Package 'pedtools'

February 1, 2026

Type Package

Title Creating and Working with Pedigrees and Marker Data

Version 2.10.0

Description A comprehensive collection of tools for creating, manipulating and visualising pedigrees and genetic marker data. Pedigrees can be read from text files or created on the fly with built-in functions. A range of utilities enable modifications like adding or removing individuals, breaking loops, and merging pedigrees. An online tool for creating pedigrees interactively, based on 'pedtools', is available at <https://magnusdv.shinyapps.io/quickped>. 'pedtools' is the hub of the 'pedsuite', a collection of packages for pedigree analysis. A detailed presentation of the 'pedsuite' is given in the book 'Pedigree Analysis in R' (Vigeland, 2021, ISBN:9780128244302).

License GPL-3

URL https://github.com/magnusdv/pedtools, https://magnusdv.github.io/pedsuite/

Depends R (>= 4.2)

Imports kinship2, pedmut

Suggests kableExtra, knitr, rmarkdown, testthat

VignetteBuilder knitr

Encoding UTF-8

Language en-GB

RoxygenNote 7.3.3

NeedsCompilation no

Author Magnus Dehli Vigeland [aut, cre] (ORCID: <https://orcid.org/0000-0002-9134-4962>)

Maintainer Magnus Dehli Vigeland <m.d.vigeland@medisin.uio.no>

Repository CRAN

Date/Publication 2026-02-01 13:50:02 UTC

Contents

`addAllele` *Add allele*

Description

Extends the allele set of a marker attached to a pedigree, by adding a single allele.

Usage

```
addAllele(x, marker, allele, freq = 0.001, adjust = c("previous", "all"))
```

Arguments

`x`	A ped object or a list of such, or a frequency database (list of numeric vectors).
`marker`	The name or index of a marker attached to `x`.
`allele`	The name of the new allele.
`freq`	The frequency of the new allele, by default 0.001.
`adjust`	Either "previous" or "all", indicating how the frequencies should be adjusted so that they sum to 1. If "previous" (default), the frequencies of the original alleles are multiplied with `1 - freq`. If "all", scaling is performed after adding the new allele, i.e., dividing all frequencies by `1 + freq`.

Value

A copy of `x` with modified marker attributes.

Examples

```
## Ped input
x = nuclearPed() |>
  addMarker(geno = c(NA, NA, "b/c"), afreq = c(b = 0.5, c = 0.5))

y = addAllele(x, marker = 1, allele = "a")
afreq(y, 1)

z = addAllele(y, marker = 1, allele = "d", freq = 0.1, adjust = "all")
afreq(z, 1)


## Database input
db = list(M1 = c(a = .2, b = .3, c = .5),
          M2 = c("7" = .9, "8.3" = .1))
addAllele(db, marker = "M2", allele = "8")
```

`as.data.frame.ped` *Convert ped to data.frame*

Description

Convert a ped object to a data.frame. The first columns are id, fid, mid and sex, followed by genotype columns for all (or a selection of) markers.

Usage

```
## S3 method for class 'ped'
as.data.frame(x, ..., markers, sep = "/", missing = "-")
```

Arguments

`x`	Object of class ped.
`...`	Further parameters
`markers`	Vector of marker names or indices. By default, all markers are included.
`sep`	A single string to be used as allele separator in marker genotypes.
`missing`	A single string to be used for missing alleles.

Details

Note that the output of `as.data.frame.ped()` is quite different from that of `as.matrix.ped()`. This reflects the fact that these functions have different purposes.

Conversion to a data frame is primarily intended for pretty printing. It uses correct labels for pedigree members and marker alleles, and pastes alleles to form nice-looking genotypes.

The matrix method, on the other hand, is a handy tool for manipulating the pedigree structure. It produces a numeric matrix, using the internal index labelling both for individuals and alleles,

making it very fast. In addition, all necessary meta information (loop breakers, allele frequencies a.s.o) is kept as attributes, which makes it possible to recreate the original ped object.

Value

A `data.frame` with `pedsize(x)` rows and `4 + nMarkers(x)` columns.

See Also

`as.matrix.ped()`

`as.matrix.ped` *Convert* ped *to matrix*

Description

Converts a ped object to a numeric matrix using internal labels, with additional info necessary to recreate the original ped attached as attributes.

Usage

```
## S3 method for class 'ped'
as.matrix(x, include.attrs = TRUE, ...)

restorePed(x, attrs = NULL, validate = TRUE)
```

Arguments

`x`	a ped object. In `restorePed`: A numerical matrix.
`include.attrs`	a logical indicating if marker annotations and other info should be attached as attributes. See Value.
`...`	not used.
`attrs`	a list containing labels and other ped info compatible with `x`, in the format produced by `as.matrix`. If NULL, the attributes of `x` itself are used.
`validate`	a logical, forwarded to `ped()`. If FALSE, no checks for pedigree errors are performed.

Details

`restorePed` is the reverse of `as.matrix.ped`.

Value

For `as.matrix`: A numerical matrix with `pedsize(x)` rows. If `include.attrs = TRUE` the following attributes are added to the matrix, allowing `x` to be exactly reproduced by `restorePed`:

- `FAMID` the family identifier (a string)
- `LABELS` the ID labels (a character vector)
- `UNBROKEN_LOOPS` a logical indicating whether `x` has unbroken loops
- `LOOP_BREAKERS` a numerical matrix, or NULL
- `markerattr` a list of length `nMarkers(x)`, containing the attributes of each marker

For `restorePed`: A ped object.

Author(s)

Magnus Dehli Vigeland

See Also

`ped()`

Examples

```
x = relabel(nuclearPed(1), letters[1:3])

# To examplify the ped -> matrix -> ped trick, we show how to
# reverse the internal ordering of the pedigree.
m = as.matrix(x, include.attrs = TRUE)
m[] = m[3:1, ]

# Must reverse the labels also:
attrs = attributes(m)
attrs$LABELS = rev(attrs$LABELS)

# Restore ped:
y = restorePed(m, attrs = attrs)

# Of course a simpler way is use reorderPed():
z = reorderPed(x, 3:1)
stopifnot(identical(y, z))
```

---

as.ped *Conversions to ped objects*

Description

Conversions to ped objects

Usage

```
as.ped(x, ...)

## S3 method for class 'data.frame'
as.ped(
  x,
  famid_col = NA,
  id_col = NA,
  fid_col = NA,
  mid_col = NA,
  sex_col = NA,
  marker_col = NA,
  locusAttributes = NULL,
  missing = 0,
  sep = NULL,
  sexCodes = NULL,
  addMissingFounders = FALSE,
  validate = TRUE,
  verbose = TRUE,
  ...
)
```

Arguments

x	Any object.
...	Not used.
famid_col	Index of family ID column. If NA, the program looks for a column named "famid" (ignoring case).
id_col	Index of individual ID column. If NA, the program looks for a column named "id" (ignoring case).
fid_col	Index of father ID column. If NA, the program looks for a column named "fid" (ignoring case).
mid_col	Index of mother ID column. If NA, the program looks for a column named "mid" (ignoring case).
sex_col	Index of column with gender codes (0 = unknown; 1 = male; 2 = female). If NA, the program looks for a column named "sex" (ignoring case). If this is not found, genders of parents are deduced from the data, leaving the remaining as unknown.
marker_col	Index vector indicating columns with marker alleles. If NA, all columns to the right of all pedigree columns are used. If sep (see below) is non-NULL, each column is interpreted as a genotype column and split into separate alleles with `strsplit(..., split = sep, fixed = TRUE)`.
locusAttributes	Passed on to setMarkers() (see explanation there).
missing	Passed on to setMarkers() (see explanation there).

sep	Passed on to setMarkers() (see explanation there).
sexCodes	A list with optional entries "male", "female" and "unknown", indicating how non-default entries in the sex column should be interpreted. Default values: male = 1, female = 2, unknown = 0.
addMissingFounders	A logical. If TRUE, any parent not included in the id column is added as a founder of corresponding sex. By default, missing founders result in an error.
validate	A logical indicating if the pedigree structure should be validated.
verbose	A logical.

Value

A ped object or a list of such.

Examples

```
df = data.frame(famid = c("S1", "S2"),
                id = c("A", "B"),
                fid = 0,
                mid = 0,
                sex = 1)

# gives a list of two singletons
as.ped(df)

# Trio
df1 = data.frame(id = 1:3, fid = c(0,0,1), mid = c(0,0,2), sex = c(1,2,1))
as.ped(df1)

# Disconnected example: Trio (1-3) + singleton (4)
df2 = data.frame(id = 1:4, fid = c(2,0,0,0), mid = c(3,0,0,0),
                 M = c("1/2", "1/1", "2/2", "3/4"))
as.ped(df2)

# Two singletons
df3 = data.frame(id = 1:2, fid = 0, mid = 0, sex = 1)
as.ped(df3)

# Add missing parents as founders
df4 = data.frame(id = 1, fid = 2, mid = 3, sex = 1)
as.ped(df4, addMissingFounders = TRUE)
```

---

as_kinship2_pedigree *Convert pedigree to kinship2 format*

Description

Convert pedigree to kinship2 format

Usage

```
as_kinship2_pedigree(
  x,
  deceased = NULL,
  aff = NULL,
  twins = NULL,
  hints = NULL
)
```

Arguments

x	A ped() object.
deceased	A vector of labels indicating deceased pedigree members.
aff	A vector of labels identifying members whose plot symbols should be filled. (This is typically used in medical pedigrees to indicate affected members.)
twins	A data frame with columns id1, id2 and code, passed on to the relation parameter of kinship2::plot.pedigree().
hints	An optional list of hints passed on to kinship2::align.pedigree().

Examples

```
x = nuclearPed()
as_kinship2_pedigree(x)
```

---

connectedComponents *Connected pedigree components*

Description

Compute the connected parts of a pedigree. This is an important step when converting pedigree data from other formats (where disconnected pedigrees may be allowed) to pedtools (which requires pedigrees to be connected).

Usage

```
connectedComponents(id, fid = NULL, mid = NULL, fidx = NULL, midx = NULL)
```

Arguments

id	A vector of ID labels (character or numeric).
fid	The ID labels of the fathers (or "0" if missing).
mid	The ID labels of the mothers (or "0" if missing).
fidx, midx	(For internal use mostly.) Integer vectors with paternal (resp. maternal) indices. These may be given instead of id, fid, mid.

Value

A list, where each element is a subset of `id` constituting a connected pedigree.

Examples

```
# A trio (1-3) and a singleton (4)
x = data.frame(id = 1:4, fid = c(2,0,0,0), mid = c(3,0,0,0))
connectedComponents(x$id, x$fid, x$mid)
```

---

distributeMarkers *Distribute markers evenly along a set of chromosomes*

Description

Create and attach identical (empty) marker objects, distributed along a set of chromosomes.

Usage

```
distributeMarkers(
  x,
  n = NULL,
  dist = NULL,
  chromLen = NULL,
  alleles = 1:2,
  afreq = NULL,
  prefix = "M"
)
```

Arguments

`x`	A ped object.
`n`	The total number of markers. Either this or `dist` must be NULL.
`dist`	A positive number; the distance (in megabases) between markers.
`chromLen`	A numeric vector indicating chromosome lengths (in Mb). By default, the lengths of the human chromosomes 1-22 are used, as returned by `sapply(ibdsim2::loadMap("decode"` `ibdsim2::physRange)`.
`alleles, afreq`	Passed onto `marker()`.
`prefix`	A string used as prefix for marker names. Default: "M".

Details

Note: When using the `dist` parameter, the function treats each chromosome separately, places one marker at the start and then every `dist` megabases. (See Examples.)

Value

A copy of x with the indicated markers attached.

Examples

```
x = distributeMarkers(nuclearPed(), n = 10)
getMap(x)

y = distributeMarkers(nuclearPed(), dist = 100)
getMap(y)
```

`expectedHomozygosity` *Expected homozygosity and heterozygosity*

Description

Computes the expected homozygosity and heterozygosity for a marker from its allele frequencies. The homozygosity is $\sum_i p_i^2$ and the heterozygosity is one minus this.

Usage

```
expectedHomozygosity(p)

expectedHeterozygosity(p)
```

Arguments

p	A numeric vector of allele frequencies, or a `marker` object (from which the frequencies are extracted with `afreq()`).

Value

A numeric value giving the expected homozygosity or heterozygosity.

Examples

```
p = c(0.2, 0.5, 0.3)
expectedHomozygosity(p)
expectedHeterozygosity(p)
```

`extractSingletons` *Extract singletons from pedigree*

Description

Extract one or more individuals from a pedigree, returning a list of singletons. Marker data and founder inbreeding (if present) are preserved.

Usage

```
extractSingletons(x, ids = NULL, simplify1 = TRUE, keepFI = TRUE)
```

Arguments

`x`	A ped object or a list of such.
`ids`	A vector of ID labels (coercible to character). If empty, all individuals are extracted.
`simplify1`	A logical indicating if the output should be simplified to a singleton object (i.e., removing the outer list structure) if `ids` has length 1.
`keepFI`	A logical indicating if founder inbreeding should be preserved, if present.

Value

A list of singletons. If `length(ids) == 1` and `simplify1 = TRUE`, a single `singleton` object is returned instead.

Examples

```
x = nuclearPed() |> addMarker(geno = c("1/1", NA, "1/2"))

# Extract father and child
extractSingletons(x, ids = c(1,3))

# Extract all members
extractSingletons(x)
```

famid *Family identifier*

Description

Functions for getting or setting the family ID of a ped object.

Usage

```
famid(x, ...)

## S3 method for class 'ped'
famid(x, ...)

famid(x, ...) <- value

## S3 replacement method for class 'ped'
famid(x, ...) <- value
```

Arguments

x	A ped object
...	(Not used)
value	The new family ID, which must be (coercible to) a character string.

Examples

```
x = nuclearPed(1)
famid(x) # empty string

famid(x) = "trio"
famid(x)
```

founderInbreeding *Inbreeding coefficients of founders*

Description

Functions to get or set inbreeding coefficients for the pedigree founders.

Usage

```
founderInbreeding(x, ids, named = FALSE, chromType = "autosomal")

founderInbreeding(x, ids, chromType = "autosomal") <- value

setFounderInbreeding(x, ids = NULL, value, chromType = "autosomal")
```

Arguments

x	A ped object or a list of such.
ids	Any subset of `founders(x)`. If `ids` is missing in `founderInbreeding()`, it is set to `founders(x)`.
named	A logical: If TRUE, the output vector is named with the ID labels.
chromType	Either "autosomal" (default) or "x".
value	A numeric of the same length as `ids`, entries in the interval `[0, 1]`. If the vector is named, then the names are interpreted as ID labels of the founders whose inbreeding coefficients should be set. In this case, the `ids` argument should not be used. (See examples.)

Value

For `founderInbreeding`, a numeric vector of the same length as `ids`, containing the founder inbreeding coefficients.

For `setFounderInbreeding()`, a copy of `x` with modified founder inbreeding.
`founderInbreeding<-` is an in-place version of `setFounderInbreeding()`.

Examples

```
x = nuclearPed(father = "fa", mother = "mo", child = 1)
founderInbreeding(x, "fa") = 1
founderInbreeding(x, named = TRUE)

# Setting all founders at once (replacement value is recycled)
founderInbreeding(x, ids = founders(x)) = 0.5
founderInbreeding(x, named = TRUE)

# Alternative syntax, using a named vector
founderInbreeding(x) = c(fa = 0.1, mo = 0.2)
founderInbreeding(x, named = TRUE)
```

---

freqDatabase *Allele frequency database*

Description

Functions for reading, setting and extracting allele frequency databases, in either "list" format, "merlin" format or "allelic ladder" format.

