

Package: simfam (via r-universe)

October 30, 2024

Title Simulate and Model Family Pedigrees with Structured Founders

Version 1.1.11.9000

Description The focus is on simulating and modeling families with founders drawn from a structured population (for example, with different ancestries or other potentially non-family relatedness), in contrast to traditional pedigree analysis that treats all founders as equally unrelated. Main function simulates a random pedigree for many generations, avoiding close relatives, pairing closest individuals according to a 1D geography and their randomly-drawn sex, and with variable children sizes to result in a target population size per generation. Auxiliary functions calculate kinship matrices, admixture matrices, and draw random genotypes across arbitrary pedigree structures starting from the corresponding founder values. The code is built around the plink FAM table format for pedigrees. There are functions that simulate independent loci and also functions that use an explicit recombination model to simulate linkage disequilibrium (LD) in the pedigree, as well as population analogs resembling the Li-Stephens model. Described in Yao and Ochoa (2023) <[doi:10.7554/eLife.79238](https://doi.org/10.7554/eLife.79238)>.

License GPL (>= 3)

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Imports Rcpp, stats, tibble, dplyr, tidyselect, rlang, Matrix, methods

Suggests testthat (>= 3.0.0), popkin, bnpsd (>= 1.3.2), genio, BEDMatrix, kinship2, RColorBrewer, knitr, rmarkdown

Config/testthat/edition 3

VignetteBuilder knitr

LinkingTo Rcpp

URL <https://github.com/OchoaLab/simfam>

BugReports <https://github.com/OchoaLab/simfam/issues>

Depends R (>= 2.10)

LazyData true

Repository https://ochoalab.r-universe.dev

RemoteUrl https://github.com/ochoalab/simfam

RemoteRef HEAD

RemoteSha a59451c1fafafa4e99d2ae7ac48c01d3ecaf7b23

Contents

admix_fam	2
admix_last_gen	4
bim_add_posg	5
draw_sex	7
fam_ancestors	7
geno_fam	8
geno_last_gen	10
geno_last_gen_admix_recomb	11
kinship_fam	14
kinship_last_gen	16
pop_recomb	17
prune_fam	19
recomb_admix_inds	20
recomb_fam	22
recomb_founder_blocks_inherited	24
recomb_geno_inds	25
recomb_haplo_inds	27
recomb_init_founders	28
recomb_last_gen	30
recomb_map_fix_ends_chr	31
recomb_map_hg	32
recomb_map_inds	33
recomb_map_simplify_chr	34
sim_pedigree	36
tidy_recomb_map_inds	38

Index **40**

admix_fam	<i>Calculate admixture matrix of a pedigree with known admixture of founders</i>
-----------	--

Description

Calculates a full admixture proportions matrix (for all individuals in the provided pedigree FAM table) starting from the admixture proportions of the founders as provided.

Usage

```
admix_fam(admix, fam, missing_vals = c("", 0))
```

Arguments

admix	The admixture proportions matrix of the founders (individuals along rows and ancestries along columns). This matrix must have row names that identify each founder (matching codes in fam\$id). Individuals may be in a different order than fam\$id. Extra individuals in admix but absent in fam\$id will be silently ignored. All values should be non-negative and each row of admix should sum to one; for speed, this code does not check that admix is valid, just averages data as-is.
fam	The pedigree data.frame, in plink FAM format. Only columns id, pat, and mat are required. id must be unique and non-missing. Founders must be present, and their pat and mat values must be missing (see below). Non-founders must have both their parents be non-missing. Parents must appear earlier than their children in the table.
missing_vals	The list of ID values treated as missing. NA is always treated as missing. By default, the empty string ("") and zero (0) are also treated as missing (remove values from here if this is a problem).

Value

The admixture proportions matrix of the entire fam table, based on the admixture of the founders. These are expectations, calculated for each individual as the average ancestry proportion of the parents. The rows of this admixture matrix correspond to fam\$id in that order. The columns (ancestries) are the same as in the input admix.

See Also

Plink FAM format reference: <https://www.cog-genomics.org/plink/1.9/formats#fam>

Examples

```
# The smallest pedigree, two parents and a child.
# A minimal fam table with the three required columns.
# Note "mother" and "father" have missing parent IDs, while "child" does not
library(tibble)
fam <- tibble(
  id = c('father', 'mother', 'child'),
  pat = c(NA, NA, 'father'),
  mat = c(NA, NA, 'mother')
)

# admixture proportions of the parents
admix <- rbind( c(0.3, 0.3, 0.4), c(0.5, 0.25, 0.25) )
# Name the parents with same codes as in `fam`
# (order can be different)
rownames( admix ) <- c('mother', 'father')
# name ancestries too
```

```

colnames( admix ) <- c('African', 'European', 'Asian')

# Calculate the full admixture proportions matrix
admix_all <- admix_fam( admix, fam )

# This is a 3x3 matrix with row names matching fam$id.
# The parent submatrix equals the input (reordered),
# but now there's admixture to the child too (averages of parents)
admix_all

```

admix_last_gen	<i>Calculate admixture matrix for last generation of a pedigree with admixture of founders</i>
----------------	--

Description

A wrapper around the more general `admix_fam()`, specialized to save memory when only the last generation is desired (`admix_fam()` returns admixture for the entire pedigree in a single matrix). This function assumes that generations are non-overlapping (met by the output of `sim_pedigree()`), in which case each generation g can be drawn from generation $g-1$ data only. That way, only two consecutive generations need be in memory at any given time. The partitioning of individuals into generations is given by the `ids` parameter (again matches the output of `sim_pedigree()`).

Usage

```
admix_last_gen(admix, fam, ids, missing_vals = c("", 0))
```

Arguments

admix	The admixture proportions matrix of the founders (individuals along rows and ancestries along columns). This matrix must have row names that identify each founder (matching codes in <code>fam\$id</code>). Individuals may be in a different order than <code>fam\$id</code> . Extra individuals in <code>admix</code> but absent in <code>fam\$id</code> will be silently ignored. All values should be non-negative and each row of <code>admix</code> should sum to one; for speed, this code does not check that <code>admix</code> is valid, just averages data as-is.
fam	The pedigree data.frame, in plink FAM format. Only columns <code>id</code> , <code>pat</code> , and <code>mat</code> are required. <code>id</code> must be unique and non-missing. Founders must be present, and their <code>pat</code> and <code>mat</code> values must be missing (see below). Non-founders must have both their parents be non-missing. Parents must appear earlier than their children in the table.
ids	A list containing vectors of IDs for each generation. All these IDs must be present in <code>fam\$id</code> . If IDs in <code>fam</code> and <code>ids</code> do not fully agree, the code processes the IDs in the intersection, which is helpful when <code>fam</code> is pruned but <code>ids</code> is the original (larger) set.
missing_vals	The list of ID values treated as missing. NA is always treated as missing. By default, the empty string ("") and zero (0) are also treated as missing (remove values from here if this is a problem).

Value

The admixture proportions matrix of the last generation (the intersection of `ids[length(ids)]` and `fam$id`). The rows of this matrix are last-generation individuals in the order that they appear in `fam$id`.

See Also

Plink FAM format reference: <https://www.cog-genomics.org/plink/1.9/formats#fam>

Examples

```
# A small pedigree, two parents and two children.
# A minimal fam table with the three required columns.
# Note "mother" and "father" have missing parent IDs, while children do not
library(tibble)
fam <- tibble(
  id = c('father', 'mother', 'child', 'sib'),
  pat = c(NA, NA, 'father', 'father'),
  mat = c(NA, NA, 'mother', 'mother')
)
# need an `ids` list separating the generations
ids <- list( c('father', 'mother'), c('child', 'sib') )

# admixture proportions of the parents
admix <- rbind( c(0.3, 0.3, 0.4), c(0.5, 0.25, 0.25) )
# Name the parents with same codes as in `fam`
# (order can be different)
rownames( admix ) <- c('mother', 'father')
# name ancestries too
colnames( admix ) <- c('African', 'European', 'Asian')

# calculate the admixture matrix of the children
admix2 <- admix_last_gen( admix, fam, ids )
admix2
```

bim_add_posg

Calculate genetic positions from base pair positions and a genetic map

Description

Given a table of base pair positions (a data frame with chromosome and position values), and a genetic map (see [recomb_map_hg](#)), this function calculates genetic positions. If genetic positions existed in input, they are overwritten.

Usage

```
bim_add_posg(bim, map)
```

Arguments

bim	The table of variants, which is a data.frame/tibble with at least two columns: chr (must be numeric between 1 and the maximum chromosome in map below for map to work, otherwise ignored with a warning) and pos (base pair position, usually an integer). If column posg is present, it will be overwritten in the output, otherwise it is created. Additional columns may be present and will be unedited.
map	The genetic map, a list of chromosomes each of which is a data.frame/tibble with columns pos for base pair position and posg for genetic position.

