Package: gbasics (via r-universe)

June 5, 2026

Title Genotype classes and support routines for CKMR

Author Mark Bravington, Shane Baylis

Description S3 classes for manipulating genetic data, primarily NGS SNPs (sequence counts and/or genotypes) but maybe more, plus support for other kin-finding/genotyping packages; ultimately meant for Close-Kin Mark-Recapture.

Maintainer Mark Bravington <markb1@summerinsouth.net>

License GPL

Imports utils, stats, mvbutils (>= 2.12), atease

Suggests tinytest, SNPRelate

KeepPlaintextDoco YES

Encoding UTF-8

Version 1.2.1

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

Date/Publication 2025-05-25 11:33:52 UTC

RemoteUrl https://github.com/markbravington/gbasics

RemoteRef HEAD

RemoteSha d381002207da746ba86c7aaef4276136b8137660

Contents

`gbasics-package` *Genotype storage ultimately for Close-Kin Mark-Recapture*

Description

`gbasics` provides classes for storing multilocus genotypes, plus some support routines. The ultimate purpose is to support Close-Kin Mark-Recapture, and specifically the finding of close-kin pairs, which is handled by the **kinference** package. The package was originally developed around 2015 for CKMR projects at CSIRO Australia, and it has been much extended since then. The only thing most (non-CSIRO) users will need is the `snpgeno` class, which stores genotypes, sample-level covariates, and locus-level metadata. `snpgeno` is the only class accepted by functions in the **kinference** package.

The `snpgeno` object requires all genotypes to have been called already (that's not our problem!). It is normally created by reading from a SNPGDS or VCF file, or by converting from an existing R object of class `snpgds` (see package **SNPRelate**). But you can also construct a `snpgeno` object manually; see `snpgeno` examples.

`gbasics` also includes a few utilities to help with genotyping and kin-finding calculations, including SaddlePoint Approximation (eg `renorm_SPA`) and parallel root-finding (`ridder`). They aren't primarily meant for the general user, but if they do prove useful to you, then that's great!

Legacy: `gbasics` also includes "legacy" support for other types of data and intermediate stages of genotyping, used in several CSIRO pipelines. In fact, there are no less than 4 S3 classes for handling multilocus genotype data (alongside sample-specific and marker-specific data), primarily NGS but also usats at a pinch.

If you are starting from "raw genetic data" and using your own genotype-calling (genocalling) process, you may need the other classes as intermediate steps. CSIRO usually does genocalling for CKMR via package **genocalldart**, which is not publicly available. The other three `gbasics` classes are `diploido`, `loc.ar`, and `NGS_count_ar`. They are the products of evolution, specifically of CSIRO's data-sources and genotyping processes, so they do have a few quirks; starting from what is we now know, we would have designed those classes differently, and there might be changes to their APIs in future. `NGS_count_ar` is perhaps the most stable.

Null alleles and missings and dropout and encodings: `gbasics` doesn't do anything with your data; it just stores it in a useful format. Therefore, this isn't really the place to go into details about what your data "should" be, because `gbasics` itself doesn't care. However, the **kinference** package is much stricter, and it might help to be aware of some of this at the reading-it-in stage.

Basically, `kinference` expects a *definite* genotype call at every locus for every sample. Of course, the call might be wrong, but that's survivable; the main thing is that it should not be missing or "dunno". `snpgeno` object *can* store missing genotype calls, but you will have to impute them or something before you can use the object in `kinference`.

kinference is designed to cope with genuine null alleles, which are common in some loci in some CKMR applications with some genotyping methods (eg based on ddRAD). A null allele is a repeatable (ie across multiple samples from the same animal), heritable mutation at a locus, that is *different* to the major & minor (reference & alternate) alleles and that doesn't show up in when genotyping in any directly-visible way. Nulls are *completely different* from dropout where an allele fails to show up in one sample due to bad DNA or processing glitches. Occasionally, both copies of a locus will be null in one sample; this is a "double null" which should be clear from the genotyping details. More commonly, just one copy is null, so that a naive user might interpret the genotype as a homozygote for whichever (visible) allele is in the other copy. That would be a huge error for kin-finding purposes, even if it doesn't much matter in other genetic applications. Fortunately, the frequency of null alleles can be estimated basically from the excess (if any) of apparent homozygotes (plus any double nulls that are present), and nulls can then be allowed for in kin-finding. The **kinference** package has more details.

One common phenomenon is for genotyping software to put "empty" or "dunno" calls in when it's not sure. "Dunno" calls are *not* the same as double-null calls; for the latter, the genotyping software should be confident that neither the Major nor the Minor allele is present. If your genotyping software is making many "dunno" calls, then you basically need to tell it to pull itself together and not be so wussy- just make the damned calls! Within limits, the **kinference** package is robust to some amount of genotyping error, but it won't accept missing data (a very deliberate decision, and a good one!).

To continue the rant: sometimes people use "null" loosely to mean "double-null" call- so that "there are very few nulls in my dataset" might really just mean "there are very few double-nulls or missings". Double-nulls aren't the main problem; *single* nulls are much commoner, not directly visible, and can have a major effect on kin-finding.

Different genotyping methods might handle nulls differently; eg some methods *can* infer the presence of a single null, albeit with uncertainty. Some methods intrinsically lead to many more nulls than other methods. Also, it's not just about nulls: some methods and datasets might conceivably have a lot of loci with more than two non-null alleles (on the way to "microhaplotypes"), which are potentially powerful for kin-finding. To allow flexibility, the snpgeno class can therefore accept different sets of possible genotype-calls, as defined by the "genotype encoding" in diplos. You will have to decide which encoding-set to use for your data. Only a few encodings are accepted by kinference, but it might be possible to convert a non-standard encoding manually before passing it to kinference; see get_genotype_encoding and snpgeno.

..WHAT.IF.I.REALLY.DON'T.HAVE.NULLS?

If you are sure that your data should not contain nulls, then you can still use the 4-way encoding and so you can load your data with the standard functions like read_<blah>2snpgeno; of course, the data shouldn't contain any double-null genotypes. When it comes to estimating allele frequencies, you can enforce a null allele frequency of zero via kinference::est_ALF_nonulls, and all the QC and kin-finding steps will respect it; your homozygotes will be treated as true homozygotes!

Not currently true: The documentation used to say this: package **gbasics** also includes some utility routines which duplicate stuff in other MVB-written packages, and which are included here to avoid excessive dependencies. These are being tidied up...

But it's no longer true; the functions have been left in mvbutils and there's no plans to manually import them into gbasics. What *is* true, though, is that there's a medium-term plan to split mvbutils into two packages, one with completely uncontroversial utilities needed by other packages such as gbasics, and one with all the complicated and sneaky code for package maintenance

and R life management that's only for enlightened users, such as our lizard-people overlords, or me ;).