Usage

```
getFreqDatabase(x, markers = NULL, format = c("list", "ladder"))

setFreqDatabase(x, database, format = c("list", "ladder"), ...)

readFreqDatabase(
  filename = NULL,
  df = NULL,
  format = c("list", "ladder", "merlin"),
  fixNames = FALSE,
  scale1 = FALSE,
  verbose = TRUE,
  ...
)

writeFreqDatabase(x, filename, markers = NULL, format = c("list", "ladder"))
```

Arguments

x	A ped object, or a list of such.
markers	A character vector (with marker names) or a numeric vector (with marker indices).
format	Either "list", "ladder" or "merlin" (only in `readFreqDatabase()`).
database	Either a list or matrix/data frame with allele frequencies, or a file path (to be passed on to `readFreqDatabase()`).
...	Optional arguments passed on to `read.table()`, e.g. `sep = "\t"` if the file is tab separated.
filename	The path to a text file containing allele frequencies either in "list" or "allelic ladder" format.
df	A data frame of allele frequencies in either "list" or "allelic ladder" format. This can be supplied instead of `filename`.
fixNames	A logical, by default FALSE. If TRUE all marker names are converted to upper case, and all periods and space characters are replaced with "_" (underscore).
scale1	A logical, by default FALSE. If TRUE, all frequency vectors are scaled to ensure that it sums to 1.
verbose	A logical.

Details

A frequency database in "list" format is a list of numeric vectors; each vector named by its allele labels, and the list itself named by its marker names.

Text files containing frequencies in "list" format should look as follows, where "M1" and "M2" are marker names, and "a1","a2",... are allele labels (which may be characters or numeric, but will always be converted to characters):

```
M1
a1 0.2
a2 0.5
a3 0.3

M2
a1 0.9
a2 0.1
```

In "merlin" format, used by the software MERLIN (Abecasis et al., 2002), the same frequency data would be presented as follows:

```
M M1
A a1 0.2
A a2 0.5
A a3 0.3
M M2
A a1 0.9
A a2 0.1
```

A database in "allelic ladder" format is rectangular, i.e., a numeric matrix (or data frame), with allele labels as row names and markers as column names. NA entries correspond to unobserved alleles.

Value

- getFreqDatabase: either a list (if format = "list") or a data frame (if format = "ladder").
- readFreqDatabase: a list of named numeric vectors.
- setFreqDatabase: a modified version of x.

See Also

setLocusAttributes(), setMarkers(), setAlleles().

Examples

```
loc1 = list(name = "m1", afreq = c(a = .1, b = .9))
loc2 = list(name = "m2", afreq = c("1" = .2, "10.2" = .3, "3" = .5))
x = setMarkers(singleton(1), locus = list(loc1, loc2))
db = getFreqDatabase(x)
db

y = setFreqDatabase(x, database = db)
stopifnot(identical(x, y))

# The database can also be read directly from file
tmp = tempfile()
write("m1\na 0.1\nb 0.9\n\nm2\n1 0.2\n3 0.5\n10.2 0.3", tmp)

z = setFreqDatabase(x, database = tmp)
stopifnot(all.equal(x, z))
```

getAlleles *Allele matrix manipulation*

Description

Functions for getting and setting the genotypes of multiple individuals/markers simultaneously

Usage

```
getAlleles(x, ids = NULL, markers = NULL)

setAlleles(x, ids = NULL, markers = NULL, alleles)

removeGenotypes(x, ids = NULL, markers = NULL)
```

Arguments

x	A ped object or a list of such
ids	A vector of ID labels. If NULL (default) all individuals are included.
markers	A vector of indices or names of markers attaches to x. If NULL (default) all markers are included.
alleles	A character of the same format and dimensions as the output of getAlleles(x, ids, markers), or an object which can be converted by as.matrix() into such a matrix. See Details.

Details

If the alleles argument of setAlleles() is not a matrix, it is recycled (if necessary), and converted into a matrix of the correct dimensions.

removeGenotypes() is a convenience function for removing the genotypes of specified individuals and markers. It is equivalent to setAlleles(..., alleles = 0). In particular, removeGenotypes(x) removes all genotypes from the pedigree x, but leaves all locus attributes intact.

Value

getAlleles() returns a character matrix with length(ids) rows and 2 * length(markers) columns. The ID labels of x are used as rownames, while the columns are named <m1>.1, <m1>.2, ... where <m1> is the name of the first marker, a.s.o.

setAlleles() returns a ped object identical to x, except for the modified alleles. In particular, all locus attributes are unchanged.

See Also

transferMarkers()

Examples

```
# Setup: Pedigree with two markers
x = nuclearPed(1)
x = addMarker(x, `2` = "1/2", alleles = 1:2, name = "m1")
x = addMarker(x, `3` = "2/2", alleles = 1:2, name = "m2")

# Extract allele matrix
mat1 = getAlleles(x)
mat2 = getAlleles(x, ids = 2:3, markers = "m2")
stopifnot(identical(mat1[2:3, 3:4], mat2))

# Remove all genotypes
y = removeGenotypes(x)
y

# Setting a single genotype
z = setAlleles(y, ids = "1", marker = "m2", alleles = 1:2)

# Alternative: In-place modification with `genotype()`
genotype(y, id = "1", marker = "m2") = "1/2"
stopifnot(identical(y,z))


### Manipulation of pedlist objects
s = transferMarkers(x, singleton("s"))
peds = list(x, s)

getAlleles(peds)

setAlleles(peds, ids = "s", marker = "m1", alleles = 1:2)
```

`getComponent` *Pedigree component*

Description

Given a list of ped objects (called pedigree components), and a vector of ID labels, find the index of the component holding each individual.

Usage

```
getComponent(x, ids, checkUnique = FALSE, errorIfUnknown = FALSE)
```

Arguments

`x`	A ped object, or a list of such.
`ids`	A vector of ID labels (coercible to character).

checkUnique	A logical, by default FALSE. If TRUE, an error is raised if any element of ids occurs more than once in x.
errorIfUnknown	A logical, by default FALSE. If TRUE, the function stops with an error if not all elements of ids are found in x.

Value

An integer vector of the same length as ids, with NA entries where the corresponding label was not found in any of the components.

See Also

internalID()

Examples

```
x = list(nuclearPed(1), singleton(id = "A"))
getComponent(x, c(3, "A"))
```

---

getGenotypes *Genotype matrix*

Description

Extract the genotypes of specified individuals and markers from a pedigree object, and return them as a character matrix.

Usage

```
getGenotypes(
  x,
  ids = NULL,
  markers = NULL,
  sep = "/",
  missing = "-",
  Xchrom = NULL
)
```

Arguments

x	A ped object or a list of such.
ids	A vector of ID labels, or a function operating on x, e.g., typedMembers(). By default (ids = NULL) all individuals are included, also non-genotyped ones.
markers	A vector of indices or names of markers attaches to x. If NULL (default) all markers are included.
sep	A single string to be used as allele separator in marker genotypes.

missing	A single string to be used for missing alleles.
Xchrom	A single logical, or NULL (default). May be used to indicate if all (or none) markers are on X, if this is known in advance.

Value

getGenotypes() returns a character matrix with length(ids) rows and length(markers) columns.

See Also

getAlleles()

Examples

```
x = nuclearPed() |>
  addMarker(`2` = "1/2", name = "m1") |>
  addMarker(`3` = "a/a", name = "m2")

getGenotypes(x)

### A list of pedigrees

s = transferMarkers(x, singleton("s"))
peds = list(x, s)

getGenotypes(peds)

# Using a function to select individuals
getGenotypes(x, ids = typedMembers)
```

---

getMap *Tabulate marker positions*

Description

Return a map of the markers attached to a pedigree.

Usage

```
getMap(x, markers = NULL, na.action = 0, merlin = FALSE, verbose = TRUE)

setMap(x, map, matchNames = NA, ...)

hasLinkedMarkers(x)
```

Arguments

x	An object of class ped or a list of such.
markers	A vector of names or indices referring to markers attached to x. By default, all markers are included.
na.action	Either 0 (default), 1 or 2. (See Details.)
merlin	A logical mostly for internal use: If TRUE the function returns a matrix instead of a data frame.
verbose	A logical.
map	Either a data frame, the path to a map file, or NULL (for removing map info). See Details regarding format.
matchNames	A logical; if TRUE, pre-existing marker names of x will be used to assign chromosome labels and positions from map.
...	Further arguments passed to read.table().

Details

The na.action argument controls how missing values are dealt with:

- na.action = 0: Return map unmodified
- na.action = 1: Replace missing values with dummy values.
- na.action = 2: Remove markers with missing data.

In setMap(), the map argument should be a data frame (or file) with the following columns in order:

1. chromosome
2. marker name
3. position (Mb)

Column names are ignored, as are any columns after the first three.

Value

getMap() returns a data frame with columns CHROM, MARKER and MB.

setMap() returns x with modified marker attributes.

hasLinkedMarkers() returns TRUE if two markers are located (with set position) on the same chromosome, and FALSE otherwise.

Examples

```
x = singleton(1) |>
  addMarker(chrom = 1, posMb = 10, name = "m1") |>
  addMarker(chrom = 1, posMb = 11) |>
  addMarker(chrom = 1)

# Compare effect of `na.action`

getMap(x, na.action = 0)
```

```
getMap(x, na.action = 1)
getMap(x, na.action = 2)


# Getting and setting map are inverses
y = setMap(x, getMap(x))
stopifnot(identical(x,y))

hasLinkedMarkers(x)
```

---

getSex *Get or set the sex of pedigree members*

Description

Functions for retrieving or changing the sex of specified pedigree members. When used in pedigree constructions, `swapSex()` is usually more convenient than `setSex()`, since it deals with spouses automatically.

Usage

```
getSex(x, ids = NULL, named = FALSE)

setSex(x, ids = NULL, sex)

swapSex(x, ids, verbose = TRUE)
```

Arguments

x	A ped object or a list of such.
ids	A vector identifying members of x, or a function, in which case it is replaced with ids(x) labels. If NULL, defaults to all members of x.
named	A logical: return a named vector or not.
sex	A numeric vector with entries 1 (= male), 2 (= female) or 0 (= unknown). If ids is NULL, sex must be named with ID labels. If sex is unnamed and shorter than ids it is recycled to length(ids).
verbose	A logical: Verbose output or not.

Details

To set unknown sex, use `setSex(x, ids, sex = 0)`. Note that if a nonfounder has unknown sex the pedigree cannot be plotted in the usual way, only with `plot(x, arrows = TRUE)`.

Value

- `getSex(x, ids)` returns an integer vector of the same length as `ids`, with entries 0 (unknown), 1 (male) or 2 (female).
- `setSex(x, ids, sex)` returns a ped object similar to `x`, but where the sex of `ids` is set according to the entries of `sex`
- `swapSex(x, ids)` returns a ped object identical to `x`, but where the sex of `ids` (and their spouses) are swapped (1 <-> 2). Individuals of unknown sex are ignored.

See Also

`ped()`

Examples

```
x = nuclearPed(father = "fa", mother = "mo", children = "ch")

stopifnot(all.equal(
  getSex(x, named = TRUE),
  c(fa = 1, mo = 2, ch = 1)
))

# Make child female
setSex(x, ids = "ch", sex = 2)

# Same, using a named vector
setSex(x, sex = c(ch = 2))

# Same, using a function (setting all leaves to be female)
setSex(x, ids = leaves, sex = 2)

# swapSex() deals with spouses automatically
swapSex(x, ids = "fa")

# setting/getting sex in a pedlist
y = singletons(id = 1:3, sex = c(2,1,1))
sx = getSex(y, named = TRUE)
y2 = setSex(y, sex = sx)

stopifnot(identical(y, y2))
```

---

harmoniseMarkers *Harmonise markers across components in a ped list*

Description

Ensures all components of a ped list contain the same markers in identical order. Missing markers are added with empty genotypes. Markers whose attributes differ between components are updated to match the first occurrence of the marker. Note that this function removes all unnamed markers, unless the input is returned unchanged (see Details).

Usage

```
harmoniseMarkers(x, verbose = TRUE)
```

Arguments

x	A list of ped objects.
verbose	A logical.

Details

If the input is a single connected pedigree, it is returned as is.

If all marker attributes are identical across all components, x is also returned unchanged.

Value

A copy of x where all components have the same markers attached, and in the same order. Unnamed markers are removed (unless x is returned unchanged, see Details).

Examples

```
x = list(
  singleton(1) |> addMarker(),  # unnamed marker will be removed
  singleton(2) |> addMarker(name = "M1", alleles = 1:2),
  singleton(3) |> addMarker(name = "M1", alleles = 1:3), # will be modified
  singleton(4) |> addMarker(geno = "3/3", alleles = 1:3, name = "M2")
)
harmoniseMarkers(x)
```

---

inbreedingLoops *Pedigree loops*

Description

Functions for identifying, breaking and restoring loops in pedigrees.

Usage

```
inbreedingLoops(x)

breakLoops(x, loopBreakers = NULL, verbose = TRUE, errorIfFail = TRUE)

tieLoops(x, verbose = TRUE)

findLoopBreakers(x)

findLoopBreakers2(x, errorIfFail = TRUE)
```

Arguments

x	a ped() object.
loopBreakers	either NULL (resulting in automatic selection of loop breakers) or a vector indicating the individuals to be used as loop breakers.
verbose	a logical: Verbose output or not?
errorIfFail	a logical: If TRUE an error is raised if the loop breaking is unsuccessful. If FALSE, the pedigree is returned unchanged.

Details

Pedigree loops are usually handled (by pedtools and related packages) under the hood – using the functions described here – without the need for explicit action from end users. When a ped object x is created, an internal routine detects if the pedigree contains loops, in which case x$UNBROKEN_LOOPS is set to TRUE.

In cases with complex inbreeding, it can be instructive to plot the pedigree after breaking the loops. Duplicated individuals are plotted with appropriate labels (see examples).

The function breakLoops breaks the loops of the input pedigree by duplicating the loop breakers. These may be given by the user; otherwise they are selected automatically. In the current implementation, only nonfounders can act as loop breakers. For automatic selection of loop breakers, breakLoops first calls findLoopBreakers, which identifies and breaks all *inbreeding loops*. If the resulting pedigree still has loops, findLoopBreakers2 is called to handle *marriage loops*. In earlier versions this required the igraph package, but now uses a custom implementation using a depth-first search algorithm to find a cycle in the marriage node graph.

Value

For breakLoops, a ped object in which the indicated loop breakers are duplicated. The returned object will also have a non-null LOOP_BREAKERS entry, namely a matrix with the IDs of the original loop breakers in the first column and the duplicates in the second. If loop breaking fails, then depending on errorIfFail either an error is raised, or the input pedigree (with loops intact) is returned.

For tieLoops, a ped object in which any duplicated individuals (as given in the x$LOOP_BREAKERS entry) are merged. For any ped object x, the call tieLoops(breakLoops(x)) should return x.

For inbreedingLoops, a list containing all inbreeding loops (not marriage loops) found in the pedigree. Each loop is represented as a list with elements top, bottom, pathA (individuals forming a path from top to bottom) and pathB (creating a different path from top to bottom, with no individuals in common with pathA). Note that the number of loops reported here counts all closed paths in the pedigree and will in general be larger than the genus of the underlying graph.

For findLoopBreakers and findLoopBreakers2, a vector of individual labels.

Examples

```
x = cousinPed(1, child = TRUE)
plot(breakLoops(x))

# Pedigree with marriage loop: Double first cousins
```

```
y = doubleCousins(1, 1, child = TRUE)
findLoopBreakers(y) # --> 9
findLoopBreakers2(y) # --> 5 and 9
y2 = breakLoops(y)
plot(y2)

# Or loop breakers chosen by user
y3 = breakLoops(y, 6:7)
plot(y3)
```

---

is.marker *Test if something is a marker*

Description

Functions for testing if something is a `marker` object, or a list of such objects.

Usage

```
is.marker(x)

is.markerList(x)
```

Arguments

x	Any object

Value

A logical

---

is.ped *Is an object a* ped *object?*

Description

Functions for checking whether an object is a `ped()` object, a `singleton()` or a list of such.

Usage

```
is.ped(x)

is.singleton(x)

is.pedList(x)
```

Arguments

x	Any R object.

Details

Note that the singleton class inherits from ped, so if x is a singleton, is.ped(x) returns TRUE.

Value

For is.ped(): TRUE if x is a ped or singleton object, otherwise FALSE.

For is.singleton(): TRUE if x is a singleton object, otherwise FALSE.

For is.pedList(): TRUE if x is a list of ped and/or singleton objects, otherwise FALSE.

Author(s)

Magnus Dehli Vigeland

See Also

ped()

Examples

```
x1 = nuclearPed(1)
x2 = singleton(1)
stopifnot(is.ped(x1), !is.singleton(x1),
          is.ped(x2), is.singleton(x2),
          is.pedList(list(x1,x2)))
```

---

isHomozygous *Find homozygous genotypes*

Description

Identifies homozygous genotypes in the marker data. A genotype is homozygous if both alleles are non-missing and equal.

