

Package ‘pedtools’

August 4, 2020

Type Package

Title Creating and Working with Pedigrees and Marker Data

Version 0.9.5

Description A lightweight, but 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. Pedigree plots are produced by wrapping the plotting functionality of the 'kinship2' package.

License GPL-3

URL <https://github.com/magnusdv/pedtools>

Encoding UTF-8

Language en-GB

LazyData true

Imports kinship2

Suggests testthat, igraph, knitr, rmarkdown, pedmut, kableExtra

RoxygenNote 7.1.1

VignetteBuilder knitr

NeedsCompilation no

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

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

Repository CRAN

Date/Publication 2020-08-04 14:10:03 UTC

R topics documented:

as.data.frame.ped	3
as.matrix.ped	4
as.ped	5

connectedComponents	7
famid	8
founderInbreeding	8
freqDatabase	9
getAlleles	11
GetComponent	13
getGenotypes	13
getMap	14
getSex	16
inbreedingLoops	17
is.marker	19
is.ped	19
locusAttributes	20
marker	22
marker_attach	24
marker_getset	26
marker_prop	30
marker_select	33
mendelianCheck	34
mergePed	35
nMarkers	36
ped	36
pedtools	38
ped_basic	38
ped_complex	41
ped_internal	42
ped_modify	44
ped_subgroups	46
ped_utils	48
plot.ped	49
plotPedList	53
print.nucleus	56
print.ped	57
randomPed	57
readPed	58
relabel	60
sortGenotypes	61
transferMarkers	62
validatePed	63
writePed	64

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	<i>Convert ped to matrix</i>
---------------	------------------------------

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 <code>ped()</code> . 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 exemplify 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,
```

```

    validate = 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 <code>strsplit(..., split = sep, fixed = TRUE)</code> .
locusAttributes	Passed on to <code>setMarkers()</code> (see explanation there).
missing	Passed on to <code>setMarkers()</code> (see explanation there).
sep	Passed on to <code>setMarkers()</code> (see explanation there).
validate	A logical indicating if the pedigree structure should be validated.

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)

```

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, mid, 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)

```

famid	<i>Family identifier</i>
-------	--------------------------

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	<i>Inbreeding coefficients of founders</i>
-------------------	--

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
```


Arguments

x	A ped object.
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 founderInbreeding<- the updated ped object is returned.

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 or "allelic ladder" format.

Usage

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

```

readFreqDatabase(filename, format = c("list", "ladder"), ...)

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

setFrequencyDatabase(...)

getFrequencyDatabase(...)

readFrequencyDatabase(...)

writeFrequencyDatabase(...)

```

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" or "ladder".
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() .
filename	The path to a text file containing allele frequencies either in "list" or "allelic ladder" format.

Details

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

Text files containing frequencies in "list" format should look as follows, where "marker1" and "marker2" are marker names, and "a1", "a2", ... are allele labels (which may be characters or numeric, but will always be converted to characters):

```

marker1
a1 0.2
a2 0.5
a3 0.3

```

```

marker2
a1 0.9
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 (also if `format = "ladder"`) 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\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)
```

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 <code>getAlleles(x, ids, markers)</code> , or an object which can be converted by <code>as.matrix()</code> 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. For example, setting `alleles = 0` gives a simple way of removing the genotypes of some or all individuals (while keeping the markers attached).

Value

`getAlleles()` returns a character matrix with `length(ids)` rows and $2 * \text{length}(\text{markers})$ 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)
m1 = marker(x, `2` = "1/2", alleles = 1:2, name = "m1")
m2 = marker(x, `3` = "2/2", alleles = 1:2, name = "m2")
x = setMarkers(x, list(m1, 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 = setAlleles(x, alleles = 0)
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	<i>Pedigree component</i>
--------------	---------------------------

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)
```

Arguments

x	A list of ped objects
ids	A vector of ID labels (coercible to character)
checkUnique	If TRUE an error is raised if any element of ids occurs more than once 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.

Examples

```
x = list(nuclearPed(1), singleton(id = "A"))
getComponent(x, c(3, "A")) # = c(1, 2)
```

getGenotypes	<i>Genotype matrix</i>
--------------	------------------------

Description

Extract the genotypes of multiple individuals/markers in form of a matrix.

Usage

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

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.
sep	A single string to be used as allele separator in marker genotypes.
missing	A single string to be used for missing alleles.

Value

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

See Also

[getAlleles\(\)](#)

Examples

```
x = nuclearPed(1)
m1 = marker(x, `2` = 1:2, alleles = 1:2, name = "m1")
m2 = marker(x, `3` = 2, alleles = 1:2, name = "m2")
x = setMarkers(x, list(m1, m2))
```

```
getGenotypes(x)
```

```
### A list of pedigrees
```

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

```
getGenotypes(peds)
```

getMap

Tabulate marker positions

Description

Return a map of the markers attached to a pedigree.

Usage

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

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

Arguments

<code>x</code>	An object of class <code>ped</code> or a list of such.
<code>markers</code>	A vector of names or indices referring to markers attached to <code>x</code> . By default, all markers are included.
<code>na.action</code>	Either 0 (default), 1 or 2. (See Details.)
<code>verbose</code>	A logical.
<code>map</code>	Either a data frame or the path to a map file.
<code>matchNames</code>	A logical; if TRUE, pre-existing marker names of <code>x</code> will be used to assign chromosome labels and positions from <code>map</code> .
<code>...</code>	Further arguments passed to <code>read.table()</code> .

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. In the `CHROM` column, missing values are set to 0. Missing entries in `MARKER` are replaced with their index (row number). If the `POS` column contain any missing data, the entire column is replaced by 1,2,... .
- `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 in megabases. 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.

