Package ‘gscramble’

February 28, 2024

Type Package

Title Simulating Admixed Genotypes Without Replacement

Version 1.0.1

Description A genomic simulation approach for creating biologically informed individual genotypes from empirical data that 1) samples alleles from populations without replacement, 2) segregates alleles based on species-specific recombination rates. 'gscramble' is a flexible simulation approach that allows users to create pedigrees of varying complexity in order to simulate admixed genotypes. Furthermore, it allows users to track haplotype blocks from the source populations through the pedigrees.

License CC0

Encoding UTF-8

LazyData true

RoxygenNote 7.2.3

Depends R (>= 3.5.0)

Imports dplyr, ggplot2, glue, magrittr, purrr, readr, rlang, stats, stringr, tibble, tidyr

Suggests cowplot, knitr, rmarkdown, tidyverse

VignetteBuilder knitr

URL `https://github.com/eriqande/gscramble`

BugReports `https://github.com/eriqande/gscramble/issues`

NeedsCompilation no

Author Eric C. Anderson [aut, cre] (<`https://orcid.org/0000-0003-1326-0840`>), Rachael M. Giglio [aut] (<`https://orcid.org/0000-0003-4183-3546`>), Matt G. DeSaix [aut] (<`https://orcid.org/0000-0002-5721-0311`>), Timothy J. Smyser [aut] (<`https://orcid.org/0000-0003-4542-3077`>)

Maintainer Eric C. Anderson <`eric.anderson@noaa.gov`>

Repository CRAN

Date/Publication 2024-02-28 19:10:05 UTC

R topics documented:

`check_chrom_lengths` *check that the chromosome lengths are acceptable given recombination rates*

Description

For 'gscramble' to work properly, all variant positions on a chromosome (found in the meta data file) must be equal to or less than the total chromosome length found in the recombination map. In other words, the variant positions must be within the the total length of each chromosome. The check_chrom_lengths() function checks that variant positions on each chromosome do not exceed the total chromosome length from the recombination map. Input files for the function include 1) the meta data file which contains 3 columns (the chromosome ID as a character, the position of the variant as a numeric, and the name of the variant as a character) and 2) the recombination map which contains 5 columns (the chromosome ID as a character, the total length of the chromosome as a numeric, the starting position of the recombination bin as a numeric, the last position of the recombination bin as a numeric, and the recombination probability for the given bin as a numeric).

Usage

```
check_chrom_lengths(meta, rec)
```

Arguments

`meta`	a tibble with meta data for the genotype data. It must consist of the columns (chrom, pos, variant_id).
`rec`	a tibble with recombination map information for your study species.It must contain 5 columns (chrom, chrom_len, start_pos, end_pos, rec_prob).

Value

This function will return a message for each chromosome stating whether the chromosome lengths are accurate

Examples

```
# The example uses the built in datasets M_meta and RecRates
# to use the check_chrom_lengths() function

check_chrom_lengths(M_meta,RecRates)
```

`check_gsp_for_validity_and_saturation`

Check that GSP does not reuse genetic material and yet uses all of it

---

Description

These conditions can be checked for a GSP with no inbreeding loops simply by ensuring that the amount of genetic material coming into each individual is the same as the amount going out (either as segregated gametes or as samples). If the amount of material coming out of any individual in the GSP is greater than the amount coming in, then an error is thrown. If the amount coming out is less than the amount coming in, then a warning about the GSP is thrown. Messages printed via `message()` and `warning()` indicate which individuals in the GSP are problematic. All problematic individuals are listed before an error is thrown with `stop()`.

Usage

```
check_gsp_for_validity_and_saturation(GP)
```

Arguments

GP	A gsp in list format as produced by the function `prep_gsp_for_hap_dropping()`. See the documentation for the return object of `prep_gsp_for_hap_dropping()` for a description.

Value

This function does not return anything.

Examples

```
# get the 13 member pedigree in tibble form as the package
# and turn it into a list
GP <- prep_gsp_for_hap_dropping(GSP)

# check it. (This passes)
check_gsp_for_validity_and_saturation(GP)

## The following will show a failure, so we wrap it in tryCatch
## so CRAN check does not flag it as a problem.
# Read in a gsp with errors and then make sure all the
# error in it are caught
bad <- readr::read_csv(system.file("extdata/13-member-ped-with-errors.csv", package = "gscramble"))

# check_gsp_for_validity_and_saturation() is called internally from
# within prep_gsp_for_hap_dropping(), after creating a list-from GSP.
# This will show the error produced by check_gsp_for_validity_and_saturation().
badL <- tryCatch(
  prep_gsp_for_hap_dropping(bad),
  error = function(x) 0,
  warning = function(x) 0
)
```

`check_pedigree_for_inbreeding`
Check the a GSP (in prepped list format) for inbreeding loops

---

Description

After a GSP has been passed through `prep_gsp_for_hap_dropping()` it is in a list format with the individuals ordered in such a way that it should be easy to check for any inbreeding loops in it (which are not allowed!). This version uses a simple recursive approach to compute the ancestry vector for each individual, and it detects inbreeding by the occurrence of the same ID in the ancestry vector more than once. This might be slow on large pedigrees, but for most that people would use, this should be fine.

Usage

```
check_pedigree_for_inbreeding(GP)
```

Arguments

GP	A gsp in list format as produced by the function `prep_gsp_for_hap_dropping()`. See the documentation for the return object of `prep_gsp_for_hap_dropping()` for a description.

Details

Note that the ancestry vector produced by this is not ordered the way the ancestry vectors are in my package CKMRpop—for simplicity I just get a list of ancestors in whatever order they happen to be reached.

Value

This function does not return anything. It throws an error via `stop()` if inbreeeding loops are found in the pedigree. Before throwing that error it lists the individuals with repeated occurrences in their ancestry vectors via the `message()` function.

Examples

```
# get the 13 member pedigree in the data object GSP and
# turn it into a list
GP <- prep_gsp_for_hap_dropping(GSP)

# check it for inbreeding. (There is none)
check_pedigree_for_inbreeding(GP)

## This one will fail, so we wrap it in tryCatch so CRAN
## check doesn't find it a problem.
# To see what happens if there are inbreeding loops, make some
GP_inbred <-  GP
# make 12 be inbred trough individual 6
GP_inbred$`12`$par1$par = "13"
# make 8 inbred (because both of its founder parents are the same!)
GP_inbred$`8`$par2$par = "4"

# now try that:
tryCatch(
  check_pedigree_for_inbreeding(GP_inbred),
  error = function(x) 0,
  warning = function(x) 0
)
```

computeQs_from_segments

return the admixture fractions of sampled individuals

Description

This operates on the output of segregate to add up the lengths of all the segments segregated to different individuals to thus compute the admixture fractions of each sampled individual.

Usage

```
computeQs_from_segments(S, check_total_length = TRUE)
```

Arguments

`S`	the tibble output from `segregate`
`check_total_length`	TRUE means it checks the total genome length in each individual to make sure it checks out.