`diploido` *Class for one diploid locus*

Description

The ancient and probably obsolete class `diploido` lets a diploid locus be treated as a 1D object (a vector), even though it's really a 2D matrix; no, I can't remember exactly why. The usual generics are available. Contrast with `loc.ar`, which works with entire data.frames most of which consists of a multilocus genotype.

Usage

```
diploido(cop1, cop2)
```

Arguments

`cop1`	integer vector
`cop2`	integer vector

See Also

loc.ar

`get_genotype_encoding` *Diploid genotype encodings*

Description

Many of the functions in `kinference` etc can accept several different encodings for diploid genotypes, depending on how the genotyeps are distinguished. For example, one encoding specifies that single-nulls are called separately from homozygotes (perhaps with error), whereas another says that single-nulls and homozygotes are not distinguished; some encodings allow a 3rd or 4th etc option for the allele; etc.

Encoding is specified by the `diplos` attribute in the `snpgeno` object. The hope is that it should normally be handled for you automatically, but you can set the encoding manually to one of a few prespecified options that `kinference` will understand; the choices are shown by `get_genotype_encoding()`, though only a few will actually work with the **kinference** package. An encoding is stored as a character vector of recognized genotype-categories; some are described next.

Some popular encodings: This is probably the *wrong* place to describe the various encodings currently used by `kinference` (a few others are or were used by CSIRO's own genotyping pipeline before the results are `kinference`-ready). Anyway, the commonest ones are:

- 4-way (`genotypes4_ambig`) allows the values "OO", "AB", "AAO", "BBO". This means nulls are allowed, double-nulls ("OO") are taken notice of, but single-nulls and homozygotes are not differentiated, so that "AAO" means "A" was present but "B" was not, and conversely for "BBO". You can use this encoding even if your data is guaranteed null-free; there just won't be any "OO", and the null-allele frequency can be set to 0, which means `kinference` will always interpret "AAO" as a homozygote.
- 3-way (`genotypes3`) comprises "AB", "AAO", and "BBOO" where the latter covers "BB" homozygotes and "BO" single-nulls *and* "OO" double-nulls. The motivation is for loci where nulls are rare but not non-existent, and where double-nulls (which *might* in practice be artefacts, although we assume they aren't) get wildly over-interpreted in a kinship setting. Merging genotype-categories never causes bias, but it does sacrifice a small amount of statistical information to gain (much more important) robustness. Nulls can actually be informative, so "blanket 3-way" is mostly not the default for CSIRO's datasets; problems only come when nulls are rare. So CSIRO has sometimes used `genotypes4_ambig` for all loci, but setting the `locinfo$useN` field to 3 for loci where double-nulls are too risky. Various functions in `kinference` look at `locinfo$useN` to decide whether an individual locus is treated *during analysis* as 3-way, 4-way, or perhaps 6-way (next) rather than following the general encoding of the whole dataset.
- 6-way (`genotypes6`) with possibilities "AA", "AB", "AO", "BB", "BO", and "OO". Here, the genotyping process has tried to distinguish between single-nulls and true homozygotes, based on read-depth. The discrimination will never be 100% accurate, even for "good" loci, so explicit allowance must be made for getting it wrong sometimes (if you see references to `snerr`, that's why). For some loci it's not even worth trying, so for them `locinfo$useN` is set to 6 if it's worth trying to differentiate single-nulls from homozygotes for that locus, or 4 if nulls are fairly common but hard to differentiate, or 3 if nulls are rare and hard to differentiate. The pipeline for 6-way genotyping uses in-house CSIRO code and is quite fiddly— so it's likely to *stay* in-house! CSIRO has used 6-way genotyping for *large* ongoing studies of SBTuna and school sharks, so this encoding will continue to be supported, but for new projects we would recommend 4-way instead, using more loci to compensate.

You will see that 4-way is a coarse-grained version of 6-way, and 3-way is a coarsened version of 4-way. The `locinfo$useN` field (a column of numbers) can be used to selectively coarsen certain loci, but of course it can't "uncoarsen" any part of a 3-way-encoded dataset if the overall encoding was too coarse. In most cases, 4-way will be the starting point.

Internal representation: Internally (though you *shouldn't* need to know this), the genotypes in a `snpgeno` are stored in a `raw`-mode matrix, where each element takes 1 byte. The elements of the encoding correspond in order to the numbers 1,2,3,.... With `genotypes4_ambig`, for example, an element with raw value 1 means "OO", with value 2 means "AB", etc. Value 0 should never be seen in real datasets, but is sometimes useful as a temporary placeholder to show that "something needs fixing"! It will display as "?".

Generally speaking, `sngpeno` objects use raw value `ff` (255) to indicate "missing" (as distinct from double-null). However, the **kinference** package deliberately does not handle missing data; *all* genotypes have to get called, even though some of those calls might be in error, and genotyping errors are allowed for (or ignored) in the subsequent statistical analyses.

Special S3 methods exist for printing, comparing, and assigning genotype encodings, so it should at least *look* clear to you, even if you have no idea what all this is about. By design, you can't directly assign integer values into genotypes; see **Examples**.

Changing encodings: Mostly, encoding should be handled automatically during the creation of your snpgeno. However, you do sometimes need do access to it, and *occasionally* you might want to manually recode your data. That takes some care because the internal representation of the encodings won't generally match up; for example, "AB" is encoded as 1 for genotypes3 but as 2 in genotypes4_ambig. See **Examples** for how to re-encode all genotypes from 4-way to 3-way (which you also could do less painfully via useN, without changing the encoding— I think useN is described more in the kinference-vignette.

Expandability: You can in fact use any character vector you like; there is no need to stick to those provided by define_genotypes, and the interpretation of the character-strings is entirely at the discretion of downstream software. For example, you could specify your own encoding scheme for microhaplotypes; from memory, I think up to 7 variants could be accommodated within the 8-bit raw storage. However, most of the kinference functions can only handle one or two of the pre-specified encodings, so you'd need new software.

Inside code: From the command-line, and in many functions, get_genotype_encoding is the most useful function. However, inside some code define_genotypes might be more useful.

It is meant only for use inside other functions, typically at the start of the function; it creates objects in whatever environment it was called from. Once you've called it, you can refer directly to a genotype "AB" as just AB etc (ie no quotes), and you can also refer to genotype-encodings that it knows about by name, eg genotypes4_ambig.

If *you* run define_genotypes() from the R prompt, those things will be created in .GlobalEnv. So, most people shouldn't. There is usually no specific need to run define_genotypes in your own code either, unless you're aiming to extend kinference; the point of documenting it here, is to explain and illustrate how the genotype encodings work. For All Practical Purposes, get_genotype_encoding should be enough.

Usage

```
# Really meant to be used early inside another function
define_genotypes(nlocal = sys.parent())
get_genotype_encoding()
```

Arguments

nlocal	the frame number, or environment, to create things in- see mlocal. Leave this alone unless you *really* know what you're doing...

