs ADMIXTURE, taking either a `gen_tibble` or a file as an input. This is a wrapper that runs ADMIXTURE from the command line, and reads the output into R. It can run multiple values of `k` and multiple repeats for each `k`.

Usage

```
gt_admixture(
  x,
  k,
  n_runs = 1,
  crossval = FALSE,
  n_cores = 1,
  seed = NULL,
  conda_env = "auto"
)
```

Arguments

`x`	a `gen_tibble` or a character giving the path of the input PLINK bed file
`k`	an integer giving the number of clusters
`n_runs`	the number of runs for each k value (defaults to 1)
`crossval`	boolean, should cross validation be used to assess the fit (defaults to FALSE)
`n_cores`	number of cores (defaults to 1)
`seed`	the seed for the random number generator (defaults to NULL)
`conda_env`	the name of the conda environment to use. "none" forces the use of a local copy, whilst any other string will direct the function to use a custom conda environment.

Details

This is a wrapper for the command line program ADMIXTURE. It can either use a binary present in the main environment, or use a copy installed in a conda environment.

Value

an object of class `gt_admix` consisting of a list with the following elements:

- `k` the number of clusters
- `Q` a matrix with the admixture proportions
- `P` a matrix with the allele frequencies

- log a log of the output generated by ADMIXTURE (usually printed on the screen when running from the command line)
- cv the cross validation error (if crossval is TRUE)
- loglik the log likelihood of the model
- id the id column of the input gen_tibble (if applicable)
- group the group column of the input gen_tibble (if applicable)

References

Alexander, D.H., Novembre, J. and Lange, K. (2009) 'Fast model-based estimation of ancestry in unrelated individuals', Genome Research, 19(9), pp. 1655–1664. Available at: https://doi.org/10.1101/gr.094052.109.

Examples

```
# run the example only if we have the package installed
## Not run:
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)
lobsters <- lobsters %>% group_by(population)
gt_admixture(lobsters,
  k = 2:3, seed = c(1, 2),
  n_runs = 2, crossval = TRUE
)

## End(Not run)
```

---

gt_admix_reorder_q *Reorder the q matrices based on the grouping variable*

Description

This function reorders the q matrices in a gt_admix object based on the grouping variable. This is useful before plotting when the samples from each group are not adjacent to each other in the q matrix.

Usage

```
gt_admix_reorder_q(x, group = NULL)
```

Arguments

x	a gt_admix object, possibly with a grouping variable
group	a character vector with the grouping variable (if there is no grouping variable info in x)

Value

a gt_admix object with the q matrices reordered

Examples

```
# run the example only if we have the package installed
if (requireNamespace("LEA", quietly = TRUE)) {
  example_gt <- load_example_gt("gen_tbl")

  # Create a gt_admix object
  admix_obj <- example_gt %>% gt_snmf(k = 1:3, project = "force")

  # The $id in admix_obj is the same as in the gen_tibble
  admix_obj$id

  # Reorder the q matrices based on the grouping variable
  admix_obj <- gt_admix_reorder_q(admix_obj,
    group = example_gt$population
  )

  # The $id in admix_obj is now reordered according to the population
  admix_obj$id
}
```

---

gt_as_genind *Convert a* gen_tibble *to a* genind *object from* adegenet

---

Description

This function converts a gen_tibble to a genind object from adegenet

Usage

```
gt_as_genind(x)
```

Arguments

x	a gen_tibble, with population coded as 'population'

Value

a genind object

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Convert to genind
gt_genind <- example_gt %>% gt_as_genind()

# Check object class
class(gt_genind)
```

`gt_as_genlight` *Convert a* `gen_tibble` *to a* `genlight` *object from* `adegenet`

Description

This function converts a `gen_tibble` to a `genlight` object from `adegenet`

Usage

```
gt_as_genlight(x)
```

Arguments

`x`	a `gen_tibble`, with population coded as 'population'

Value

a `genlight` object

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Convert to genlight
gt_genlight <- example_gt %>% gt_as_genlight()

# Check object class
class(gt_genlight)
```

`gt_as_geno_lea` *Convert a* `gentibble` *to a .geno file for sNMF from the LEA package*

Description

This function writes a .geno file from a `gen_tibble`. Unless a file path is given, a file with suffix .geno is written in the same location as the .rds and .bk files that underpin the `gen_tibble`.

Usage

```
gt_as_geno_lea(x, file = NULL)
```

Arguments

`x`	a `gen_tibble`
`file`	the .geno filename with a path, or NULL (the default) to use the location of the backing files.

Details

NOTE that we currently read all the data into memory to write the file, so this function is not suitable for very large datasets.

Value

the path of the .geno file

See Also

`LEA::geno()`

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Write a geno file
gt_as_geno_lea(example_gt, file = paste0(tempfile(), "_example.geno"))
```

gt_as_hierfstat *Convert a* gen_tibble *to a data.frame compatible with* hierfstat

Description

This function converts a gen_tibble to a data.frame formatted to be used by hierfstat functions.

Usage

```
gt_as_hierfstat(x)
```

Arguments

x	a gen_tibble, with population coded as 'population'

Value

a data.frame with a column 'pop' and further column representing the genotypes (with alleles recoded as 1 and 2)

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Convert to hierfstat format
gt_hierfstat <- example_gt %>% gt_as_hierfstat()

# Check object class
class(gt_hierfstat)
```

gt_as_plink *Export a* gen_tibble *object to PLINK bed format*

Description

This function exports all the information of a gen_tibble object into a PLINK bed, ped or raw file (and associated files, i.e. .bim and .fam for .bed; .fam for .ped).

Usage

```
gt_as_plink(
  x,
  file = NULL,
  type = c("bed", "ped", "raw"),
  overwrite = TRUE,
  chromosomes_as_int = FALSE
)
```

Arguments

x	a gen_tibble object
file	a character string giving the path to output file. If left to NULL, the output file will have the same path and prefix of the backingfile.
type	one of "bed", "ped" or "raw"
overwrite	boolean whether to overwrite the file.
chromosomes_as_int	boolean whether to use the integer representation of the chromosomes

Details

If the gen_tibble has been read in from vcf format, family.ID in the resulting plink files will be the same as sample.ID. If the gen_tibble has a grouping variable, this will be used as the family.ID in the resulting plink files. NOTE that writing to bed has been optimised for speed, but writing to ped or raw is slower, especially for large datasets.

Value

the path of the saved file

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Write a bed file
example_gt %>% gt_as_plink(type = "bed", file = paste0(tempfile(), "_plink"))

# Write a ped file
example_gt %>% gt_as_plink(type = "ped", file = paste0(tempfile(), "_plink"))

# Write a raw file
example_gt %>% gt_as_plink(type = "raw", file = paste0(tempfile(), "_plink"))
```

---

gt_as_vcf *Convert a* gen_tibble *to a VCF*

Description

This function write a VCF from a gen_tibble.

Usage

```
gt_as_vcf(x, file = NULL, chunk_size = NULL, overwrite = FALSE)
```

Arguments

x	a gen_tibble, with population coded as 'population'
file	the .vcf file name with a path, or NULL (the default) to use the location of the backing files.
chunk_size	the number of loci processed at a time. Automatically set if left to NULL
overwrite	logical, should the file be overwritten if it already exists?

Value

the path of the .vcf file

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Write a vcf file
example_gt %>% gt_as_vcf()
```

---

gt_cluster_pca *Run K-clustering on principal components*

Description

This function implements the clustering procedure used in Discriminant Analysis of Principal Components (DAPC, Jombart et al. 2010). This procedure consists in running successive K-means with an increasing number of clusters (k), after transforming data using a principal component analysis (PCA). For each model, several statistical measures of goodness of fit are computed, which allows to choose the optimal k using the function gt_cluster_pca_best_k(). See details for a description of how to select the optimal k and vignette("adegenet-dapc") for a tutorial.

Usage

```
gt_cluster_pca(
  x = NULL,
  n_pca = NULL,
  k_clusters = c(1, round(nrow(x$u)/10)),
  method = c("kmeans", "ward"),
  n_iter = 1e+05,
  n_start = 10,
  quiet = FALSE
)
```

Arguments

x	a gt_pca object returned by one of the gt_pca_* functions.
n_pca	number of principal components to be fed to the LDA.
k_clusters	number of clusters to explore, either a single value, or a vector of length 2 giving the minimum and maximum (e.g. 1:5). If left NULL, it will use 1 to the number of pca components divided by 10 (a reasonable guess).
method	either 'kmeans' or 'ward'
n_iter	number of iterations for kmeans (only used if method="kmeans")
n_start	number of starting points for kmeans (only used if method="kmeans")
quiet	boolean on whether to silence outputting information to the screen (defaults to FALSE)

Value

a gt_cluster_pca object, which is a subclass of gt_pca with an additional element 'cluster', a list with elements:

- 'method' the clustering method (either kmeans or ward)
- 'n_pca' number of principal components used for clustering
- 'k' the k values explored by the function
- 'WSS' within sum of squares for each k
- 'AIC' the AIC for each k
- 'BIC' the BIC for each k
- 'groups' a list, with each element giving the group assignments for a given k

References

Jombart T, Devillard S and Balloux F (2010) Discriminant analysis of principal components: a new method for the analysis of genetically structured populations. BMC Genetics 11:94. doi:10.1186/1471-2156-11-94

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

# Create PCA object
```

```
pca <- gt_pca_partialSVD(lobsters)

# Run clustering on the first 10 PCs
gt_cluster_pca(
  x = pca,
  n_pca = 10,
  k_clusters = c(1, 5),
  method = "kmeans",
  n_iter = 1e5,
  n_start = 10,
  quiet = FALSE
)

# Alternatively, use method "ward"
gt_cluster_pca(
  x = pca,
  n_pca = 10,
  k_clusters = c(1, 5),
  method = "ward",
  quiet = FALSE
)
```

`gt_cluster_pca_best_k` *Find the best number of clusters based on principal components*

Description

This function selects the best k value based on a chosen metric and criterion. It is equivalent to plotting the metric against the *k* values, and selecting the *k* that fulfills a given criterion (see details for an explanation of each criterion). This function simply adds an element 'best_k' to the `gt_cluster_pca` returned by `gt_cluster_pca()`. The choice can be over-ridden simply by assigning a different value to that element (e.g. for an object x and a desired *k* of 8, simply use `x$best_k <- 8`)

Usage

```
gt_cluster_pca_best_k(
  x,
  stat = c("BIC", "AIC", "WSS"),
  criterion = c("diffNgroup", "min", "goesup", "smoothNgoesup", "goodfit"),
  quiet = FALSE
)
```

Arguments

`x`	a `gt_cluster_pca` object obtained with `gt_cluster_pca()`
`stat`	a statistics, one of "BIC", "AIC" or "WSS"

criterion	one of "diffNgroup", "min", "goesup", "smoothNgoesup", "goodfit", see details for a discussion of each approach.
quiet	boolean on whether to silence outputting information to the screen (defaults to FALSE)

Details

The analysis of data simulated under various population genetics models (see reference) suggested an ad-hoc rule for the selection of the optimal number of clusters. First important result is that BIC seems more efficient than AIC and WSS to select the appropriate number of clusters (see example). The rule of thumb consists in increasing K until it no longer leads to an appreciable improvement of fit (i.e., to a decrease of BIC). In the most simple models (island models), BIC decreases until it reaches the optimal K, and then increases. In these cases, the best rule amounts to choosing the lowest K. In other models such as stepping stones, the decrease of BIC often continues after the optimal K, but is much less steep, so a change in slope can be taken as an indication of where the best *k* lies.

This function provides a programmatic way to select *k*. Note that it is highly recommended to look at the graph of BIC versus the numbers of clusters, to understand and validate the programmatic selection. The criteria available in this function are:

- "diffNgroup": differences between successive values of the summary statistics (by default, BIC) are split into two groups using a Ward's clustering method (see ?hclust), to differentiate sharp decrease from mild decreases or increases. The retained K is the one before the first group switch. This criterion appears to work well for island/hierarchical models, and decently for isolation by distance models, albeit with some instability. It can be confounded by an initial, very sharp decrease of the test statistics. IF UNSURE ABOUT THE CRITERION TO USE, USE THIS ONE.
- "min": the model with the minimum summary statistics (as specified by stat argument, BIC by default) is retained. Is likely to work for simple island model, using BIC. It is likely to fail in models relating to stepping stones, where the BIC always decreases (albeit by a small amount) as K increases. In general, this approach tends to over-estimate the number of clusters.
- "goesup": the selected model is the K after which increasing the number of clusters leads to increasing the summary statistics. Suffers from inaccuracy, since i) a steep decrease might follow a small 'bump' of increase of the statistics, and ii) increase might never happen, or happen after negligible decreases. Is likely to work only for clear-cut island models.
- "smoothNgoesup": a variant of "goesup", in which the summary statistics is first smoothed using a lowess approach. Is meant to be more accurate than "goesup" as it is less prone to stopping to small 'bumps' in the decrease of the statistics.
- "goodfit": another criterion seeking a good fit with a minimum number of clusters. This approach does not rely on differences between successive statistics, but on absolute fit. It selects the model with the smallest K so that the overall fit is above a given threshold.

Value

a 'gt_cluster_pca' object with an added element 'best_k'

References

Jombart T, Devillard S and Balloux F (2010) Discriminant analysis of principal components: a new method for the analysis of genetically structured populations. BMC Genetics 11:94. doi:10.1186/1471-2156-11-94

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

# Create PCA object
pca <- gt_pca_partialSVD(lobsters)

# Run clustering on the first 10 PCs
cluster_pca <- gt_cluster_pca(
  x = pca,
  n_pca = 10,
  k_clusters = c(1, 5),
  method = "kmeans",
  n_iter = 1e5,
  n_start = 10,
  quiet = FALSE
)

# Find best K through minimum BIC
cluster_pca <- gt_cluster_pca_best_k(cluster_pca,
  stat = "BIC",
  criterion = "min",
  quiet = FALSE
)
# Best K is stored in the object
cluster_pca$best_k
```

---

gt_dapc *Discriminant Analysis of Principal Components for gen_tibble*

Description

This function implements the Discriminant Analysis of Principal Components (DAPC, Jombart et al. 2010). This method describes the diversity between pre-defined groups. When groups are

unknown, use gt_cluster_pca() to infer genetic clusters. See 'details' section for a succinct description of the method, and the vignette in the package adegenet ("adegenet-dapc") for a tutorial.

Usage

```
gt_dapc(x, pop = NULL, n_pca = NULL, n_da = NULL, loadings_by_locus = TRUE)
```

Arguments

x	an object of class gt_pca, or its subclass gt_cluster_pca
pop	either a factor indicating the group membership of individuals; or an integer defining the desired *k* if x is a gt_cluster_pca; or NULL, if 'x' is a gt_cluster_pca and contain an element 'best_k', usually generated with gt_cluster_pca_best_k(), which will be used to select the clustering level.
n_pca	number of principal components to be used in the Discriminant Analysis. If NULL, k-1 will be used.
n_da	an integer indicating the number of axes retained in the Discriminant Analysis step.
loadings_by_locus	a logical indicating whether the loadings and contribution of each locus should be stored (TRUE, default) or not (FALSE). Such output can be useful, but can also create large matrices when there are a lot of loci and many dimensions.

Details

The Discriminant Analysis of Principal Components (DAPC) is designed to investigate the genetic structure of biological populations. This multivariate method consists in a two-steps procedure. First, genetic data are transformed (centred, possibly scaled) and submitted to a Principal Component Analysis (PCA). Second, principal components of PCA are submitted to a Linear Discriminant Analysis (LDA). A trivial matrix operation allows to express discriminant functions as linear combination of alleles, therefore allowing one to compute allele contributions. More details about the computation of DAPC are to be found in the indicated reference.

Results can be visualised with autoplot.gt_dapc(), see the help of that method for the available plots. There are also gt_dapc_tidiers for manipulating the results. For the moment, this function returns objects of class adegenet::dapc which are compatible with methods from adegenet; graphical methods for DAPC are documented in adegenet::scatter.dapc (see ?scatter.dapc). This is likely to change in the future, so make sure you do not rely on the objects remaining compatible.

This function aligns with the guidelines proposed by Thia (2023) for the standardized application of DAPC to genotype data. Our default settings are designed to follow these recommendations, so that the number of principal components (n_pca) defaults to the smaller of *k*-1 and the number of available principal components (where *k* is the number of populations or clusters), and the number of discriminant functions (n_da) is set to the minimum of *k*-1 and n_pca. The user can override these defaults by specifying the n_pca and n_da arguments, but caution is advised when adjusting n_pca to avoid potential overfitting. We recommend users consult these guidelines and consider their individual dataset to ensure best practices.

Note that there is no current method to predict scores for individuals not included in the original analysis. This is because we currently do not have a mechanism to store the pca information in the object, and that is needed for prediction.

