# Package 'simfam'

July 23, 2025

**Title** Simulate and Model Family Pedigrees with Structured Founders

**Version** 1.1.6

**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. Described in Yao and Ochoa (2022) <doi:10.1101/2022.03.25.485885>.

**License** GPL (>= 3)

**Encoding** UTF-8

**RoxygenNote** 7.2.3

**Imports** Rcpp, stats, tibble

**Suggests** testthat (>= 3.0.0), popkin, bnpsd (>= 1.3.2), 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

**NeedsCompilation** yes

**Author** Alejandro Ochoa [aut, cre] (ORCID: <https://orcid.org/0000-0003-4928-3403>)

**Maintainer** Alejandro Ochoa <alejandro.ochoa@duke.edu>

**Repository** CRAN

**Date/Publication** 2023-01-09 21:50:02 UTC

# Contents



---

`admix_fam` *Calculate admixture matrix of a pedigree with known admixture of founders*

---

## 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 *Calculate admixture matrix for last generation of a pedigree with admixture of founders*

## 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 fam$id). Individuals may be in a different order than fam$id. Extra individuals in admix but absent in fam$id will be silently ignored. All values should be non-negative and each row of admix should sum to one; for speed, this code does not check that admix is valid, just averages data as-is. |
| fam | The pedigree data.frame, in plink FAM format. Only columns id, pat, and mat are required. id must be unique and non-missing. Founders must be present, and their pat and mat values must be missing (see below). Non-founders must have both their parents be non-missing. Parents must appear earlier than their children in the table. |
| ids | A list containing vectors of IDs for each generation. All these IDs must be present in fam$id. If IDs in fam and ids do not fully agree, the code processes the IDs in the intersection, which is helpful when fam is pruned but ids 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
```

---

`draw_sex` *Draw sex values randomly for a list of individuals*

---

## 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` *Construct an ancestors-only pedigree for one person* `G`*-generations deep*

---

## 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` *Draw random genotypes for last generation of a pedigree with known founder genotypes*

---

## 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

| | |
|---|---|
| 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$id will be silently ignored. All values should be in c(0L, 1L, 2L); for speed, this code does not check that X is valid (i.e. fractional values between 0 and 2 may not cause errors). |
| fam | The pedigree data.frame, in plink FAM format. Only columns id, pat, and mat are required. id must be unique and non-missing. Founders must be present, and their pat and mat values must be missing (see below). Non-founders must have both their parents be non-missing. Parents must appear earlier than their children in the table. |
| ids | A list containing vectors of IDs for each generation. All these IDs must be present in fam$id. If IDs in fam and ids do not fully agree, the code processes the IDs in the intersection, which is helpful when fam is pruned but ids 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 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
```

---

`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). |
| `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 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` *Calculate kinship matrix for last generation of a pedigree with structured founders*

---

## 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 fam$id). Individuals may be in a different order than fam$id. Extra individuals in kinship but absent in fam$id will be silently ignored. A traditional pedigree calculation would use kinship = diag(n)/2 (plus appropriate column/row names), where n is the number of founders, to model unrelated and outbred founders. However, if kinship measures the population kinship estimates between founders, the output is also a population kinship matrix (which combines the structural/ancestral and local/pedigree relatedness values into one). |
| fam | The pedigree data.frame, in plink FAM format. Only columns id, pat, and mat are required. id must be unique and non-missing. Founders must be present, and their pat and mat values must be missing (see below). Non-founders must have both their parents be non-missing. Parents must appear earlier than their children in the table. |
| ids | A list containing vectors of IDs for each generation. All these IDs must be present in fam$id. If IDs in fam and ids do not fully agree, the code processes the IDs in the intersection, which is helpful when fam is pruned but ids 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
```

---

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` *Reduce haplotype ancestry data to population ancestry dosage matrices*

---

## 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_geno_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 `recomb_init_founders()`. Each element of this list is a diploid individual, which 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 (`posg` is end coordinates in cM; start is the end of the previous block, zero for the first block) and ancestors `anc` 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 `fam$id`). Individuals may be in a different order than `fam$id`. Extra individuals in `founders` but absent in `fam$id` will be silently ignored. |
| `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 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_geno_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_geno_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 recomb_haplo_inds() 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. |

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 )
```

---

`recomb_init_founders` *Initialize chromosome structures for founders*

---

## 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` *Draw recombination breaks for autosomes for last generation of a pedigree*

---

## 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

| | |
|---|---|
| `founders` | The named list of founders with their chromosomes. For unstructured founders, initialize with `recomb_init_founders()`. Each element of this list is a diploid individual, which 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 (`posg` is end coordinates in cM; start is the end of the previous block, zero for the first block) and ancestors `anc` 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 `fam$id`). Individuals may be in a different order than `fam$id`. Extra individuals in `founders` but absent in `fam$id` will be silently ignored. |
| `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 | A list containing vectors of IDs for each generation. All these IDs must be present in fam$id. If IDs in fam and ids do not fully agree, the code processes the IDs in the intersection, which is helpful when fam is pruned but ids 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 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 extrapolated with an error of up to `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 *Map recombination breaks from genetic positions to base pair coordinates*

### 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 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 (posg is end coordinates in cM; start is the end of the previous block, zero for the first block) and ancestors anc as strings. |
| 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

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 distance 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

| | |
|---|---|
| `map` | A tibble with two columns: `pos` position in base pairs, and `posg` position in centiMorgans (cM). |
| `tol` | 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 >= 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 >= 2` if `n` is a scalar. If both `G` is specified and `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 `draw_sex()`. |
| `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
```

# Index