Value

get_genotype_encoding returns a list of character vectors. Their names aren't "sacred"; the only thing that matters to kinference is the character-vector of strings in the encoding. define_genotypes returns nothing, but various objects are created; see the code.

See Also

snpgeno; cq and mlocal in package **mvbutils**

Examples

```
# Use of define_genotypes() inside a function:
library( mvbutils) # just for '%is.a%'
count_hetz <- function( x){
    stopifnot( x %is.a% 'snpgeno')
    define_genotypes()
    # Now eg genotypes4_ambig is directly available...
    # and so is AB instead of "AB", etc. Cor blimey.
  return( sum( x==AB)) # thus saving a pair of quotes...
  }
data( snpgarbage)
count_hetz( snpgarbage)
sum( snpgarbage=='AB') # had to type two quotes...
# Recode snpgarbage as 3-way (no good reason)
get_genotype_encoding() # for inspection; returns them all
diplos( snpgarbage) # aha! Looks like 'genotypes4_ambig'
# Make a copy; *don't* try to modify-in-place!
sg3 <- snpgarbage
sg3[] <- as.raw( 0) # will flag an error later if we forget something
# sg3[] <- 0 won't work; guard against user boo-boo
# Set the desired encoding...
diplos( sg3) <- get_genotype_encoding()$genotypes3
# ... and re-map _all_ values from the old encoding
# Note that AB and AA0 may not use the same raw code in both
# encodings, so you gotta explicitly set those ones too
sg3[ snpgarbage=='00'] <- 'BB00'
sg3[ snpgarbage=='AA0'] <- 'AA0'
sg3[ snpgarbage=='AB'] <- 'AB'
sg3[ snpgarbage=='BB0'] <- 'BB00'
snpgarbage # out with the old...
sg3 # ... in with the new
```

`loc.ar` *Create & manipulate locus-arrays*

Description

A `loc.ar` object is meant for "numerical diploid genotype data". The class is *not* used by the **kinference** package, is not particularly well thought out, and may not be properly documented here. We no longer use `loc.ar`s much at CSIRO except as intermediate steps in constructing a `snpgeno`, but in the past we have used them for: (i) microsat genotypes, with each sample-locus being a pair of numbers for allele-length; and (ii) biallelic SNP read-depth pairs, with the genotypes (if decided) stored as a hidden extra. The latter purpose is generalized to "microhaplotypes" (multiple alleles per locus) by `NGS_count_ar` (qv), which is probably more "future-proofed". `loc.ar` probably won't go away from `gbasics`, but it is not guaranteed to be actively maintained (whereas `snpgeno` and `NGS_count_ar` probably will be); then again, perhaps it will be improved in future. Who knows?

There are methods for the usual things like subsetting (see **Details**), `rbind`, `cbind`, `print`, `head`, `tail`, `str`. There is also a constructor `loc.ar`- an S3 generic- but currently there are *no* methods

for it (at least not in package **gbasics**; a dodgy one has been moved to `genocalldart` for now). Any `loc.ar` object has to be built manually, as done by a few functions in package **genocalldart**, such as `read_count_dart`.

At heart, a `loc.ar` is a 3D numeric (or integer) array with dimension `(n_samples,n_loci,X)` where X is a fixed maximum number of non-null alleles for each locus (always either 2 or 3 in our applications to date), but with attributes that contains auxiliary data: sample-specific, locus-specific, optionally sample-by-locus, plus whatever other fixed attributes you want to give it. It is broadly similar to snpgeno, which is better documented, except for the details of its contents (3D numeric, vs 2D `raw`-mode interpreted as actual called genotypes). Currently, there are methods for printing, subsetting, rbind/cbind, and for transforming to/from dataframes with two cols per locus. Subsetting can handle the auxiliary data automatically, as described below.

Auxiliary data is stored in attributes which can be accessed via the `$` operator. Sample-specific auxiliary info (which must exist) can be accessed via `x$info`; it should be a dataframe containing a column "Our_sample". Locus-specific auxiliary info is optional (but you'd be brave not to have it...); it should live in a dataframe containing one column "Locus" (the name), and is stored in `x$locinfo`. Loci are *rows* here, whereas loci are *columns* in `x` itself. Other than subsetting, there are currently no special operations on `locinfo`; for example, `print( x)` does not show it (whereas, for snpgeno objects, the field "Locus" is shown vertically as the column name).

Further auxiliaries, such as SNP genotypes (sample-by-locus), can also be tacked onto the attributes as you please. All attributes are copied by subsetting. By default, the whole attribute is copied, except for (i) `info` and `locinfo`, which subset automatically in the appropriate way, and (ii) any auxiliaries named in the attribute `x$subset_like_both` (a character vector that you can set manually), which should be arrays/matrices whose first two dimensions pertain to sample and to locus respectively (just like the main data in a `loc.ar`); they can have more dimensions, too.

I sometimes add class `specialprint` from package **mvbutils** to `loc.ar` objects, to *suppress* printing of some auxiliaries. See EXAMPLES.

Internal note: I used to use class diploido to hold microsat genotypes inside dataframes, but that was less flexible. The `genocalldart::SBTlike_loc.ar` creator function can take input from "diploido"-class objects in dataframes, rather than just dataframes with e.g. "D225_A.1" and "D225_A.2" columns. If any column is of class "diploido", all are assumed to be, and "normal" loci are ignored.

Usage

```
loc.ar( x, ...) # generic
unloc.ar( la)
```

Arguments

`x`	in theory, something to be converted into a `loc.ar` (but there are no methods yet!)
`la`	a `loc.ar` with exactly 2 alleles per locus (thus an `S*L*2` array) to be turned into a dataframe with the `info` columns, plus two columns for each locus. Why, you may ask?
`...`	just in case

Details

Sample info and locus info are stored as dataframes in `attr(<x>,"info")` and `attr(<x>,"locinfo")` respectively. Internally, the code uses the **atease** package so it can just write e.g. `x@info` instead, but you should not need to load `atease` because you can always use the `$` and `$<-` accessors instead.

For subsetting, note that `k`, or `j` *and* `k`, can be omitted, in which cases `i` and `j`, or `i` alone, will be used as the *first* subscript; this is *unlike* dataframes, where a single subscript gets used as the *second* subscript. If the resulting array contains only one locus, it will by default be collapsed to an n-by-2 matrix, unless `drop=FALSE` explicitly. Otherwise, the result will stay as a `loc.ar` provided the 3rd dimension stays at length 2. The first dimension (samples) is never dropped.

`unloc.ar` should probably acquire a "diploido.use" argument, to optionally return loci as `diploido` objects rather than pairs of columns.

Value

`loc.ar`, `[`, `rbind`	a `loc.ar`
`unloc.ar`	a dataframe with two cols per locus

See Also

NGS_count_ar, snpgeno

---

make_genopairer *Mapping between pairs of possible genotypes and compressed form*

Description

You almost certainly do not need, nor want, to call this. But it needs to be exported so that `kinference` and other packages can find it.

