Package: kinference (via r-universe)

October 22, 2026

Title Pairwise kin-finding from genotypes

Description Kin-finding and QC tools for Close-Kin Mark-Recapture

License GPL (>=3)

Depends R (>= 4.3.0), gbasics (>= 1.2)

Imports Rcpp (>= 0.12.11), atease, mvbutils (>= 2.12), vecless

Suggests knitr, rmarkdown, tinytest

VignetteBuilder knitr, rmarkdown

LinkingTo Rcpp, RcppArmadillo

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

KeepPlaintextDoco YES

NeedsCompilation YES

Encoding UTF-8

KeepSource TRUE

Collate kinference.R Cloaders_kinference.R

LazyData yes

LazyDataCompression xz

Version 1.2.48

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

Date/Publication 2026-06-30 03:47:22 UTC

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

RemoteRef HEAD

RemoteSha 6a440c1dbd708903876025aca3b4174ab94d7a96

Contents

kinference-package 2
autopick_threshold 5
bluefin 8

chain_pairwise 8
check6and4 9
drop_dups_pairwise_equiv 11
dropbears 12
est_ALF_6way 13
est_ALF_ABCO 14
est_ALF_ABO_quick 15
find_duplicates 17
find_dups_with_missing 24
hetzminoo_fancy 25
histoPLOD 26
ilglk_geno 28
kin_power 29
kinPalette 30
prepare_PLOD_SPA 31
split_FSPs_from_HSPs 32
upairid 34
var_PLOD_kin 35

Index **38**

`kinference-package` *Kin-finding and QC for Close-Kin Mark-Recapture*

Description

`kinference` covers the preparatory steps for Close-Kin Mark-Recapture, for a multilocus SNP dataset from many individuals that have already been genotyped. Specifically, it covers:

allele frequency estimation, followed by...

... QC of samples and loci, with the goal of then being able to...

... find close-kin pairs and duplicated samples.

These tasks break down into finer categories, and the various functions under each are mentioned under **See also** below. There is a vignette, `kinference-vignette`, which demonstrates many but not all features. For detailed explanations of the algorithms, see the manuscript under **References**.

The kin-finding process entails several steps, each of which needs to be examined by a human being to make sure it has worked properly, before moving to the next. Sometimes one has to go back to an earlier step, eg to tighten/loosen quality control. The whole process should not be treated as automatic, and there is deliberately no `kinference::shut_up_and_find_the_pairs()` function! For the same reason, most of the functions do not have default parameter values, except perhaps for visual display; you are expected to think about appropriate choices for your dataset.

Note that `kinference` does not build or fit CKMR models per se. It "just" deals with the key preparatory step of finding kin-pairs, but the model per se is up to you!

Genetic inputs: kinference handles diploid biallelic SNP genotypes for thousands of loci. Genotyping errors are tolerated, but error rate should be low: no "3X coverage" etc! Loci are assumed to be in HWE, which can be checked. Genuine null alleles (see below) are allowed for, but uncalled/unknown genotypes are *not* permitted in most functions; every sample and locus must be called, even if some calls are wrong. The types of close-kin considered are POP, FSP, and 2KPs (2nd-order kin, ie HSP, GGP, FTP), which is the limit of resolution in the absence of genome-assembly data (outside the scope of kinference version 1).

kinference does *not* call/score the genotypes- you need to do that beforehand. The starting point for kinference must always be a snpgeno object (see package **gbasics**) containing already-called genotypes for *every* sample and locus, plus the crucial sample-specific information ("metadata" to geneticists, "covariates" to population dynamicists and statisticians!) such as sampling-year, age, sex, etc depending on the dataset. The original object gets augmented with extra data (eg allele and genotype frequency estimates) as the steps proceed.

There's no requirement for one particular genotyping method (kinference has been used successfully with sequencer data and with microarrays) but there *are* limitations and expectations on the properties of the data. Some of these are mentioned here, and you may find more information in kinference-vignette (qv).

Missing genotypes: You basically can't have any. Almost all functions in kinference expect a definite (albeit possibly wrong) genotype for every sample at every locus. Sometimes the call might be that neither the reference nor the alternative SNP are present; that's still a definite call, but it's treated as a double-null (see below) not as a "missing". The two exceptions are find_dups_with_missing (qv) and est_ALF_nonulls. The former is not intended for use with CKMR data, and the latter is a first step that might be useful if you need to "impute" (ie make up) missing genotypes.

The rationale for insisting on non-missingness, is that many of the statistics calculated by kinference have null distributions (i.e., under some null hypothesis about the true situation) that can be readily calculated *iff* all samples are scored at all loci. Knowing the null distribution has been invaluable to us on many occasions, for seeing whether things have gone badly wrong. Trying to allow for missingness would put the calculations somewhere on the scale between unreliable, horribly complicated, or impossible.

Genotyping errors: The functions in kinference can tolerate some level of genotyping error. However, if your data has high levels of allelic dropout due to low coverage, then it's not suitable for kinference. And samples with dodgy or contaminated DNA won't work either; hopefully the QC routines will weed them out.

Null alleles: Some genotyping methods, when applied to some species, produce a substantial proportion of null alleles. "Null" here means a *heritable* and *repeatable* but undetectable allele, eg due to mutations at restriction site or indels nearby. It specifically *excludes* dropout due to low coverage, which is neither repeatable nor heritable.

Note that a sample might well have a *single* copy of a null allele, in which case it will "look like" a homozygote for the other (non-null) allele. Only if a locus has a high null-allele frequency, will there be many double-nulls (i.e. where neither the "major/reference" nor the "minor/alternate" SNP is present). Thus, the evidence of nulls is not mostly from entirely-missing double-null calls, but rather from inflated "homozygote" proportions, and (if POPs are present) from apparent Mendelian exclusions within obvious POPs. The interpretation and handling of nulls is very important for kin-finding (perhaps less so for other genetic applications), so you really have to understand your genotyping method: some software is reluctant to call a genotype unless the

result is extremely clear, leading to "missing" genotypes that are *not* the same as double-nulls! In that case, force your software to damn well make the damn calls...

Of course, `kinference` can perfectly well cope with datasets *without* nulls, too, as the vignette shows. Deciding what to do about nulls requires you to understand the properties of your genotyping method (software pipeline as well as technological underpinnings). There are several options for handling nulls in `kinference`, determined by how you "encode" your genotypes; see `gbasics::snpgeno`.

Linkage and linkage disequilibrium: `kinference` is designed for datasets of a few thousand SNPs, where Linkage Disequilibrium (LD) is unlikely to be substantial; almost inevitably, a few loci *will* be in LD just because they happen to be close together genomically, but that won't noticeably affect the results. If you have zillions of SNPs, eg from Whole-Genome-Sequencing (WGS), then LD *will* be a problem and you won't be able to get `kinference` to work properly unless you thin your dataset a lot. However, a more significant problem with WGS is likely to be low coverage, leading to unacceptable levels of dropout...

Given its intended use, `kinference` quite reasonably ignores LD. However, *linkage* per se- the fact that long consecutive stretches from the same strand of DNA are inherited at meiosis, rather than each locus being inherited independently- is absolutely *not* ignorable for finding 2KPs (or FSPs). `kinference` has quite a bit of code that deals practically with linkage, as inferred from pairs that clearly are 2KP.

References

Bravington, Mark and Baylis, Shane and Eveson, Paige and Feutry, Pierre (2026): "kinference: Pairwise kinship detection for Close-Kin Mark-Recapture", bioRxiv (being uploaded now: DOI to come).

Historical note: `kinference` has evolved considerably since its birth at CSIRO in ~2018. It was originally developed for ddRAD datasets back in the "Bronze Age", and a couple of those datasets are still in practical use within CSIRO. Thus, `kinference` has to support some legacy types of data that would best be avoided in new projects- most notably, the challenging (albeit successful) "6-way genotyping" that tried to explicitly distinguish single-nulls from homozygotes, based on read-depth. That, and a few other quirky cases, lead to some pretty complicated code in parts of `kinference`, and to several options (especially for null alleles) which might be confusing for new users who "just want to analyse their data". Such is life- sorry!

See Also

`vignette("kinference")` (qv).

And for the categories of task, and the functions within each:

Locus qc: `check6and4`

Sample qc: `ilglk_geno`, `hetzminoo_fancy`, `find_duplicates`, `find_dups_with_missing`, `drop_dups_pairwise_equiv` (and, after kin-finding, `chain_pairwise`).

Allele frequency estimation: `est_ALF_ABO_quick`, `est_ALF_nonulls`, or less likely `est_ALF_6way`, `est_ALF_ABCO`, `re_est_ALF`

Pairwise kin finding: find_POPs and find_POPs_lglk; find_HSPs; histoPLOD and 'k inPalette' for graphics; split_FSPs_from_HSPs and other split_x_from_y functions; autopick_threshold and var_PLOD_kin for separating 2KP from more-distant kin. Also chain_pairwise and get_chain for post-hoc consistency checks on likely HSPs.

Predicting locus power for kin finding: kin_power (qv)

Example datasets: dropbears, bluefin

---

autopick_threshold *PLOD threshold for HSPs*

Description

This function proposes a PLOD threshold for excluding almost all 3KPs and 4KPs from true HSPs, and computes the associated false-negative probability (i.e., that a true HSP will have a PLOD below that threshold).

The modus operandi is to fit a mixture distribution to observed PLODs within some limited range that is expected to contain only 2KPs, 3KPs, and *perhaps* a few 4KPs (and specifically excluding 1KPs). It is implicitly assumed that there *is* a clear 2KP bump, and that it peaks close to the theoretical mean; if not, there is no point proceeding until you find more 2KPs (and perhaps tackle any QC issues). The threshold is then "chosen" (or proposed; the actual choice is really up to you) so that the expected number of false-positives from 3KPs plus 4KPs (i.e., with PLODs above the threshold) matches whatever you decide. A histogram with expected values is plotted (unless you tell it not to).

The mixture distribution treats the expected PLOD values for true 2KPs, 3KPs, and (if used) 4KPs as known (since those expectations can be calculated just from allele frequencies). However, because of linkage, the corresponding variances cannot be predicted a priori. Instead, the variance of the 2KP bump is estimated empirically from those PLODs which are above the 2KP mean, on the assumption that there will be few if any 3KPs with PLODs that large (hence the need for an upper limit on the included PLODs). Then, var_PLOD_kin (qv) is used to infer the extent of linkage and to predict what the 3KP and 4KP variances might be, under two extreme assumptions about the detailed nature of linkage. The two assumptions lead to upper and lower bounds on the variances of 3KPs and 4KPs. Next, a Normal mixture distribution is fitted by ML, to estimate the height and variance of the 3KP bump, subject to those bounds; it can optionally allow for the presence of 4KPs as well. Once those parameters have been estimated, autopick_threshold calculates the threshold PLOD above which FPtol_pairs of 3KPs (and perhaps 4KPs) would be expected. Finally, it calculates the proportion of 2KP PLODs that would be expected to fall below that threshold.