Value

an object of class adegenet::dapc

References

Jombart T, Devillard S and Balloux F (2010) Discriminant analysis of principal components: a new method for the analysis of genetically structured populations. BMC Genetics 11:94. doi:10.1186/1471-2156-11-94 Thia, J. A. (2023). Guidelines for standardizing the application of discriminant analysis of principal components to genotype data. Molecular Ecology Resources, 23, 523–538. https://doi.org/10.1111/1755-0998.13706

See Also

gt_cluster_pca() gt_cluster_pca_best_k() adegenet::dapc()

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

# Create PCA object
pca <- gt_pca_partialSVD(lobsters)

# Run DAPC on the `gt_pca` object, providing `pop` as factor
populations <- as.factor(lobsters$population)
gt_dapc(pca, n_pca = 6, n_da = 2, pop = populations)

# Run clustering on the first 10 PCs
cluster_pca <- gt_cluster_pca(
  x = pca,
  n_pca = 10,
  k_clusters = c(1, 5),
  method = "kmeans",
  n_iter = 1e5,
  n_start = 10,
  quiet = FALSE
)

# Find best k
cluster_pca <- gt_cluster_pca_best_k(cluster_pca,
  stat = "BIC",
  criterion = "min"
```

```
)

# Run DAPC on the `gt_cluster_pca` object
gt_dapc(cluster_pca, n_pca = 10, n_da = 2)

#  should be stored (TRUE, default) or not (FALSE). This information is
#  required to predict group membership of new individuals using predict, but
#  makes the object slightly bigger.
```

`gt_extract_f2` *Compute and store blocked f2 statistics for ADMIXTOOLS 2*

Description

Compute and store blocked f2 statistics for ADMIXTOOLS 2

Usage

```
gt_extract_f2(
  .x,
  outdir = NULL,
  blgsize = 0.05,
  maxmem = 8000,
  maxmiss = 0,
  minmaf = 0,
  maxmaf = 0.5,
  minac2 = FALSE,
  outpop = NULL,
  outpop_scale = TRUE,
  transitions = TRUE,
  transversions = TRUE,
  overwrite = FALSE,
  adjust_pseudohaploid = NULL,
  fst = TRUE,
  afprod = TRUE,
  poly_only = c("f2"),
  apply_corr = TRUE,
  n_cores = 1,
  quiet = FALSE
)
```

Arguments

`.x`	a `gen_tibble`
`outdir`	Directory where data will be stored.
`blgsize`	SNP block size in Morgan. Default is 0.05 (5 cM). If `blgsize` is 100 or greater, it will be interpreted as base pair distance rather than centimorgan distance.

maxmem — Maximum amount of memory to be used. If the required amount of memory exceeds maxmem, allele frequency data will be split into blocks, and the computation will be performed separately on each block pair. This doesn't put a precise cap on the amount of memory used (it used to at some point). Set this parameter to lower values if you run out of memory while running this function. Set it to higher values if this function is too slow and you have lots of memory.

maxmiss — Discard SNPs which are missing in a fraction of populations higher than maxmiss

minmaf — Discard SNPs with minor allele frequency less than minmaf

maxmaf — Discard SNPs with minor allele frequency greater than maxmaf

minac2 — Discard SNPs with allele count lower than 2 in any population (default FALSE). This option should be set to TRUE when computing f3-statistics where one population consists mostly of pseudohaploid samples. Otherwise heterozygosity estimates and thus f3-estimates can be biased. minac2 == 2 will discard SNPs with allele count lower than 2 in any non-singleton population (this option is experimental and is based on the hypothesis that using SNPs with allele count lower than 2 only leads to biases in non-singleton populations). Note that while the minac2 option discards SNPs with allele count lower than 2 in any population, the qp3pop function will only discard SNPs with allele count lower than 2 in the first (target) population (when the first argument is the prefix of a genotype file; i.e. it is applied directly to a genotype file, not via precomputing f2 from a gen_tibble).

outpop — Keep only SNPs which are heterozygous in this population

outpop_scale — Scale f2-statistics by the inverse outpop heterozygosity (1/(p*(1-p))). Providing outpop and setting outpop_scale to TRUE will give the same results as the original *qpGraph* when the outpop parameter has been set, but it has the disadvantage of treating one population different from the others. This may limit the use of these f2-statistics for other models.

transitions — Set this to FALSE to exclude transition SNPs

transversions — Set this to FALSE to exclude transversion SNPs

overwrite — Overwrite existing files in outdir

adjust_pseudohaploid — Genotypes of pseudohaploid samples are usually coded as 0 or 2, even though only one allele is observed. adjust_pseudohaploid ensures that the observed allele count increases only by 1 for each pseudohaploid sample. If TRUE (default), samples that don't have any genotypes coded as 1 among the first 1000 SNPs are automatically identified as pseudohaploid. This leads to slightly more accurate estimates of f-statistics. Setting this parameter to FALSE treats all samples as diploid and is equivalent to the *ADMIXTOOLS* inbreed: NO option. Setting adjust_pseudohaploid to an integer n will check the first n SNPs instead of the first 1000 SNPs. NOW DEPRECATED, set the ploidy of the gen_tibble with gt_pseudohaploid().

fst — Write files with pairwise FST for every population pair. Setting this to FALSE can make extract_f2 faster and will require less memory.

afprod — Write files with allele frequency products for every population pair. Setting this to FALSE can make extract_f2 faster and will require less memory.

poly_only	Specify whether SNPs with identical allele frequencies in every population should be discarded (poly_only = TRUE), or whether they should be used (poly_only = FALSE). By default (poly_only = c("f2")), these SNPs will be used to compute FST and allele frequency products, but not to compute f2 (this is the default option in the original ADMIXTOOLS).
apply_corr	Apply small-sample-size correction when computing f2-statistics (default TRUE)
n_cores	Parallelize computation across n_cores cores.
quiet	Suppress printing of progress updates

Value

SNP metadata (invisibly)

References

Maier R, Patterson N (2024). admixtools: Inferring demographic history from genetic data. R package version 2.0.4, https://github.com/uqrmaie1/admixtools.

This function prepares data for various *ADMIXTOOLS 2* functions from the package *ADMIXTOOLS 2*. It takes a gen_tibble, computes allele frequencies and blocked f2-statistics, and writes the results to outdir. It is equivalent to admixtools::extract_f2().

Examples

```
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)
lobsters <- lobsters %>% group_by(population)
f2_path <- tempfile()
gt_extract_f2(lobsters, outdir = f2_path, quiet = TRUE)
admixtools::f2_from_precomp(f2_path, verbose = FALSE)
```

---

gt_from_genlight *Convert a* genlight *object from adegenet to a* gen_tibble

---

Description

This function converts a genlight object from the adegenet package to a gen_tibble object

Usage

```
gt_from_genlight(x, backingfile = NULL, ...)
```

Arguments

x	A genlight object
backingfile	the path, including the file name without extension, for backing files used to store the data (they will be given a .bk and .rds automatically). If NULL (default), backing files are placed in the temporary directory.
...	Additional arguments passed to gen_tibble().

Details

- Currently supports diploid genlight objects only (all values in @ploidy must be 2).
- Requires non-missing slots: loc.names, n.loc, loc.all, chromosome, position, ploidy, ind.names. The pop slot is optional; if absent, the returned gen_tibble will omit the population column.

Value

A gen_tibble object

Examples

```
# Create a simple genlight object
x <- new("genlight",
  list(
    indiv1 = c(1, 1, 0, 1, 1, 0),
    indiv2 = c(2, 1, 1, 0, 0, 0)
  ),
  ploidy = c(2, 2),
  loc.names = paste0("locus", 1:6),
  chromosome = c("chr1", "chr1", "chr2", "chr2", "chr3", "chr3"),
  position = c(100, 200, 150, 250, 300, 400),
  loc.all = c("A/T", "C/G", "G/C", "A/T", "T/C", "G/A"),
  pop = c("pop1", "pop2")
)


file <- paste0(tempfile(), "gt_from_genlight")
# Convert to gen_tibble
new_gt <- gt_from_genlight(x, backingfile = file)
```

---

gt_get_file_names *Get the names of files storing the genotypes of a* gen_tibble

---

Description

A function to return the names of the files used to store data in a gen_tibble. Specifically, this returns the .rds file storing the big

Usage

```
gt_get_file_names(x)
```

Arguments

x	a gen_tibble

Value

a character vector with the names and paths of the two files

Examples

```
example_gt <- load_example_gt("gen_tbl")

# To retrieve the names of and paths to the .bk and .rds files use:
gt_get_file_names(example_gt)
```

---

gt_has_imputed *Checks if a* gen_tibble *has been imputed*

Description

This function checks if a dataset has been imputed. Note that having imputation does not mean that the imputed values are used.

Usage

```
gt_has_imputed(x)
```

Arguments

x	a gen_tibble

Value

boolean TRUE or FALSE depending on whether the dataset has been imputed

Examples

```
example_gt <- load_example_gt("gen_tbl")

# The initial gen_tibble contains no imputed values
example_gt %>% gt_has_imputed()

# Now impute the gen_tibble
example_gt <- example_gt %>% gt_impute_simple()
```

```
# And we can check it has been imputed
example_gt %>% gt_has_imputed()
```

`gt_impute_simple` *Simple imputation based on allele frequencies*

Description

This function provides a very simple imputation algorithm for `gen_tibble` objects by using the mode, mean or sampling from the allele frequencies. Each locus is imputed independently (and thus linkage information is ignored).

Usage

```
gt_impute_simple(x, method = c("mode", "mean0", "random"), n_cores = 1)
```

Arguments

`x`	a gen_tibble with missing data
`method`	one of • 'mode': the most frequent genotype • 'mean0': the mean rounded to the nearest integer • 'random': randomly sample a genotype based on the observed allele frequencies
`n_cores`	the number of cores to be used

Details

This function is a wrapper around `bigsnpr::snp_fastImputeSimple()`.

Value

a gen_tibble with imputed genotypes

See Also

`bigsnpr::snp_fastImputeSimple()` which this function wraps.

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Impute the gen_tibble
example_gt <- example_gt %>% gt_impute_simple()

# And we can check it has been imputed
example_gt %>% gt_has_imputed()
```

`gt_impute_xgboost` *Imputation based XGBoost*

Description

This function provides a simple imputation algorithm for `gen_tibble` objects based on local XGBoost models.

Usage

```
gt_impute_xgboost(
  x,
  alpha = 1e-04,
  size = 200,
  p_train = 0.8,
  n_cor = nrow(x),
  seed = NA,
  n_cores = 1,
  append_error = TRUE
)
```

Arguments

`x`	a gen_tibble with missing data
`alpha`	Type-I error for testing correlations. Default is `1e-4`.
`size`	Number of neighbour SNPs to be possibly included in the model imputing this particular SNP. Default is `200`.
`p_train`	Proportion of non missing genotypes that are used for training the imputation model while the rest is used to assess the accuracy of this imputation model. Default is `0.8`.
`n_cor`	Number of rows that are used to estimate correlations. Default uses them all.
`seed`	An integer, for reproducibility. Default doesn't use seeds.
`n_cores`	the number of cores to be used
`append_error`	boolean, should the xgboost error estimates be appended as an attribute to the genotype column of the gen_tibble. If TRUE (the default), a matrix of two rows (the number of missing values, and the error estimate) and as many columns as the number of loci will be appended to the gen_tibble. attr(missing_gt$genotypes, "imputed_errors")

Details

This function is a wrapper around `bigsnpr::snp_fastImpute()`. The error rates from the xgboost, if appended, can be retrieved with `attr(x$genotypes, "imputed_errors")` where `x` is the `gen_tibble`.

Value

a gen_tibble with imputed genotypes

See Also

`bigsnpr::snp_fastImpute()` which this function wraps.

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Impute the gen_tibble
example_gt <- example_gt %>% gt_impute_xgboost()

# And we can check it has been imputed
example_gt %>% gt_has_imputed()
```

---

gt_load *Load a gen_tibble*

Description

Load a `gen_tibble` previously saved with `gt_save()`. If the *.rds* and *.bk* files have not been moved, they should be found automatically. If they were moved, use `reattach_to` to point to the *.rds* file (the *.bk* file needs to be in the same directory as the *.rds* file).

Usage

```
gt_load(file = NULL, reattach_to = NULL)
```

Arguments

`file`	the file name, including the full path. If it does not end with *.gt*, the extension will be added.
`reattach_to`	the file name, including the full path, of the *.rds* file if it was moved. It assumes that the *.bk* file is found in the same path. You should be able to leave this to NULL unless you have moved the files.

Value

a `gen_tibble`

See Also

`gt_save()`

Examples

```
example_gt <- load_example_gt("gen_tbl")

# remove some individuals
example_gt_filtered <- example_gt %>% filter(id != "a")

# save the filtered gen_tibble object
backing_files <- gt_save(example_gt_filtered,
  file_name = paste0(tempfile(), "_example_filtered")
)

# backing_files[1] contains the name of the saved .gt file
backing_files[1]

# To load the saved gen_tibble object, use the path to the saved .gt file
reloaded_gt <- gt_load(backing_files[1])

# And we have loaded the gt without individual "a"
reloaded_gt
```

`gt_order_loci` *Order the loci table of a gen_tibble*

Description

This function reorders the loci table so that positions within a chromosome are sequential. It also re-saves the genotypes into a new file backed matrix with the new order, so that it can be used by functions such as `loci_ld_clump()` and `gt_pca_autoSVD()`. If the loci table is already ordered, the original `gen_tibble` is returned. This function will update the backingfiles of the `gen_tibble` and return the `gen_tibble` object, use `<-` as per the example provided to ensure that the names of the newly updated backingfiles are stored in the `gen_tibble` object.

Usage

```
gt_order_loci(
  .x,
  use_current_table = FALSE,
  ignore_genetic_dist = TRUE,
  quiet = FALSE,
  ...
)
```

Arguments

`.x`	a `gen_tibble`
`use_current_table`	boolean, if FALSE (the default), the table will be reordered; if TRUE, then the current loci table, which might have been reordered manually, will be used, but only if the positions within each chromosome are sequential

ignore_genetic_dist
: boolean to ignore the genetic distance when checking. Note that, if genetic_dist are being ignored and they are not sorted, the function will set them to zero to avoid problems with other software.

quiet
: boolean to suppress information about the files

...
: other arguments

Value

A gen_tibble

Examples

```
example_gt <- load_example_gt("gen_tbl") %>% select_loci(c(1, 5, 2, 6, 4, 3))

# Loci are in the wrong order
show_loci(example_gt)

# Reorder the loci, ignoring genetic distance
example_gt_ordered <- gt_order_loci(example_gt, ignore_genetic_dist = TRUE)

# Loci are now in the correct order
show_loci(example_gt_ordered)
```

---

gt_pca *Principal Component Analysis for* gen_tibble *objects*

Description

There are a number of PCA methods available for gen_tibble objects. They are mostly designed to work on very large datasets, so they only compute a limited number of components. For smaller datasets, gt_partialSVD allows the use of partial (truncated) SVD to fit the PCA; this method is suitable when the number of individuals is much smaller than the number of loci. For larger dataset, gt_randomSVD is more appropriate. Finally, there is a method specifically designed for dealing with LD in large datasets, gt_autoSVD. Whilst this is arguably the best option, it is somewhat data hungry, and so only suitable for very large datasets (hundreds of individuals with several hundred thousands markers, or larger).

Details

NOTE: using gt_pca_autoSVD with a small dataset will likely cause an error, see man page for details.

NOTE: monomorphic markers must be removed before PCA is computed. The error message 'Error: some variables have zero scaling; remove them before attempting to scale.' indicates that monomorphic markers are present.

gt_pcadapt *pcadapt analysis on a* gen_tibble *object*

Description

pcadapt is an algorithm that detects genetic markers under selection. It is based on the principal component analysis (PCA) of the genotypes of the individuals. The method is described in Luu et al. (2017). See the R package pcadapt, which provides extensive documentation and examples.

Usage

```
gt_pcadapt(x, pca, k, n_cores = 1)
```

Arguments

x	A gen_tibble object.
pca	a gt_pca object, as returned by gt_pca_partialSVD() or gt_pca_randomSVD().
k	Number of principal components to use in the analysis.
n_cores	Number of cores to use.

Details

Internally, this function uses the snp_pcadapt function from the bigsnpr package.

Value

An object of subclass gt_pcadapt, a subclass of mhtest.

References

Luu, K., Bazin, E., Blum, M. G. B., & François, O. (2017). pcadapt: an R package for genome scans for selection based on principal component analysis. Molecular Ecology Resources, 17(1), 67–77.

See Also

bigsnpr::snp_pcadapt() which this function wraps.

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)
```

```
# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

# Create PCA object
pca <- gt_pca_partialSVD(lobsters)

# Create a gt_pcadapt object
gt_pcadapt(lobsters, pca, k = 2)
```

---

gt_pca_autoSVD *PCA controlling for LD for* gen_tibble *objects*

Description

This function performs Principal Component Analysis on a gen_tibble, using a fast truncated SVD with initial pruning and then iterative removal of long-range LD regions. This function is a wrapper for bigsnpr::snp_autoSVD()

Usage

```
gt_pca_autoSVD(
  x,
  k = 10,
  fun_scaling = bigsnpr::snp_scaleBinom(),
  thr_r2 = 0.2,
  use_positions = TRUE,
  size = 100/thr_r2,
  roll_size = 50,
  int_min_size = 20,
  alpha_tukey = 0.05,
  min_mac = 10,
  max_iter = 5,
  n_cores = 1,
  verbose = TRUE,
  total_var = TRUE
)
```

Arguments

x	a gen_tbl object
k	Number of singular vectors/values to compute. Default is 10. **This algorithm should be used to compute a few singular vectors/values.**
fun_scaling	Usually this can be left unset, as it defaults to bigsnpr::snp_scaleBinom(), which is the appropriate function for biallelic SNPs. Alternatively it is possible to use custom function (see bigsnpr::snp_autoSVD() for details.

thr_r2	Threshold over the squared correlation between two SNPs. Default is 0.2. Use NA if you want to skip the clumping step. size
use_positions	a boolean on whether the position is used to define size, or whether the size should be in number of SNPs. Default is TRUE
size	For one SNP, window size around this SNP to compute correlations. Default is 100 / thr_r2 for clumping (0.2 -> 500; 0.1 -> 1000; 0.5 -> 200). If not providing infos.pos (NULL, the default), this is a window in number of SNPs, otherwise it is a window in kb (genetic distance). I recommend that you provide the positions if available.
roll_size	Radius of rolling windows to smooth log-p-values. Default is 50.
int_min_size	Minimum number of consecutive outlier SNPs in order to be reported as long-range LD region. Default is 20.
alpha_tukey	Default is 0.05. The type-I error rate in outlier detection (that is further corrected for multiple testing).
min_mac	Minimum minor allele count (MAC) for variants to be included. Default is 10.
max_iter	Maximum number of iterations of outlier detection. Default is 5.
n_cores	Number of cores used. Default doesn't use parallelism. You may use bigstatsr::nb_cores().
verbose	Output some information on the iterations? Default is TRUE.
total_var	a boolean indicating whether to compute the total variance of the matrix. Default is TRUE. Using FALSE will speed up computation, but the total variance will not be stored in the output (and thus it will not be possible to assign a proportion of variance explained to the components).

Details

Using gt_pca_autoSVD requires a reasonably large dataset, as the function iteratively removes regions of long range LD. If you encounter: 'Error in rollmean(): Parameter 'size' is too large.', roll_size exceeds the number of variants on at least one of your chromosomes. Try reducing 'roll_size' to avoid this error.

Note: rather than accessing these elements directly, it is better to use tidy and augment. See gt_pca_tidiers.