Examples

```
x = singleton(1)
m1 = marker(x, chrom = 1, posMb = 10, name = "m1")
m2 = marker(x, chrom = 1, posMb = 11)
m3 = marker(x, chrom = 1)
x = setMarkers(x, list(m1, m2, m3))

# 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))
identical(x,y)
```

getSex

*Get or set the sex of pedigree members***Description**

Functions for retrieving or changing the gender codes of specified pedigree members.

Usage

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

```
setSex(x, ids = NULL, sex)
```

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

Arguments

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

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 identical 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 gender codes of `ids` (and their spouses) are swapped (1 <-> 2).

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))

# Swapping sex is sometimes easier,
# since spouses are dealt with automatically
swapSex(x, ids = "fa")

# setting/getting sex in a pedlist
y = list.singleton(1, sex = 2), singleton(2), singleton(3))
sx = getSex(y, named = TRUE)
y2 = setSex(y, sex = sx)

stopifnot(identical(y, y2))

```

inbreedingLoops	<i>Pedigree loops</i>
-----------------	-----------------------

Description

Functions for identifying, breaking and restoring loops in pedigrees.

Usage

```

inbreedingLoops(x)

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

tieLoops(x, verbose = TRUE)

findLoopBreakers(x)

findLoopBreakers2(x, errorIfFail = TRUE)

```

Arguments

x	a <code>ped()</code> object.
loopBreakers	either NULL (resulting in automatic selection of loop breakers) or a numeric containing IDs of 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.
loop_breakers	Deprecated; renamed to loopBreakers.

Details

Pedigree loops are usually handled (by pedtools and related packages) under the hood - using the functions described here - without 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 `findLoopBreakers` identifies a set of individuals breaking all inbreeding loops, but not marriage loops. These require more machinery for efficient detection, and pedtools does this in a separate function, `findLoopBreakers2`, utilizing methods from the `igraph` package. Since this is rarely needed for most users, `igraph` is not imported when loading pedtools, only when `findLoopBreakers2` is called.

In practice, `breakLoops` first calls `findLoopBreakers` and breaks at the returned individuals. If the resulting ped object still has loops, `findLoopBreakers2` is called to break any marriage loops.

Value

For `breakLoops`, a ped object in which the indicated loop breakers are duplicated. The returned object will also have a non-null `loopBreakers` 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 is returned, still containing unbroken loops.

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 numeric vector of individual ID's.

Author(s)

Magnus Dehli Vigeland

Examples

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

```

# Pedigree with marriage loop: Double first cousins
if(requireNamespace("igraph", quietly = TRUE)) {
  y = doubleCousins(1, 1, child = TRUE)
  findLoopBreakers(y) # --> 9
  findLoopBreakers2(y) # --> 7 and 9
  y2 = breakLoops(y)
  plot(y2)

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

```

is.marker	<i>Test if something is a marker</i>
-----------	--------------------------------------

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	<i>Is an object a ped object?</i>
--------	-----------------------------------

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)))
```

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")
)

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")
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 select 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 these 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 parenthesis):

- 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
- setLocusAttributes : a modified version of x.

Examples

```

x = singleton(1)
x = addMarkers(x, marker(x, name = "m1", alleles = 1:2))
x = addMarkers(x, marker(x, name = "m2", alleles = letters[1:2], chrom = "X"))

# Change frequencies at both loci
y = setLocusAttributes(x, markers = 1:2, loc = list(afreq = c(.1, .9)))
getMarkers(y, 1)

# Set the same mutation model at both loci
z = setLocusAttributes(x, markers = 1:2, loc = list(mutmod = "proportional", rate = .1))
mutmod(z, 1)

# 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]))
w = setLocusAttributes(x, loc = locs)
getMarkers(w, 1:2)

# If `erase = TRUE` attributes not explicitly given are erased
w2 = setLocusAttributes(x, loc = locs, erase = TRUE)
chrom(w2, 2) # not "X" anymore

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

```

marker

Marker objects

Description

Creating a marker object associated with a pedigree.

Usage

```

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

```

```

    rate = NULL,
    validate = TRUE
)

```

Arguments

x	a ped object
...	one or more expressions of the form <code>id = genotype</code> , where <code>id</code> is the ID label of a member of <code>x</code> , and <code>genotype</code> is a numeric or character vector of length 1 or 2 (see Examples).
geno	a character vector of length <code>pedsize(x)</code> , with genotypes written in the format "a/b".
allelematrix	a matrix with 2 columns and <code>pedsize(x)</code> rows. If this is non-NULL, then ... must be empty.
alleles	a character (or coercible to character) containing allele names. If not given, and <code>afreq</code> is named, <code>names(afreq)</code> is used. The default action is to take the sorted vector of distinct alleles occurring in <code>allelematrix</code> , <code>geno</code> or ...
afreq	a numeric of the same length as <code>alleles</code> , 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 <code>alleles</code> is not NULL, an error is raised if <code>setequal(names(afreq), alleles)</code> is not TRUE. If <code>afreq</code> 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.
NAstrings	A character vector containing strings to be treated as missing alleles. Default: <code>c("", "0", NA, "-")</code> .
mutmod, rate	mutation model parameters. These are passed directly to <code>pedmut::mutationModel()</code> ; see there for details. Note: <code>mutmod</code> corresponds to the <code>model</code> parameter. Default: NULL (no mutation model).
validate	if TRUE, the validity of the created marker object is checked.