Since it is better to make the threshold too high rather than too low, you can use the selecto option to choose how the 3KP variance is estimated, subject to its bounds. The options are (i) strict ML estimation, to give the best fit to the overall PLODs, or (ii) choosing whichever 3KP variance leads to the *highest* threshold (but still fitting the 3KP bump height by ML conditional on that variance).

The calculations in var_PLOD_kin are in fact geared specifically to HSPs, HTPs, and HC1Ps, as opposed to the other subtypes of 2/3/4KPs. Accordingly, it is a good idea to restrict the pairs to those which, if the are indeed 2KPs, are a priori likely to be HSP rather than GGP or HTP. (If they are not 2KPs, then it probably matters less exactly what subtype they are.) Having said that,

the sky will *probably* not fall in if some GGPs are also included; in some circumstances it might even be worth doing so deliberately, in order to increase the 2KP sample size. But the general principle of kin-finding applies: it is best to concentrate on the comparisons that yield the outcomes you're interested in, rather than including lots of extraneous comparisons and then trying to filter the results.

Despite the name `auto...`, *you* still have to supply sensible values for a couple of parameters, based on looking at your data and understanding what you are trying to do. So it's not *completely* automated process, and it never will be! Choosing a threshold is *not* an "optimization process" with explicit bias/variance tradeoffs; rather, it's about ensuring that you have adequate "engineering tolerance" in the next stage of CKMR. The False-Negative Probability will, to a great extent, compensate for the choice of threshold (i.e. removing any bias in the fitted CKMR model) *unless* you set the threshold too low, and thus end up with some 3KPs in your set of "definite 2KPs".

Fitrange and use4th: Assuming that you are using a 2KP-oriented PLOD, then the range of PLODs you fit to should extend from somewhere above 0 (which is very close to the expected PLOD for 3KPs), up to the RHS of the 2KP bump, but not so large as to include any FSPs and POPs. You can use the `fitrange_PLOD` parameter to control this, instead of subsetting the `kin` argument manually. If you set the lower limit high enough, then you don't need to worry about 4KPs intruding (their contribution would be negligible), so you can get away with fitting a 2-component mixture (simpler, less to go wrong...) by setting `use4th=FALSE`. But if you push the lower range closer to 0 (which does give you a larger sample size for fitting), then you might need to set `use4th=TRUE`. The (substantial) downside of doing that, is that there are often more PLODs close to 0 than near the 2KP mean, so the mixture-fit (which has to make more assumptions when also using 4KPs, and those assumptions may not be perfect) will concentrate its efforts on getting a good fit near 0, rather than near the 2KP mean which is what really matters. It is worth experimenting.

Usage

```
autopick_threshold(
  x,
  kin,
  fitrange_PLOD,
  FPtol_pairs,
  use4th,
  selecto = c("ML", "paranoid"),
  NVAR = 10,
  plot_bins = NULL,
  shading_density = 10,
  want_all_results = FALSE,
  ...
)
```

Arguments

`x`	a `snpgeno` or its `locinfo` attribute. Must already have been prepared by running `kin_power`.
`kin`	a dataframe of "close-ish" kin-pairs and their PLODs, presumably from running `find_HSPs`; must have a column "PLOD".

`fitrange_PLOD` two numbers, specifying the range of PLODs from `kin` to use in *fitting* (though all are *plotted*, by default)

`FPtol_pairs` how many expected False-Positive 3KPs (and 4KPs, if `use4th=TRUE`) should the threshold exclude?

`use4th` whether to allow for 4KPs when fitting.

`selecto` whether to choose the threshold based on the best mixture fit ("ML"), or the most conservative ("paranoid").

`NVAR` how many variances to try, between the limits set by `var_PLOD_kin`

`plot_bins` bin-width for histogram plotting. Default NULL means no plot.

`shading_density`
By default, all PLODs in `kin` will be included in the histogram, even though only a subset are used in fitting. The histogram bars *not* used in fitting (i.e., below `fitrange_PLOD[1]`) will be lightened in colour, according to this parameter. Results are graphics-device-dependent, so you may need to experiment away from the default; larger numbers usually mean lighter shading. Setting `shading_density=NA` should result in a light transparent rectangle covering the entire LHS of the graph, which you might prefer. You can also set `xlim` as usual, to remove those left-hand bars altogether.

`want_all_results`
if TRUE, return dataframe(s) containing results for each variance explored. This lets you examine "sensitivity".

`...` other parameters passed to `hist`, eg `xlim`, `ylim`, `col`. Many others will be ignored, and some will cause problems.

Value

The proposed threshold, with lots of attributes. You can use those to calculate false-neg probabilities for *other* possible thresholds, as per **Examples**; you *don't* have to accept the one that is proposed here! Threshold choice is *up to you* (and badly-chosen thresholds are not the fault of `kinference`)!

See Also

`kin_power`, `var_PLOD_kin`

Examples

```
library( atease) # for convenient 'thresh@info' below
dropbears1 <- kin_power( dropbears, k = 0.5)
hsps <- find_HSPs( dropbears1, keep_thresh = -10)
histoPLOD( hsps, log=TRUE) ## observe HSP - POP gap centred ~ PLOD = 120.
## Use that air-gap to set fitrange_PLOD.
thresh <- autopick_threshold( x= dropbears1, kin= hsps,
  fitrange_PLOD= c(0, 120), FPtol_pairs= 1, use4th= TRUE, plot_bins= 5)
thresh ## the 2nd order / 3rd order threshold value
thresh@info["Pr_FNeg"] ## ...
## ... the estimated 2KP false-neg rate, given that threshold
```

bluefin *Southern Bluefin Tuna data*

Description

An anonymised snpgeno dataset of Southern Bluefin Tuna (SBT) genotypes for 1038 individuals at 1510 loci. This is a small anonymised subset of the data described in Farley et al. 2024 (full reference below).

This number of loci would be too small for finding HSPs if we only had biallelic loci with no nulls. However, SBT genotyping is "6-way"; that is, true nulls are present (in fact very common), and the genotyping process attempts to distinguish single-nulls from homozygotes based on read-depth (which is pretty high, typically 50-100 per copy). It does so pretty well but not perfectly, so there is an estimated error rate between single-nulls and homozygotes which is allowed for in the kin-finding calculations. The separation of single-nulls from homozygotes adds considerable statistical power, though also a lot of statistical pain. Calling of genotypes was done by in-house CSIRO software.

Note that this genotyping system is "legacy"; CSIRO is unlikely to use it in future projects, since economics (and statistical simplicity) now favours more loci at lower read-depth, and technological changes have reduced the prevalence of true nulls. However, it is still being used successfully for ongoing SBT kin-finding and CKMR.

Usage

```
bluefin
```

Format

An object of class gbasics::snpgeno()

Reference

Farley J, Eveson P, Gunasekera R. 2024. Update on the SBT close-kin tissue sampling, processing and kin finding 2024. CCSBT-ESC/2409/09.

https://www.ccsbt.org/system/files/2025-07/ESC30_08_CCSBT_CKMR.pdf

chain_pairwise *Sib-groups within HSPs*

Description

For checking veracity of *potential* half-sibs or other kin-pairs. `chain_pairwise` organizes them into chains within which each sample can be linked to another by a succession of direct pairwise links. The general idea is that real HSPs will be in clusters of 2 or 3; a spurious sample with a "lucky" genotype that wants to be everybody's mate will appear in a big but incomplete chain of mostly false-positives, where the direct pairwise links between the other chain-members are weak. You'd only run `chain_pairwise` for pairs with a PLOD (or whatever statistic is being used) within a particular suspect range, so each chain may have false-negatives (i.e. missing direct links), but the general idea should be clear.

`chain_pairwise` and `get_chain` are diagnostic tools for when `kinference` goes wrong. For a comprehensive approach to complicated links between *genuine* sib-groups (full and half), e.g. within larval samples such as for Atlantic Bluefin Tuna, see the **sibgrouper** package (if not on R-universe, then contact the `kinference` authors).

Usage

```
chain_pairwise( thing)
get_chain( thing, seed)
```

Arguments

`thing`	output from `find_HSPs` or `find_POPs` etc, or some subset thereof
`seed`	one sample ID, interpreted as a row-number in `thing`. To do:also allow names, via `info` attr.

Details

`get_chain` finds the chain for one specific sample.

Value

`chain_pairwise` returns a list of matrices, each for one chain; the rows and columns of each matrix are the samples in that chain. A "+" in the matrix indicates that those two samples have a direct pairwise link (i.e., they appear together in one row of `thing`); a "." means not. The rows and columns of each matrix are sorted so that the linkiest samples are on the bottom and right. `get_chain` returns the row-subset of `thing` that is chained to `seed`.

`check6and4` *Locus QC check*

Description

Compares 6-way (if available) and 4-way genotype counts to HWE expectations, based on already-estimated allele frequencies. Plots a histogram of p-values across all loci, and (more importantly) plots observed / expected counts for each genotype by locus.

An overall p-value is calculated for each locus, based on chisq(1) of the G-statistic; it is not obvious how many DoF should be used. Anyway, the p-value itself is just indicative; few if any CKMR datasets actually "fit properly" because sample sizes are so large that even trivial misfits lead to significant departures from HWE null hypothesis, even though the final outcomes of kin-finding (ie PLODdograms) can look perfectly good. This is usually obviuos from the initial histogram of p-values, which in theory should be uniform for the "good" loci; in practice, it usually has heavy left-skew. Thus, the p-values are not to be taken literally, but just as a relative guide; the worst loci have the smallest p-values, and you can use the p-value as a criterion for dropping some loci. Since QC is iterative (samples, loci, samples, loci, ...) you can always revisit the decision later.

The p-value threshold (`thresh_pchisq_6and4`) simply determines the colour used to plot each locus. You can manually changing the threshold and re-run until you have a visually satisfactory pattern of orange vs green. Green means good (or good enough); orange means bad, ie below the lower threshold. (You are actually allowed to supply two numbers for the threshold, in which case loci with in-between values will display pink; that is probably a design flaw, because it's confusing to have more than one threshold). Once you have identified a threshold that *looks good*, then you can use the `pval4` or `pval6` return-value to keep or discard loci, depending whether they are above or below that threshold.

The plots never look absolutely perfect, and there is no absolute criterion for "how bad is too bad". So, judgement and experience are required. But remember that keep-or-drop decisions aren't final; the whole QC process can be somewhat iterative, and the ultimate test is the PLODdogram(s) at the end. If it looks bad, ie bumps in the wrong places, then you may not have been restrictive enough (ie your threshold might be too high); whereas if the bumps are in the right places but are too wide for kin-finding, then your threshold might be too low.