Value

This function returns a tibble with the following columns:

- `gpp`: the genomic simulation pedigree within which the individual sample was simulated.
- `index`: the index which gives which instance of the GSP the sample is from
- `ped_sample_id`: the id number of that the sampled individual had in the genomic simulation pedigree.
- `samp_index`: the index of the sample taken. Some individuals in some genomic simulation pedigrees can produce more than one sample. This number tells you which sample it is.
- `pop_origin`: the "pedigree" population of origin of the segments that contributed to the `group_length`. These are the simple "A", "B", "C", etc. designations given in the genomic simulation pedigree.
- `group_origin`: Which group of samples the segments contributing to the `group_length` originated from. These are the groups of samples that were mapped onto the simple pedigree `pop_origins` by the reppop request.
- `group_length`: the total length of segments from this group in this individual in this reppop index from this gpp (in bases).
- `tot_length`: the total number of bases from all origins carried by this individual.
- `admixture_fraction`: the fraction of all bases in the simulated individual that originate from the group in `group_origin`.

Examples

```
#### Get output from segregate to use as input ####
# We construct an example here where we will request segregation
# down a GSP with two F1s and F1B backcrosses between two hypothetical
# populations, A and B.
gsp_f1f1b <- create_GSP("A", "B", F1 = TRUE, F1B = TRUE)

# We will imagine that in our marker data there are three groups
# labelled "grp1", "grp2", and "grp3", and we want to create the F1Bs with backcrossing
# only to grp3.
reppop <- tibble::tibble(
  index = as.integer(c(1, 1, 2, 2)),
  pop = c("A", "B", "A", "B"),
  group = c("grp3", "grp1", "grp3", "grp2")
)

# combine those into a request
request <- tibble::tibble(
  gpp = list(gsp_f1f1b),
  reppop = list(reppop)
)

# now run it through segregate()
set.seed(5)  # just for reproducibility in example...
simSegs <- segregate(request, RecRates)

#### Now we can run those through computeQs_from_segments() ####
Qs <- computeQs_from_segments(simSegs)

Qs
```

---

create_GSP — *Create a GSP from user inputs about what type of hybrids from which populations to create hybrids*

Description

This function allows the user to choose two populations and create a GSP input for 'gscramble'. The input requires two different population IDs of class character as well as at least one TRUE statement for one of the hybrid parameters (F1, F2, F1B, F1B2). The GSP will indicate hybrid individuals that will be sampled based on which F1, F2, F1B, and F1B2 parameters are TRUE. F1= TRUE means F1 hybrids will be sampled. F2=TRUE means F2 hybrids will be sampled. F1B2=TRUE means F1 backcross hybrids will be sampled. F1B2=TRUE means F1 backcross 2 hybrids will be sampled. Default setting for these parameters is FALSE. The function then outputs a GSP in tibble format that can be used for other functions in 'gscramble' including `check_pedigree_for_inbreeding()` and `prep_gsp_for_hap_dropping()`.

Usage

```
create_GSP(
  pop1,
  pop2,
  F1 = FALSE,
  F2 = FALSE,
  F1B = FALSE,
  F1B2 = FALSE,
  AllowSinglePop = FALSE
)
```

Arguments

`pop1`	character name for population 1
`pop2`	character name for population 2
`F1`	logical indicating whether you would like to have sampled F1 hybrids in the output.
`F2`	logical indicating whether you would like to have sampled F2 hybrids in the output.
`F1B`	logical indicating whether you would like to have sampled F1 backcross hybrids in the output.
`F1B2`	logical indicating whether you would like to have sampled F1 backcross 2 hybrids in the output.
`AllowSinglePop`	Set to true if you want all the founders from just a single population. This has some use cases...

Value

This function returns a GSP in tibble format with the user argument for pop1 and pop2 autopopulated in the hpop1 and hpop2 columns.

Examples

```
# create a GSP that generates hybrids and samples F1s and F1B's from pops A and B
gsp <- create_GSP("A", "B", F1 = TRUE, F1B = TRUE)

#create a GSP that generates hybrids and samples F1s, F2s, F1Bs, F1B2s from pops A and B
gsp <- create_GSP("Pop_X", "Pop_Z", F1 = TRUE, F2 = TRUE, F1B = TRUE, F1B2 = TRUE)
```

`drop_segs_down_gsp` *High level function for dropping segments down a GSP*

Description

This one asks for the number of reps to do, and it also automatically does it over chromosomes and returns the results in a nice tidy tibble.

Usage

```
drop_segs_down_gsp(GSP, RR, Reps)
```

Arguments

`GSP`	the pedigree to use for the simulation, in the format of the package data `GSP`.
`RR`	the recombination rates in the format of the package data `RecRates`
`Reps`	the number of times to do the simulation. Different replicates are denoted by the index column in the output tibble.

Value

The output from this function is a tibble. Each row represents one segment of genetic material amongst the sampled individuals from the genomic permutation pedigrees. The columns give information about the provenance and destination of that segment as follows. Each segment exists in one of the samples (`samp_index`) from a sampled individual with a `ped_sample_id` in a given `samp_index` within the individual. Further, it is on one of two gametes (`gamete_index`) that segregated into the individual, and it came from a certain founding population (`pop_origin`). And, of course, the segment occupies the space from `start` to `end` on a chromosome `chrom`. Finally, the index of the founder haplotype on the given gpp that this segement descended from is given in `rs_founder_haplotype` which is short for "rep-specific founder haplotype". This final piece of information is crucial for segregating variation from the individuals in the `Geno` file onto these segments. The `gamete_segments` column is a list column with duplicated entries for each chromosome in an individual.

Examples

```
simSegs <- drop_segs_down_gsp(GSP, RecRates, 4)
```

`example_chrom_lengths` *Lengths of the three chromosomes used in the example data set*

Description

These are for the example of how to use `plink_map2rec_rates()`.

Format

A tibble with two columns: `chrom` and `bp`.

Source

These lengths were taken from the maximal values in Tortereau, Flavie, et al. "A high density recombination map of the pig reveals a correlation between sex-specific recombination and GC content." BMC genomics 13.1 (2012): 1-12. It is available for download from `https://static-content.springer.com/esm/art%3A10.1186%2F1471-2164-13-586/MediaObjects/12864_2012_4363_MOESM1_ESM.txt`. After downloading the data were processed to remove inconsistencies with the marker data set used for M_meta and Geno.

`example_segments` *Example of a segments tibble*

Description

This is the Segments object from the 'gscramble' tutorial. It is included as a data object to use in the example for the function `plot_simulated_chromosome_segments()`.

Format

A tibble like that produced by the `segregate()` function.

Source

Made from package functions

Geno *Genotype matrix of 78 individuals and 100 SNP markers*

Description

This dataset represents 3 distinct populations of feral swine in the United States. To make this dataset computationally efficient, only 3 chromosomes (12, 17, 18) from the pig genome were used. Further, loci were reduced to the 100 most informative loci for distinguishing the 3 populations. Each row of the genotype matrix includes the genotypes for a single individual. The genotype matrix is in a standard "two-column" format for a diploid species where two adjacent columns make up a locus and each column of a locus contains an allele. Genotype data is stored in the character class. Missing data is represented by NAs. Individual IDs and population assignment information can be found in `I_meta` Locus and chromosome information for genotypes can be found in `M_meta`

Format

A character matrix. Each row represents an individual and columns contain the genotypes for individuals in a "two-column" format where two adjacent columns make up a locus with an allele in each column.