Details

Base pair positions are converted to genetic positions from the provided map using linear interpolation, using `stats::approx()` with options `rule = 2` (out of range cases are set to nearest end's value) and `ties = list('ordered', mean)` (assume data is ordered, interpolate ties in base pair positions in map using mean of genetic positions). Output will be incorrect, without throwing errors, if genetic map is not ordered.

Value

The bim input with new or overwritten column posg of genetic positions in cM. Rows with values of chr that are not numeric or are out of range (for given map) are unedited if the posg column was present, or assigned NA otherwise.

See Also

[recomb_map_hg](#) for simplified human recombination maps included in this package.

Examples

```
# let's define a very simple table of base pair positions, with minimal information
library(tibble)
bim <- tibble(
  chr = c(1, 1, 2, 2),
  pos = c(50, 200, 30, 123) * 1000000
)
# use latest human recombination map
map <- recomb_map_hg38

# now use this function to add genetic positions column to `bim`!
bim <- bim_add_posg( bim, map )
```

draw_sex	<i>Draw sex values randomly for a list of individuals</i>
----------	---

Description

Each individual has their sex drawn between male and female with equal probability. Sex is encoded numerically following the convention for plink FAM files (see below).

Usage

```
draw_sex(n)
```

Arguments

n The number of individuals.

Value

The length-n vector of integer sex assignments: 1L corresponds to male, 2L to female.

See Also

Plink FAM format reference: <https://www.cog-genomics.org/plink/1.9/formats#fam>

Examples

```
draw_sex( 10 )
```

fam_ancestors	<i>Construct an ancestors-only pedigree for one person G-generations deep</i>
---------------	---

Description

Creates an idealized pedigree listing all ancestors of one individual starting from G generations ago, without inbreeding (a binary tree). IDs are automatically generated strings indicating generation and individual number within generation. Useful for simple simulations of individuals with explicit ancestors.

Usage

```
fam_ancestors(G)
```

Arguments

- G The desired number of generations. G=1 returns a trivial pedigree with a single individual; G=2 an individual and its two parents; G=3 an individual, its parents and grandparents, etc.

Value

A list with two named elements:

- fam: a tibble describing the pedigree, with the following columns
 - id: The ID of each individual, a string in the format "g-i" joining with a dash the generation number ("g", numbered backward in time) and the individual number within the generation ("i").
 - pat: The paternal ID. For individual "g-i" parent is (g+1)"-(2*i-1), except for last generation it is NA (their parents are missing).
 - mat: The maternal ID. For individual "g-i" parent is (g+1)"-(2*i), except for last generation it is NA (their parents are missing).
 - sex: 1 (male) for all odd-numbered individuals, 2 (female) for even-numbered individuals, consistent with pedigree structure. Side-effect is first-generation individual ("1-1") is always male (edit afterwards as desired).
- ids: A list containing vectors of IDs separated by generation, but here starting from the last generation (highest "g"), to be consistent with output of `sim_pedigree()` and the expected input of all `*_last_gen` functions.

See Also

`sim_pedigree()` to simulate a random pedigree with a given number of generations, generation sizes, and other parameters.

Examples

```
# construct the 8-generation ancestor tree of one individual:
data <- fam_ancestors( 8 )
# this is the pedigree
fam <- data$fam
# and this is the handy list of IDs by discrete generation,
# used by `*_last_gen` functions to reduce memory usage
ids <- data$ids
```

geno_fam

Draw random genotypes on a pedigree with known founder genotypes

Description

Constructs a random genotype matrix (for all individuals in the provided pedigree FAM table) starting from the genotype matrix of the founders as provided.

Usage

```
geno_fam(X, fam, missing_vals = c("", 0))
```

Arguments

X	The genotype matrix of the founders (loci along rows, individuals along columns). This matrix must have column names that identify each founder (matching codes in fam\$id). Individuals may be in a different order than fam\$id. Extra individuals in admix but absent in fam\$id will be silently ignored. All values should be in c(0L, 1L, 2L); for speed, this code does not check that X is valid (i.e. fractional values between 0 and 2 may not cause errors).
fam	The pedigree data.frame, in plink FAM format. Only columns id, pat, and mat are required. id must be unique and non-missing. Founders must be present, and their pat and mat values must be missing (see below). Non-founders must have both their parents be non-missing. Parents must appear earlier than their children in the table.
missing_vals	The list of ID values treated as missing. NA is always treated as missing. By default, the empty string ("") and zero (0) are also treated as missing (remove values from here if this is a problem).

Value

The random genotype matrix of the entire fam table, starting from the genotypes of the founders. The columns of this matrix correspond to fam\$id in that order. The rows (loci) are the same as in the input X.

See Also

Plink FAM format reference: <https://www.cog-genomics.org/plink/1.9/formats#fam>

Examples

```
# The smallest pedigree, two parents and a child.
# A minimal fam table with the three required columns.
# Note "mother" and "father" have missing parent IDs, while "child" does not
library(tibble)
fam <- tibble(
  id = c('father', 'mother', 'child'),
  pat = c(NA, NA, 'father'),
  mat = c(NA, NA, 'mother')
)

# genotypes of the parents at 4 loci
X <- cbind( c(1, 2, 0, 2), c(0, 2, 2, 1) )
# Name the parents with same codes as in `fam`
# (order can be different)
colnames( X ) <- c('mother', 'father')
# name loci too
rownames( X ) <- paste0( 'rs', 1:4 )
```

```
# Draw the full genotype matrix
X_all <- geno_fam( X, fam )

# This is a 4x3 matrix with column names matching fam$id.
# The parent submatrix equals the input (reordered),
# but now there's random genotypes for the child too
X_all
```

geno_last_gen	<i>Draw random genotypes for last generation of a pedigree with known founder genotypes</i>
---------------	---

Description

A wrapper around the more general `geno_fam()`, specialized to save memory when only the last generation is desired (`geno_fam()` returns genotypes for the entire pedigree in a single matrix). This function assumes that generations are non-overlapping (met by the output of `sim_pedigree()`), in which case each generation g can be drawn from generation $g-1$ data only. That way, only two consecutive generations need be in memory at any given time. The partitioning of individuals into generations is given by the `ids` parameter (again matches the output of `sim_pedigree()`).

Usage

```
geno_last_gen(X, fam, ids, missing_vals = c("", 0))
```

Arguments

<code>X</code>	The genotype matrix of the founders (loci along rows, individuals along columns). This matrix must have column names that identify each founder (matching codes in <code>fam\$id</code>). Individuals may be in a different order than <code>fam\$id</code> . Extra individuals in <code>admix</code> but absent in <code>fam\$id</code> will be silently ignored. All values should be in <code>c(0L, 1L, 2L)</code> ; for speed, this code does not check that <code>X</code> is valid (i.e. fractional values between 0 and 2 may not cause errors).
<code>fam</code>	The pedigree data.frame, in plink FAM format. Only columns <code>id</code> , <code>pat</code> , and <code>mat</code> are required. <code>id</code> must be unique and non-missing. Founders must be present, and their <code>pat</code> and <code>mat</code> values must be missing (see below). Non-founders must have both their parents be non-missing. Parents must appear earlier than their children in the table.
<code>ids</code>	A list containing vectors of IDs for each generation. All these IDs must be present in <code>fam\$id</code> . If IDs in <code>fam</code> and <code>ids</code> do not fully agree, the code processes the IDs in the intersection, which is helpful when <code>fam</code> is pruned but <code>ids</code> is the original (larger) set.
<code>missing_vals</code>	The list of ID values treated as missing. <code>NA</code> is always treated as missing. By default, the empty string (<code>"</code>) and zero (<code>0</code>) are also treated as missing (remove values from here if this is a problem).

Value

The random genotype matrix of the last generation (the intersection of `ids[length(ids)]` and `fam$id`). The columns of this matrix are last-generation individuals in the order that they appear in `fam$id`. The rows (loci) are the same as in the input `X`.

See Also

Plink FAM format reference: <https://www.cog-genomics.org/plink/1.9/formats#fam>

Examples

```
# A small pedigree, two parents and two children.
# A minimal fam table with the three required columns.
# Note "mother" and "father" have missing parent IDs, while children do not
library(tibble)
fam <- tibble(
  id = c('father', 'mother', 'child', 'sib'),
  pat = c(NA, NA, 'father', 'father'),
  mat = c(NA, NA, 'mother', 'mother')
)
# need an `ids` list separating the generations
ids <- list( c('father', 'mother'), c('child', 'sib') )

# genotypes of the parents at 4 loci
X <- cbind( c(1, 2, 0, 2), c(0, 2, 2, 1) )
# Name the parents with same codes as in `fam`
# (order can be different)
colnames( X ) <- c('mother', 'father')
# name loci too
rownames( X ) <- paste0( 'rs', 1:4 )

# Draw the genotype matrix of the children
X2 <- geno_last_gen( X, fam, ids )
X2
```

geno_last_gen_admix_recomb

Simulate an admixed family efficiently with founders with LD

Description

This function in essence combines `pop_recomb()` to simulate founders of known ancestries with LD (following a Li-Stephens-like model), draws recombination breaks of focal last-generation descendants from the specified pedigree using `recomb_last_gen()`, and their genomes from the founders variants using `recomb_haplo_inds()`. However, since a limited portion of founder sequences is actually inherited, the simulation is made much more efficient by simulating only those subsequences that were inherited, which saves time, and utilizing sparse matrices, which saves memory too. See below for a more detailed algorithm.