Value

a gt_pca object, which is a subclass of bigSVD; this is an S3 list with elements: A named list (an S3 class "big_SVD") of

- d, the eigenvalues (singular values, i.e. as variances),
- u, the scores for each sample on each component (the left singular vectors)
- v, the loadings (the right singular vectors)
- center, the centering vector,
- scale, the scaling vector,
- method, a string defining the method (in this case 'autoSVD'),
- call, the call that generated the object.
- loci, the loci used after long range LD removal.

See Also

bigsnpr::snp_autoSVD() which this function wraps.

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

show_loci(lobsters)$chromosome <- "1"

# Create PCA object, including total variance
gt_pca_autoSVD(lobsters,
  k = 10,
  roll_size = 20,
  total_var = TRUE
)
# Change number of components and exclude total variance
gt_pca_autoSVD(lobsters,
  k = 5,
  roll_size = 20,
  total_var = FALSE
)
```

---

gt_pca_partialSVD *PCA for* gen_tibble *objects by partial SVD*

Description

This function performs Principal Component Analysis on a gen_tibble, by partial SVD through the eigen decomposition of the covariance. It works well if the number of individuals is much smaller than the number of loci; otherwise, gt_pca_randomSVD() is a better option. This function is a wrapper for bigstatsr::big_SVD().

Usage

```
gt_pca_partialSVD(
  x,
  k = 10,
```

```
  fun_scaling = bigsnpr::snp_scaleBinom(),
  total_var = TRUE
)
```

Arguments

x	a gen_tbl object
k	Number of singular vectors/values to compute. Default is 10. **This algorithm should be used to compute a few singular vectors/values.**
fun_scaling	Usually this can be left unset, as it defaults to bigsnpr::snp_scaleBinom(), which is the appropriate function for biallelic SNPs. Alternatively it is possible to use custom function (see bigsnpr::snp_autoSVD() for details.
total_var	a boolean indicating whether to compute the total variance of the matrix. Default is TRUE. Using FALSE will speed up computation, but the total variance will not be stored in the output (and thus it will not be possible to assign a proportion of variance explained to the components).

Value

a gt_pca object, which is a subclass of bigSVD; this is an S3 list with elements: A named list (an S3 class "big_SVD") of

- d, the eigenvalues (singular values, i.e. as variances),
- u, the scores for each sample on each component (the left singular vectors)
- v, the loadings (the right singular vectors)
- center, the centering vector,
- scale, the scaling vector,
- method, a string defining the method (in this case 'partialSVD'),
- call, the call that generated the object.
- square_frobenius, used to compute the proportion of variance explained by the components (optional)

  Note: rather than accessing these elements directly, it is better to use tidy and augment. See gt_pca_tidiers.

See Also

bigstatsr::big_SVD() which this function wraps.

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)
```

```
# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

# Create PCA object, including total variance
gt_pca_partialSVD(lobsters,
  k = 10,
  total_var = TRUE
)
# Change number of components and exclude total variance
gt_pca_partialSVD(lobsters,
  k = 5,
  total_var = FALSE
)
```

`gt_pca_randomSVD` *PCA for* `gen_tibble` *objects by randomized partial SVD*

Description

This function performs Principal Component Analysis on a `gen_tibble`, by randomised partial SVD based on the algorithm in RSpectra (by Yixuan Qiu and Jiali Mei).
This algorithm is linear in time in all dimensions and is very memory-efficient. Thus, it can be used on very large big.matrices. This function is a wrapper for `bigstatsr::big_randomSVD()`

Usage

```
gt_pca_randomSVD(
  x,
  k = 10,
  fun_scaling = bigsnpr::snp_scaleBinom(),
  tol = 1e-04,
  verbose = FALSE,
  n_cores = 1,
  fun_prod = bigstatsr::big_prodVec,
  fun_cprod = bigstatsr::big_cprodVec,
  total_var = TRUE
)
```

Arguments

`x`	a `gen_tibble` object
`k`	Number of singular vectors/values to compute. Default is `10`. **This algorithm should be used to compute a few singular vectors/values.**
`fun_scaling`	Usually this can be left unset, as it defaults to `bigsnpr::snp_scaleBinom()`, which is the appropriate function for biallelic SNPs. Alternatively it is possible to use custom function (see `bigsnpr::snp_autoSVD()` for details.

tol	Precision parameter of svds. Default is 1e-4.
verbose	Should some progress be printed? Default is FALSE.
n_cores	Number of cores used.
fun_prod	Function that takes 6 arguments (in this order): • a matrix-like object X, • a vector x, • a vector of row indices ind.row of X, • a vector of column indices ind.col of X, • a vector of column centers (corresponding to ind.col), • a vector of column scales (corresponding to ind.col), and compute the product of X (subsetted and scaled) with x.
fun_cprod	Same as fun.prod, but for the *transpose* of X.
total_var	a boolean indicating whether to compute the total variance of the matrix. Default is TRUE. Using FALSE will speed up computation, but the total variance will not be stored in the output (and thus it will not be possible to assign a proportion of variance explained to the components).

Value

a gt_pca object, which is a subclass of bigSVD; this is an S3 list with elements: A named list (an S3 class "big_SVD") of

- d, the eigenvalues (singular values, i.e. as variances),
- u, the scores for each sample on each component (the left singular vectors)
- v, the loadings (the right singular vectors)
- center, the centering vector,
- scale, the scaling vector,
- method, a string defining the method (in this case 'randomSVD'),
- call, the call that generated the object.

Note: rather than accessing these elements directly, it is better to use tidy and augment. See gt_pca_tidiers.

See Also

bigstatsr::big_randomSVD() which this function wraps.

Examples

```
vcf_path <-
  system.file("extdata", "anolis",
    "punctatus_t70_s10_n46_filtered.recode.vcf.gz",
    package = "tidypopgen"
  )
anole_gt <-
```

```
  gen_tibble(vcf_path, quiet = TRUE, backingfile = tempfile("anolis_"))

# Remove monomorphic loci and impute
anole_gt <- anole_gt %>% select_loci_if(loci_maf(genotypes) > 0)
anole_gt <- gt_impute_simple(anole_gt, method = "mode")

# Create PCA object, including total variance
gt_pca_randomSVD(anole_gt, k = 10, total_var = TRUE)
```

`gt_pseudohaploid` *Set the ploidy of a* `gen_tibble` *to include pseudohaploids*

Description

If a `gen_tibble` includes pseudohaploid data, its ploidy is set to -2 to indicate that some individuals are coded as pseudohaploids. The ploidy of the individuals is updated, with pseudohaploids set to 1 and diploids set to 2. However, the dosages are not changed, meaning that pseudohaploids are still coded as 0 or 2. If the `gen_tibble` is already set to pseudohaploid, running gt_pseudohaploid will update the ploidy values again, if pseudohaploid individuals have been removed then ploidy is reset to 2.

Usage

```
gt_pseudohaploid(x, test_n_loci = 10000)
```

Arguments

`x`	a `gen_tibble` object
`test_n_loci`	the number of loci to test to determine if an individual is pseudohaploid. If there are no heterozygotes in the first `test_n_loci` loci, the individual is considered a pseudohaploid. If `NULL`, all loci are tested.

Value

a `gen_tibble` object with the ploidy set to -2 and the individual ploidy values updated to 1 or 2.

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Detect pseudohaploids and set ploidy for the whole gen_tibble
example_gt <- example_gt %>% gt_pseudohaploid(test_n_loci = 3)

# Ploidy is now set to -2
show_ploidy(example_gt)

# Individual ploidy now varies between 1 (pseudohaploid) and 2 (diploid)
indiv_ploidy(example_gt)
```

gt_save *Save a gen_tibble*

Description

Save the tibble (and update the backing files). The gen_tibble object is saved to a file with extension *.gt*, together with update its *.rds* and *.bk* files. Note that multiple *.gt* files can be linked to the same *.rds* and *.bk* files; generally, this occurs when we create multiple subsets of the data. The *.gt* file then stores the information on what subset of the full dataset we are interested in, whilst the *.rds* and *.bk* file store the full dataset. To reload a gen_tibble, you can pass the name of the *.gt* file with gt_load().

Usage

```
gt_save(x, file_name = NULL, quiet = FALSE)
```

Arguments

x	a gen_tibble
file_name	the file name, including the full path. If it does not end with *.gt*, the extension will be added.
quiet	boolean to suppress information about the files

Value

the file name and path of the *.gt* file, together with the *.rds* and *.bk* files

See Also

gt_load()

Examples

```
example_gt <- load_example_gt("gen_tbl")

# remove some individuals
example_gt <- example_gt %>% filter(id != "a")

# save filtered gen_tibble object
gt_save(example_gt, file_name = paste0(tempfile(), "_example_filtered"))
```

gt_set_imputed *Sets a* gen_tibble *to use imputed data*

Description

This function sets or unsets the use of imputed data. For some analysis, such as PCA, that does not allow for missing data, we have to use imputation, but for other analysis it might be preferable to allow for missing data.

Usage

```
gt_set_imputed(x, set = NULL)
```

Arguments

x	a gen_tibble
set	a boolean defining whether imputed data should be used

Value

the gen_tibble, invisibly

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Impute the gen_tibble
example_gt <- example_gt %>% gt_impute_simple()

# Check whether the gen_tibble uses imputed values
example_gt %>% gt_uses_imputed()

# Set the gen_tibble to use imputed values
example_gt %>% gt_set_imputed(TRUE)

# And check that the gen_tibble uses imputed values again
example_gt %>% gt_uses_imputed()
```

gt_snmf *Run SNMF from R in tidypopgen*

Description

Run SNMF from R in tidypopgen

Usage

```
gt_snmf(
  x,
  k,
  project = "continue",
  n_runs = 1,
  alpha,
  tolerance = 1e-05,
  entropy = FALSE,
  percentage = 0.05,
  I,
  iterations = 200,
  ploidy = 2,
  seed = -1
)
```

Arguments

x	a gen_tibble or a character giving the path to the input geno file
k	an integer giving the number of clusters
project	one of "continue", "new", and "force": "continue" stores files in the current project, "new" creates a new project, and "force" stores results in the current project even if the .geno input file has been altered,
n_runs	the number of runs for each k value (defaults to 1)
alpha	numeric snmf regularization parameter. See LEA::snmf for details
tolerance	numeric value of tolerance (default 0.00001)
entropy	boolean indicating whether to estimate cross-entropy
percentage	numeric value indicating percentage of masked genotypes, ranging between 0 and 1, to be used when entropy = TRUE
I	number of SNPs for initialising the snmf algorithm
iterations	numeric integer for maximum iterations (default 200)
ploidy	the ploidy of the input data (defaults to 2)
seed	the seed for the random number generator

Details

This is a wrapper for `LEA::snmf()`.

Value

an object of class `gt_admix` consisting of a list with the following elements:

- `k` the number of clusters
- `Q` a matrix with the admixture proportions
- `P` a matrix with the allele frequencies

- log a log of the output generated by ADMIXTURE (usually printed on the screen when running from the command line)
- cv the masked cross-entropy (if entropy is TRUE)
- loglik the log likelihood of the model
- id the id column of the input gen_tibble (if applicable)
- group the group column of the input gen_tibble (if applicable)

See Also

LEA::snmf()

Examples

```
# run the example only if we have the package installed
example_gt <- load_example_gt("gen_tbl")

# To run SNMF on a gen_tibble:
example_gt %>% gt_snmf(
  k = 1:3, project = "force", entropy = TRUE,
  percentage = 0.5, n_runs = 1, seed = 1, alpha = 100
)
```

---

gt_update_backingfile *Update the backing matrix*

Description

This functions forces a re-write of the file backing matrix to match the gen_tibble. Individuals and loci are subsetted and reordered according to the current state of the gen_tibble. A .gt, .bk and .rds file will be created.

Usage

```
gt_update_backingfile(
  .x,
  backingfile = NULL,
  chunk_size = NULL,
  rm_unsorted_dist = TRUE,
  quiet = FALSE
)
```

Arguments

.x	a gen_tibble object
backingfile	the path, including the file name without extension, for backing files used to store the data (they will be given a .bk and .rds automatically). If left to NULL (the default), the file name will be based on the name f the current backing file.
chunk_size	the number of loci to process at once
rm_unsorted_dist	boolean to set genetic_dist to zero (i.e. remove it) if it is unsorted within the chromosomes.
quiet	boolean to suppress information about the files

Details

This function does not check whether the positions of your genetic loci are sorted. To check this, and update the file backing matrix, use gt_order_loci(). Tests for this function are in test_gt_order_loci.R

Value

a gen_tibble with updated .gt, .bk, and .rds files (i.e. a new File Backed Matrix)

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Here, new backingfiles are created, but without using `<-` the object
# loaded in the R session is not updated
gt_update_backingfile(example_gt)

# Make sure to use `<-` to update the file names associated with the
# gen_tibble object loaded in the R session
example_gt <- example_gt %>% gt_update_backingfile()
```

---

gt_uses_imputed *Checks if a* gen_tibble *uses imputed data*

Description

This function checks if a dataset uses imputed data. Note that it is possible to have a dataset that has been imputed but it is currently not using imputation.

Usage

```
gt_uses_imputed(x)
```

Arguments

x	a gen_tibble

Value

boolean TRUE or FALSE depending on whether the dataset is using the imputed values

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Impute the gen_tibble
example_gt <- example_gt %>% gt_impute_simple()

# Check whether the gen_tibble uses imputed values
example_gt %>% gt_uses_imputed()
```

---

indiv_het_obs *Estimate individual observed heterozygosity*

Description

Estimate observed heterozygosity (H_obs) for each individual (i.e. the frequency of loci that are heterozygous in an individual).

Usage

```
indiv_het_obs(.x, as_counts = FALSE, ...)

## S3 method for class 'tbl_df'
indiv_het_obs(.x, as_counts = FALSE, ...)

## S3 method for class 'vctrs_bigSNP'
indiv_het_obs(.x, as_counts = FALSE, ...)
```

Arguments

.x	a vector of class vctrs_bigSNP (usually the genotype column of a gen_tibble object), or a gen_tibble.
as_counts	logical, if TRUE, return a matrix with two columns: the number of heterozygotes and the number of missing values for each individual. These quantities can be useful to compute more complex quantities.
...	currently unused.

Value

either:

- a vector of heterozygosities, one per individuals in the gen_tibble
- a matrix with two columns, where the first is the number of heterozygous loci for each individual and the second is the number of missing values for each individual

Examples

```
example_gt <- load_example_gt("gen_tbl")

example_gt %>% indiv_het_obs()

# For observed heterozygosity as counts:
example_gt %>% indiv_het_obs(as_counts = TRUE)
```

---

indiv_inbreeding *Individual inbreeding coefficient*

Description

This function calculates the inbreeding coefficient for each individual based on the beta estimate from Weir and Goudet (2017).

Usage

```
indiv_inbreeding(.x, method = c("WG17"), allele_sharing_mat = NULL, ...)

## S3 method for class 'tbl_df'
indiv_inbreeding(.x, method = c("WG17"), allele_sharing_mat = NULL, ...)

## S3 method for class 'vctrs_bigSNP'
indiv_inbreeding(.x, method = c("WG17"), allele_sharing_mat = NULL, ...)

## S3 method for class 'grouped_df'
indiv_inbreeding(.x, method = c("WG17"), allele_sharing_mat = NULL, ...)
```

Arguments

.x	a vector of class vctrs_bigSNP (usually the genotype column of a gen_tibble object), or a gen_tibble.
method	currently only "WG17" (for Weir and Goudet 2017).
allele_sharing_mat	optional and only relevant for "WG17", the matrix of Allele Sharing returned by pairwise_allele_sharing() with as_matrix=TRUE. As a number of statistics can be derived from the Allele Sharing matrix, it is sometimes more efficient to pre-compute this matrix. It is not possible to use this with grouped tibbles.
...	currently unused.

Value

a numeric vector of inbreeding coefficients.

References

Weir, BS and Goudet J (2017) A Unified Characterization of Population Structure and Relatedness. Genetics (2017) 206:2085

Examples

```
example_gt <- load_example_gt("gen_tbl")

example_gt %>% indiv_inbreeding(method = "WG17")
```

---

indiv_missingness *Estimate individual missingness*

Description

Estimate missingness for each individual (i.e. the frequency of missing genotypes in an individual).

Usage

```
indiv_missingness(.x, as_counts, block_size, ...)

## S3 method for class 'tbl_df'
indiv_missingness(
  .x,
  as_counts = FALSE,
  block_size = bigstatsr::block_size(nrow(.x), 1),
  ...
)

## S3 method for class 'vctrs_bigSNP'
indiv_missingness(
  .x,
  as_counts = FALSE,
  block_size = bigstatsr::block_size(length(.x), 1),
  ...
)
```

Arguments

.x	a vector of class vctrs_bigSNP (usually the genotype column of a gen_tibble object), or a gen_tibble.

as_counts	boolean defining whether the count of NAs (rather than the rate) should be returned. It defaults to FALSE (i.e. rates are returned by default).
block_size	maximum number of loci read at once.
...	currently unused.

Value

a vector of missingness, one per individuals in the gen_tibble

Examples

```
example_gt <- load_example_gt("gen_tbl")

example_gt %>% indiv_missingness()

# For missingness as counts:
example_gt %>% indiv_missingness(as_counts = TRUE)
```

---

indiv_ploidy *Return individual ploidy*

Description

Returns the ploidy for each individual.

Usage

```
indiv_ploidy(.x, ...)

## S3 method for class 'tbl_df'
indiv_ploidy(.x, ...)

## S3 method for class 'vctrs_bigSNP'
indiv_ploidy(.x, ...)
```

Arguments

.x	a gen_tibble, or a vector of class vctrs_bigSNP (usually the genotype column of a gen_tibble object)
...	currently unused.

Value

a vector of ploidy, one per individuals in the gen_tibble

Examples

```
example_gt <- load_example_gt("gen_tbl")

example_gt %>% indiv_ploidy()
```

`is_loci_table_ordered` *Test if the loci table is ordered*

Description

This functions checks that all SNPs in a chromosome are adjacent in the loci table, and that positions are sorted within chromosomes.