With diploid genotypes such as "AA" and "AB" at a locus, the pairwise probs under given co-inheritance (0, 1, or 2 copies) are symmetric. It is daft to store the full symm matrix, and causes some boring technical problems too. So, `make_genopairer` creates a mapping between a vector of pair-names such as "AA/AB" and the underlying pairs. It allows eg a 2000x6x6 array to be stored as 2000x15, and makes passing it to Rcpp faaaaar easier.

Usage

```
make_genopairer(genotypes)
```

Arguments

`genotypes`	character vector, produced by e.g. `define_genotypes`, which you probably shouldn't be calling either.

Value

Matrix with rows & columns being the individual genotypes, and entries a lookup from that pair into a character vector of possible *ordered* pairs, which is stored as the "what" attribute. See **Examples**.

Examples

```
make_genopairer( c( 'AA', 'AB', 'BB'))
#   AA AB BB
#AA  1  2  3
#AB  2  4  5
#BB  3  5  6
#attr(,"what")
#[1] "AA/AA" "AB/AA" "BB/AA" "AB/AB" "BB/AB" "BB/BB"
```

NGS_count_ar *Next-Gen sequencer count data*

Description

NGS_count_ar is a class for raw counts by sequence-variant within locus, per sample. The number of alleles (variants) can differ between loci. It's not used by any functions in the **kinference** package, which expects snpgeno objects throughout. However, it's used extensively in CSIRO's private genocalldart package. It's an early stage in processing DartSeq/DartCap/DartTag data—long before genotypes are called— and can also be used to hold similar data from other formats such as VCF, via read_vcf2NGS_count_ar(). It's also a generic S3 method for creating such an object, with one low-level method for matrices (which you probably shouldn't call yourself) and another method for converting from loc.ar objects, which will presumably have come from genocalldart::read_count_dart or such. The default method (ie if NGS_count_ar is called on any old rubbish) will stop(), by design. See **Details** for internal structure, and what you can add/expect.

There are methods for basic stuff: try methods(class="NGS_count_ar"). dim(x) returns #samples*#loci; for the number of alleles, use nrow(x$seqinfo). The print method tries to make things clearly readable, and has a few extra arguments to help control that- see **Arguments**. Subsetting can have up to 3 indices i, j, and k; j can be numeric, logical, or character (i.e. locus names, which are matched against x$locinfo$Locus). If k is omitted, another NGS_count_ar object is returned, as controlled by i and/or j. If length(k)==1, then the result is a 2D array of sample-by-locus. If length(k)>1, the result is a 3D array of sample-by-locus-by-allele, with NA counts added when k refers to an allele "number" that doesn't exist for that locus; obviously, all loci then have the same number of "alleles" in the new structure. dimnames are set for the 2nd (locus) dimension only. These "pure array" results don't preserve the detailed sample or locus information (except via the dimnames) but can be useful for subsequent manipulation.

Obscure note: A similar class is loc.ar, which preserves counts but allows max 3 non-null alleles per locus. It's not clear which of loc.ar or NGS_count_ar is "more advanced"- depends on the context. However, most later stages in genocalldart pipeline (eg for bait selection) require loc.ar. To go back the other way, you probably need to call pick_ref_alt (qv); see the pipeline examples.

Usage

```
NGS_count_ar( x, ...) # generic
## S3 method for class 'matrix'
```

```
NGS_count_ar( x,  # S3 method for "matrix'
    sampinfo, locinfo, seqinfo,
    strip_numerics = TRUE,  rename = TRUE, ...) # S3 method for matrix
## S3 method for class 'NGS_count_ar'
print( x,  # S3 method for "NGS_count_ar'
    trailing_dot = getOption("trailing_dot_NGS_count_ar", FALSE),
    dot_for_0 = getOption("dot_for_zero_NGS_count_ar", FALSE),
    center_dot=centre_dot,
    centre_dot= getOption( 'centre_dot', getOption( 'center_dot', FALSE)),
    ...) # S3 method for NGS_count_ar
```

Arguments

`x`	for the constructor, an integer matrix with dimensions (allele,sample), i.e. the *transpose* of final storage. Or a `loc.ar`. For `print`, an `NGS_count_ar`
`sampinfo`	dataframe of sample info
`locinfo`	dataframe of locus info
`seqinfo`	dataframe of sequence info (including which locus the sequence belongs to)
`strip_numerics`	if TRUE, remove any non-integer columns from `locinfo`- typically, these are summary data in a CSV that you can live without
`rename`	certain names are expected by fun/xctions that handle `NGS_count_ar` objects; this patches up "alternative" names that are generated by `read_cluster_dart3`
`trailing_dot`	set TRUE if you want to show that count data is non-integer (eg after norming by sample-total-reads) so that all counts end with a period. The post-decimal-place digits are not normally important, but if you really need to see them, you can do so by setting a `k` subscript. Or set the global option, as per **Usage**.
`dot_for_0`	set TRUE if you want zero-counts replaced by a dot. Using the global option is often helpful. Or set the global option, as per **Usage**.
`center_dot`, `centre_dot`	iff `dot_for_0` is TRUE, whether to use a central dot (Latin-1 and Unicode 0xb7) or a period (if FALSE) to replace leading zeros.
`...`	passed to other methods

Details

The allele counts are stored as a (sample*sequence) matrix. There are also three key attributes, all accessible using the `$` operator, and each a dataframe:

info Metadata on each sample, including "Our_sample" and, usually for Dart data at least, "TargetID"

locinfo Metadata on each locus; see below

seqinfo Metadata on each allele, including "Locus", "FullAlleleSeq", and probably others

The key fields in `x$locinfo` (there might be others) are

Locus Unique strings

n_alleles Number of sequence variants (presumably 2 or more)

consensus String of the overall sequence, with dots at SNP sites

var_pos String showing the positions of SNP sites, eg "17,41"

end_col of the sequences found at that locus (needed for lookup into the main matrix)

You can also add other attributes that apply to the whole object. One that's used in `genocalldart` is `mean_fish_tot` (used in norming count data from new samples), but it's rather dicey for general use because it does depend on the set of loci; so if you subset by loci, things (should) change...

See Also

`loc.ar`, `snpgeno`

`read_snpgds2snpgeno` *Create a 'snpgeno' from 'snpgds' file or object*

Description

`read_snpgds2snpgeno` is a wrapper allowing one-line conversion from a `snpgds`-format *file* to a `snpgeno`. Needs the **SNPRelate** package.

