Package 'vcfR'

January 10, 2020

Title Manipulate and Visualize VCF Data

Description Facilitates easy manipulation of variant call format (VCF) data. Functions are provided to rapidly read from and write to VCF files. Once VCF data is read into R a parser function extracts matrices of data. This information can then be used for quality control or other purposes. Additional functions provide visualization of genomic data. Once processing is complete data may be written to a VCF file (*.vcf.gz). It also may be converted into other popular R objects (e.g., genlight, DNAbin). VcfR provides a link between VCF data and familiar R software.

Version 1.9.0

Maintainer Brian J. Knaus <briank.lists@gmail.com>

URL https://github.com/knausb/vcfR, https://knausb.github.io/vcfR_documentation/

Depends R (>= 3.0.1)

LinkingTo Rcpp

Imports ape, dplyr, graphics, grDevices, magrittr, memuse, methods, pinfsc50, Rcpp, stats, stringr, tibble, utils, vegan, viridisLite

Suggests adegenet, ggplot2, knitr, poppr, reshape2, rmarkdown, scales, testthat, tidyr

VignetteBuilder knitr

License GPL-3

RoxygenNote 7.0.2

Encoding UTF-8

NeedsCompilation yes

Author Brian J. Knaus [cre, aut] (<https://orcid.org/0000-0003-1665-4343>), Niklaus J. Grunwald [aut] (<https://orcid.org/0000-0003-1656-7602>), Eric C. Anderson [ctb], David J. Winter [ctb], Zhian N. Kamvar [ctb] (<https://orcid.org/0000-0003-1458-7108>), Javier F. Tabima [ctb] (<https://orcid.org/0000-0002-3603-2691>)

Repository CRAN

Date/Publication 2020-01-10 10:50:03 UTC

R topics documented:

`addID` *Populate the ID column of VCF data*

Description

Populate the ID column of VCF data by concatenating the chromosome, position and optionally an index.

Usage

```
addID(x, sep = "_")
```

Arguments

`x`	an object of class vcfR or chromR.
`sep`	a character string to separate the terms.

Details

Variant callers typically leave the ID column empty in VCF data. However, the VCF data may potentially include variants with IDs as well as variants without. This function populates the missing elements by concatenating the chromosome and position. When this concatenation results in non-unique names, an index is added to force uniqueness.

Examples

```
data(vcfR_test)
head(vcfR_test)
vcfR_test <- addID(vcfR_test)
head(vcfR_test)
```

AD_frequency *AD_frequency*

Description

Create allele frequencies from matrices of allelic depths (AD)

Usage

```
AD_frequency(ad, delim = ",", allele = 1L, sum_type = 0L, decreasing = 1L)
```

Arguments

ad	a matrix of allele depths (e.g., "7,2")
delim	character that delimits values
allele	which (1-based) allele to report frequency for
sum_type	type of sum to calculate, see details
decreasing	should the values be sorted decreasing (1) or increasing (0)?

Details

Files containing VCF data frequently include data on allelic depth (e.g., AD). This is the number of times each allele has been sequenced. Our naive assumption for diploids is that these alleles should be observed at a frequency of 1 or zero for homozygous positions and near 0.5 for heterozygous positions. Deviations from this expectation may indicate allelic imbalance or ploidy differences. This function is intended to facilitate the exploration of allele frequencies for all positions in a sample.

The alleles are sorted by their frequency within the function. The user can then specify is the would like to calculate the frequency of the most frequent allele (allele = 1), the second most frequent allele (allele = 2) and so one. If an allele is requested that does not exist it should result in NA for that position and sample.

There are two methods to calculate a sum for the denominator of the frequency. When sum_type = 0 the alleles are sorted decendingly and the first two allele counts are used for the sum. This may be useful when a state of diploidy may be known to be appropriate and other alleles may be interpreted as erroneous. When sum_type = 1 a sum is taken over all the observed alleles for a variant.

Value

A numeric matrix of frequencies

Examples

```
set.seed(999)
x1 <- round(rnorm(n=9, mean=10, sd=2))
x2 <- round(rnorm(n=9, mean=20, sd=2))
ad <- matrix(paste(x1, x2, sep=","), nrow=3, ncol=3)
colnames(ad) <- paste('Sample', 1:3, sep="_")
rownames(ad) <- paste('Variant', 1:3, sep="_")
ad[1,1] <- "9,23,12"
AD_frequency(ad=ad)
```

---

check_keys *Check that INFO and FORMAT keys are unique*

Description

The INFO and FORMAT columns contain information in key-value pairs. If for some reason a key is not unique it will create issues in retrieving this information. This function checks the keys defined in the meta section to make sure they are unique. Note that it does not actually check the INFO and FORMAT columns, just their definitions in the meta section. This is because each variant can have different information in their INFO and META cells. Checking these on large files will tehrefore come with a performance cost.

Usage

```
check_keys(x)
```

Arguments

x	an oblect of class vcfR

See Also

queryMETA()

Examples

```
data(vcfR_test)
check_keys(vcfR_test)
queryMETA(vcfR_test)
queryMETA(vcfR_test, element = 'DP')
# Note that DP occurs as unique in INFO and FORMAT but they may be different.
```

chromo_plot *Plot chromR object*

Description

plot chromR objects

Usage

```
chromo(
  chrom,
  boxp = TRUE,
  dp.alpha = TRUE,
  chrom.s = 1,
  chrom.e = NULL,
  drlist1 = NULL,
  drlist2 = NULL,
  drlist3 = NULL,
  ...
)

chromoqc(chrom, boxp = TRUE, dp.alpha = 255, ...)
```

Arguments

chrom	an object of class chrom.
boxp	logical specifying whether marginal boxplots should be plotted [T/F].
dp.alpha	degree of transparency applied to points in dot plots [0-255].
chrom.s	start position for the chromosome. (Deprecated. use xlim)
chrom.e	end position for the chromosome. (Deprecated. use xlim)
drlist1	a named list containing elements to create a drplot
drlist2	a named list containing elements to create a drplot
drlist3	a named list containing elements to create a drplot
...	arguments to be passed to other methods.

Details

Each **drlist** parameter is a list containing elements necessarry to plot a dr.plot. This list should contain up to seven elements named title, dmat, rlist, dcol, rcol, rbcol and bwcol. These elements are documented in the dr.plot page where they are presented as individual parameters. The one exception is bwcol which is a vector of colors for the marginal box and whisker plot. This is provided so that different colors may be used in the dot plot and the box and whisker plot. For example, transparency may be desired in the dot plot but not the box and whisker plot. When one (or more) of these elements is omitted an attempt to use default values is made.

Value

Returns an invisible NULL.

See Also

dr.plot

---

chromR functions *chromR_functions*

Description

Functions which act on chromR objects

Usage

```
masker(
  x,
  min_QUAL = 1,
  min_DP = 1,
  max_DP = 10000,
  min_MQ = 20,
  max_MQ = 100,
  preserve = FALSE,
  ...
)

variant.table(x)

win.table(x)
```

Arguments

x	object of class chromR
min_QUAL	minimum variant quality
min_DP	minimum cumulative depth
max_DP	maximum cumulative depth
min_MQ	minimum mapping quality
max_MQ	maximum mapping quality
preserve	a logical indicating whether or not to preserve the state of the current mask field. Defaults to FALSE
...	arguments to be passed to methods

Details

The function **masker** creates a logical vector that determines which variants are masked. By masking certain variants, instead of deleting them, it preserves the dimensions of the data structure until a change needs to be committed. Variants can be masked based on the value of the QUAL column of the vcf object. Experience seems to show that this value is either at its maximum (999) or a rather low value. The maximum and minimum sequence depth can also be used (mindp and maxdp). The default is to mask all variants with depths of less than the 0.25 quantile and greater than the 0.75 quantile (these are also known as the lower and upper quartile). The minimum and maximum mapping qualities (minmq, maxmq) can also be used.

This vector is stored in the var.info$mask slot of a chromR object.

The function **variant.table** creates a data.frame containing information about variants.

The funciton **win.table**

`chromR-class` *chromR class*

Description

A class for representing chromosomes (or supercontigs, contigs, scaffolds, etc.).

Details

Defines a class for chromosomal or contig data. This

This object has a number of slots.

- **name** name of the object (character)
- **len** length of the sequence (integer)
- **window_size** window size for windowing analyses (integer)
- **seq** object of class ape::DNAbin
- **vcf** object of class vcfR
- **ann** annotation data in a gff-like data.frame
- **var.info** a data.frame containing information on variants
- **win.info** a data.frame containing information on windows
- **seq.info** a list containing information on the sequence

The **seq** slot contains an object of class ape::DNAbin. A DNAbin object is typically either a matrix or list of DNAbin objects. The matrix form appears to be better behaved than the list form. Because of this behavior this slot should be the matrix form. When this slot is not populated it is of class "NULL" instead of "DNAbin". Note that characters need to be lower case when inserted into an object of class DNAbin. The function `tolower` can facilitate this.

