Package: sibgrouper (via r-universe)

May 31, 2026

Title Group siblings

Author Mark Bravington

Maintainer Mark Bravington <markb2@summerinsouth.net>

Description Distinct-Parent-Groups from pairwise sibship comparisons, plus tools to diagnose misclassification

Imports Matrix, mvbutils, atease, kinference

Suggests igraph

Additional_repositories https://markbravington.r-universe.dev

License GPL3

Version 1.1.17

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

Date/Publication 2026-05-17 10:18:09 UTC

RemoteUrl https://github.com/closekin/sibgrouper

RemoteRef HEAD

RemoteSha 35710b9dea215187c92fac8ec608087ce939d694

Contents

`sibgrouper-package` *How to use the sibgrouper package*

Description

Package **sibgrouper** is meant for sorting a collection of sib-pairs (full and half) into Distinct-Parent-Groups, which are the raw ingredients for "sib-robust CKMR" (instead of individual samples). The workhorse is the eponymous function `sibgrouper`. You will need this iff you have within-cohort sib-rich samples, eg from larvae. If you don't need this, don't use it!

Some pairwise classifications may be wrong (eg full misclassed as half, or non-sib misclassed as half, or vice versas, or whatever). Sometimes but not always, the misclassifications will manifest as inconsistencies during `sibgrouper`, which will complain about them. If so, you will need to diagnose and fix the classifications manually before `sibgrouper` can succeed, and there are tools to help you do that: `familize`, `andij`, etc.

Distinct parent groups: A DPG comprises all offspring of one single (unknown) adult that appear in the sample. Each sample is a member of exactly two DPGs, one for its Mother and one for its Father (assuming no hermaphroditism... oh yes, BTW that's a pre-condition of all of this). That's true even for a pair of full-sibs who are not sibbed with anyone else; they are both the members of one Maternal DPG and of one Paternal DPG.

You can think of the DP(G) either as the linked set of offspring, or as the (unknown) parent that bred them; if the latter, then a "link" between any two DP(G)s means that they have one or more joint offspring.

Generally we do not know the "ernity" of any particular DPG, ie whether it's linked via shared maternity or shared paternity. You can figga that out with mtDNA haplotypes, and you'll have to in the end because ernity *does* matter for CKMR. `sibgrouper` does check for "ernal consistency" (ie that no sample is alleged to have either two mummies or two daddies), and will give the ernity of each DPG if the check succeeds. In principle, you should fix any ernity problems before proceeding to CKMR, but you can ask `sibgrouper` to return the DPGs anyway.

If you are a tightwad and really trust the results[*], then you could use the chains to decide the minimum reasonable set of samples to mtDNAtype. The ernities of linked DPGs must alternate (think about it... hard!) so in principle you don't need the mtDNA of *every* sib-pair to deduce the full set of ernities. But that "saving" might very well be much more trouble than it's worth.

Even if ernities are inconsistent— ie if there is ernal inconsistency :) — `sibgrouper` will show how the DPGs are linked into chains. Within each chain, you could get from any DPG to any other by following links.

[*] and sometimes I would fit that description myself

Genealogical pedantry: Yes, Half-Sib Pairs are indeed functionally identical PLOD-wise to Grandparent-Grandchild Pairs and to Full-Thiatic Pairs (eg aunt/nephew)– the more exact term would be "2nd order kin". However, if you're sibgrouping, you're not gonna be too worried about GGPs or FTPs, which is lucky cos they obviously don't have the same (in)consistency properties and wouldn't work with `sibgrouper`. (But, they can happen— and if so will likely be a source of inconsistency. Sometimes you can diagnose that.) So here I'll keep referring to "HSPs" rather than "2nd order kin", even though in other aspects of CKMR (e.g. cross-cohort sib-oriented comparisons) it's often better to do the opposite.

sibgrouper is easy when there are no inconsistencies. The more sibs you have, though, the more chance of inconsistencies. They need to be fixed *manually* (automatic is impossible), and fixing a lot of them is a *gruelling* miserable process. Here are some reasons why it's hard:

- long tails to PLOD distros (especially lower tails?)
- 3rd-order kin are a real thing
- correlated PLODs from FS to their HS
- occasional FTPs (lookalikes to HSPs, but completely different kinsequences)
- very occasional inbreeding?