With 6-way genotyping (eg SBTuna), calculations are done separately for the 6-way and 4-way versions. It is quite possible for a locus to look bad in the 6-way version, but good in the 4-way version; if so, don't just throw it out entirely, but try setting `useN=4` for that locus, or `useN=3` if null frequency is dangerously low.

For examples, see the `kinference-vignette`.

Usage

```
check6and4(
  geno6,
  thresh_pchisq_6and4,
  return_what= c("just_pvals", "all"),
  extra_title= "",
  show6= FALSE
)
```

Arguments

`geno6`	a `snpgeno` object with 4-way (or optionally 6-way) genotypes
`thresh_pchisq_6and4`	a pair of thresholds for "bad" and "really bad" p-values. These determine the color in which each locus appears in all subplots.
`return_what`	one of `just_pvals` or `all`; see value

extra_title	a character string to be added to the bottom-right corner of all plots. Best if < 25 characters.
show6	show the plots for 6-way goodness-of-fit? Defaults to TRUE. If diplos is anything other than genotypes6, should be FALSE.

Value

Creates per-locus vectors pval6 and pval4 for 6-way and 4-way genotypes respectively. If return_what="just_pvals", these are returned in a list; if return_what="all", they are added as columns to geno6$locinfo.

---

drop_dups_pairwise_equiv

Grouping duplicate samples

Description

find_duplicates only does pairwise comparisons. However, tissue from the same animal may turn up in multiple samples, so that one sample may turn up in many duplicate-pairs, and the pairs are linked. This function constructs equivalence classes- each corresponding notionally to one *animal*- showing which samples belong in each class. It can either return the entire set of classes, or it can pick just one sample from each class and then return the "surplus" duplicate samples. With the latter, if you then drop those elements as per vignette("kinference"), only one sample from each animal will be retained.

The algorithm merges two classes whenever any sample in the first class is flagged as a duplicate of any sample in the second. Thus, it will be sensitive to false-positive duplicates (though less so to false-negative ones). It's up to you to make sure that the input really contains true duplicates!

Usage

```
drop_dups_pairwise_equiv(ij, want_groups = FALSE)
```

Arguments

ij	2-column matrix or data.frame; possibly row numbers in a dataset, or strings as per rowid_field (qv).
want_groups	if TRUE, also return the equivalence-classes themselves, as attribute groups.

Details

Input should be row numbers in a snpgeno objects of duplicates, as a two-column data.frame or matrix with each row being a pair of duplicates, or the output from find_duplicates (a 3-col matrix). Identifies groups of equivalent observations (e.g., if i and j are duplicates, and j and k are duplicates, then i, j, and k are all equivalent). Outputs a vector of the row numbers for all-but-one of each group.

Value

Surplus elements in `ij`, perhaps plus attributes `groups` if `want_groups=TRUE`. You can look at that to figure out which elements are being retained (one "representative" from each equiv class). If `ij` has no rows, an empty integer vector is returned, without any attributes.

See Also

chain.pairwise

Examples

```
pairs <- matrix( c(
294, 289,
328, 294,
904, 857,
905, 904),
    ncol=2, byrow=TRUE)
drop_dups_pairwise_equiv( pairs, TRUE)
#[1] 289 328 857 905
#attr(,"groups")
#attr(,"groups")$`5`
#[1] 294 328 289
#
#attr(,"groups")$`6`
#[1] 904 905 857
```

---

dropbears *Red-rumped Dropbear data*

Description

An anonymised snpgeno dataset of Red-rumped Dropbear (*Thylarctos plummetus*, ssp. *haemorhous*) genotypes for 480 individuals at 2000 loci. Owing to difficulties retaining field staff for dropbear research, these data represent only a single sampling year.

Genotypes are "4-way", ie single-nulls and homozygotes are not distinguished, and double-nulls are potentially allowed for, although there aren't any in this dataset.

Allele frequencies were estimated using `est_ALF_ABO_quick` (qv) which *does* estimate a null-allele frequency. It's arguable whether that was the best choice, given that nulls seems so rare; we might have used `est_ALF_nonulls` (qv) instead. See `kinference-vignette` for more.

```
diplos( dropbears) # 4-way encoding
dropbears[ 1:5, 1:3] # subset of covariates and genotypes
sum( dropbears=='OO') # no double-nulls...
sum( dropbears=='AB') # ... but lots of heterozygotes!
```

Usage

```
dropbears
```

Format

An object of class gbasics::snpgeno()

Author(s)

Shane M Baylis <email: shane.baylis@csiro.au>

`est_ALF_6way` *Estimate ALFs from 6-way genotypes and snerr*

Description

Performs 6-way re-estimation of ALFs, given 6-way genotypes, starting 4-way estimates of ALFs (from est_ALF_ABO_quick), and estimates of "snerr" (single-to-null error rates for apparent homozgyotes, per locus). Used as a second-pass estimate after ALFs have already been estimated roughly from 4-way genotypes (hence the phrase "re-estimation"). Used in CSIRO pipelines, e.g. for SBTuna, where the genotyping and estimation of snerr has already been done by routines in the **genocalldart** package. If this makes no sense to you, then just stay away from 6-way genotyping!

Usage

```
est_ALF_6way(snpg, control = list())
```

Arguments

snpg	a snpgeno object with 6-way genotypes (i.e., diplos(snpg) matches get_genotype_enconding()$genc with snerr and pbonzer included
control	as per nlminb

See Also

est_ALF_ABCO, re_est_ALF, and est_ALF_ABO_quick

Examples

```
head( bluefin$locinfo$snerr) ## has to exist for 6-way genotypes
bluefin$locinfo$pbonzer <- NULL ## remove pre-existing allele freq
## estimates!
bluefin <- est_ALF_ABO_quick( bluefin)
head( bluefin$locinfo$pbonzer)
bluefin <- est_ALF_6way( bluefin)
head( bluefin$locinfo$pbonzer)
```

`est_ALF_ABCO` *Estimate allele frequencies olde-style, including nulls*

Description

Most users should avoid these! `re_est_ALF` calculates maximum-likelihood estimates of minor, null, and 3rd-allele freqs from a `snpgeno` object. Its workhorse is `est_ALF_ABCO` which has a weirder syntax unless you are using an old CSIRO "pipeline", and is documented here for that reason only.

`re_est_ALF` will accept genotypes 4-way genotypes including "AAO" and "BBO", 6-way genotypes (but there are better options in that case; see below), or triallelic "ABCO" genotypes. The latter corresponds to `genotypes_ambig` as seen in the code of `define_genotypes`; they allow for an optional 3rd allele and nulls, but do not distinguish between single-nulls and homozygotes. Historically, for CSIRO users, "ABCO"-type genotypes are produced by `genocalldart::geno_deambig_ABC`.

Null-allele frequency has to be estimated from HWE deviations, so good estimates require a decent sample size.

The original use-case for these functions was datasets where bona fide 3rd alleles are common; even though they are not used in any `kinference` step (because they get recoded to nulls, i.e. "neither A nor B" en route), it's useful to have them around for ALF estimation. There's a bit of history behind this, which I shan't go into here.

If there are no 3rd alleles, then `re_est_ALF` is not the best choice. If you really do have 6-way genotypes (i.e. differentiating single-nulls from homozygotes, at least approximately) then you *could* use `re_est_ALF` but the problems are:

`re_est_ALF` does not use the extra statistical information on single-null vs homozygote that is available with 6-way genotypes, whereas `est_ALF_6way` does;

and in the absence of that information, it's much slower than `est_ALF_ABO_quick` would be, so why not use that instead?!

Usage

```
re_est_ALF( snpg)
est_ALF_ABCO( lociar, geno_amb = attr( lociar, 'geno_amb'))
```

Arguments

`snpg`	a `snpgeno` object.
`lociar`	a `loc.ar` or `snpgeno` object, normally with a `geno_amb` attribute from 4-way genotyping (see next).
`geno_amb`	a set of 4-way genotypes. In CSIRO's 6-way genotyping pipeline (in package **genocalldart**, not for general use), 4-way genotypes get called first and stored in a `geno_amb` attribute, before making initial allele frequency estimates with `est_ALF_ABCO`. those are then used as starting values in 6-way genotype-calling and allele-frequency estimation, and the `geno_amb` attribute gets discarded..

Value

re_est_ALF returns the input, adding a 4-column matrix pbonzer to the $locinfo attribute, plus attributes gobs and gpred showing observed and expected counts of each genotype per locus. pbonzer is what you want for subsequent calculations. est_ALF_ABCO is similar but creates a new attribute $locinfo$pambig, instead of creating/modifying pbonzer.

See Also

est_ALF_ABO_quick for normal users, est_ALF_6way for special 6-way people.

Examples

```
head( bluefin$pbonzer) # C alleles sometimes present
bluefin$locinfo$pbonzer <- NULL ## remove pre-existing ALFs
bluefin$locinfo$snerr <- NULL ## remove pre-existing snerr
bluefin <- re_est_ALF( bluefin)
head( bluefin$locinfo$pbonzer) # C-alleles have gone to 0
```

---

est_ALF_ABO_quick *Estimate allele frequencies with nulls*

Description

est_ALF_ABO_quick is the recommended way to estimate allele frequencies from called genotypes, *provided that* you are using biallelic SNPs wtih 4-way genotyping and you believe that null alleles are a real, repeatable thing in your data. If you have biallelic SNPs but you don't believe you have repeatable, heritable null alleles, then use est_ALF_nonulls instead. If you are using genuine 6-way genotyping with nulls, see **Beyond 4 way genotypes**.

So... est_ALF_ABO_quick is for fast maximum-likelihood estimation of A (nominally major), B (nominally minor), and O (bona fide null) allele frequencies for a set of loci. It uses the EM algorithm plus Aitken acceleration; this means the whole calculation can be vectorized across loci, which (compared to direct maximization of the log-likelihood) more than compensates for the notorious inefficiency of EM (and also Aitken helps *a lot*). See the MS or the code for more details. Missing data not allowed.

est_ALF_nonulls is very simple; it just looks at the ratio of A to B alleles across all samples at each locus, and sets the null frequency to 0. This *can* cope with missing data, and any sample-loci recorded as "OO" (double null) is treated as such. Thus, you could do est_ALF_nonulls followed by a random-imputation step to fill in the missings, if you really can't unmissingize them (eg by using a more confident genotype-calling algorithm to your raw data). See **Examples**.