Usage

```
is_loci_table_ordered(
  .x,
  error_on_false = FALSE,
  ignore_genetic_dist = TRUE,
  ...
)

## S3 method for class 'tbl_df'
is_loci_table_ordered(
  .x,
  error_on_false = FALSE,
  ignore_genetic_dist = TRUE,
  ...
)

## S3 method for class 'vctrs_bigSNP'
is_loci_table_ordered(
  .x,
  error_on_false = FALSE,
  ignore_genetic_dist = TRUE,
  ...
)
```

Arguments

`.x`	a vector of class `vctrs_bigSNP` (usually the `genotype` column of a `gen_tibble` object), or a `gen_tibble`.
`error_on_false`	logical, if `TRUE` an error is thrown if the loci are not ordered.
`ignore_genetic_dist`	logical, if `TRUE` the physical position is not checked.
`...`	other arguments passed to specific methods.

Value

a logical vector defining which loci are transversions

Examples

```
example_gt <- load_example_gt("gen_tbl")

example_gt %>% is_loci_table_ordered()
```

---

load_example_gt *Load example gen_tibble*

Description

This function creates a gen_tibble object for use in examples in documentation.

Usage

```
load_example_gt(
  type = c("gen_tbl", "grouped_gen_tbl", "grouped_gen_tbl_sf", "gen_tbl_sf")
)
```

Arguments

type	a character string indicating the type of gen_tibble to create: • "gen_tbl": a basic gen_tibble with genotype data and metadata • "grouped_gen_tbl": same as "gen_tbl" but grouped by population • "grouped_gen_tbl_sf": adds spatial features (longitude/latitude) and groups by population • "gen_tbl_sf": adds spatial features without grouping

Value

an example object of the class gen_tbl.

Examples

```
# This function creates an example gen_tibble object
example_gt <- load_example_gt("gen_tbl")
```

`loci_alt_freq` *Estimate allele frequencies at each locus*

Description

Allele frequencies can be estimates as minimum allele frequencies (MAF) with `loci_maf()` or the frequency of the alternate allele (with `loci_alt_freq()`). The latter are in line with the genotypes matrix (e.g. as extracted by `show_loci()`). Most users will be in interested in the MAF, but the raw frequencies might be useful when computing aggregated statistics. Both `loci_maf()` and `loci_alt_freq()` have efficient methods to support grouped `gen_tibble` objects. These can return a tidied tibble, a list, or a matrix.

Usage

```
loci_alt_freq(
  .x,
  .col = "genotypes",
  as_counts = FALSE,
  n_cores,
  block_size,
  type,
  ...
)

## S3 method for class 'tbl_df'
loci_alt_freq(
  .x,
  .col = "genotypes",
  as_counts = FALSE,
  n_cores = bigstatsr::nb_cores(),
  block_size = bigstatsr::block_size(nrow(.x), 1),
  ...
)

## S3 method for class 'vctrs_bigSNP'
loci_alt_freq(
  .x,
  .col = "genotypes",
  as_counts = FALSE,
  n_cores = bigstatsr::nb_cores(),
  block_size = bigstatsr::block_size(length(.x), 1),
  ...
)

## S3 method for class 'grouped_df'
loci_alt_freq(
  .x,
```

```
  .col = "genotypes",
  as_counts = FALSE,
  n_cores = bigstatsr::nb_cores(),
  block_size = bigstatsr::block_size(nrow(.x), 1),
  type = c("tidy", "list", "matrix"),
  ...
)

loci_maf(.x, .col = "genotypes", n_cores, block_size, type, ...)

## S3 method for class 'tbl_df'
loci_maf(
  .x,
  .col = "genotypes",
  n_cores = bigstatsr::nb_cores(),
  block_size = bigstatsr::block_size(nrow(.x), 1),
  ...
)

## S3 method for class 'vctrs_bigSNP'
loci_maf(
  .x,
  .col = "genotypes",
  n_cores = bigstatsr::nb_cores(),
  block_size = bigstatsr::block_size(length(.x), 1),
  ...
)

## S3 method for class 'grouped_df'
loci_maf(
  .x,
  .col = "genotypes",
  n_cores = bigstatsr::nb_cores(),
  block_size = bigstatsr::block_size(nrow(.x), 1),
  type = c("tidy", "list", "matrix"),
  ...
)
```

Arguments

`.x`	a vector of class `vctrs_bigSNP` (usually the `genotypes` column of a `gen_tibble` object), or a `gen_tibble`.
`.col`	the column to be used when a tibble (or grouped tibble is passed directly to the function). This defaults to "genotypes" and can only take that value. There is no need for the user to set it, but it is included to resolve certain tidyselect operations.
`as_counts`	boolean defining whether the count of alternate and valid (i.e. total number) alleles (rather than the frequencies) should be returned. It defaults to FALSE

	(i.e. frequencies are returned by default).
n_cores	number of cores to be used, it defaults to bigstatsr::nb_cores()
block_size	maximum number of loci read at once.
type	type of object to return, if using grouped method. One of "tidy", "list", or "matrix". Default is "tidy".
...	other arguments passed to specific methods, currently unused.

Value

a vector of frequencies, one per locus, if as_counts = FALSE; else a matrix of two columns, the count of alternate alleles and the count valid alleles (i.e. the sum of alternate and reference)

Examples

```
example_gt <- load_example_gt("gen_tbl")

# For alternate allele frequency
example_gt %>% loci_alt_freq()

# For alternate allele frequency per locus per population
example_gt %>%
  group_by(population) %>%
  loci_alt_freq()
# alternatively, return a list of populations with their frequencies
example_gt %>%
  group_by(population) %>%
  loci_alt_freq(type = "list")
# or a matrix with populations in columns and loci in rows
example_gt %>%
  group_by(population) %>%
  loci_alt_freq(type = "matrix")
# or within reframe (not recommended, as it much less efficient
# than using it directly as shown above)
library(dplyr)
example_gt %>%
  group_by(population) %>%
  reframe(alt_freq = loci_alt_freq(genotypes))
# For MAF
example_gt %>% loci_maf()

# For minor allele frequency per locus per population
example_gt %>%
  group_by(population) %>%
  loci_maf()
# alternatively, return a list of populations with their frequencies
example_gt %>%
  group_by(population) %>%
  loci_maf(type = "list")
# or a matrix with populations in columns and loci in rows
example_gt %>%
  group_by(population) %>%
```

```
loci_maf(type = "matrix")
```

`loci_chromosomes` *Get the chromosomes of loci in a* `gen_tibble`

Description

Extract the loci chromosomes from a `gen_tibble` (or directly from its `genotype` column).

Usage

```
loci_chromosomes(.x, .col = "genotypes", ...)

## S3 method for class 'tbl_df'
loci_chromosomes(.x, .col = "genotypes", ...)

## S3 method for class 'vctrs_bigSNP'
loci_chromosomes(.x, .col = "genotypes", ...)
```

Arguments

`.x`	a `gen_tibble`, or a vector of class `vctrs_bigSNP` (usually the `genotype` column of a `gen_tibble` object).
`.col`	the column to be used when a tibble (or grouped tibble is passed directly to the function). This defaults to "genotypes" and can only take that value. There is no need for the user to set it, but it is included to resolve certain tidyselect operations.
`...`	currently unused.

Value

a character vector of chromosomes

Examples

```
example_gt <- load_example_gt("gen_tbl")
example_gt %>% loci_chromosomes()
```

loci_hwe *Test Hardy-Weinberg equilibrium at each locus*

Description

Return the p-value from an exact test of HWE.

Usage

```
loci_hwe(.x, .col = "genotypes", ...)

## S3 method for class 'tbl_df'
loci_hwe(.x, .col = "genotypes", mid_p = TRUE, ...)

## S3 method for class 'vctrs_bigSNP'
loci_hwe(.x, .col = "genotypes", mid_p = TRUE, ...)

## S3 method for class 'grouped_df'
loci_hwe(
  .x,
  .col = "genotypes",
  mid_p = TRUE,
  n_cores = bigstatsr::nb_cores(),
  block_size = bigstatsr::block_size(nrow(.x), 1),
  type = c("tidy", "list", "matrix"),
  ...
)
```

Arguments

.x	a vector of class vctrs_bigSNP (usually the genotypes column of a gen_tibble object), or a gen_tibble.
.col	the column to be used when a tibble (or grouped tibble is passed directly to the function). This defaults to "genotypes" and can only take that value. There is no need for the user to set it, but it is included to resolve certain tidyselect operations.
...	not used.
mid_p	boolean on whether the mid-p value should be computed. Default is TRUE, as in PLINK.
n_cores	number of cores to be used, it defaults to bigstatsr::nb_cores()
block_size	maximum number of loci read at once.
type	type of object to return, if using grouped method. One of "tidy", "list", or "matrix". Default is "tidy".

Details

This function uses the original C++ algorithm from PLINK 1.90.

Value

a vector of probabilities from HWE exact test, one per locus

Author(s)

the C++ algorithm was written by Christopher Chang for PLINK 1.90, based on original code by Jan Wigginton (the code was released under GPL3).

Examples

```
example_gt <- load_example_gt("gen_tbl")

# For HWE
example_gt %>% loci_hwe()

# For loci_hwe per locus per population, use reframe
example_gt %>%
  group_by(population) %>%
  reframe(loci_hwe = loci_hwe(genotypes))
```

`loci_ld_clump` *Clump loci based on a Linkage Disequilibrium threshold*

Description

This function uses clumping to remove SNPs at high LD. When used with its default options, clumping based on MAF is similar to standard pruning (as done by PLINK with "–indep-pairwise (size+1) 1 thr.r2", but it results in a better spread of SNPs over the chromosome. This function is a wrapper around `bigsnpr::snp_clumping()`. See https://privefl.github.io/bigsnpr/articles/pruning-vs-clumping.html for more information on the differences between pruning and clumping.

Usage

```
loci_ld_clump(.x, .col = "genotypes", ...)

## S3 method for class 'tbl_df'
loci_ld_clump(.x, .col = "genotypes", ...)

## S3 method for class 'vctrs_bigSNP'
loci_ld_clump(
  .x,
  .col = "genotypes",
  S = NULL,
```

```
  thr_r2 = 0.2,
  size = 100/thr_r2,
  exclude = NULL,
  use_positions = TRUE,
  n_cores = 1,
  return_id = FALSE,
  ...
)
```

Arguments

.x	a gen_tibble object
.col	the column to be used when a tibble (or grouped tibble is passed directly to the function). This defaults to "genotypes" and can only take that value. There is no need for the user to set it, but it is included to resolve certain tidyselect operations.
...	currently not used.
S	A vector of loci statistics which express the importance of each SNP (the more important is the SNP, the greater should be the corresponding statistic). For example, if S follows the standard normal distribution, and "important" means significantly different from 0, you must use abs(S) instead. **If not specified, MAFs are computed and used.**
thr_r2	Threshold over the squared correlation between two SNPs. Default is 0.2.
size	For one SNP, window size around this SNP to compute correlations. Default is 100 / thr_r2 for clumping (0.2 -> 500; 0.1 -> 1000; 0.5 -> 200). If use_positions = FALSE, this is a window in number of SNPs, otherwise it is a window in kb (genetic distance). Ideally, use positions, as they provide a more sensible approach.
exclude	Vector of SNP indices to exclude anyway. For example, can be used to exclude long-range LD regions (see Price2008). Another use can be for thresholding with respect to p-values associated with S.
use_positions	boolean, if TRUE (the default), size is in kb, if FALSE size is the number of SNPs.
n_cores	number of cores to be used
return_id	boolean on whether the id of SNPs to keep should be returned. It defaults to FALSE, which returns a vector of booleans (TRUE or FALSE)

Details

Any missing values in the genotypes of a gen_tibble passed to loci_ld_clump will cause an error. To deal with missingness, see gt_impute_simple().

Value

a boolean vector indicating whether the SNP should be kept (if 'return_id = FALSE', the default), else a vector of SNP indices to be kept (if 'return_id = TRUE')

See Also

bigsnpr::snp_clumping() which this function wraps.

Examples

```
example_gt <- load_example_gt("gen_tbl") %>% gt_impute_simple()

# To return a boolean vector indicating whether the SNP should be kept
example_gt %>% loci_ld_clump()
# To return a vector of SNP indices to be kept
example_gt %>% loci_ld_clump(return_id = TRUE)
```

---

loci_missingness *Estimate missingness at each locus*

Description

Estimate the rate of missingness at each locus. This function has an efficient method to support grouped gen_tibble objects, which can return a tidied tibble, a list, or a matrix.

Usage

```
loci_missingness(
  .x,
  .col = "genotypes",
  as_counts = FALSE,
  n_cores = bigstatsr::nb_cores(),
  block_size,
  type,
  ...
)

## S3 method for class 'tbl_df'
loci_missingness(
  .x,
  .col = "genotypes",
  as_counts = FALSE,
  n_cores = bigstatsr::nb_cores(),
  block_size = bigstatsr::block_size(nrow(.x), 1),
  ...
)

## S3 method for class 'vctrs_bigSNP'
loci_missingness(
  .x,
```

```
  .col = "genotypes",
  as_counts = FALSE,
  n_cores = bigstatsr::nb_cores(),
  block_size = bigstatsr::block_size(length(.x), 1),
  ...
)

## S3 method for class 'grouped_df'
loci_missingness(
  .x,
  .col = "genotypes",
  as_counts = FALSE,
  n_cores = bigstatsr::nb_cores(),
  block_size = bigstatsr::block_size(nrow(.x), 1),
  type = c("tidy", "list", "matrix"),
  ...
)
```

Arguments

.x	a vector of class vctrs_bigSNP (usually the genotypes column of a gen_tibble object), or a gen_tibble.
.col	the column to be used when a tibble (or grouped tibble is passed directly to the function). This defaults to "genotypes" and can only take that value. There is no need for the user to set it, but it is included to resolve certain tidyselect operations.
as_counts	boolean defining whether the count of NAs (rather than the rate) should be returned. It defaults to FALSE (i.e. rates are returned by default).
n_cores	number of cores to be used, it defaults to bigstatsr::nb_cores()
block_size	maximum number of loci read at once.
type	type of object to return, if using grouped method. One of "tidy", "list", or "matrix". Default is "tidy".
...	other arguments passed to specific methods.

Value

a vector of frequencies, one per locus

Examples

```
example_gt <- load_example_gt("gen_tbl")

# For missingness
example_gt %>% loci_missingness()

# For missingness per locus per population
example_gt %>%
```

```
  group_by(population) %>%
  loci_missingness()
# alternatively, return a list of populations with their missingness
example_gt %>%
  group_by(population) %>%
  loci_missingness(type = "list")
# or a matrix with populations in columns and loci in rows
example_gt %>%
  group_by(population) %>%
  loci_missingness(type = "matrix")
# or within reframe (not recommended, as it much less efficient
# than using it directly as shown above)
example_gt %>%
  group_by(population) %>%
  reframe(missing = loci_missingness(genotypes))
```

`loci_names` *Get the names of loci in a* `gen_tibble`

Description

Extract the loci names from a `gen_tibble` (or directly from its `genotype` column).

Usage

```
loci_names(.x, .col = "genotypes", ...)

## S3 method for class 'tbl_df'
loci_names(.x, .col = "genotypes", ...)

## S3 method for class 'vctrs_bigSNP'
loci_names(.x, .col = "genotypes", ...)
```

Arguments

`.x`	a vector of class `vctrs_bigSNP` (usually the `genotype` column of a `gen_tibble` object), or a `gen_tibble`.
`.col`	the column to be used when a tibble (or grouped tibble is passed directly to the function). This defaults to "genotypes" and can only take that value. There is no need for the user to set it, but it is included to resolve certain tidyselect operations.
`...`	currently unused.

Value

a character vector of names

Examples

```
example_gt <- load_example_gt("gen_tbl")
example_gt %>% loci_names()
```

---

loci_pi *Estimate nucleotide diversity (pi) at each locus*

Description

Estimate nucleotide diversity (pi) at each locus, accounting for missing values. This uses the formula: c_0 * c_1 / (n * (n-1) / 2)

Usage

```
loci_pi(.x, .col = "genotypes", n_cores, block_size, type, ...)

## S3 method for class 'tbl_df'
loci_pi(
  .x,
  .col = "genotypes",
  n_cores = bigstatsr::nb_cores(),
  block_size = bigstatsr::block_size(nrow(.x), 1),
  ...
)

## S3 method for class 'vctrs_bigSNP'
loci_pi(
  .x,
  .col = "genotypes",
  n_cores = bigstatsr::nb_cores(),
  block_size = bigstatsr::block_size(length(.x), 1),
  ...
)

## S3 method for class 'grouped_df'
loci_pi(
  .x,
  .col = "genotypes",
  n_cores = bigstatsr::nb_cores(),
  block_size = bigstatsr::block_size(nrow(.x), 1),
  type = c("tidy", "list", "matrix"),
  ...
)
```

Arguments

.x	a vector of class vctrs_bigSNP (usually the genotypes column of a gen_tibble object), or a gen_tibble.
.col	the column to be used when a tibble (or grouped tibble is passed directly to the function). This defaults to "genotypes" and can only take that value. There is no need for the user to set it, but it is included to resolve certain tidyselect operations.
n_cores	number of cores to be used, it defaults to bigstatsr::nb_cores()
block_size	maximum number of loci read at once.
type	type of object to return, if using grouped method. One of "tidy", "list", or "matrix". Default is "tidy".
...	other arguments passed to specific methods, currently unused.

Value

a vector of frequencies, one per locus

Examples

```
example_gt <- load_example_gt("grouped_gen_tbl")

# For pi
example_gt %>% loci_pi()

# For pi per locus per population
example_gt %>%
  group_by(population) %>%
  loci_pi()
# alternatively, return a list of populations with their pi
example_gt %>%
  group_by(population) %>%
  loci_pi(type = "list")
# or a matrix with populations in columns and loci in rows
example_gt %>%
  group_by(population) %>%
  loci_pi(type = "matrix")
# or within reframe (not recommended, as it much less efficient
# than using it directly as shown above)
example_gt %>%
  group_by(population) %>%
  reframe(pi = loci_pi(genotypes))
```

loci_transitions *Find transitions*

Description

Use the loci table to define which loci are transitions

Usage

```
loci_transitions(.x, .col = "genotypes", ...)

## S3 method for class 'tbl_df'
loci_transitions(.x, .col = "genotypes", ...)

## S3 method for class 'vctrs_bigSNP'
loci_transitions(.x, .col = "genotypes", ...)
```

Arguments

.x	a vector of class vctrs_bigSNP (usually the genotype column of a gen_tibble object), or a gen_tibble.
.col	the column to be used when a tibble (or grouped tibble is passed directly to the function). This defaults to "genotypes" and can only take that value. There is no need for the user to set it, but it is included to resolve certain tidyselect operations.
...	other arguments passed to specific methods.