INPUT AND WORKFLOW

The input is a dataframe of putative sib-pairs first produced by `kinference::find_HSPs` (qv) or equivalent, then augmented and tweaked by You. It must have two columns `i` and `j` referencing sample-pairs (either by number, or by some character code that you understand), a numeric column `PLOD`, and a mandatory character column `sibness` to show the classification of each pair. You will create `sibness` yourself, and you may then need to manually tweak some classifications later if sibgrouper finds inconsistencies. The dataframe is *not* supposed to contain *all* pairwise comparisons, just those that surely are or might just be sib-pairs.

There are 3 classifications allowed in `sibness`, which is case-insensitive:

- Full-sibs are `F` or `B` (for "Both parents shared")
- Half-sibs are `H` or `O` (for "One parent shared")
- Non-sibs are "X" or "" or " ".

Although sibgrouper ignores the non-sibs, it's perhaps useful to include them so that you can check whether they are misclassed, and fix them more easily.

As an example, you'd start with something from `kinference::find_HSPs` (qv), with a generous `keep.thresh` (so that you're pretty sure no HSPs will be left out), eg

```
allsibsplus <- kinference::find_HSPs( ..., keep_thresh=<blah>)
```

Have a look at a histo to see where the dips are that most cleanly separate HSPs from non-sibs, and also where you can probably set a safe threshold that avoids carrying a vast number of obvious non-sibs along. (*Don't* use these exact numbers— do I really need to say that?!) Then make *provisional* classification in a `sibness` column, eg:

```
siblist <- allsibsplus %where% (PLOD > -10))
isibord <- findInterval( siblist$PLOD, c( 20, 130) # 0, 1, or 2
siblist$sibness <- c( '', 'h', 'F')[ 1+isibord]
```

Now try sibgrouper. If it runs thru without warnings, then all your classifications are plausible, and you can go on to the more important task of using the DPGs (or one representative of each) in CK comparisons with **other** sets of samples, and analysing the outcomes via CKMR.

Oh dear inconsistencies: But you're still reading, so I guess it didn't just run thru :(

There are four levels of checks in sibgrouper. They are checked in order, and as soon as one level of check fails, sibgrouper returns with information on the failures. IE, it does not try to run the other checks, nor to produce the DPGs. Failing a check means that a specific attribute will be set, and there will be a warning. (Still **to do**: change these attributes into regular elements of the list, that can be accessed with `$`. There's no point in keeping the distinction.)

The first basic idea are that a pair of Full-sibs must have the same *ancestral* kinship as each other to any other sample (since they had the same mum and the same dad). The second idea is that Half-sibs must share either a mum, or a dad, but not both; so the half-sibs of any sample must fall into at most two sets, one linked via maternity and one paternity, and all those in each set must be half (or full) sib to each other. The third idea is that not only does each sample have exactly two parents, but it has one female one and one male one— not two mummies nor two daddies.

The checks are as follows, with the first two being concerned with Full-sib mutual consistency, the third with Half-sib consistency, and the fourth with

mummy-vs-daddy stuff.

1. For each Full-sibset of samples (ie where a chain of pairwise full-sibness connects every two members), all members should be Full-sib to each other. Attribute of failure is `finco`.

2. For each Full-sibset: any animal that is Half-sib to at least one member of the Full-sibset, should be Half-sib to all of them. Attribute of failure is `fhinco`.

Now we can reduce each Full-sibset to a single representative (since the Full-sibset members are now known to have identical kinships to any other sample). Those, plus all samples that don't have any Full-sibs, can now only be related to each other thru at most one parent; ie, they are Half-sibs or "unrelated". Since Full-sibs are out of the picture, the next check is:

3. For each remaining sample, do its Half-sibs fall into exactly two subsets, each subset being entirely Half-sib to itself and itself only? (One or both subsets can be empty.). Attribute of failure is `hinco`.