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; default `rep.int(1/length(alleles), length(alleles))`)
- `chrom` (chromosome number; default = NA)
- `posMb` (physical location in megabases; default = NA)
- `name` (marker identifier; default = NA)
- `mutmod` (a list of two (male and female) mutation matrices; default = NULL)

See Also[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"))

# 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 ...
marker(x, fa = "1/1", mo = "1/2")

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

# For homozygous genotypes, a single allele suffices
marker(x, fa = 1)

# Attaching a marker to the pedigree
m = marker(x) # By default a SNP with alleles 1,2
x = setMarkers(x, m)

# A marker with a "proportional" mutation model,
# with different rates for males and females
mutrates = list(female = 0.1, male = 0.2)
marker(x, alleles = 1:2, mutmod = "prop", rate = mutrates)
```

`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.

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 <code>pedsize(x)</code> 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 <code>sep</code> 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 <code>alleleMatrix</code> is interpreted as a genotype, and will be split by calling <code>strsplit(..., split = sep, fixed = TRUE)</code> . If <code>alleleMatrix</code> contains entries with <code>"/"</code> , this will be taken as separator by default. (To override this behaviour, put <code>sep = FALSE</code> .)
checkCons	A logical. If TRUE (default), each marker is checked for consistency with <code>x</code> .

Details

The most general format of `locusAttributes` 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 (`rep.int(1/L, L)`, where `L = length(alleles)`)
- `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 just a single list of attributes (not a list of lists), then it is repeated to match the number of markers.

Two alternative format of `locusAttributes` are allowed: If a `data.frame` or `matrix` is given, an attempt is made to interpret it as a frequency database in allelic ladder format. Such an interpretation is also attempted if `locusAttributes` is a list of named frequency vectors (where the names are the allele labels).

Value

A ped object.

Examples

```
x = singleton(1)
m1 = marker(x, '1' = 1:2)
m2 = marker(x, '1' = 'a')

x = setMarkers(x, m1)
x = addMarkers(x, m2)
x

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

marker_getset

Get/set marker attributes

Description

These functions can be used to manipulate a single attribute of one or several markers. Each get-er/setter can be used in two ways: Either directly on a marker object, or on a ped object which has markers attached to it.

Usage

```
genotype(x, ...)

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

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

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, ...)

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

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

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

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

alleles(x, ...)

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

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

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, ...)

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, ...)

## 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, ...)

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, ...)

## 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, ...)

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, ...)

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

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

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	A ped object or a marker object
...	Further arguments, not used in most of these functions
id	The ID label of a single pedigree member
value	Replacement value(s)
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

The getters return the value of the query. The setters perform in-place modification of the input.

Examples

```

x = nuclearPed(1)
x = setMarkers(x, locusAttributes = list(name = "M", alleles = 1:2))

# Set genotype
genotype(x, marker = "M", id = 1) = 1:2
genotype(x, marker = "M", id = 3) = 1

# Genotypes are returned as a vector of length 2
genotype(x, marker = "M", id = 1)

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

```

```
# Check the new frequencies
afreq(x, "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 = seq_len(nMarkers(x)), ...)
```

```
## S3 method for class 'list'
```

```
emptyMarker(x, markers = seq_len(nMarkers(x)), ...)
```

```
nTyped(x, ...)
```

```
## Default S3 method:
```

```
nTyped(x, ...)
```

```
## S3 method for class 'marker'
```

```
nTyped(x, ...)
```

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

```
nTyped(x, markers = seq_len(nMarkers(x)), ...)
```

```
## S3 method for class 'list'
```

```
nTyped(x, markers = seq_len(nMarkers(x)), ...)
```

```
nAlleles(x, ...)
```

```
## Default S3 method:
```

```
nAlleles(x, ...)
```

```

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

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

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

isXmarker(x, ...)

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

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

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

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

allowsMutations(x, ...)

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

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

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

## S3 method for class 'list'
allowsMutations(x, markers = seq_len(nMarkers(x)), ...)

```

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, in the case that x is a ped object or a list of such. 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.

If x is a ped object, or a list of such, the output is a vector of the same length as markers (which includes all attached markers by default), 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
genotype(x[[2]], marker = 3, id = 10) = 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	<i>Select or remove attached markers</i>
---------------	--

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 <code>nMarkers(x)</code>), or <code>NULL</code> .
chroms	A vector of chromosome names, or <code>NULL</code>
fromPos	A single number or <code>NULL</code>
toPos	A single number or <code>NULL</code>

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\(\)](#)

mendelianCheck	<i>Check for Mendelian errors</i>
----------------	-----------------------------------

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(3)

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

mendelianCheck(x)
```

mergePed	<i>Merge two pedigrees</i>
----------	----------------------------

Description

This function merges two ped objects, joining them at the individuals with equal ID labels. This is especially useful for building 'top-heavy' pedigrees. Only ped objects without marker data are supported.

Usage

```
mergePed(x, y, ...)
```

Arguments

x, y	ped() objects
...	further arguments passed along to ped(), e.g. famid, validate and reorder.

Value

A ped object.

Author(s)

Magnus Dehli Vigeland

Examples

```
# Creating a trio where each parent have first cousin parents.  
# (Alternatively, this could be built using many calls to addParents().)  
  
# Paternal family  
x = cousinPed(1, child = TRUE)  
x = addSon(x, 9)  
  
# Maternal family  
y = cousinPed(1, child = TRUE)  
y = relabel(y, c(101:108, 10))  
y = swapSex(y, 10)  
  
# Joining x and y at the common individuals (in this case: `10`)  
z = mergePed(x, y)  
  
# Plot all three pedigrees  
opar = par(mfrow = c(1, 3))  
plot(x); plot(y); plot(z, col = list(red = labels(y)))  
  
# Reset graphical parameters
```

par(opar)

nMarkers	<i>The number of markers attached to a pedigree</i>
----------	---

Description

The number of markers attached to a pedigree

Usage

nMarkers(x)

hasMarkers(x)

Arguments

x A ped object or a list of such (see Value).