Value

a logical vector defining which loci are transitions

Examples

```
example_gt <- load_example_gt("gen_tbl")
example_gt %>% loci_transitions()
```

loci_transversions *Find transversions*

Description

Use the loci table to define which loci are transversions

Usage

```
loci_transversions(.x, .col = "genotypes", ...)

## S3 method for class 'tbl_df'
loci_transversions(.x, .col = "genotypes", ...)

## S3 method for class 'vctrs_bigSNP'
loci_transversions(.x, .col = "genotypes", ...)
```

Arguments

.x	a vector of class vctrs_bigSNP (usually the genotype column of a gen_tibble object), or a gen_tibble.
.col	the column to be used when a tibble (or grouped tibble is passed directly to the function). This defaults to "genotypes" and can only take that value. There is no need for the user to set it, but it is included to resolve certain tidyselect operations.
...	other arguments passed to specific methods.

Value

a logical vector defining which loci are transversions

Examples

```
example_gt <- load_example_gt("gen_tbl")
example_gt %>% loci_transversions()
```

---

mutate.gen_tbl *A mutate method for* gen_tibble *objects*

Description

A mutate method for gen_tibble objects

Usage

```
## S3 method for class 'gen_tbl'
mutate(..., deparse.level = 1)
```

Arguments

...	a gen_tibble and a data.frame or tibble
deparse.level	an integer controlling the construction of column names.

Value

a gen_tibble

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Add a new column
example_gt %>% mutate(region = "East")
```

---

mutate.grouped_gen_tbl

A mutate method for grouped gen_tibble *objects*

Description

A mutate method for grouped gen_tibble objects

Usage

```
## S3 method for class 'grouped_gen_tbl'
mutate(..., deparse.level = 1)
```

Arguments

...	a gen_tibble and a data.frame or tibble
deparse.level	an integer controlling the construction of column names.

Value

a grouped gen_tibble

Examples

```
test_gt <- load_example_gt("grouped_gen_tbl")
test_gt %>% mutate(region = "East")
test_gt <- load_example_gt("grouped_gen_tbl_sf")
test_gt %>% mutate(region = "East")
```

nwise_pop_pbs *Compute the Population Branch Statistics for each combination of populations*

Description

The function computes the population branch statistics (PBS) for each combination of populations at each locus. The PBS is a measure of the genetic differentiation between one focal population and two reference populations, and is used to identify outlier loci that may be under selection.

Usage

```
nwise_pop_pbs(
  .x,
  type = c("tidy", "matrix"),
  fst_method = c("Hudson", "Nei87", "WC84"),
  return_fst = FALSE
)
```

Arguments

.x	A grouped gen_tibble
type	type of object to return. One of "tidy" or "matrix". Default is "tidy".
fst_method	the method to use for calculating Fst, one of 'Hudson', 'Nei87', and 'WC84'. See pairwise_pop_fst() for details.
return_fst	A logical value indicating whether to return the Fst values along with the PBS values. Default is FALSE.

Value

Either a matrix with locus ID as rownames and the following columns:

- pbs_a.b.c: the PBS value for population a given b & c (there will be multiple such columns covering all 3 way combinations of populations in the grouped gen_tibble object)
- pbsn1_a.b.c: the normalized PBS value for population a given b & c.
- fst_a.b: the Fst value for population a and b, if return_fst is TRUE or a tidy tibble with the following columns:
- loci: the locus ID
- stat_name: the name of populations used in the pbs calculation (e.g. "pbs_pop1.pop2.pop3"). If return_fst is TRUE, stat_name will also include "fst" calculations in the same column (e.g. "fst_pop1.pop2").
- value: the pbs value for the populations

References

Yi X, et al. (2010) Sequencing of 50 human exomes reveals adaptation to high altitude. Science 329: 75-78.

Examples

```
example_gt <- load_example_gt()

# We can compute the PBS for all populations using "Hudson" method
example_gt %>%
  group_by(population) %>%
  nwise_pop_pbs(fst_method = "Hudson")
```

---

pairwise_allele_sharing

Compute the Pairwise Allele Sharing Matrix for a `gen_tibble` *object*

Description

This function computes the Allele Sharing matrix. Estimates Allele Sharing (equivalent to the quantity estimated by `hierfstat::matching()`) between pairs of individuals (for each locus, gives 1 if the two individuals are homozygous for the same allele, 0 if they are homozygous for a different allele, and 1/2 if at least one individual is heterozygous. Matching is the average of these 0, 1/2 and 1s)

Usage

```
pairwise_allele_sharing(
  x,
  as_matrix = FALSE,
  block_size = bigstatsr::block_size(nrow(x))
)
```

Arguments

`x`	a `gen_tibble` object.
`as_matrix`	boolean, determining whether the results should be a square symmetrical matrix (TRUE), or a tidied tibble (FALSE, the default)
`block_size`	maximum number of loci read at once. More loci should improve speed, but will tax memory.

Value

a matrix of allele sharing between all pairs of individuals

See Also

`hierfstat::matching()`

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Compute allele sharing between individuals
example_gt %>% pairwise_allele_sharing(as_matrix = FALSE)

# Alternatively, return as a tibble
example_gt %>% pairwise_allele_sharing(as_matrix = TRUE)
```

---

pairwise_grm *Compute the Genomic Relationship Matrix for a* gen_tibble *object*

Description

This function computes the Genomic Relationship Matrix (GRM). This is estimated by computing the pairwise kinship coefficients (coancestries) between all pairs of individuals from a matrix of Allele Sharing following the approach of Weir and Goudet 2017 based on beta estimators).

Usage

```
pairwise_grm(
  x,
  allele_sharing_mat = NULL,
  block_size = bigstatsr::block_size(nrow(x))
)
```

Arguments

x	a gen_tibble object.
allele_sharing_mat	optional, the matrix of Allele Sharing returned by pairwise_allele_sharing() with as_matrix=TRUE. As a number of statistics can be derived from the Allele Sharing matrix, it is sometimes more efficient to pre-compute this matrix.
block_size	the size of the blocks to use for the computation of the allele sharing matrix.

Details

The GRM is twice the coancestry matrix (e.g. as estimated by hierfstat::beta.dosage() with inb=FALSE).

Value

a matrix of GR between all pairs of individuals

See Also

hierfstat::beta.dosage()

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Compute the GRM from the allele sharing matrix
example_gt %>% pairwise_grm()

# To calculate using a precomputed allele sharing matrix, use:
allele_sharing <- example_gt %>% pairwise_allele_sharing(as_matrix = TRUE)
example_gt %>% pairwise_grm(allele_sharing_mat = allele_sharing)
```

---

pairwise_ibs *Compute the Identity by State Matrix for a* gen_tibble *object*

Description

This function computes the IBS matrix.

Usage

```
pairwise_ibs(
  x,
  as_matrix = FALSE,
  type = c("proportion", "adjusted_counts", "raw_counts"),
  block_size = bigstatsr::block_size(nrow(x))
)
```

Arguments

x	a gen_tibble object.
as_matrix	boolean, determining whether the results should be a square symmetrical matrix (TRUE), or a tidied tibble (FALSE, the default)
type	one of "proportion" (equivalent to "ibs" in PLINK), "adjusted_counts" ("distance" in PLINK), and "raw_counts" (the counts of identical alleles and non-missing alleles, from which the two other quantities are computed)
block_size	maximum number of loci read at once. More loci should improve speed, but will tax memory.

Details

Note that monomorphic sites are currently considered. Remove monomorphic sites before running pairwise_king if this is a concern.

Value

a bigstatsr::FBM of proportion or adjusted counts, or a list of two bigstatsr::FBM matrices, one of counts of IBS by alleles, and one of number of valid alleles (i.e. 2*n_loci* - 2missing_loci)

Examples

```
example_gt <- load_example_gt("gen_tbl")

pairwise_ibs(example_gt, type = "proportion")

# Alternatively, return a matrix
pairwise_ibs(example_gt, type = "proportion", as_matrix = TRUE)

# Adjust block_size
pairwise_ibs(example_gt, block_size = 2)

# Change type
pairwise_ibs(example_gt, type = "adjusted_counts")
pairwise_ibs(example_gt, type = "raw_counts")
```

---

pairwise_king *Compute the KING-robust Matrix for a* gen_tibble *object*

Description

This function computes the KING-robust estimator of kinship, reimplementing the KING algorithm of Manichaikul et al. (2010).

Usage

```
pairwise_king(
  x,
  as_matrix = FALSE,
  block_size = bigstatsr::block_size(nrow(x))
)
```

Arguments

x	a gen_tibble object.
as_matrix	boolean, determining whether the results should be a square symmetrical matrix (TRUE), or a tidied tibble (FALSE, the default)
block_size	maximum number of loci read at once. More loci should improve speed, but will tax memory.

Value

a square symmetrical matrix of relationship coefficients between individuals if as_matrix is TRUE, or a tidied tibble of coefficients if as_matrix is FALSE.

References

Manichaikul, A. et al. (2010) Robust relationship inference in genome-wide association studies. Bioinformatics, 26(22), 2867–2873. https://doi.org/10.1093/bioinformatics/btq559.

Note that monomorphic sites are currently considered. Remove monomorphic sites before running pairwise_king if this is a concern.

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Compute the KING-robust matrix
pairwise_king(example_gt, as_matrix = TRUE)

# Or return a tidy tibble
pairwise_king(example_gt, as_matrix = FALSE)

# Adjust block_size
pairwise_king(example_gt, block_size = 2)
```

---

pairwise_pop_fst *Compute pairwise population Fst*

Description

This function computes pairwise Fst. The following methods are implemented:

- 'Hudson': Hudson's formulation, as derived in Bhatia et al (2013) for diploids. This is the only method that can also be used with pseudohaploid data.
- 'Nei87' : Fst according to Nei (1987) - includes the correction for heterozygosity when computing Ht (it uses the same formulation as in `hierfstat::pairwise.neifst()`),
- 'WC84' : Weir and Cockerham (1984), correcting for missing data (it uses the same formulation as in `hierfstat::pairwise.WCfst()`).

Usage

```
pairwise_pop_fst(
  .x,
  type = c("tidy", "pairwise"),
  by_locus = FALSE,
  by_locus_type = c("tidy", "matrix", "list"),
  method = c("Hudson", "Nei87", "WC84"),
  return_num_dem = FALSE,
  n_cores = bigstatsr::nb_cores()
)
```

Arguments

.x	a grouped gen_tibble (as obtained by using dplyr::group_by())
type	type of object to return One of "tidy" or "pairwise" for a pairwise matrix of populations. Default is "tidy".
by_locus	boolean, determining whether Fst should be returned by locus(TRUE), or as a single genome wide value obtained by taking the ratio of the mean numerator and denominator (FALSE, the default).
by_locus_type	type of object to return. One of "tidy", "matrix" or "list". Default is "tidy".
method	one of 'Hudson', 'Nei87', and 'WC84'
return_num_dem	returns a list of numerators and denominators for each locus. This is useful for creating windowed estimates of Fst (as we need to compute the mean numerator and denominator within each window). Default is FALSE.
n_cores	number of cores to be used, it defaults to bigstatsr::nb_cores()

Details

For all formulae, the genome wide estimate is obtained by taking the ratio of the mean numerators and denominators over all relevant SNPs.

Value

if type=tidy, a tibble of genome-wide pairwise Fst values with each pairwise combination as a row if "by_locus=FALSE", else a list including the tibble of genome-wide values as well as a matrix with pairwise Fst by locus with loci as rows and and pairwise combinations as columns. If type=pairwise, a matrix of genome-wide pairwise Fst values is returned.

References

Bhatia G, Patterson N, Sankararaman S, Price AL. (2013) Estimating and Interpreting FST: The Impact of Rare Variants. Genome Research, 23(9):1514–1521.

Nei, M. (1987) Molecular Evolutionary Genetics. Columbia University Press

Weir, B. S., & Cockerham, C. C. (1984). Estimating F-statistics for the analysis of population structure. Evolution, 38(6): 1358–1370.

See Also

hierfstat::pairwise.neifst()

Examples

```
example_gt <- load_example_gt("gen_tbl")

# For a basic global pairwise Fst calculation:
example_gt %>%
  group_by(population) %>%
  pairwise_pop_fst(method = "Nei87")
```

```
# With a pairwise matrix:
example_gt %>%
  group_by(population) %>%
  pairwise_pop_fst(method = "Nei87", type = "pairwise")

# To calculate Fst by locus:
example_gt %>%
  group_by(population) %>%
  pairwise_pop_fst(method = "Hudson", by_locus = TRUE)
```

`pop_fis` *Compute population specific FIS*

Description

This function computes population specific FIS, using either the approach of Nei 1987 (with an algorithm equivalent to the one used by `hierfstat::basic.stats()`) or of Weir and Goudet 2017 (with an algorithm equivalent to the one used by `hierfstat::fis.dosage()`).

Usage

```
pop_fis(
  .x,
  method = c("Nei87", "WG17"),
  by_locus = FALSE,
  include_global = FALSE,
  allele_sharing_mat = NULL
)
```

Arguments

`.x`	a grouped `gen_tibble` (as obtained by using `dplyr::group_by()`)
`method`	one of "Nei87" (based on Nei 1987, eqn 7.41) or "WG17" (for Weir and Goudet 2017) to compute FIS
`by_locus`	boolean, determining whether FIS should be returned by locus(TRUE), or as a single genome wide value (FALSE, the default). Note that this is only relevant for "Nei87", as "WG17" always returns a single value.
`include_global`	boolean determining whether, besides the population specific estimates, a global estimate should be appended. Note that this will return a vector of n populations plus 1 (the global value), or a matrix with n+1 columns if `by_locus=TRUE`.
`allele_sharing_mat`	optional and only relevant for "WG17", the matrix of Allele Sharing returned by `pairwise_allele_sharing()` with `as_matrix=TRUE`. As a number of statistics can be derived from the Allele Sharing matrix, it is sometimes more efficient to pre-compute this matrix.

Value

a vector of population specific fis (plus the global value if `include_global=TRUE`)

References

Nei M. (1987) Molecular Evolutionary Genetics. Columbia University Press Weir, BS and Goudet J (2017) A Unified Characterization of Population Structure and Relatedness. Genetics (2017) 206:2085

See Also

`hierfstat::basic.stats()` `hierfstat::fis.dosage()`

Examples

```
example_gt <- load_example_gt("grouped_gen_tbl")

# Compute FIS using Nei87
example_gt %>% pop_fis(method = "Nei87")

# Compute FIS using WG17
example_gt %>% pop_fis(method = "WG17")

# To include the global FIS, set include_global = TRUE
example_gt %>% pop_fis(method = "Nei87", include_global = TRUE)

# To return FIS by locus, set by_locus = TRUE
example_gt %>% pop_fis(method = "Nei87", by_locus = TRUE)

# To calculate from a pre-computed allele sharing matrix:
allele_sharing_mat <- pairwise_allele_sharing(example_gt, as_matrix = TRUE)
example_gt %>% pop_fis(
  method = "WG17",
  allele_sharing_mat = allele_sharing_mat
)
```

---

pop_fst *Compute population specific Fst*

Description

This function computes population specific Fst, using the approach in Weir and Goudet 2017 (as computed by `hierfstat::fst.dosage()`).

Usage

```
pop_fst(.x, include_global = FALSE, allele_sharing_mat = NULL)
```

Arguments

.x	a grouped gen_tibble (as obtained by using dplyr::group_by())
include_global	boolean determining whether, besides the population specific Fst, a global Fst should be appended. Note that this will return a vector of n populations plus 1 (the global value)
allele_sharing_mat	optional, the matrix of Allele Sharing returned by pairwise_allele_sharing() with as_matrix=TRUE. As a number of statistics can be derived from the Allele Sharing matrix,

Value

a vector of population specific Fst (plus the global value if include_global=TRUE)

References

Weir, BS and Goudet J (2017) A Unified Characterization of Population Structure and Relatedness. Genetics (2017) 206:2085

See Also

hierfstat::fst.dosage()

Examples

```
example_gt <- load_example_gt("grouped_gen_tbl")

# Compute FIS using Nei87
example_gt %>% pop_fst()

# To include the global Fst, set include_global = TRUE
example_gt %>% pop_fst(include_global = TRUE)

# To calculate from a pre-computed allele sharing matrix:
allele_sharing_mat <- pairwise_allele_sharing(example_gt, as_matrix = TRUE)
example_gt %>% pop_fst(allele_sharing_mat = allele_sharing_mat)
```

---

pop_global_stats *Compute basic population global statistics*

Description

This function computes basic population global statistics, following the notation in Nei 1987 (which in turn is based on Nei and Chesser 1983):

- observed heterozygosity ($\hat{h}_o$, column header Ho)
- expected heterozygosity, also known as gene diversity ($\hat{h}_s$, Hs)

- total heterozygosity ($\hat{h}_t$, `Ht`)
- genetic differentiation between subpopulations (D_{st}, `Dst`)
- corrected total population diversity (h'_t, `Htp`)
- corrected genetic differentiation between subpopulations (D'_{st}, `Dstp`)
- $\hat{F}_{ST}$ (column header, `Fst`)
- corrected $\hat{F'}_{ST}$ (column header `Fstp`)
- $\hat{F}_{IS}$ (column header, `Fis`)
- Jost's $\hat{D}$ (column header, `Dest`)

Usage

```
pop_global_stats(.x, by_locus = FALSE, n_cores = bigstatsr::nb_cores())
```

Arguments

`.x`	a `gen_tibble` (usually grouped, as obtained by using `dplyr::group_by()`; use on a single population will return a number of quantities as NA/NaN)
`by_locus`	boolean, determining whether the statistics should be returned by locus(TRUE), or as a single genome wide value (FALSE, the default).
`n_cores`	number of cores to be used, it defaults to `bigstatsr::nb_cores()`

Details

We use the notation of Nei 1987. That notation was for loci with m alleles, but in our case we only have two alleles, so `m=2`.