Usage

```

geno_last_gen_admix_recomb(
  anc_haps,
  bim,
  map,
  G,
  fam,
  ids,
  founders_anc,
  loci_on_cols = FALSE,
  missing_vals = c("", 0)
)

```

Arguments

anc_haps	A named list that maps the code used for each ancestry to its haplotype matrix. Each of the haplotype matrices the argument haps passed to <code>pop_recomb()</code> , namely is a regular matrix or <code>BEDMatrix</code> object of haplotype values, one row per locus, one column per haplotype (half individual), or transposed if <code>loci_on_cols = TRUE</code> and for <code>BEDMatrix</code> objects. Here, these values must be numeric (recommended are zeroes and ones, indicating absence or presence of reference allele).
bim	The table of variants of haps, which is a <code>data.frame/tibble</code> with at least two columns: <code>chr</code> (must be numeric between 1 and the maximum chromosome in <code>map</code> below for <code>map</code> to work) and <code>pos</code> (base pair position, usually an integer).
map	The genetic map, a list of chromosomes each of which is a <code>data.frame/tibble</code> with columns <code>pos</code> for base pair position and <code>posg</code> for genetic position.
G	Number of generations since most recent common ancestor of population (to multiply standard recombination rate)
fam	The pedigree <code>data.frame</code> , in plink FAM format. Only columns <code>id</code> , <code>pat</code> , and <code>mat</code> are required. <code>id</code> must be unique and non-missing. Founders must be present, and their <code>pat</code> and <code>mat</code> values must be missing (see below). Non-founders must have both their parents be non-missing. Parents must appear earlier than their children in the table.
ids	A list containing vectors of IDs for each generation. All these IDs must be present in <code>fam\$id</code> . If IDs in <code>fam</code> and <code>ids</code> do not fully agree, the code processes the IDs in the intersection, which is helpful when <code>fam</code> is pruned but <code>ids</code> is the original (larger) set.
founders_anc	a named vector that maps every founder haplotype (the names of this vector) to its ancestry code. Ancestry codes must match the codes used in <code>anc_haps</code> above. Founder haplotypes are the founder individual IDs from the pedigree (values in <code>ids[[1]]</code>) appearing twice, suffixed with <code>"_pat"</code> and <code>"_mat"</code> , respectively (so the parents of the founders are unadmixed, though founders be first generation admixed this way).
loci_on_cols	If <code>TRUE</code> , haps has loci on columns and individuals on rows; if <code>FALSE</code> (default), loci are on rows and individuals on columns. If haps is a <code>BEDMatrix</code> object, <code>loci_on_cols</code> is ignored (set automatically to <code>TRUE</code> internally).

`missing_vals` The list of ID values treated as missing. NA is always treated as missing. By default, the empty string ("") and zero (0) are also treated as missing (remove values from here if this is a problem).

Details

This function wraps around several exported package functions to achieve its objectives, which are roughly grouped into the following 4 phases. Phase 1 simulates recombination in the family without explicit sequences. In particular, it initializes the founder haplotype structure (without variants yet) using `recomb_init_founders()`, then simulates recombination breaks along the pedigree and identifies all the founder haplotype blocks in the focal individuals using `recomb_last_gen()`, and maps recombination breaks in cM to basepairs using `recomb_map_inds()`. Phase 2 reorganizes this data to identify the unique founder blocks that were inherited, first by making the data tidy with `tidy_recomb_map_inds()`, then applying `recomb_founder_blocks_inherited()`. Phase 3 initializes founder haplotypes using sparse matrices from the package `Matrix`, and draws inherited founder subsequences according to their known ancestries and using the Li-Stephens-like haplotype model of `pop_recomb()`. Phase 4 constructs the genotype matrices of focal individuals using the haplotypes of the founders drawn in phase 3 and the known origin of focal blocks from founders from phase 1, first constructing this data at the phased haplotype level with `recomb_haplo_inds()`, reencoding as unphased genotypes using `recomb_geno_inds()`, and constructing the corresponding local ancestry dosages using `recomb_admix_inds()`.

Value

A named list with three elements:

- `X`: the genotype matrix of the focal individuals, as returned by `recomb_geno_inds()`.
- `Ls`: a list, mapping each ancestry to its matrix of local ancestry dosages, as returned by `recomb_admix_inds()`.
- `haplos`: a phased version of the haplotypes and local ancestries of the focal individuals, structured as nested lists, as returned by `recomb_haplo_inds()`.

See Also

`recomb_init_founders()`, `recomb_last_gen()`, `recomb_map_inds()`, `tidy_recomb_map_inds()`, `recomb_founder_blocks_inherited()`, `pop_recomb()`, `recomb_haplo_inds()`, `recomb_geno_inds()`, `recomb_admix_inds()`.

Examples

```
library(tibble)

# simulate random haplotypes for example
# this toy data has 10 SNPs per chromosome, in fixed positions for simplicity
bim <- tibble( chr = rep( 1 : 22, each = 10 ), pos = rep( (1:10) * 1e6, 22 ) )
# and random haplotype data to go with this
n_ind_hap <- 10
m_loci <- nrow( bim )
# NOTE ancestry labels can be anything but must match `founders_anc` below
anc_haps <- list(
```

```

'AFR' = matrix( rbinom( m_loci * n_ind_hap, 1L, 0.5 ), nrow = m_loci, ncol = n_ind_hap ),
'EUR' = matrix( rbinom( m_loci * n_ind_hap, 1L, 0.2 ), nrow = m_loci, ncol = n_ind_hap )
)

# now simulate a very small family with one individual, 2 parents, 4 implicit grandparents
data <- fam_ancestors( 2 )
fam <- data$fam
ids <- data$ids
# select ancestries for each of the 4 grandparents / founder haplotypes (unadmixed)
founders_anc <- c('AFR', 'AFR', 'AFR', 'EUR')
# set names of founders with _pat/mat, needed to match recombination structure
# order is odd but choices were random so that doesn't matter
names( founders_anc ) <- c(
  paste0( ids[[1]], '_pat' ),
  paste0( ids[[1]], '_mat' )
)

# this performs the simulation!
data <- geno_last_gen_admix_recomb( anc_haps, bim, recomb_map_hg38, 10, fam, ids, founders_anc )
# this is the genotype matrix for the one admixed individual
data$X
# the corresponding local ancestry dosage matrices
# names match input labels
data$Ls$AFR
data$Ls$EUR
# if desired, a more complete but more complicated structure holding phased haplotypes
# and phased local ancestry information
data$haplos

```

 kinship_fam

Calculate kinship matrix of a pedigree with structured founders

Description

Calculates a full kinship matrix (between all individuals in the provided pedigree FAM table) taking into account the relatedness of the founders as provided. Output agrees with `kinship2::kinship()` but only when founders are unrelated/outbred (in other words, that function does not allow relatedness between founders).

Usage

```
kinship_fam(kinship, fam, missing_vals = c("", 0))
```

Arguments

`kinship` The kinship matrix of the founders. This matrix must have column and row names that identify each founder (matching codes in `fam$id`). Individuals may be in a different order than `fam$id`. Extra individuals in `kinship` but absent in `fam$id` will be silently ignored. A traditional pedigree calculation would

use `kinship = diag(n)/2` (plus appropriate column/row names), where `n` is the number of founders, to model unrelated and outbred founders. However, if `kinship` measures the population kinship estimates between founders, the output is also a population kinship matrix (which combines the structural/ancestral and local/pedigree relatedness values into one).

<code>fam</code>	The pedigree data.frame, in plink FAM format. Only columns <code>id</code> , <code>pat</code> , and <code>mat</code> are required. <code>id</code> must be unique and non-missing. Founders must be present, and their <code>pat</code> and <code>mat</code> values must be missing (see below). Non-founders must have both their parents be non-missing. Parents must appear earlier than their children in the table.
<code>missing_vals</code>	The list of ID values treated as missing. <code>NA</code> is always treated as missing. By default, the empty string (<code>"</code>) and zero (<code>0</code>) are also treated as missing (remove values from here if this is a problem).

Value

The kinship matrix of the entire `fam` table, taking the relatedness of the founders into account. The rows and columns of this kinship matrix correspond to `fam$id` in that order.