Value

The function nMarkers returns the number of marker objects attached to x. If x is a list of pedigrees, an error is raised unless all of them have the same number of markers.

The function hasMarkers returns TRUE if nMarkers(x) > 0.

ped	<i>Pedigree construction</i>
-----	------------------------------

Description

This is the basic constructor of ped objects. Utility functions for creating many common pedigree structures are described in [ped_basic](#).

Usage

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

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

Arguments

id	a vector (numeric or character) of individual ID labels.
fid	a vector of the same length as id, containing the labels of the fathers. In other words fid[i] is the father of id[i], or 0 if id[i] is a founder.
mid	a vector of the same length as id, containing the labels of the mothers. In other words mid[i] is the mother of id[i], or 0 if id[i] is a founder.
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. If TRUE, the pedigree is reordered so that all parents precede their children.
validate	a logical. If TRUE, <code>validatePed()</code> is run before returning the pedigree.
verbose	a logical.

Details

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

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 gender (`sex = 0`).

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

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 $j = 1, 2, \dots$, the entry FIDX[j] is 0 if ID[j] has no father within the pedigree; otherwise ID[FIDX[j]] is the father of ID[j].
- MIDX : An integer vector with maternal indices: For each $j = 1, 2, \dots$, the entry MIDX[j] is 0 if ID[j] has no mother within the pedigree; 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: TRUE if the pedigree is inbred.
- 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[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
w = ped(id = 1:4, fid = c(2,0,0,0), mid = c(3,0,0,0), sex = c(1,1,2,1))
stopifnot(is.pedList(w), length(w) == 2)
```

pedtools*pedtools: Tools for working with pedigrees in R*

Description

A collection of tools for creating, manipulating and visualising pedigrees.

ped_basic*Create simple 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, removal = 0, side = c("right", "left"), child = FALSE)

halfCousinPed(degree, removal = 0, side = c("right", "left"), 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: "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.
child	A logical: Should an inbred child be added to the two cousins?
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	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. (Similarly for halfCousinPed.)

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 by using 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)
```

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.

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 $k = 0, \dots, n-1$.

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.
```

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)

parentsBeforeChildren(x)

hasParentsBeforeChildren(x)

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

Arguments

x	A ped object
neworder	A permutation of labels(x) or of vector 1:pedsize(x). By default, the sorting order of the ID labels is used.
ids	A character vector (or coercible to one) of original ID labels.
errorIfUnknown	A logical. If TRUE (default), internalID(x, ids) results in an error if not all elements of ids are recognised as names of members in x.

Details

The internal ordering is usually of little importance for end users, with one important exception: Certain pedigree-traversing algorithms require parents to precede their children. A special function, parentsBeforeChildren() is provided for this purpose. This is a wrapper of the more general reorderPed() which allows any permutation of the members.

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 utility internalID() converts ID labels to indices in the internal ordering.

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
internalID(x, ids = 3)
internalID(x, ids = "3")

y = parentsBeforeChildren(x)
internalID(y, ids = 3)

# A different ordering
reorderPed(x, c(2,1,3))
```

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 = 1,
  sex = 1,
  ids = NULL,
  verbose = TRUE
)
```

```
addSon(x, parent, id = NULL, verbose = TRUE)
```

```
addDaughter(x, parent, id = NULL, verbose = TRUE)
```

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

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

```
branch(x, id)
```

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

Arguments

x	A ped object.
father, mother	Single ID labels. At least one of these must belong to an existing pedigree member. The other label may either: 1) belong to an existing member, 2) not belong to any existing member, or 3) be missing (i.e. not included in the function call). In cases 2 and 3 a new founder is added to the pedigree. In case 2 its label is the one given, while in case 3 a suitable label is created by the program (see Details).
nch	A positive integer indicating the number of children to be created. Default: 1.
sex	Gender codes of the created children (recycled if needed).
ids	A character vector (or coercible to such) with ID labels. In addChildren the (optional) ids argument is used to specify labels for the created children. If given, its length must equal nch. If not given, labels are assigned automatically as explained in Details.

verbose	A logical: Verbose output or not.
parent	The ID label (coercible to character) of a single pedigree member, which will be the father or mother (depending on its gender) of the new child.
id	The ID label of some existing pedigree member.
subset	A character vector (or coercible to such) with ID labels forming a connected sub-pedigree.
...	Not used.

Details

In `addChildren()` and `addParents()`, labels of added individuals are created automatically if they are not specified by the user. In the automatic case, the labelling depends on whether the existing labels are integer-like or not (i.e. if `labels(x)` equals `as.character(as.integer(labels(x)))`.) If so, the new labels are integers subsequent to the largest of the existing labels. If not, the new labels are "NN_1", "NN_2", ... If any such label already exists, the numbers are adjusted accordingly.

`addSon()` and `addDaughter()` are wrappers for a common use of `addChildren()`, namely adding a single child to a pedigree member. Note that its argument `parent` is gender-neutral, unlike in `addChildren()` where you have to know the parental genders. Also note that the other parent is always created as a new individual. Thus, applying `addDaughter()` twice with the same parent will create half sisters.

In `removeIndividuals()` all descendants of `ids` are also removed. Any individuals (spouses) left unconnected to the remaining pedigree are also removed.

The `branch()` function extracts the sub-pedigree formed by `id` and all his/her spouses and descendants.

Finally, `subset()` can be used to extract any connected sub-pedigree. (Note that in the current implementation, the function does not actually check that the indicated subset forms a connected pedigree; failing to comply with this may lead to obscure errors.)

Value

The modified ped object.

Author(s)

Magnus Dehli Vigeland

See Also

[ped\(\)](#), [relabel\(\)](#), [swapSex\(\)](#)

Examples