Usage

```
read_snpgds2snpgeno(
    filename,
    locusID,
    sampleID,
    infoFrame = NULL,
    infoFields = NULL,
    locinfoFrame = NULL,
    locinfoFields = NULL,
    plateField = NULL)
```

Arguments

`filename`	string giving the snpgds file name, contents formatted as per `snpgdsExampleFileName` in package **SNPRelate**.
`locusID`	string namgiving the name for locus IDs. Cannot be blank
`sampleID`	string naming the column for sample IDs. Cannot be blank
`infoFrame`	optional string naming the sample metadata dataframe
`infoFields`	optional character vector naming the sample metadata variables
`locinfoFrame`	optional string naming the locus metadata dataframe
`locinfoFields`	optional character vector naming the locus metadata variables
`plateField`	optional string naming the column for sample-specific plate ID

Details

Locus-specific metadata and sample-specific metadata may each be supplied as either a single dataframe (using a non-NULL `infoFrame` and/or `locinfoFrame`), or as a vector of field names (using `infoFields` and/or `locinfoFields`). If all are NULL, no metadata other than the locus ID and sample ID are read in.

Examples

```
# From SMB:
#if (!requireNamespace("BiocManager", quietly=TRUE))
#    install.packages("BiocManager")
# BiocManager::install("SNPRelate")
if( requireNamespace( 'SNPRelate')){
  # Simplest possible case (no locus or sample metadata other than IDs;
  # will add an all-one plate field):
  sg <- read_snpgds2snpgeno(filename = SNPRelate::snpgdsExampleFileName(),
      locusID = "snp.id", sampleID = "sample.id")
  # More likely: either sample or locus metadata is a single frame,
  # and the other is a bunch of separate fields. In this case, sample
  # metadata is a single frame and locus metadata is separate fields.
  sg <- read_snpgds2snpgeno(filename = SNPRelate::snpgdsExampleFileName(),
     locusID = "snp.id", sampleID = "sample.id",
     infoFrame = "sample.annot",
     locinfoFields = c("snp.rs.id", "snp.position", "snp.chromosome", "snp.allele"))
}
```

`read_vcf2snpgeno` *Read VCF genotype data*

Description

`read_vcf2snpgeno` and `read_vcf2NGS_count_ar` try to read the genotype part of a VCF file, and load it into a `gbasics` object: either a `snpgeno` (genotype calls) or a `NGS_count_ar` (sequence counts). They discard the "metadata" at the top of the file, and the "data lines" describing parts of the genome.

The `snpgeno` version expects a "GT" subfield containing genotypes. The `NGS_count_ar` expects a subfield "AD" (Allele Depth).

VCF format is complicated, and this isn't fully tested; the goal is to produce something that will get through the next few steps of the kinference process. If you want something more sophisticated, please feel free to hack this code (and if you do a good job of it, please let us know!). There are other R packages out there which- to *some* extent- handle VCF, and you may be able to use one of those to create a `snpgds`, which can then be converted by `sngeno` (to the latter class).

Usage

```
# BLOCK has same default in both functions
read_vcf2snpgeno(
```

```
  vfilename, BLOCK=formals( read_vcf2NGS_count_ar)$BLOCK,
  allow_disordered_unphased= TRUE)
read_vcf2NGS_count_ar(
  vfilename, biforce = TRUE, BLOCK = 1000,
    allow_disordered_unphased= TRUE, majjik_bleeble = "")
```

Arguments

vfilename	Filename, or an existing connection that is already open for reading. If vfilename is a connection, it is *not* closed on exit.
biforce	if TRUE, condense all multi-allelic loci into just one Ref (as defined in the VCF- it's the first count) and one "Alt", the latter consisting of *all* other sequence counts for that locus.
BLOCK	how many lines to read in at once. Default should be fine.
allow_disordered_unphased	By default, unphased genotypes are allowed in any order, so "1/0" and "0/1" are both tolerated and they mean the same thing. I'm not sure that's actually intended according to VCF specifications, though; my initial reading was that only non-decreasing (e.g. "0/1") was allowed, but I've now relaxed it to save having to write too many explanations like this one. Set the parameter to FALSE to enforce a stop() if any decreasing pairs are found.
majjik_bleeble	Do not mess with this.

Value

NGS_count_ar (qv) or snpgeno object. If the latter, then the genotype encoding (the diplos attribute) is currently genotypes4_ambig, ie AB/AAO/BBO/OO. The info attribute contains the following fields:

Our_sample	column name from VCF
Fishtot	total counts for that sample, or NA if snpgeno
File	filename, or description of the connection
MD5	md5sum() of the file, or NA if a connection NB:"File" and "MD5" are character vectors, but also with class dull from package **mvbutils**, so they don't clutter the printout. It's only printing that's affected; and you can see their contents via unclass(x$info$File). The locinfo attribute always contains the following fields:
Locus	a name for the locus, concocted from CHROM and POS in the VCF
CHROM, POS, ID, REF, ALT, QUAL, FILTER, INFO	as per VCF (these are mandatory fields) and for NGS_count_ar objects, locinfo will also contain these, which are needed by some functions that operate on NGS_count_ar objects:
consensus	same as ID
var_pos	info on variant position

n_alleles what it says There are a couple of other housekeeping fields required by NGS_count_ar (qv), which need not concern us here. For NGS_count_ar objects, there is also a seqinfo attribute which, aside from housekeeping fields, contains the field "FullAlleleSeq", which is concocted from the ID and REF and ALT fields. I haven't tried to disentangle ALT (it doesn't look easy!), so if there's more than two allele (variant) at a locus, then the multiple ALTs are distinguished by ".1", ".2", etc after the *entire* ALT field in the VCF. If the locus is biallelic, there's no ".1".

See Also

snpgeno, NGS_count_ar

Examples

```
# package BinaryDosage has some nice small VCFs
# The one below just has genotypes
# But I'm not putting in Suggests; hassle of versions etc
# Workaround for CRANkiness:
re_bloody_quireNamespace <- get(
    sprintf( '%s%s', 're', 'quireNamespace'), baseenv()) # anti CRANky
if( re_bloody_quireNamespace( 'BinaryDosage')){
  thrubb <- read_vcf2snpgeno( system.file(
      'extdata/set1b.vcf', package='BinaryDosage'))
  print( thrubb)
}
# Should also have a small public example file for 'read_vcf2NGS_count_ar' but
# must have AD subfield
# I don't have a good one
```

---

renorm_SPA *Saddlepoint approximation support*

Description

These return functions with which you can then evaluate various renormalized univariate SPAs of PDF, CDF and inverse CDF. "All" you have to do is provide the KGF and derivatives, then run one of these routines once; then you will have a single function which you can repeatedly call easily and fairly cheaply to get your SPA values, without knowing what in hell you are doing or what on earth a saddlepoint approximation is. I like to preserve a little mystique, after all.