- Within population observed heterozygosity $\hat{h}_o$ for a locus with m alleles is defined as:
  $\hat{h}_o = 1 - \sum_{k=1}^{s} \sum_{i=1}^{m} \hat{X}_{kii}/s$
  where
  $\hat{X}_{kii}$ represents the proportion of homozygote i in the sample for the kth population and
  s the number of populations,
  following equation 7.38 in Nei(1987) on pp.164.

- Within population expected heterozygosity (gene diversity) $\hat{h}_s$ for a locus with m alleles is defined as:
  $\hat{h}_s = (\tilde{n}/(\tilde{n}-1))[1 - \sum_{i=1}^{m} \hat{\bar{x}}_i^2 - \hat{h}_o/2\tilde{n}]$
  #nolint where
  $\tilde{n} = s/\sum_k 1/n_k$ (i.e the harmonic mean of n_k) and
  $\hat{\bar{x}}_i^2 = \sum_k \hat{x}_{ki}^2/s$
  following equation 7.39 in Nei(1987) on pp.164.
- Total heterozygosity (total gene diversity) $\hat{h}_t$ for a locus with m alleles is defined as:
  $\hat{h}_t = 1 - \sum_{i=1}^{m} \hat{\bar{x}}_i^2 + \hat{h}_s/(\tilde{n}s) - \hat{h}_o/(2\tilde{n}s)$
  where
  $\hat{x}_i = \sum_k \hat{x}_{ki}/s$
  following equation 7.40 in Nei(1987) on pp.164.

- The amount of gene diversity among samples D_{ST} is defined as:
  $D_{ST} = \hat{h}_t - \hat{h}_s$
  following the equation provided in the text at the top of page 165 in Nei(1987).
- The corrected amount of gene diversity among samples D'_{ST} is defined as:
  $D'_{ST} = (s/(s-1))D'_{ST}$
  following the equation provided in the text at the top of page 165 in Nei(1987).
- Total corrected heterozygosity (total gene diversity) $\hat{h}_t$ is defined as:
  $\hat{h'}_t = \hat{h}_s + D'_{ST}$
  following the equation provided in the text at the top of page 165 in Nei(1987).
- $\hat{F}_{IS}$ is defined as:
  $\hat{F}_{IS} = 1 - \hat{h}_o/\hat{h}_s$
  following equation 7.41 in Nei(1987) on pp.164.
- $\hat{F}_{ST}$ is defined as:
  $\hat{F}_{ST} = 1 - \hat{h}_s/\hat{h}_t = D_{ST}/\hat{h}_t$
  following equation 7.43 in Nei(1987) on pp.165.
- $\hat{F'}_{ST}$ is defined as:
  $\hat{F'}_{ST} = D'_{ST}/\hat{h'}_t$
  following the explanation provided in the text at the top of page 165 in Nei(1987).
- Jost's $\hat{D}$ is defined as:
  $\hat{D} = (s/(s-1))((\hat{h'}_t - \hat{h}_s)/(1 - \hat{h}_s))$
  as defined by Jost(2008)

  All these statistics are first computed by locus, and then averaged across loci (including any monomorphic locus) to obtain genome-wide values. The function uses the same algorithm as `hierfstat::basic.stats()` but is optimized for speed and memory usage.

Value

a tibble of population statistics, with populations as rows and statistics as columns

References

Nei M, Chesser R (1983) Estimation of fixation indexes and gene diversities. Annals of Human Genetics, 47, 253-259.

Nei M. (1987) Molecular Evolutionary Genetics. Columbia University Press, pp. 164-165.

Jost L (2008) GST and its relatives do not measure differentiation. Molecular Ecology, 17, 4015-4026.

See Also

`hierfstat::basic.stats()`

Examples

```
example_gt <- load_example_gt("grouped_gen_tbl")

# Compute population global statistics
example_gt %>% pop_global_stats()

# To return by locus, set by_locus = TRUE
example_gt %>% pop_global_stats(by_locus = TRUE)
```

---

pop_het_exp *Compute the population expected heterozygosity*

Description

This function computes expected population heterozygosity (also referred to as gene diversity, to avoid the potentially misleading use of the term "expected" in this context), using the formula of Nei (1987).

Usage

```
pop_het_exp(
  .x,
  by_locus = FALSE,
  include_global = FALSE,
  n_cores = bigstatsr::nb_cores()
)

pop_gene_div(
  .x,
  by_locus = FALSE,
  include_global = FALSE,
  n_cores = bigstatsr::nb_cores()
)
```

Arguments

.x	a gen_tibble (usually grouped, as obtained by using dplyr::group_by(), otherwise the full tibble will be considered as belonging to a single population).
by_locus	boolean, determining whether Hs should be returned by locus(TRUE), or as a single genome wide value (FALSE, the default).
include_global	boolean determining whether, besides the population specific estimates, a global estimate should be appended. Note that this will return a vector of n populations plus 1 (the global value), or a matrix with n+1 columns if by_locus=TRUE.
n_cores	number of cores to be used, it defaults to bigstatsr::nb_cores()

Details

Within population expected heterozygosity (gene diversity) $\hat{h}_s$ for a locus with m alleles is defined as:
$\hat{h}_s = \tilde{n}/(\tilde{n}-1)[1 - \sum_i^m \hat{\bar{x}}_i^2 - \hat{h}_o/2\tilde{n}]$
#nolint

where
$\tilde{n} = s/\sum_k 1/n_k$ (i.e the harmonic mean of n_k) and
$\hat{\bar{x}}_i^2 = \sum_k \hat{x}_{ki}^2/s$
following equation 7.39 in Nei(1987) on pp.164. In our specific case, there are only two alleles, so $m = 2$. $\hat{h}_s$ at the genome level for each population is simply the mean of the locus estimates for each population.

Value

a vector of mean population observed heterozygosities (if `by_locus=FALSE`), or a matrix of estimates by locus (rows are loci, columns are populations, `by_locus=TRUE`)

References

Nei M. (1987) Molecular Evolutionary Genetics. Columbia University Press

Examples

```
example_gt <- load_example_gt("grouped_gen_tbl")

# Compute expected heterozygosity
example_gt %>% pop_het_exp()

# To include the global expected heterozygosity, set include_global = TRUE
example_gt %>% pop_het_exp(include_global = TRUE)

# To return by locus, set by_locus = TRUE
example_gt %>% pop_het_exp(by_locus = TRUE)
```

---

pop_het_obs *Compute the population observed heterozygosity*

Description

This function computes population heterozygosity, using the formula of Nei (1987).

Usage

```
pop_het_obs(
  .x,
  by_locus = FALSE,
  include_global = FALSE,
  n_cores = bigstatsr::nb_cores()
)
```

Arguments

.x	a gen_tibble (usually grouped, as obtained by using dplyr::group_by(), otherwise the full tibble will be considered as belonging to a single population).
by_locus	boolean, determining whether Ho should be returned by locus(TRUE), or as a single genome wide value (FALSE, the default).
include_global	boolean determining whether, besides the population specific estimates, a global estimate should be appended. Note that this will return a vector of n populations plus 1 (the global value), or a matrix with n+1 columns if by_locus=TRUE.
n_cores	number of cores to be used, it defaults to bigstatsr::nb_cores()

Details

Within population observed heterozygosity $\hat{h}_o$ for a locus with m alleles is defined as:
$\hat{h}_o = 1 - \sum_{k=1}^{s} \sum_{i=1}^{m} \hat{X}_{kii}/s$
where
$\hat{X}_{kii}$ represents the proportion of homozygote i in the sample for the kth population and
s the number of populations,
following equation 7.38 in Nei(1987) on pp.164. In our specific case, there are only two alleles, so $m = 2$. For population specific estimates, the sum is done over a single value of k. $\hat{h}_o$ at the genome level is simply the mean of the locus estimates.

Value

a vector of mean population observed heterozygosities (if by_locus=FALSE), or a matrix of estimates by locus (rows are loci, columns are populations, by_locus=TRUE)

References

Nei M. (1987) Molecular Evolutionary Genetics. Columbia University Press

Examples

```
example_gt <- load_example_gt("grouped_gen_tbl")

# Compute expected heterozygosity
example_gt %>% pop_het_obs()

# To include the global expected heterozygosity, set include_global = TRUE
example_gt %>% pop_het_obs(include_global = TRUE)
```

```
# To return by locus, set by_locus = TRUE
example_gt %>% pop_het_obs(by_locus = TRUE)
```

---

pop_tajimas_d *Estimate Tajima's D for the whole genome*

Description

Note that Tajima's D estimates from data that have been filtered or ascertained can be difficult to interpret. This function should ideally be used on sequence data prior to filtering.

Usage

```
pop_tajimas_d(.x, n_cores, block_size, ...)

## S3 method for class 'tbl_df'
pop_tajimas_d(
  .x,
  n_cores = bigstatsr::nb_cores(),
  block_size = bigstatsr::block_size(nrow(.x), 1),
  ...
)

## S3 method for class 'vctrs_bigSNP'
pop_tajimas_d(
  .x,
  n_cores = bigstatsr::nb_cores(),
  block_size = bigstatsr::block_size(length(.x), 1),
  ...
)

## S3 method for class 'grouped_df'
pop_tajimas_d(
  .x,
  n_cores = bigstatsr::nb_cores(),
  block_size = bigstatsr::block_size(nrow(.x), 1),
  ...
)
```

Arguments

.x	a vector of class vctrs_bigSNP (usually the genotypes column of a gen_tibble object), or a gen_tibble.
n_cores	number of cores to be used, it defaults to bigstatsr::nb_cores()
block_size	maximum number of loci read at once.
...	other arguments passed to specific methods, currently unused.

Value

A single numeric value (Tajima's D D) for the whole data set, NA when the statistic is not defined. For grouped data a list of Tajima's D D values (one per group) is returned.

Examples

```
example_gt <- load_example_gt("grouped_gen_tbl")

# Compute Tajima's D
example_gt %>% pop_tajimas_d()
```

---

predict.gt_pca *Predict scores of a PCA*

Description

Predict the PCA scores for a gt_pca, either for the original data or projecting new data.

Usage

```
## S3 method for class 'gt_pca'
predict(
  object,
  new_data = NULL,
  project_method = c("none", "simple", "OADP", "least_squares"),
  lsq_pcs = c(1, 2),
  block_size = NULL,
  n_cores = 1,
  as_matrix = TRUE,
  ...
)
```

Arguments

object	the gt_pca object
new_data	a gen_tibble if scores are requested for a new dataset
project_method	a string taking the value of either "simple", "OADP" (Online Augmentation, Decomposition, and Procrustes (OADP) projection), or "least_squares" (as done by SMARTPCA)
lsq_pcs	a vector of the indices of the principal components to use for the least square fitting. Only relevant if project_method = 'least_squares'. It defaults to the first two components.
block_size	number of loci read simultaneously (larger values will speed up computation, but require more memory)

n_cores	number of cores
as_matrix	logical, whether to return the result as a matrix (default) or a tibble.
...	no used

Value

a matrix of predictions (in line with predict using a prcomp object) or a tibble, with samples as rows and components as columns. The number of components depends on how many were estimated in the gt_pca object.

References

Zhang et al (2020). Fast and robust ancestry prediction using principal component analysis 36(11): 3439–3446.

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

# Subset into two datasets: one original and one to predict
original_lobsters <- lobsters[c(1:150), ]
new_lobsters <- lobsters[c(151:176), ]

# Create PCA object
pca <- gt_pca_partialSVD(original_lobsters)

# Predict
predict(pca, new_data = new_lobsters, project_method = "simple")

# Predict with OADP
predict(pca, new_data = new_lobsters, project_method = "OADP")

# Predict with least squares
predict(pca,
  new_data = new_lobsters,
  project_method = "least_squares", lsq_pcs = c(1, 2, 3)
)

# Return a tibble
predict(pca, new_data = new_lobsters, as_matrix = FALSE)
```

```
# Adjust block.size
predict(pca, new_data = new_lobsters, block_size = 10)
```

`qc_report_indiv` *Create a Quality Control report for individuals*

Description

Return QC information to assess loci (Observed heterozygosity and missingness).

Usage

```
qc_report_indiv(.x, ...)

## S3 method for class 'tbl_df'
qc_report_indiv(.x, kings_threshold = NULL, ...)

## S3 method for class 'grouped_df'
qc_report_indiv(.x, kings_threshold = NULL, ...)
```

Arguments

`.x`	either a `gen_tibble` object or a grouped `gen_tibble` (as obtained by using `dplyr::group_by()`)
`...`	further arguments to pass
`kings_threshold`	an optional numeric giving a KING kinship coefficient, or one of: • "first": removing first degree relatives, equivalent to a kinship coefficient of 0.177 or more • "second": removing second degree relatives, equivalent to a kinship coefficient of 0.088 or more

Details

Providing the parameter kings_threshold will return two additional columns, 'id' containing the ID of individuals, and 'to_keep' a logical vector describing whether the individual should be removed to retain the largest possible set of individuals with no relationships above the threshold. The calculated pairwise KING relationship matrix is also returned as an attribute of 'to_keep'. The kings_threshold parameter can be either a numeric KING kinship coefficient or a string of either "first" or "second", to remove any first degree or second degree relationships from the dataset. This second option is similar to using –unrelated –degree 1 or –unrelated –degree 2 in KING. For pseudohaploid data, only missingness and ploidy are reported.

Value

If no kings_threshold is provided, a tibble with 2 elements: het_obs and missingness. If kings_threshold is provided, a tibble with 4 elements: het_obs, missingness, id and to_keep. For pseudohaploid data, a tibble with ploidy and missingness.

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
example_gt <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Get QC report for individuals
example_gt %>% qc_report_indiv()

# Get QC report with kinship filtering
example_gt %>% qc_report_indiv(kings_threshold = "first")
```

---

qc_report_loci *Create a Quality Control report for loci*

Description

Return QC information to assess loci (MAF, missingness and HWE test). For pseudohaploid data, HWE test is not calculated.

Usage

```
qc_report_loci(.x, ...)

## S3 method for class 'tbl_df'
qc_report_loci(.x, ...)

## S3 method for class 'grouped_df'
qc_report_loci(.x, ...)
```

Arguments

.x	a gen_tibble object.
...	currently unused

Value

either a tibble with 3 elements (maf, missingness and hwe_p). For pseudohaploid data, a tibble with 2 elements (maf and missingness).

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
example_gt <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Get a QC report for the loci
example_gt %>% qc_report_loci()

# Group by population to calculate HWE within populations
example_gt <- example_gt %>% group_by(population)
example_gt %>% qc_report_loci()
```

---

q_matrix *Convert a standard matrix to a* q_matrix *object*

Description

Takes a single Q matrix that exists as either a matrix or a data frame and returns a q_matrix object.

Usage

```
q_matrix(x)
```

Arguments

x	A matrix or a data frame

Value

A q_matrix object

Examples

```
# Read in a single .Q file
q_mat <- read.table(system.file("extdata", "anolis", "anolis_ld_run1.3.Q",
  package = "tidypopgen"
))
class(q_mat)

# Convert to a Q matrix object
q_mat <- q_matrix(q_mat)
class(q_mat)
```

`rbind.gen_tbl` *Combine two gen_tibbles*

Description

This function combined two gen_tibbles. By defaults, it subsets the loci and swaps ref and alt alleles to make the two datasets compatible (this behaviour can be switched off with `as_is`). The first object is used as a "reference" , and SNPs in the other dataset will be flipped and/or alleles swapped as needed. SNPs that have different alleles in the two datasets (i.e. triallelic) will also be dropped. There are also options (NOT default) to attempt strand flipping to match alleles (often needed in human datasets from different SNP chips), and remove ambiguous alleles (C/G and A/T) where the correct strand can not be guessed.

Usage

```
## S3 method for class 'gen_tbl'
rbind(
  ...,
  as_is = FALSE,
  flip_strand = FALSE,
  use_position = FALSE,
  quiet = FALSE,
  backingfile = NULL
)
```

Arguments

`...`	two `gen_tibble` objects. Note that this function can not take more objects, `rbind` has to be done sequentially for large sets of objects.
`as_is`	boolean determining whether the loci should be left as they are before merging. If FALSE (the defaults), `rbind` will attempt to subset and swap alleles as needed.
`flip_strand`	boolean on whether strand flipping should be checked to match the two datasets. If this is set to TRUE, ambiguous SNPs (i.e. A/T and C/G) will also be removed. It defaults to FALSE
`use_position`	boolean of whether a combination of chromosome and position should be used for matching SNPs. By default, `rbind` uses the locus name, so this is set to FALSE. When using 'use_position=TRUE', make sure chromosomes are coded in the same way in both `gen_tibbles` (a mix of e.g. 'chr1', '1' or 'chromosome1' can be the reasons if an unexpectedly large number variants are dropped when merging).
`quiet`	boolean whether to omit reporting to screen
`backingfile`	the path and prefix of the files used to store the merged data (it will be a .RDS to store the `bigSNP` object and a .bk file as its backing file for the FBM)

Details

rbind differs from merging data with plink, which swaps the order of allele1 and allele2 according to minor allele frequency when merging datasets. rbind flips and/or swaps alleles according to the reference dataset, not according to allele frequency.

Value

a gen_tibble with the merged data.

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Create a second gen_tibble to merge
test_indiv_meta <- data.frame(
  id = c("x", "y", "z"),
  population = c("pop1", "pop1", "pop2")
)
test_genotypes <- rbind(
  c(1, 1, 0, 1, 1, 0),
  c(2, 1, 0, 0, 0, 0),
  c(2, 2, 0, 0, 1, 1)
)
test_loci <- data.frame(
  name = paste0("rs", 1:6),
  chromosome = paste0("chr", c(1, 1, 1, 1, 2, 2)),
  position = as.integer(c(3, 5, 65, 343, 23, 456)),
  genetic_dist = as.double(rep(0, 6)),
  allele_ref = c("A", "T", "C", "G", "C", "T"),
  allele_alt = c("T", "C", NA, "C", "G", "A")
)

test_gt <- gen_tibble(
  x = test_genotypes,
  loci = test_loci,
  indiv_meta = test_indiv_meta,
  valid_alleles = c("A", "T", "C", "G"),
  quiet = TRUE
)

# Merge the datasets using rbind
merged_gt <- rbind(ref = example_gt, target = test_gt, flip_strand = TRUE)

merged_gt
```

---

rbind_dry_run *Generate a report of what would happen to each SNP in a merge*

Description

This function provides an overview of the fate of each SNP in two gen_tibble objects in the case of a merge. Only SNPs found in both objects will be kept. One object is used as a reference, and SNPs in the other dataset will be flipped and/or alleles swapped as needed. SNPs that have different alleles in the two datasets will also be dropped.