Source

USDA-APHIS-WS-National Wildlife Research Center Feral Swine Genetic Archive

gscramble2newhybrids *Convert 'gscramble' output to newhybrids format*

Description

This function turns character-based alleles into integers and writes the necessary headers, etc. It preferentially uses the "id" column if it exists in `M$ret_ids`. Otherwise it uses the `indiv` column for the sample names.

Usage

```
gscramble2newhybrids(
  M,
  M_meta,
  z = NULL,
  s = NULL,
  retain = NULL,
  outfile = tempfile()
)
```

Arguments

`M`	the output from `segments2markers()` from 'gscramble'. This could have an added `id` column on it, which will then be used for the sample names.
`M_meta`	the Marker meta data file.
`z`	A vector of length two. The values are regular expressions that the sample names that you want to have -z 0 or -z 1 should match. For example `c("SH", "CCT")` means any sample matching "SH" would get z0 and any sample matchine "CCT" would get z1.
`s`	a single regular expressions that matches individuals that should be given the -s option. For example "SH|CCT"
`retain`	a vector of loci to retain.
`outfile`	path to the file to write the newhybrids data set to. For CRAN compliance, this is, by default, a temp file. But you can change it to be anything valid.

Details

It allows you to set the -s and -z through some regular expression mapping.

This function relies a lot on some tidyverse functions for pivoting, etc. As such, it is not intended for data sets with tens of thousands of markers. You oughtn't be using NewHybrids with so many markers, anyway!

Value

A list with three components:

- `outfile`: outfile name of saved data.
- `genos`: Genotypes
- `allele_names`: Allele names.

Examples

```
# get output from segments2markers():
example("segments2markers")
# copy that result to a new variable
M <- s2m_result

# then run it
gscramble2newhybrids(M, M_meta)
```

gscramble2plink — *Write 'gscramble' I_meta, M_meta, and Geno to a plink file*

Description

Writes genetic and individual information in 'gscramble's I_meta, M_meta, and Geno like objects into a uncompressed plink `.ped` and `.map` files.

Usage

```
gscramble2plink(I_meta, M_meta, Geno, prefix = tempfile())
```

Arguments

`I_meta`	a tibble of individual meta data with at least the columns of `group` and `indiv`.
`M_meta`	a tibble of marker meta data with at least the columns of `chrom`, `pos`, and `variant_id`.
`Geno`	a character matrix of genotypes. Num-indivs rows and num-markers * 2 columns, with missing denoted denoted by NA.
`prefix`	the file path and prefix into which to write out the files

Value

Returns TRUE if successful.

Examples

```
gscramble2plink(I_meta, M_meta, Geno)
```

GSP — *Example Genomic Simulation Pedigree, GSP, with 13 members*

Description

A GSP is a pedigree with no inbreeding which specifies This is a relatively complex GSP. The tibble `GSP` specifies its structure using the following columns:

- `ind`: the numeric identifier for the individual specific to the row (we will call that the "focal individual").
- `par1`: numeric identifier of the first parent of the focal individual. Must be NA for pedigree founders.
- `par2`: numeric identifier of the second parent of the focal individual. Must be NA for pedigree founders.

- `ipar1`: the number of gametes that will be incoming from the first parent to the focal individual. Must be NA for pedigree founders. Note that this must be equal to `ipar2`.
- `ipar2`: the number of gametes that will be incoming from the second parent to the focal individual. Must be NA for pedigree founders. Note that this must be equal to `ipar1`.
- `hap1`: character name for the first haplotype of a founder (if the focal individual is a founder). Must be NA for pedigree non-founders.
- `hap2`: character name for the second haplotype of a founder (if the focal individual is a founder). Must be NA for pedigree non-founders.
- `hpop1`: character ID of the population from which haplotype 1 comes from. Must be NA for pedigree non-founders.
- `hpop2`: character ID of the population from which haplotype 2 comes from. Must be NA for pedigree non-founders.
- `sample`: character ID for the sample from the focal individual. NA if no samples are taken from the focal individual, and must be NA for any pedigree founders.
- `osample`: numeric value giving the number or samples that are taken from this individual. `osample` must be less than or equal to `ipar1` and `ipar2`. If `osample` is less than `ipar1` and `ipar2`, then some gametes must get passed on to descendants of the focal individual.

Format

A tibble

Details

The CSV version of this is in `extdata/13-member-ped.csv`.

Source

Created by the developers.

`gsp2dot` *Write a dot file to represent a genome simulation pedigree*

Description

This takes the tibble representation of a GSP and writes it to a dot file to be rendered into a graph using the dot command from the GraphViz package. You can easily get GraphViz using Miniconda or check out the GraphViz downloads page. If you have the dot executable in your PATH, then dot will be run on the dot file and an SVG and a PNG image of the graph.

Usage

```
gsp2dot(
  g,
  path = file.path(tempfile(), "file_prefix"),
  edge_label_font_size = 18,
  indiv_node_label_font_size = 18,
  sample_node_label_font_size = 18,
  haplo_origin_colors = c("lightblue", "orange", "blue", "green", "cadetblue",
   "dodgerblue3", "darkolivegreen1", "forestgreen", "lightpink", "red2", "sandybrown",
     "orangered", "plum3", "purple4", "palegoldenrod", "peru"),
  sam_node_color = "violet",
  sample_edge_label_color = "purple",
  parent_edge_label_color = "red"
)
```

Arguments

`g`	a GSP tibble.
`path`	the path to the file prefix to use (to this will be appended .dot, and .png or .svg, if dot is on your system). By default these paths are in a temporary directory, because packages are not allowed to write to user home directories by default. Supply a path with prefix, like `my_path/myfile` to get the output file `mypath.myfile.dot`
`edge_label_font_size`	The font size of the number annotations along the edges.
`indiv_node_label_font_size`	the font size of the labels for the individual nodes
`sample_node_label_font_size`	the font size of the labels for the individual nodes
`haplo_origin_colors`	The colors for different origins of haplotypes. By default there are only sixteen. If you have more populations that founders may come from, you should provide a vector with more than 16 colors.
`sam_node_color`	The color given to the sample nodes in the GSP.
`sample_edge_label_color`	Color for the numeric annotations along the edges leading to samples.
`parent_edge_label_color`	Color for the numeric annotations along the edges leading from parents to offspring.

Details

It can be tricky knowing whether or not R or Rstudio will read your Unix rc files and populate your paths appropriately. If you want to test whether `dot` in on your PATH as it is when running under R, try: `Sys.which("dot")` at your R console. If that returns an empty string, (`""`), then you need to do something else to make sure R can find `dot` on your system.

Value

A vector of file paths. The first is the path of the dot file that was produced. The second and third are only present if dot was found in the PATH. They are the paths of the png and svg files that were produced.

Examples

```
gsp_file <- system.file("extdata/13-member-ped.csv", package = "gscramble")
g <- readr::read_csv(gsp_file)

paths <- gsp2dot(g)
paths
```

---

gsp3 *Tibble holding specification for a 5 member genomic permutation pedigree.*

Description

This has 3 founders

Format

A tibble

Details

For details on the columns, see the documentation for GSP.

Source

Created by the developers.