Usage

```
# PDF:
renorm_SPA(K, dK, ddK, return_what = c("func", "mulfuncby"),
    tol = formals(ridder)$tol,
    sd_half_range = 10, already_vectorized=TRUE,
    limits = c( -Inf, Inf),
```

```
    try_reducing_range_if_NA = TRUE
    )
# CDF and inverse CDF, via integration of the SPA PDF and...
# ... the monotone interpolating spline in stats::splinefun( method="hyman")
renorm_SPA_cumul(K, dK, ddK,
    sd_half_range = 10, n_pts = 2001, already_vectorized=TRUE)
# Inverse CDF directly via Lugannini-Rice-type formula
inv_CDF_SPA2( p, K, dK, ddK,
    tol = formals(ridder)$tol, already_vectorized=TRUE)
```

Arguments

K, dK, ddK	KGF single-argument functions for the 1D KGF and its 1D derivatives.
already_vectorized	Use TRUE (the default) if you have prepared K etc to take vector arguments (i.e. multiple values of the intrinsically-scalar KGF parameter, to be computed "in parallel"). FALSE means that Vectorize will be called to do it for you. If you can reasonably code K etc yourself in a vectorized way, then that will run faster than already_vectorized=FALSE. But you don't have to.
return_what	(renorm_SPA only) "func" gets you a renormalized PDF SPA that you can just, like, call. "mulfuncby" gives you the renormalization constant, i.e. the scalar you should multiply the unnormalized PDF SPA by to get the renormalized version.
sd_half_range	(not inv_CDF_SPA2) how many Standard Deviations each side of the mean to span in the numerical integration required for renormalization. The default of 10 is pretty massive and should be OK for "reasonable" distros, but may be too big (leading to infinities/NAs) for some distros, or too small (not capturing enough of the probability mass) for other distros. So it is up to you to either check or gamble.
n_pts	(renorm_SPA_cumul) number of points to base the renormalization on. The default of 2001 is often excessive; 201 has been fine in my applications (and fewer means faster).
tol	tolerance for root-finding the SPA tilt for each desired PDF argument. (Not required in renorm_SPA_cumul since no root-finding is used there.)
limits	(currently only renorm_SPA) if absolute limits are available, you can pass these and it won't try integrating outside them. EG if the true distro is discrete with finite range of support, go slightly within that range. You may well get away without setting this, but there is then a risk that the default sd_half_range will try to integrate in a range where the SPA breaks down altogether (e.g. beyond the support).
try_reducing_range_if_NA	(currently only renorm_SPA) If the renormalizing-integration step hits an NA, then integrate() will barf; it's *probably* just because the integration range was ludicrously wide (the default is 10). So the default TRUE for this parameter tries repeatedly reducing sd_half_range to 90% of previous value until either (i) success or (ii) it hits 4 SD, at which point it will stop(). If you're getting

NAs that "close" to the mean, it's a bad sign. FALSE means `stop()` if the original `sd_half_range` doesn't work.

`p` What quantile to calculate.

Value

Function(s) with a special environment within which things like the renormalization constant are embedded. `renorm_SPA_cumul` returns a list of two functions, `CDF` and `inv_CDF`; the others return just one.

`ridder` *Parallel root-finder*

Description

Ridder's root-finder for many univariate functions in parallel. Faster than looping over each function in turn, IME.

Ridder's method is a very good 1D root-finding algorithm; see `https://doi.org/10.1109%2FTCS.1979.1084580` for original, or the section in "Numerical Recipes" (probably chapter 9, for the 2007 edition) by Press, Teukolsky, Vetterling, Flannery.

One weakness of this implementation is that all components of `FUN` get evaluated in each iteration, until the very last component has converged. It would be faster if I allowed `FUN` to have a logical-index argument, saying which components to actually evaluate each time.

Usage

```
ridder(FUN, lo = NULL, hi = NULL, tol = 0.000001, skip_bounds = FALSE)
```

Arguments

`FUN`	function with one *vector* argument. If your underlying function has other args, you need to wrap it first so that you can pass a one-argument function to `ridder`.
`lo, hi`	bounds for search that should bracket the roots (can be vectors)
`tol`	for roots (absolute tolerance- a scalar, duhhh)
`skip_bounds`	if TRUE, start the searches exactly at the bounds; quicker, but a bad idea if infinite values result. If FALSE, the search will start inside the bounds and automatically do some bracketing, being sure to avoid the bounds.

Value

The roots (a vector).

snpgarbage *Specimen 'snpgeno' object filled with garbage data*

Description

This is a pretty minimal snpgeno object with random contents, generated by the example code in snpgeno. The reasons for also including it as a data object, are (i) *your* convenience, and (ii) so it can be used in the example of get_genotype_encoding (qv).

Usage

```
data( snpgarbage)
```

Format

An object of class snpgeno.

Author(s)

Mark Bravington <email: markb1@summerinsouth.net>

snpgeno *Class for SNP genotypes*

Description

snpgeno is an S3 class for storing *already-called* genotypes from multiple samples and loci, as well as associated information about those samples and loci. Pretty much all the functions in the **kinference** package expect a snpgeno input, in many cases requiring extra information about the loci, such as allele frequency estimates, which has usually been added by other functions in kinference. You can extend the per-sample and per-locus information by adding fields in the usual R manner, as well as adding extra information that is per-sample-and-locus (such as number-of-reads); there is generally no need to make an "inherited class" for such ad-lib extensions. Printing looks IMO nice, and is succinct. See how elegantly the locus-names are shown (vertically), to save space!

There are a couple of ways to create a snpgeno. One is to call read_vcf2snpgeno (qv) on a VCF file that already contains called genotypes, or read_snpgds2snpgeno (qv) for a SNPDGS file. Another is via the constructor snpgeno, which is an S3 generic with a default method (plus a method for the obscure class loc.ar; see **Note**). The normal way to call the default, would be to give it a character matrix of genotypes, as well as sample-specific and locus-specific data. You can also use the default to give you an "empty" snpgeno of the right size but without genocalls. That might be useful if and only if you are planning to call your own genotypes and want to create the snpgeno manually, in which case the code of snpgeno.default and snpgeno.loc.ar may be informative.

A snpgeno can be subsetted by samples and/or loci, using numeric, logical, or character indices; see **Subsetting**.

snpgeno specific methods currently exist for: rbind, cbind, dollar and dollar-assign (direct access to attributes such as sample-specific covariates and locus-specific information), subset (see **Subsetting**), (in)equality (== and !=), as.character, print, str.

Genotype encoding and storage: (See also get_genotype_encoding for the same information, written slightly differently.)

The word "genotype" can be a bit ambiguous, so we here use "genocall" to describe an already-called genotype (which may be ambiguous, null, etc) specifically for *one* sample and *one* locus. Genocalls are stored internally in a matrix of mode raw (1 byte per entry), for efficiency; if you do unclass(x)[1:5, 1:3] you can see the guts. As the next paragraph explains, the raw values are automatically converted to characters for printing, assignment, and equality-testing (about the only test you can apply to a genocall). Thus, things like x[1,1]=="AA" or x[2,3] <- "BBO" or x[4,5]==x[17,5]' should work fine. To do anything more sophisticated, first call as.character(x) to convert the raw values into a character matrix; don't try to handle the raw elements directly.