Usage

```
est_ALF_ABO_quick(x = NULL, AB, AAO, BBO, OO,
    tol = 0.0000001, EMtol = 0.001,  quietly = FALSE,
    MAX_AITKEN= 40, return_unconverged= FALSE)
est_ALF_nonulls( x=NULL, AB, AA, BB,
  pbonzer_format= FALSE)
```

Arguments

x	a snpgeno object, or NULL to use the next 4 or 3 args explicitly. diplos(x) should either be genotypes6 or genotypes4_ambig. For est_ALF_nonulls, the latter is treated as if AAO always truly means AA, and BBO means BB; OO is treated as missing.
AB, AAO, BBO, OO, AA, BB	vectors (per locus) of counts of these genotypes. Can't mix with non-null x.
tol	final convergence tolerance (in Aitken steps)
EMtol	tolerance within the EM steps; after this is achieved, try an Aitken step
quietly	if TRUE, then at the end print information on the number of iterations required
MAX_AITKEN	Maximum number of Aitken-accelerations to allow. Most loci converge within 10. We have seen a few where over 30 Aitkens are needed; however, the results after just 10 were still pretty good. You can try pushing it higher, but do check your weird loci for extreme weirdness (see next argument).
return_unconverged	if TRUE and some loci still haven't converged after MAX_AITKEN iterations, then just return the indices of those loci. By default (ie if this is FALSE), you'll just get the estimates with a warning, and that's probably fine.
pbonzer_format	if est_ALF_nonulls is called directly on AB etc rather than on x, then pbonzer_format determines the format of the returned allele frequency estimates:TRUE means you get a 4-column matrix suitable for kinference (to go in $locinfo$pbonzer) and FALSE means you just get a vector of the major (A) allele frequencies.

Value

If x is supplied, then its locinfo attribute will be augmented with the pbonzer (allele frequency) matrix required by most kinference functions. Note that pbonzer has 4 columns always, so here the 3rd column ("C") is set to zero. If x is not supplied, then a 3-column matrix matrix is returned. Rowsums of the matrix are always 1 in either case.

Beyond 4 way genotypes

est_ALF_ABO_quick does actually accept 6-way genotypes, but it would be a bit weird to use it, because the first thing it then does is to re-encode the genotypes as 4-way, thus sacrificing some statistical information. It starts by merging the single-nulls with true homozygotes, and then tries to "unmerge" them statistically! It would be better to use est_ALF_6way, though you might need to first run est_ALF_ABCO or est_ALF_ABO_quick to get starting values.

est_ALF_ABCO (qv) is a much slower version that can handle triallelic SNPs (ie potentially with a C allele). Like est_ALF_ABO, it does not distinguish between single-nulls and homozygotes. It should give the same results as est_ALF_ABO for loci without a C allele. Syntax is a bit different because of its role in the legacy 6-way pipeline, so you might prefer to use re_est_ALF (qv) which does the same thing but with different (easier?) formatting of the input.

est_ALF_6way (qv) uses 6-way genotypes, where single nulls are called separately from homozygotes, but potentially with error. It requires error estimates in $locinfo$snerr. Unless you are dealing with a legacy 6-way dataset, you don't want to go near this!

Examples

```
#dropbears$locinfo$pbonzer <- NULL ## no population allele frequency estimates!
#dropbears <- est_ALF_ABO_quick( dropbears)
#head( dropbears$locinfo$pbonzer)
#dropbears <- est_ALF_nonulls( dropbears)
#head( dropbears$locinfo$pbonzer)
## Randomly make some values missing...
#nb <- nrow( dropbears)
#nl <- ncol( dropbears)
#nmiss <- round( 0.1 * nb * nl)
#missij <- cbind( rsample( nmiss, 1:nb, replace=TRUE),
#    rsample( nmiss, 1:nl, replace=TRUE))
#missbears <- dropbears
#missbears[ missij] <- 'OO'
#missbears <- est_ALF_nonulls( missbears)
#plot( missbears$locinfo$pbonzer[,1], dropbears$locinfo$pbonzer[,1]) # very similar
#abline( 0, 1)
#imputor <- function( x){
#  # I don't *recommend* this, but you *could* randomly impute "missing" (OO) genotypes like this
#  # if you are sure there's no nulls
#  # I haven't tested the code!
#  # Caveat emptor... and read it carefully to work out what it's
#  # (hopefully)
#  # doing...
#  pA <- x$locinfo$pbonzer[,1]
#  pB <- 1-pA
#  pAB <- 2*pA*pB
#  pAA <- sqr( pA)
#  pBB <- sqr( pB)
#  misso <- which( x=='OO', arr.ind=TRUE)
#  r <- runif( nrow( misso))
#  x[ misso] <- 'BB' # default
#  x[ misso[ r < (pAB+pAA)[ misso[,2]],] <- 'AA'
#  x[ misso[ r < pAB[ misso[,2]],] <- 'AB'
#return( x)
#}
```

`find_duplicates` *Kin-finders for loads-of-SNPs datasets*

Description

These take a snpgeno dataset that has been processed as far as `check6and4` (and for HSPs, `kin_power`) and find various relations between the samples. Relationships include duplicates (DUPs/dupes/dups), parent-offspring pairs (POPs) and half-sibling pairs (HSPs) or other 2nd-order kin, plus of course unrelated pairs (UPs). You can specify the same or different subsets of the snpgeno for comparison: e.g., first subset for the adults, second for the juveniles.

There are two versions aimed at POPs currently called `find_POPs` and `find_POPs_lglk`. The former uses a "weighted pseudo-exclusion" ("wpsex") statistic that allows for null alleles and is robust

to genotyping errors. The latter uses a likelihood-based statistic (again allowing for nulls), but you do have to provide a guesstimate of genotyping error rate (to robustify the calculation- otherwise, a single genotyping error in a true POP could give a log-likelihood of -Inf). `find_POPs_lglk` is newer, easier to explain, and perhaps less arbitrary, but we have used the "wpsex" version on all our real CKMR datasets (>10). Time will tell whether one is better/easier than the other; finding POPs ought to be pretty easy, so the results really should be the same.

`find_HSPs` should really be called `find_2KPs` because it cannot discriminate amongst second-order kinships; there is no way to distinguish genetically between HSPs, Grandparent-Grandchild Pairs, and Full-Thiatic Pairs (eg aunt/nephew) with `snpgeno` data alone. But, for historical reasons, it's still called `find_HSPs`. Note that `find_HSPs` can also be tricked into targeting [some] other types of kin, such as 3KPs; see **Details**, but watch out.

Usage

```
find_duplicates(
  snpg,
  subset1 = 1 %upto% nrow(snpg),
  subset2 = subset1,
  max_diff_loci,
  limit_pairs = 0.5 * nrow(snpg),
  nbins = 50,
  maxbin = ncol(snpg)/2,
  show_plot = TRUE,
  ij_numeric= is.null( rowid_field( snpg))
)
find_HSPs(
  snpg,
  subset1 = 1 %upto% nrow(snpg),
  subset2 = subset1,
  limit_pairs = 0.5 * nrow(snpg),
  keep_thresh,
  eta = NULL,
  nbins = 50,
  minbin = NULL,
  maxbin = NULL,
  ij_numeric= is.null( rowid_field( snpg))
)
find_POPs(
  snpg,
  subset1 = 1 %upto% nrow(snpg),
  subset2 = subset1,
  limit_pairs = 0.5 * nrow(snpg),
  keep_thresh,
  eta = NULL,
  nbins = 50,
  maxbin = NULL,
  WPSEX_UP_POP_balance = 0.99,
  ij_numeric= is.null( rowid_field( snpg))
```

```
)
find_POPs_lglk(
  snpg,
  subset1 = 1 %upto% nrow(snpg),
  subset2 = subset1,
  gerr,
  limit_pairs = 0.5 * nrow(snpg),
  keep_thresh,
  eta = NULL,
  nbins = 50,
  minbin = NULL,
  maxbin = NULL,
  ij_numeric= is.null( rowid_field( snpg))
)
```

Arguments

snpg	a snpgeno object
subset1, subset2	numeric vectors of which samples to use (not logical, not negative). Defaults to all of them. Iff subset1 and subset2 are identical, only half the comparisons are done (i.e., not *i* with *j* then *j* with *i*). Some sanity checks are made.
max_diff_loci	(find_duplicates) max number of discrepant 4-way genotypes to tolerate in "identical" fish. Only the pairs with fewer than max_diff_loci discrepancies will be retained. Try increasing this from say 10 upwards, and hopefully nothing much will change (though at some point things will change a lot, as you get into the non-duplicate bit of the distribution). Once you have identified duplicates, see drop_dups_pairwise_equiv for how to remove them.
limit_pairs	Integer. Defines the *maximum* number of candidate pairs to keep. Will provide a warning if the number of identified pairs equals limit_pairs.
nbins, minbin, maxbin	find_XXX functions summarise their pairwise comparison statistics into bins (in the part of the range where exact values are uninteresting), as well as returning specific pairs that pass the "interesting" threshold. nbins sets the number of bins, minbin sets the top value of the lowest bin (so that bin stretches from -Inf to minbin for HSPs); maxbin sets the highest. For HSPs, the minimum is 3 bins (-Inf:minbin),[minbin:maxbin),[maxbin:Inf). minbin is not used for duplicates, nor for the non-likelihood find_POPs, since the statistics there are defined so that the lowest possible value is 0. The defaults for minbin and/or maxbin may not be what you need in all cases, so be prepared to select manually and then re-run. For duplicates, where calculations can be slow for big datasets, you can set nbins=0 to disable binning and focus instead on just finding the pairs with fewer than max_diff_loci discrepancies. Each pairwise calculation normally loops over all the loci, but is aborted when the running total of discrepant loci reaches maxbin (or, if nbins=0, when it reaches max_diff_loci), thus saving considerable time. It is therefore not sensible to have maxbin<max_diff_loci (think about it!).

show_plot — whether to plot log histogram. Regardless, plot will not be shown if other arguments would lead to stupid result (e.g. no bins...).

ij_numeric — if FALSE, use the rowid field (see with_rowid_field) to label the pair-members, rather than their row numbers.

keep_thresh — (find_HSPs and find_POPs) is the analog of max_diff_loci for find_duplicates. It determines which pairs to retain for individual inspection. For find_HSPs and find_POPs_lglk, this is the lowest retained PLOD; for wpsex-based find_POPs, it's the highest retained wpsex. Set it with the aim of including anything interesting (ie not *missing* any interesting pairs) and do expect false positives; that is, be willing to have some weaker kin in there, and to subsequently filter those out yourself, as per vignette. For HSPs, and for POPs with find_POPs_lglk, values like 0 (near the HTP mean) or -5 are a good start. For POPs with wpsex-based find_POPs, experiment (perhaps starting with 0.1). You may have to re-run the function a few times if you have been too brutal or too generous here - though "too generous" can be fixed post hoc just by filtering the result, as long as you haven't generated tooooo many pairs (see parameter limit_pairs).

eta — (find_HSPs, find_POPs_lglk, and find_POPs) Not essential; limit for calculating empirical mean and var PLOD, to compare with theoretical mean_UP and var_UP. If you care about this (and you might not, since for find_HSPs the observed/expected binwise comparison is perhaps clearest), then set it to somewhere above 0 that should include almost all UPs and exclude most strong kin; that's an *upper* limit for HSPs and lglk-based POPs, but a *lower* limit for wpsex-based POPs in find_POPs. To use eta, be prepared to look at the histograms and think. The general idea is that the number of UPs should dominate any other kin-type in large sparsely-sampled datasets, so there shouldn't be much problem even if you do accidentally "contaminate" the empirical UP statistics with a few weakish kin at the top end.