---

gsp4 *Tibble holding specification for a 7 member genomic permutation pedigree.*

Description

This has 4 founders, each one from a different population, and it provides four samples that are the product of F1 matings (an A x B F1 crossing with a C x D F1.)

Format

A tibble

Details

For details on the columns, see the documentation for `GSP`.

Source

Created by the developers.

`GSP_opts` *A list of tibbles specifying the pedigrees available from* `createGSP()`

Description

This is a list that is used by the function `createGSP()`. There are 15 different genomic simulation pedigrees, specified as tibbles, in this list.

Format

A list of 15 tibbles

Source

Written by package authors Rachael and Tim

`I_meta` *Metadata for 78 individuals*

Description

The tibble 'I_meta' contains the Individuals IDs and group specifications for 78 individuals. Each row of 'I_meta' contains the metadata for an individual. 'I_meta' has two columns, a 'group' column with the group or population assignment for a given individual and an 'indiv' column with the individual sample IDs. The information in each column are stored as characters. The order of the individual IDs corresponds to the genotypes found in `Geno`.

Format

A tibble with two columns: group and indiv.

Source

USDA-APHIS-WS-National Wildlife Research Center Feral Swine Genetic Archive

mat_scramble *Scramble a matrix of genotype data*

Description

This function assumes that M is a matrix with L rows (number of markers) and 2 * N (N = number of individuals) columns. There are two ways that the data might be permuted. In the first, obtained with `preserve_haplotypes = FALSE`, the position of missing data within the matrix is held constant, but all non-missing sites within a row (i.e. all gene copies at a locus) get scrambled amongst the samples. In the second way, just the columns are permuted. This preserves haplotypes in the data, if there are any. The second approach should only be used if haplotypes are inferred in the individuals.

Usage

```
mat_scramble(
  M,
  preserve_haplotypes = FALSE,
  row_groups = NULL,
  preserve_individuals = FALSE
)
```

Arguments

`M`	a matrix with L rows (number of markers) and 2 * N columns where N is the number of individuals. Missing data must be coded as NA
`preserve_haplotypes`	logical indicating whether the haplotypes set to be TRUE
`row_groups`	if not NULL must be a list of indexes of adjacent rows that are all in the same groups. For example: `list(1:10, 11:15, 16:30)`. They should be in order and complete. In practice, these should correspond to the indexes of markers on different chromosomes.
`preserve_individuals`	logical indicating whether the genes within each individual should stay togeter.

Details

There is now an additional way of permuting: if `preserve_individuals = TRUE`, then entire individuals are permuted. If `preserve_haplotypes = FALSE`, then the gene copies at each locus are randomly ordered within each individual before permuating them. If `preserve_haplotypes = TRUE` then that initial permutation is not done. This should only be done if the individuals are phased and that phasing is represented in how the genotypes are stored in the matrix.

Value

This function returns a matrix of the same dimensions and storage.mode as the input, `M`; however the elements have been permuted according to the options specified by the users.

Examples

```
# make a matrix with alleles named as I.M.g, where I is individual
# number, M is marker number, and g is either "a" or "b" depending
# on which gene copy in the diploid it is.  4 indivs and 7 markers...
Mat <- matrix(
 paste(
   rep(1:4, each = 7 * 2),
   rep(1:7, 4 * 2),
   rep(c("a", "b"), each = 7),
   sep = "."
 ),
 nrow = 7
)

# without preserving haplotypes
S1 <- mat_scramble(Mat)

# preserving haplotypes with markers 1-7 all on one chromosome
S2 <- mat_scramble(Mat, preserve_haplotypes = TRUE)

# preserving haplotypes with markers 1-3 on one chromosome and 4-7 on another
S3 <- mat_scramble(Mat, row_groups = list(1:3, 4:7))

# preserving individuals, but not haplotypes, with two chromosomes
S4 <- mat_scramble(Mat, row_groups = list(1:3, 4:7), preserve_individuals = TRUE)

# preserving individuals by chromosome, but not haplotypes, with two chromosomes
S5 <- mat_scramble(Mat, row_groups = list(1:3, 4:7), preserve_individuals = "BY_CHROM")

# preserving individuals by chromosome, and preserving haplotypes, with two chromosomes
S6 <- mat_scramble(Mat, row_groups = list(1:3, 4:7),
                 preserve_individuals = "BY_CHROM", preserve_haplotypes = TRUE)
```

M_meta *Metadata for 100 molecular markers*

Description

This data set contains the metadata for the 100 most divergent SNP loci for three feral swine populations sampled in the United States. To make the dataset more computationally efficient, only 3 chromosomes were used (12,17, and 18). The metadata for the SNP loci is in a tibble with three columns: chrom (character), pos (numeric), variant_id (character). The column 'chrom' contains the chromosome ID where the SNP is located, the column 'pos' gives the base pair location on the chromosome, and the column 'variant_id' gives the name of the SNP. Each row of the metadata tibble contains the metadata for a given SNP locus. The individual genotypes for each of these SNP loci can be found in `Geno`.

Format

A tibble with three columns: chrom, pos, and variant_id.

Source

Chromosome, position, and variant IDs are from the Sus scrofa 10.2 genome `https://www.ncbi.nlm.nih.gov/assembly/GCF_000003025.5/`.

`perm_gs_by_pops` *Take the output of rearrange_genos and permute everyone by population*

Description

This is done prior to assigning random genomic fragments of individuals in the sample to the founders of the GSP, to be dropped to the samples.

Usage

```
perm_gs_by_pops(GS, preserve_haplotypes = FALSE, preserve_individuals = FALSE)
```

Arguments

`GS`	the tibble that is the output from rearrange_genos
`preserve_haplotypes`	If TRUE then the Geno data is assumed phased (first allele at an individual on one haplotype and second allele on the other) and those haplotypes are preserved in this permutation of genomic material amongst the founders.
`preserve_individuals`	If TRUE then whole individuals are permuted around the data set and the two gene copies at each locus are randomly permuted within each individual. If `preserve_individuals = "BY_CHROM"`, then the the two copies of each chromosome in an individual are permuted together. Thus a permuted individual may have two copies of one chromosome from one individual, and two copies of another chromosome from a different individual. (If `preserve_haplotypes = TRUE` then the gene copies are not permuted within individuals. You should only ever use `preserve_haplotypes = TRUE` if you have phased data.)

Value

Returns a list of the same format as the output of `rearrange_genos`. Plus one additional component. Each component of the return list is itself an unnamed list with one component (makes it easier to use `bind_rows` to create a tibble of list columns from these). The components, once unlisted are:

- `G`: a matrix—the original genotype data matrix
- `I`: the I_meta tibble
- `M`: the M_meta tibble
- `G_permed`: the genotype matrix after permutation.

Examples

```
# first get the output of rearrange_genos
RG <- rearrange_genos(Geno, I_meta, M_meta)

# then permute by the populations
PG <- perm_gs_by_pops(RG)
```

`plink2gscramble` *read plink-formatted .map and .ped files into 'gscramble' format*

Description

This will read .ped and .map files (which can be gzipped, but cannot be the binary .bed or .bim plink format). The population specifier of each individual is assumed to be the first column (the FID column) in the .ped file.