If those three checks all pass, it is possible to produce DPGs (and you can force `sibgrouper` to do so)— but they may not imply consistent sexes for the parents. The final check is:

4. Can the DPGs (two for each sample) be linked so that each sample has one DPG (and thus one parent) of each "ernity"?

Oh dear workflow: The rest of the workflow is about manually adjusting some entries in `sibness` until you get sensible, or at least consistent, results. It's always a good idea to keep track of "manual" adjustments to datasets, so let's first add a column for that:

```
siblist$manual <- FALSE
```

(Even if you don't do so, `resib` will add it for you automatically.)

The dataframes used in all this are pretty simple so you *can* do adjustments and inspections purely by hand— but the pairwise `i` and `j` (which are not necessarily in order) make it pretty tedious. It's easier to use the function `resib`, as shown below.

Anyway, you could start your adjustment process by using `kinference::split_FSPs_from_HSPs` (qv) to help improve decisions near the HSP/FSP "boundary". I won't show that here in detail, but say you thereby decided that the pair (1055, 2741) should actually be FSP not HSP even though their PLOD was below 120. Then you can do

```
siblist <- resib( siblist, 1055, 2741, 'F')
```

You can inspect what's happened via either via some ugly handwritten code:

```
siblist %where% (((i==1055) & (j==2741)) | (i==2741) & (j==1055)))
```

or more simply via `andij( siblist, c( 1055, 2741))`. Note that `andij` and its mates `orij` and `notij` can handle more than two samples, but here we only want to see a single pair.

Ernity failure: The first three checks are *relatively* easy to resolve because they are "local" to each sample, only involving one or two degrees of separation in kinspace and therefore not too many samples. However, ernity checks are global, and may involve hundreds or thousands of samples. They might be really hard to diagnose just from PLOD stuff...

By far the easiest and best way to check, is with mtDNA data for every sample. In principle you might be able get away with many fewer samples via the ernity-chains; but if ernity has failed, that's not gonna work, is it?! Of course, if the number of haplotypes is limited, you will still have to *think* slightly even with mtDNA data, but at least you should not need a quantum computer to figga it out.

Diagnosing ernal inconsistencies: There are two further tools which might help narrow down the search for ernal inconsistencies (it is a bit overwhelming to discover merely that there's an ernity problem somewhere in a linked set of 1013 larvae...). The first, `find_odd_short_loops`, finds the smallest set(s) within which there must be ernal inconsistency somewhere, within some set of samples; this works "bottom-up" to find shortest self-loops from a sample back to itself, and if that loop has an odd number of steps, then there's ernal inconsistency within it.

The second tool, `loopagroopa`, isn't coded (yet) but apparently I have worked out an algorithm and written in up on my Remarkable... It breaks up chains into singly-connected parts that cannot loop back to themselves; any ernity failure must occur only within such a part, not between them, so this can break the problem up in a "top-down" way. The algorithm is apparently a bit similar to `find_odd_short_loops`...

Mtdna and ernity: At some point you'll need mtDNA, and that should help a huge amount with diagnosing ernity woes. There's no code for that yet, but I guess I will add some when I've tried it for real. The first place to start is with estimating haplotype frequency (obviuosly) and then error rates based on full-sib groups, and then on half-sib groups that appear "likely" to be Mat-ernal (ie to possess Mat-ernity). Then I think it will be pairwise comps between linked DP(g)s, looking for links that appear to be between two Maternal or two Paternal DPGs.

Details

Dataframe that normally comes originally from `kinference::find_HSPs` (qv) but has been augmented.

Faq

- Can't I just call an `autofix` to sort out the inconsistencies?

– Ha! Young people these days... Absolutely no chance. Your data, your problem. AFAICS it's impossible to autofix anything except the ones that are so simple that you can do them manually.