Usage

```
isHomozygous(x, ids = typedMembers(x), count = FALSE)
```

Arguments

x	A ped object or a list of such. An error is raised if x has no marker data.
ids	A vector of individual ID labels. Defaults to all typed members of x.
count	A logical. If TRUE, return counts per individual.

Value

By default, a logical matrix of dimension `N x L`, where N is `length(ids)` and L is the number of markers. If `count = TRUE`, a numeric vector of length N giving the number of homozygous genotypes for each individual.

Examples

```
x = nuclearPed(father = "fa", mother = "mo", children = "ch") |>
  addMarker(name = "m1", geno = c(NA, "1/1", "1/2")) |>
  addMarker(name = "m2", geno = c(NA, "2/2", "2/2"))

isHomozygous(x)
isHomozygous(x, ids = "mo")
isHomozygous(x, count = TRUE)
```

---

locusAttributes *Get or set locus attributes*

Description

Retrieve or modify the attributes of attached markers

Usage

```
getLocusAttributes(
  x,
  markers = NULL,
  attribs = c("alleles", "afreq", "name", "chrom", "posMb", "mutmod"),
  checkComps = FALSE,
  simplify = FALSE
)

setLocusAttributes(
  x,
  markers = NULL,
  locusAttributes,
  matchNames = NA,
  erase = FALSE
)
```

Arguments

x	A ped object, or a list of such.
markers	A character vector (with marker names) or a numeric vector (with marker indices). If NULL (default), the behaviour depends on `matchNames`, see Details.

attribs	A subset of the character vector c("alleles", "afreq", "name" ,"chrom" ,"posMb", "mutmod", "rate").
checkComps	A logical. If TRUE, and x is a list of pedigrees, an error is raised if marker attributes differ between components.
simplify	A logical. If TRUE, and attribs is a single element, the output is flattened to a simple list.
locusAttributes	A list of lists, with attributes for each marker.
matchNames	A logical, only relevant if markers = NULL. If TRUE, then the markers to be modified are identified by the 'name' component of each locusAttributes entry. If FALSE, all markers attached to x are selected in order.
erase	A logical. If TRUE, all previous attributes of the selected markers are erased. If FALSE, attributes not affected by the submitted locusAttributes remain untouched.

Details

The default setting markers = NULL selects markers automatically, depending on the matchNames argument. If matchNames = FALSE, all markers are chosen. If matchNames = TRUE, markers will be matched against the name entries in locusAttributes (and an error issued if any are missing).

Note that the default value NA of matchNames is changed to TRUE if all entries of locusAttributes have a name component which matches the name of an attached marker.

Possible attributes given in locusAttributes are as follows (default values in parentheses):

- alleles: a character vector with allele labels
- afreq: a numeric vector with allele frequencies (rep.int(1/L, L), where L = length(alleles))
- name: marker name (NA)
- chrom: chromosome number (NA)
- posMb: physical location in megabases (NA)
- mutmod: mutation model, or model name (NULL)
- rate: mutation model parameter (NULL)

Value

- getLocusAttributes: a list of lists. If the markers have names, these are used to name the outer list. If simplify = TRUE and attribs is a single element, the output is a simple list.
- setLocusAttributes: a modified version of x.

Examples

```
x = singleton(1) |>
  addMarker(name = "m1", alleles = 1:2) |>
  addMarker(name = "m2", alleles = letters[1:2], chrom = "X")

# By default, the markers to be modified are identified by name
locs = list(list(name = "m1", alleles = 1:10),
```

```
            list(name = "m2", alleles = letters[1:10]))
y = setLocusAttributes(x, locusAttributes = locs)
getMarkers(y, 1:2)

# If `erase = TRUE` attributes not explicitly given are erased
y2 = setLocusAttributes(x, locusAttributes = locs, erase = TRUE)
chrom(y2, 2) # not "X" anymore

# The getter and setter are inverses
newx = setLocusAttributes(x, locusAttributes = getLocusAttributes(x))
stopifnot(identical(x, newx))
```

`marker` *Marker objects*

Description

Creating a marker object associated with a pedigree. The function `marker()` returns a marker object, while `addMarker()` first creates the marker and then attaches it to `x`.

Usage

```
marker(
  x,
  ...,
  geno = NULL,
  allelematrix = NULL,
  alleles = NULL,
  afreq = NULL,
  chrom = NA,
  posMb = NA,
  name = NA,
  mutmod = NULL,
  rate = NULL,
  NAstrings = c(0, "", NA, "-"),
  validate = TRUE,
  validateMut = validate
)

addMarker(
  x,
  ...,
  geno = NULL,
  allelematrix = NULL,
  alleles = NULL,
  afreq = NULL,
  chrom = NA,
```

```
  posMb = NA,
  name = NA,
  mutmod = NULL,
  rate = NULL,
  locusAttr = NULL,
  NAstrings = c(0, "", NA, "-"),
  validate = TRUE
)
```

Arguments

x	A ped object.
...	One or more expressions of the form id = genotype, where id is the ID label of a member of x, and genotype is a numeric or character vector of length 1 or 2 (see Examples).
geno	A character vector of length pedsize(x), with genotypes written in the format "a/b".
allelematrix	A matrix with 2 columns and pedsize(x) rows. If this is non-NULL, then ... must be empty.
alleles	A vector of allele names. If not given, and afreq is named, names(afreq) is used. Otherwise, the default action is to use all distinct alleles occurring in allelematrix, geno or ....
afreq	A numeric of the same length as alleles, indicating the population frequency of each allele. A warning is issued if the frequencies don't sum to 1 after rounding to 3 decimals. If the vector is named, and alleles is not NULL, an error is raised if setequal(names(afreq), alleles) is not TRUE. If afreq is not specified, all alleles are given equal frequencies.
chrom	A single integer: the chromosome number. Default: NA.
posMb	A nonnegative real number: the physical position of the marker, in megabases. Default: NA.
name	A character string: the name of the marker. Default: NA.
mutmod, rate	Mutation model parameters to be passed on to pedmut::mutationModel(); see there for details. Note: mutmod corresponds to the model parameter. Default: NULL (no mutation model).
NAstrings	A character vector containing strings to be treated as missing alleles. Default: c("", "0", NA, "-").
validate	A logical indicating if the validity of the marker object should be checked. Default: TRUE.
validateMut	A logical indicating if the mutation model (if present) should be checked. Default: TRUE
locusAttr	A list with names alleles, afreq, chrom, name, posMb, mutmod, rate (or a subset of these). This can be used as an alternative to entering the arguments as function parameters.

Value

An object of class `marker`. This is an integer matrix with 2 columns and one row per individual, and the following attributes:

- `alleles` (a character vector with allele labels)
- `afreq` (allele frequencies; defaults to equal frequencies)
- `chrom` (chromosome number; default = NA)
- `posMb` (physical location in megabases; default = NA)
- `name` (marker identifier; default = NA)
- `mutmod` (a list of two (male/female) mutation matrices; default = NULL)

See Also

Get/set marker attributes: marker_getattr, marker_setattr.

Retrieve various marker properties: marker_prop, `nMarkers()`,

Add alleles to an existing marker: `addAllele()`

Attach multiple markers: marker_attach

Examples

```
x = nuclearPed(father = "fa", mother = "mo", children = "child")

# An empty SNP with alleles "A" and "B"
marker(x, alleles = c("A", "B"))

# Creating and attaching to `x`
addMarker(x, alleles = c("A", "B"))

# Alleles/frequencies can be given jointly or separately
stopifnot(identical(
  marker(x, afreq = c(A = 0.01, B = 0.99)),
  marker(x, alleles = c("A", "B"), afreq = c(0.01, 0.99)),
  ))

# Genotypes can be assigned individually ...
addMarker(x, fa = "1/1", mo = "1/2")

# ... or using the `geno` vector (all members in order)
addMarker(x, geno = c("1/1", "1/2", NA))

# A marker with an "equal" mutation model
y = addMarker(x, alleles = 1:2, name = "M", mutmod = "equal", rate = 0.01)
mutmod(y, "M")
```

marker_attach *Attach markers to pedigrees*

Description

In many applications it is useful to *attach* markers to their associated ped object. In particular for bigger projects with many markers, this makes it easier to manipulate the dataset as a unit. The function setMarkers() replaces all existing markers with the supplied ones, while addMarkers() appends the supplied markers to any existing ones. Note that there is also the function addMarker(), which creates and attaches a single marker in one go.

Usage

```
setMarkers(
  x,
  m = NULL,
  alleleMatrix = NULL,
  locusAttributes = NULL,
  missing = 0,
  sep = NULL,
  checkCons = TRUE
)

addMarkers(
  x,
  m = NULL,
  alleleMatrix = NULL,
  locusAttributes = NULL,
  missing = 0,
  sep = NULL,
  checkCons = TRUE
)
```

Arguments

x	A ped object
m	Either a single marker object or a list of marker objects
alleleMatrix	A matrix with pedsize(x) rows, containing the observed alleles for one or several markers. The matrix must have either 1 or 2 columns per marker. If the former, then a sep string must be given, and will be used to split all entries.
locusAttributes	A list of lists, with attributes for each marker. See Details for possible attributes.
missing	A single character (or coercible to one) indicating the symbol for missing alleles.
sep	If this is a single string, each entry of alleleMatrix is interpreted as a genotype, and will be split by calling strsplit(..., split = sep, fixed = TRUE).

If `alleleMatrix` contains entries with "/", this will be taken as separator by default. (To override this behaviour, put `sep = FALSE`.)

`checkCons` A logical. If TRUE (default), each marker is checked for consistency with `x`.

Details

The most general format of `locusAttributes` is a list of lists, one for each marker, where possible entries in the inner lists are as follows (default values in parenthesis):

- `alleles`: a character vector with allele labels
- `afreq`: a numeric vector with allele frequencies (equal)
- `chrom`: chromosome number (NA)
- `posMb`: physical location in megabases (NA)
- `name`: marker name (NA)
- `mutmod`: mutation model, or model name (NULL)
- `rate`: mutation model parameter (NULL)

If `locusAttributes` is a single list of attributes (not a list of lists), then it is repeated to match the number of markers.

Alternative formats of `locusAttributes`::

- A data frame or matrix. In this case an attempt is made to interpret it as a frequency database in `allelic ladder` format.
- A list of frequency vectors. All vectors should sum to 1, and be named (with allele labels)
- Shortcut for simple SNP data: The argument `locusAttributes = "snp-AB"` sets all markers to be equifrequent SNPs with alleles A and B. The letters A and B may be replaced by other single-character letters or numbers.

Value

A ped object.

See Also

`addMarker()`

Examples

```
x = singleton(1)
m1 = marker(x, `1` = "1/2")
m2 = marker(x, `1` = "a/b")

# Attach to x
x1 = setMarkers(x, list(m1, m2))

# Reversing the order of the markers
setMarkers(x, list(m2, m1))

# Alternative syntax, adding one marker at a time
```

```
x2 = x |>
  addMarker(`1` = "1/2") |>
  addMarker(`1` = "a/b")

stopifnot(identical(x1, x2))
```

---

marker_getattr *Get marker attributes*

Description

S3 methods retrieving marker attributes. They work on single marker objects and markers attached to ped objects (or lists of such).

Usage

```
genotype(x, ...)

## S3 method for class 'marker'
genotype(x, id, ...)

## S3 method for class 'ped'
genotype(x, markers = NULL, id, ...)

mutmod(x, ...)

## S3 method for class 'marker'
mutmod(x, ...)

## S3 method for class 'ped'
mutmod(x, marker, ...)

## S3 method for class 'list'
mutmod(x, marker, ...)

alleles(x, ...)

## S3 method for class 'marker'
alleles(x, ...)

## S3 method for class 'ped'
alleles(x, marker = NULL, simplify1 = TRUE, ...)

## S3 method for class 'list'
alleles(x, marker = NULL, simplify1 = TRUE, ...)
```

```
afreq(x, ...)

## S3 method for class 'marker'
afreq(x, ...)

## S3 method for class 'ped'
afreq(x, marker, ...)

## S3 method for class 'list'
afreq(x, marker, ...)

name(x, ...)

## S3 method for class 'marker'
name(x, ...)

## S3 method for class 'ped'
name(x, markers = NULL, ...)

## S3 method for class 'list'
name(x, markers = NULL, ...)

chrom(x, ...)

## S3 method for class 'marker'
chrom(x, ...)

## S3 method for class 'ped'
chrom(x, markers = NULL, ...)

## S3 method for class 'list'
chrom(x, markers = NULL, ...)

posMb(x, ...)

## S3 method for class 'marker'
posMb(x, ...)

## S3 method for class 'ped'
posMb(x, markers = NULL, ...)
```

Arguments

x	Either a marker object, a ped object or a list of ped objects.
...	Further arguments, not used.
id	The ID label of a single pedigree member.
marker, markers	The index or name of a marker (or a vector indicating several markers) attached to x.

simplify1 A logical (default: TRUE) indicating if the output should be simplified to an unnamed vector if the input is a single marker.

Value

The associated marker attributes.

See Also

Setting marker attributes: marker_setattr and marker_inplace.

Examples

```
x = nuclearPed(1)
x = addMarker(x) # add empty marker

# Inspect default attributes
alleles(x, marker = 1)
afreq(x, marker = 1)
name(x, marker = 1)  # NA
chrom(x, marker = 1) # NA
```

---

marker_inplace *Set marker attributes*

Description

These S3 methods perform in-place modifications of marker attributes. They work on single marker objects and markers attached to ped objects (or lists of such). Although these functions will continue to exist, we recommend the newer alternatives `setGenotype()`, `setAfreq()`, ... in most cases.

Usage

```
genotype(x, ...) <- value

## S3 replacement method for class 'marker'
genotype(x, id, ...) <- value

## S3 replacement method for class 'ped'
genotype(x, marker, id, ...) <- value

mutmod(x, ...) <- value

## S3 replacement method for class 'marker'
mutmod(x, ...) <- value

## S3 replacement method for class 'ped'
```

```
mutmod(x, marker = NULL, ...) <- value

## S3 replacement method for class 'list'
mutmod(x, marker = NULL, ...) <- value

afreq(x, ...) <- value

## S3 replacement method for class 'marker'
afreq(x, ...) <- value

## S3 replacement method for class 'ped'
afreq(x, marker, ...) <- value

## S3 replacement method for class 'list'
afreq(x, marker, ...) <- value

name(x, ...) <- value

## S3 replacement method for class 'marker'
name(x, ...) <- value

## S3 replacement method for class 'ped'
name(x, markers = NULL, ...) <- value

## S3 replacement method for class 'list'
name(x, markers = NULL, ...) <- value

chrom(x, ...) <- value

## S3 replacement method for class 'marker'
chrom(x, ...) <- value

## S3 replacement method for class 'ped'
chrom(x, markers = NULL, ...) <- value

## S3 replacement method for class 'list'
chrom(x, markers = NULL, ...) <- value

posMb(x, ...) <- value

## S3 replacement method for class 'marker'
posMb(x, ...) <- value

## S3 replacement method for class 'ped'
posMb(x, markers = NULL, ...) <- value
```

Arguments

x	Either a marker object, a ped object or a list of ped objects.

...	Further arguments, not used.
value	Replacement value(s).
id	The ID label of a single pedigree member.
marker, markers	The index or name of a marker (or a vector indicating several markers) attached to ped. Used if x is a ped object.

Value

These functions perform in-place modification of x.

See Also

Alternative setters (not in-place): marker_setattr. Marker attribute getters: marker_getattr.

Examples

```
x = nuclearPed(1)
x = addMarker(x, alleles = 1:2)

# Set genotypes
genotype(x, marker = 1, id = 1) = "1/2"

# Set marker name
name(x, 1) = "M"

# Change allele freqs
afreq(x, "M") = c(`1` = 0.1, `2` = 0.9)

# Set position
chrom(x, "M") = 1
posMb(x, "M") = 123.45

# Check result
m = marker(x, `1` = "1/2", name = "M", afreq = c(`1` = 0.1, `2` = 0.9),
           chrom = 1, posMb = 123.45)
stopifnot(identical(x$MARKERS[[1]], m))
```

---

marker_prop *Marker properties*

Description

These functions are used to retrieve various properties of marker objects. Each function accepts as input either a single marker object, a ped object, or a list of ped objects.