The **vcf** slot is an object of class vcfR `vcfR-class`.

The **ann** slot is a data.frame containing gff format data. When this slot is not populated it has nrows equal to zero.

The **var.info** slot contains a data.frame containing information about variants. Every row of this data.frame is a variant. Columns will typically contain the chromosome name, the position of the variant (POS), the mask as well as any other per variant information.

The **win.info** slot contains a data.frame containing information about windows. For example, window, start, end, length, A, C, G, T, N, other, variants and genic fields are stored here.

The **seq.info** slot is a list containing two matrices. The first matrix describes rectangles for called nucleotides and the second describes rectangles for 'N' calls. Within each matrix, the first column indicates the start position and the second column indicates the end position of each rectangle.

See Also

vcfR-class, DNAbin, VCF specification gff3 format

---

chromR2vcfR *Convert chrom objects to vcfR objects*

Description

Convert chrom objects to vcfR objects.

Usage

```
chromR2vcfR(x, use.mask = FALSE)
```

Arguments

x	Object of class chrom
use.mask	Logical, determine if mask from chrom object should be used to subset vcf data

Details

The chrom object is subset and recast as a vcfR object. When use.mask is set to TRUE (the default), the object is subset to only the variants (rows) indicated to include by the mask. When use.mask is set to FALSE, all variants (rows) from the chrom object are included in the new vcfR object.

Value

Returns an object of class vcfR.

`chromR_example` *Example chromR object.*

Description

An example chromR object containing parts of the *Phytophthora infestans* genome.

Format

A chromR object

Details

This data is a subset of the pinfsc50 dataset. It has been subset to positions between 500 and 600 kbp. The coordinate systems of the vcf and gff file have been altered by subtracting 500,000. This results in a 100 kbp section of supercontig_1.50 that has positional data ranging from 1 to 100 kbp.

Examples

```
data(chromR_example)
```

`Convert to tidy data frames`

Convert vcfR objects to tidy data frames

Description

Convert the information in a vcfR object to a long-format data frame suitable for analysis or use with Hadley Wickham's packages, dplyr, tidyr, and ggplot2. These packages have been optimized for operation on large data frames, and, though they can bog down with very large data sets, they provide a good framework for handling and filtering large variant data sets. For some background on the benefits of such "tidy" data frames, see this article.

For some filtering operations, such as those where one wants to filter genotypes upon GT fields in combination with INFO fields, or more complex operations in which one wants to filter loci based upon the number of individuals having greater than a certain quality score, it will be advantageous to put all the information into a long format data frame and use `dplyr` to perform the operations. Additionally, a long data format is required for using `ggplot2`. These functions convert vcfR objects to long format data frames.

Usage

```
vcfR2tidy(
  x,
  info_only = FALSE,
  single_frame = FALSE,
  toss_INFO_column = TRUE,
  ...
)

extract_info_tidy(x, info_fields = NULL, info_types = TRUE, info_sep = ";")

extract_gt_tidy(
  x,
  format_fields = NULL,
  format_types = TRUE,
  dot_is_NA = TRUE,
  alleles = TRUE,
  allele.sep = "/",
  gt_column_prepend = "gt_",
  verbose = TRUE
)

vcf_field_names(x, tag = "INFO")
```

Arguments

`x`	an object of class vcfR
`info_only`	if TRUE return a list with only a `fix` component (a single data frame that has the parsed INFO information) and a `meta` component. Don't extract any of the FORMAT fields.
`single_frame`	return a single tidy data frame in list component `dat` rather returning it in components `fix` and/or `gt`.
`toss_INFO_column`	if TRUE (the default) the INFO column will be removed from output as its consituent parts will have been parsed into separate columns.
`...`	more options to pass to `extract_info_tidy` and `extract_gt_tidy`. See parameters listed below.
`info_fields`	names of the fields to be extracted from the INFO column into a long format data frame. If this is left as NULL (the default) then the function returns a column for every INFO field listed in the metadata.
`info_types`	named vector of "i" or "n" if you want the fields extracted from the INFO column to be converted to integer or numeric types, respectively. When set to NULL they will be characters. The names have to be the exact names of the fields. For example `info_types = c(AF = "n",DP = "i")` will convert column AF to numeric and DP to integer. If you would like the function to try to figure out the conversion from the metadata information, then set `info_types = TRUE`.

Anything with Number == 1 and (Type == Integer or Type == Numeric) will then be converted accordingly.

`info_sep` the delimiter used in the data portion of the INFO fields to separate different entries. By default it is ";", but earlier versions of the VCF standard apparently used ":" as a delimiter.

`format_fields` names of the fields in the FORMAT column to be extracted from each individual in the vcfR object into a long format data frame. If left as NULL, the function will extract all the FORMAT columns that were documented in the meta section of the VCF file.

`format_types` named vector of "i" or "n" if you want the fields extracted according to the FORMAT column to be converted to integer or numeric types, respectively. When set to TRUE an attempt to determine their type will be made from the meta information. When set to NULL they will be characters. The names have to be the exact names of the format_fields. Works equivalently to the `info_types` argument in `extract_info_tidy`, i.e., if you set it to TRUE then it uses the information in the meta section of the VCF to coerce to types as indicated.

`dot_is_NA` if TRUE then a single "." in a character field will be set to NA. If FALSE no conversion is done. Note that "." in a numeric or integer field (according to format_types) with Number == 1 is always going to be set to NA.

`alleles` if TRUE (the default) then this will return a column, `gt_GT_alleles` that has the genotype of the individual expressed as the alleles rather than as 0/1.

`allele.sep` character which delimits the alleles in a genotype (/ or |) to be passed to `extract.gt`. Here this is not used for a regex (as it is in other functions), but merely for output formatting.

`gt_column_prepend`

string to prepend to the names of the FORMAT columns

`verbose` logical to specify if verbose output should be produced in the output so that they do not conflict with any INFO columns in the output. Default is "gt_". Should be a valid R name. (i.e. don't start with a number, have a space in it, etc.)

`tag` name of the lines in the metadata section of the VCF file to parse out. Default is "INFO". The only other one tested and supported, currently is, "FORMAT".

Details

The function **vcfR2tidy** is the main function in this series. It takes a vcfR object and converts the information to a list of long-format data frames. The user can specify whether only the INFO or both the INFO and the FORMAT columns should be extracted, and also which INFO and FORMAT fields to extract. If no specific INFO or FORMAT fields are asked for, then they will all be returned. If `single_frame == FALSE` and `info_only == FALSE` (the default), the function returns a list with three components: `fix`, `gt`, and `meta` as follows:

1. `fix` A data frame of the fixed information columns and the parsed INFO columns, and an additional column, `ChromKey`—an integer identifier for each locus, ordered by their appearance in the original data frame—that serves together with POS as a key back to rows in `gt`.
2. `gt` A data frame of the genotype-related fields. Column names are the names of the FORMAT fields with `gt_column_prepend` (by default, "gt_") prepended to them. Additionally there are columns `ChromKey`, and `POS` that can be used to associate each row in `gt` with a row in `fix`.

3. `meta` The meta-data associated with the columns that were extracted from the INFO and FORMAT columns in a tbl_df-ed data frame.

This is the default return object because it might be space-inefficient to return a single tidy data frame if there are many individuals and the CHROM names are long and/or there are many INFO fields. However, if `single_frame = TRUE`, then the results are returned as a list with component `meta` as before, but rather than having `fix` and `gt` as before, both those data frames have been joined into component `dat` and a ChromKey column is not returned, because the CHROM column is available.

If `info_only == FALSE`, then just the fixed columns and the parsed INFO columns are returned, and the FORMAT fields are not parsed at all. The return value is a list with components `fix` and `meta`. No column ChromKey appears.

The following functions are called by **vcfR2tidy** but are documented below because they may be useful individually.

The function **extract_info_tidy** let's you pass in a vector of the INFO fields that you want extracted to a long format data frame. If you don't tell it which fields to extract it will extract all the INFO columns detailed in the VCF meta section. The function returns a tbl_df data frame of the INFO fields along with with an additional integer column `Key` that associates each row in the output data frame with each row (i.e. each CHROM-POS combination) in the original vcfR object `x`.

The function **extract_gt_tidy** let's you pass in a vector of the FORMAT fields that you want extracted to a long format data frame. If you don't tell it which fields to extract it will extract all the FORMAT columns detailed in the VCF meta section. The function returns a tbl_df data frame of the FORMAT fields with an additional integer column `Key` that associates each row in the output data frame with each row (i.e. each CHROM-POS combination), in the original vcfR object `x`, and an additional column `Indiv` that gives the name of the individual.

The function **vcf_field_names** is a helper function that parses information from the metadata section of the VCF file to return a data frame with the *metadata* information about either the INFO or FORMAT tags. It returns a `tbl_df`-ed data frame with column names: "Tag", "ID", "Number","Type", "Description", "Source", and "Version".