See Also

Plink FAM format reference: <https://www.cog-genomics.org/plink/1.9/formats#fam>

Examples

```
# The smallest pedigree, two parents and a child.
# A minimal fam table with the three required columns.
# Note "mother" and "father" have missing parent IDs, while "child" does not
library(tibble)
fam <- tibble(
  id = c('father', 'mother', 'child'),
  pat = c(NA, NA, 'father'),
  mat = c(NA, NA, 'mother')
)

# Kinship of the parents, here two unrelated/outbred individuals:
kinship <- diag(2)/2
# Name the parents with same codes as in `fam`
# (order can be different)
colnames( kinship ) <- c('mother', 'father')
rownames( kinship ) <- c('mother', 'father')
# For a clearer example, make the father slightly inbred
# (a self-kinship value that exceeds 1/2):
kinship[2,2] <- 0.6

# Calculate the full kinship matrix
kinship_all <- kinship_fam( kinship, fam )

# This is a 3x3 matrix with row/col names matching fam$id.
# The parent submatrix equals the input (reordered),
# but now there's relatedness to the child too
```

kinship_all

kinship_last_gen	<i>Calculate kinship matrix for last generation of a pedigree with structured founders</i>
------------------	--

Description

A wrapper around the more general `kinship_fam()`, specialized to save memory when only the last generation is desired (`kinship_fam()` returns kinship for the entire pedigree in a single matrix). This function assumes that generations are non-overlapping (met by the output of `sim_pedigree()`), in which case each generation g can be drawn from generation $g-1$ data only. That way, only two consecutive generations need be in memory at any given time. The partitioning of individuals into generations is given by the `ids` parameter (again matches the output of `sim_pedigree()`).

Usage

```
kinship_last_gen(kinship, fam, ids, missing_vals = c("", 0))
```

Arguments

kinship	The kinship matrix of the founders. This matrix must have column and row names that identify each founder (matching codes in <code>fam\$id</code>). Individuals may be in a different order than <code>fam\$id</code> . Extra individuals in <code>kinship</code> but absent in <code>fam\$id</code> will be silently ignored. A traditional pedigree calculation would use <code>kinship = diag(n)/2</code> (plus appropriate column/row names), where n is the number of founders, to model unrelated and outbred founders. However, if <code>kinship</code> measures the population kinship estimates between founders, the output is also a population kinship matrix (which combines the structural/ancestral and local/pedigree relatedness values into one).
fam	The pedigree data.frame, in plink FAM format. Only columns <code>id</code> , <code>pat</code> , and <code>mat</code> are required. <code>id</code> must be unique and non-missing. Founders must be present, and their <code>pat</code> and <code>mat</code> values must be missing (see below). Non-founders must have both their parents be non-missing. Parents must appear earlier than their children in the table.
ids	A list containing vectors of IDs for each generation. All these IDs must be present in <code>fam\$id</code> . If IDs in <code>fam</code> and <code>ids</code> do not fully agree, the code processes the IDs in the intersection, which is helpful when <code>fam</code> is pruned but <code>ids</code> is the original (larger) set.
missing_vals	The list of ID values treated as missing. NA is always treated as missing. By default, the empty string ("") and zero (0) are also treated as missing (remove values from here if this is a problem).

Value

The kinship matrix of the last generation (the intersection of `ids[length(ids)]` and `fam$id`). The columns/rows of this matrix are last-generation individuals in the order that they appear in `fam$id`.

See Also

Plink FAM format reference: <https://www.cog-genomics.org/plink/1.9/formats#fam>

Examples

```
# A small pedigree, two parents and two children.
# A minimal fam table with the three required columns.
# Note "mother" and "father" have missing parent IDs, while children do not
library(tibble)
fam <- tibble(
  id = c('father', 'mother', 'child', 'sib'),
  pat = c(NA, NA, 'father', 'father'),
  mat = c(NA, NA, 'mother', 'mother')
)
# need an `ids` list separating the generations
ids <- list( c('father', 'mother'), c('child', 'sib') )

# Kinship of the parents, here two unrelated/outbred individuals:
kinship <- diag(2)/2
# Name the parents with same codes as in `fam`
# (order can be different)
colnames( kinship ) <- c('mother', 'father')
rownames( kinship ) <- c('mother', 'father')
# For a clearer example, make the father slightly inbred
# (a self-kinship value that exceeds 1/2):
kinship[2,2] <- 0.6

# calculate the kinship matrix of the children
kinship2 <- kinship_last_gen( kinship, fam, ids )
kinship2
```

pop_recomb

Simulate genotypes with linkage disequilibrium (LD) given a population of haplotypes, using a Li-Stephens-like model of haplotype copying

Description

Each new genome has breaks drawn from the recombination model accelerated for the desired number of generations G , and haplotypes are drawn and copied randomly from the population. This results in a population with individuals drawn independently and identically distributed but with LD within individuals, since the ancestral LD is preserved to some extent (attenuated by recombination

after G generations). If there is no LD in haps, then the output will not have LD either except when the number of columns of haps is very small (which resembles a bottleneck). This model does not introduce mutations (unlike the original Li-Stephens). Genotypes, when requested, are simply sums of independently drawn haplotype values.

Usage

```
pop_recomb(
  haps,
  bim,
  map,
  G,
  n_ind,
  geno = TRUE,
  loci_on_cols = FALSE,
  indexes_loci = NULL
)
```

Arguments

haps	Regular matrix or <code>BEDMatrix</code> object of haplotype values, one row per locus, one column per haplotype (half individual), or transposed if <code>loci_on_cols = TRUE</code> and for <code>BEDMatrix</code> objects. If <code>geno = TRUE</code> (default), these values should be numeric (recommended are zeroes and ones, indicating absence or presence of reference allele), but if <code>geno = FALSE</code> code will work with any values, including strings, which are just copied to outputs in blocks.
bim	The table of variants of haps, which is a <code>data.frame/tibble</code> with at least two columns: <code>chr</code> (must be numeric between 1 and the maximum chromosome in <code>map</code> below for <code>map</code> to work) and <code>pos</code> (base pair position, usually an integer).
map	The genetic map, a list of chromosomes each of which is a <code>data.frame/tibble</code> with columns <code>pos</code> for base pair position and <code>posg</code> for genetic position.
G	Number of generations since most recent common ancestor of population (to multiply standard recombination rate)
n_ind	Number of individuals (if <code>geno = TRUE</code>) or haplotypes (half individuals, if <code>geno = FALSE</code>) desired in output
geno	If <code>TRUE</code> (default) returns matrix of genotypes (values in 0,1,2 if haps is binary, otherwise double whatever the range of values in haps is), otherwise returns matrix of haplotypes (half individuals, same values of input haps)
loci_on_cols	If <code>TRUE</code> , haps has loci on columns and individuals on rows; if <code>FALSE</code> (default), loci are on rows and individuals on columns. If haps is a <code>BEDMatrix</code> object, <code>loci_on_cols</code> is ignored (set automatically to <code>TRUE</code> internally).
indexes_loci	Vector of indexes of loci to simulate, to request to simulate only a subset of the loci in haps/bim. Default <code>NULL</code> simulates all loci in haps/bim.

Value

A matrix with the same number of rows as haps and n_ind columns, with values copied from haps in (recombination) blocks if geno = FALSE, or sums of two such values drawn independently when geno = TRUE.

Examples

```
# simulate a tiny population with few SNPs for example
library(tibble)
bim <- tibble(
  chr = c(2, 2, 3, 3, 22),
  pos = c(100, 121, 53, 154, 66) * 1e6
)
m_loci <- nrow( bim )
# Most often, haplotypes are binary data as simulated here.
# Here haplotypes will be totally unstructured, but to have LD in the output use real human data
# or data simulated to have LD
n_ind_haps <- 5
haps <- matrix(
  rbinom( m_loci * n_ind_haps, 1, 0.5 ),
  nrow = m_loci,
  ncol = n_ind_haps
)
# makes sense to have a lot of recombination at the population level
G <- 500
# ask for small output for example
n_ind <- 7
# use the recombination map for the same genome build as your data!
map <- recomb_map_hg38

# simulate genotypes! (Usually more convenient, but phase information is lost)
X <- pop_recomb( haps, bim, map, G, n_ind )

# simulate haplotypes instead (preserves true phase)
H <- pop_recomb( haps, bim, map, G, n_ind, geno = FALSE )
```

prune_fam

Remove non-ancestors of a set of individuals from pedigree

Description

This function accepts an input pedigree and a list of individuals of interest, and returns the subset of the pedigree including only the individuals of interest and their direct ancestors. This is useful in simulations, to avoid modeling/drawing genotypes of individuals without descendants in the last generation.

Usage

```
prune_fam(fam, ids, missing_vals = c("", 0))
```

Arguments

fam	The pedigree data.frame, in plink FAM format. Only columns id, pat, and mat are required. id must be unique and non-missing. Founders must be present, and their pat and mat values must be missing (see below). Non-founders must have both their parents be non-missing. Parents must appear earlier than their children in the table.
ids	The list of individuals of interest, whose ancestors we want to keep. All must be present in fam\$id.
missing_vals	The list of ID values treated as missing. NA is always treated as missing. By default, the empty string ("") and zero (0) are also treated as missing (remove values from here if this is a problem).