WPSEX_UP_POP_balance — (find_POPs) loci receive a weight which is proportional to (difference in probability of pseudo-exclusion between UP and POP) / (variance of indicator of pseudo-exclusion). But, should this be variance assuming UP or POP? This parameter sets the balance; bigger values make it more UPpity, so placing more emphasis on avoiding false-positives, which is probably the Right Thing To Do in most cases. 0.99 could be completely fine... but hopefully the value won't affect the result much anyway.

gerr — (find_POPs_lglk) Genotyping error rate (apart from any AA/AO-type errors)-which had better be a small number. You have to pick it yourself, but it is only used to "robustify" the lglk-based (PO)PLOD for testing POPs vs UPs, and thus can be a rough guesstimate. FWIW we have used 0.01 (i.e. 1%), which is considerably higher than suggested by an analysis of our replicate samples, but it is "safe" while still being small enough not to muck up overall statistical performance. You should really do the same thing yourself, and if you are very paranoid then try sensitivity analyses; but in practice, the results of find_POPs_lglk are liable to be so clear-cut that you may not feel it necessary to try more than one small value.

Details

Some categories will "catch" others (e.g. `find_HSPs` will certainly include any POPs too), so you may need the splitter routines such as `split_POPs_from_HSPs` afterwards. The safest general-purpose strategy - but often *not* the most sensible, if your data is nicely organized and you know what you want - would be:

`find_duplicates` and then get rid of them

`find_HSPs` to get *all* kin (though you will usually have to sacrifice some HSPs to false-neg because you'll need a threshold)

`split_POPs_from_HSPs` to split HSPs from POPs/FSPs

`split_POPs_from_FSPs` to split the latter.

The non-splitter functions, i.e. `find_XXX`, might be run on huge numbers of samples, entailing a `choose(huge, 2)` number of comparisons. You don't want all those individual comparison results, and your computer certainly wouldn't enjoy trying to keep them! So the general idea is to set a threshold for what constitutes "maybe worth keeping individually" (that you expect will be generous enough to contain everything you *do* want, plus some dross), and then to retain just binned counts of the relevant comp statistic for all comps (usually, the vast majority) which don't make your threshold.

In addition, the `limit_pairs` argument is there to prevent your computer locking out with bazillions of unwanted pairs (in case you guess the bin limit inapproriately); the comparisons will be stopped if `limit_pairs` is hit, with a warning. In that case, you probably need to change a threshold, or re-run with larger `limit_pairs`. The default isn't meant to correspond to any biomathematical logic, it's just to stop blue smoke coming out your USB ports.

For `find_duplicates`, there are at least two different use-cases. First, you might want an initial run on a non-too-large subset of your data, to check that dups *can* be clearly distinguished and to look at typical extent of genotyping errors (based on clear duplicates that don't match at every locus). For that, you can set `nbins` and choose some reasonable guess as to `max_diff_loci` (say, 50 loci). Because you set `nbins>0`, *every* pair (almost...) gets checked at *all* loci, so it can be slow. Thus, if you have done this before and have a good sense of "how bad can a real duplicate be?", then set `nbins=0` (and `max_diff_geno` to a small but safe value that won't miss any realistic duplicate-with-genotyping-error) so it will abort a comparison early as soon as it reaches `max_diff_geno` differing loci. That saves a *lot* of time on big datasets! You won't get a histo of number-of-diffs, but you don't need one for that use-case. The "almost" is that `find_duplicates` uses "transitivity" (if A is a dup of B and of C, then we don't need to check B vs C), so it only counts differences for not-yet-known duplicates *based on* `max_diff_loci`. To discard duplicates and to find entire equivalence-classes of duplicates, e.g. from a control specimen included in numerous plates, see `drop_dup_pairwise_equiv`.

`find_HSPs` relies on pre-computed values of "LOD" and "PUP" that have been set by `kin_power`. Normally you would call the latter with `k=0.5`, since that's what HSPs are. However, the devious user can try *different* values of `k`- which is how `find_POPs_lglk` works- and then the target of `find_HSPs` will become "kin with that value of `k`". Be very careful!!!

Value

A `data.frame` with extra attributes (see below) and at least 3 columns: statistic `PLOD` or `wpsex` or `ndiff` (number of mismatching genotypes), then `i` and `j` which are the indices/rows in `snpg` of the two members of each pair. (In ancient times `i` and `j` misleadingly referred to the subsets instead,

but we have moved on!) The attributes in all cases include bins (upper boundaries), some kind of count statistic for number of comparisons in each bin (names vary), binprobs (theoretical CDF for UPs in find_HSPs; should also exist for *POPs* in the old "wpsex"-based find_POPs, but currently doesn't), some of the input parameters, and the call that invoked the function. find_POPs adds a column named nABOO, showing the number of AB/OO exclusions for that potential POP. This is a useful additional diagnostic; it should be close to 0 for true POPs (it can only result from genotyping error or mutation, whereas AAO/BBO can result from nulls). For UPs, we have seen values typically in the low 20s, which is pretty good separation. A 2D scatter-plot of wpsex and nABOO can be more informative than either statistic on its own. find_HSPs and find_POPs have a bunch of extra attributes which should be reasonably clear. For find_HSPs, mean_sub_PLOD and var_sub_PLOD are the empirical means & var below eta, andy they should be close to mean_UP and var_UP *iff* eta has been chosen sensibly. For find_POPs, the same goes for mean_wpsex_hi and var_wpsex_hi. For duplicates, not *all* pairwise duplicates are recorded, unless the subsets are different - otherwise you could have quadratic horror of enormous numbers of pairs arising from a cluster of say 100 identical controls! Since duplication is transitive (ie if *i* & *j* are the same, and *i* & *k* are the same, then *j* & *k* must also be the same), only the necessary ones are recorded to allow you to filter out yourself afterwards. For example, if samples 1, 3, 5, and 6 are all duplicates, you'll get this:

```
i j:
[1] 3 1:
[2] 4 3:
[3] 6 4
```

but you won't see the pairings for 1/4, 1/6, 3/6. If you just want to strip out all duplicates bar one in each group (and you don't care which one is kept), then you can use the function drop_dups_pairwise_equiv - see **Examples**. For POPs and HSPs, the items below are also returned as attributes (which can be more conveniently accessed by @ if atease is loaded, as per **Examples**). The main point is that the "boring" below-threshold pairs get put into bins and are not kept individually. You often don't need to access these directly, since histoPLOD may be enough. The names sometimes change depending on which statistic is being used.

- eta, keep_thresh
  see above.
- <mean/var>_sub_PLOD, <mean/var>_hi_wpsex
  empirical values for the statistic when it is on the boring (UP) side of eta (ie nearly always).
- <mean/var>_<kin>
  (where kin is UP, POP, or sometimes HSP, FSP) predictions from theory, to compare with previous and show graphically.
- n_<stat>_in_bin
  (where stat is PLOD, wpsex, or ndiff):number of pairs whose statistic fell into each bin.
- bins
  Upper cutpoints for the bins.

Examples

```
## find_duplicates
```

```
library( atease) # @ used below for clarity
library( mvbutils) # IDNK if this is needed explicitly
define_genotypes()    ## creating 'genotypes4_ambig' etc
x <- matrix(sample(c("AAO", "AB", "BBO", "OO"), 10000, TRUE,
    prob = c(0.2, 0.45, 0.2, 0.15)), nrow = 100, ncol = 100)
minisnpg <- snpgeno(x = x, diplos = genotypes4_ambig)
## seed some duplicates in. Sample 2 will be a copy of sample 1,
## and samples 21, 22, and 23 will be an exactly-matching group.
minisnpg[2,] <- minisnpg[1,]
minisnpg[ 22,] <- minisnpg[ 23,] <- minisnpg[ 21,]
## seed some inexact duplicates in. Make sample 3 nearly match
## sample 4, and the same for samples 13 and 14
minisnpg[4,1:80] <- minisnpg[3,1:80]
minisnpg[14,1:80] <- minisnpg[13,1:80]
## find exact duplicates
exact <- find_duplicates( minisnpg, max_diff_loci = 0, show_plot=FALSE)
exact ## finds ij pairs {1, 2} and the group {21, 22, 23}
## as pairs {21, 22} and {22, 23}
## find (in-)exact duplicates
## first a plot, as a guide to where to look...
find_duplicates( minisnpg,
    maxbin= 100, max_diff_loci= 1, show_plot=TRUE)
## looks like there's a few inexact ones
inexact <- find_duplicates( minisnpg,
    maxbin= 100, max_diff_loci= 25, show_plot=FALSE)
inexact ## finds the exactly-matching pairs as before, plus
## the inexactly-matching row pairs {3, 4} and {13, 14} with
## >0 differences
## to remove duplicates, keeping only one member of each
## group, use drop_dups_pairwise_equiv
droppers <- drop_dups_pairwise_equiv( inexact[,2:3])
droppers     ## note that _all but one_ of each _group_ of
## (near-)duplicates is included in 'droppers'
## Drop all-but-one of each set of duplicates:
minisnpg_nodups <- minisnpg[ - droppers,]
## find_HSPs (PLOD_HU)
## bluefin data
## stripped-down data-cleaning for example - see the
## vignette for approach for real data!
pvals <- check6and4( bluefin, thresh_pchisq_6and4 = c( 0.001, 0.0001))
bluefin_1 <- bluefin[ , pvals$pval4 > 0.01]
ilglks <- ilglk_geno( bluefin_1)
bluefin_2 <- bluefin_1[ ilglks > -1050,]
bluefin_3 <- est_ALF_ABO_quick( bluefin_2)
bluefin_4 <- bluefin_3[ , bluefin_3$locinfo$pbonzer[,"B"] > 0.02]
bluefin_5 <- kin_power( bluefin_4, k = 0.5)
dups <- find_duplicates( bluefin_5, max_diff_loci = 20)
bluefin_6 <- bluefin_5[ -c(drop_dups_pairwise_equiv( dups)) ,]
hsps <- find_HSPs( bluefin_6, keep_thresh = 0)
histoPLOD( hsps, log=FALSE, lb = 0, fullsib_cut = 75)
head( hsps)     ## PLODs and row numbers for each pair member in bluefin
library( atease)  ## and then it's sooo much easier to just write...
hsps@mean_HSP     ## mean expected PLOD for true HSPs. mean_UP (unrelated),
```

```
## mean_POP (parent-offspring), mean_FSP (full-sibling) follow the
## same format
## subset comparisons
## limit comparisons to those between animals on plate 3 and animals on
## the other two plates. Useful when, e.g., looking for kinships between
## adults and juveniles, but not kinships between adults and other
## adults. For demonstration, treat plate 3 as adults and plates 1 and 2
## as juveniles (they're not really).
adults <- which( bluefin_6$info$Our_plate == "plate3")
juvs <- which( bluefin_6$info$Our_plate %in% c( "plate1", "plate2"))
hsps_subset <- find_HSPs( bluefin_6,
    subset1 = adults, subset2 = juvs, keep_thresh = 0)
hsps_subset$iplate <- bluefin_6$info$Our_plate[ hsps_subset$i]
hsps_subset$jplate <- bluefin_6$info$Our_plate[ hsps_subset$j]
## all pairs are between a plate 3 fish and a fish on plate 1 or plate 2
head(hsps_subset)
## find_POPs and find_POPs_lglk
pops <- find_POPs(bluefin_6, keep_thresh = 0.1)
## keep_thresh = 0.1 is not magic, but often OK. If you
## don't see a big spike out to the right of your plot,
## you should generally set keep_thresh higher
pops@mean_UP     ## expected mean 'wpsex' for unrelated pairs
pops@var_UP     ## expected variance 'wpsex' for unrelated pairs
hist( pops$wpsex)     ## true POPs should have a wpsex near zero, and
## a nABOO of exactly zero, unless sequencing error has occurred
with( pops, plot.default( wpsex ~ nABOO))
## plot the full distribution from binned records
plot( pops@bins, pops@n_wpsex_in_bin, type = "s")
## add expected mean wpsex for unrelated pairs
abline( v = pops@mean_UP, col = kinPalette("UP"), lwd = 2)
pops_lglk <- find_POPs_lglk(bluefin_6, keep_thresh = 0, gerr = 0.01)
pops_lglk@mean_UP     ## as before
pops_lglk@mean_POP
## plot the full distribution from binned records
histoPLOD( pops_lglk, log=TRUE)
```