Value

An object of class tidy::data_frame or a list where every element is of class tidy::data_frame.

Note

To run all the examples, you can issue this: `example("vcfR2tidy")`

Author(s)

Eric C. Anderson <eric.anderson@noaa.gov>

See Also

dplyr, tidyr.

Examples

```
# load the data
data("vcfR_test")
vcf <- vcfR_test


# extract all the INFO and FORMAT fields into a list of tidy
# data frames: fix, gt, and meta. Here we don't coerce columns
# to integer or numeric types...
Z <- vcfR2tidy(vcf)
names(Z)


# here is the meta data in a table
Z$meta


# here is the fixed info
Z$fix


# here are the GT fields.  Note that ChromKey and POS are keys
# back to Z$fix
Z$gt


# Note that if you wanted to tidy this data set even further
# you could break up the comma-delimited columns easily
# using tidyr::separate




# here we put the data into a single, joined data frame (list component
# dat in the returned list) and the meta data.  Let's just pick out a
# few fields:
vcfR2tidy(vcf,
          single_frame = TRUE,
          info_fields = c("AC", "AN", "MQ"),
          format_fields = c("GT", "PL"))


# note that the "gt_GT_alleles" column is always returned when any
# FORMAT fields are extracted.




# Here we extract a single frame with all fields but we automatically change
# types of the columns according to the entries in the metadata.
vcfR2tidy(vcf, single_frame = TRUE, info_types = TRUE, format_types = TRUE)
```

```
# for comparison, here note that all the INFO and FORMAT fields that were
# extracted are left as character ("chr" in the dplyr summary)
vcfR2tidy(vcf, single_frame = TRUE)




# Below are some examples with the vcfR2tidy "subfunctions"


# extract the AC, AN, and MQ fields from the INFO column into
# a data frame and convert the AN values integers and the MQ
# values into numerics.
extract_info_tidy(vcf, info_fields = c("AC", "AN", "MQ"), info_types = c(AN = "i", MQ = "n"))

# extract all fields from the INFO column but leave
# them as character vectors
extract_info_tidy(vcf)

# extract all fields from the INFO column and coerce
# types according to metadata info
extract_info_tidy(vcf, info_types = TRUE)

# get the INFO field metadata in a data frame
vcf_field_names(vcf, tag = "INFO")

# get the FORMAT field metadata in a data frame
vcf_field_names(vcf, tag = "FORMAT")
```

`create.chromR` *Create chromR object*

Description

Creates and populates an object of class chromR.

Usage

```
create.chromR(vcf, name = "CHROM", seq = NULL, ann = NULL, verbose = TRUE)

vcfR2chromR(x, vcf)

seq2chromR(x, seq = NULL)
```

```
ann2chromR(x, gff)
```

Arguments

vcf	an object of class vcfR
name	a name for the chromosome (for plotting purposes)
seq	a sequence as a DNAbin object
ann	an annotation file (gff-like)
verbose	should verbose output be printed to the console?
x	an object of class chromR
gff	a data.frame containing annotation data in the gff format

Details

Creates and names a chromR object from a name, a chromosome (an ape::DNAbin object), variant data (a vcfR object) and annotation data (gff-like). The function **create.chromR** is a wrapper which calls functions to populate the slots of the chromR object.

The function **vcf2chromR** is called by create.chromR and transfers the data from the slots of a vcfR object to the slots of a chromR object. It also tries to extract the 'DP' and 'MQ' fileds (when present) from the fix slot's INFO column. It is not anticipated that a user would need to use this function directly, but its placed here in case they do.

The function **seq2chromR** is currently defined as a generic function. This may change in the future. This function takes an object of class DNAbin and assigns it to the 'seq' slot of a chromR object.

The function **ann2chromR** is called by create.chromR and transfers the information from a gff-like object to the 'ann' slot of a chromR object. It is not anticipated that a user would need to use this function directly, but its placed here in case they do.

See Also

chromR-class, vcfR-class, DNAbin, VCF specification gff3 format

Examples

```
library(vcfR)
data(vcfR_example)
chrom <- create.chromR('sc50', seq=dna, vcf=vcf, ann=gff)
head(chrom)
chrom
plot(chrom)

chrom <- masker(chrom, min_QUAL = 1, min_DP = 300, max_DP = 700, min_MQ = 59, max_MQ = 61)
chrom <- proc.chromR(chrom, win.size=1000)

plot(chrom)
chromoqc(chrom)
```

dr.plot elements *dr.plot elements*

Description

Plot chromR objects and their components

Usage

```
dr.plot(
  dmat = NULL,
  rlst = NULL,
  chrom.s = 1,
  chrom.e = NULL,
  title = NULL,
  hline = NULL,
  dcol = NULL,
  rcol = NULL,
  rbcol = NULL,
  ...
)

null.plot()
```

Arguments

dmat	a numeric matrix for dot plots where the first column is position (POS) and subsequent columns are y-values.
rlst	a list containing numeric matrices containing rectangle coordinates.
chrom.s	start position for the chromosome
chrom.e	end position for the chromosome
title	optional string to be used for the plot title.
hline	vector of positions to be used for horizontal lines.
dcol	vector of colors to be used for dot plots.
rcol	vector of colors to be used for rectangle plots.
rbcol	vector of colors to be used for rectangle borders.
...	arguments to be passed to other methods.

Details

Plot details The parameter **rlist** is list of numeric matrices containing rectangle coordinates. The first column of each matrix is the left positions, the second column is the bottom coordinates, the third column is the right coordinates and the fourth column is the top coordinates.

Value

Returns the y-axis minimum and maximum values invisibly.

See Also

rect chromo

---

extract.gt *Extract elements from vcfR objects*

Description

Extract elements from the 'gt' slot, convert extracted genotypes to their allelic state, extract indels from the data structure or extract elements from the INFO column of the 'fix' slot.

Usage

```
extract.gt(
  x,
  element = "GT",
  mask = FALSE,
  as.numeric = FALSE,
  return.alleles = FALSE,
  IDtoRowNames = TRUE,
  extract = TRUE,
  convertNA = TRUE
)

extract.haps(x, mask = FALSE, unphased_as_NA = TRUE, verbose = TRUE)

is.indel(x)

extract.indels(x, return.indels = FALSE)

extract.info(x, element, as.numeric = FALSE, mask = FALSE)
```

Arguments

x	An object of class chromR or vcfR
element	element to extract from vcf genotype data. Common options include "DP", "GT" and "GQ"
mask	a logical indicating whether to apply the mask (TRUE) or return all variants (FALSE). Alternatively, a vector of logicals may be provided.
as.numeric	logical, should the matrix be converted to numerics
return.alleles	logical indicating whether to return the genotypes (0/1) or alleles (A/T)

IDtoRowNames	logical specifying whether to use the ID column from the FIX region as row-names
extract	logical indicating whether to return the extracted element or the remaining string
convertNA	logical indicating whether to convert "." to NA.
unphased_as_NA	logical specifying how to handle unphased genotypes
verbose	should verbose output be generated
return.indels	logical indicating whether to return indels or not

Details

The function **extract.gt** isolates elements from the 'gt' portion of vcf data. Fields available for extraction are listed in the FORMAT column of the 'gt' slot. Because different vcf producing software produce different fields the options will vary by software. The mask parameter allows the mask to be implemented when using a chromR object. The 'as.numeric' option will convert the results from a character to a numeric. Note that if the data is not actually numeric, it will result in a numeric result which may not be interpretable. The 'return.alleles' option allows the default behavior of numerically encoded genotypes (e.g., 0/1) to be converted to their nucleic acid representation (e.g., A/T). Note that this is not used for a regular expression as similar parameters are used in other functions. Extract allows the user to extract just the specified element (TRUE) or every element except the one specified.

Note that when 'as.numeric' is set to 'TRUE' but the data are not actually numeric, unexpected results will likely occur. For example, the genotype field will typically be populated with values such as "0/1" or "1|0". Although these may appear numeric, they contain a delimiter (the forward slash or the pipe) that is non-numeric. This means that there is no straight forward conversion to a numeric and unexpected values should be expected.

The function **extract.haps** uses extract.gt to isolate genotypes. It then uses the information in the REF and ALT columns as well as an allele delimiter (gt_split) to split genotypes into their allelic state. Ploidy is determined by the first non-NA genotype in the first sample.

The VCF specification allows for genotypes to be delimited with a '|' when they are phased and a '/' when unphased. This becomes important when dividing a genotype into two haplotypes. When the alleels are phased this is straight forward. When the alleles are unphased it presents a decision. The default is to handle unphased data by converting them to NAs. When unphased_as_NA is set to TRUE the alleles will be returned in the order they appear in the genotype. This does not assign each allele to it's correct chromosome. It becomes the user's responsibility to make informed decisions at this point.

The function **is.indel** returns a logical vector indicating which variants are indels (variants where an allele is greater than one character).