Usage

```
plink2gscramble(ped = NULL, map = NULL, prefix = NULL, gz_ext = FALSE)
```

Arguments

`ped`	path to the plink `.ped` file holding information about the individuals and their genotypes. This file can also be gzipped. The function assumes that the second column of this file is unique across all family IDs. If this is not the case, the function throws a warning. It is assumed that missing genotypes are denoted by 0's in this file.
`map`	path to the plink `.map` file holding information about the markers. This file can be gzipped
`prefix`	If map and ped are not given as explicit paths to the file, you can give the prefix, and it will search for the two files with the .ped and .map extensions on the end of the prefix.
`gz_ext`	Logical. If TRUE, and specifying files by prefix, this will add a .gz extension to the map and ped files.

Value

A list with three components:

- `I_meta`: meta data about the individuals in the file. This will include the columns of `group` (value of the first column of the ped file) and `indiv` (the ID of the individual stored in second column of the ped file). And wil also include the other four columns of the plink ped specification, named as follows: `pa` `ma`, `sex_code`, `pheno`.
- `M_meta`: meta data about the markers. A tibble with the columns `chrom`, `pos`, and `variant_id` and `link_pos`. The `link_pos` column holds the information about marker position in Morgans or cM that was included in the `map` file.
- `Geno`: a character matrix of genotypes with number-of-indviduals rows and number-of-markers * 2 columns. Missing genotypes in this matrix are coded as `NA`.

Examples

```
ped_plink <- system.file("extdata/example-plink.ped.gz", package = "gscramble")
map_plink <- system.file("extdata/example-plink.map.gz", package = "gscramble")

result <- plink2gscramble(ped_plink, map_plink)
```

`plink_map2rec_rates` *Convert a PLINK map file to 'gscramble' RecRates bins in a tibble*

Description

This is a convenience function to convert PLINK map format to the format used in the 'gscramble' `RecRates` object. By default, this function will use the positions of the markers and assume recombination rates of 1 cM per megabase. If the marker positions are also available in Morgans in the PLINK map file, the these can be used by setting `use_morgans` to TRUE.

Usage

```
plink_map2rec_rates(
  map,
  use_morgans = FALSE,
  cM_per_Mb = 1,
  chrom_lengths = NULL
)
```

Arguments

`map`	path to the plink `.map` file holding information about the markers. This file can be gzipped.
`use_morgans`	logical. IF true, the third column in the PLINK map file (assumed to have the position of the markers in Morgans) will be used to calculate the `rec_probs` in the bins of the RecRates object.
`cM_per_Mb`	numeric. If `use_morgans` is `FALSE`, physical positions will be converted to recombination fractions as `cM_per_Mb` centiMorgans per megabase. Default is 1. This is also used to determine the recombination probability on the last segment of the chromosome (beyond the last marker) if `chrom_lengths` is used.
`chrom_lengths`	if you know the full length of each chromosome, you can add those in a tibble with columns `chrom` and `bp` where `chrom` *must be a character vector* (Don't leave them in as numerics) and `bp` must be a numeric vector of the number of base pairs of length of each chromosome.

Details

For simplicity, this function will assume that the length of the chromosome is just one base pair beyond the last marker. That is typically not correct but will have no effect, since there are no markers to be typed out beyond that point. However, if you know the lengths of the chromosomes and want to add those in there, then pass them into the `chrom_lengths` option.

Value

A tibble that provides the recombination rates for the segments of the genome.

Examples

```
mapfile <- system.file(
    "extdata/example-plink-with-morgans.map.gz",
    package = "gscramble"
 )

# get a rec-rates tibble from the positions of the markers,
# assuming 1 cM per megabase.
rec_rates_from_positions <- plink_map2rec_rates(mapfile)

# get a rec-rates tibble from the positions of the markers,
# assuming 1.5 cM per megabase.
rec_rates_from_positions_1.5 <- plink_map2rec_rates(
    mapfile,
    cM_per_Mb = 1.5
)

# get a rec-rates tibble from the cumulative Morgans position
# in the plink map file
rec_rates_from_positions_Morg <- plink_map2rec_rates(
    mapfile,
    use_morgans = TRUE
)

# get a rec-rates tibble from the cumulative Morgans position
# in the plink map file, and extend it out to the full length
# of the chromosome (assuming for that last part of the chromosome
# a map of 1.2 cM per megabase.)
rec_rates_from_positions_Morg_fl <- plink_map2rec_rates(
    mapfile,
    use_morgans = TRUE,
    cM_per_Mb = 1.2,
    chrom_lengths = example_chrom_lengths
)
```

`plot_simulated_chromomsome_segments` *Plot the simulated chromosomes of an individual*

Description

This function uses the information in the tibble about segments dropped down a genome simulation pedigree to plot the chromomomes of an individual colored by the population of origin of each segment.

Usage

```
plot_simulated_chromomsome_segments(
  Segs,
  RR = NULL,
  fill_by_group_origin = FALSE,
  rel_heights = c(chrom_ht = 4, chrom_gap = 0.8, spark_gap = 0.2 * !is.null(RR),
    spark_box = 2.6 * !is.null(RR), unit_gap = 4),
  bottom_gap = 0.3,
  spark_thick = 0.2,
  spark_splat = 0.25
)
```

Arguments

`Segs`	a tibble of segments
`RR`	a tibble of recombination rates in bins in the format of RecRates. If this is included, the recombination rates in cM/Mb are plotted atop the chromosomes as a little sparkline. If it is not included, then the there are no little sparklines above the chromosomes.
`fill_by_group_origin`	If FALSE (the default) the fill color of segments is mapped to the pop_origin, which is where the founder haplotypes came from according to the `hpop1` and `hpop2` columns in the GSP specification. If you set this to TRUE, then we map the "group" column of the reppop to fill.
`rel_heights`	a vector the the relative heights of the different elements of each chromosomal unit of the plot. This is a named vector with the following elements, listed in order of the bottom of each unit to the top: • `chrom_ht`: the height of the bars for each of the two chromosomes of the pair in a chromosome unit. • `chrom_gap`: The gap between the two homologous chromosomes of the individual. • `spark_gap`: the gap between the top chromosome and the sparkline box for recombination rates. • `spark_box`: height of the box within which the sparkline goes. Note that the sparkline itself will be scaled so that the highest rate anywhere within the genome will correspond to the top of the spark box. • `unit_gap`: The relative height of the gap between one chromosome unit and the next.
`bottom_gap`	the y value of the bottom chromosome unit. Basically the absolute distance between the y=0 line and the start of the plotted material. Should typically be between 0 and 1.
`spark_thick`	thickness of the line that draws the recombination rate sparkline.
`spark_splat`	fraction by which the unit gap should be reduced when there are sparklines being drawn.

Value

This function returns a ggplot object. Each facet of the plot shows the chromosomes of a different sampled individual from a particular replicate simulation from a particular genome simulation pedigree. The facets are titled like: `GSP 1, Idx 2, ID 8[3]`, which means that the chromosomes shown in the panel are from the third sampled set of chromosomes from the individual with ID 8 from the simulation from genome simulation pedigree 1 with index 2.