- What advice would you give on working with sibby larval datasets?

– Try to avoid larvae if you can (but you can't always)

– Don't sample too many of the bloody things.

- I have ernity failure in a dataset of thousands of larvae. I don't want to wait for mtDNA (though I promise that the mtDNA data is coming). What should I do meanwhile?

– Well, my *guess* is that ernity failures are most likely thru samples that have just two "Half-sibs" of different ernity (ie not HS to each other)— but one of those "HSPs" is actually a misclassed UP (or weaker kin), so that two chains have been linked up that shouldn't be. Just get that mtDNA...

`familize` *Sibgrouper diagnostic aids*

Description

Helper functions for subsetting pairwise sib collections and visualizing potential "families" within it. `familize` is for visual display, the others are tools for looking at subsets.

`andij`, `orij`, and `notij` extract subsets, as per their names. `orij` returns all kin of `ij`, ie all pairs where at least one member is in `ij`; `andij` returns the sib-pattern of `ij`, ie only the pairs with both members in `ij`; `notij` returns all pairs *not* containing a member of `ij`.

`ijunique` returns a vector of unique samples, sorted (useful since many samples will appear multiple times in a pairwise collection).

`ijexpandF` produces an augmented version of `ij`, also containing any Full-sibs of `ij` that occur in `siblist`.

`familize` presents matrices showing all pairwise sib classifications and PLODs, for a subset of potential sib-pairs (i.e. for all samples mentioned in that subset). The subset is meant to be *one* set of "extended kin", so try to make it reasonably small! The aim is to help you spot mis-classifications (eg clear HSP called FSP, etc). It tries to improve visual clarity layout by re-ordering rows and columns, using clustering; treat the order as arbitrary. Basically if there's one full-sib group with two "wings" (one wing for each of the two kinds of half-sibs that members of that full-sib group can have) then the full-sib group should be a block in the middle, one wing should come before it, and one wing after it. Of course, you probably wouldn't be using `familize` unless there was a problem with the "family", and in that case clustering may struggle to make the order sensible. But you should *see* the results without it...

Usage

```
familize( siblist,
  want_numeric= FALSE,
  order= NULL,
  symbols= kin_symbols(),
  compact= FALSE,
  whatever_you_say_boss= FALSE)
andij( siblist, ij)
orij( siblist, ij)
notij( siblist, ij)
ijunique( siblist)
ijexpandF( siblist, ij)
```

Arguments

siblist	dataframe with columns i, j, PLOD, sibness; presumably once from kinference::find_HSPs, augmented by manual sibness. For familize, *don't* make this too big; it's meant to capture one group of "extended kin", and the ancient base-R clustering code it uses is uninterruptable and takes a long time for more than a few hundred entries...
ij	vector of sample references to look up in the i and j columns of siblist—usually numbers.
want_numeric	do ya? Prolly not. See **Value**.
order	normally NULL to let hclust try to deduce a nice layout. But you can force a particular order by setting it explicitly (eg to compare results from two versions of a dataset). Elements must exactly correspond to all in $i and $j of siblist, or the same preceded by a period "." (which is how they appear as dimnames in familize output).
symbols	symbols to use for "UP", HSP, FSP, and DUP (self) respectively; see kin_symbols. BTW I tried heavy-plus "\u2795" instead of "+" but R thinks it has width 2 (which it doesn't, at least in Consolas; ?nchar does say "approximation").
compact	if TRUE, give the kinmat component an S3 class "compacto", so it prints as densely as possible; see mvbutils::compacto for more info if you must. This only affects printing.
whatever_you_say_boss	R's built-in clustering code takes *forever* if there's a huge number of samples, locking up my computer impressively, and it can't be interrupted much with <ESC> or whatever. By default, familize therefore quits if you give it more than 100 samples (you wouldn't be able to decipher the output anyway). If you want to force it, set this argument to TRUE.