The function **extract.indels** is used to remove indels from SNPs. The function queries the 'REF' and 'ALT' columns of the 'fix' slot to see if any alleles are greater than one character in length. When the parameter return_indels is FALSE only SNPs will be returned. When the parameter return_indels is TRUE only indels will be returned.

The function **extract.info** is used to isolate elements from the INFO column of vcf data.

See Also

is.polymorphic

Examples

```
data(vcfR_test)
gt <- extract.gt(vcfR_test)
gt <- extract.gt(vcfR_test, return.alleles = TRUE)

data(vcfR_test)
is.indel(vcfR_test)


data(vcfR_test)
getFIX(vcfR_test)
vcf <- extract.indels(vcfR_test)
getFIX(vcf)
vcf@fix[nrow(vcf@fix),'ALT'] <- ".,A"
vcf <- extract.indels(vcf)
getFIX(vcf)

data(vcfR_test)
vcfR_test@fix[1,'ALT'] <- "<NON_REF>"
vcf <- extract.indels(vcfR_test)
getFIX(vcf)

data(vcfR_test)
extract.haps(vcfR_test, unphased_as_NA = FALSE)
extract.haps(vcfR_test)
```

Format conversion *Convert vcfR objects to other formats*

Description

Convert vcfR objects to objects supported by other R packages

Usage

```
vcfR2genind(x, sep = "[|/]", return.alleles = FALSE, ...)

vcfR2loci(x, return.alleles = FALSE)

vcfR2genlight(x, n.cores = 1)
```

Arguments

x	an object of class chromR or vcfR
sep	character (to be used in a regular expression) to delimit the alleles of genotypes
return.alleles	should the VCF encoding of the alleles be returned (FALSE) or the actual alleles (TRUE).

`...`	pass other parameters to adegenet::df2genlight
`n.cores`	integer specifying the number of cores to use.

Details

After processing vcf data in vcfR, one will likely proceed to an analysis step. Within R, three obvious choices are: pegas, adegenet and poppr. The package pegas uses objects of type loci. The function **vcfR2loci** calls extract.gt to create a matrix of genotypes which is then converted into an object of type loci.

The packages adegenet and poppr use the genind object. The function **vcfR2genind** uses extract.gt to create a matrix of genotypes and uses the adegenet function df2genind to create a genind object. The package poppr additionally uses objects of class genclone which can be created from genind objects using poppr::as.genclone. A genind object can be converted to a genclone object with the function poppr::as.genclone.

The function vcfR2genlight calls the 'new' method for the genlight object. This method implements multi-threading through calls to the function `parallel::mclapply`. Because 'forks' do not exist in the windows environment, this will only work for windows users when n.cores=1. In the Unix environment, users may increase this number to allow the use of multiple threads (i.e., cores).

The parameter **...** is used to pass parameters to other functions. In `vcfR2genind` it is used to pass parameters to `adegenet::df2genind`. For example, setting `check.ploidy=FALSE` may improve the performance of `adegenet::df2genind`, as long as you know the ploidy. See `?adegenet::df2genind` to see these options.

Note

For users of poppr: If you wish to use `vcfR2genind()`, it is **strongly recommended** to use it with the option `return.alleles = TRUE`. The reason for this is because the **poppr** package accomodates mixed-ploidy data by interpreting "0" alleles *in genind objects* to be NULL alleles in both `poppr::poppr.amova()` and `poppr::locus_table()`.

See Also

`extract.gt`, `alleles2consensus`, `adegenet::df2genind`, `adegenet::genind`, pegas, adegenet, and poppr. To convert to objects of class **DNAbin** see `vcfR2DNAbin`.

Examples

```
adegenet_installed <- require("adegenet")
if (adegenet_installed) {
  data(vcfR_test)
  # convert to genlight (preferred method with bi-allelic SNPs)
  gl <- vcfR2genlight(vcfR_test)

  # convert to genind, keeping information about allelic state
  # (slightly slower, but preferred method for use with the "poppr" package)
  gid <- vcfR2genind(vcfR_test, return.alleles = TRUE)

  # convert to genind, returning allelic states as 0, 1, 2, etc.
  # (not preferred, but slightly faster)
```

```
  gid2 <- vcfR2genind(vcfR_test, return.alleles = FALSE)
}
```

---

freq_peak *freq_peak*

Description

Find density peaks in frequency data.

Usage

```
freq_peak(myMat, pos, winsize = 10000L, bin_width = 0.02, lhs = TRUE)
```

Arguments

myMat	a matrix of frequencies [0-1].
pos	a numeric vector describing the position of variants in myMat.
winsize	sliding window size.
bin_width	Width of bins to summarize ferequencies in (0-1].
lhs	logical specifying whether the search for the bin of greatest density should favor values from the left hand side.

Details

Noisy data, such as genomic data, lack a clear consensus. Summaries may be made in an attempt to 'clean it up.' Common summaries, such as the mean, rely on an assumption of normalicy. An assumption that frequently can be violated. This leaves a conundrum as to how to effectively summarize these data.

Here we implement an attempt to summarize noisy data through binning the data and selecting the bin containing the greatest density of data. The data are first divided into parameter sized windows. Next the data are categorized by parameterizable bin widths. Finally, the bin with the greatest density, the greatest count of data, is used as a summary. Because this method is based on binning the data it does not rely on a distributional assumption.

The parameter `lhs` specifyies whether the search for the bin of greatest density should be performed from the left hand side. The default value of TRUE starts at the left hand side, or zero, and selects a new bin as having the greatest density only if a new bin has a greater density. If the new bin has an equal density then no update is made. This causees the analysis to select lower frequencies. When this parameter is set to FALSE ties result in an update of the bin of greatest density. This causes the analysis to select higher frequencies. It is recommended that when testing the most abundant allele (typically [0.5-1]) to use the default of TRUE so that a low value is preferred. Similarly, when testing the less abundant alleles it is recommended to set this value at FALSE to preferentially select high values.

Value

A freq_peak object (a list) containing:

- The window size
- The binwidth used for peak binning
- a matrix containing window coordinates
- a matrix containing peak locations
- a matrix containing the counts of variants for each sample in each window

The window matrix contains start and end coordinates for each window, the rows of the original matrix that demarcate each window and the position of the variants that begin and end each window.

The matrix of peak locations contains the midpoint for the bin of greatest density for each sample and each window. Alternatively, if 'count = TRUE' the number of non-missing values in each window is reported. The number of non-mising values in each window may be used to censor windows containing low quantities of data.

See Also

peak_to_ploid, freq_peak_plot

Examples

```
data(vcfR_example)
gt <- extract.gt(vcf)
hets <- is_het(gt)
# Censor non-heterozygous positions.
is.na(vcf@gt[,-1][!hets]) <- TRUE
# Extract allele depths.
ad <- extract.gt(vcf, element = "AD")
ad1 <- masplit(ad, record = 1)
ad2 <- masplit(ad, record = 2)
freq1 <- ad1/(ad1+ad2)
freq2 <- ad2/(ad1+ad2)
myPeaks1 <- freq_peak(freq1, getPOS(vcf))
is.na(myPeaks1$peaks[myPeaks1$counts < 20]) <- TRUE
myPeaks2 <- freq_peak(freq2, getPOS(vcf), lhs = FALSE)
is.na(myPeaks2$peaks[myPeaks2$counts < 20]) <- TRUE
myPeaks1

# Visualize
mySample <- "P17777us22"
myWin <- 2
hist(freq1[myPeaks1$wins[myWin,'START_row']:myPeaks1$wins[myWin,'END_row'], mySample],
     breaks=seq(0,1,by=0.02), col="#A6CEE3", main="", xlab="", xaxt="n")
hist(freq2[myPeaks2$wins[myWin,'START_row']:myPeaks2$wins[myWin,'END_row'], mySample],
     breaks=seq(0,1,by=0.02), col="#1F78B4", main="", xlab="", xaxt="n", add = TRUE)
axis(side=1, at=c(0,0.25,0.333,0.5,0.666,0.75,1),
     labels=c(0,'1/4','1/3','1/2','2/3','3/4',1), las=3)
abline(v=myPeaks1$peaks[myWin,mySample], col=2, lwd=2)
abline(v=myPeaks2$peaks[myWin,mySample], col=2, lwd=2)
```

```
# Visualize #2
mySample <- "P17777us22"
plot(getPOS(vcf), freq1[,mySample], ylim=c(0,1), type="n", yaxt='n',
     main = mySample, xlab = "POS", ylab = "Allele balance")
axis(side=2, at=c(0,0.25,0.333,0.5,0.666,0.75,1),
     labels=c(0,'1/4','1/3','1/2','2/3','3/4',1), las=1)
abline(h=c(0.25,0.333,0.5,0.666,0.75), col=8)
points(getPOS(vcf), freq1[,mySample], pch = 20, col= "#A6CEE3")
points(getPOS(vcf), freq2[,mySample], pch = 20, col= "#1F78B4")
segments(x0=myPeaks1$wins[,'START_pos'], y0=myPeaks1$peaks[,mySample],
         x1=myPeaks1$wins[,'END_pos'], lwd=3)
segments(x0=myPeaks1$wins[,'START_pos'], y0=myPeaks2$peaks[,mySample],
         x1=myPeaks1$wins[,'END_pos'], lwd=3)
```

`freq_peak_plot` *Plot freq_peak object*

Description

Converts allele balance data produced by `freq_peak()` to a copy number by assinging the allele balance data (frequencies) to its closest expected ratio.