Examples

```
s <- example_segments
rr <- RecRates
g <- plot_simulated_chromomsome_segments(s)
g_with_sparklines <- plot_simulated_chromomsome_segments(s, rr)
```

`prep_gsp_for_hap_dropping`
Take a gsp in tibble form and make a list suitable for gene dropping

---

Description

Just a simple function that makes a list-based data structure that makes it easy to gene-drop chromosome segments down the gsp. The basic idea is to get everyone into a data structure in which they are ordered such that by the time you get to segregating segments *from* anyone, you already have the segments segregated *into* them. This works by each individual having a list of gametes (post-recombination) coming out of them, and a list of "uniters" which are the gametes coming into them. We just point the uniters to gametes in the previous generation and then make sure that we shuffle the order of the gametes when they come out of someone.

Usage

```
prep_gsp_for_hap_dropping(gsp)
```

Arguments

`gsp`	A tibble that holds the genome simulation pedigree (GSP). This is a tibble in which each row specifies an individual in the GSP. The columns of the tibble are: • `ind`: a numeric identifier for that row's indvidual. • `par`: the numeric ID of the first parent of the individual (NA if the individual is a founder of the pedigree). • `par2`: the numeric ID of the second parent of the individual (NA if the individual is a founder) • `ipar1`: the number of gametes that `par1` must segregate "through" `ind` in order to exhaust all the genetic material in the GSP. These values are given by the red numerals alongside the edge connecting parents to offspring in the GSP images defined by `gsp2dot()`. See the vignette `gscramble-tutorial`, for an example. (NA if `ind` is a founder).

- ipar2: the number of gametes that par2 must segregate through ind. (NA if ind is a founder).
- hap1: a unique character label given to the first haplotype in ind if ind is a founder. If ind is not a founder, this must be NA.
- hap2: unique character label given to the second haplotype in ind. NA if ind is not a founder.
- hpop1: character label giving the population of origin of the first haplotype (i.e., hap1) in ind, if ind is a founder. NA otherwise.
- hpop2: character label giving the population of origin of the second haplotype (i.e., hap2) in ind. NA if ind is not a founder.
- sample: unique character label for the outcoming diploid sample from the pedigree member ind. NA if ind is not sampled.
- osample: the number of diploid samples that come out of ind. NA if ind is not sampled.

Value

This function returns a named list, which is a linked-list type of structure that contains the same information that is in gsp, but it makes it easier to access when traversing the pedigree.

The length of the list is nrow(gsp). The names are the character versions of the ind column. Each component of the list refers to an individual row from gsp. All of these list elements are themselves lists. (i.e., the information about a single individual is stored in a list.) Every such individual list has at least the two elements:

- isSample: TRUE if samples are taken from the individual. FALSE otherwise.
- isFounder: TRUE if the individual is a founder. FALSE otherwise.
- nGamete: The total number of gametes that will be segregated *out* of this individual along edges to the *offspring* of the individual in the pedigree. This is the sum of all the red numbers alongside edges below the individual in the GSP.

If an individual's isSample is TRUE, then its list also contains the following elements:

- nSamples: the number of diploid genomes sampled out of this individual. This is the purple number along the edge to the sample node below the individual in the GSP "picture".

If an individual's, isFounder is TRUE then its list also contains the following elements:

- hpop1: the population from which haplotype 1 in this (founder) individual originated.
- hpop2: the population from which haplotype 2 in this (founder) individual originated.
- fh_idx1: this stands for "founding haplotype index 1. It is a unique integer that identifies haplotype one in this founder individual. This integer is unique over all haplotypes in all the founder individuals.
- fh_idx2' the unique integer identifier for haplotype two in this founder individual.

If an individual's isFounder is FALSE, then its list also contains the following elements:

- par1 and par2. Each of these is a list with the elements:

  - `par`: the character identifier of the first (if in `par1`) of the second (if in `par2`) parent of the individual.
  - `gam_idx`: This tells us which of the gametes in the parent (1 or 2) depending on if this is in `par1` or `par2`, gets segregated to the individual. NEED TO EXPLAIN MORE. SINCE THINGS GET PERMUTED, ETC.

Examples

```
# get the 13 member complex pedigree in tibble form as the
# package data object GSP and prep it:
GSP_list <- prep_gsp_for_hap_dropping(GSP)
```

`rearrange_genos` *rearrange genotypes into separate columns for each haplotype.*

Description

This function first reorders individuals in the columns of the matrix so that every population is together. Then it rearranges genotypes into separate columns for each haplotype (or "halflotype" if they are unphased.) This prepares the matrix for different kinds of a permutation (within or between populations, for example).

Usage

```
rearrange_genos(G, Im, Mm)
```

Arguments

`G`	the genotype matrix (N rows and 2L columns)
`Im`	the meta data for the N samples.
`Mm`	the meta data for the L markers.

Details

It returns a list. One component is the matrix, another is the updated individual meta data, and the third is the marker meta data.

Value

Returns a list. Each component of the return list is itself an unnamed list with one component (makes it easier to use `bind_rows` to create a tibble of list columns from these). The components, once unlisted are:

- `G`: a matrix—the rearranged genotype data matrix
- `I`: the I_meta tibble
- `M`: the M_meta tibble

Examples

```
RG <- rearrange_genos(Geno, I_meta, M_meta)
```

`recomb_point` *Randomly sample the positions of recombinations on a chromosome*

Description

This function uses the observed recombination fractions such as those in the data object RecRates. These are observed recombination fractions in a series of adjacent small bins that are defined by a start position `start_pos` and ending position `end_pos`. This function operates on the recombination rates for only a single chromosome at a time, so it will typically be wrapped up inside a `purrr::map()` function to operate over multiple chromosomes.

Usage

```
recomb_point(M, at_least_one = TRUE)
```

Arguments

`M`	a tibble that has the columns start_pos, end_pos, and rec_prob (where rec_prob is the probability of a recombination occurring during meiosis within the interval defined by start_pos and end_pos.
`at_least_one`	if this is TRUE then at least one recombination occurs on every chromosome (see Details). If FALSE then the total number of recombinations is simulated as a Poisson r.v. with mean equal to the sum of the recombination fractions.

Details

There are two main modes by which this function operates. If `at_least_one == TRUE`, then the chromosome will always have at least one recombination. In this case, the position of that first recombination is chosen according to the recombination rates. Subsequently, the remaining number of recombinations is chosen by the random variable Y, which is the greater of zero and X - 1, where X is a Poisson r.v. with mean given by the sum of the recombination fractions. These additional recombinations are placed randomly according to the `rec_probs` but without interference.

If `at_least_one == FALSE` then the total number of recombinations is simulated as a Poisson r.v. with mean equal to the sum of the recombination fractions. Again, their placement assumes no interference.

Locations within each bin are chosen uniformly. These locations are represented as real numbers (rather than as integers) and those are used for describing segments, as well. This simplifies matters such as condensing information about multiple recombinations that occurred at the same base pair. In practice, this will have negligible effects, since it is so unlikely that a recombination will ever occur in the same place.

If no recombination occurs, this just returns a zero-length numeric vector.

Value

Returns a numeric vector of recombination breakpoints along the chromome. The values are sorted in ascending order.