```
x = nuclearPed(1)

# To see the effect of each command below, use plot(x) in between.
x = addSon(x, 3)
x = addParents(x, id = 4, father = 6, mother = 7)
```

```
x = removeIndividuals(x, 4)
```

ped_subgroups	<i>Pedigree subgroups</i>
---------------	---------------------------

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)
```

```
offspring(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)
```

```
cousins(x, id, degree = 1, removal = 0, half = NA, internal = FALSE)
```

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

```
ancestors(x, id, inclusive = FALSE, internal = FALSE)
```

```
commonAncestors(x, ids, inclusive = FALSE, internal = FALSE)
```

```
descendants(x, id, inclusive = FALSE, internal = FALSE)
```

```
commonDescendants(x, ids, inclusive = FALSE, internal = FALSE)
```

Arguments

<code>x</code>	A <code>ped()</code> object.
<code>internal</code>	A logical indicating whether <code>id</code> (or <code>ids</code>) refers to the internal order.
<code>id, ids</code>	A character (or coercible to such) with one or several ID labels.
<code>degree, removal</code>	Non-negative integers.
<code>half</code>	a logical or NA. If TRUE (resp FALSE), only half (resp. full) siblings/cousins/nephews/nieces are returned. If NA, both categories are included.
<code>inclusive</code>	A logical indicating whether an individual should be counted among his or her own ancestors/descendants

Value

The functions `ancestors(x, id)` and `descendants(x, id)` return a vector 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.

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

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`, `cousins`, `grandparents`, `nephews_nieces`, `children`, `parents`, `siblings`, `spouses`, `unrelated`, each returns a vector containing the IDs of all pedigree members having the specified relationship with `id`.

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))
stopifnot(setequal(spouses(x, 2), c(3,8)),
          setequal(children(x, 2), c(4,9)),
```

```
setequal(descendants(x, 2), c(4,6,7,9)),
setequal(leaves(x), c(6,7,9)))
```

ped_utils

Pedigree utilities

Description

Various utility functions for ped objects

Usage

```
pedsize(x)
```

```
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.
chromType	Either "autosomal" (default) or "x".

Details

The functions `pedsize()`, `hasUnbrokenLoops()`, `hasInbredFounders()` and `hasSelfing()` allow as input either a single ped object or a list of such. In the latter case each function returns TRUE if it is TRUE for any of the components.

Value

- `pedsize(x)` returns the number of pedigree members in each component of `x`.
- `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 if 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 it *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))

# 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.ped

Plot pedigrees with genotypes

Description

This is the main function for pedigree plotting, with many options for controlling the appearance of pedigree symbols and accompanying labels. Most of the work is done by the plotting functionality in the `kinship2` package.

Usage

```
## S3 method for class 'ped'
plot(
  x,
  marker = NULL,
  sep = "/",
  missing = "-",
  showEmpty = FALSE,
  labs = labels(x),
  title = NULL,
  col = 1,
  aff = NULL,
  hatched = NULL,
  shaded = NULL,
  deceased = NULL,
  starred = NULL,
  hints = NULL,
  fouInb = "autosomal",
  margins = c(0.6, 1, 4.1, 1),
  keep.par = FALSE,
  skipEmptyGenotypes = NULL,
  skip.empty.genotypes = NULL,
  id.labels = NULL,
  ...
)

## S3 method for class 'singleton'
plot(
  x,
  marker = NULL,
  sep = "/",
  missing = "-",
  showEmpty = FALSE,
  labs = labels(x),
  title = NULL,
  col = 1,
  aff = NULL,
  hatched = NULL,
  shaded = NULL,
  deceased = NULL,
  starred = NULL,
  fouInb = "autosomal",
  margins = c(8, 0, 0, 0),
  yadj = 0,
  id.labels = NULL,
  ...
)
```

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

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

Arguments

x	a <code>ped()</code> object.
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 <code>sep</code> and <code>missing</code> . See also <code>showEmpty</code> below. If <code>NULL</code> (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 included in the plot. Alternative forms: <ul style="list-style-type: none"> • If <code>labs</code> is a vector with nonempty intersection with <code>labels(x)</code>, these individuals will be labelled. If the vector is named, then the (non-empty) names are used instead of the ID label. (See Examples.) • If <code>labs</code> is <code>NULL</code>, or has nonempty intersection with <code>labels(x)</code>, then no labels are drawn. • If <code>labs</code> is the word "num", then all individuals are numerically labelled following the internal ordering. • If <code>labs</code> is a function, it will be replaced with <code>labs(x)</code> and handled as above. (See Examples.)
title	the plot title. If <code>NULL</code> or "", no title is added to the plot.
col	a vector of colours for the pedigree members, recycled if necessary. Alternatively, <code>col</code> can be a list assigning colours to specific members. For example if <code>col = list(red = "a", blue = c("b", "c"))</code> then individual "a" will be red, "b" and "c" blue, and everyone else black. By default everyone is drawn black.
aff	a vector of ID labels indicating pedigree members whose plot symbols should be filled.
hatched	a vector of ID labels indicating pedigree members whose plot symbols should be hatched.
shaded	(Deprecated) synonym of <code>hatched</code>
deceased	a vector of ID labels indicating deceased pedigree members.
starred	a vector of ID labels indicating pedigree members that should be marked with a star in the pedigree plot.
hints	a list with alignment hints passed on to <code>kinship2::align.pedigree()</code> . Usually not necessary.
fouInb	either "autosomal" (default), "x" or <code>NULL</code> . If "autosomal" or "x", inbreeding coefficients are added to the plot above the inbred founders. If <code>NULL</code> , or if no founders are inbred, nothing is added.