Normally all the genocall-encoding stuff (the mapping between raws and characters) will be set up for you automatically, through the constructor call or via read_vcf2snpgeno etc. You should almost never need to worry about the raw values themselves- and it's dangerous to mess with them. But for the record: the encoding, which applies to *all* loci, is via the diplos attribute, a character vector which you can obtain by diplos(x). It is indexed by the raw values, starting at 01. So, if your diplos attribute was c("AB","AA","BB"), then a raw value of 02 would denote an "AA" homozygote (and would print as such). This allows for flexible encodings according to the nature of the data, including null alleles (via deliberately ambiguous genocalls such as"AAO"- either single-null, or reference homozygote) and multallelic loci with up to 6 alleles (though multiallelics are not used by package **kinference** version 1.x).

The interpretation of the encoding is determined entirely by whatever code processes it; there's no intrinsic meaning attached to the character representations. Most kinference functions assume some specific encoding, amongst one of those pre-defined by define_genotypes (qv), and they will check. You can convert manually between encodings, if you are feeling brave; see get_genotype_encodings for an example.

Missing genocalls should be encoded as raw value ff. They (along with any raw values outside the range 1:length(diplos(x))), such as as.raw(0)) will print as "?" and are converted to NA by as.character(x), but there's otherwise no guarantee they're properly supported. Missing genocalls are specifically *not* allowed in our kinference applications, so I haven't made much effort.

You *can* do some dodgy things, such as assigning non-character items directly to elements of x, but you almost certainly *shouldn't*. I'm not going to list them. Just don't.

Other data: Data about individual samples ("metadata" for geneticists; "covariates" for statisticians) is stored in the info attribute, a dataframe which you can access via e.g. my_sngeno$info (thanks to S3 magic, because snpgeno is internally *not* a list; it's just a matrix, with extra attributes). There must be as many rows in info as there are in the main genocall matrix. You can add whatever sample-specific fields you like to x$info. As SUBSETTING explains, you can later nominate one field as "sample ID".

The "locinfo" attribute, which must have as many rows as there are columns in x, can be accessed via x$locinfo. One column *must* be called "Locus". Again, you can add whatever locus-specific fields you like (including matrices) to x$locinfo.

snpgeno methods do their damnedest to ensure that rownames are NULL for info, locinfo, and any other data named in <snpgeno>$subset_like_both (see **User defined extras**). This might

not always work... nevertheless, `rownames` on dataframes are pretty disastrous IMO (and I suspect they're an early design decision much regretted by guRus) and if you *try* to use them with `snpgeno` objects, you *will* hit some kind of trouble. But, see next subsection for a solution...

Subsetting: You can subscript via `x[i,]` or `x[,j]` or `x[i,j]`- but not as `x[single_subscript]`, nor via matrix-subscripting. As usual in R, the subscripts can be integers, logicals, missing, or character. If the `j` subscript (for loci) is character, then it should match elements of `x$locinfo$Locus`. To use characters as `i` subscripts (i.e. by sample), see `with_rowid_field`. However, note that many functions in `kinference` require an `Our_sample` field (which should certainly be unique) and will not pay attention to `rowid_field` even if it nominates a different field.

```
my_snpg <- with_rowid_field( mysnpg, 'UniqueSampleID')
my_snpg[ c( 'Abner12', 'Zoe9'),] # assuming...
# ... that those names appear in 'my_snpg$info$UniqueSampleID'
%%#
```

Subsetting gets applied automatically to the `x$info` and `x$locinfo` attributes, and also to any attributes whose names are given in `x$subset_like_both`. Subset-replacement also works as usual.

User defined extras: Aside from things that belong in `x$info` or `x$locinfo`, you can add arbitrary extra attributes via the "$<-" operator, e.g. `x$extra <- stuff`. These will not be printed by `print(x)`, but of course you can extract and manipulate them yourself if you want. Occasionally, you might want to add something that is per-genocall, so also has dimensions of sample * locus (and possibly extra dimensions afterwards, too): per-allele counts would be an example. In that case, just add its name to the character-vector attribute `subset_like_both` (which will be non-existent if there are none such). For example:

```
x$qualitee <- array( 0, c( nrow( x), ncol( x), 3))
x$subset_like_both <- c( x$subset_like_both, 'qualitee')
```

Thereafter, `$qualitee` should behave sensibly when `x` is subsetted (and allegedly also with `rbind` and `cbind`, though I'm skeptical). Note that it's up to you to make sure the something really has the its first two dimensions correct; this is S3, there is no safety net. There are no `dimnames` built into these attributes, but you can add them yourself; if so, presumably the first two should always equal `x$info$Our_sample` and `x$locinfo$Locus`.
For extra attributes that are not part of `x$subset_like_both`: `cbind(x,y,...)` and `rbind(x,y,...)` should preserve their values from `x` ie the first argument, but will discard all other copies (in `y` and so on).

Usage

```
snpgeno( x, ...) # generic
## Default S3 method:
snpgeno( x, diplos, n_samples, n_loci, info, locinfo,  allow_nonchar, ...) # S3 method for default
diplos( x)
diplos( x) <- value
```

Arguments

`x`	thing to convert. For the *default* constructor, this would normally be a character matrix of genocalls, but NULL is also allowed for an empty result that you can

fill in manually later; iff you set `allow_nonchar=TRUE`, you can also pass in `raw` or `integer` values too, which must not exceed `length( diplos)`.

`diplos, value` encoding for the genocalls, usually one of those in `define_genotypes` (qv). It's a character vector defining the genocalls to which the raw elements in `unclass(x)` will correspond.

`n_samples, n_loci` for default constructor. If not specified, these will probably be deduced either from the dimensions of `x`, or from `info` and `locinfo`. Specifying them explicitly can be useful if you merely want an "empty shell" `snpgeno`.

`info` dataframe with subject-specific usually-non-genetic data (name, date, size, ...). Must contain a field `Our_sample` (unique identifier for that sample/"library"/replicate/...); some downstream functions may require other fields too. For the default constructor, a placeholder will be constructed if `info` is not supplied.

`locinfo` dataframe with locus-specific data. Must include "Locus" (name/definition); some downstream functions may require other fields too. For the default constructor, a placeholder will be constructed if `locinfo` is not supplied.

`allow_nonchar` whether to allow raw or integer genocalls as input to the default constructor, for possible "efficiency"- but then it's the user's responsibility to ensure they do correspond appropriately to `diplos`).

`...` Extra user-defined attributes for the `snpgeno` object (default `snpgeno` constructor only). For `str.snpgeno`, the dots have no effect but must exist Becos R.