Usage

```
freq_peak_plot(
  pos,
  posUnits = "bp",
  ab1 = NULL,
  ab2 = NULL,
  fp1 = NULL,
  fp2 = NULL,
  mySamp = 1,
  col1 = "#A6CEE3",
  col2 = "#1F78B4",
  alpha = 44,
  main = NULL,
  mhist = TRUE,
  layout = TRUE,
  ...
)
```

Arguments

`pos`	chromosomal position of variants

posUnits	units ('bp', 'Kbp', 'Mbp', 'Gbp') for 'pos' to be converted to in the main plot
ab1	matrix of allele balances for allele 1
ab2	matrix of allele balances for allele 2
fp1	freq_peak object for allele 1
fp2	freq_peak object for allele 2
mySamp	sample indicator
col1	color 1
col2	color 2
alpha	sets the transparency for dot plot (0-255)
main	main plot title.
mhist	logical indicating to include a marginal histogram
layout	call layout
...	parameters passed on to other functions

Details

Creates a visualization of allele balance data consisting of a dot plot with position as the x-axis and frequency on the y-axis and an optional marginal histogram. The only required information is a vector of chromosomal positions, however this is probably not going to create an interesting plot.

Value

An invisible NULL.

See Also

freq_peak, peak_to_ploid

Examples

```
# An empty plot.
freq_peak_plot(pos=1:40)

data(vcfR_example)
gt <- extract.gt(vcf)
hets <- is_het(gt)
# Censor non-heterozygous positions.
is.na(vcf@gt[,-1][!hets]) <- TRUE
# Extract allele depths.
ad <- extract.gt(vcf, element = "AD")
ad1 <- masplit(ad, record = 1)
ad2 <- masplit(ad, record = 2)
freq1 <- ad1/(ad1+ad2)
freq2 <- ad2/(ad1+ad2)
myPeaks1 <- freq_peak(freq1, getPOS(vcf))
is.na(myPeaks1$peaks[myPeaks1$counts < 20]) <- TRUE
```

```
myPeaks2 <- freq_peak(freq2, getPOS(vcf), lhs = FALSE)
is.na(myPeaks2$peaks[myPeaks2$counts < 20]) <- TRUE
freq_peak_plot(pos = getPOS(vcf), ab1 = freq1, ab2 = freq2, fp1 = myPeaks1, fp2=myPeaks2)
```

`genetic_diff` *Genetic differentiation*

Description

Calculate measures of genetic differentiation.

Usage

```
genetic_diff(vcf, pops, method = "nei")
```

Arguments

`vcf`	a vcfR object
`pops`	factor indicating populations
`method`	the method to measure differentiation

Details

Measures of genetic differentiation, or fixation indicies, are commonly reported population genetic parameters. This function reports genetic differentiation for all variants presented to it.

The method **nei** returns Nei's Gst as well as Hedrick's G'st, a correction for high alleleism (Hedrick 2005). Here it is calculated as in equation 2 from Hedrick (2005) with the exception that the heterozygosities are weighted by the number of alleles observed in each subpopulation. This is similar to `hierfstat::pairwise.fst()` but by using the number of alleles instead of the number of individuals it avoids making an assumption about how many alleles are contributed by each individual. G'st is calculated as in equation 4b from Hedrick (2005). This method is based on heterozygosity where all of the alleles in a population are used to calculate allele frequecies. This may make this a good choice when there is a mixture of ploidies in the sample.

The method **jost** return's Jost's D as a measure of differentiation. This is calculated as in equation 13 from Jost (2008). Examples are available at Jost's website: http://www.loujost.com.

A nice review of Fst and some of its analogues can be found in Holsinger and Weir (2009).

References

Hedrick, Philip W. "A standardized genetic differentiation measure." Evolution 59.8 (2005): 1633-1638.

Holsinger, Kent E., and Bruce S. Weir. "Genetics in geographically structured populations: defining, estimating and interpreting FST." Nature Reviews Genetics 10.9 (2009): 639-650.

Jost, Lou. "GST and its relatives do not measure differentiation." Molecular ecology 17.18 (2008): 4015-4026.

Whitlock, Michael C. "G'ST and D do not replace FST." Molecular Ecology 20.6 (2011): 1083-1091.

See Also

poppr.amova in poppr, amova in ade4, amova in pegas, hierfstat, DEMEtics, and, mmod.

Examples

```
data(vcfR_example)
myPops <- as.factor(rep(c('a','b'), each = 9))
myDiff <- genetic_diff(vcf, myPops, method = "nei")
colMeans(myDiff[,c(3:8,11)], na.rm = TRUE)
hist(myDiff$Gprimest, xlab = expression(italic("G'"["ST"])),
     col='skyblue', breaks = seq(0, 1, by = 0.01))
```

`Genotype matrix functions`

Genotype matrix functions

Description

Functions which modify a matrix or vector of genotypes.

Usage

```
alleles2consensus(x, sep = "/", NA_to_n = TRUE)

get.alleles(x2, split = "/", na.rm = FALSE, as.numeric = FALSE)
```

Arguments

`x`	a matrix of alleles as genotypes (e.g., A/A, C/G, etc.)
`sep`	a character which delimits the alleles in a genotype (/ or |)
`NA_to_n`	logical indicating whether NAs should be scores as n
`x2`	a vector of genotypes
`split`	character passed to strsplit to split the genotype into alleles
`na.rm`	logical indicating whether to remove NAs
`as.numeric`	logical specifying whether to convert to a numeric

Details

The function **alleles2consensus** converts genotypes to a single consensus allele using IUPAC ambiguity codes for heterozygotes. Note that some functions, such as ape::seg.sites do not recognize ambiguity characters (other than 'n'). This means that these functions, as well as functions that depend on them (e.g., pegas::tajima.test), will produce unexpected results.

Missing data are handled in a number of steps. When both alleles are missing ('.') the genotype is converted to NA. Secondly, if one of the alleles is missing ('.') the genotype is converted to NA> Lastly, NAs can be optionally converted to 'n' for compatibility with DNAbin objects.

The function **get.alleles** takes a vector of genotypes and returns the unique alleles.

---

getFIX *Get elements from the fixed region of a VCF file*

Description

Both chromR objects and vcfR objects contain a region with fixed variables. These accessors allow you to isolate these variables from these objects.

Usage

```
getFIX(x, getINFO = FALSE)

getCHROM(x)

getPOS(x)

getQUAL(x)

getALT(x)

getREF(x)

getID(x)

getFILTER(x)

getINFO(x)
```

Arguments

x	a vcfR or chromR object
getINFO	logical specifying whether getFIX should return the INFO column

Value

a vector or data frame

Examples

```
library("vcfR")
data("vcfR_example")
data("chromR_example")
getFIX(vcf) %>% head
getFIX(chrom) %>% head

getCHROM(vcf) %>% head
getCHROM(chrom) %>% head

getPOS(vcf) %>% head
getPOS(chrom) %>% head

getID(vcf) %>% head
getID(chrom) %>% head

getREF(vcf) %>% head
getREF(chrom) %>% head

getALT(vcf) %>% head
getALT(chrom) %>% head

getQUAL(vcf) %>% head
getQUAL(chrom) %>% head

getFILTER(vcf) %>% head
getFILTER(chrom) %>% head

getINFO(vcf) %>% head
getINFO(chrom) %>% head
```

---

gt2popsum *Population genetics summaries*

Description

Functions that make population genetics summaries

Usage

```
gt2popsum(x, deprecated = TRUE)

gt.to.popsum(x)
```

Arguments

x	object of class chromR or vcfR
deprecated	logical specifying whether to run the function (FALSE) or present deprecation message (TRUE).

Details

The function ‘gt2popsum‘ was deprecated in vcfR 1.8.0. This was because it was written entirely in R and did not perform well. Users should use ‘gt.to.popsum()‘ instead because it has similar functionality but includes calls to C++ to increase its performance.

This function creates common population genetic summaries from either a chromR or vcfR object. The default is to return a matrix containing allele counts, He, and Ne. **Allele_counts** is the a comma delimited string of counts. The first position is the count of reference alleles, the second positions is the count of the first alternate alleles, the third is the count of second alternate alleles, and so on. **He** is the gene diversity, or heterozygosity, of the population. This is $1 - \sum x_i^2$, or the probability that two alleles sampled from the population are different, following Nei (1973). **Ne** is the effective number of alleles in the population. This is $1/\sum x_i^2$ or one minus the homozygosity, from Nei (1987) equation 8.17.