Usage

```
rbind_dry_run(
  ref,
  target,
  use_position = FALSE,
  flip_strand = FALSE,
  quiet = FALSE
)
```

Arguments

ref	either a gen_tibble object, or the path to the PLINK bim file; the alleles in this objects will be used as template to flip the ones in target and/or swap their order as necessary.
target	either a gen_tibble object, or the path to the PLINK bim file
use_position	boolean of whether a combination of chromosome and position should be used for matching SNPs. By default, rbind uses the locus name, so this is set to FALSE. When using 'use_position=TRUE', make sure chromosomes are coded in the same way in both gen_tibbles (a mix of e.g. 'chr1', '1' or 'chromosome1' can be the reasons if an unexpectedly large number variants are dropped when merging).
flip_strand	boolean on whether strand flipping should be checked to match the two datasets. Ambiguous SNPs (i.e. A/T and C/G) will also be removed. It defaults to FALSE
quiet	boolean whether to omit reporting to screen

Value

a list with two data.frames, named target and ref. Each data.frame has nrow() equal to the number of loci in the respective dataset, a column id with the locus name, and boolean columns to_keep (the valid loci that will be kept in the merge), alleles_mismatched (loci found in both datasets but with mismatched alleles, leading to those loci being dropped), to_flip (loci that need to be flipped to align the two datasets, only found in target data.frame) and to_swap (loci for which the order of alleles needs to be swapped to align the two datasets, target data.frame)

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Create a second gen_tibble to merge
test_indiv_meta <- data.frame(
  id = c("x", "y", "z"),
```

```
  population = c("pop1", "pop1", "pop2")
)
test_genotypes <- rbind(
  c(1, 1, 2, 1, 1),
  c(2, 1, 2, 0, 0),
  c(2, 2, 2, 0, 1)
)
test_loci <- data.frame(
  name = paste0("rs", 1:5),
  chromosome = paste0("chr", c(1, 1, 1, 1, 2)),
  position = as.integer(c(3, 5, 65, 343, 23)),
  genetic_dist = as.double(rep(0, 5)),
  allele_ref = c("A", "T", "C", "G", "C"),
  allele_alt = c("T", "C", NA, "C", "G")
)

test_gt <- gen_tibble(
  x = test_genotypes,
  loci = test_loci,
  indiv_meta = test_indiv_meta,
  valid_alleles = c("A", "T", "C", "G"),
  quiet = TRUE
)

# Create an rbind report using rbind_dry_run
rbind_dry_run(example_gt, test_gt, flip_strand = TRUE)
```

`read_q_files` *Read and structure .Q files or existing matrices as* `q_matrix` *or* `gt_admix` *objects.*

Description

This function reads .Q matrix files generated by external clustering algorithms (such as ADMIXTURE) and transforms them into `gt_admix` objects.

Usage

```
read_q_files(x)
```

Arguments

`x`	can be: • a path to a directory containing .Q files

Value

- a `gt_admix` object containing a list of Q matrices and a list of indices for each Q matrix separated by K

Examples

```
q_files_path <- system.file("extdata", "anolis", package = "tidypopgen")

admix_obj <- read_q_files(q_files_path)
summary(admix_obj)
```

`scale_fill_distruct` *Scale constructor using the distruct colours*

Description

A wrapper around `ggplot2::scale_fill_manual()`, using the distruct colours from `distruct_colours`.

Usage

```
scale_fill_distruct(guide = "none", ...)
```

Arguments

`guide`	guide function passed to `ggplot2::scale_fill_manual()`. Defaults to "none", set to "legend" if a legend is required.
`...`	further parameters to be passed to `ggplot2::scale_fill_manual()`

Value

a scale constructor to be used with ggplot

See Also

`ggplot2::scale_fill_manual()` which this function wraps.

Examples

```
library(ggplot2)
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

# Create PCA object
pca <- gt_pca_partialSVD(lobsters)
```

```
# Colour by population
autoplot(pca, type = "scores") +
  aes(colour = lobsters$population) + scale_fill_distruct()
```

`select_loci` *The* `select` *verb for* `loci`

Description

An equivalent to `dplyr::select()` that works on the genotype column of a `gen_tibble`, using the mini-grammar available for `tidyselect`. The `select`-like evaluation only has access to the names of the loci (i.e. it can select only based on names, not summary statistics of those loci; look at `select_loci_if()` for that feature.

Usage

```
select_loci(.data, .sel_arg)
```

Arguments

`.data`	a `gen_tibble`
`.sel_arg`	one unquoted expression, using the mini-grammar of `dplyr::select()` to select loci. Variable names can be used as if they were positions in the data frame, so expressions like x:y can be used to select a range of variables.

Details

Note that the `select_loci` verb does not modify the backing FBM files, but rather it subsets the list of loci to be used stored in the `gen_tibble`.

Value

a `gen_tibble` with a subset of the loci.

See Also

`dplyr::select()`

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Select loci by name
example_gt_subset <- example_gt %>%
  select_loci(all_of(c("rs1", "rs2", "rs3")))
show_loci(example_gt_subset)

# Select loci by index
example_gt_subset <- example_gt %>% select_loci(all_of(c(4, 2, 1)))
```

```
show_loci(example_gt_subset)
```

`select_loci_if` *The* `select_if` *verb for* `loci`

Description

An equivalent to `dplyr::select_if()` that works on the `genotype` column of a `gen_tibble`. This function has access to the genotypes (and thus can work on summary statistics to select), but not the names of the loci (look at `select_loci()` for that feature.

Usage

```
select_loci_if(.data, .sel_logical)
```

Arguments

`.data`	a `gen_tibble`
`.sel_logical`	a logical vector of length equal to the number of loci, or an expression that will tidy evaluate to such a vector. Only loci for which .sel_logical is TRUE will be selected; NA will be treated as FALSE.

Details

Note that the `select_loci_if` verb does not modify the backing FBM files, but rather it subsets the list of loci to be used stored in the `gen_tibble`.

Value

a subset of the list of loci in the `gen_tibble`

See Also

`dplyr::select_if()`

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Select loci by chromosome
example_gt_subset <- example_gt %>%
  select_loci_if(loci_chromosomes(genotypes) == "chr2")
show_loci(example_gt_subset)

# Select loci by a summary statistic
example_gt_subset <- example_gt %>%
  select_loci_if(loci_maf(genotypes) > 0.2)
show_loci(example_gt_subset)
```

show_genotypes *Show the genotypes of a* gen_tibble

Description

Extract the genotypes (as a matrix) from a gen_tibble.

Usage

```
show_genotypes(.x, indiv_indices = NULL, loci_indices = NULL, ...)

## S3 method for class 'tbl_df'
show_genotypes(.x, indiv_indices = NULL, loci_indices = NULL, ...)

## S3 method for class 'vctrs_bigSNP'
show_genotypes(.x, indiv_indices = NULL, loci_indices = NULL, ...)
```

Arguments

.x	a vector of class vctrs_bigSNP (usually the genotype column of a gen_tibble object), or a gen_tibble.
indiv_indices	indices of individuals
loci_indices	indices of loci
...	currently unused.

Value

a matrix of counts of the alternative alleles (see show_loci()) to extract information on the alleles for those loci from a gen_tibble.

Examples

```
example_gt <- load_example_gt("gen_tbl")

example_gt %>% show_genotypes()
```

show_loci *Show the loci information of a* gen_tibble

Description

Extract and set the information on loci from a gen_tibble.

Usage

```
show_loci(.x, ...)

## S3 method for class 'tbl_df'
show_loci(.x, ...)

## S3 method for class 'vctrs_bigSNP'
show_loci(.x, ...)

show_loci(.x) <- value

## S3 replacement method for class 'tbl_df'
show_loci(.x) <- value

## S3 replacement method for class 'vctrs_bigSNP'
show_loci(.x) <- value
```

Arguments

.x	a vector of class vctrs_bigSNP (usually the genotype column of a gen_tibble object), or a gen_tibble.
...	currently unused.
value	a data.frame or tibble of loci information to replace the current one.

Value

a tibble::tibble of information (see gen_tibble for details on compulsory columns that will always be present)

Examples

```
example_gt <- load_example_gt("gen_tbl")

example_gt %>% show_loci()
```

---

show_ploidy *Show the ploidy information of a* gen_tibble

Description

Extract the ploidy information from a gen_tibble. NOTE that this function does not return the ploidy level for each individual (that is obtained with indiv_ploidy); instead, it returns an integer which is either the ploidy level of all individuals (e.g. 2 indicates all individuals are diploid), or a 0 to indicate mixed ploidy. The special case of -2 is used to indicate the presence of pseudo-haploids (i.e. individuals with a ploidy of 2 but for which we only have information for one allele; the dosages are 0 or 2 for these individuals).

Usage

```
show_ploidy(.x, ...)

## S3 method for class 'tbl_df'
show_ploidy(.x, ...)

## S3 method for class 'vctrs_bigSNP'
show_ploidy(.x, ...)
```

Arguments

.x	a vector of class vctrs_bigSNP (usually the genotype column of a gen_tibble object), or a gen_tibble.
...	currently unused.

Value

the ploidy (0 indicates mixed ploidy)

See Also

indiv_ploidy()

Examples

```
example_gt <- load_example_gt("gen_tbl")

example_gt %>% show_ploidy()
```

---

snp_allele_sharing *Compute the Pairwise Allele Sharing Matrix for a bigSNP object*

Description

This function computes the Allele Sharing matrix. Estimates Allele Sharing (matching in hierfstat)) between pairs of individuals (for each locus, gives 1 if the two individuals are homozygous for the same allele, 0 if they are homozygous for a different allele, and 1/2 if at least one individual is heterozygous. Matching is the average of these 0, 1/2 and 1s)

Usage

```
snp_allele_sharing(
  X,
  ind.row = bigstatsr::rows_along(X),
  ind.col = bigstatsr::cols_along(X),
  block.size = bigstatsr::block_size(nrow(X))
)
```

Arguments

X	a bigstatsr::FBM.code256 matrix (as found in the genotypes slot of a bigsnpr::bigSNP object).
ind.row	An optional vector of the row indices that are used. If not specified, all rows are used. Don't use negative indices.
ind.col	An optional vector of the column indices that are used. If not specified, all columns are used. Don't use negative indices.
block.size	maximum number of columns read at once. Note that, to optimise the speed of matrix operations, we have to store in memory 3 times the columns.

Value

a matrix of allele sharing between all pairs of individuals

See Also

pairwise_allele_sharing() hierfstat::matching()

Examples

```
example_gt <- load_example_gt("gen_tbl")

X <- attr(example_gt$genotypes, "fbm")
snp_allele_sharing(X)

# Compute for individuals 1 to 5
snp_allele_sharing(X, ind.row = 1:5, ind.col = 1:5)

# Adjust block size
snp_allele_sharing(X, block.size = 2)
```

---

snp_ibs *Compute the Identity by State Matrix for a bigSNP object*

Description

This function computes the IBS matrix.

Usage

```
snp_ibs(
  X,
  ind.row = bigstatsr::rows_along(X),
  ind.col = bigstatsr::cols_along(X),
  type = c("proportion", "adjusted_counts", "raw_counts"),
  block.size = bigstatsr::block_size(nrow(X))
)
```

Arguments

X	a bigstatsr::FBM.code256 matrix (as found in the genotypes slot of a bigsnpr::bigSNP object).
ind.row	An optional vector of the row indices that are used. If not specified, all rows are used. Don't use negative indices.
ind.col	An optional vector of the column indices that are used. If not specified, all columns are used. Don't use negative indices.
type	one of "proportion" (equivalent to "ibs" in PLINK), "adjusted_counts" ("distance" in PLINK), and "raw_counts" (the counts of identical alleles and non-missing alleles, from which the two other quantities are computed)
block.size	maximum number of columns read at once. Note that, to optimise the speed of matrix operations, we have to store in memory 3 times the columns.

Details

Note that monomorphic sites are currently counted. Should we filter them beforehand? What does plink do?

Value

if as.counts = TRUE function returns a list of two bigstatsr::FBM matrices, one of counts of IBS by alleles (i.e. 2*n loci), and one of valid alleles (i.e. 2 * n_loci - 2 * missing_loci). If as.counts = FALSE returns a single matrix of IBS proportions.

Examples

```
example_gt <- load_example_gt("gen_tbl")

X <- attr(example_gt$genotypes, "fbm")
snp_ibs(X)

# Compute for individuals 1 to 5
snp_ibs(X, ind.row = 1:5, ind.col = 1:5)

# Adjust block.size
snp_ibs(X, block.size = 2)

# Change type
snp_ibs(X, type = "proportion")
snp_ibs(X, type = "adjusted_counts")
snp_ibs(X, type = "raw_counts")
```

snp_king *Compute the KING-robust Matrix for a bigSNP object*

Description

This function computes the KING-robust estimator of kinship, reimplementing the KING algorithm of Manichaikul et al. (2010).

Usage

```
snp_king(
  X,
  ind.row = bigstatsr::rows_along(X),
  ind.col = bigstatsr::cols_along(X),
  block.size = bigstatsr::block_size(nrow(X)) * 4
)
```

Arguments

X	a bigstatsr::FBM.code256 matrix (as found in the genotypes slot of a bigsnpr::bigSNP object).
ind.row	An optional vector of the row indices that are used. If not specified, all rows are used. Don't use negative indices.
ind.col	An optional vector of the column indices that are used. If not specified, all columns are used. Don't use negative indices.
block.size	maximum number of columns read at once.

Value

a square symmetrical matrix of relationship coefficients between individuals

References

Manichaikul, A. et al. (2010) Robust relationship inference in genome-wide association studies. Bioinformatics, 26(22), 2867–2873. https://doi.org/10.1093/bioinformatics/btq559.

Examples

```
example_gt <- load_example_gt("gen_tbl")

X <- attr(example_gt$genotypes, "fbm")
snp_king(X)

# Compute for individuals 1 to 5
snp_king(X, ind.row = 1:5, ind.col = 1:5)

# Adjust block size
```

```
snp_king(X, block.size = 2)
```

---

summary.gt_admix *Summary method for gt_admix objects*

Description

Summary method for gt_admix objects

Usage

```
## S3 method for class 'gt_admix'
summary(object, ...)
```

Arguments

object	a gt_admix object
...	unused (necessary for compatibility with generic function)

Value

A summary of the gt_admix object

Examples

```
# run the example only if we have the package installed
if (requireNamespace("LEA", quietly = TRUE)) {
  example_gt <- load_example_gt("gen_tbl")

  # Create a gt_admix object
  admix_obj <- example_gt %>% gt_snmf(k = 1:3, project = "force")

  # Print a summary
  summary(admix_obj)
}
```

summary.rbind_report *Print a summary of a merge report*

Description

This function creates a summary of the merge report generated by rbind_dry_run()

Usage

```
## S3 method for class 'rbind_report'
summary(object, ..., ref_label = "reference", target_label = "target")
```

Arguments

object	a list generated by rbind_dry_run()
...	unused (necessary for compatibility with generic function)
ref_label	the label for the reference dataset (defaults to "reference")
target_label	the label for the target dataset (defaults to "target")

Value

NULL (prints a summary to the console)

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Create a second gen_tibble to merge
test_indiv_meta <- data.frame(
  id = c("x", "y", "z"),
  population = c("pop1", "pop1", "pop2")
)
test_genotypes <- rbind(
  c(1, 1, 0, 1, 1, 0),
  c(2, 1, 0, 0, 0, 0),
  c(2, 2, 0, 0, 1, 1)
)
test_loci <- data.frame(
  name = paste0("rs", 1:6),
  chromosome = paste0("chr", c(1, 1, 1, 1, 2, 2)),
  position = as.integer(c(3, 5, 65, 343, 23, 456)),
  genetic_dist = as.double(rep(0, 6)),
  allele_ref = c("A", "T", "C", "G", "C", "T"),
  allele_alt = c("T", "C", NA, "C", "G", "A")
)

test_gt <- gen_tibble(
  x = test_genotypes,
```

```
  loci = test_loci,
  indiv_meta = test_indiv_meta,
  valid_alleles = c("A", "T", "C", "G"),
  quiet = TRUE
)

# Merge the datasets using rbind
report <- rbind_dry_run(
  ref = example_gt, target = test_gt,
  flip_strand = TRUE, quiet = TRUE
)

# Get the summary
summary(report)
```

`theme_distruct` *A theme to match the output of distruct*

Description

A theme to remove most plot decorations, matching the look of plots created with distruct.

Usage

```
theme_distruct()
```

Value

a ggplot2::theme

Examples

```
# Read example gt_admix object
admix_obj <-
  readRDS(system.file("extdata", "anolis", "anole_adm_k3.rds",
    package = "tidypopgen"
  ))

# Basic barplot with disstruct theme
autoplot(admix_obj, k = 3, run = 1, type = "barplot") +
  theme_distruct()
```

`tidy.gt_dapc` *Tidy a* `gt_dapc` *object*

Description

This summarizes information about the components of a `gt_dapc` from the `tidypopgen` package. The parameter `matrix` determines which element is returned.

Usage

```
## S3 method for class 'gt_dapc'
tidy(x, matrix = "eigenvalues", ...)
```

Arguments

`x`	A `gt_dapc` object (as returned by `gt_dapc()`).
`matrix`	Character specifying which component of the DAPC should be tidied. • `"samples"`, `"scores"`, or `"x"`: returns information about the map from the original space into the least discriminant axes. • `"v"`, `"rotation"`, `"loadings"` or `"variables"`: returns information about the map from discriminant axes space back into the original space (i.e. the genotype frequencies). Note that this are different from the loadings linking to the PCA scores (which are available in the element $loadings of the dapc object). • `"d"`, `"eigenvalues"` or `"lds"`: returns information about the eigenvalues.
`...`	Not used. Needed to match generic signature only.

Value

A tibble::tibble with columns depending on the component of DAPC being tidied.

If `"scores"` each row in the tidied output corresponds to the original data in PCA space. The columns are:

`row`	ID of the original observation (i.e. rowname from original data).
`LD`	Integer indicating a principal component.
`value`	The score of the observation for that particular principal component. That is, the location of the observation in PCA space.