`find_dups_with_missing` *Duplicate-finding with some missing genotypes*

Description

This function is not for CKMR datasets (where missingness is not allowed) but rather for "Gene-Tagging" (individual mark-recapture using genotypes as tags). For GT, the genotyping method has to be cheap rather than high-quality, so that some genotypes just end up missing (ie the genotyping pipeline has decided they are unscorable). Rather than "imputing" those missing genotypes (which you'd have to do for finding kin-pairs), for the specific case of finding duplicate samples it is probably better to just compare across the loci that *are* both called in each pair of samples.

find_dups_with_missing looks at all pairwise comparisons and "accepts" any where there are not too many inconsistencies (which can arise within true duplicates, from allelic dropout when read-depth is low). This means that a very-poor-quality-DNA sample might actually match with lots of others, simply because it has so few genotypes called! So, subset your data beforehand to remove Bad Eggs.

Usage

```
find_dups_with_missing(
  snpg,
  subset1 = 1 %upto% nrow(snpg),
  subset2 = subset1,
  max_diff_ppn,
  limit = 10000,
  ij_numeric= is.null( rowid_field( snpg))
)
```

Arguments

snpg	a snpgeno object
subset1, subset2	numeric vectors of which samples to use (not logical, not negative). Defaults to all of them. Iff subset1 and subset2 are identical, only half the comparisons are done (i.e., not *i* with *j* then *j* with *i*). Some sanity checks are made.
max_diff_ppn	What *proportion* of non-missing (ie scored-in-both) loci to treat as the threshold for duplicity?
limit	if you hit this many "duplicates", it will stop, to avoid blowing out memory. It means you set max_diff_ppn too high. For consistency, we should probably have called this keep_n as per other find_XXX functions.
ij_numeric	if FALSE, use the rowid field (see with_rowid_field) to label the pair-members, rather than their row numbers.

Value

A dataframe with columns ppn_diff, then i and j which show pairs of samples that are within max_diff_ppn. (Note that i and j refer to rows in snpg itself, not to whatever was passed in the subsets. This is what any normal person would expect, but long ago the software worked differently...)

---

hetzminoo_fancy *QC checks on sample heterozygosity*

Description

This test looks for samples with anomalous numbers of heterozygotes and/or double-nulls, which can result from (i) degraded DNA or (ii) sample contamination. Useful both for finding outlier samples, and for checking whether the loci are collectively working as they should (as is assumed by all the calculations in kinference). The histogram should coincide nicely with its predicted line.

The basis for the "hetzminoo" statistic is explained in the accompanying MS. Note that the target argument provides two variants, "rich" and "poor", designed to test for contamination and degradation respectively. With "rich", you should look for outlying samples on the *RHS* of the histogram (too many heterozygotes): with "poor", on the *LHS* (too few). The choice of target affects the weightings across loci, as explained in the MS. In practice, there is often little visual difference, and a bad sample looks bad bad in both. Nevertheless, you *should* run both variants. See kinference-vignette (qv) for examples.

Usage

```
hetzminoo_fancy(
  snpg,
  target = c("rich", "poor"),
  hist_pars = list(),
  showPlot = TRUE
)
```

Arguments

snpg	a snpgeno object, with allele frequencies already estimated and kin_power (qv) already run.
target	which potential problem to focus on.
hist_pars	list of parameters to pass to hist. If you are very sneaky, you can pass in an expression to be evaluated inline instead (ie fly-hacking). No, there's no example showing that!
showPlot	show the plot? Default TRUE

Value

A vector of "hetzminoo" scores. You can then use it to subset your data by removing samples with unpleasant values.

---

histoPLOD *Histogram PLODs*

Description

histoPLOD shows a histogram of PLODs across pairwise comparisons, where PLOD has been calculated by find_HSPs (qv), find_POPs_lglk (qv), or something similar. The "histogram" can either be on a log-scale or the more familiar unlogged scale; the problem with the latter is that there are often so many UPs (we have hundreds of millions for tuna) that the interesting kin-bumps become squashed to the point of invisibility. The plots should show various bumps for the different kinships, overlapping to some extent; the theoretical locations of the bump-centres are shown by vertical lines for the most common kinships.

The typical workflow would be to first call with log=TRUE, to check that the UP bump looks good and that CK bumps (if any) are centred in the right places. For the UP bump only, the width can also be predicted theoretically, and so the entire predicted curve is shown (and it had better be a good match to the empirical distribution, otherwise there is a QC problem). Then you can focus in on interesting bits and actual pairs, by setting log=FALSE and zooming with the lb and ub parameters.

For log=FALSE, the lower bound lb should be set to exclude almost all of the UP bump, which will otherwise swamp the signal from the close-kin pairs. Specifically for HSPs or other 2KPs, it is also possible to show the entire "expected" distribution by setting HSP_distro_show=TRUE. However, unlike the UP bump, its variance and vertical scaling have to be calculated empirically. histoPLOD does that from PLODs between that mean and some upper limit fullsib_cut, which must be chosen manually. Hopefully the FSP/HSP gap is clear, so it won't matter much what you choose within that.

The kinference-vignette has examples.

Usage

```
histoPLOD( PLODs, log = c(FALSE, TRUE)[0],
    mean_show = c( 'POP', 'UP',
        if( !isFALSE( PLODs@trustplot_FSP_HSP_means)) c( 'HSP', 'FSP')),
    UP_distro_show = c(SPA = TRUE, Normal = FALSE),
    HSP_distro_show = !log && !is.null( PLODs$mean_HSP),
    lb,
    ub = NULL,
    fullsib_cut = NULL,
    bin = 5,
    main = deparse1(substitute(PLODs), width.cutoff = 50),
    ...)
```

Arguments

PLODs	dataframe from find_HSPs or conceivably a future find_kin3 etc
log	TRUE or FALSE to call PLOD_loghisto or HSP_histo respectively
mean_show	for which kinships should expected PLODs values be shown? There's no harm in showing all of them, but some kinships won't make sense in particular applications, so you can turn them off with this argument. (Note that the default won't show FSP or HSP "means" for find_POPs_lglk, since they are calculated wrongly at present.)
main	graph title, defaulting to the name of PLODs argument

`...`	passed to the plotting routine, which is `plot` for `log=TRUE` and `hist` for `log=FALSE`.
`UP_distro_show`	Only for `log=TRUE`, this controls which approximations to show for the theoretical PLODs distribution of UPs. In practice, they look damn similar! The remaining args only apply when `log=FALSE`:
`HSP_distro_show`	Only for `log=FALSE`:show the theoretical PLODs distro for HSPs, based on *empirical* variance and number of "definite" 2KPs.
`lb, ub`	only for `log=FALSE`. `lb` is a *mandatory* cutoff for which PLODs to include. `ub` is optional, but can be used to visually exclude very high PLODs:eg to exclude POPs when the real interest lies in HSPs.
`fullsib_cut`	only if `log=FALSE` and `HSP_distro_show=TRUE`, then use this to determine which PLODs to include when calculating empirical variance of "HSP PLODs".
`bin`	bin width for histogram. Only for `log=FALSE` since most PLODs (which will be for UPs) are already binned during `find_HSPs` (qv).

`ilglk_geno` *Check individual multilocus genotypes for typicality*

Description

`ilglk_geno` computes the per-sample log-likelihood across its entire genotype, i.e. sum log Pr g(i,l); and compares the distribution across individuals to the theoretical distribution given allele frequencies. Some mismatch is normal (and can arise just from noise in allele-frequency estimates), but substantial mismatch is bad. You get to define "substantial". `ilglk_geno` can also detect outlier individuals, usually with lglks that are much too low rather than too high; it's not obvious what could generate genomes that are "too typical" at the individual level.

You can use `locator(1)` to click the histogram to figure out where to adjust the `xlim/ylim` values to change the range of the data to inspect more closely. In other words, you can then re-run the function with its `...hist_par` argument set accordingly.

Usage

```
ilglk_geno(snpg, hist_pars = list(), showPlot = TRUE)
```

Arguments

`snpg`	a `snpgeno`. Genotype encoding (see `diplos`) must be one of 4-way, 3-way, or 6-way, as determined by `diplos` and the `$locinfo$useN` field.
`hist_pars`	`list()` passed to `hist` for controlling histogram, e.g. `hist_pars=list(xlim=c(-12000, -6000))`, or use `FALSE` to not plot.
`showPlot`	show the histogram? Defaults to TRUE, but overrideen by `hist_pars=FALSE`.

Value

Vector of log-likelihood for each individual; also usually (but optionally), a histogram of log-likelihood values across individuals.