Note: For constructing from an existing `loc.ar` object, `x$locinfo` must already include `x$locinfo$pambig` (a 4-column matrix of ABCO allele frequency estimates) and `x$geno_amb` (provisional genocalls). Those are generated as part of CSIRO's CKMR-genocalling pipeline (not yet public), and hopefully are documented in there...

Value

A `snpgeno` object.

See Also

`NGS_count_ar`, `loc.ar`, `define_genotypes`

Examples

```
# Create a snpgeno from scratch, filled with garbage.
set.seed(1111)
library( mvbutils) # becoz I say so
## Genotyping encoding? Show (current & legacy) possibilities
get_genotype_encoding()#
# ... let's use 4-way genotype encoding (probably commonest)
genotypes4_ambig <- get_genotype_encoding()$genotypes4_ambig
## Random genotypes...
n_loci <- 5
n_samples <- 3
genomat <- matrix(
    rsample( n_samples * n_loci, genotypes4_ambig, replace=TRUE),
```

```
    n_samples, n_loci)
## Locus information:
locodat <- data.frame(
      # "Locus" must be present: unique ID strings
    Locus= sprintf( 'L%i', 1:n_loci),
    lenseq= rsample( n_loci, 100:200),
    chromo= rsample( n_loci, 1:24, replace=TRUE),
    poschro= round( 1e7 * runif( n_loci))
  )
## Sample information (ie covariates AKA "metadata")
sampodat <- data.frame(
      # "Our_sample" must be present: unique ID string
    Our_sample= sprintf( "S%i", 1:n_samples),
    Year= rsample( n_samples, 2001:2004, replace=TRUE),
    Weight= runif( n_samples, 1, 5)
  )
## Put theem together
snpgarbage <- snpgeno(
    x = genomat,
    diplos = genotypes4_ambig,
    info = sampodat,
    locinfo = locodat
  )
snpgarbage
diplos( snpgarbage) # what encoding?
snpgarbage$info     # sample info
snpgarbage$locinfo  # locus info
str( snpgarbage) # ... confirming @reliability is there
# Subsetting:
mini <- snpgarbage[ 1:2, 1:2] # subset; also deals with $locinfo & $info
mini
mini$info
mini$locinfo
# Fix a "mistake" (manual editing):
snpgarbage[ 1, 1] <- 'OO'
# Some protection against users:
snpgarbage[ 1, 1] <- 'womble'
snpgarbage
table( as.character( snpgarbage))
# Other aspects of each genocall (a "user-defined extra"):
library( atease) # for x@y instead of attr( x, "y") etc
# which is very convenient
snpgarbage@manual <-
    matrix( FALSE, n_samples, n_loci)
snpgarbage@manual[1,1] <- TRUE # we fixed that one...
snpgarbage@subset_like_both <- 'manual'
snpgarbage    # @manul is not printed, but...
snpgarbage@manual # ... it is there
snpgarbage[ 1:2, 1:2]@manual     # ... and it subsets nicely
str( snpgarbage) # ... confirming @manual is there
```

str.snpgeno *Summaries for various genotype classes*

Description

Default `str` causes horrible crashes on these objects: too long, or something. So, use these instead.

Usage

```
## S3 method for class 'snpgeno'
str( object, loci = TRUE, keys = TRUE, ...) # S3 method for snpgeno
## S3 method for class 'loc.ar'
str( object, loci = TRUE, keys = TRUE, ...) # S3 method for loc.ar
## S3 method for class 'NGS_count_ar'
str( object, loci = TRUE, keys = TRUE, ...) # S3 method for NGS_count_ar
```

Arguments

`object`	of whatever class
`loci`	TRUE or FALSE to show (some) locus names
`keys`	TRUE or FALSE to show (most or all) sample-specific fields
`...`	ignored AFAIK

with_rowid_field *Facilitate sample-based subscripting of sngpenos*

Description

`with_rowid_field` can be applied to a snpgeno to specify which "sample ID field" to use when subsetting a snpgeno by row (ie sample), using character "sample ID" of your choice instead of numeric or logical index. This is often a good idea. If the rowid field has been set, then because `find_HSPs` etc will by default label its kin-pairs with that field, rather than with row-numbers; then, if you subset the main dataset, it's easy to just subset the kin-pairs too (you don't have to keep track of changing row numbers).

You can already subscript a snpgeno by locus with a character index, which is looked up in `<snpgeno>$locinfo$Locus`. The choice of "Locus" for field name is pretty obvious and uncontroversial! But for samples, there's no corresponding uniquely obvious field name so you can use `with_rowid_field` to specify the one you want.

Specifically, that field in `<snpgeno>$info` will be used to look up character-mode `i` in `<snpgeno>[i,...]` subsetting. Then you can subset to samples using their "identifiers", rather than having to use logical or numeric lookup. Also, if `with_rowid_field` has been called, then subsequent calls to `find_HSPs` (qv) and friends will return the rowid_fields in `i` and `j` rather than row-numbers, which is much less error-prone for downstream use.

You certainly don't have to call `with_rowid_field` on your snpgeno (it didn't exist until 2023...), but unless you do, you'll be stuck with logical/character subsetting by sample. Of course, you can still use those afterwards, as well.

`rowid_field` is a convenience lookup function to remind yourself which field you specified...

Details: `with_rowid_field` is actually a generic which can mark two (currently) classes of object: snpgeno and `data.frame` (or descendents of the latter). If you call it on a snpgeno, it actually gets applied to `<snpgeno>$info` which is a `data.frame`; this means that you can subset not just `<snpgeno>` itself, but also `<snpgeno>$info`, with character sample info. Note that this might theoretically *contradict* the default behaviour of a `data.frame`, which is to look up character row indices in the highly-unreliable `rownames(<data.frame>)`; I force rownames *off* (ie NULL) in a `<snpgeno>$info` so this shouldn't be a problem.

`with_rowid_field` augments the S3 class of a `data.frame`, but not of a snpgeno (the functionality of using rowid is built into the latter, and the rowid label is attached to `<snpgeno>$info` not to `<snpgeno>` itself). I know it does say "DETAILS" here but there are limits, so I'm not going to explain why.

There are methods for `[`, `[<-`, `rbind`, and `cbind`, which attempt to do the right thing...

Usage

```
with_rowid_field( x, rowid_field) # generic
rowid_field( x) # generic
```

Arguments

`x`	thing that you'll want to do character-based row subsets on
`rowid_field`	which column name to look up.

Value

`with_rowid_field` returns the original object marked (somewhere in its internals) with a `rowid_field` attribute. `rowid_field` returns a string, or NULL.

Examples

```
## Not run:
sn <- snpgeno(...) # with "sampID" as a field in 'info'
try( sn[ 'samp_X12',]) # crash
sn <- with_rowid_field( sn, 'sampID')
sn[ 'samp_X12',] # goodo
rowid_field( sn) # [1] "sampID"
rowid_field( sn$info) # ditto
sn$info[ 'samp_X12', ] # whatever

## End(Not run)
```

Index