Usage

```
emptyMarker(x, ...)

## Default S3 method:
emptyMarker(x, ...)

## S3 method for class 'marker'
emptyMarker(x, ...)

## S3 method for class 'ped'
emptyMarker(x, markers = NULL, ...)

## S3 method for class 'list'
emptyMarker(x, markers = NULL, ...)

nTyped(x, ...)

## Default S3 method:
nTyped(x, ...)

## S3 method for class 'marker'
nTyped(x, ...)

## S3 method for class 'ped'
nTyped(x, markers = NULL, ...)

## S3 method for class 'list'
nTyped(x, markers = NULL, ...)

nAlleles(x, ...)

## Default S3 method:
nAlleles(x, ...)

## S3 method for class 'marker'
nAlleles(x, ...)

## S3 method for class 'ped'
nAlleles(x, markers = NULL, ...)

## S3 method for class 'list'
nAlleles(x, markers = NULL, ...)

isXmarker(x, ...)

## Default S3 method:
isXmarker(x, ...)
```

```
## S3 method for class 'marker'
isXmarker(x, ...)

## S3 method for class 'ped'
isXmarker(x, markers = NULL, ...)

## S3 method for class 'list'
isXmarker(x, markers = NULL, ...)

allowsMutations(x, ...)

## Default S3 method:
allowsMutations(x, ...)

## S3 method for class 'marker'
allowsMutations(x, ...)

## S3 method for class 'ped'
allowsMutations(x, markers = NULL, ...)

## S3 method for class 'list'
allowsMutations(x, markers = NULL, ...)
```

Arguments

x	A single marker object or a ped object (or a list of such)
...	Not used.
markers	A vector of names or indices of markers attached to x. By default all attached markers are selected.

Details

emptyMarker() returns TRUE for markers with no genotypes. If the input is a list of pedigrees, all must be empty for the result to be TRUE.

nTyped() returns the number of typed individuals for each marker. Note that if the input is a list of pedigrees, the function returns the sum over all components.

nAlleles() returns the number of alleles of each marker.

isXmarker() returns TRUE for markers whose chrom attribute is either "X" or 23.

allowsMutations returns TRUE for markers whose mutmod attribute is non-NULL and differs from the identity matrix.

Value

If x is a single marker object, the output is a vector of length 1.

Otherwise, a vector of length nMarkers(x) (default) or length(markers), reporting the property of each marker.

Examples

```
cmp1 = nuclearPed(1)
cmp2 = singleton(10)
loc = list(alleles = 1:2)
x = setMarkers(list(cmp1, cmp2), locus = rep(list(loc), 3))

#-------- nAlleles() ------------
# All markers have 2 alleles
stopifnot(identical(nAlleles(x), c(2L,2L,2L)))

#-------- emptyMarkers() ------------
# Add genotype for indiv 1 at marker 1
genotype(x[[1]], 1, 1) = "1/2"

# Check that markers 2 and 3 are empty
stopifnot(identical(emptyMarker(x), c(FALSE,TRUE,TRUE)),
          identical(emptyMarker(x[[1]]), c(FALSE,TRUE,TRUE)),
          identical(emptyMarker(x[[2]]), c(TRUE,TRUE,TRUE)),
          identical(emptyMarker(x, markers = c(3,1)), c(TRUE,FALSE)))

#-------- nTyped() ------------
stopifnot(identical(nTyped(x), c(1L,0L,0L)))

# Add genotypes for third marker
genotype(x[[1]], marker = 3, id = 1:3) = "1/1"
genotype(x[[2]], marker = 3, id = 10) = "2/2"

# nTyped() returns total over all components
stopifnot(identical(nTyped(x), c(1L,0L,4L)))

#-------- allowsMutations() ------------
# Marker 2 allows mutations
mutmod(x, 2) = list("prop", rate = 0.1)

stopifnot(identical(allowsMutations(x), c(FALSE,TRUE,FALSE)),
          identical(allowsMutations(x, markers = 2:3), c(TRUE,FALSE)))

#-------- isXmarker() ------------
# Make marker 3 X-linked
chrom(x[[1]], 3) = "X"
chrom(x[[2]], 3) = "X"

stopifnot(identical(isXmarker(x), c(FALSE,FALSE,TRUE)))
```

`marker_select` *Select or remove attached markers*

Description

Functions for manipulating markers attached to ped objects.

Usage

```
selectMarkers(x, markers = NULL, chroms = NULL, fromPos = NULL, toPos = NULL)

getMarkers(x, markers = NULL, chroms = NULL, fromPos = NULL, toPos = NULL)

removeMarkers(x, markers = NULL, chroms = NULL, fromPos = NULL, toPos = NULL)

whichMarkers(x, markers = NULL, chroms = NULL, fromPos = NULL, toPos = NULL)
```

Arguments

x	A ped object, or a list of such
markers	Either a character vector (with marker names), a numeric vector (with marker indices), a logical (of length nMarkers(x)), or NULL.
chroms	A vector of chromosome names, or NULL
fromPos	A single number or NULL
toPos	A single number or NULL

Details

If markers consists of negative integers, it will be converted to its complement within 1:nMarkers(x).

Value

The return values of these functions are:

- selectMarkers(): an object identical to x, but where only the indicated markers are kept
- removeMarkers(): an object identical to x, but where the indicated markers are removed
- getMarkers(): a list of marker objects. Note: If x is a list of pedigrees, the marker objects attached to the first component will be returned.
- whichMarkers(): an integer vector with indices of the indicated markers. If x is a list of pedigrees an error is raised unless whichMarkers() gives the same result for all components.

See Also

setMarkers()

marker_setattr *Set marker attributes*

Description

These functions set or modify various attributes of markers attached to a pedigree. They are sometimes more convenient (and pipe-friendly) than the in-place modifiers described in marker_inplace.

Usage

```
setGenotype(x, marker = NULL, ids = NULL, geno = NULL, id = NULL)

setAfreq(x, marker, afreq, strict = TRUE)

setAlleleLabels(x, marker, alleles)

setMarkername(x, marker = NULL, name)

setChrom(x, marker = NULL, chrom)

setPosition(x, marker = NULL, posMb)
```

Arguments

x	A ped object or a list of ped objects.
marker	A vector of indices or names of one or several markers attached to x.
geno	A character vector of length pedsize(x), with genotypes written in the format "a/b".
id, ids	A vector naming one or several pedigree members, or a function (e.g., founders()).
afreq	A numeric of the same length as alleles, indicating the population frequency of each allele. A warning is issued if the frequencies don't sum to 1 after rounding to 3 decimals. If the vector is named, and alleles is not NULL, an error is raised if setequal(names(afreq), alleles) is not TRUE. If afreq is not specified, all alleles are given equal frequencies.
strict	A logical. If TRUE (default) the new frequencies cannot remove or add any alleles.
alleles	A vector of allele names. If not given, and afreq is named, names(afreq) is used. Otherwise, the default action is to use all distinct alleles occurring in allelematrix, geno or ....
name	A character of the same length as marker (recycled if length 1), with new marker names. Use NULL or NA to remove names.
chrom	A character of the same length as marker (recycled if length 1), with chromosome labels. Use NULL or NA to remove chromosome info.
posMb	A numeric of the same length as marker (recycled if length 1), containing physical marker positions in Mb. Use NULL or NA to remove position info.

Value

A copy of x with modified attributes.

Examples

```
x = nuclearPed() |>
  addMarker(alleles = 1:2) |>
  setMarkername(marker = 1, name = "M") |>
  setGenotype(marker = "M", ids = 1, geno = "1/2") |>
  setAfreq(marker = "M", afreq = c(`1` = 0.1, `2` = 0.9)) |>
  setChrom(marker = "M", chrom = 1) |>
  setPosition(marker = "M", posMb = 123.45)

# Alternatively, all of this could have been done on creation:
y = addMarker(nuclearPed(),
              `1` = "1/2",
              afreq = c(`1` = 0.1, `2` = 0.9),
              name = "M",
              chrom = 1,
              posMb = 123.45)
stopifnot(identical(x, y))
```

---

maskPed *Mask and unmask pedigree datasets*

Description

The `maskPed()` function replaces the individual IDs, marker names and allele names with generic labels, and randomly changes their internal order. For markers with stepwise mutation models, the allelic ladder is simply translated to start at 1, thereby preserving the intra-allelic differences.

Usage

```
maskPed(
  x,
  ids = NULL,
  markerNames = NULL,
  markerShuffle = TRUE,
  alleleLabels = NULL,
  alleleShuffle = TRUE,
  seed = NULL
)

unmaskPed(x, keys)
```

Arguments

x	A ped object or a list of such.
ids	(Optional) A named character with the new IDs, written as c(old = new, ...). By default: 1, 2, ...
markerNames	(Optional) A named character with the new marker names (and order), written as c(old = new, ...). By default: M1, M2, ...
markerShuffle	A logical: Randomly reorder the markers? (Default: TRUE)
alleleLabels	(Optional) A list of character vectors. The list names should be the original marker names. Each vector gives the new allele labels, as c(old = new, ...). By default, each marker gets alleles 1, 2, ...
alleleShuffle	A logical: Randomly reorder the alleles? (Default: TRUE)
seed	An optional seed for the random number generator.
keys	A list with entries ids, markerNames, alleleLabels.

Details

Note that in order to preserve likelihoods, the allele frequencies are not modified. Thus, if the data uses a publicly available frequency databases, the result cannot be considered to be fully anonymised, since one could (at least in theory) deduce the original marker names and alleles from the frequencies.)

Value

An object similar to x but with replaced ID labels, marker names and allele labels.

Examples

```
x = nuclearPed(father = "fa", mother = "mo", children = "ch") |>
  addMarker(name = "myMarker", ch = "b/c", afreq = c(a=0.2, b=0.3, c=0.5)) |>
  setMutmod(model = "proportional", rate = 0.01)

# Mask
y = maskPed(x, seed = 1729)

# Unmask
z = unmaskPed(y$maskedPed, keys = y$keys)
stopifnot(identical(x, z))

# With stepwise model
x2 = x |>
  addMarker(name = "mySTR", ch = "7.2/8.2",
            alleles = c("7", "7.2", "8", "8.2")) |>
  setMutmod(marker = 2, model = "stepwise", rate = 0.1, rate2 = 1e-6,
            range = 0.1)

y2 = maskPed(x2, seed = 1729)

z2 = unmaskPed(y2$maskedPed, keys = y2$keys)
```

```
stopifnot(identical(x2, z2))

# Check likelihoods with pedprobr:
# stopifnot(setequal(likelihood(x2), likelihood(y2$maskedPed)))
```

`mendelianCheck` *Check for Mendelian errors*

Description

Check marker data for Mendelian inconsistencies

Usage

```
mendelianCheck(x, remove = FALSE, verbose = !remove)
```

Arguments

`x`	a `ped()` object
`remove`	a logical. If FALSE, the function returns the indices of markers found to incorrect. If TRUE, a new ped object is returned, where the incorrect markers have been deleted.
`verbose`	a logical. If TRUE, details of the markers failing the tests are shown.

Value

A numeric containing the indices of the markers that did not pass all tests, or (if `remove = TRUE`) a new ped object where the failing markers are removed.

Author(s)

Magnus Dehli Vigeland

Examples

```
x = nuclearPed()

# Add a SNP with Mendelian error
m = marker(x, '1' = "1/1", '2' = "1/1", '3' = "1/2")
x = setMarkers(x, m)

mendelianCheck(x)
```

mergePed *Merge two pedigrees*

Description

This function merges two pedigrees, joining them at the indicated individuals.

Usage

```
mergePed(x, y, by = NULL, relabel = FALSE, ...)
```

Arguments

x, y	Two ped() objects.
by	The individuals to merge by. The most general form uses a named vector with entries of the form id.x = id.y (see Examples). If the vector is unnamed, it is assumed that the individuals have the same labels in both pedigrees. By default set to intersect(labels(x), labels(y)).
relabel	A logical, by default FALSE. If TRUE, relabel(..., "asPlot") is run on the merged pedigree before returning.
...	further arguments passed along to ped(), e.g. famid, validate and reorder.

Details

Some internal checks are done to ensure that merging individuals are compatible in terms of sex and parents.

If relabel = FALSE, some relabelling might still be performed in order to ensure unique labels for everyone. Specifically, this is the case if some ID labels occur in both x and y other than those given in the by argument. In such cases, the relevant members of y get a suffix .y.

Value

A ped object.

Author(s)

Magnus Dehli Vigeland

Examples

```
############
# Example 1: Merge 2 trios by fusing the fathers
x1 = x2 = nuclearPed()
x = mergePed(x1, x2, by = c("1" = "1"))
plot(x)
```

```
#################################
# Example 2: Double first cousins
#################################

# First cousins, whose fathers are brothers
y = cousinPed(degree = 1)

# Create two sisters
sisters = nuclearPed(2, sex = 2)

# Plot to see who is who: `plot(list(y, sisters))`

# Merge
z = mergePed(y, sisters, by = c("4" = 3, "6" = 4), relabel = TRUE)
plot(z)
```

`newMarker` *Internal marker constructor*

Description

This is the internal constructor of `marker` objects. It does not do any input validation and should only be used in programming scenarios, and only if you know what you are doing. Most users are recommended to use the regular constructor `marker()`.

Usage

```
newMarker(
  alleleMatrixInt,
  alleles,
  afreq,
  name = NA_character_,
  chrom = NA_character_,
  posMb = NA_real_,
  mutmod = NULL,
  pedmembers,
  sex
)
```

Arguments

`alleleMatrixInt`	An integer matrix.
`alleles`	A character vector.
`afreq`	A numeric vector.
`name`	A character of length 1.
`chrom`	A character of length 1.

posMb	A numeric of length 1.
mutmod	A mutation model.
pedmembers	A character vector.
sex	An integer vector.

Details

See marker() for more details about the marker attributes.

Value

A marker object.

Examples

```
newMarker(matrix(c(1L, 0L, 1L, 1L, 0L, 2L), ncol = 2),
          alleles = c("A", "B"), afreq = c(0.1, 0.9), name = "M",
          pedmembers = c("1", "2", "3"), sex = c(1L, 2L, 1L))
```

---

newPed *Internal ped constructor*

Description

This is the internal constructor of ped objects. It does not do any validation of input other than simple type checking. In particular it should only be used in programming scenarios where it is known that the input is a valid, connected pedigree. End users are recommended to use the regular constructor ped().

Usage

```
newPed(ID, FIDX, MIDX, SEX, FAMID, detectLoops = TRUE)
```

Arguments

ID	A character vector.
FIDX	An integer vector.
MIDX	An integer vector.
SEX	An integer vector.
FAMID	A string.
detectLoops	A logical.

Details

See ped() for details about the input parameters.

Value

A ped object.

Examples

```
newPed("a", 0L, 0L, 1L, "")
```

---

nMarkers *The number of markers attached to a pedigree*

Description

The number of markers attached to a pedigree

Usage

```
nMarkers(x, compwise = FALSE)

hasMarkers(x, compwise = FALSE)
```

Arguments

x	A ped object or a list of such.
compwise	A logical, only relevant if x is a ped list. Default FALSE.

Value

`nMarkers()` by default returns a single number; the number of marker objects attached to `x`. If `x` is a ped list, an error is raised if the components have different numbers of markers. This check can be skipped by setting `compwise = TRUE`, in which case the function returns a vector of the component-wise marker numbers.

The function `hasMarkers(x)` returns TRUE if (at least component of) `x` has attached markers, otherwise FALSE. If `compwise = TRUE`, a logical vector of the same length as `x`.

Examples

```
x = nuclearPed() |> addMarker()
nMarkers(x) # = 1

y = list(x, singleton(1))
nMarkers(y, compwise = TRUE) # c(1,0)

hasMarkers(y) # TRUE
hasMarkers(y, compwise = TRUE) # c(TRUE, FALSE)
```

ped *Pedigree construction*

Description

This is the basic constructor of ped objects. Utility functions for creating many common pedigree structures are described in ped_basic. See also `as.ped()` and `readPed()`, which are more liberal regarding the input format.