Examples

```
## get rid of really bad loci
pvals <- check6and4( bluefin, thresh_pchisq_6and4 = c( 0.001, 0.0001))
bluefin_1 <- bluefin[ , pvals$pval4 > 0.01] # drastic QC!
## check for samples that are not like the others
ilglks <- ilglk_geno( bluefin_1)
## looks like anything with a lglk < -1030 is definitely abnormal
bluefin_2 <- bluefin_1[ ilglks > -1030,]
ilglks <- ilglk_geno( bluefin_2) ## much better, but not perfect -
## see other cleaning steps in the vignette
```

---

kin_power *Locus selection for kin-finding*

Description

kin_power is an essential preliminary step for various QC and PLOD calculations on real datasets; it calculates summary quantities for each locus based on allele frequency estimates. For real data, the results per se aren't of direct interest to most users. However, as its name suggests, kin_power can also be used to predict how well a proposed set of loci would work for finding 2KPs. (It can also do the same for 3KPs or other specified weaker kin, although the answer will always be "nope" for those cases anyway.)

kin_power returns its input snpgeno object after adding extra columns to the locinfo attribute, related to the per-locus mean and variance of LOD (presumably an HSP/UP PLOD, though not inevitably) for different true kinships. It respects the per-locus decision about how precisely to genotype (useN=6/4/3).

Usage

```
kin_power( lociar, want_LOD_table=TRUE, k,
  hack_LOD= NULL, sd_half_range= 10)
```

Arguments

lociar	snpgeno objects with the necessary ingredients
want_LOD_table	can't think why you'd set this to FALSE
k	target average kinship for LOD; 0.5 for HSPs, 0.25 for HTPs, etc.
hack_LOD	Don't mess around with this; it's for internal black magic
sd_half_range	Normally leave this alone, but in case the final prepare_PLOD_SPA (qv) step gives an error (rare but possible), try reducing it below the default.

Value

A snpgeno object with augmented columns in `$locinfo`. Those are:

`E_UP`, `V_UP`	mean & variance for UPs
`E_HSP`, `E_POP`, `E_FSP`	as you would expect
`Ediff`	`E_HSP-E_POP` ie the "absolute" power of that locus
`sdiff`	`(E_HSP-E_POP)/sqrt(V_UP)` which is arguably better than `Ediff` for ranking loci It also attaches `LOD`, `PUP`, and `ev01` elements (each a matrix) to `$locinfo`. They have been made dull (see `mvbutils::make_dull`) to improve your viewing experience, but they work fine for all normal purposes (and you can always `unclass` them to remove the S3 class `dull`).

Examples

```
## Next will fail cozza missing pre-calculated objects necessary for kin-finding:
try( hsps <- find_HSPs( dropbears, keep_thresh = 0))
dropbears_1 <- kin_power( dropbears, k = 0.5)
## works now
hsps <- find_HSPs( dropbears_1, keep_thresh = 0)
```

---

kinPalette *Colors for different kinships*

Description

`kinPalette` allows consistent colours for different kinships when plotting; it's used by various `kinference` plotting functions, and you can also use it yourself to add lines, points, etc. The colours are taken from `viridisLite::viridis` (see **References**); see the code for the hex values.

`kinPalette` returns a named vector of hex values, either for all kinships or just for those you specify. So you can do something like this:

```
# plot something...
abline( v=17, col=kinPalette('UP') # use colour for UPs
```

Good manners: Optionally, `kinPalette` will also call `grdevices::palette` for you, overwriting the existing numeric color definitions. Afterwards, `plot(...,col=2)` will show a different color. This isn't normally a good idea (you can always refer to the kin-colors by name; numbers are flaky) so the default is not to (from `kinference` v1.1 onwards). If you really want to do it yourself, then a perhaps-better approach is:

```
kp <- kinPalette()
old_palette <- palette( kp)
# on.exit( old_palette) # if inside a funciotn
```

so that any palette changes are temporary and can be undone.

Usage

```
kinPalette( kinships= names( kincolours), setPalette= FALSE)
```

Arguments

kinships	which kinships to return colors for. Default is all of them.
setPalette	set to TRUE if you really want to set graphics palette.

Value

A character vector of hex codes, with names "POP" etc. If the kinships argument is specified, then just the corresponding elements will be returned.

References

S. Garnier, N. Ross, A. Camargo, B. Rudis, K. Woo, & M. Sciaini. (2023). sjmgarnier/viridisLite: CRAN release v0.4.2 (v0.4.2CRAN). Zenodo. https://doi.org/10.5281/zenodo.7890875

(NB I had to abbreviate the authors' first names to avoid stupid unnecessary warning from R CMD CHECK about non-ASCII characters... in a raw string... in the days of mandatory UTF-8... sigh)

---

prepare_PLOD_SPA *Prepare for kin-finding*

Description

prepare_PLOD_SPA is something you used to have to run before using some kin-finding/QC tools, to set up your snpgeno object for fancy maths woooo (saddlepoint approximations). There are no meaningful options, you just have to run this. It can be *slightly* slow which is why it was a separate step. However, nowadays I don't think you need to run it at all, because it's built into kin_power (qv).

Usage

```
prepare_PLOD_SPA(geno6, n_pts_SPA_renorm= 201, sd_half_range= 10)
```

Arguments

geno6	a snpgeno object that has been thru kin_power
n_pts_SPA_renorm	how accurate to make the approximation. Default should be fine.
sd_half_range	How many SD's into the tails to push the approximation. The default of 10 is massively far. Normally this is fine, but if you get an error (probably from an NA cropping up during extreme calculations), then trying making it smaller.

Value

Another sngeno object with an environment Kenv, which contains functions (with their own preloaded data) allowing null distributions (eg PLODs for true UPs) to be calculated. Various sanity checks are incorporated to try to stop you from stuffing up with out-of-synch loci etc later.

---

split_FSPs_from_HSPs *Discriminate between kinships of known close-kin*

Description

These are for pairs already picked as likely close-kin via one of the find_XXX functions, but whose exact kinship is uncertain; e.g., they might clearly be either POPs or FSPs, but it's not obvious which. The split_XXX_from_YYY functions apply a more powerful likelihood-based test statistic to each pair, to help decide what it is. All these functions use 4-way genotypes, i.e. not relying on 6-way genotyping. They probably should be adapted to cope with 3-way (ie not trusting double-nulls) but currently they aren't (so they do trust double-nulls).

Usage

```
split_FSPs_from_HSPs(snpg, candipairs)
split_FSPs_from_POPs( snpg, candipairs, gerr, use_obsolete_version=FALSE)
split_HSPs_from_HTPs(snpg, candipairs)
```

Arguments

snpg	a snpgeno object
candipairs	normally, a dataframe with rows being pairs and columns *i* and *j* (and possibly others) e.g. from find_POPs or find_HSPs. Can also be a 2-column matrix (each row again one pair).
gerr	genotyping error rate, where 0.01 would mean 1%. It's there to make the POP case robust; you have to choose it, but the precise value should not matter. See **Details**.
use_obsolete_version	the original code for POPs-vs-FSPs was based on a different non-likelihood-based statistic. It turned out to have low statistical power, but did not require specifying a gerr. For replicability purposes, you can still run it by setting this parameter to TRUE, but otherwise don't bother.

Details

The idea of split_FSPs_from_POPs- though this is not the only possible workflow- is that pairs which are *either* POPs *or* FSPs should stand out very clearly from everything else, via find_POPs. Then the job is to pick between those possibilities. The workflow is supposed to be:

nail POPs/FSPs first with find_POPs;

pick between them with split_FSPs_from_POPs, making use of eg age data too;

look for HSPs (and potentially some HTPs) and filter out already-known POPs and FSPs;

filter out HTPs from the remaining set of HSPs with `split_HSPs_from_HTPs` and/or `autopick_threshold`.

However, an equally reasonable workflow might be:

nail HSPs and everything stronger (and potentially some HTPs) with `find_HSPs`;

split HSPs/HTPs from POPs/FSPs with `split_FSPs_from_HSPs`;

filter out HTPs from the remaining set of HSPs with `split_HSPs_from_HTPs` and/or `autopick_threshold`;

use `split_FSPs_from_POPs` to do just what it says. (Again, age data may help in marginal cases.)

All `split_<blah>` functions return expected values under different possible kin-types (but not variances, since these cannot be predicted for all kin-types).

The `gerr` parameter in `split_FSPs_from_POPs` is there to alleviate the problem that a single locus displaying apparent Mendelian exclusion is in theory reason enough to prove that a pair is *not* a POP (if a likelihood-based criterion is used). But, of course, we can have genotyping errors (and, in a reasonably big dataset, mutations). Allowing for a small amount of error gives the method much mor flexibility, without paying a high price in statistical efficiency (provided `gerr` is small). The technical interpretation is that, if a genotyping error occurs at a locus, then the true genotype is replaced by a randomly-drawn genotype from the marginal distro of genotypes at that locus. Real genotyping errors don't work like that, but it is mathematically convenient and achieves the desired effect of robustifying the FSP-vs-POP statistic. The value to use is up to you; you can experiment; if you really want to estimate it, then look at replicate genotypes.

Value

A dataframe with columns `PLOD_FH` (or similar depending on what is being split), `i`, and `j`, containing the indices of each pair. Theoretical means for the two kinships in the function's name are returned as attributes (variances cannot be predicted), as is the `call`.

Examples

```
library( atease) # so x@att--- easier than attr( x, 'att')
## bluefin data
## stripped-down data-cleaning for example - see the
## vignette for approach with real data!
pvals <- check6and4( bluefin, thresh_pchisq_6and4 = c( 0.001, 0.0001))
bluefin_1 <- bluefin[ , pvals$pval4 > 0.01]
ilglks <- ilglk_geno( bluefin_1)
bluefin_2 <- bluefin_1[ ilglks > -1050,]
bluefin_3 <- est_ALF_ABO_quick( bluefin_2)
bluefin_4 <- bluefin_3[ , bluefin_3@locinfo$pbonzer[,"B"] > 0.02]
bluefin_5 <- kin_power( bluefin_4, k = 0.5)
dups <- find_duplicates( bluefin_5, max_diff_loci = 20)
bluefin_6 <- bluefin_5[ -c(drop_dups_pairwise_equiv( dups)) ,]
## via find_HSPs for isolated split of 'FSPsOrPOPs'
hsps <- find_HSPs( bluefin_6, keep_thresh = 0)
histoPLOD( hsps, log= FALSE, lb= 0, fullsib_cut= 70, HSP_distro_show= TRUE)
## note gap between PLOD  60 -- 70, separating likely HSPs
## from likely POPs and FSPs
FSPsOrPOPs <- hsps[ hsps$PLOD > 70,]
HSPsOrHTPs <- hsps[ hsps$PLOD < 70,]
```

```
## ... implicitly >0 from keep_thresh in find_HSPs() call
## via find_POPs_lglk for 'laser-focus on POPs'
maybePOPs <- find_POPs_lglk( bluefin_6, gerr = 0.01, keep_thresh = -30)
hist( maybePOPs$PLOD, breaks = 20)
abline( v = maybePOPs@mean_POP, col = kinPalette("POP"), lwd = 2)
abline( v = maybePOPs@mean_UP, col = kinPalette("UP"), lwd = 2)
## note the 6 pairs clustered around the expected mean PLOD for
## POPs (all > 60). They are probably POPs, but may include FSPs.
maybePOPs <- maybePOPs[ maybePOPs$PLOD > 60,]
## split_FSPs_from_POPs
### using FSPsOrPOPs
splitFSPsOrPOPs <- split_FSPs_from_POPs( bluefin_6,
    candipairs= FSPsOrPOPs, gerr = 0.01)
hist(splitFSPsOrPOPs$PLOD)
## add line of expected PLOD if FSP
abline( v = splitFSPsOrPOPs@E_FSP, lwd = 2, col = kinPalette("FSP"))
## add line of expected PLOD if POP
abline( v = splitFSPsOrPOPs@E_POP, lwd = 2, col = kinPalette("POP"))
## Expected PLOD is -ve for true POPs, +ve for true FSPs
### using maybePOPs
splitmaybePOPs <- split_FSPs_from_POPs( bluefin_6,
    candipairs= maybePOPs, gerr = 0.01)
hist(splitmaybePOPs$PLOD)
## add line of expected PLOD if FSP
abline( v = splitmaybePOPs@E_FSP, lwd = 2, col = kinPalette("FSP"))
## add line of expected PLOD if POP
abline( v = splitmaybePOPs@E_POP, lwd = 2, col = kinPalette("POP"))
## Expected PLOD is -ve for true POPs, +ve for true FSPs
## split_HSPs_from_HTPs
splitHSPsOrHTPs <- split_HSPs_from_HTPs( bluefin_6, HSPsOrHTPs)
hist(splitHSPsOrHTPs$PLOD, breaks = 15)
## add line of expected PLOD if HSP
abline( v = splitHSPsOrHTPs@E_HSP, lwd = 2, col = kinPalette("HSP"))
## add line of expected PLOD if HTP
abline( v = splitHSPsOrHTPs@E_HTP, lwd = 2, col = kinPalette("HTP"))
## Expected PLOD is +ve for true HSPs, -ve for true HTPs
```