Nei, M., 1973. Analysis of gene diversity in subdivided populations. Proceedings of the National Academy of Sciences, 70(12), pp.3321-3323.

Nei, M., 1987. Molecular evolutionary genetics. Columbia University Press.

Examples

```
data(vcfR_test)
# Check the genotypes.
extract.gt(vcfR_test)
# Summarize the genotypes.
gt.to.popsum(vcfR_test)
```

---

heatmap.bp *Heatmap with barplots*

Description

Heatmap of a numeric matrix with barplots summarizing columns and rows.

Usage

```
heatmap.bp(
  x,
  cbarplot = TRUE,
  rbarplot = TRUE,
  legend = TRUE,
  clabels = TRUE,
  rlabels = TRUE,
  na.rm = TRUE,
  scale = c("row", "column", "none"),
  col.ramp = viridisLite::viridis(n = 100, alpha = 1),
  ...
)
```

Arguments

x	a numeric matrix.
cbarplot	a logical indicating whether the columns should be summarized with a barplot.
rbarplot	a logical indicating whether the rows should be summarized with a barplot.
legend	a logical indicating whether a legend should be plotted.
clabels	a logical indicating whether column labels should be included.
rlabels	a logical indicating whether row labels should be included.
na.rm	a logical indicating whether missing values should be removed.
scale	character indicating if the values should be centered and scaled in either the row direction or the column direction, or none. The default is "none".
col.ramp	vector of colors to be used for the color ramp.
...	additional arguments to be passed on.

Details

The function heatmap.bp creates a heatmap from a numeric matrix with optional barplots to summarize the rows and columns.

See Also

heatmap, image, heatmap2 in gplots, pheatmap.

Examples

```
library(vcfR)

x  <- as.matrix(mtcars)

heatmap.bp(x)
heatmap.bp(x, scale="col")
# Use an alternate color ramp
heatmap.bp(x, col.ramp = colorRampPalette(c("red", "yellow", "#008000"))(100))
heatmap.bp(x)

## Not run:
heatmap.bp(x, cbarplot = FALSE, rbarplot = FALSE, legend = FALSE)
heatmap.bp(x, cbarplot = FALSE, rbarplot = TRUE, legend = FALSE)
heatmap.bp(x, cbarplot = FALSE, rbarplot = FALSE, legend = TRUE)
heatmap.bp(x, cbarplot = FALSE, rbarplot = TRUE, legend = TRUE)

heatmap.bp(x, cbarplot = TRUE, rbarplot = FALSE, legend = FALSE)
heatmap.bp(x, cbarplot = TRUE, rbarplot = TRUE, legend = FALSE)
heatmap.bp(x, cbarplot = TRUE, rbarplot = FALSE, legend = TRUE)
heatmap.bp(x, cbarplot = TRUE, rbarplot = TRUE, legend = TRUE)

## End(Not run)
```

INFO2df *Reformat INFO data as a data.frame*

Description

Reformat INFO data as a data.frame and handle class when possible.

Usage

```
INFO2df(x)

metaINFO2df(x, field = "INFO")
```

Arguments

x	an object of class vcfR or chromR.
field	should either the INFo or FORMAT data be returned?

Details

The INFO column of VCF data contains descriptors for each variant. Because this column may contain many comma delimited descriptors it may be difficult to interpret. The function INFO2df converts the data into a data.frame. The function metaINFO2df extracts the information in the meta section that describes the INFO descriptors. This function is called by INFO2df to help it handle the class of the data.

Value

A data.frame

Examples

```
data(vcfR_test)
metaINFO2df(vcfR_test)
getINFO(vcfR_test)
INFO2df(vcfR_test)
```

is_het *Query genotypes for heterozygotes*

Description

Query a matrix of genotypes for heterozygotes

Usage

```
is_het(x, na_is_false = TRUE)

is.het(x, na_is_false = TRUE)
```

Arguments

x	a matrix of genotypes
na_is_false	should missing data be returned as NA (FALSE) or FALSE (TRUE)

Details

This function was designed to identify heterozygous positions in a matrix of genotypes. The matrix of genotypes can be created with extract.gt. Because the goal was to identify heterozygotes it may be reasonable to ignore missing values by setting na_is_false to TRUE so that the resulting matrix will consist of only TRUE and FALSE. In order to preserve missing data as missing na_is_false can be set to FALSE where if at least one allele is missing NA is returned.

See Also

extract.gt

Examples

```
data(vcfR_test)
gt <- extract.gt(vcfR_test)
hets <- is_het(gt)
# Censor non-heterozygous positions.
is.na(vcfR_test@gt[,-1][!hets]) <- TRUE
```

maf *Minor allele frequency*

Description

Calculate the minor (or other) allele frequency.

Usage

```
maf(x, element = 2)
```

Arguments

x	an object of class vcfR or chromR
element	specify the allele number to return

Details

The function maf() calculates the counts and frequency for an allele. A variant may contain more than two alleles. Rare alleles may be true rare alleles or the result of genotyping error. In an attempt to address these competing issues we sort the alleles by their frequency and the report statistics based on their position. For example, setting element=1 would return information about the major (most common) allele. Setting element=2 returns information about the second allele.

Value

a matrix of four columns. The first column is the total number of alleles, the second is the number of NA genotypes, the third is the count and fourth the frequency.

masplit *masplit*

Description

Split a matrix of delimited strings.

Usage

```
masplit(
  myMat,
  delim = ",",
  count = 0L,
  record = 1L,
  sort = 1L,
  decreasing = 1L
)
```

Arguments

myMat	a matrix of delimited strings (e.g., "7,2").
delim	character that delimits values.
count	return the count of delimited records.
record	which (1-based) record to return.
sort	should the records be sorted prior to selecting the element (0,1)?
decreasing	should the values be sorted decreasing (1) or increasing (0)?

Details

Split a matrix of delimited strings that represent numerics into numerics. The parameter **count** returns a matrix of integers indicating how many delimited records exist in each element. This is intended to help if you do not know how many records are in each element particularly if there is a mixture of numbers of records. The parameter **record** indicates which record to return (first, second, third, ...). The parameter **sort** indicates whether the records in each element should be sorted (1) or not (0) prior to selection. When sorting has been selected **decreasing** indicates if the sorting should be performed in a decreasing (1) or increasing (0) manner prior to selection.

Value

A numeric matrix

Examples

```
set.seed(999)
x1 <- round(rnorm(n=9, mean=10, sd=2))
x2 <- round(rnorm(n=9, mean=20, sd=2))
ad <- matrix(paste(x1, x2, sep=","), nrow=3, ncol=3)
colnames(ad) <- paste('Sample', 1:3, sep="_")
rownames(ad) <- paste('Variant', 1:3, sep="_")
ad[1,1] <- "9,23,12"
is.na(ad[3,1]) <- TRUE

ad
masplit(ad, count = 1)
masplit(ad, sort = 0)
masplit(ad, sort = 0, record = 2)
masplit(ad, sort = 0, record = 3)
masplit(ad, sort = 1, decreasing = 0)
```

`ordisample` *Ordinate a sample's data*

Description

Ordinate information from a sample's GT region and INFO column.

Usage

```
ordisample(
  x,
  sample,
  distance = "bray",
  plot = TRUE,
  alpha = 88,
  verbose = TRUE,
  ...
)
```

Arguments

`x`	an object of class vcfR or chromR.
`sample`	a sample number where the first sample (column) is 2
`distance`	metric to be used for ordination, options are in `vegdist`
`plot`	logical specifying whether to plot the ordination
`alpha`	alpha channel (transparency) ranging from 0-255
`verbose`	logical specifying whether to produce verbose output
`...`	parameters to be passed to child processes

Details

The INFO column of VCF data contains descriptors for each variant. Each sample typically includes several descriptors of each variant in the GT region as well. This can present an overwhelming amount of information. Ordination is used in this function to reduce this complexity.

The ordination procedure can be rather time consuming depending on how much data is used. I good recommendation is to always start with a small subset of your full dataset and slowly scale up. There are several steps in this function that attempt to eliminate variants or characters that have missing values in them. This that while starting with a small number is good, you will need to have a large enough number so that a substantial amount of the data make it to the ordination step. In the example I use 100 variants which appears to be a reasonable compromise.

The data contained in VCF files can frequently contain a large fraction of missing data. I advocate censoring data that does not meet quality control thresholds as missing which compounds the problem. An attempt is made to omit these missing data by querying the GT and INFO data for missingness and omitting the missing variants. The data may also include characters (columns) that

contain all missing values which are omitted as well. When verbose == TRUE these omissions are reported as messages.

Some data may contain multiple values. For example, AD is the sequence depth for each observed allele. In these instances the values are sorted and the largest value is used.