Usage

```
ped(
  id,
  fid,
  mid,
  sex,
  famid = "",
  reorder = TRUE,
  validate = TRUE,
  detectLoops = TRUE,
  isConnected = FALSE,
  verbose = FALSE
)

singleton(id = 1, sex = 1, famid = "")

singletons(id, sex = 1)
```

Arguments

`id`	A character (or coercible to character) of individual ID labels.
`fid, mid`	Vectors of the same length as `id`, naming each individual's father and mother. Missing parents (of founders) may be entered as "0", "" or NA.
`sex`	A numeric of the same length as `id`, describing the genders of the individuals (in the same order as `id`.) Each entry must be either 1 (=male), 2 (=female) or 0 (=unknown).
`famid`	A character string. Default: An empty string.
`reorder`	A logical indicating if the pedigree should be reordered so that all parents precede their children. Default: TRUE.
`validate`	A logical indicating if a validation of the pedigree structure should be performed. Default: TRUE.
`detectLoops`	A logical indicating if the presence of loops should be detected. Setting this to FALSE may speed up the processing of large pedigrees. Default: TRUE.
`isConnected`	A logical indicating if the input is known to be a connected pedigree. Setting this to TRUE speeds up the processing. Default: FALSE.
`verbose`	A logical.

Details

Each individual must have either both parents specified, or no parents. Missing parents are indicated with entries "0", "" or NA in `fid` and `mid`. Note that `id`,`fid`,`mid` are all converted to character vectors before matching to establish the parent connections.

If the pedigree is disconnected, it is split into its connected components and returned as a list of `ped` objects.

A singleton is a special `ped` object whose pedigree contains 1 individual. The class attribute of a singleton is `c('singleton', 'ped')`.

`singletons()` creates a list of singletons with the indicated labels and sexes.

Selfing, i.e. the presence of pedigree members whose father and mother are the same individual, is allowed in `ped` objects. Any such "self-fertilizing" parent must have undecided sex (`sex = 0`).

Value

A `ped` object, which is essentially a list with the following entries:

- `ID`: A character vector of ID labels. Unless the pedigree is reordered during creation, this equals `as.character(id)`
- `FIDX`: An integer vector with paternal indices: For each $j = 1, 2, ...$, `FIDX[j]` is 0 if `ID[j]` has no father; otherwise `ID[FIDX[j]]` is the father of `ID[j]`.
- `MIDX`: An integer vector with maternal indices: For each $j = 1, 2, ...$, `MIDX[j]` is 0 if `ID[j]` has no mother; otherwise `ID[MIDX[j]]` is the mother of `ID[j]`.
- `SEX`: An integer vector with gender codes. Unless the pedigree is reordered, this equals `as.integer(sex)`.
- `FAMID`: The family ID.
- `UNBROKEN_LOOPS`: A logical indicating if the pedigree has unbroken loops, or NA if the status is currently unknown.
- `LOOP_BREAKERS`: A matrix with loop breaker ID's in the first column and their duplicates in the second column. All entries refer to the internal IDs. This is usually set by `breakLoops()`.
- `FOUNDER_INBREEDING`: A list of two potential entries, "autosomal" and "x"; both numeric vectors with the same length as `founders(x)`. `FOUNDER_INBREEDING` is always NULL when a new `ped` is created. See `founderInbreeding()`.
- `MARKERS`: A list of `marker` objects, or NULL.

Author(s)

Magnus Dehli Vigeland

See Also

`newPed()`, ped_basic, ped_modify, ped_subgroups, `relabel()`

Examples

```
# Trio
x = ped(id = 1:3, fid = c(0,0,1), mid = c(0,0,2), sex = c(1,2,1))

# Female singleton
y = singleton('NN', sex = 2)

# Selfing
z = ped(id = 1:2, fid = 0:1, mid = 0:1, sex = 0:1)
stopifnot(hasSelfing(z))

# Disconnected pedigree: Trio + singleton
ped(id = 1:4, fid = c(2,0,0,0), mid = c(3,0,0,0), sex = c(1,1,2,1))

# List of singletons
singletons(1:2)
```

---

ped_basic *Create basic pedigrees*

Description

Utility functions for creating some common pedigree structures.

Usage

```
nuclearPed(nch = 1, sex = 1, father = "1", mother = "2", children = NULL)

halfSibPed(
  nch1 = 1,
  nch2 = 1,
  sex1 = 1,
  sex2 = 1,
  type = c("paternal", "maternal")
)

linearPed(n, sex = 1)

cousinPed(
  degree = 1,
  removal = 0,
  side = c("right", "left"),
  half = FALSE,
  symmetric = FALSE,
  child = FALSE
)
```

```
avuncularPed(
  top = c("uncle", "aunt"),
  bottom = c("nephew", "niece"),
  side = c("right", "left"),
  type = c("paternal", "maternal"),
  removal = 1,
  half = FALSE
)

halfCousinPed(
  degree = 1,
  removal = 0,
  side = c("right", "left"),
  symmetric = FALSE,
  child = FALSE
)

ancestralPed(g)

selfingPed(s, sex = 1)
```

Arguments

nch	The number of children, by default 1. If children is not NULL, nch is set to length(children).
sex	A vector with integer gender codes (0=unknown, 1=male, 2=female). In nuclearPed(), it contains the genders of the children and is recycled (if necessary) to length nch. In linearPed() it also contains the genders of the children (1 in each generation) and should have length at most n (recycled if shorter than this). In selfingPed() it should be a single number, indicating the gender of the last individual (the others must necessarily have gender code 0.)
father	The label of the father. Default: "1".
mother	The label of the mother. Default: "2".
children	A character with labels of the children. Default: c("3", "4", ...)
nch1, nch2	The number of children in each sibship.
sex1, sex2	Vectors of gender codes for the children in each sibship. Recycled (if necessary) to lengths nch1 and nch2 respectively.
type	Either "paternal" or "maternal".
n	The number of generations, not including the initial founders.
degree	A non-negative integer: 0=siblings, 1=first cousins; 2=second cousins, a.s.o.
removal	A non-negative integer. See Details and Examples.
side	Either "right" or "left"; the side on which removals should be added.
half	A logical indicating if the relationship should be "half-like". Default: FALSE.
symmetric	A logical, by default FALSE. If TRUE, the cousin pedigree uses female connections on the left side, giving a more symmetric appearance when plotted.

child	A logical: Should an inbred child be added to the two bottom individuals?
top, bottom	Strings specifying the avuncular relationships. The first must be either "uncle" or "aunt", while the second is "nephew" or "niece". Both can be abbreviated.
g	A nonnegative integer indicating the number of ancestral generations to include. The resulting pedigree has 2^(g+1)-1 members. The case g = 0 results in a singleton.
s	Either a character vector of ID labels, or a nonnegative integer indicating the number of consecutive selfings. The case s = 0 results in a singleton.

Details

halfSibPed(nch1, nch2) produces a pedigree containing two sibships (of sizes nch1 and nch2) with the same father, but different mothers. If maternal half sibs are wanted instead, add type = "maternal".

cousinPed(degree = n, removal = k) creates a pedigree with two n'th cousins, k times removed. By default, removals are added on the right side, but this can be changed by adding side = left.

halfCousinPed(...) is a synonym for cousinPed(..., half = TRUE).

avuncularPed() creates uncle/aunt - nephew/niece pedigrees. The empty call avuncularPed() is equivalent to avuncularPed("uncle", "nephew"). Note that the arguments can be abbreviated, so that e.g. avuncularPed("a", "ni") produces an aunt-niece relationship. Grand (and great-grand etc) uncles/aunts can be produced by specifying removal greater than 1.

ancestralPed(g) returns the family tree of a single individual, including all ancestors g generations back.

selfingPed(s) returns a line of s consecutive selfings.

Value

A ped object.

See Also

ped(), singleton(), ped_complex, ped_subgroups

Examples

```
# A nuclear family with 2 boys and 3 girls
nuclearPed(5, sex = c(1, 1, 2, 2, 2))

# A straight line of females
linearPed(3, sex = 2)

# Paternal half brothers
halfSibPed()

# Maternal half sisters
halfSibPed(sex1 = 2, sex2 = 2, type = "maternal")

# Larger half sibships: boy and girl on one side; 3 girls on the other
```

```
halfSibPed(nch1 = 2, sex = 1:2, nch2 = 3, sex2 = 2)

# Grand aunt:
cousinPed(degree = 0, removal = 2)

# Second cousins once removed.
cousinPed(degree = 2, removal = 1)

# Same, but with the 'removal' on the left side.
cousinPed(2, 1, side = "left")

# A child of half first cousins.
halfCousinPed(degree = 1, child = TRUE)

# The 'family tree' of a person
ancestralPed(g = 2)
```

---

ped_complex *Complex pedigree structures*

Description

Functions for creating a selection of pedigrees that are awkward to construct from scratch or with the simple structures described in ped_basic.

Usage

```
doubleCousins(
  degree1,
  degree2,
  removal1 = 0,
  removal2 = 0,
  half1 = FALSE,
  half2 = FALSE,
  child = FALSE
)

doubleFirstCousins()

quadHalfFirstCousins()

fullSibMating(n)

halfSibStack(n)

halfSibTriangle(g)
```

Arguments

degree1, degree2, removal1, removal2	Nonnegative integers.
half1, half2	Logicals, indicating if the fathers (resp. mothers) should be full or half cousins.
child	A logical: Should a child be added to the double cousins?
n	A positive integer indicating the number of crossings.
g	A positive integer; the number of generations.

Details

The function doubleCousins returns a pedigree linking two individuals who are simultaneous paternal and maternal cousins. More precisely, they are:

- paternal (full or half) cousins of type (degree1, removal1)
- maternal (full or half) cousins of type (degree2, removal2).

For convenience, a wrapper doubleFirstCousins is provided for the most common case, double first cousins.

quadHalfFirstCousins produces a pedigree with quadruple half first cousins.

fullSibMating crosses full sibs consecutively n times.

halfSibStack produces a breeding scheme where the two individuals in the final generation are simultaneous half k'th cousins, for each k = 0,...,n-1.

halfSibTriangle produces a triangular pedigree in which every pair of parents are half siblings.

Value

A ped object.

See Also

ped_basic

Examples

```
# Consecutive brother-sister matings.
x = fullSibMating(2)
# plot(x)

# Simultaneous half siblings and half first cousins
x = halfSibStack(2)
# plot(x)

# Double first cousins
x = doubleFirstCousins()
# plot(x)

# Quadruple half first cousins
x = quadHalfFirstCousins()
```

```
# plot(x) # Weird plotting behaviour for this pedigree.

# Triangular half-sib pattern
x = halfSibTriangle(4)
# plot(x)
```

---

ped_internal *Internal ordering of pedigree members*

Description

These functions give access to - and enable modifications of - the order in which the members of a pedigree are stored. (This is the order in which the members are listed when a ped object is printed to the screen.)

Usage

```
reorderPed(x, neworder = NULL, internal = FALSE)

parentsBeforeChildren(x)

hasParentsBeforeChildren(x)

foundersFirst(x)

internalID(x, ids, errorIfUnknown = TRUE)
```

Arguments

x	A ped object. Most of these functions also accepts ped lists.
neworder	A permutation of labels(x) (or a subset of this), indicating the new internal ordering. If internal = TRUE, neworder refers to the internal ordering, so must be numeric. By default, the natural order of the ID labels is used.
internal	A logical (default: FALSE). If TRUE, neworder is interpreted as referring to the internal ordering.
ids	A character vector (or coercible to one) of original ID labels.
errorIfUnknown	A logical. If TRUE (default), the function stops with an error if not all elements of ids are recognised as names of members in x.

Details

While the internal pedigree ordering rarely matters, it is occasionally important. The function reorderPed() permutes the internal ordering as specified by the user. The most common use of this function is perhaps in parentsBeforeChildren(), which ensures that all parents precede their children. This is required by many pedigree-traversing algorithms.

It should be noted that ped() by default calls parentsBeforeChildren() whenever a pedigree is created, unless explicitly avoided with reorder = FALSE.

hasParentsBeforeChildren() can be used as a quick test to decide if it is necessary to call parentsBeforeChildren().

The foundersFirst() function reorders the pedigree so that all the founders come first.

The utility internalID() converts ID labels to indices in the internal ordering. If x is a list of pedigrees, the output is a data frame containing both the component number and internal ID (within the component).

See Also

ped()

Examples

```
x = ped(id = 3:1, fid = c(1,0,0), mid = c(2,0,0), sex = c(1,2,1), reorder = FALSE)
x

# The 'ids' argument is converted to character, hence these are the same:
internalID(x, ids = 3)
internalID(x, ids = "3")

hasParentsBeforeChildren(x)

# Put parents first
parentsBeforeChildren(x)

# Typical use of reorderPed: Swap sibling plot order
y = nuclearPed(2) |> reorderPed(4:3)
plot(y)


### If labels are numeric, argument `internal` is important
z = singleton(1) |> addParents(1)
z
reorderPed(z, 1:3, internal = FALSE) # ID order = "1","2","3"
reorderPed(z, 1:3, internal = TRUE)  # index order: 1,2,3 (i.e., no change)
```

---

ped_modify *Add/remove pedigree members*

Description

Functions for adding or removing individuals in a 'ped' object.

Usage

```
addChildren(
  x,
  father = NULL,
  mother = NULL,
  nch = NULL,
  sex = 1L,
  ids = NULL,
  verbose = TRUE
)

addChild(x, parents, id = NULL, sex = 1, verbose = TRUE)

addSon(x, parents, id = NULL, verbose = TRUE)

addDaughter(x, parents, id = NULL, verbose = TRUE)

addParents(x, id, father = NULL, mother = NULL, verbose = TRUE)

addSibling(x, id, sex = 1, side = c("right", "left"), verbose = TRUE)

removeIndividuals(
  x,
  ids,
  remove = c("descendants", "ancestors"),
  returnLabs = FALSE,
  verbose = TRUE
)

trim(x, uninformative, verbose = TRUE)

branch(x, id)

## S3 method for class 'ped'
subset(x, subset, ..., missingParents = c("error", "exclude", "include"))
```

Arguments

x	A ped object, or a list of such.
father, mother	Single ID labels. At least one of these must be an existing member of x. The other may be (i) another existing member, (ii) a new founder to be created, or (iii) missing (i.e., NULL), in which case the other parent is created and given a suitable name.
nch	A positive integer indicating the number of children to be created. Default: 1.
sex	A vector of integers indicating the sex of the created individuals. (1 = male, 2 = female, 0 = unknown). Recycled as needed.

ids	A vector of ID labels. In addChildren() these are the children to be created. If NULL (default) given, automatic labels are generated.
verbose	A logical.
parents	A vector of 1 or 2 ID labels, of which at least one must be an existing member of x.
id	The ID label of a pedigree member.
side	A word indicating whether a new sibling should be placed to the left or right of the indicated individual. Default: "right".
remove	Either "ancestors" or "descendants" (default), dictating the method of removing pedigree members. Abbreviations are allowed.
returnLabs	A logical, by default FALSE. If TRUE, removeIndividuals() returns only the labels of all members to be removed, instead of actually removing them.
uninformative	A vector naming individuals considered "uninformative", or a function (typically a helper function like untypedMembers()). Uninformative leaves are removed iteratively until no more can be found.
subset	A character vector (or coercible to such): A subset of the ID labels of x.
...	Not used.
missingParents	A word indicating how to deal with single parents in the subset. Either "error" (default), "exclude" (remove the parent-child connection) or "include" (expand subset to include the missing parents).

Details

In addChildren() and addParents(), labels of added individuals are generated automatically if they are not specified by the user. The automatic labelling uses the smallest integers not already in use.

addChild(), addSon() and addDaughter() are convenient wrappers for the most common use of addChildren(), namely adding a single child to a pedigree. Note that the parents can be given in any order. If only one parent is supplied, the other is created as a new individual.

addSibling() adds a sibling to the indicated individual, creating parents if necessary. The new sibling is placed either directly before or after the indicated individual, depending on the side argument. (But note that the internal ordering of individuals is not always respected when plotting the pedigree.)

removeIndividuals() removes the individuals indicated with ids along with all of their ancestors OR descendants, depending on the remove argument. Leftover spouses disconnected from the remaining pedigree are also removed.

The branch() function extracts the sub-pedigree formed by id and all his/her descendants, and all necessary spouses. Note that some structure may be lost in this process; for instance, compare x = halfSibTriangle(3) with branch(x, 1).