margin	a numeric of length 4 indicating the plot margins. For singletons only the first element (the 'bottom' margin) is used.
keep.par	A logical (default = FALSE). If TRUE, the graphical parameters are not reset after plotting, which may be useful for adding additional annotation.
skipEmptyGenotypes	Deprecated; use showEmpty instead.
skip.empty.genotypes	Deprecated; use showEmpty instead.
id.labels	Deprecated; use labs instead
...	arguments passed on to plot.pedigree in the kinship2 package. In particular symbolsize and cex can be useful.
yadj	A tiny adjustment sometimes needed to fix the appearance of singletons.

Details

plot.ped is in essence an elaborate wrapper for [kinship2::plot.pedigree\(\)](#).

Author(s)

Magnus Dehli Vigeland

See Also

[kinship2::plot.pedigree\(\)](#)

Examples

```
x = nuclearPed(father = "fa", mother = "mo", child = "boy")
m = marker(x, fa = "1/1", boy = "1/2", name = "SNP")

plot(x, marker = m)

# Markers attached to `x` may be called by name
x = setMarkers(x, m)
plot(x, marker = "SNP")

# Other options
plot(x, marker = "SNP", hatched = typedMembers(x),
      starred = "fa", deceased = "mo")

# Filled symbols
plot(x, aff = males(x))

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

# Label only some members; rename the father
plot(x, labs = c(FATHER = "fa", "boy"))
```

```

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

# Colours
plot(x, col = list(red = "fa", green = "boy"), hatched = "boy")

# Founder inbreeding is shown by default
founderInbreeding(x, "mo") = 0.1
plot(x)

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

#-----
# In some cases, the plotting machinery of `kinship2` needs a hint
# (see ?kinship2::align.pedigree)

# Example with 3/4-siblings
y = nuclearPed(2)
y = addChildren(y, 3, mother = 5, nch = 1)
y = addChildren(y, 4, mother = 5, nch = 1)

plot(y) # bad

hints = list(order = 1:7, spouse = rbind(c(3,5,0), c(5,4,0)))
plot(y, hints = hints) # good

```

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,
  frames = TRUE,
  fmar = NULL,
  frametitles = 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 <code>plot.ped</code> . These elements must be correctly named. See examples below.
widths	A numeric vector of relative widths of the subplots. Recycled to <code>length(plots)</code> if necessary, before passed on to <code>layout()</code> . 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.
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.
frametitles	Deprecated; use <code>titles</code> instead.
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 <code>plot.ped()</code> .

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 #
#####

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

# Modify the relative widths (which are not guessed)
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), titles = 1:2)

# Parameters added in the main call are used in each sub-plot
plotPedList(peds, widths = w, margins = c(2, 4, 2, 4), labs = leaves,
            hatched = leaves, symbolsize = 1.3, col = list(red = 1))

dev.off()

#####
# Example of automatic grouping #
#####
H1 = nuclearPed()
H2 = list(singleton(1), singleton(3)) # grouped!

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

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)
m1 = marker(x1, `3` = 1:2)
marg1 = c(7, 4, 7, 4)
plot1 = list(x1, marker = m1, margins = marg1, title = "Plot 1",
            deceased = 1:2, cex = 1.3)
```

```

x2 = cousinPed(2)
m2 = marker(x2, alleles = "A")
genotype(m2, leaves(x2)) = "A"
marg2 = c(3, 4, 2, 4)
plot2 = list(x2, marker = m2, margins = marg2, title = "Plot 2",
            symbolsize = 1.2, labs = NULL)

x3 = singleton("Mr. X")
marg3 = c(10, 0, 0, 0)
plot3 = list(x3, margins = marg3, title = "Plot 3",
            symbolsize = 1, cex = 2)

x4 = halfSibPed()
hatched = 4:5
col = list(red = founders(x4), blue = leaves(x4))
marg4 = marg1
plot4 = list(x4, margins = marg4, title = "Plot 4", cex = 1.3,
            hatched = hatched, col = col)

plotPedList(list(plot1, plot2, plot3, plot4), widths = c(2,3,1,2),
            groups = list(1, 2:3, 4), newdev = TRUE)

dev.off()

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

# Important to set device dimensions here

plotPedList(list(halfCousinPed(4), cousinPed(7)),
            titles = c("Large", "Very large"),
            dev.height = 8, dev.width = 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	<i>Printing pedigrees</i>
-----------	---------------------------

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	<i>Random pedigree</i>
-----------	------------------------

Description

Generate a random pedigree by applying random mating starting from a finite population. The resulting pedigree will have $f + g$ members, where f is the number of founders and g is the number of matings.

Usage

```
randomPed(g, founders = rpois(1, 3) + 1, selfing = FALSE, seed = NULL)
```

Arguments

<code>g</code>	A positive integer: The number of matings.
<code>founders</code>	A positive integer: The size of the initial population.
<code>selfing</code>	A logical indicating if selfing is allowed.
<code>seed</code>	A numerical seed for random number generation. (Optional.)

Details

The sampling scheme for choosing parents in each mating depends on the `selfing` parameter. If `selfing = FALSE`, a father is randomly sampled from the existing males, and a mother from the existing females. If `selfing = TRUE` then one parent P1 is sampled first (among all members), and then a second parent from the set consisting of P1 and all members of the opposite sex. The gender of the child is randomly chosen with equal probabilities.

Value

A ped object.