Several of the steps of this ordination make distributional assumptions. That is, they assume the data to be normally distributed. There is no real reason to assume this assumption to be valid with VCF data. It has been my experience that this assumption is frequently violated with VCF data. It is therefore suggested to use this funciton as an exploratory tool that may help inform other decisions. These analyst may be able to address these issues through data transformation or other topics beyond the scope of this function. This function is intended to provide a rapid assessment of the data which may help determine if more elegant handling of the data may be required. Interpretation of the results of this function need to take into account that assumptions may have been violated.

Value

A list consisting of two objects.

- an object of class 'metaMDS' created by the function vegan::metaMDS
- an object of class 'envfit' created by the function vegan::envfit

This list is returned invisibly.

See Also

metaMDS, vegdist, monoMDS, isoMDS

Examples

```
## Not run:

# Example of normally distributed, random data.
set.seed(9)
x1 <- rnorm(500)
set.seed(99)
y1 <- rnorm(500)
plot(x1, y1, pch=20, col="#8B451388", main="Normal, random, bivariate data")

data(vcfR_example)
ordisample(vcf[1:100,], sample = "P17777us22")

vars <- 1:100
myOrd <- ordisample(vcf[vars,], sample = "P17777us22", plot = FALSE)
names(myOrd)
plot(myOrd$metaMDS, type = "n")
points(myOrd$metaMDS, display = "sites", pch=20, col="#8B451366")
text(myOrd$metaMDS, display = "spec", col="blue")
plot(myOrd$envfit, col = "#008000", add = TRUE)
head(myOrd$metaMDS$points)
myOrd$envfit
pairs(myOrd$data1)
```

```
# Seperate heterozygotes and homozygotes.
gt <- extract.gt(vcf)
hets <- is_het(gt, na_is_false = FALSE)
vcfhe <- vcf
vcfhe@gt[,-1][ !hets & !is.na(hets)  ] <- NA
vcfho <- vcf
vcfho@gt[,-1][ hets & !is.na(hets) ] <- NA

myOrdhe <- ordisample(vcfhe[vars,], sample = "P17777us22", plot = FALSE)
myOrdho <- ordisample(vcfho[vars,], sample = "P17777us22", plot = FALSE)
pairs(myOrdhe$data1)
pairs(myOrdho$data1)
hist(myOrdho$data1$PL, breaks = seq(0,9000, by=100), col="#8B4513")

## End(Not run)
```

`pairwise_genetic_diff` *Pairwise genetic differentiation across populations*

Description

`pairwise_genetic_diff` Calculate measures of genetic differentiation across all population pairs.

Usage

```
pairwise_genetic_diff(vcf, pops, method = "nei")
```

Arguments

`vcf`	a vcfR object
`pops`	factor indicating populations
`method`	the method to measure differentiation

Value

a data frame containing the pairwise population differentiation indices of interest across all pairs of populations in the population factor.

Author(s)

Javier F. Tabima

See Also

`genetic_diff` in `vcfR`

Examples

```
data(vcfR_example)
pops <- as.factor(rep(c('a','b'), each = 9))
myDiff <- pairwise_genetic_diff(vcf, pops, method = "nei")
colMeans(myDiff[,c(4:ncol(myDiff))], na.rm = TRUE)
pops <- as.factor(rep(c('a','b','c'), each = 6))
myDiff <- pairwise_genetic_diff(vcf, pops, method = "nei")
colMeans(myDiff[,c(4:ncol(myDiff))], na.rm = TRUE)
```

---

peak_to_ploid *Convert allele balance peaks to ploidy*

Description

Converts allele balance data produced by `freq_peak()` to a copy number by assinging the allele balance data (frequencies) to its closest expected ratio.

Usage

```
peak_to_ploid(x)
```

Arguments

x	an object produced by `freq_peak()`.

Details

Converts allele balance data produced by `freq_peak()` to copy number. See the examples section for a graphical representation of the expectations and the bins around them. Once a copy number has called a distance from expectation (dfe) is calculated as a form of confidence. The bins around different copy numbers are of different width, so the dfe is scaled by its respective bin width. This results in a dfe that is 0 when it is exactly at our expectation (high confidence) and at 1 when it is half way between two expectations (low confidence).

Value

A list consisting of two matrices containing the calls and the distance from expectation (i.e., confidence).

See Also

`freq_peak`, `freq_peak_plot`

Examples

```
# Thresholds.
plot(c(0.0, 1), c(0,1), type = "n", xaxt = "n", xlab = "Expectation", ylab = "Allele balance")
myCalls <-  c(1/5, 1/4, 1/3, 1/2, 2/3, 3/4, 4/5)
axis(side = 1, at = myCalls, labels = c('1/5', '1/4', '1/3','1/2', '2/3', '3/4', '4/5'), las=2)
abline(v=myCalls)
abline(v=c(7/40, 9/40, 7/24, 5/12), lty=3, col ="#B22222")
abline(v=c(7/12, 17/24, 31/40, 33/40), lty=3, col ="#B22222")
text(x=7/40, y=0.1, labels = "7/40", srt = 90)
text(x=9/40, y=0.1, labels = "9/40", srt = 90)
text(x=7/24, y=0.1, labels = "7/24", srt = 90)
text(x=5/12, y=0.1, labels = "5/12", srt = 90)
text(x=7/12, y=0.1, labels = "7/12", srt = 90)
text(x=17/24, y=0.1, labels = "17/24", srt = 90)
text(x=31/40, y=0.1, labels = "31/40", srt = 90)
text(x=33/40, y=0.1, labels = "33/40", srt = 90)

# Prepare data and visualize
data(vcfR_example)
gt <- extract.gt(vcf)
# Censor non-heterozygous positions.
hets <- is_het(gt)
is.na(vcf@gt[,-1][!hets]) <- TRUE
# Extract allele depths.
ad <- extract.gt(vcf, element = "AD")
ad1 <- masplit(ad, record = 1)
ad2 <- masplit(ad, record = 2)
freq1 <- ad1/(ad1+ad2)
freq2 <- ad2/(ad1+ad2)
myPeaks1 <- freq_peak(freq1, getPOS(vcf))
# Censor windows with fewer than 20 heterozygous positions
is.na(myPeaks1$peaks[myPeaks1$counts < 20]) <- TRUE
# Convert peaks to ploidy call
peak_to_ploid(myPeaks1)
```

`Process chromR objects`

Process chromR object

Description

Functions which process chromR objects

Create representation of a sequence. Begining and end points are determined for stretches of nucleotides. Stretches are determined by querying each nucleotides in a sequence to determine if it is represented in the database of characters (chars).

Usage

```
proc.chromR(x, win.size = 1000, verbose = TRUE)

regex.win(x, max.win = 1000, regex = "[acgtwsmkrybdhv]")

seq2rects(x, chars = "acgtwsmkrybdhv", lower = TRUE)

var.win(x, win.size = 1000)
```

Arguments

`x`	object of class chromR
`win.size`	integer indicating size for windowing processes
`verbose`	logical indicating whether verbose output should be reported
`max.win`	maximum window size
`regex`	a regular expression to indicate nucleotides to be searched for
`chars`	a vector of characters to be used as a database for inclusion in rectangles
`lower`	converts the sequence and database to lower case, making the search case insensitive
`...`	arguments to be passed to methods

Details

The function **proc_chromR()** calls helper functions to process the data present in a chromR object into summaries statistics.

The function **regex.win()** is used to generate coordinates to define rectangles to represent regions of the chromosome containing called nucleotides (acgtwsmkrybdhv). It is then called a second time to generate coordinates to define rectangles to represent regions called as uncalled nucleotides (n, but not gaps).

The function **gt2popsum** is called to create summaries of the variant data.

The function **var.win** is called to create windowized summaries of the chromR object.

Each **window** receives a **name** and its coordinates. Several attempts are made to name the windows appropriately. First, the CHROM column of vcfR@fix is queried for a name. Next, the label of the sequence is queried for a name. Next, the first cell of the annotation matrix is queried. If an appropriate name was not found in the above locations the chromR object's 'name' slot is used. Note that the 'name' slot has a default value. If this default value is not updated then all of your windows may receive the same name.

query.gt *Query the gt slot*

Description

Query the 'gt' slot of objects of class vcfR

Usage

```
is.polymorphic(x, na.omit = FALSE)

is.biallelic(x)
```

Arguments

x	an object of class vcfR
na.omit	logical to omit missing data

Details

The function **is_polymorphic** returns a vector of logicals indicating whether a variant is polymorphic. Only variable sites are reported in vcf files. However, once someone manipulates a vcfR object, a site may become invariant. For example, if a sample is removed it may result in a site becoming invariant. This function queries the sites in a vcfR object and returns a vector of logicals (TRUE/FALSE) to indicate if they are actually variable.

The function **is_bialleleic** returns a vector of logicals indicating whether a variant is biallelic. Some analyses or downstream analyses only work with biallelic loci. This function can help manage this.