Examples

```
# for an example, create a tibble of bins, roughly 1 Mb each,
# on a chromosome of length roughly 150 Mb, and we assign each
# a rec_prob around 0.01
ends <- seq(1e6, 150e6, by = 1e6)
ends <- ends + floor(runif(length(ends), min = -1e4, max = 1e4))
set.seed(5)
M <- tibble::tibble(
    start_pos = c(0, ends[-length(ends)] + 1),
    end_pos = ends,
    rec_prob = abs(rnorm(length(ends), 0.01, 0.004))
)
recomb_point(M)
```

`RecRates` *Recombination rate data for many roughly 1 Mb bins*

Description

Chromosome, start position and end position and probability of recombination within the bin for chromosomes in pigs.

Format

A tibble with four columns: chrom, chrom_len, start_pos, end_pos, and rec_prob.

Source

These rates were estimated in: Tortereau, Flavie, et al. "A high density recombination map of the pig reveals a correlation between sex-specific recombination and GC content." BMC genomics 13.1 (2012): 1-12. It is available for download from `https://static-content.springer.com/esm/art%3A10.1186%2F1471-2164-13-586/MediaObjects/12864_2012_4363_MOESM1_ESM.txt`. After downloading the data were processed to remove inconsistencies with the marker data set used for M_meta and Geno.

renumber_GSP *Renumber GSP members by adding a constant to each*

Description

This function assumes that all individuals are named as numerics and that their haplotypes in hap1 and hap2 are named Xa and Xb,respectively, and their samples are named sX, where X is an integer,

Usage

```
renumber_GSP(G, add)
```

Arguments

`G`	a GSP tibble
`add`	amount to add to each label

Value

Returns a GSP just like the input, but with the identity numbers of the individuals in it incremented by `add`.

Examples

```
# get an example GSP
G <- create_GSP(pop1 = "p1", pop2 = "p2", F1B2 = TRUE)
```

RepPop1 *A simple example of a reppop table*

Description

A `reppop` table in 'gscramble' is used to define how the founder populations in a GSP (typically named something like "A", "B", etc.) are mapped to the groups/populations of individuals, as specified in the individual meta data (an example of which is found in `I_meta`).

Format

A tibble with three columns: `index`, which must be of type integer, `pop`, and `group` of type character.

Details

This particular version shows a situation where individuals from group Pop1 will be sampled as founders from A and from group Pop2 will be sampled as founders from B into the GSP. Since this `RepPop1` example has two values of index: 1 and 2, it specifies that individuals from the populations will be sampled without replacment, two times into the founders on the pedigree.

Source

The developers created this.

RepPop4 *Another simple example of a reppop table*

Description

A `reppop` table in 'gscramble' is used to define how the founder populations in a GSP (typically named something like "A", "B", etc.) are mapped to the groups/populations of individuals, as specified in the individual meta data (an example of which is found in `I_meta`).

Format

A tibble with three columns: `index`, which must be of type integer, `pop`, and `group` of type character.

Details

This particular version shows a situation where individuals from four different groups (Pop1, Pop2, Pop3, and Pop4) get mapped to four different founder groups (A, B, C, D) in the GSP. Since this `RepPop4` example has three values of index: 1, 2, and 3, it specifies that there will be three rounds of sampling of individuals from the populations to be the founders on the pedigree. That will be done entirely without replacement (individuals are not replaced after each round!)

Source

The developers created this.

---

seg2tib *Takes a gamete in segment format and returns a tibble with Pop and indiv_index*

Description

A small helper function.

Usage

```
seg2tib(s)
```

Arguments

`s`	a gamete in segment format

Value

Returns a tibble with columns `tmp_seg_names`, `start`, and end, that show the origin (in `tmp_seg_names`) of segments that start at `start` and end at end.

Examples

```
# first make a segment that has pieces from a few different founders
 V <- c(Amy = 0, Bob = 10000, Joe = 30000, Frieda = 40000)
 seg2tib(V)
```

`segments2markers` *Map alleles from scrambled founders to the sampled segments from a GSP.*

Description

Map alleles from scrambled founders to the sampled segments from a GSP.

Usage

```
segments2markers(
  Segs,
  Im,
  Mm,
  G,
  preserve_haplotypes = FALSE,
  preserve_individuals = FALSE
)
```

Arguments

`Segs`	the simulated segments. A tibble like that returned from `segregate()`.
`Im`	the individual meta data, like that in `I_meta`. A tibble with columns `group` and `indiv`.
`Mm`	the marker meta data formatted like that in `M_meta`. A tibble with columns `chrom`, `pos`, and `variant_id`.
`G`	the marker genotype data as a matrix like `Geno`. This is a character matrix. Each row is an individual, and each pair of columns are the alleles at a locus. Thus it is N x 2L where N is the number of individuals and L is the number of markers.
`preserve_haplotypes`	If TRUE then the Geno data is assumed phased (first allele at an individual on one haplotype and second allele on the other) and those haplotypes are preserved in this permutation of genomic material amongst the founders.

`preserve_individuals`
: If TRUE then whole individuals are permuted around the data set and the two gene copies at each locus are randomly permuted within each individual. If `preserve_individuals = "BY_CHROM"`, then the the two copies of each chromosome in an individual are permuted together. Thus a permuted individual may have two copies of one chromosome from one individual, and two copies of another chromosome from a different individual. (If `preserve_haplotypes = TRUE` then the gene copies are not permuted within individuals. You should only ever use `preserve_haplotypes = TRUE` if you have phased data.)

Value

A list with three components:

- `ret_geno`: A character matrix where each row is an individual and each pair of columns are the alleles at a locus, thus it is N x 2L where N is the number of individuals and L is the number of markers.
- `ret_ids`: A tibble providing the individual meta data with columns `groups` and `indiv`.
- `hyb_Qs`: A tibble of the admixture Q values.

Examples

```
#### First, get input segments for the function ####
# We construct an example here where we will request segregation
# down a GSP with two F1s and F1B backcrosses between two hypothetical
# populations, A and B.
set.seed(5)
gsp_f1f1b <- create_GSP("A", "B", F1 = TRUE, F1B = TRUE)

# We will imagine that in our marker data there are three groups
# labelled "Pop1", "Pop2", and "Pop3", and we want to create the F1Bs with backcrossing
# only to Pop3.
reppop <- tibble::tibble(
    index = as.integer(c(1, 1, 2, 2)),
    pop = c("A", "B", "A", "B"),
    group = c("Pop3", "Pop1", "Pop3", "Pop2")
)

# combine those into a request
request <- tibble::tibble(
   gpp = list(gsp_f1f1b),
   reppop = list(reppop)
)

# now segegate segments.  Explicitly pass the markers
# in M_meta so that the order of the markers is set efficiently.
segs <- segregate(request, RecRates, M_meta)

#### Now, use segs in an example with segments2markers() ####
# this uses several package data objects that are there for examples
# and illustration.
s2m_result <- segments2markers(segs, I_meta, M_meta, Geno)
```

`segregate` *Segregate segments down genomic simulation pedigrees*

Description

Given a collection of genomic simulation pedigrees and requests for how many simulations should be done (in the `request` input), as well as recombination rates, this simulates the segregation of segments down through the pedigrees