Value

The filtered FAM table with non-ancestors of ids excluded. IDs that are NA-equivalent (see missing_vals) will be mapped to NA.

Examples

```
# construct a family with three founders, but one "bob" has no descendants
library(tibble)
fam <- tibble(
  id = c('mom', 'dad', 'bob', 'child'),
  pat = c( NA, NA, NA, 'dad'),
  mat = c( NA, NA, NA, 'mom')
)
# only want 'child' and its ancestors
ids <- 'child'
fam2 <- prune_fam( fam, ids )
# the filtered pedigree has "bob" removed:
fam2
```

recomb_admix_inds	<i>Reduce haplotype ancestry data to population ancestry dosage matrices</i>
-------------------	--

Description

This function accepts haplotype data, such as the output from [recomb_haplo_inds\(\)](#) with `ret_anc = TRUE` (required), and reduces it to a list of population ancestry dosage matrices. In this context, "ancestors/ancestry" refer to haplotype blocks from specific ancestor individuals, whereas "population ancestry" groups these ancestors into populations (such as African, European, etc.). Although the haplotype data separates individuals and chromosomes into lists (the way it is simulated), the output matrices concatenates data from all chromosomes into a single matrix, as it appears in simpler simulations and real data, and matching the format of [recomb_genos_inds\(\)](#).

Usage

```
recomb_admix_inds(haplos, anc_map, pops = sort(unique(anc_map$pop)))
```

Arguments

haplos	A list of diploid individuals, each of which is a list with two haploid individuals named pat and mat, each of which is a list of chromosomes, each of which must be a list with a named element anc must give the vector of ancestor names per position (the output format from recomb_haplo_inds() with ret_anc = TRUE).
anc_map	A data.frame or tibble with two columns: anc lists every ancestor haplotype name present in haplos, and pop the population assignment of that haplotype.
pops	Optional order of populations in output, by default sorted alphabetically from anc_map\$pop.

Value

A named list of population ancestry dosage matrices, ordered as in pops, each of which counts populations in both alleles (in 0, 1, 2), with individuals along columns in same order as haplos list, and loci along rows in order of appearance concatenating chromosomes in numerical order.

See Also

[recomb_fam\(\)](#) for drawing recombination (ancestor) blocks, defined in terms of genetic distance.
[recomb_map_inds\(\)](#) for transforming genetic to basepair coordinates given a genetic map.
[recomb_haplo_inds\(\)](#) for determining haplotypes of descendants given ancestral haplotypes (creates input to this function).

Examples

```
# Lengthy code creates individuals with recombination data to map
# The smallest pedigree, two parents and a child (minimal fam table).
library(tibble)
fam <- tibble(
  id = c('father', 'mother', 'child'),
  pat = c(NA, NA, 'father'),
  mat = c(NA, NA, 'mother')
)
# use latest human recombination map, but just first two chrs to keep this example fast
map <- recomb_map_hg38[ 1L:2L ]
# initialize parents with this other function
founders <- recomb_init_founders( c('father', 'mother'), map )
# draw recombination breaks for child
inds <- recomb_fam( founders, fam )
# now add base pair coordinates to recombination breaks
inds <- recomb_map_inds( inds, map )

# also need ancestral haplotypes
# these should be simulated carefully as needed, but for this example we make random data
haplo <- vector( 'list', length( map ) )
```

```

# names of ancestor haplotypes for this scenario
# (founders of fam$id but each with "_pat" and "_mat" suffixes)
anc_names <- c( 'father_pat', 'father_mat', 'mother_pat', 'mother_mat' )
n_ind <- length( anc_names )
# number of loci per chr, for toy test
m_loci <- 10L
for ( chr in 1L : length( map ) ) {
  # draw random positions
  pos_chr <- sample.int( max( map[[ chr ]]$pos ), m_loci )
  # draw haplotypes
  X_chr <- matrix(
    rbinom( m_loci * n_ind, 1L, 0.5 ),
    nrow = m_loci,
    ncol = n_ind
  )
  # required column names!
  colnames( X_chr ) <- anc_names
  # add to structure, in a list
  haplo[[ chr ]] <- list( X = X_chr, pos = pos_chr )
}
# determine haplotypes and per-position ancestries of descendants given ancestral haplotypes
haplos <- recomb_haplo_inds( inds, haplo, ret_anc = TRUE )

# define individual to population ancestry map
# take four ancestral haplotypes from above, assign them population labels
anc_map <- tibble(
  anc = anc_names,
  pop = c('African', 'European', 'African', 'African')
)

# finally, run desired function!
# convert haplotypes structure to list of population ancestry dosage matrices
Xs <- recomb_admix_inds( haplos, anc_map )

```

recomb_fam

Draw recombination breaks for autosomes from a pedigree

Description

Create random recombination breaks for all autosomes of all individuals in the provided pedigree FAM table. Recombination lengths follow an exponential distribution with mean of 100 centiMorgans (cM). The output specifies identical-by-descent (IBD) blocks as ranges per chromosome (per individual) and the founder chromosome they arose from (are IBD with). All calculations are in terms of genetic distance (not base pairs), and no genotypes are constructed/drawn in this step.

Usage

```
recomb_fam(founders, fam, missing_vals = c("", 0))
```

Arguments

founders	The named list of founders with their chromosomes. For unstructured founders, initialize with <code>recomb_init_founders()</code> . Each element of this list is a diploid individual, which is a list with two haploid individuals named <code>pat</code> and <code>mat</code> , each of which is a list of chromosomes (always identified by number, but may also be named arbitrarily), each of which is a <code>data.frame/tibble</code> with implicit ranges (<code>posg</code> is end coordinates in cM; <code>start</code> is the end of the previous block, zero for the first block) and ancestors <code>anc</code> as strings. For true founders each chromosome may be trivial (each chromosome is a single block with ID equal to itself but distinguishing maternal from paternal copy), but input itself can be recombined (for iterating). This list must have names that identify each founder (matching codes in <code>fam\$id</code>). Individuals may be in a different order than <code>fam\$id</code> . Extra individuals in <code>founders</code> but absent in <code>fam\$id</code> will be silently ignored.
fam	The pedigree <code>data.frame</code> , in plink FAM format. Only columns <code>id</code> , <code>pat</code> , and <code>mat</code> are required. <code>id</code> must be unique and non-missing. Founders must be present, and their <code>pat</code> and <code>mat</code> values must be missing (see below). Non-founders must have both their parents be non-missing. Parents must appear earlier than their children in the table.
missing_vals	The list of ID values treated as missing. <code>NA</code> is always treated as missing. By default, the empty string (<code>"</code>) and zero (<code>0</code>) are also treated as missing (remove values from here if this is a problem).

Value

The list of individuals with recombined chromosomes of the entire `fam` table, in the same format as `founders` above. The names of this list correspond to `fam$id` in that order.

See Also

`recomb_init_founders()` to initialize `founders` for this function.

Plink FAM format reference: <https://www.cog-genomics.org/plink/1.9/formats#fam>

Examples

```
# The smallest pedigree, two parents and a child.
# A minimal fam table with the three required columns.
# Note "mother" and "father" have missing parent IDs, while "child" does not
library(tibble)
fam <- tibble(
  id = c('father', 'mother', 'child'),
  pat = c(NA, NA, 'father'),
  mat = c(NA, NA, 'mother')
)

# initialize parents with this other function
# Name the parents with same codes as in `fam`
# (order can be different)
ids <- c('mother', 'father')
# simulate three chromosomes with these lengths in cM
```

```
lengs <- c(50, 100, 150)
founders <- recomb_init_founders( ids, lengs )

# draw recombination breaks for the whole fam table now:
inds <- recomb_fam( founders, fam )

# This is a length-3 list with names matching fam$id.
# The parent data equals the input (reordered),
# but now there's data to the child too
inds
```

recomb_founder_blocks_inherited

Identify IBD blocks of founders that were inherited by at least one individual in this data

Description

Transforms a tidy table describing the founder blocks inherited by each individual into a more summarized table describing the contiguous blocks of each founder inherited by at least one individual. This is especially useful when the focal individuals are several generations removed from the founders so only a small fraction of founder chromosomes are inherited.

Usage

```
recomb_founder_blocks_inherited(inds)
```

Arguments

inds The tidy table of recombination block data inherited in individuals from founders, such as the return value of `tidy_recomb_map_inds()`. More broadly, it is a table with a row for every IBD block, and at least these columns: "anc" is the label of the founder individual, "chr" is the chromosome number, "start" and "end" are the range of the block. Although this data usually also has columns "ind" and "parent" identifying the individual and parental haplotype that inherited the block, this information is not used by this function and is removed if present.

Value

A table in a similar format as the input (columns "anc", "chr", "start", and "end") where each row is a contiguous IBD block whose each basepair is inherited by at least one individual.