Value

andij and friends return a subset of siblist. ijunique returns a vector, presumably numeric (if siblist$i is). familize returns a list of two matrices: kinmat holds the sibness assignments from siblist, and PLODmat holds the PLOD values. These are both character objects, for optimal display purposes; if you want the actual values instead, set want_numeric=TRUE (whereupon kinmat will contain 0/1/2/3 for UP/HSP/FSP/Self).

See Also

sibgrouper, kin_symbols

Examples

```
# Not compulsory to have an EXAMPLES -- you can put examples into other sections.
# Here's how to make a "don't run" example:
## Not run:
reformat.my.hard.drive()

## End(Not run)
```

`find_odd_short_loops` *Diagnostics for ernity failure*

Description

Ernity failures are harder to narrow down than "local" DPG problems with HSPs and FSPs. If overall ernity fails inside `sibgrouper`, then it runs `find_odd_short_loops` to find the *smallest* sets of DPGs within which ernity has definitely gone wrong, and returns that in the `$erninco` element. It doesn't return *all* such sets because longer ones might overlap with shorter ones, and thus be artefacts of simpler problems. Note that the DPGs appearing in an odd-length loop *cannot* all be correct; something needs fixing!

You can plot the alleged connections within a loop via eg `plot_oddloop( <X>, <X>$erninco[[1]])`. This requires that you have the **igraph** package installed; it's not compulsory for `sibgrouper`, so you'll have to install it manually.

You can also run `find_odd_short_loops` and then `plot_oddloop` separately from `sibgrouper`. The former expects either the entire output from `sibgrouper` (which must have at least gotten past all `finco/fhinco/hinco` checks) or just the `$shadow_dpg` element, which shows which other DPGs each sample in a DPG is connected to.

Ultimately you're probably gonna need mtDNA to sort out `ernity`— and of course you will need mtDNA for *some* samples to establish *absolute* ernity of some DPGs, rather than just the relative ernities between DPGs. (The rest could then *perhaps* be filled in using relative ernities.) And NB that passing `erninco` checks is a necessary but not sufficient condition for correct DPGs; mtDNA may yet contradict even an apparently-successful ernity resolution.

Usage

```
# plot_oddloop( <X>, <X>$erninco) is usual
find_odd_short_loops( sdpg)
plot_oddloop( sdpg, oddloop, cex= 2, color= 'LightGrey', border= NA, ...)
```

Arguments

`sdpg`	`sibgrouper` output, or its `$shadow_dpg` component.
`oddloop`	result of a call to `find_odd_short_loops`, or the `$erninco` component of a failed `sibgrouper` run (which would have come from a call to `find_odd_short_loops`).
`cex`, `color`, `border`, `...`	passed to the `plot` method for the `igraph::make_undirected_graph` object, as `vertex.label.cex`, `vertex.color`, `vertex.frame.color`, and `...` respectively.

Value

`find_odd_short_loops` returns a list of odd-length loops. They will all have the same length, which is the *shortest* encountered. Thus, there may be longer odd-length loops too, but they aren't returned; this is deliberate, since they might just be symptoms of overlap/confusion/whatnot caused by the shorter ones. Fix the simplest ones before proceeding!

See Also

sibgrouper

Examples

```
# Not compulsory to have an EXAMPLES -- you can put examples into other sections.
# Here's how to make a "don't run" example:
## Not run:
reformat.my.hard.drive()

## End(Not run)
```

---

first_fhinco *Maybe help untangle sib mess*

Description

If sibgrouper complains at the fhinco or hinco stage, you can use this to pull out all fhinco/hincos which overlap with the first one— cos you might wanna fix all associated miscalls together. However, I've recently been avoiding it cos it does lead to more complicated (albeit more complete) putative family-clusters; instead I've preferred to stick with X@fhinco[[1]] and X@hinco[[1]] (after library(atease)). You can always expand the family set

As a reminder, fhinco looks at each Full-Sib Group (fgroup), and checks that its members have identical HS patterns. hinco (the chequerboard version, anyway) looks at boiled-down set after each fgroup have been replaced by one representative, and checks that the two half-sibsets for each sample (there should be exactly two) are each internally complete (ie all HSP to each other) and not overlapping at all.