Usage

```
segregate(request, RR, MM = NULL)
```

Arguments

`request`	a tibble with list columns "gpp" and "reppop". Each element of the gpp column is a tibble giving a genomic simulation pedigree as documented as the input for `prep_gsp_for_hap_dropping()`. Each element of the "reppop" column is a tibble with columns `index`, `pop`, `group`, to indicate which of the founding populations ("A", "B", etc.) correspond to the different groups (from the `group` column in, for example, the meta data for individuals in your genotype data set, like the data object `I_meta`). Because it is quite likely that you might wish to iterate the segregation procedure multiple times in a single simulation, you can specify that by doing multiple "reps" (replicates) of the procedure. **BIG NOTE**: The values in the index column that you choose must start at 1 and should be dense within. In other words, if the max value in the index column is N, then every integer from 1 to N must be in there.
`RR`	the recombination rates in the format of the package data
`MM`	the marker meta data tibble (like M_meta). If this is NULL (the default) that is fine. If not, then it uses the order of the markers in MM to define the levels of a chrom_f column so that we can sort the rows of the output correctly, with respect to markers in the Genotype data frame. This will let us more efficiently subscript the markers out of the matrix. If MM is not present, then the function will create `chrom_f` by using the order of the chromosomes from RR. If MM is not NULL, then the function will also check to make sure that the markers are within the extent of the recombination rate bins, giving an error otherwise.

Value

The output from this function is a tibble. Each row represents one segment of genetic material amongst the sampled individuals from the genomic permutation pedigrees. The columns give information about the provenance and destination of that segment as follows. Each segment exists in one of the samples (`samp_index`) from a sampled individual with a `ped_sample_id` on a given `gpp` (the index giving the row of the request input tibble) in a given `index` within the individual. Further, it is on one of two gametes (`gamete_index`) that segregated into the individual, and it came from a certain founding population (`pop_origin`) that corresponds to the named groups in the genotype file (`group_origin`). And, of course, the segment occupies the space from `start` to `end` on a chromosome `chrom`. Finally, the index of the founder haplotype on the given gpp that this segement descended from is given in `rs_founder_haplotype` which is short for "rep-specific founder haplotype". This final piece of information is crucial for segregating variation from the individuals in the `Geno` file onto these segments. Finally, the column `sim_level_founder_haplo` assigns a unique index for each founder haplotype. This is necessary because any simulation can involve multiple gpps and/or indexes of gpps, and the founders in each of those must all be unique within a simulation. so that those haplotypes can all, eventually, be accessed easily out of the genotype matrix.

Examples

```
# We construct an example here where we will request segregation
# down a GSP with two F1s and F1B backcrosses between two hypothetical
# populations, A and B.
gsp_f1f1b <- create_GSP("A", "B", F1 = TRUE, F1B = TRUE)

# We will imagine that in our marker data there are three groups
# labelled "grp1", "grp2", and "grp3", and we want to create the F1Bs with backcrossing
# only to grp3.
reppop <- tibble::tibble(
    index = as.integer(c(1, 1, 2, 2)),
    pop = c("A", "B", "A", "B"),
    group = c("grp3", "grp1", "grp3", "grp2")
)

# combine those into a request
request <- tibble::tibble(
   gpp = list(gsp_f1f1b),
   reppop = list(reppop)
)


result1 <- segregate(request, RecRates)

# here we pass it some markers, too
result2 <- segregate(request, RecRates, M_meta)

result1

result2
```

`sim_level_founder_haplos` *computes the simulation-level founder haplotype index for each founder haplo*

Description

This takes the output of `segregate()` and deals with the multiple gpp's and reps to come up with a unique index for each found haplotype, so that those haplotypes can all, eventually, be accessed easily out of the genotype matrix. Along the way, this function does some light checking to make sure that the `rs_founder_haplo` values are dense within `gpp` and `index` as they should be.

Usage

```
sim_level_founder_haplos(S)
```

Arguments

`S`	tibble of segments like that produced by `segregate`.

Value

This function returns a result that is basically the output of `segregate()` with an additional column added to it: `sim_level_founder_haplo`. This is the index of the haplotype within each `group_origin` that should be used. For details of the other columns in the output tibble, see the documentation for `segregate`.

Examples

```
#### Get output from segregate to use as input ####
# We construct an example here where we will request segregation
# down a GSP with two F1s and F1B backcrosses between two hypothetical
# populations, A and B.
gsp_f1f1b <- create_GSP("A", "B", F1 = TRUE, F1B = TRUE)

# We will imagine that in our marker data there are three groups
# labelled "grp1", "grp2", and "grp3", and we want to create the F1Bs with backcrossing
# only to grp3.
reppop <- tibble::tibble(
  index = as.integer(c(1, 1, 2, 2)),
  pop = c("A", "B", "A", "B"),
  group = c("grp3", "grp1", "grp3", "grp2")
)

# combine those into a request
request <- tibble::tibble(
  gpp = list(gsp_f1f1b),
  reppop = list(reppop)
)

# now run it through segregate()
set.seed(5)  # just for reproducibility in example...
simSegs <- segregate(request, RecRates)

#### Now we can run those through sim_level_founder_haplos() ####
fh <- sim_level_founder_haplos(simSegs)
fh
```

xover *internal function to do crossovers and create recombinations*

Description

This function doesn't choose the recombination points. That is done with the function `recomb_point()`, and the results are passed into this function. The two inputs `V1` and `V2` represent the two gametes coming into an individual on the pedigree. Recombination occurs within that individual, and the two resulting gametes from that recombination are the output. Typically this is not the way things happen, of course. Generally, only one of the two resulting gametes from the recombination will be segregated to a surviving offspring. But, since we are interested in segregating genetic material without duplicating or destroying any of it, we keep track of both gametes that result from the meiosis/recombination.

Usage

```
xover(V1, V2, R)
```

Arguments

`V1`	integer vector of recombination points already existing on the first incoming gamete. Its names are the names of the founder haplotype that the left end originates from (i.e. it is from the named haplotype up until it changes at each point). For example c(A = 0, B = 12890, B = 30000) would work for a 30 Kb chromosome in which there is a single recombination just to the right of the point 12890. (In that example, positions 1 through 12890 are from founder haplotype A, while positions 12891 to 30000 are from founder haplotype B.) Note that these vectors have to have a first value of 0 and a final value of the chromosome length.
`V2`	integer vector of breakpoints of the second incoming gamete. Format is just like it is for V1.
`R`	a vector of new breakpoints to insert into the existing ones on each gamete. This is usually returned from the function `recomb_point()`.

Value

This sends back two updated gametes, V1 and V2, but with the new points of recombination stuck in there. Note, for two incoming gametes there are two outgoing gametes, but we aren't "re-using" any genomic sequence.

Examples

```
#' # make the two gametes/chromosomes coming into the function.
#' # Each one has length 30000 and a single existing recombination
V1 <- c(A = 0, B = 10000, B = 30000)
V2 <- c(C = 0, D = 20000, D = 30000)
```

```
# now, set a new recombination point at position 15000
xover(V1, V2, R = 15000)

# set three recombination points at 5,000, 15,000, and 25,000:
xover(V1, V2, R = c(5000, 15000, 25000))

# no recombinations (R is a zero length numeric vector)
xover(V1, V2, R = numeric(0))
```

Index