Examples

```
randomPed(3, 3)
randomPed(3, 3, selfing = TRUE)
```

<code>readPed</code>	<i>Read a pedigree from file</i>
----------------------	----------------------------------

Description

Read a pedigree from file

Usage

```
readPed(
  pedfile,
  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,
  validate = TRUE,
  ...
)
```

Arguments

pedfile	A file name
header	A logical. If NA, the program will interpret the first line as a header line if the first entry contains "id" AND the word "sex" is an entry.
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 <code>strsplit(..., split = sep, fixed = TRUE)</code> .
locusAttributes	Passed on to <code>setMarkers()</code> (see explanation there).
missing	Passed on to <code>setMarkers()</code> (see explanation there).
sep	Passed on to <code>setMarkers()</code> (see explanation there).
validate	A logical indicating if the pedigree structure should be validated.
...	Further parameters passed on to <code>read.table()</code> , e.g. <code>sep</code> , <code>comment.char</code> and <code>quote</code> .

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, sep = "/")
```

```

# With marker data in two allele columns
trio.marker2 = cbind(trio, A1 = c(1,2,1), A2 = 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, sep = "/")

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

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

# With column names
write.table(trio2, file = tf, col.names = FALSE, row.names = FALSE)
readPed(tf, famid = 1, id = 2, fid = 3, mid = 4, sex = 5)

# 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, old = labels(x), reorder = FALSE)
```

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

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

Arguments

x A ped object or a list of such.

new, old	Character vectors (or coercible to character) of the same length. ID labels in old are replaced by those in new.
reorder	A logical. If TRUE, <code>reorderPed()</code> is called on x after relabelling. Default: FALSE.
object	A ped object
...	Not used

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()` returns ped object similar to the input except for the labels.

Author(s)

Magnus Dehli Vigeland

See Also

[ped\(\)](#)

Examples

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

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

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` = c('b','a'))
m3 = marker(x, `1` = c("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))
```

transferMarkers

Transfer marker data

Description

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

Usage

```
transferMarkers(from, to, ids = NULL, erase = TRUE, matchNames = TRUE)
```

Arguments

<code>from</code>	a ped or singleton object, or a list of such objects.
<code>to</code>	a ped or singleton object, or a list of such objects.
<code>ids</code>	a vector of ID labels. The indicated individuals must be present in both pedigrees. By default, genotypes are transferred between all shared individuals.
<code>erase</code>	a logical. If TRUE (default), all markers attached to <code>to</code> are erased prior to transfer, and new marker objects are created with the same attributes as in <code>from</code> . If FALSE no new marker objects are attached to <code>to</code> . Only the genotypes of the <code>ids</code> individuals are modified, while genotypes for other pedigree members - and marker attributes - remain untouched.
<code>matchNames</code>	a logical, only relevant if <code>erase = FALSE</code> . If <code>matchNames = TRUE</code> (default) marker names are used to ensure genotypes are transferred into the right markers. The output will contain only the markers in <code>from</code> , in the same order. (An error is raised if the markers are not named.) In this case

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 = "father", mo = "mother", children = "boy")
m = marker(x, father = 1:2, mother = 1, boy = 1:2)
x = setMarkers(x, m)

y = list(singleton("father"), nuclearPed(mo = "mother", children = "boy"))

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

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

# Transfer without erasing marker attributes or others genotypes
# Note that `erase = FALSE` requires markers to be named
z = nuclearPed(children = "boy")
z = setMarkers(z, marker(z, '1' = c(2,2), alleles = 1:2, afreq = c(.1, .9)))
name(x, 1) = name(z, 1) = 'M1'
z2 = transferMarkers(x, z, ids = "boy", erase = FALSE)
z2
# Frequencies are not transferred
afreq(z2, 1)
```

 validatePed

Pedigree errors

Description

Validate the internal structure of a ped object.

Usage