Usage

```
first_fhinco( X, which=c( 'fhinco', 'hinco'))
first_hinco( X)
```

Arguments

X	result of a call to sibgrouper, presumbably one that warned about "Inconsistent H's for some F-groups".
which	leave it alone...

Value

Numeric vector of indices to check via familize, andij, etc.

Example

```
## Don't run
# At least not til there's some data
X <- sibgrouper( pairset)
# warning about fhinco or hinco
familize( pairset, X@hinco[[1]])
familize( pairset, first_hinco( X)) # similar but maybe bigger
## End Don't run
```

`kin_symbols` *Display of pairwise kinship matrices*

Description

Controls what symbols to use in `familize` for the four kinships UP, HSP, FSP, and DUP (self). It's hard to past " " and "+" for the first two. For FSPs and DUPs, the defaults (available via `default_kin_symbols`) use UTF8 characters and work very well visually IMO. But they are non-ASCII and RCMD CHECK (up to at least R 4.6) moans about their use in

the demo file of this package. And I suppose there might be other situations where the defaults aren't nice— sigh. So you can change them for all future calls to `familize` by calling `kin_symbols`.

Usage

```
kin_symbols( syms = NULL,
    UP = ksenv$kinsymbols["UP"], HSP = ksenv$kinsymbols["HSP"],
    FSP = ksenv$kinsymbols["FSP"], DUP = ksenv$kinsymbols["DUP"])
default_kin_symbols()
```

Arguments

`syms`	optional named character vector with at most four entries, each a length-1 string (according to `nchar`). Names should be a subset of "UP", "HSP", "FSP", "DUP".
`UP`, `HSP`, `FSP`, `DUP`	optional names of individual elements to change to length-1 strings. Ignored if `syms` is set. Don't try to understand the defaults; the point is, they default to whatever's already been set.

Value

The old value of `kin_symbols`.

See Also

familize

Examples

```
default_kin_symbols() # has non-UTF8 characters so I can't show output here FFS :/
ks <- kin_symbols() # shows current values. See previous comment
kin_symbols( c( UP=' ', HSP='+', FSP='#', DUP='o'))
# Same as
kin_symbols( FSP='#', DUP='o')
```

---

resib *Manually change classification of some sib-pairs*

Description

That's it, really. `resib` also sets the `$manual` entries to TRUE.

Usage

```
resib( siblist, I, J, sibness, get_from=NULL)
```

Arguments

`siblist`, `I`, `J`, `sibness`	see `package?sibgrouper` or `andij`. Will be recycled if necessary. As per package doc, `sibness` should be one of the letters "BFHOX " (upper or lower case; NB the space as one option for "unrelated") or the empty string, but this (deliberately) isnt' checked.
`get_from`	non-NULL means to look up any unfound I/J pairs in the (presumably larger) sib-pair dataframe `get_from`, and add them. Occasionally needed if you overtightened the selection for `siblist` and left out some "severe lower tail" 2nd-order pairs; this way, you can copy their actual PLODs back into your working set.

Value

A new dataframe with modified rows (and *occasionally* added ones, if `get_from` is needed).

Examples

```
# See demo
```

`sibgrouper` *Sibgroup ernity*

Description

Sift full- and half-sibs into maternal/paternal Distinct-Parent-Groups *without* mtDNA. Each DPG comprises all sampled offspring of one particular adult. You won't know the "ernity" of each half-sibgroup, i.e. whether it's via mat-ernity or pat-ernity, but it's definitely all-one or all-the-other. Also, links the half-sibgroups into "chains" s.t. for any sample in a chain, it is linked by some sequence of half-sib relationships to all other samples in that chain, and not to any outside that chain.