The trim() function iteratively removes uninformative leaves (i.e., members without children) from the pedigree. Note that the definition of "uninformative" is entirely user-defined. For example, trim(x, untypedMembers), will remove untyped individuals from the bottom until the process stops.

Finally, `subset()` can be used to extract any sub-pedigree, returning a list of pedigrees if the result is disconnected. By default, an error is raised if some individuals would be left with exactly one parent. Alternatively, `missingParents = "exclude"` removes the parent-child connection in such cases, while `missingParents = "include"` adds the missing parent to the subset.

Value

The modified ped object.

See Also

`ped()`, `relabel()`, `swapSex()`

Examples

```
x = nuclearPed(1) |>
  addSon(3) |>
  addParents(4, father = 6, mother = 7) |>
  addChildren(father = 6, mother = 7, nch = 3, sex = c(2,1,2))

# Remove 6 and 7 and their descendants
y1 = removeIndividuals(x, 6:7)

# Remove 8-10 and their parents
y2 = removeIndividuals(x, 8:10, remove = "ancestors")

# Adding a child across components
z = singletons(1:2, sex = 1:2) |> addDaughter(1:2)

# Extract a branch
w = cousinPed(1, child = TRUE)
w4 = branch(w, 4)
plot(list(w, w4))

# General subsetting depends on `missingParent`:
subset(w, c(3,7), missingParents = "exclude")
subset(w, c(3,7), missingParents = "include")
```

---

ped_subgroups *Pedigree subgroups*

Description

A collection of utility functions for identifying pedigree members with certain properties.

Usage

```
founders(x, internal = FALSE)

nonfounders(x, internal = FALSE)

leaves(x, internal = FALSE)

males(x, internal = FALSE)

females(x, internal = FALSE)

typedMembers(x, internal = FALSE)

untypedMembers(x, internal = FALSE)

father(x, id, internal = FALSE)

mother(x, id, internal = FALSE)

children(x, id, internal = FALSE)

spouses(x, id, internal = FALSE)

unrelated(x, id, internal = FALSE)

parents(x, id, internal = FALSE)

grandparents(x, id, degree = 2, internal = FALSE)

siblings(x, id, half = NA, internal = FALSE)

nephews_nieces(x, id, removal = 1, half = NA, internal = FALSE)

niblings(x, id, half = NA, internal = FALSE)

piblings(x, id, half = NA, internal = FALSE)

ancestors(x, id, maxGen = Inf, inclusive = FALSE, internal = FALSE)

commonAncestors(x, ids, maxGen = Inf, inclusive = FALSE, internal = FALSE)

descendants(x, id, maxGen = Inf, inclusive = FALSE, internal = FALSE)

commonDescendants(x, ids, maxGen = Inf, inclusive = FALSE, internal = FALSE)

descentPaths(x, ids = founders(x), internal = FALSE)
```

Arguments

x	A ped() object or a list of such.
internal	A logical indicating whether id (or ids) refers to the internal order.
id, ids	A character (or coercible to character) of one or more ID labels. If internal is TRUE, id and ids should be positive integers.
degree, removal	Non-negative integers.
half	a logical or NA. If TRUE (resp. FALSE), only half (resp. full) siblings/cousins/nephews/nieces are returned. If NA, both categories are included.
maxGen	The number of generations to include. Default: Inf (no limit).
inclusive	A logical indicating whether an individual should be counted among his or her own ancestors/descendants

Value

The functions founders, nonfounders, males, females, leaves each return a vector containing the IDs of all pedigree members with the wanted property. (Recall that a founder is a member without parents in the pedigree, and that a leaf is a member without children in the pedigree.)

The functions father, mother, parents, children, siblings, grandparents, spouses, niblings (nephews + nieces), piblings (aunts

- uncles) and unrelated, each returns a vector naming all pedigree members with the specified relationship to id.

The commands ancestors(x, id) and descendants(x, id) return vectors containing the IDs of all ancestors (resp. descendants) of the individual id within the pedigree x. If inclusive = TRUE, id is included in the output, otherwise not. To cut off at a specific number of generations, use maxGen.

For commonAncestors(x, ids) and commonDescendants(x, ids), the output is a vector containing the IDs of common ancestors (descendants) to all of ids.

Finally, descentPaths(x, ids) returns a list of lists, containing all pedigree paths descending from each individual in ids (by default all founders).

Author(s)

Magnus Dehli Vigeland

Examples

```
x = ped(id = 2:9,
        fid = c(0,0,2,0,4,4,0,2),
        mid = c(0,0,3,0,5,5,0,8),
        sex = c(1,2,1,2,1,2,2,2))

spouses(x, id = 2) # 3, 8
children(x, 2)     # 4, 9
siblings(x, 4)     # 9 (full or half)
unrelated(x, 4)    # 5, 8
```

```
father(x, 4)       # 2
mother(x, 4)       # 3

siblings(x, 4, half = FALSE) # none
siblings(x, 4, half = TRUE)  # 9

niblings(x, 9) # 6, 7
niblings(x, 9, half = FALSE) # none

piblings(x, 6) # 9
piblings(x, 6, half = FALSE) # none

ancestors(x, 6)                              # 2, 3, 4, 5
ancestors(x, 6, maxGen = 2, inclusive = TRUE) # 4, 5, 6

descendants(x, 2)                              # 4, 6, 7, 9
descendants(x, 2, maxGen = 2, inclusive = TRUE)  # 2, 4, 9

leaves(x)          # 6, 7, 9
founders(x)        # 2, 3, 5, 8
```

---

ped_utils *Pedigree utilities*

Description

Various utility functions for ped objects.

Usage

```
pedsize(x)

generations(x, what = c("max", "compMax", "indiv", "depth"))

nChildren(x, ids = labels(x), named = FALSE)

hasUnbrokenLoops(x)

hasInbredFounders(x, chromType = "autosomal")

hasSelfing(x)

hasCommonAncestor(x)

subnucs(x)

peelingOrder(x)
```

Arguments

x	A ped object, or (in some functions - see Details) a list of such.
what	Either "max", "compMax", "indiv" or "depth" (See Value.)
ids	A vector of individual IDs.
named	A logical.
chromType	Either "autosomal" (default) or "x".

Value

- `pedsize(x)` returns the number of pedigree members in each component of `x`.
- `generations(x)` by default returns the number of generations in `x`, defined as the number of individuals in the longest line of parent-child links. (Note that this is well-defined also if `x` has loops and/or cross-generational marriages.) For individual generation numbers, use `what = "indiv"` (generation numbering as in the plot) or `what = "depth"` (length of the longest chain up to a founder) the function returns a vector with the generation count from each component.
- `nChildren(x, ids)` returns an integer vector containing the number of children of each indicated individual. It is equivalent to, but more efficient than, `lengths(lapply(ids, function(id) children(x, id)))`.
- `hasUnbrokenLoops(x)` returns TRUE if `x` has loops, otherwise FALSE. (No computation is done here; the function simply returns the value of `x$UNBROKEN_LOOPS`).
- `hasInbredFounders(x)` returns TRUE is founder inbreeding is specified for `x` and at least one founder has positive inbreeding coefficient. See `founderInbreeding()` for details.
- `hasSelfing(x)` returns TRUE if the pedigree contains selfing events. This is recognised by father and mother begin equal for some child. (Note that for this to be allowed, the gender code of the parent must be 0.)
- `hasCommonAncestor(x)` computes a logical matrix `A` whose entry `A[i,j]` is TRUE if pedigree members i and j have a common ancestor in `x`, and FALSE otherwise. By convention, `A[i,i]` is TRUE for all i.
- `subnucs(x)` returns a list of all nuclear sub-pedigrees of `x`, wrapped as `nucleus` objects. Each nucleus is a list with entries `father`, `mother` and `children`.
- `peelingOrder(x)` calls `subnucs(x)` and extends each entry with a `link` individual, indicating a member linking the nucleus to the remaining pedigree. One application of this function is the fact that if *fails* to find a complete peeling order if and only if the pedigree has loops. (In fact it is called each time a new ped object is created by `ped()` in order to detect loops.) The main purpose of the function, however, is to prepare for probability calculations in other packages, as e.g. in `pedprobr::likelihood`.

Examples

```
x = fullSibMating(1)
stopifnot(pedsize(x) == 6)
stopifnot(hasUnbrokenLoops(x))
stopifnot(generations(x) == 3)

# All members have common ancestors except the grandparents
```

```
CA = hasCommonAncestor(x)
stopifnot(!CA[1,2], !CA[2,1], sum(CA) == length(CA) - 2)

# Effect of breaking the loop
y = breakLoops(x)
stopifnot(!hasUnbrokenLoops(y))
stopifnot(pedsize(y) == 7)

# A pedigree with selfing (note the necessary `sex = 0`)
z1 = singleton(1, sex = 0)
z2 = addChildren(z1, father = 1, mother = 1, nch = 1)
stopifnot(!hasSelfing(z1), hasSelfing(z2))

# Nucleus sub-pedigrees
stopifnot(length(subnucs(z1)) == 0)
peelingOrder(cousinPed(1))

# Plot with generation numbers as labels
w = cousinPed(1)
g = generations(w, what = "indiv")
labs = setNames(labels(w), g)
plot(w, labs = labs)

# ... compare with
plot(relabel(w, "generations"))
```

`plot.ped` *Plot pedigree*

Description

This is the main function for plotting pedigrees. Many options are available for controlling the appearance of pedigree symbols and accompanying labels. The most important ones are illustrated in the Examples section below; for a complete overview, see the separate page plotmethods, which also explains the plotting procedure in more detail.

Usage

```
## S3 method for class 'ped'
plot(x, draw = TRUE, keep.par = FALSE, ...)

drawPed(
  alignment,
  annotation = NULL,
  scaling = NULL,
  keep.par = FALSE,
  draw = TRUE,
  ...
```

```
)

## S3 method for class 'pedList'
plot(x, ...)

## S3 method for class 'list'
plot(x, ...)
```

Arguments

x	A ped() object or a list of such.
draw	A logical, by default TRUE. If FALSE, no plot is produced, only the plotting parameters are returned.
keep.par	A logical, by default FALSE. If TRUE, the graphical parameters are not reset after plotting, which may be useful for adding additional annotation.
...	Arguments passed on to the internal plot functions. For a complete list of parameters, see plotmethods. The most important ones are illustrated in the Examples below.
alignment	List of alignment details, as returned by .pedAlignment().
annotation	List of annotation details as returned by .pedAnnotation().
scaling	List of scaling parameters as returned by .pedScaling().

Details

The main pedigree layout is calculated with the kinship2 package, see kinship2::align.pedigree for details. Unlike kinship2, the implementation here also supports singletons, and plotting pedigrees as DAGs. In addition, some minor adjustments have been made to improve scaling and avoid unneeded duplications.

If x is a list of ped objects, these are plotted next to each other, vertically centred in the plotting window. For finer control, and possibly nested lists of pedigrees, use plotPedList().

Value

A list of three lists with various plot details: alignment, annotation, scaling.

See Also

plotPedList(), kinship2::plot.pedigree(). Plot options are documented in plotmethods.

Examples

```
# Singleton
plot(singleton(1))

# Trio
x = nuclearPed(father = "fa", mother = "mo", child = "boy")
plot(x)
```

```
#' # Modify margins
plot(x, margins = 6)
plot(x, margins = c(0,0,6,6)) # b,l,t,r

# Larger text and symbols
plot(x, cex = 1.5)

# Enlarge symbols only
plot(x, symbolsize = 1.5)

# Various annotations
plot(x, hatched = "boy", starred = "fa", deceased = "mo", title = "Fam 1")

# Swap spouse order
plot(x, spouseOrder = c("mo", "fa"))

#----- ID labels -----

# Label only some members
plot(x, labs = c("fa", "mo"))

# Label males only
plot(x, labs = males)

# Rename some individuals
plot(x, labs = c(FATHER = "fa", "boy"))

# By default, long names are folded to width ~12 characters
plot(x, labs = c("Very long father's name" = "fa"), margin = 2)

# Folding width may be adjusted ...
plot(x, labs = c("Very long father's name" = "fa"), foldLabs = 6)

# ... or switched off (requires larger margin!)
plot(x, labs = c("Very long father's name" = "fa"), foldLabs = FALSE)

# By default, labels are trimmed for initial/trailing line breaks ...
plot(x, labs = c("\nFA" = "fa"))

# ... but this can be overridden
plot(x, labs = c("\nFA" = "fa"), trimLabs = FALSE)

#----- Colours -----

plot(x, col = c(fa = "red"), fill = c(mo = "green", boy = "blue"))

# Non-black hatch colours are specified with the `fill` argument
plot(x, hatched = labels, fill = c(boy = "red"))

# Use functions to specify colours
plot(x, fill = list(red = leaves, blue = ancestors(x, "boy")))

#----- Symbol line types and widths -----
```

```
# Dotted, thick symbols
plot(x, lty = 3, lwd = 4, cex = 2)

# Detailed specification of line types and width
plot(x, lty = list(dashed = founders), lwd = c(boy = 4))

#----- Genotypes -----

x = nuclearPed(father = "fa", mother = "mo", child = "boy") |>
  addMarker(fa = "1/1", boy = "1/2", name = "SNP") |>
  addMarker(boy = "a/b")

# Show genotypes for first marker
plot(x, marker = 1)

# Show empty genotypes for untyped individuas
plot(x, marker = 1, showEmpty = TRUE)

# Markers can also be called by name
plot(x, marker = "SNP")

# Multiple markers
plot(x, marker = 1:2)

#----- Further text annotation -----

# Founder inbreeding is shown by default
xinb = x |> setFounderInbreeding("mo", value = 0.1)
plot(xinb)

# ... but can be suppressed
plot(xinb, fouInb = NULL)

# Text can be placed around and inside symbols
plot(x, textAnnot = list(topright = 1:3, inside = LETTERS[1:3]))

# Use lists to add further options; see `?text()`
plot(x, margin = 2, textAnnot = list(
  topright = list(1:3, cex = 0.8, col = 2, font = 2, offset = 0.1),
  left = list(c(boy = "comment"), cex = 2, col = 4, offset = 2, srt = 20)))

# Exhaustive list of annotation positions
plot(singleton(1), cex = 3, textAnnot = list(top="top", left="left",
  right="right", bottom="bottom", topleft="topleft", topright="topright",
  bottomleft="bottomleft", bottomright="bottomright", inside="inside"))

#----- Special pedigrees -----

# Plot as DAG (directed acyclic graph)
plot(x, arrows = TRUE, title = "DAG")

# Medical pedigree
```

```
plot(x, carrier = "mo", aff = "boy", proband = "boy")

# Miscarriage
plot(x, miscarriage = "boy", deceased = "boy", labs = founders)

# Affected child of first cousins
y = cousinPed(1, symmetric = TRUE, child = TRUE)
plot(y, aff = leaves, proband = leaves)

# Same, with straight legs
plot(y, aff = leaves, proband = leaves, straight = TRUE)

# Twins
z = nuclearPed(children = c("tw1", "tw2", "tw3"))
plot(z, twins = data.frame(id1 = "tw1", id2 = "tw2", code = 1)) # MZ
plot(z, twins = data.frame(id1 = "tw1", id2 = "tw2", code = 2)) # DZ

# Triplets
plot(z, twins = data.frame(id1 = c("tw1", "tw2"),
                           id2 = c("tw2", "tw3"),
                           code = 2))

# Selfing
plot(selfingPed(2))

# Complex pedigree: Quadruple half first cousins
plot(quadHalfFirstCousins())

# Lists of multiple pedigree
plot(list(singleton(1), nuclearPed(1), linearPed(2)))

# Use of `drawPed()`
dat = plot(nuclearPed(), draw = FALSE)
drawPed(dat$alignment, dat$annotation, dat$scaling)
```

`plotmethods` *Internal plot methods*

Description

The main purpose of this page is to document the many options for pedigree plotting. Most of the arguments shown here may be supplied directly in `plot(x, ...)`, where `x` is a pedigree. See `plot.ped()` for many examples.