```
validatePed(x)
```

Arguments

`x` object of class `ped`.

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.

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 <code>prefix = "myped"</code> and <code>what = c("ped", "map")</code> , the output files are "myped.ped" and "myped.map" in the current directory. Paths to other folder may be included, e.g. <code>prefix = "path-to-my-dir/myped"</code> .
what	A subset of the character vector <code>c("ped", "map", "dat", "freq")</code> , indicating which files should be created. By default only the "ped" file is created. This option is ignored if <code>merlin = TRUE</code> .
famid	A logical indicating if family ID should be included as the first column in the ped file. The family ID is taken from <code>famid(x)</code> . If <code>x</code> is a pedlist, the family IDs are taken from <code>names(x)</code> , or if this is <code>NULL</code> , the component-wise <code>famid()</code> values. Missing values are replaced by natural numbers. This option is ignored if <code>merlin = TRUE</code> .
header	A logical indicating if column names should be included in the ped file. This option is ignored if <code>merlin = TRUE</code> .
merlin	A logical. If <code>TRUE</code> , "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 = setMarkers(x, marker(x, "3" = "a/b", name = "m1"))

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

# Read
y = readPed(fn)

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

Index

addChildren (ped_modify), 44
addDaughter (ped_modify), 44
addMarkers (marker_attach), 24
addParents (ped_modify), 44
addSon (ped_modify), 44
afreq (marker_getset), 26
afreq<- (marker_getset), 26
alleles (marker_getset), 26
allowsMutations (marker_prop), 30
ancestors (ped_subgroups), 46
ancestralPed (ped_basic), 38
as.data.frame.ped, 3
as.data.frame.ped(), 3, 57
as.matrix.ped, 4
as.matrix.ped(), 3
as.ped, 5
as_kinship2_pedigree (plot.ped), 49

branch (ped_modify), 44
breakLoops (inbreedingLoops), 17
breakLoops(), 37

children (ped_subgroups), 46
chrom (marker_getset), 26
chrom<- (marker_getset), 26
commonAncestors (ped_subgroups), 46
commonDescendants (ped_subgroups), 46
connectedComponents, 7
cousinPed (ped_basic), 38
cousins (ped_subgroups), 46

descendants (ped_subgroups), 46
dev.size(), 54
doubleCousins (ped_complex), 41
doubleFirstCousins (ped_complex), 41

emptyMarker (marker_prop), 30

famid, 8
famid<- (famid), 8
father (ped_subgroups), 46

females (ped_subgroups), 46
findLoopBreakers (inbreedingLoops), 17
findLoopBreakers2 (inbreedingLoops), 17
founderInbreeding, 8
founderInbreeding(), 37, 48
founderInbreeding<-
 (founderInbreeding), 8
founders (ped_subgroups), 46
freqDatabase, 9
fullSibMating (ped_complex), 41

genotype (marker_getset), 26
genotype<- (marker_getset), 26
getAlleles, 11
getAlleles(), 14
getComponent, 13
getFreqDatabase (freqDatabase), 9
getFrequencyDatabase (freqDatabase), 9
getGenotypes, 13
getLocusAttributes (locusAttributes), 20
getMap, 14
getMarkers (marker_select), 33
getSex, 16
grandparents (ped_subgroups), 46

halfCousinPed (ped_basic), 38
halfSibPed (ped_basic), 38
halfSibStack (ped_complex), 41
hasCommonAncestor (ped_utils), 48
hasInbredFounders (ped_utils), 48
hasMarkers (nMarkers), 36
hasParentsBeforeChildren
 (ped_internal), 42
hasSelfing (ped_utils), 48
hasUnbrokenLoops (ped_utils), 48

inbreedingLoops, 17
internalID (ped_internal), 42
is.marker, 19
is.markerList (is.marker), 19

- is.ped, 19
- is.pedList (is.ped), 19
- is.singleton (is.ped), 19
- isXmarker (marker_prop), 30

- kinship2::plot.pedigree(), 52

- labels.list (relabel), 60
- labels.ped (relabel), 60
- layout(), 54
- leaves (ped_subgroups), 46
- linearPed (ped_basic), 38
- locusAttributes, 20

- males (ped_subgroups), 46
- marker, 22, 29
- marker_attach, 24, 24
- marker_getset, 26
- marker_prop, 30
- marker_select, 33
- mendelianCheck, 34
- mergePed, 35
- mother (ped_subgroups), 46
- mutmod (marker_getset), 26
- mutmod<- (marker_getset), 26

- nAlleles (marker_prop), 30
- name (marker_getset), 26
- name<- (marker_getset), 26
- nephews_nieces (ped_subgroups), 46
- nMarkers, 36
- nonfounders (ped_subgroups), 46
- nTyped (marker_prop), 30
- nuclearPed (ped_basic), 38

- offspring (ped_subgroups), 46

- parents (ped_subgroups), 46
- parentsBeforeChildren (ped_internal), 42
- ped, 23, 29, 36, 42, 59
- ped(), 4, 5, 16, 17, 19, 20, 34, 35, 40, 43, 45, 47, 49, 51, 61
- ped_basic, 36, 38, 38, 41, 42
- ped_complex, 40, 41
- ped_internal, 42
- ped_modify, 38, 44
- ped_subgroups, 38, 40, 46
- ped_utils, 48
- pedmut::mutationModel(), 23
- pedsize (ped_utils), 48

- pedtools, 38
- peelingOrder (ped_utils), 48
- plot.ped, 49
- plot.ped(), 53, 54
- plot.pedList (plot.ped), 49
- plot.singleton (plot.ped), 49
- plotPedList, 53
- posMb (marker_getset), 26
- posMb<- (marker_getset), 26
- print.data.frame(), 57
- print.nucleus, 56
- print.ped, 57

- quadHalfFirstCousins (ped_complex), 41

- randomPed, 57
- read.table(), 10, 59
- readFreqDatabase (freqDatabase), 9
- readFrequencyDatabase (freqDatabase), 9
- readPed, 58
- relabel, 60
- relabel(), 38, 45
- removeIndividuals (ped_modify), 44
- removeMarkers (marker_select), 33
- reorderPed (ped_internal), 42
- reorderPed(), 61
- restorePed (as.matrix.ped), 4

- selectMarkers (marker_select), 33
- selfingPed (ped_basic), 38
- setAlleles (getAlleles), 11
- setAlleles(), 11
- setFreqDatabase (freqDatabase), 9
- setFrequencyDatabase (freqDatabase), 9
- setLocusAttributes (locusAttributes), 20
- setLocusAttributes(), 11
- setMap (getMap), 14
- setMarkers (marker_attach), 24
- setMarkers(), 6, 11, 34, 59
- setSex (getSex), 16
- siblings (ped_subgroups), 46
- singleton (ped), 36
- singleton(), 19, 40
- sortGenotypes, 61
- spouses (ped_subgroups), 46
- subnucs (ped_utils), 48
- subset.ped (ped_modify), 44
- swapSex (getSex), 16
- swapSex(), 45

tieLoops (inbreedingLoops), 17
transferMarkers, 62
transferMarkers(), 12
typedMembers (ped_subgroups), 46

unrelated (ped_subgroups), 46
untypedMembers (ped_subgroups), 46

validatePed, 63
validatePed(), 37

whichMarkers (marker_select), 33
writeFreqDatabase (freqDatabase), 9
writeFrequencyDatabase (freqDatabase), 9
writePed, 64