One practical outcome is that, if your data are really sib-dense, then you can reduce the amount of mtDNA genotyping needed (if you are rather brave). In principle, you only need to check mtDNA from 2 individuals within each half-sib-group, because if their mtDNA is the same then that whole group is Mat-Ernal, or Pat-Ernal if not (assuming highly variable mtDNA). But in fact you can do better: you only need check 2 animals from each *chain*, making sure they are from the same half-sibgroup. In practice, though, once you need to mtDNA-genotype more than say 10% of your larvae anyway, it's probably simpler to do *all* of them; this will also save you trouble (years) later, when you find XHSPs and need to establish *their* Ernity.

Full-sibs must of course have identical kinship patterns. So the first step is to boil down all full-sib-groups into a single Representative, and only that Rep is used in determining the half-sib-groups. (The full-groups are returned as attributes, so you can decode the output).

The algorithm assumes that the input pairwise sibship data is completely accurate— no false-neg and no false-pos. However, that *won't* be true when there are 100s or 1000s of pairs. `sibgrouper` includes a series of consistency checks (which means that the real algorithm is more complicated than shown here), and reports any failures so that you can go back and adjust the inputs until things work. There are other functions in the **sibgrouper** package to help you with that; see `package?sibgrouper`.

The basic algorithm is quite simple, *if* all checks pass (except perhaps ernity, which is the last one applied and which operates on DPGs rather than individuals):

- Reduce each full-sib-group to a single Representative, and only compare Representatives thereafter
- For each animal Alex with any *half*-sibs:

– Colour Alex's first half-sib, Betty, Blue

– For each subsequent half-sib of Alex:

— if it is a half-sib of Betty, then colour it Blue, else colour it Red

– Add Alex to both the Blue and Red groups

– The Blues and the Reds are the two half-sib-groups of Alex (there may be no Reds other than Alex)

- Once you have all the half-sibgroups, build the chains by connecting each half-sibgroup to any other half-sibgroups that contain any of its members, and repeating until that chain stops.

NB this only works for non-hermaphrodite species. I think the chains are just the "equivalence classes" of any single sample in that chain, where "equivalence" is defined by "is linked by half-sib steps" and not all equivalences are directly noted in the data (which only shows "direct" HSPs).

The return value is a list with many wonderful things; read the rest of this for details.

Usage

```
sibgrouper(
  pairs,
  sibless= NULL,
  nsibless= length( sibless),
  want= c( "all", "fgroup", "ndpg"),
  dpg_format= c( "individual", "string", "both"),
  overlook_ernity= FALSE,
  fhinco_count_type= c( "tight", "loose")
)
```

Arguments

`pairs`	dataframe with columns `i`, `j`, `sibness`, and perhaps more; see `package?sibgrouper`. You can use character `i` & `j` for clarity to tie back to specific samples, though numbers ("sample rows" in some other dataset) are probably the default.
`sibless`, `nsibless`	if you want it to tabulate the DPGs, you need to tell it how many samples have no sibs at all— hence `nsibless`. Each will contribute 2 DPGs of size 1. If you actually want to generate DPGs that include the labels/numbers of those specific sibless ("only-children") individuals, then supply those as a vector via `sibless`; they should be the same mode as `pairs$i`.
`dpg_format`	Hmmm, not sure if this is used...
`want`	The default of "all" returns all information about DPGs (see **Value**), but *only* if all steps run thru without inconsistencies. "fgroup" will return immediately after the `finco` (full-sib-only) check, with one component called `fgroup` if the check succeeded, or `finco` if it failed. Otherwise, if [almost] all checks pass (see next arg) then `want="ndpg"` will return the tabulation of DPG sizes, which is succinct and the first thing you wanna look at really.
`overlook_ernity`	Provided that the `finco`, `fhinco`, and `hinco` checks all pass, it is possible to produce DPGs. Those DPGs may however imply inconsistent ernities, in which case `sibgrouper` will normally abort with a warning and an `erninco` attribute (see also `find_odd_short_loops`). You can force it to produce the DPGs anyway by setting `overlook_ernity=TRUE`; it still does the checks, and you'll still get the warning, but it will proceed to the end.
`fhinco_count_type`	How many comparisons "count" during the `fhinco` check? For a given Full-sib group, `tight` means only against samples that are Half-sib to at least one member of the Fgroup, whereas `loose` means every single comparison against another sample. You could argue for ages about which one is best. Let's not.