Usage

```
.pedAlignment(
  x = NULL,
  plist = NULL,
```

```
    arrows = FALSE,
    twins = NULL,
    miscarriage = NULL,
    packed = TRUE,
    width = 10,
    straight = FALSE,
    align = NULL,
    spouseOrder = NULL,
    hints = NULL,
    ...
)

.pedAnnotation(
    x,
    title = NULL,
    marker = NULL,
    sep = "/",
    missing = "-",
    showEmpty = FALSE,
    labs = labels(x),
    foldLabs = 12,
    trimLabs = TRUE,
    col = 1,
    fill = NA,
    lty = 1,
    lwd = 1,
    hatched = NULL,
    hatchDensity = 25,
    aff = NULL,
    carrier = NULL,
    deceased = NULL,
    starred = NULL,
    proband = NULL,
    textAnnot = NULL,
    textInside = NULL,
    textAbove = NULL,
    fouInb = "autosomal",
    ...
)

.pedScaling(
    alignment,
    annotation,
    cex = 1,
    symbolsize = 1,
    margins = 1,
    addSpace = 0,
    xlim = NULL,
```

```
  ylim = NULL,
  vsep2 = FALSE,
  autoScale = FALSE,
  minsize = 0.15,
  debug = FALSE,
  ...
)

.drawPed(alignment, annotation, scaling)

.annotatePed(
  alignment,
  annotation,
  scaling,
  font = NULL,
  fam = NULL,
  col = NULL,
  colUnder = 1,
  colInside = 1,
  colAbove = 1,
  cex.main = NULL,
  font.main = NULL,
  col.main = NULL,
  line.main = NA,
  ...
)
```

Arguments

x	A ped() object.
plist	Alignment list with format similar to kinship2::align.pedigree().
arrows	A logical (default = FALSE). If TRUE, the pedigree is plotted as a DAG, i.e., with arrows connecting parent-child pairs.
twins	A data frame with columns id1, id2 and code, passed on to the relation parameter of kinship2::plot.pedigree().
miscarriage	A vector of labels indicating miscarriages, shown as triangles in the pedigree plot.
packed, width, align	Parameters passed on to kinship2::align.pedigree(). Can usually be left untouched.
straight	A logical, indicating if the plot should (attempt to) use straight lines everywhere. Default: FALSE.
spouseOrder	An optional vector (or list of vectors) indicating plot ordering for spouses. (This is converted into a matrix and forward as hints; see below.)
hints	An optional list of hints passed on to kinship2::align.pedigree().
...	Further parameters passed between methods.

`title` The plot title. If NULL (default) or '', no title is added to the plot.

`marker` Either a vector of names or indices referring to markers attached to `x`, a `marker` object, or a list of such. The genotypes for the chosen markers are written below each individual in the pedigree, in the format determined by `sep` and `missing`. See also `showEmpty`. If NULL (the default), no genotypes are plotted.

`sep` A character of length 1 separating alleles for diploid markers.

`missing` The symbol (integer or character) for missing alleles.

`showEmpty` A logical, indicating if empty genotypes should be included.

`labs` A vector or function controlling the individual labels in the plot. By default, `labels(x)` are used. See Details for valid formats.

`foldLabs` A number or function controlling the folding of long labels. If a number, line breaks are inserted at roughly this width, trying to break at break-friendly characters. If a function, this is applied to each label.

`trimLabs` A logical, by default TRUE. Removes line breaks and tabs from both ends of the labels (after adding genotypes, if `marker` is not NULL).

`col` A vector or list specifying outline colours for the pedigree members. See Details for valid formats.

`fill` A vector or list specifying fill/hatch colours for the pedigree members. See Details for valid formats. Note that if `fill` is unnamed, and either `aff` or `hatched` are given, then the fill colour is applied only to those.

`lty`, `lwd` Vectors or lists specifying linetype and width of pedigree symbol outlines. See Details for valid formats.

`hatched` A vector of labels identifying members whose plot symbols should be hatched.

`hatchDensity` A number specifying the hatch density in lines per inch. Default: 25.

`aff` A vector of labels identifying members whose plot symbols should be filled. (This is typically used in medical pedigrees to indicate affected members.)

`carrier` A vector of labels identifying members whose plot symbols should be marked with a dot. (This is typically used in medical pedigrees to indicate unaffected carriers of the disease allele.)

`deceased` A vector of labels indicating deceased pedigree members.

`starred` A vector of labels indicating pedigree members that should be marked with a star in the pedigree plot.

`proband` A vector of labels indicating proband individuals, to be marked with an arrow in the plot.

`textAnnot` A list specifying further text annotation around or inside the pedigree symbols. See Details for more information.

`textInside`, `textAbove` Character vectors of text to be printed inside or above pedigree symbols. [Soft deprecated; replaced by `textAnnot`.]

`fouInb` Either "autosomal" (default), "x" or NULL. If "autosomal" or "x", inbreeding coefficients are added to the plot above the inbred founders. If NULL, or if no founders are inbred, nothing is added.

alignment	List of alignment details, as returned by .pedAlignment().
annotation	List of annotation details as returned by .pedAnnotation().
cex	Expansion factor controlling font size. This also affects symbol sizes, which by default have the width of 2.5 characters. Default: 1.
symbolsize	Expansion factor for pedigree symbols. Default: 1.
margins	A numeric indicating the plot margins. If a single number is given, it is recycled to length 4.
addSpace	A numeric of length 4, indicating extra padding (in inches) around the pedigree inside the plot region. Default: 0.
xlim, ylim	Numeric vectors of length 2, used to set par("usr") explicitly. Rarely needed by end users.
vsep2	A logical; for internal use.
autoScale	A logical. It TRUE, an attempt is made to adjust cex so that the symbol dimensions are at least minsize inches. Default: FALSE.
minsize	A positive number, by default 0.15. (See autoScale.)
debug	A logical, turning on messages from the autoscale algorithm.
scaling	List of scaling parameters as returned by .pedScaling().
font, fam	Arguments passed on to text().
colUnder, colInside, colAbove	Colour vectors.
cex.main, line.main, col.main, font.main	Parameters passed on to title().

Details

The workflow of plot.ped(x, ...) is approximately as follows:

```
# Calculate plot parameters
align = .pedAlignment(x, ...)
annot = .pedAnnotation(x, ...)
scale = .pedScaling(align, annot, ...)

# Produce plot
.drawPed(align, annot, scale)
.annotatePed(align, annot, scale)  # if `annot` contains text annotation etc
```

The labs argument controls the individual ID labels printed below the pedigree symbols. By default the output of labels(x) is used, but there are several alternative forms:

- If labs is a vector with nonempty intersection with labels(x), only these individuals will be labelled. If the vector is named, then the names are used instead of the ID label. (See Examples.)
- If labs is the word "num", then all individuals are numerically labelled following the internal ordering.

- Use `labs = NULL` to remove all labels.
- If `labs` is a function, it is replaced with `labs(x)` and handled as above. (See Examples.)

The argument `textAnnot` allows customised annotation around and inside each symbol. This takes a list of lists, whose names may include "topleft", "topright", "left", "right", "bottomleft", "bottom", "bottomright" and "inside". Each inner list should contain a character vector as its first element (with the text to be printed), followed by further arguments passed to `text()`. For example, `textAnnot = list(left = list(c(A = "1"), cex = 2))` prints a large number "1" to the left of individual A (if such an individual exists in the pedigree). See Examples.

The arguments `col`, `fill`, `lty` and `lwd` can all be indicated in a number of ways:

- An unnamed vector. This will be recycled and applied to all members. For example, `lty = 2` gives everyone a dashed outline.
- A named vector. Only pedigree members appearing in the names are affected. Example: `fill = c("1" = "red", foo = "blue")` fills individual 1 red and foo blue.
- A list of ID vectors, where the list names indicate the parameter values. Example: `col = list(red = 1:2, blue = 3:5)`.
- List entries may also be functions, taking the pedigree `x` as input and producing a vector of ID labels. The many built-in functions in ped_subgroups are particularly handy here, e.g.: `fill = list(red = founders, blue = leaves)`.

Examples

```
x = nuclearPed()

align = .pedAlignment(x)
annot = .pedAnnotation(x)
scale = .pedScaling(align, annot)

drawPed(align, annot, scale)
```

`plotPedList` *Plot a collection of pedigrees.*

Description

This function creates a row of pedigree plots, each created by `plot.ped()`. Any parameter accepted by `plot.ped()` can be applied, either to all plots simultaneously, or to individual plots. Some effort is made to guess a reasonable window size and margins, but in general the user must be prepared to do manual resizing of the plot window. See various examples in the Examples section below.

Usage

```
plotPedList(
  plots,
  widths = NULL,
  groups = NULL,
  titles = NULL,
  grouptitlesArgs = NULL,
  frames = TRUE,
  fmar = NULL,
  source = NULL,
  dev.height = NULL,
  dev.width = NULL,
  newdev = !is.null(dev.height) || !is.null(dev.width),
  verbose = FALSE,
  ...
)
```

Arguments

plots	A list of lists. Each element of plots is a list, where the first element is a pedigree, and the remaining elements are passed on to plot.ped. These elements must be correctly named. See examples below.
widths	A numeric vector of relative widths of the subplots. Recycled to length(plots) if necessary, before passed on to layout(). Note that the vector does not need to sum to 1.
groups	A list of vectors, each consisting of consecutive integers, indicating subplots to be grouped. By default the grouping follows the list structure of plots.
titles	A character vector of titles for each group. Overrides titles given in individuals subplots.
grouptitlesArgs	A list of arguments passed on to mtext() for titles.
frames	A logical indicating if groups should be framed.
fmar	A single number in the interval $[0, 0.5)$ controlling the position of the frames.
source	NULL (default), or the name or index of an element of plots. If given, marker data is temporarily transferred from this to all the other pedigrees. This may save some typing when plotting the same genotypes on several pedigrees.
dev.height, dev.width	The dimensions of the new plot window. If these are NA suitable values are guessed from the pedigree sizes.
newdev	A logical, indicating if a new plot window should be opened.
verbose	A logical.
...	Further arguments passed on to each call to plot.ped().

Details

Note that for tweaking dev.height and dev.width the function dev.size() is useful to determine the size of the active device.

Author(s)

Magnus Dehli Vigeland

See Also

plot.ped()

Examples

```
##################
# Basic examples #
##################

# Simples use: Just give a list of ped objects.
peds = list(nuclearPed(3), cousinPed(2), singleton(12), halfSibPed())
plotPedList(peds, newdev = TRUE)

# Override automatic determination of relative widths
w = c(2, 3, 1, 2)
plotPedList(peds, widths = w)

# In most cases the guessed dimensions are ok but not perfect.
# Resize plot window manually and re-plot with `newdev = FALSE` (default)
# plotPedList(peds, widths = w)

## Remove frames
plotPedList(peds, widths = w, frames = FALSE)

# Non-default grouping
plotPedList(peds, widths = w, groups = list(1, 2:3, 4), titles = 1:3)

# Parameters added in the main call are used in each sub-plot
plotPedList(peds, widths = w, labs = leaves, hatched = leaves,
            col = list(blue = males, red = females), symbolsize = 1.3)

dev.off()

################################
# Example of automatic grouping #
################################
H1 = nuclearPed()
H2 = singletons(id = c(1,3))

plotPedList(list(H1, H2), dev.height = 3, dev.width = 4,
            titles = c(expression(H[1]), expression(H[2])),
            cex = 1.5, cex.main = 1.3)

dev.off()

#############################################################
# Complex example with individual parameters for each plot #
#############################################################
```

```
# For more control of individual plots, each plot and all
# its parameters can be specified in its own list.

x1 = nuclearPed(nch = 3) |>
  addMarker(`3` = "1/2")
plot1 = list(x1, title = "Plot 1", marker = 1, deceased = 1:2, cex = 1.3,
             margins = c(7, 4, 7, 4))

x2 = cousinPed(2) |>
  addMarker(`11` = "A/A", `12` = "A/A")
plot2 = list(x2, title = "Family", marker = 1, symbolsize = 1.2, labs = NULL,
             margins = c(3, 4, 2, 4))

x3 = singleton("NN")
plot3 = list(x3, cex = 2, carrier = "NN", lty = c(NN = 2))

x4 = halfSibPed()
plot4 = list(x4, title = "Half sibs", cex = 1.3, hatched = leaves,
             col = list(red = founders), fill = list(blue = leaves),
             margins = c(7, 4, 7, 4))

plotPedList(list(plot1, plot2, plot3, plot4), widths = c(2,3,1,2),
            fmar = 0.03, groups = list(1, 2:3, 4), newdev = TRUE,
            cex.main = 1.5)

dev.off()

################################
# Example with large pedigrees #
################################

# Important to set device dimensions here

plotPedList(list(halfCousinPed(4), cousinPed(7)),
            titles = c("Large", "Very large"), widths = c(1, 1.3),
            dev.height = 8, dev.width = 6, margins = 1.5)

dev.off()
```

`print.nucleus` *S3 methods*

Description

S3 methods

Usage

```
## S3 method for class 'nucleus'
print(x, ...)
```

Arguments

x	An object
...	Not used

`print.ped` *Printing pedigrees*

Description

Print a ped object using original labels.

Usage

```
## S3 method for class 'ped'
print(x, ..., markers, verbose = TRUE)
```

Arguments

x	object of class ped.
...	(optional) arguments passed on to `print.data.frame()`.
markers	(optional) vector of marker indices. If missing, and x has less than 10 markers, they are all displayed. If x has 10 or more markers, the first 5 are displayed.
verbose	If TRUE, a message is printed if only the first 5 markers are printed. (See above).

Details

This first calls `as.data.frame.ped()` and then prints the resulting data.frame. The data.frame is returned invisibly.

randomPed *Random pedigree*

Description

Generate a random connected pedigree by applying random mating starting from a finite population.

Usage

```
randomPed(n, founders = 2, maxDirectGap = 1, selfing = FALSE, seed = NULL)
```

Arguments

n	A positive integer: the total number of individuals. Must be at least 3.
founders	A positive integer: the number of founders. Must be at least 2 unless selfing is allowed.
maxDirectGap	An integer; the maximum distance between direct descendants allowed to mate. For example, the default value of 1 allows parent-child mating, but not grandparent-grandchild. Use Inf or NULL for no restrictions.
selfing	A logical indicating if selfing is allowed. Default: FALSE.
seed	An integer seed for the random number generator (optional).

Details

Starting from an initial set of founders, a sequence of n - founders random matings is simulated. The sampling of parents in each mating is set up to ensure that the final result is connected.

Value

A connected pedigree returned as a ped object.

Examples

```
plot(randomPed(n = 7, seed = 12))

# Disallow mating between direct descendants
plot(randomPed(n = 7, seed = 12, maxDirectGap = 0))

# No restrictions on mating between direct descendants
plot(randomPed(n = 7, seed = 12, maxDirectGap = Inf))

# Allow selfing
y = randomPed(5, seed = 2, selfing = TRUE)
hasSelfing(y)
y
plot(y, arrows = TRUE)
```

readPed *Read a pedigree from file*

Description

Reads a text file in pedigree format, or something fairly close to it.

Usage

```
readPed(
  pedfile,
  colSep = "",
  header = NA,
  famid_col = NA,
  id_col = NA,
  fid_col = NA,
  mid_col = NA,
  sex_col = NA,
  marker_col = NA,
  locusAttributes = NULL,
  missing = 0,
  sep = NULL,
  colSkip = NULL,
  sexCodes = NULL,
  addMissingFounders = FALSE,
  validate = TRUE,
  ...
)
```

Arguments

pedfile	A file name
colSep	A column separator character, passed on as the sep argument of read.table(). The default is to separate on white space, that is, one or more spaces, tabs, newlines or carriage returns. (Note: the parameter sep is used to indicate allele separation in genotypes.)
header	A logical. If NA, the program will interpret the first line as a header line it contains both "id" and "sex" as part of some entries (ignoring case).
famid_col	Index of family ID column. If NA, the program looks for a column named "famid" (ignoring case).
id_col	Index of individual ID column. If NA, the program looks for a column named "id" (ignoring case).
fid_col	Index of father ID column. If NA, the program looks for a column named "fid" (ignoring case).
mid_col	Index of mother ID column. If NA, the program looks for a column named "mid" (ignoring case).

`sex_col` Index of column with gender codes (0 = unknown; 1 = male; 2 = female). If NA, the program looks for a column named "sex" (ignoring case). If this is not found, genders of parents are deduced from the data, leaving the remaining as unknown.