If `matrix` is `"loadings"`, each row in the tidied output corresponds to information about the principle components in the original space. The columns are:

`row`	The variable labels (colnames) of the data set on which PCA was performed.
`LD`	An integer vector indicating the principal component.
`value`	The value of the eigenvector (axis score) on the indicated principal component.

If "eigenvalues", the columns are:

LD	An integer vector indicating the discriminant axis.
std.dev	Standard deviation (i.e. sqrt(eig/(n-1))) explained by this DA (for compatibility with prcomp.
cumulative	Cumulative variation explained by principal components up to this component (note that this is NOT phrased as a percentage of total variance, since many methods only estimate a truncated SVD.

See Also

gt_dapc() augment.gt_dapc()

Examples

```
#' # Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

# Create PCA and run DAPC
pca <- gt_pca_partialSVD(lobsters)
populations <- as.factor(lobsters$population)
dapc_res <- gt_dapc(pca, n_pca = 6, n_da = 2, pop = populations)

# Tidy scores
tidy(dapc_res, matrix = "scores")

# Tidy eigenvalues
tidy(dapc_res, matrix = "eigenvalues")

# Tidy loadings
tidy(dapc_res, matrix = "loadings")
```

---

tidy.gt_pca *Tidy a* gt_pca *object*

Description

This summarizes information about the components of a gt_pca from the tidypopgen package. The parameter matrix determines which element is returned. Column names of the tidied output match those returned by broom::tidy.prcomp, the tidier for the standard PCA objects returned by stats::prcomp.

Usage

```
## S3 method for class 'gt_pca'
tidy(x, matrix = "eigenvalues", ...)
```

Arguments

x	A gt_pca object returned by one of the gt_pca_* functions.
matrix	Character specifying which component of the PCA should be tidied. • "samples", "scores", or "x": returns information about the map from the original space into principle components space (this is equivalent to product of u and d). • "v", "rotation", "loadings" or "variables": returns information about the map from principle components space back into the original space. • "d", "eigenvalues" or "pcs": returns information about the eigenvalues.
...	Not used. Needed to match generic signature only.

Value

A tibble::tibble with columns depending on the component of PCA being tidied.

If "scores" each row in the tidied output corresponds to the original data in PCA space. The columns are:

row	ID of the original observation (i.e. rowname from original data).
PC	Integer indicating a principal component.
value	The score of the observation for that particular principal component. That is, the location of the observation in PCA space.

If matrix is "loadings", each row in the tidied output corresponds to information about the principle components in the original space. The columns are:

row	The variable labels (colnames) of the data set on which PCA was performed.
PC	An integer vector indicating the principal component.
value	The value of the eigenvector (axis score) on the indicated principal component.

If "eigenvalues", the columns are:

PC	An integer vector indicating the principal component.
std.dev	Standard deviation (i.e. sqrt(eig/(n-1))) explained by this PC (for compatibility with prcomp.
cumulative	Cumulative variation explained by principal components up to this component (note that this is NOT phrased as a percentage of total variance, since many methods only estimate a truncated SVD.

See Also

gt_pca_autoSVD() augment_gt_pca

Examples

```
# Create a gen_tibble of lobster genotypes
bed_file <-
  system.file("extdata", "lobster", "lobster.bed", package = "tidypopgen")
lobsters <- gen_tibble(bed_file,
  backingfile = tempfile("lobsters"),
  quiet = TRUE
)

# Remove monomorphic loci and impute
lobsters <- lobsters %>% select_loci_if(loci_maf(genotypes) > 0)
lobsters <- gt_impute_simple(lobsters, method = "mode")

# Create PCA object
pca <- gt_pca_partialSVD(lobsters)

# Tidy the PCA object
tidy(pca)

# Tidy the PCA object for eigenvalues
tidy(pca, matrix = "eigenvalues")

# Tidy the PCA object for loadings
tidy(pca, matrix = "loadings")

# Tidy the PCA object for scores
tidy(pca, matrix = "scores")
```

---

tidy.q_matrix *Tidy a Q matrix*

Description

Takes a `q_matrix` object, which is a matrix, and returns a tidied tibble.

Usage

```
## S3 method for class 'q_matrix'
tidy(x, data, ...)
```

Arguments

`x`	A Q matrix object (as returned by `q_matrix`).
`data`	An associated tibble (e.g. a `gen_tibble`), with the individuals in the same order as the data used to generate the Q matrix
`...`	not currently used

Value

A tidied tibble containing columns:

row	ID of the original observation (i.e. rowname from original data).
Q	Integer indicating a Q component.
value	The proportion for that particular Q value.

Examples

```
# run the example only if we have the package installed
if (requireNamespace("LEA", quietly = TRUE)) {
  example_gt <- load_example_gt("gen_tbl")

  # Create a gt_admix object
  admix_obj <- example_gt %>% gt_snmf(k = 1:3, project = "force")

  # Extract a Q matrix
  q_mat_k3 <- get_q_matrix(admix_obj, k = 3, run = 1)

  tidy(q_mat_k3, data = example_gt)
}
```

---

windows_indiv_roh *Detect runs of homozygosity using a sliding-window approach*

Description

This function uses a sliding-window approach to look for runs of homozygosity (or heterozygosity) in a diploid genome. It is based on the package `detectRUNS`, which implements an approach equivalent to the one in PLINK.

Usage

```
windows_indiv_roh(
  .x,
  window_size = 15,
  threshold = 0.05,
  min_snp = 3,
  heterozygosity = FALSE,
  max_opp_window = 1,
  max_miss_window = 1,
  max_gap = 10^6,
  min_length_bps = 1000,
  min_density = 1/1000,
  max_opp_run = NULL,
  max_miss_run = NULL
)
```

```
gt_roh_window(
  .x,
  window_size = 15,
  threshold = 0.05,
  min_snp = 3,
  heterozygosity = FALSE,
  max_opp_window = 1,
  max_miss_window = 1,
  max_gap = 10^6,
  min_length_bps = 1000,
  min_density = 1/1000,
  max_opp_run = NULL,
  max_miss_run = NULL
)
```

Arguments

.x	a gen_tibble
window_size	the size of sliding window (number of SNP loci) (default = 15)
threshold	the threshold of overlapping windows of the same state (homozygous/heterozygous) to call a SNP in a RUN (default = 0.05)
min_snp	minimum n. of SNP in a RUN (default = 3)
heterozygosity	should we look for runs of heterozygosity (instead of homozygosity? (default = FALSE)
max_opp_window	max n. of SNPs of the opposite type (e.g. heterozygous snps for runs of homozygosity) in the sliding window (default = 1)
max_miss_window	max. n. of missing SNP in the sliding window (default = 1)
max_gap	max distance between consecutive SNP to be still considered a potential run (default = 10^6 bps)
min_length_bps	minimum length of run in bps (defaults to 1000 bps = 1 kbps)
min_density	minimum n. of SNP per kbps (defaults to 0.1 = 1 SNP every 10 kbps)
max_opp_run	max n. of opposite genotype SNPs in the run (optional)
max_miss_run	max n. of missing SNPs in the run (optional)

Details

This function returns a data frame with all runs detected in the dataset. The data frame is, in turn, the input for other functions of the detectRUNS package that create plots and produce statistics from the results (see plots and statistics functions in this manual, and/or refer to the detectRUNS vignette).

If the gen_tibble is grouped, then the grouping variable is used to fill in the 'group' column. Otherwise, the 'group' column is filled with the same values as the 'id' column. Note that this behaviour is different from other windowed operations in tidypopgen, which return a list for grouped gen_tibbles; this different behaviour is designed to maintain compatibility with detectRUNS.

The old name for this function, `gt_roh_window`, is still available, but it is soft deprecated and will be removed in future versions of `tidypopgen`.

Value

A dataframe with RUNs of Homozygosity or Heterozygosity in the analysed dataset. The returned dataframe contains the following seven columns: "group", "id", "chrom", "nSNP", "from", "to", "lengthBps" (group: population, breed, case/control etc.; id: individual identifier; chrom: chromosome on which the run is located; nSNP: number of SNPs in the run; from: starting position of the run, in bps; to: end position of the run, in bps; lengthBps: size of the run)

See Also

`detectRUNS::slidingRUNS.run()` which this function wraps.

Examples

```
sheep_ped <- system.file("extdata", "Kijas2016_Sheep_subset.ped",
  package = "detectRUNS"
)
sheep_gt <- tidypopgen::gen_tibble(sheep_ped,
  backingfile = tempfile(),
  quiet = TRUE
)
sheep_gt <- sheep_gt %>% group_by(population)
sheep_roh <- windows_indiv_roh(sheep_gt)
detectRUNS::plot_Runs(runs = sheep_roh)
```

`windows_nwise_pop_pbs` *Compute the Population Branch Statistics over a sliding window*

Description

The function computes the population branch statistics (PBS) for a sliding window for each combination of populations at each locus. The PBS is a measure of the genetic differentiation between one focal population and two reference populations, and is used to identify outlier loci that may be under selection.

Usage

```
windows_nwise_pop_pbs(
  .x,
  type = c("matrix", "tidy"),
  fst_method = c("Hudson", "Nei87", "WC84"),
  return_fst = FALSE,
  window_size,
  step_size,
```

```
  size_unit = c("snp", "bp"),
  min_loci = 1,
  complete = FALSE
)
```

Arguments

.x	a grouped gen_tibble object
type	type of object to return. One of "matrix" or "tidy". Default is "matrix". "matrix" returns a dataframe where each row is a window, followed by columns of pbs values for each population comparison. "tidy" returns a tidy tibble of the same data in 'long' format, where each row is one window for one population comparison.
fst_method	the method to use for calculating Fst, one of 'Hudson', 'Nei87', and 'WC84'. See pairwise_pop_fst() for details.
return_fst	a logical value indicating whether to return the Fst values
window_size	The size of the window to use for the estimates.
step_size	The step size to use for the windows.
size_unit	Either "snp" or "bp". If "snp", the window size and step size are in number of SNPs. If "bp", the window size and step size are in base pairs.
min_loci	The minimum number of loci required to calculate a window statistic. If the number of loci in a window is less than this, the window statistic will be NA.
complete	Should the function be evaluated on complete windows only? If FALSE, the default, then partial computations will be allowed at the end of the chromosome.

Value

either a data frame with the following columns:

- chromosome: the chromosome for the window
- start: the starting locus of the window
- end: the ending locus of the window
- pbs_a.b.c: the PBS value for population a given b & c (there will be multiple such columns covering all 3 way combinations of populations in the grouped gen_tibble object)
- fst_a.b: the Fst value for population a and b, if return_fst is TRUE or a tidy tibble with the following columns:
- chromosome: the chromosome for the window
- start: the starting locus of the window
- end: the ending locus of the window
- stat_name: the name of populations used in the pbs calculation (e.g. "pbs_pop1.pop2.pop3"). If return_fst is TRUE, stat_name will also include "fst" calculations in the same column (e.g. "fst_pop1.pop2").
- value: the pbs value for the populations

Examples

```
example_gt <- load_example_gt("grouped_gen_tbl")

# Calculate nwise pbs across a window of 3 SNPs, with a step size of 2 SNPs
example_gt %>%
  windows_nwise_pop_pbs(
    window_size = 3, step_size = 2,
    size_unit = "snp", min_loci = 2
  )
```

`windows_pairwise_pop_fst` *Compute pairwise Fst for a sliding window*

Description

This function computes pairwise Fst for a sliding window across each chromosome.

Usage

```
windows_pairwise_pop_fst(
  .x,
  type = c("matrix", "tidy"),
  method = c("Hudson", "Nei87", "WC84"),
  window_size,
  step_size,
  size_unit = c("snp", "bp"),
  min_loci = 1,
  complete = FALSE
)
```

Arguments

`.x`	a grouped `gen_tibble` object
`type`	type of object to return. One of "matrix" or "tidy". Default is "matrix". "matrix" returns a dataframe where each row is a window, followed by columns of Fst values for each pairwise population a and b comparison. "tidy" returns a tidy tibble of the same data in 'long' format, where each row is one window for one pairwise population a and b comparison.
`method`	the method to use for calculating Fst, one of 'Hudson', 'Nei87', and 'WC84'. See `pairwise_pop_fst()` for details.
`window_size`	The size of the window to use for the estimates.
`step_size`	The step size to use for the windows.
`size_unit`	Either "snp" or "bp". If "snp", the window size and step size are in number of SNPs. If "bp", the window size and step size are in base pairs.

min_loci	The minimum number of loci required to calculate a window statistic. If the number of loci in a window is less than this, the window statistic will be NA.
complete	Should the function be evaluated on complete windows only? If FALSE, the default, then partial computations will be allowed at the end of the chromosome.

Value

either a data frame with the following columns:

- chromosome: the chromosome for the window
- start: the starting locus of the window
- end: the ending locus of the window
- fst_a.b: the pairwise Fst value for the population a and b (there will be multiple such columns if there are more than two populations) or a tidy tibble with the following columns:
- chromosome: the chromosome for the window
- start: the starting locus of the window
- end: the ending locus of the window
- stat_name: the name of population a and b used in the pairwise Fst calculation (e.g. "fst_pop1.pop2")
- value: the pairwise Fst value for the population a and b

Examples

```
example_gt <- load_example_gt("gen_tbl")

example_gt %>%
  group_by(population) %>%
  windows_pairwise_pop_fst(
    window_size = 3, step_size = 2,
    size_unit = "snp", min_loci = 2
  )
```

---

windows_pop_tajimas_d *Compute Tajima's D for a sliding window*

Description

This function computes Tajima's D for a sliding window across each chromosome.

Usage

```
windows_pop_tajimas_d(
  .x,
  type = c("matrix", "tidy", "list"),
  window_size,
  step_size,
  size_unit = c("snp", "bp"),
  min_loci = 1,
  complete = FALSE
)
```

Arguments

.x	a (potentially grouped) gen_tibble object
type	type of object to return, if using grouped method. One of "matrix", "tidy", or "list". Default is "matrix".
window_size	The size of the window to use for the estimates.
step_size	The step size to use for the windows.
size_unit	Either "snp" or "bp". If "snp", the window size and step size are in number of SNPs. If "bp", the window size and step size are in base pairs.
min_loci	The minimum number of loci required to calculate a window statistic. If the number of loci in a window is less than this, the window statistic will be NA.
complete	Should the function be evaluated on complete windows only? If FALSE, the default, then partial computations will be allowed at the end of the chromosome.

Value

if data is not grouped, a data frame with the following columns:

- chromosome: the chromosome for the window
- start: the starting locus of the window
- end: the ending locus of the window
- tajimas_d: the Tajima's D for the population if data are grouped, either: a data frame as above with the following columns:
- chromosome: the chromosome for the window
- start: the starting locus of the window
- end: the ending locus of the window
- n_loci: the number of loci in the window
- group: the Tajima's D for the group for the given window (there will be as many of these columns as groups in the gen_tibble, and they will be named by the grouping levels) a tidy tibble with the following columns:
- chromosome: the chromosome for the window
- start: the starting locus of the window

- end: the ending locus of the window
- n_loci: the number of loci in the window
- group: the name of the group
- stat: the Tajima's D for the given group at the given window or a list of data frames, one per group, with the following columns:
- chromosome: the chromosome for the window
- start: the starting locus of the window
- end: the ending locus of the window
- stat: the Tajima's D for the given window
- n_loci: the number of loci in the window

Examples

```
example_gt <- load_example_gt("grouped_gen_tbl")

# Calculate Tajima's D across a window of 3 SNPs, with a step size of 2 SNPs
example_gt %>%
  windows_pop_tajimas_d(
    window_size = 3, step_size = 2,
    size_unit = "snp", min_loci = 2
  )
```

---

windows_stats_generic *Estimate window statistics from per locus estimates*

Description

This function is mostly designed for developers: it is a general function to estimate window statistics from per locus estimates. This function takes a vector of per locus estimates, and aggregates them by sum or mean per window. To compute specific quantities directly from a gen_tibble, use the appropriate window_* functions, e.g windows_pairwise_pop_fst() to compute pairwise Fst.

Usage

```
windows_stats_generic(
  .x,
  loci_table,
  operator = c("mean", "sum", "custom"),
  window_size,
  step_size,
  size_unit = c("snp", "bp"),
  min_loci = 1,
  complete = FALSE,
  f = NULL,
  ...
)
```

Arguments

`.x`	A vector containing the per locus estimates.
`loci_table`	a dataframe including at least a column 'chromosome', and additionally a column 'position' if `size_unit` is "bp".
`operator`	The operator to use for the window statistics. Either "mean", "sum" or "custom" to use a custom function `.f`.
`window_size`	The size of the window to use for the estimates.
`step_size`	The step size to use for the windows.
`size_unit`	Either "snp" or "bp". If "snp", the window size and step size are in number of SNPs. If "bp", the window size and step size are in base pairs.
`min_loci`	The minimum number of loci required to calculate a window statistic. If the number of loci in a window is less than this, the window statistic will be NA.
`complete`	Should the function be evaluated on complete windows only? If FALSE, the default, then partial computations will be allowed at the end of the chromosome.
`f`	a custom function to use for the window statistics. This function should take a vector of locus estimates and return a single value.
`...`	Additional arguments to be passed to the custom operator function.

Value

A tibble with columns: 'chromosome', 'start', 'end', 'stats', and 'n_loci'. The 'stats' column contains the mean of the per locus estimates in the window, and 'n_loci' contains the number of loci in the window.

Examples

```
example_gt <- load_example_gt("gen_tbl")

miss_by_locus <- loci_missingness(example_gt)

# Calculate mean missingness across windows
windows_stats_generic(miss_by_locus,
  loci_table = show_loci(example_gt),
  operator = "mean", window_size = 1000,
  step_size = 1000, size_unit = "bp",
  min_loci = 1, complete = FALSE
)
```

`$<-.gen_tbl` *A $ method for* `gen_tibble` *objects*

Description

A $ method for `gen_tibble` objects

Usage

```
## S3 replacement method for class 'gen_tbl'
x$i <- value
```

Arguments

`x`	a gen_tibble
`i`	column name
`value`	a value to assign

Value

a `gen_tibble`

Examples

```
example_gt <- load_example_gt("gen_tbl")

# Add a new column
example_gt$region <- "East"

example_gt
```

Index