See Also

[tidy_recomb_map_inds\(\)](#)

Examples

```

# manually construct a toy sample input,
# with individuals marked for clarity although they are not required
# note first two individuals
library(tibble)
inds <- tibble(
  ind = letters[1:3],
  parent = c('pat', 'mat', 'mat'),
  chr = c(1, 1, 2),
  start = c( 1, 800, 5),
  end = c(1000, 2000, 100),
  anc = c('f1', 'f1', 'f2')
)

# the new table merges the first two rows,
# because they overlapped from the same ancestor,
# while the third row stays unchanged (after removing individual info)
founder_blocks <- recomb_founder_blocks_inherited( inds )

```

recomb_genos_inds

*Reduce haplotype data to genotype matrix***Description**

This function accepts haplotype data, such as the output from `recomb_haplo_inds()`, and reduces it to a genotype matrix. The haplotype data is more detailed because it is phased, while phase is lost in the genotype representation. Moreover, the haplotype data separates individuals and chromosomes into lists (the way it is simulated), but the output genotype matrix concatenates data from all chromosomes into a single matrix, as it appears in simpler simulations and real data.

Usage

```
recomb_genos_inds(haplos)
```

Arguments

haplos	A list of diploid individuals, each of which is a list with two haploid individuals named "pat" and "mat", each of which is a list of chromosomes. Each chromosome can be a list, in which case the named element "x" must give the haplotype vector (ideally with values in zero and one counting reference alleles, including NA), otherwise the chromosome must be this vector (accommodating both output formats from <code>recomb_haplo_inds()</code> automatically).
--------	--

Value

The genotype matrix, which is the sum of the haplotype values (with values in 0, 1, 2, and NA, counting reference alleles), with individuals along columns in same order as haplos list, and loci along rows in order of appearance concatenating chromosomes in numerical order.

See Also

[recomb_fam\(\)](#) for drawing recombination (ancestor) blocks, defined in terms of genetic distance.
[recomb_map_inds\(\)](#) for transforming genetic to basepair coordinates given a genetic map.
[recomb_haplo_inds\(\)](#) for determining haplotypes of descendants given ancestral haplotypes (creates input to this function).

Examples

```
# Lengthy code creates individuals with recombination data to map
# The smallest pedigree, two parents and a child (minimal fam table).
library(tibble)
fam <- tibble(
  id = c('father', 'mother', 'child'),
  pat = c(NA, NA, 'father'),
  mat = c(NA, NA, 'mother')
)
# use latest human recombination map, but just first two chrs to keep this example fast
map <- recomb_map_hg38[ 1L:2L ]
# initialize parents with this other function
founders <- recomb_init_founders( c('father', 'mother'), map )
# draw recombination breaks for child
inds <- recomb_fam( founders, fam )
# now add base pair coordinates to recombination breaks
inds <- recomb_map_inds( inds, map )

# also need ancestral haplotypes
# these should be simulated carefully as needed, but for this example we make random data
haplo <- vector( 'list', length( map ) )
# names of ancestor haplotypes for this scenario
# (founders of fam$id but each with "_pat" and "_mat" suffixes)
anc_names <- c( 'father_pat', 'father_mat', 'mother_pat', 'mother_mat' )
n_ind <- length( anc_names )
# number of loci per chr, for toy test
m_loci <- 10L
for ( chr in 1L : length( map ) ) {
  # draw random positions
  pos_chr <- sample.int( max( map[[ chr ]]$pos ), m_loci )
  # draw haplotypes
  X_chr <- matrix(
    rbinom( m_loci * n_ind, 1L, 0.5 ),
    nrow = m_loci,
    ncol = n_ind
  )
  # required column names!
  colnames( X_chr ) <- anc_names
  # add to structure, in a list
  haplo[[ chr ]] <- list( X = X_chr, pos = pos_chr )
}
# determine haplotypes of descendants given ancestral haplotypes
haplos <- recomb_haplo_inds( inds, haplo )
```

```
# finally, run desired function!
# convert haplotypes structure to a plain genotype matrix
X <- recomb_geno_inds( haplos )
```

recomb_haplo_inds *Construct haplotypes of individuals given their ancestral blocks and the ancestral haplotype variants*

Description

Construct haplotypes of individuals given their ancestral blocks and the ancestral haplotype variants

Usage

```
recomb_haplo_inds(inds, haplo, ret_anc = FALSE)
```

Arguments

inds	A list of individuals in the same format as the output of recomb_fam() after being processed with recomb_map_inds() . More specifically, each individual is a list with two haploid individuals named pat and mat, each of which is a list of chromosomes (always identified by number, but may also be named arbitrarily), each of which is a data.frame/tibble with implicit ranges (pos is end coordinates in base pairs; start is the end of the previous block plus one, 1 for the first block) and ancestors anc as strings.
haplo	The ancestral haplotypes, which is a list of chromosomes, each of which is a list with two named elements: X is a matrix of haplotype markers (loci along rows, ancestral individuals along columns, which must be named as in anc strings in inds above), and pos is a vector of locus positions in base pair coordinates. Haplotype matrix can be a sparse matrix from the package Matrix.
ret_anc	If TRUE, returns local ancestries (per position) along with haplotypes, otherwise only haplotypes are returned.

Value

A list of diploid individuals, each of which is a list with two haploid individuals named pat and mat, each of which is a list of chromosomes. If `ret_anc = FALSE` (default), each chromosome is a haplotype (vector of values copied from ancestors in haplo); if `ret_anc = TRUE`, each chromosome is a list with named elements x for the haplotype vector and anc for the vector of ancestor name per position.

See Also

[recomb_fam\(\)](#) for drawing recombination (ancestor) blocks, defined in terms of genetic distance.
[recomb_map_inds\(\)](#) for transforming genetic to basepair coordinates given a genetic map.
[recomb_geno_inds\(\)](#) for transforming the output of this function from haplotypes (a nested lists structure) to a plain genotype matrix.

Examples

```

# Lengthy code creates individuals with recombination data to map
# The smallest pedigree, two parents and a child (minimal fam table).
library(tibble)
fam <- tibble(
  id = c('father', 'mother', 'child'),
  pat = c(NA, NA, 'father'),
  mat = c(NA, NA, 'mother')
)
# use latest human recombination map, but just first two chrs to keep this example fast
map <- recomb_map_hg38[ 1L:2L ]
# initialize parents with this other function
founders <- recomb_init_founders( c('father', 'mother'), map )
# draw recombination breaks for child
inds <- recomb_fam( founders, fam )
# now add base pair coordinates to recombination breaks
inds <- recomb_map_inds( inds, map )

# also need ancestral haplotypes
# these should be simulated carefully as needed, but for this example we make random data
haplo <- vector( 'list', length( map ) )
# names of ancestor haplotypes for this scenario
# (founders of fam$id but each with "_pat" and "_mat" suffixes)
anc_names <- c( 'father_pat', 'father_mat', 'mother_pat', 'mother_mat' )
n_ind <- length( anc_names )
# number of loci per chr, for toy test
m_loci <- 10L
for ( chr in 1L : length( map ) ) {
  # draw random positions
  pos_chr <- sample.int( max( map[[ chr ]]$pos ), m_loci )
  # draw haplotypes
  X_chr <- matrix(
    rbinom( m_loci * n_ind, 1L, 0.5 ),
    nrow = m_loci,
    ncol = n_ind
  )
  # required column names!
  colnames( X_chr ) <- anc_names
  # add to structure, in a list
  haplo[[ chr ]] <- list( X = X_chr, pos = pos_chr )
}

# finally, run desired function!
# determine haplotypes of descendants given ancestral haplotypes
data <- recomb_haplo_inds( inds, haplo )

```

Description

This function initializes what is otherwise a tedious structure for founders, to be used for simulating recombination in a pedigree. The genetic structure is trivial, in that these "founder" chromosomes are each of a single ancestral individual (none are recombined).

Usage

```
recomb_init_founders(ids, lengs)
```

Arguments

ids	The list of IDs to use for each individual
lengs	The lengths of each chromosome in centiMorgans (cM). If this vector is named, the output inherits these chromosome names. If it is a list, it is assumed to be a recombination map (see recomb_map_hg for examples) and the desired lengths extracted automatically (taken as the last value of column posg of each chromosome).

Value

A named list of diploid individuals, each of which is a list with two haploid individuals named pat and mat, each of which is a list of chromosomes (inherits names of lengs if present), each of which is a tibble with a single row and two columns: posg equals the chromosome length, and anc equals the ID of the individual (from ids) concatenated with either _pat or _mat depending on which parent it is.

See Also

[recomb_fam\(\)](#) to simulate recombination across a pedigree using the founders initialized here.