`upairid` *Label kin-pairs*

Description

`find_HSPs` etc return *pairs* of kin, labelled by `i` and `j`. If you want to see whether one or more pairs also occur in another set of kin, eg from calling `find_POPs_lglk`, it's tempting to `paste` together each set's `i` and `j`, and then use `match`. But that will go fail if the order has switched, so that `i` in the first set corresponds to `j` in the second, and so on- which can definitely happen with `find_XXX` (for good reasons). To avoid that, `upairid` sorts each (i,j) pair into order before pasting them, so that you can use `match( upairid( first_set), upairid( second_set))` with no need to worry about order-switching.

Usage

```
upairid(i, j, sep='/')
```

Arguments

i, j	If i is a list (presumably from a call to find_XXX) and j is missing, then i. Otherwise, i and j can be vectors of numbers or characters.
sep	character (or string) to put between each i and j.

Value

A character vector of pairwise-sorted, pasted, IDs.

Examples

```
upairid( 1:5, 5:1)
d1 <- list( i=1:3, j=4:6)
d2 <- list( i=4:6, j=1:3) # swapped
match( paste( d1$i, d1$j), paste( d2$i, d2$j)) # nope
match( upairid( d1), upairid( d2)) # yep
```

---

var_PLOD_kin *Predict variance of PLOD for HTPs and HCPs*

Description

var_PLOD_kin infers the extent of genetic linkage based on the empirical (i.e. observed) variance of PLODs among (presumed) true HSPs, by comparing it to the (smaller) variance that would theoretically apply if all loci were inherited independently. It then predicts the empirical variance for HTPs and/or HC1Ps, based on one of two extreme assumptions about the detailed nature of recombination. The truth will likely be between those extremes. var_PLOD_kin is called internally by autopick_threshold (qv), and few users will want to call it directly.

Although the biggest linkage effect comes from the main "order" of kinship (2KP, 3KP, etc, as controlled by the n_meio argument), this is one situation that is also affected somewhat by the "subtype" of kinship (HSP vs GGP vs FTP, etc). For CKMR purposes, the commonest type of kin of given order are typically those born closest in time, so for a given order of kinship, the var_PLOD_kin always assumes that the relevant kin-pairs will consist of the subtype with *single* shared ancestor and minimax number of generations since shared ancestor. This means HSPs for n_meio=2 (FTPs have 2 shared ancestors; GGPs entail 2 generations of gap, whereas HSPs have only 1), HTPs for n_meio=3, HC1Ps for n_meio=4, etc.

var_PLOD_kin can also be used for "simulation", e.g. to explore the effects of different types of linkage, or of different numbers of markers; see the linfo and C_equiv arguments, plus the code itself, for hints.

Nature of linkage: The two extreme alternatives considered by `var_PLOD_kin` each use one parameter to capture "linkage", but the parameter has quite different meanings. The two versions are:

separate chromosomes, each containing the same number of loci, with *no* recombination. The linkage parameter is the number of such chromosomes.

one single chromosome containing all the loci, equally spaced, with random recombination at multiple places. The linkaage parameter is the probability of recombination between adjacent loci.

Usage

```
var_PLOD_kin(
  linfo,
  emp_V_HSP = V_noX(C_equiv, 2),
  n_meio,
  debug = FALSE,
  C_equiv = NULL
)
```

Arguments

`linfo`	either a `snpgeno` object, or its `$locinfo` attribute (or a fake one). Respectively, either `linfo$locinfo` or `linfo` itself should be a dataframe containing columns `e0`, `e1`, `v0`, `v1`, `count`. Each row is one "type" of locus, i.e., with roughly the same values of e/v 0/1, and `count` says how many such loci there are (for simulation purposes; with real data, `count==1`). `e/v 0/1` are means and variances of the per-locus LOD (note no P) when the locus is or isn't co-inherited. See *Details*
`emp_V_HSP`	empirical variance of PLOD among definite HSPs, normally from `autopick_threshold` (qv). You're supposed to be running this on real data, so that `emp_V_HSP` is an actual number; however, for testing purposes, you can set up an artificial version via `C_equiv` below.
`n_meio`	Target number of meioses:2 for HSPs, 3 for HTPs, 4 for HCPs, etc. This is by far the main driver of variance, but technically the only one; see *Subtypes of kin*.
`debug`	Logical flag, not for regular folks to use! Defaults to FALSE. The **debug** package needs to be loaded beforehand.
`C_equiv`	for artificial test, with `emp_V_HSP` set to the no-crossover variance from `C_equiv` chromos (need not be integer). Ignored if `emp_V_HSP` is set.

Value

Matrix with two rows `V0` and `Vx`, and one column for each element of `n_meio` (which is always augmented to include 2), named "M2" etc. The two rows pertain respectively to the no-crossover multiple-chromosome scenario, and the single-chromosome multiple-crossover scenario. The matrix also has an attribute `info`, which is a numeric vector of elements named `V_UP`, `V_HSP`, `C_hat`, and `rho_hat` (note that `V_HSP` should duplicate the first column of the matrix) `C_hat` is estimated equivalent number of chromosomes for the no-crossover scenario, and `rho_hat` is per-locus crossover rate for the all-crossover scenario.

Gruesome details

The "per-locus LOD" (whose properties are stored in the columns `e0`, `e1`, `v0`, `v1` in `linfo`) is created by calling `kin_power` (qv). The normal use-case would be from `kin_power(...,k=0.5)`, so that the (P)LOD pertains to 2KP/UP comparisons. However, if you called `kin_power(...,k=0.25)` then the (P)LOD would be designed for 3KP/UP comparisons, and so on. In fact, you could even hand-tweak the calculations to contain LODs for 3KP/2KP comparisons, which *might* in principle improve the resolution (but you'd have to fiddle manually; you could actually do it based on two calls to `kin_power`, one with `k=0.5` and one with `k=0.25`, and manipulating the results).

Examples

```
# COMPLETELY MADE-UP e/v values! Nothing to do with genetics :)
var_PLOD_kin( data.frame( count=45, ev01= I( cbind( e0=-1, e1=2, v0=0.03, v1=0.02))), C_equiv=22, n_meio=3:4)
#        M2    M3    M4
#  V0 208.2 156.6  91.9
#  Vx 208.2 186.2 101.8
#  attr(,"info")
#      V_UP    V_HSP    C_hat  rho_hat
#    1.3500 208.2273  22.0000   0.2616
```

Index

∗ **data**
    bluefin, 8
    dropbears, 12
∗ **misc**
    autopick_threshold, 5
    chain_pairwise, 8
    check6and4, 9
    drop_dups_pairwise_equiv, 11
    est_ALF_6way, 13
    est_ALF_ABCO, 14
    est_ALF_ABO_quick, 15
    find_duplicates, 17
    find_dups_with_missing, 24
    hetzminoo_fancy, 25
    histoPLOD, 26
    ilglk_geno, 28
    kin_power, 29
    kinference-package, 2
    kinPalette, 30
    prepare_PLOD_SPA, 31
    split_FSPs_from_HSPs, 32
    upairid, 34
    var_PLOD_kin, 35
%upto% *(*find_duplicates*)*, 17

autopick_threshold, 5, *5*, *33*, *35*

bluefin, *5*, 8

chain_pairwise, *4, 5*, 8
check6and4, *4*, 9, *17*

data *(*bluefin*)*, 8
diplos, *28*
drop_dups_pairwise_equiv, *4*, 11
dropbears, *5*, 12

est_ALF_6way, *4*, 13, *14–16*
est_ALF_ABCO, *4*, *13*, 14, *16*
est_ALF_ABO_quick, *4*, *12–14*, 15, *15*
est_ALF_nonulls, *3, 4*, *12*
est_ALF_nonulls *(*est_ALF_ABO_quick*)*, 15

find_duplicates, *4*, *11*, 17
find_dups_with_missing, *3, 4*, 24
find_HSPs, *5*, *27*, *33, 34*
find_HSPs *(*find_duplicates*)*, 17
find_POPs, *5*, *32*
find_POPs *(*find_duplicates*)*, 17
find_POPs_lglk, *5*, *27*, *34*
find_POPs_lglk *(*find_duplicates*)*, 17

get_chain, *5*
get_chain *(*chain_pairwise*)*, 8

hetzminoo_fancy, *4*, 25
histoPLOD, *5*, 26

ilglk_geno, *4*, 28

kin_power, *5*, *7*, *17*, *21*, 29, *31*, *37*
kinference, *7*, *9*, *14*, *16*, *26*, *30*
kinference *(*kinference-package*)*, 2
kinference-package, 2
kinPalette, 30

prepare_PLOD_SPA, 31

re_est_ALF, *4*, *13*, *16*
re_est_ALF *(*est_ALF_ABCO*)*, 14

split_FSPs_from_HSPs, *5*, 32
split_FSPs_from_POPs
        *(*split_FSPs_from_HSPs*)*, 32
split_HSPs_from_HTPs
        *(*split_FSPs_from_HSPs*)*, 32

upairid, 34

var_PLOD_kin, *5*, *7*, 35