`marker_col` Index vector indicating columns with marker alleles. If NA, all columns to the right of all pedigree columns are used. If `sep` (see below) is non-NULL, each column is interpreted as a genotype column and split into separate alleles with `strsplit(..., split = sep, fixed = TRUE)`.

`locusAttributes` Passed on to `setMarkers()` (see explanation there).

`missing` Passed on to `setMarkers()` (see explanation there).

`sep` Passed on to `setMarkers()` (see explanation there).

`colSkip` Columns to skip, given as a vector of indices or columns names. If given, these columns are removed directly after `read.table()`, before any other processing.

`sexCodes` A list with optional entries "male", "female" and "unknown", indicating how non-default entries in the `sex` column should be interpreted. Default values: male = 1, female = 2, unknown = 0.

`addMissingFounders` A logical. If TRUE, any parent not included in the `id` column is added as a founder of corresponding sex. By default, missing founders result in an error.

`validate` A logical indicating if the pedigree structure should be validated.

`...` Further parameters passed on to `read.table()`, e.g. `comment.char` and `quote`.

Details

If there are no headers, and no column information is provided by the user, the program assumes the following column order:

- family ID (optional; guessed from the data)
- individual ID
- father's ID
- mother's ID
- sex
- marker data (remaining columns)

Reading SNP data:

Adding the argument `locusAttributes = "snp-AB"`, sets all markers to be equifrequent SNPs with alleles A and B. Moreover, the letters A and B may be replaced by other single-character letters or numbers, e.g., "snp-12" gives alleles 1 and 2.

Value

A ped object or a list of such.

Examples

```
tf = tempfile()

### Write and read a trio
trio = data.frame(id = 1:3, fid = c(0,0,1), mid = c(0,0,2), sex = c(1,2,1))
write.table(trio, file = tf, row.names = FALSE)
readPed(tf)

# With marker data in one column
trio.marker = cbind(trio, M = c("1/1", "2/2", "1/2"))
write.table(trio.marker, file = tf, row.names = FALSE)
readPed(tf)

# With marker data in two allele columns
trio.marker2 = cbind(trio, M.1 = c(1,2,1), M.2 = c(1,2,2))
write.table(trio.marker2, file = tf, row.names = FALSE)
readPed(tf)

### Two singletons in the same file
singles = data.frame(id = c("S1", "S2"),
                     fid = c(0,0), mid = c(0,0), sex = c(2,1),
                     M = c("9/14.2", "9/9"))
write.table(singles, file = tf, row.names = FALSE)
readPed(tf)

### Two trios in the same file
trio2 = cbind(famid = rep(c("trio1", "trio2"), each = 3), rbind(trio, trio))

# With column names
write.table(trio2, file = tf, col.names = TRUE, row.names = FALSE)
readPed(tf)

# Without column names
write.table(trio2, file = tf, col.names = FALSE, row.names = FALSE)
readPed(tf)

### With non-standard `sex` codes
trio3 = data.frame(id = 1:3, fid = c(0,0,1), mid = c(0,0,2),
                   sex = c("male","female","?"))
write.table(trio3, file = tf, row.names = FALSE)
readPed(tf, sexCodes = list(male = "male", female = "female", unknown = "?"))

# Cleanup
unlink(tf)
```

`relabel` *Get or modify pedigree labels*

Description

Functions for getting or changing the ID labels of pedigree members.

Usage

```
relabel(
  x,
  new = "asPlot",
  old = labels(x),
  reorder = FALSE,
  returnLabs = FALSE,
  .alignment = NULL
)

## S3 method for class 'ped'
labels(object, ...)

## S3 method for class 'list'
labels(object, ..., unlist = TRUE)
```

Arguments

x	A ped object or a list of such.
new	The following values are valid (see Details and Examples): • a character vector containing new labels. If named, interpreted as old = new • a function, which should take the old labels as input and output a character of the same length • one of the special keywords "asPlot" (default) or "generations"
old	A vector of ID labels, of the same length as new. (Ignored if new is one of the special words.) If not given, taken from the names of new if these exist.
reorder	A logical. If TRUE, reorderPed() is called on x after relabelling. Default: FALSE.
returnLabs	A logical. If TRUE, the new labels are returned as a named character vector.
.alignment	A list of alignment details for x, used if new equals "asPlot" or "generations". If not supplied, this is computed internally with .pedAlignment().
object	A ped object.
...	Not used.
unlist	A logical; if TRUE (default), the output is unlisted to a single character vector.

Details

By default, relabel(x) relabels everyone as 1, 2, ..., in the order given by the plot (top to bottom; left to right).

Alternatively, relabel(x, "generations") labels the members in the top generation I-1, I-2, ..., in the second generation II-1, II-2, ..., etc.

Value

- `labels()` returns a character vector containing the ID labels of all pedigree members. If the input is a list of ped objects, the output is a list of character vectors.
- `relabel()` by default returns a ped object similar to `x`, but with modified labels. If `returnLabs` is TRUE, the new labels are returned as a named character vector

See Also

`ped()`

Examples

```
x = nuclearPed()
x
labels(x)

y = relabel(x, new = "girl", old = 3)
y

# Back to the numeric labels
z = relabel(y)
stopifnot(identical(x,z))

# Generation labels
relabel(x, "generations")
```

---

sameGenotype *Find markers for which two individuals have the same genotype*

Description

Identifies markers for which two individuals have the same (non-missing) genotype. The comparison is done after sorting the genotypes internally.

Usage

```
sameGenotype(x, ids = typedMembers(x), count = FALSE)
```

Arguments

`x`	A ped object or a list of such. An error is raised if `x` has no marker data.
`ids`	A vector of two individual ID labels.
`count`	A logical. If TRUE, return the number of markers with shared genotype.

Value

A logical vector with one entry per marker (`NA` if either genotype is missing). If `count = TRUE`, the number of `TRUE` entries.

See Also

isHomozygous(), sortGenotypes()

Examples

```
x = nuclearPed() |>
  addMarker(name = "m1", geno = c(NA, "1/1", "1/2")) |>
  addMarker(name = "m2", geno = c(NA, "1/2", "2/1"))

sameGenotype(x, 2:3)
sameGenotype(x, 2:3, count = TRUE)
```

---

setMutmod *Set a mutation model*

Description

This function offers a convenient way to set or modify mutation models to markers attached to a pedigree. It wraps pedmut::mutationModel(), which does the main work of creating the models, but relieves the user from having to loop through the markers in order to supply the correct alleles and frequencies for each marker.

Usage

```
setMutmod(x, markers = NULL, ..., update = FALSE)
```

Arguments

x	A ped object or a list of such.
markers	A vector of names or indices referring to markers attached to x. (Default: All markers.)
...	Arguments forwarded to pedmut::mutationModel(), e.g., model, rate, etc.
update	A logical. If TRUE, existing mutation models (if present) are updated with the parameters specified in ... . If FALSE (default), any previous models are ignored, and new mutation models are created from the parameters in ... .

Details

Currently, the following models are supported:

- equal: All mutations equally likely; probability 1 - rate of no mutation
- proportional: Mutation probabilities are proportional to the target allele frequencies
- onestep: A simple model for microsatellite markers, in which mutations are only allowed to the nearest neighbours in the allelic ladder. For example, '10' may mutate to either '9' or '11' (unless '10' is the lowest allele, in which case '11' is the only option). Not applicable to loci with non-integral microvariants.

- `stepwise`: A common model for microsatellite markers. Mutation rates depend on the step size in the allelic ladder, and also the allelic classes: integral repeats like '16', versus non-integer microvariants like '16.3'.
- `custom`: Allows any mutation matrix to be provided by the user, in the `matrix` parameter
- `random`: This produces a matrix of random numbers, where each row is normalised so that it sums to 1
- `trivial`: The identity matrix; no mutations are possible

Value

An object similar to `x`.

Examples

```
### Example requires the pedmut package ###

if (requireNamespace("pedmut", quietly = TRUE)){

# A pedigree with 1 empty marker; attach 'equal' mutation model
x = nuclearPed(1) |>
  addMarker() |>
  setMutmod(model = "equal", rate = 0.01)

mutmod(x, 1)

# Update rate (but still "equal" model)
y = setMutmod(x, rate = 0.05, update = TRUE)
mutmod(y, 1)

# Change to stepwise model
z = setMutmod(x, model = "stepwise",
              rate = list(female = 0.01, male = 0.02),
              range = 0.1, rate2 = 1e-6)
mutmod(z, 1)

# Remove mutation model
w = setMutmod(x, model = NULL)
mutmod(w, 1)

}
```

---

setSNPs *Attach SNP loci to a pedigree*

Description

Create and attach a list of empty SNP markers with specified position and allele frequencies.

Usage

```
setSNPs(x, snpData)
```

Arguments

x	A ped object.
snpData	A data frame with at least 6 columns. See Details.

Details

The first 6 columns of snpData should be as follows, in order. (The column names do not matter.)

- CHROM: Chromosome (character)
- MARKER: Marker name (character)
- MB: Physical position in megabases (numeric)
- A1: First allele (single-letter character)
- A2: Second allele (single-letter character)
- FREQ1: Allele frequency of A1 (number in [0,1])

Each column must be of the stated type, or coercible to it. (For example, CHROM, A1 and A2 may be given as numbers, but will be internally converted to characters.)

Subsequent columns are assumed to contain genotypes. These columns must be named with the IDs matching individuals in x. The genotypes must use the alleles given in A1 and A2, and can be formatted with or without separator, e.g. A/C or AC.

Value

A copy of x with the indicated SNP markers attached.

Examples

```
snps = data.frame(
  CHROM  = 1:2,
  MARKER = c("M1", "M2"),
  MB     = c(1.23, 2.34),
  A1     = c("A", "G"),
  A2     = c("C", "C"),
  FREQ1  = c(0.7, 0.12),
  `2`    = c("A/C", "G/C"),
  check.names = FALSE)       # Note: `check.names = FALSE`!

x = setSNPs(nuclearPed(), snpData = snps)
x

# Inspect the results:
getMap(x)
getFreqDatabase(x)
```

sortGenotypes *Sort the alleles in each genotype*

Description

Ensure that all genotypes are sorted internally. For example, if a marker attached to x has alleles 1 and 2, then running this function will replace all genotypes "2/1" by "1/2".

Usage

```
sortGenotypes(x)
```

Arguments

x	A ped object or a list of such.

Value

An object identical to x except that the all genotypes are sorted.

Examples

```
x = singleton(1)

# Various markers with misordered genotypes
m1 = marker(x, `1` = "2/1")
m2 = marker(x, `1` = "b/a")
m3 = marker(x, `1` = "100.3/99.1")
x = setMarkers(x, list(m1, m2, m3))
x

# Sort all genotypes
y = sortGenotypes(x)
y

# Also works when input is a list of peds
sortGenotypes(list(x, x))
```

swapGenotypes *Swap genotypes between individuals*

Description

Swap genotypes between individuals

Usage

```
swapGenotypes(x, ids = NULL)
```

Arguments

x	A ped object or a list of such.
ids	A vector of 2 members of x.

Value

An object identical to x, except that the genotypes of the ids pair have been swapped.

See Also

transferMarkers()

Examples

```
x = nuclearPed() |>
  addMarker(geno = c("1/1", "2/2", "3/3"))

swapGenotypes(x, ids = 1:2)
```

---

transferMarkers *Transfer marker data*

Description

Transfer marker data between pedigrees. Any markers attached to the target are overwritten.

Usage

```
transferMarkers(
  from,
  to,
  ids = NULL,
  idsFrom = ids,
  idsTo = ids,
  erase = TRUE,
  matchNames = TRUE,
  checkSex = FALSE,
  checkAttrs = TRUE
)
```

Arguments

from	A ped or singleton object, or a list of such objects.
to	A ped or singleton object, or a list of such objects.
ids	A vector of ID labels. This should be used only if the individuals have the same name in both pedigrees; otherwise use idsFrom and idsTo instead.
idsFrom, idsTo	Vectors of equal length, denoting source individuals (in the from pedigree) and target individuals (in the to pedigree), respectively.
erase	A logical. If TRUE (default), all markers attached to to are erased prior to transfer, and new marker objects are created with the same attributes as in from. If FALSE no new marker objects are attached to to. Only the genotypes of the ids individuals are modified, while genotypes for other pedigree members - and marker attributes - remain untouched.
matchNames	A logical, only relevant if erase = FALSE. If matchNames = TRUE (default) marker names are used to ensure that genotypes are transferred into the right markers. The output only contains markers present in from, and in the same order. An error is raised if the markers are not named.
checkSex	A logical. If TRUE, it is checked that fromIds and toIds have the same sex. Default: FALSE.
checkAttrs	A logical. If TRUE, and from is a list of pedigrees, an error is raised if marker attributes differ between components. Default: TRUE.

Details

By default, genotypes are transferred between all individuals present in both pedigrees.

Value

A ped object (or a list of such) similar to to, but where all individuals also present in from have marker genotypes copied over. Any previous marker data is erased.

Examples

```
x = nuclearPed(fa = "A", mo = "B", child = "C")
x = addMarker(x, A = "1/2", B = "1/1", C = "1/2", name = "M1")

y = list(singleton("A"), nuclearPed(fa = "D", mo = "B", child = "C"))

# By default all common individuals are transferred
transferMarkers(x, y)

# Transfer data for the boy only
transferMarkers(x, y, ids = "C")

# Transfer without first erasing the target markers
z = nuclearPed(fa = "A", mo = "B", child = "C")
z = addMarker(z, A = "1/1", alleles = 1:2, name = "M1")

transferMarkers(x, z, ids = "C", erase = FALSE)
```

```
transferMarkers(x, z, ids = "C", erase = TRUE) # note the difference
```

`validatePed` *Pedigree errors*

Description

Validate the internal pedigree structure. The input may be either a (possibly malformed) `ped()` object, or its defining vectors `id`, `fid`, `mid`, `sex`.

Usage

```
validatePed(x = NULL, id = NULL, fid = NULL, mid = NULL, sex = NULL)
```

Arguments

`x`	A ped object.
`id`	A character (or coercible to character) of individual ID labels.
`fid`, `mid`	Vectors of the same length as `id`, naming each individual's father and mother. Missing parents (of founders) may be entered as "0", "" or NA.
`sex`	A numeric of the same length as `id`, describing the genders of the individuals (in the same order as `id`.) Each entry must be either 1 (=male), 2 (=female) or 0 (=unknown).

Value

If no errors are detected, the function returns NULL invisibly. Otherwise, messages describing the errors are printed to the screen and an error is raised.

Examples

```
x = nuclearPed()
validatePed(x)

# Various errors
# validatePed(id = c(1,2), fid = c(2,0), mid = c(0,1), sex = c(1,2))
```

`writePed` *Write a pedigree to file*

Description

Write a pedigree to file

Usage

```
writePed(
  x,
  prefix,
  what = "ped",
  famid = is.pedList(x),
  header = TRUE,
  merlin = FALSE,
  verbose = TRUE
)
```

Arguments

`x`	A ped object
`prefix`	A character string giving the prefix of the files. For instance, if `prefix = "myped"` and `what = c("ped", "map")`, the output files are "myped.ped" and "myped.map" in the current directory. Paths to other folder may be included, e.g. `prefix = "path-to-my-dir/myped"`.
`what`	A subset of the character vector `c("ped", "map", "dat", "freq")`, indicating which files should be created. By default only the "ped" file is created. This option is ignored if `merlin = TRUE`.
`famid`	A logical indicating if family ID should be included as the first column in the ped file. The family ID is taken from `famid(x)`. If `x` is a list of pedigrees, the family IDs are taken from `names(x)`, or if this is NULL, the component-wise `famid()` values. Missing values are replaced by natural numbers. This option is ignored if `merlin = TRUE`.
`header`	A logical indicating if column names should be included in the ped file. This option is ignored if `merlin = TRUE`.
`merlin`	A logical. If TRUE, "ped", "map", "dat" and "freq" files are written in a format readable by the MERLIN software. In particular MERLIN requires non-numerical allele labels in the frequency file.
`verbose`	A logical.

Value

A character vector with the file names.

Examples

```
x = nuclearPed(1)
x = addMarker(x, "3" = "a/b", name = "m1")

# Write to file
fn = writePed(x, prefix = tempfile("test"))

# Read
y = readPed(fn)

stopifnot(identical(x, y))
```

Index