Examples

```
# version with explicit recombination lengths
ancs <- recomb_init_founders( c('a', 'b'), c(100, 200) )
ancs

# version using genetic map (uses provided human map) from which lengths are extracted
ancs <- recomb_init_founders( c('a', 'b'), recomb_map_hg38 )
ancs
```

recomb_last_gen	<i>Draw recombination breaks for autosomes for last generation of a pedigree</i>
-----------------	--

Description

A wrapper around the more general `recomb_fam()`, specialized to save memory when only the last generation is desired (`recomb_fam()` returns recombination blocks for the entire pedigree). This function assumes that generations are non-overlapping (met by the output of `sim_pedigree()`), in which case each generation g can be drawn from generation $g-1$ data only. That way, only two consecutive generations need be in memory at any given time. The partitioning of individuals into generations is given by the `ids` parameter (again matches the output of `sim_pedigree()`).

Usage

```
recomb_last_gen(founders, fam, ids, missing_vals = c("", 0))
```

Arguments

<code>founders</code>	The named list of founders with their chromosomes. For unstructured founders, initialize with <code>recomb_init_founders()</code> . Each element of this list is a diploid individual, which is a list with two haploid individuals named <code>pat</code> and <code>mat</code> , each of which is a list of chromosomes (always identified by number, but may also be named arbitrarily), each of which is a <code>data.frame/tibble</code> with implicit ranges (<code>posg</code> is end coordinates in cM; <code>start</code> is the end of the previous block, zero for the first block) and ancestors <code>anc</code> as strings. For true founders each chromosome may be trivial (each chromosome is a single block with ID equal to itself but distinguishing maternal from paternal copy), but input itself can be recombined (for iterating). This list must have names that identify each founder (matching codes in <code>fam\$id</code>). Individuals may be in a different order than <code>fam\$id</code> . Extra individuals in <code>founders</code> but absent in <code>fam\$id</code> will be silently ignored.
<code>fam</code>	The pedigree <code>data.frame</code> , in plink FAM format. Only columns <code>id</code> , <code>pat</code> , and <code>mat</code> are required. <code>id</code> must be unique and non-missing. Founders must be present, and their <code>pat</code> and <code>mat</code> values must be missing (see below). Non-founders must have both their parents be non-missing. Parents must appear earlier than their children in the table.
<code>ids</code>	A list containing vectors of IDs for each generation. All these IDs must be present in <code>fam\$id</code> . If IDs in <code>fam</code> and <code>ids</code> do not fully agree, the code processes the IDs in the intersection, which is helpful when <code>fam</code> is pruned but <code>ids</code> is the original (larger) set.
<code>missing_vals</code>	The list of ID values treated as missing. NA is always treated as missing. By default, the empty string ("") and zero (0) are also treated as missing (remove values from here if this is a problem).

Value

The list of individuals with recombined chromosomes of the last generation (the intersection of `ids[length(ids)]` and `fam$id`), in the same format as `founders` above. The names of this list are last-generation individuals in the order that they appear in `fam$id`.

See Also

Plink FAM format reference: <https://www.cog-genomics.org/plink/1.9/formats#fam>

Examples

```
# A small pedigree, two parents and two children.
# A minimal fam table with the three required columns.
# Note "mother" and "father" have missing parent IDs, while children do not
library(tibble)
fam <- tibble(
  id = c('father', 'mother', 'child', 'sib'),
  pat = c(NA, NA, 'father', 'father'),
  mat = c(NA, NA, 'mother', 'mother')
)
# need an `ids` list separating the generations
ids <- list( c('father', 'mother'), c('child', 'sib') )

# initialize parents with this other function
# simulate three chromosomes with these lengths in cM
lengs <- c(50, 100, 150)
founders <- recomb_init_founders( ids[[1]], lengs )

# draw recombination breaks for the children
inds <- recomb_last_gen( founders, fam, ids )
```

```
recomb_map_fix_ends_chr
```

Extrapolate and shift recombination map of one chromosome to ends

Description

Given an existing recombination map and a chromosome length in base pairs, extrapolates the map to ensure all positions are covered, and shifts to ensure position one in basepairs corresponds to position 0 in genetic position. Recombination rates are extrapolated from the first and last 10Mb of data by default (separately per end). Therefore fixes the fact that common maps start genetic position zero at base pair position >> 1 and do not extend to ends (some SNPs from modern projects fall out of range without fixes).

Usage

```
recomb_map_fix_ends_chr(map, pos_length, pos_delta = 10000000L)
```

Arguments

map	A tibble with two columns: pos position in base pairs, and posg position in centiMorgans (cM).
pos_length	The length of the chromosome in base pairs.
pos_delta	The size of the window used to extrapolate recombination rates.

Value

The extrapolated recombination map, shifted so the first non-trivial position maps to the genetic distance expected from the extrapolated rate at the beginning, then added a first trivial position (pos=1, posg=0) and final basepair position at length of chromosome and expected genetic position from end extrapolation.

See Also

[recomb_map_simplify_chr\(\)](#) to simplify recombination maps to a desired numerical accuracy.

Examples

```
library(tibble)
# create a toy recombination map with at least 10Mb at each end
map <- tibble(
  pos = c( 3L, 15L, 100L, 120L ) * 1e6L,
  posg = c( 0, 10.4, 90.1, 110 )
)
# and length
pos_length <- 150L * 1e6L

# apply function!
map_fixed <- recomb_map_fix_ends_chr( map, pos_length )
# inspect
map_fixed
```

recomb_map_hg

Simplified recombination maps for human genomes

Description

Human genetic recombination maps for builds 38 (GRCh38/hg38) and 37 (GRCh37/hg19, below suffixed as hg37 for simplicity although technically incorrect). Processed each first with [recomb_map_fix_ends_chr\(\)](#) to shift and extrapolate to sequence ends, then simplified with [recomb_map_simplify_chr\(\)](#) to remove all values that can be interpolated with an error of up to $\text{tol} = 0.1$, in order to reduce their sizes and interpolation runtime. Defaults were used, which resulted in extrapolated recombination rates close to and centered around the average of $1e-6$ cM/base). Autosomes only.

Usage

```
recomb_map_hg38
```

```
recomb_map_hg37
```

Format

A list with 22 elements (autosomes, not named), each a tibble with two columns defining the recombination map at that chromosome:

- pos: position in base pairs
- posg: position in centiMorgans (cM)

An object of class `list` of length 22.

Source

Raw genetic maps downloaded from this location prior to above processing: https://bochet.gcc.biostat.washington.edu/beagle/genetic_maps/

Chromosome lengths from: <https://www.ncbi.nlm.nih.gov/grc/human/data>

recomb_map_inds	<i>Map recombination breaks from genetic positions to base pair coordinates</i>
-----------------	---

Description

Given a list of individuals with recombination breaks given in genetic distance (such as the output of `recomb_fam()`), and a genetic map (see `recomb_map_hg`), this function determines all positions in base pair coordinates. If base pair positions existed in input, they are overwritten.

Usage

```
recomb_map_inds(inds, map)
```

Arguments

inds	The list of individuals, each of which is a list with two haploid individuals named <code>pat</code> and <code>mat</code> , each of which is a list of chromosomes (always identified by number, but may also be named arbitrarily), each of which is a <code>data.frame/tibble</code> with implicit ranges (<code>posg</code> is end coordinates in cM; <code>start</code> is the end of the previous block, zero for the first block) and ancestors <code>anc</code> as strings.
map	The genetic map, a list of chromosomes each of which is a <code>data.frame/tibble</code> with columns <code>pos</code> for base pair position and <code>posg</code> for genetic position.

Details

Genetic positions are converted to base pair positions from the provided map using linear interpolation, using `stats::approx()` with options `rule = 2` (out of range cases are set to nearest end's value) and `ties = list('ordered', mean)` (assume data is ordered, interpolate ties in genetic position in map using mean of base pair positions). Output will be incorrect, without throwing errors, if genetic map is not ordered. Base pair positions are rounded to integers.

Value

The input list of individuals, with each chromosome added column `pos` corresponding to end coordinate in base pairs. Each chromosome has columns reordered so `pos`, `posg`, and `anc` appear first, and any additional columns appear afterwards.

See Also

[recomb_fam\(\)](#) for drawing recombination breaks of individuals from a pedigree.

[recomb_map_hg](#) for simplified human recombination maps included in this package.

Examples

```
# Lengthy code creates individuals with recombination data to map
# The smallest pedigree, two parents and a child (minimal fam table).
library(tibble)
fam <- tibble(
  id = c('father', 'mother', 'child'),
  pat = c(NA, NA, 'father'),
  mat = c(NA, NA, 'mother')
)
# use latest human recombination map, but just first two chrs to keep this example fast
map <- recomb_map_hg38[ 1:2 ]
# initialize parents with this other function
founders <- recomb_init_founders( c('father', 'mother'), map )
# draw recombination breaks for child
inds <- recomb_fam( founders, fam )

# now use this function to add base pair coordinates for recombination breaks!
inds <- recomb_map_inds( inds, map )
```

recomb_map_simplify_chr

Simplify recombination map of one chromosome to a desired numerical precision

Description

Given an input recombination map, this function iteratively removes rows that can be interpolated to less than a given error `tol`. This is a heuristic that works very well in practice, resulting in average interpolation errors well below `tol`, and maximum final errors no greater than $3 * tol$ in our internal benchmarks (expected in extremely concave or convex regions of the map; final errors are rarely above `tol` with few exceptions).