Value

A list with potentially lots of things. You can get rid of most of them by setting `want="ndpg"` or even `want="fgroup"` (but the latter was really just meant for testing). If any check fails, there'll be one element named `<X>inco` where `<X>` is one of `f`, `h`, `fh`, or `ern`– see **Inconsistencies**. And that'll be almost all you get, except perhaps with ernity checks); the other return-values don't make sense if the sibships are internally inconsistent, and as soon as one check fails, there's no meaningful way to proceed to the next. If all checks pass, or if the only failure is ernity (so there is an `erninco` element) but you have set `overlook_ernity=TRUE`, then there will be these elements, which are *really* what you want:

`ndpg`	tabulation of number of DPGs of each size. See also `nsibless` arg. Ernity is not used (wouldn't make sense to). Sneaky tip:the clearest way to see the actual sizes, is `matrix(<result>$ndpg,1)`.
`fgroup`	List of all the full-sib-groups; each element consists only of a bunch of mutual full-sibs.
`dpg`	List of the DPGs; each element contains the sample IDs in that DPG, including full-sibs.
`shadow_dpg`	List with the same structure as `dpg`, but each element of each element now shows the *other* DPG that the sample-in-question is in. You could reconstruct this from `$dpg` and `$samp_dpg` but this saves you effort— noice!
`ernity`	A vector of positive and negative integers, one element per DPG. Any two DPGs with the same value have the same ernity; any two with the same *absolute* value but opposite signs have the opposing ernity.
`chainid`	DPGs can be grouped into chains connected by "degrees of sibaration"; which chain is each DPG in? No ernity checks are made for `chainid`; however, if the ernity check succeeds, the chains should correspond to the absolute values of `$ernity`.
`samp_dpg`	2-col matrix of DPG names, one row per sample. Full-sibs are included, not just the Rep of each `fgroup`.
`check_count`	how many comparisons were made in each check. This is the least important thing really, but it does let you report what proportion of tweaks were necessary (and thus infer roughly how many undetectable errors there might still be). There are two ways you could count the comps for `fhinco`; see arg `fhinco_count_type`.

Most of these are long and their contents are not interesting except as grist to the data-mill; try `want="ndpg"` for a digestible summary (but you'll need more than just that to actually do supersibby CKMR).

Inconsistencies: Inconsistencies are currently listed (sort of) as attributes `finco`, `fhinco`, `hinco`, or `erninco`. Only one of these can be non-NULL, since `sibgrouper` aborts after the first inco failure. I will eventually *change* things so that these are regular list elements, but for now, just do `library(atease)` and then you can work easily via eg `myresult@finco`. You can use `familize` and other tools to help diagnose the underlying problems, which you can adjust with `resib`; see the demo too (or vignette, if I've made one— but the demo is pretty comprehensive).

`finco` is a list of 2-col matrices, each row showing a pair who ought to be FSP in terms of their other alleged FS. Each list element refers to one "fgroup" (chain of samples linked thru FSPs).

`fhinco` is (I think) a list of vectors of alleged full-and-half-sibs that are fhinconsistent. Check the demo.

`hinco` is similar.

`erninco` is a list of the shortest odd-length loops between DPGs; a true set of DPGs will not contain any odd-length loops (think about it... hard!). There might also be *longer* odd-length loops present, but because loops can overlap and you'll have to fix them up one-at-a-time anyway, the longer ones aren't reliable (and thus not worth reporting) until the shorter ones get fixed. You can plot any one such loop via `plot_oddloop` (qv).

Examples

```
## sim_sibgrouper call should go here...
```

Index