Usage

```
recomb_map_simplify_chr(map, tol = 0.1)
```

Arguments

<code>map</code>	A tibble with two columns: <code>pos</code> position in base pairs, and <code>posg</code> position in centiMorgans (cM).
<code>tol</code>	Tolerance of interpolation errors, in cM.

Details

This function reduces recombination map sizes drastically, in order to include them in packages, and also makes linear interpolation faster. This simplification operation can be justified as the precision of many existing maps is both limited and overstated, and a high accuracy is not needed for simulations with many other approximations in place.

Value

The recombination map with rows (positions) removed (if they are interpolated with errors below `tol` in most cases).

See Also

[recomb_map_fix_ends_chr\(\)](#) to shift and extrapolate recombination map to ends of chromosome.

Examples

```
library(tibble)
# create a toy recombination map to simplify
# in this case all middle rows can be interpolated from the ends with practically no error
map <- tibble(
  pos = c( 1L, 1e6L, 2e6L, 3e6L ),
  posg = c( 0.0, 1.0, 2.0, 3.0 )
)

# simplify map!
map_simple <- recomb_map_simplify_chr( map )
# inspect
map_simple
```

sim_pedigree

*Construct a random pedigree***Description**

Specify the number of individuals per generation, and some other optional parameters, and a single pedigree with those properties will be simulated, where close relatives are never paired, sex is drawn randomly per individual and pairings are strictly across opposite-sex individuals, and otherwise closest individuals (on an underlying 1D geography given by their index) are paired in a random order. Pairs are reordered based on the average of their indexes, where their children are placed (determines their indexes in the 1D geography). The procedure may leave some individuals unpaired in the next generation, and family sizes vary randomly (with a fixed minimum family size) to achieve the desired population size in each generation.

Usage

```
sim_pedigree(
  n,
  G = length(n),
  sex = draw_sex(n[1]),
  kinship_local = diag(n[1])/2,
  cutoff = 1/4^3,
  children_min = 1L,
  full = FALSE
)
```

Arguments

n	The number of individuals per generation. If scalar, the number of generations $G \geq 2$ must also be specified. Otherwise, the length of n is the number of generations.
G	The number of generations (optional). Note $G = 1$ is founders only, so it is invalid (there is no pedigree). Must specify a $G \geq 2$ if n is a scalar. If both G is specified and $\text{length}(n) > 1$, both values must agree.
sex	The numeric sex values for the founders (1L for male, 2L for female). By default they are drawn randomly using <code>draw_sex()</code> .
kinship_local	The local kinship matrix of the founder population. The default value is half the identity matrix, which corresponds to locally unrelated and locally outbred founders. This "local" kinship is the basis for all kinship calculations used to decide on close relative avoidance. The goal is to make a decision to not pair close relatives based on the pedigree only (and not based on population structure, which otherwise increases all kinship values), so the default value is appropriate.
cutoff	Local kinship values strictly less than cutoff are required for pairs. The default value of $1/4^3$ corresponds to second cousins, so those and closer relatives are forbidden pairs (but a third cousin pair is allowed).

children_min	The minimum number of children per family. Must be 0 or larger, but not exceed the average number of children per family in each generation (varies depending on how many individuals were left unpaired, but this upper limit is approximately $2 * n[i] / n[i-1]$ for generation i). The number of children for each given family is first chosen as children_min plus a Poisson random variable with parameter equal to the mean number of children per family needed to achieve the desired population size (n) minus children_min. As these numbers may not exactly equal the target population size, random families are incremented or decremented (respecting the minimum family size) by single counts until the target population size is met.
full	If TRUE, part of the return object is a list of local kinship matrices for every generation. If FALSE (default), only the local kinship matrix of the last generation is returned.

Value

A list with these named elements:

- fam: the pedigree, a tibble in plink FAM format. Following the column naming convention of the related genio package, it contains columns:
 - fam: Family ID, trivial "fam1" for all individuals
 - id: Individual ID, in this case a code of format (in regular expression) " $(\d+)-(\d+)$ " where the first integer is the generation number and the second integer is the index number (1 to $n[g]$ for generation g).
 - pat: Paternal ID. Matches an id except for founders, which have fathers set to NA.
 - mat: Maternal ID. Matches an id except for founders, which have mothers set to NA.
 - sex: integers 1L (male) or 2L (female) which were drawn randomly; no other values occur in these outputs.
 - pheno: Phenotype, here all 0 (missing value).
- ids: a list of IDs for each generation (indexed in the list by generation).
- kinship_local: if full = FALSE, the local kinship matrix of the last generation, otherwise a list of local kinship matrices for every generation.

See Also

Plink FAM format reference: <https://www.cog-genomics.org/plink/1.9/formats#fam>

Examples

```
# number of individuals for each generation
n <- c(15, 20, 25)

# create random pedigree with 3 generations, etc
data <- sim_pedigree( n )

# this is the FAM table defining the entire pedigree,
# which is the most important piece of information desired!
data$fam
```

```
# the IDs separated by generation
data$ids

# bonus: the local kinship matrix of the final generation
data$kinship_local
```

tidy_recomb_map_inds *Tidy recombination block data inherited in individuals from founders*

Description

This function takes a compact structure of nested lists and tables describing the founder blocks of identity by descent (IBD) inherited in a pedigree with recombination, and outputs the same data organized as a single table following tidy conventions, which is easy to manipulate externally although it is also larger and more redundant.

Usage

```
tidy_recomb_map_inds(inds)
```

Arguments

inds The output value from `recomb_map_inds`. In particular, a list of named individuals, each of which is a list with two parents labeled "pat" and "mat", each of which is a list of chromosomes identified by index, each of which is a table with one row per IBD block, and at least two columns, labeled "anc" is the label of the founder individual from which this block was derived, and "pos" which is the end coordinate in basepairs of the block (the start of the block is given implicitly by the "pos" of the previous row plus 1, or 1 for the first block).

Value

A table with one row per IBD block, and the following columns: "ind" containing the label of the individual as given in the input object, "parent" equal to either "pat" or "mat", "chr" the chromosome, "start" and "end" the range of the block in basepairs, and "anc" the label of the founder individual.

See Also

[recomb_map_inds\(\)](#)

Examples

```
# simulate the ancestors of one person to 3 generations
obj <- fam_ancestors( 3 )
fam <- obj$fam
ids <- obj$ids
# initialize founders
```

```
founders <- recomb_init_founders( ids[[ 1 ]], recomb_map_hg38 )
# draw recombination breaks along pedigree, with coordinates in genetic distance (centiMorgans),
# with information for last generation only
inds <- recomb_last_gen( founders, fam, ids )
# map recombination break coordinates to base pairs
inds <- recomb_map_inds( inds, recomb_map_hg38 )

# now that the input structure is ready, this function returns a tidy table version!
inds_tidy <- tidy_recomb_map_inds( inds )
```

Index

* datasets

recomb_map_hg, 32

admix_fam, 2
admix_fam(), 4
admix_last_gen, 4

bim_add_posg, 5

draw_sex, 7
draw_sex(), 36

fam_ancestors, 7

geno_fam, 8
geno_fam(), 10
geno_last_gen, 10
geno_last_gen_admix_recomb, 11

kinship2::kinship(), 14
kinship_fam, 14
kinship_fam(), 16
kinship_last_gen, 16

pop_recomb, 17
pop_recomb(), 11–13
prune_fam, 19

recomb_admix_inds, 20
recomb_admix_inds(), 13
recomb_fam, 22
recomb_fam(), 21, 26, 27, 29, 30, 33, 34
recomb_founder_blocks_inherited, 24
recomb_founder_blocks_inherited(), 13
recomb_geno_inds, 25
recomb_geno_inds(), 13, 20, 27
recomb_haplo_inds, 27
recomb_haplo_inds(), 11, 13, 20, 21, 25, 26
recomb_init_founders, 28
recomb_init_founders(), 13, 23, 30
recomb_last_gen, 30

recomb_last_gen(), 11, 13
recomb_map_fix_ends_chr, 31
recomb_map_fix_ends_chr(), 32, 35
recomb_map_hg, 5, 6, 29, 32, 33, 34
recomb_map_hg37(recomb_map_hg), 32
recomb_map_hg38(recomb_map_hg), 32
recomb_map_inds, 33
recomb_map_inds(), 13, 21, 26, 27, 38
recomb_map_simplify_chr, 34
recomb_map_simplify_chr(), 32

sim_pedigree, 36
sim_pedigree(), 4, 8, 10, 16, 30
stats::approx(), 6, 34

tidy_recomb_map_inds, 38
tidy_recomb_map_inds(), 13, 24