Note that **is_bialleleic** queries the ALT column in the fix slot to count alleles. If you remove samples from the gt slot you may invalidate the information in the fix slot. For example, if you remove the samples with the alternate allele you will make the position invariant and this function will provide inaccurate information. So use caution if you've made many modifications to your data.

See Also

extract.gt

queryMETA *Query the META section of VCF data*

Description

Query the META section of VCF data for information about acronyms.

Usage

```
queryMETA(x, element = NULL, nice = TRUE)
```

Arguments

x	an object of class vcfR or chromR.
element	an acronym to search for in the META portion of the VCF data.
nice	logical indicating whether to format the data in a 'nice' manner.

Details

The META portion of VCF data defines acronyms that are used elsewhere in the data. In order to better understand these acronyms they should be referenced. This function facilitates looking up of acronyms to present their relevant information. When 'element' is 'NULL' (the default), all acronyms from the META region are returned. When 'element' is specified an attempt is made to return information about the provided element. The function `grep` is used to perform this query. If 'nice' is set to FALSE then the data is presented as it was in the file. If 'nice' is set to TRUE the data is processed to make it appear more 'nice'.

See Also

grep, regex.

Examples

```
data(vcfR_test)
queryMETA(vcfR_test)
queryMETA(vcfR_test, element = "DP")
```

Ranking *Ranking variants within windows*

Description

Rank variants within windows.

Usage

```
rank.variants.chromR(x, scores)
```

Arguments

x	an object of class Crhom or a data.frame containing...
scores	a vector of scores for each variant to be used to rank the data

rePOS *Create non-overlapping positions (POS) for VCF data*

Description

Converts allele balance data produced by `freq_peak()` to a copy number by assinging the allele balance data (frequencies) to its closest expected ratio.

Usage

```
rePOS(x, lens, ret.lens = FALSE, buff = 0)
```

Arguments

`x`	a vcfR object
`lens`	a data.frame describing the reference
`ret.lens`	logical specifying whether lens should be returned
`buff`	an integer indicating buffer length

Details

Each chromosome in a genome typically begins with position one. This creates a problem when plotting the data associated with each chromosome because the information will overlap. This function uses the information in the data.frame `lens` to create a new coordinate system where chromosomes do not overlap.

The data.frame **lens** should have a row for each chromosome and two columns. The first column is the name of each chromosome as it appears in the vcfR object. The second column is the length of each chromosome.

The parameter **buff** indicates the length of a buffer to put in between each chromosome. This buffer may help distinguish chromosomes from one another.

In order to create the new coordinates the `lens` data.frame is updated with the new start positions. The parameter

Value

Either a vector of integers that represent the new coordinate system or a list containing the vector of integers and the lens data.frame.

Examples

```
# Create some VCF data.
data(vcfR_example)
vcf1 <-vcf[1:500,]
vcf2 <-vcf[500:1500,]
vcf3 <- vcf[1500:2533]
vcf1@fix[,'CHROM'] <- 'chrom1'
```

```
vcf2@fix[,'CHROM'] <- 'chrom2'
vcf3@fix[,'CHROM'] <- 'chrom3'
vcf2@fix[,'POS'] <- as.character(getPOS(vcf2) - 21900)
vcf3@fix[,'POS'] <- as.character(getPOS(vcf3) - 67900)
vcf <- rbind2(vcf1, vcf2)
vcf <- rbind2(vcf, vcf3)
rm(vcf1, vcf2, vcf3)

# Create lens
lens <- data.frame(matrix(nrow=3, ncol=2))
lens[1,1] <- 'chrom1'
lens[2,1] <- 'chrom2'
lens[3,1] <- 'chrom3'
lens[1,2] <- 22000
lens[2,2] <- 47000
lens[3,2] <- 32089

# Illustrate the issue.
dp <- extract.info(vcf, element="DP", as.numeric=TRUE)
plot(getPOS(vcf), dp, col=as.factor(getCHROM(vcf)))

# Resolve the issue.
newPOS <- rePOS(vcf, lens)
dp <- extract.info(vcf, element="DP", as.numeric=TRUE)
plot(newPOS, dp, col=as.factor(getCHROM(vcf)))

# Illustrate the buffer
newPOS <- rePOS(vcf, lens, buff=10000)
dp <- extract.info(vcf, element="DP", as.numeric=TRUE)
plot(newPOS, dp, col=as.factor(getCHROM(vcf)))
```

---

show,chromR-method *chromR-method*

Description

Methods that act on objects of class chromR

Usage

```
## S4 method for signature 'chromR'
show(object)

## S4 method for signature 'chromR'
plot(x, y, ...)

## S4 method for signature 'chromR'
print(x, y, ...)
```

```
## S4 method for signature 'chromR'
head(x, n = 6)

## S4 replacement method for signature 'chromR,character'
names(x) <- value

## S4 method for signature 'chromR'
length(x)
```

Arguments

object	an object of class chromR
x	an object of class chromR
y	not currently used
...	Arguments to be passed to methods
n	integer indicating the number of elements to be printed from an object
value	a character containing a name

Details

Methods that act on objects of class chromR.

---

show,vcfR-method *show*

Description

Display a summary of a vcfR object.

head returns the first parts of an object of class vcfR.

The brackets ('[]') subset objects of class vcfR

The **plot** method visualizes objects of class vcfR

Usage

```
## S4 method for signature 'vcfR'
show(object)

## S4 method for signature 'vcfR'
head(x, n = 6, maxchar = 80)

## S4 method for signature 'vcfR,ANY,ANY,ANY'
x[i, j, samples = NULL, ..., drop]

## S4 method for signature 'vcfR'
```

```
plot(x, y, ...)

## S4 method for signature 'vcfR,missing'
rbind2(x, y, ...)

## S4 method for signature 'vcfR,ANY'
rbind2(x, y, ...)

## S4 method for signature 'vcfR,vcfR'
rbind2(x, y, ...)

## S4 method for signature 'vcfR'
dim(x)

## S4 method for signature 'vcfR'
nrow(x)
```

Arguments

object	a vcfR object
x	object of class vcfR
n	number of rows to print
maxchar	maximum number of characters to print per line
i	vector of rows (variants) to include
j	vector of columns (samples) to include
samples	vector (numeric, character or logical) specifying samples, see details
...	arguments to be passed to other methods
drop	delete the dimensions of an array which only has one level
y	not used

Details

The method **show** is used to display an object. Because vcf data are relatively large, this has been abbreviated. Here we display the first four lines of the meta section, and truncate them to no more than 80 characters. The first eight columns and six rows of the fix section are also displayed.

The method **head** is similar to show, but is more flexible. The number of rows displayed is parameterized by the variable n. And the maximum number of characters to print per line (row) is also parameterized. In contract to show, head includes a summary of the gt portion of the vcfR object.

The **square brackets ([])** are used to subset objects of class vcfR. Rows are subset by providing a vector i to specify which rows to use. The columns in the fix slot will not be subset by j. The parameter j is a vector used to subset the columns of the gt slot. Note that it is essential to include the first column here (FORMAT) or downsream processes will encounter trouble.

The **samples** parameter allows another way to select samples. Because the first column of the gt section is the FORMAT column you typically need to include that column and sample numbers therefore begin at two. Use of the samples parameter allows you to select columns by a vector of

numerics, logicals or characters. When numerics are used the samples can be selected starting at one. The function will then add one to this vector and include one to select the desired samples and the FORMAT column. When a vector of characters is used it should contain the desired sample names. The function will add the FORMAT column if it is not the first element. When a vector of logicals is used a TRUE will be added to the vector to ensure the FORMAT column is selected. Note that specification of samples will override specification of j.

The **plot** method generates a histogram from data found in the 'QUAL' column from the 'fix' slot.

`VCF input and output` *Read and write vcf format files*

Description

Read and files in the *.vcf structured text format, as well as the compressed *.vcf.gz format. Write objects of class vcfR to *.vcf.gz.

Usage

```
read.vcfR(
  file,
  limit = 1e+07,
  nrows = -1,
  skip = 0,
  cols = NULL,
  convertNA = TRUE,
  checkFile = TRUE,
  check_keys = TRUE,
  verbose = TRUE
)

write.vcf(x, file = "", mask = FALSE, APPEND = FALSE)
```

Arguments

`file`	A filename for a variant call format (vcf) file.
`limit`	amount of memory (in bytes) not to exceed when reading in a file.
`nrows`	integer specifying the maximum number of rows (variants) to read in.
`skip`	integer specifying the number of rows (variants) to skip before beginning to read data.
`cols`	vector of column numbers to extract from file.
`convertNA`	logical specifying to convert VCF missing data to NA.
`checkFile`	test if the first line follows the VCF specification.
`check_keys`	logical determining if `check_keys()` is called to test if INFO and FORMAT keys are unique.

`verbose`	report verbose progress.
`x`	An object of class vcfR or chromR.
`mask`	logical vector indicating rows to use.
`APPEND`	logical indicating whether to append to existing vcf file or write a new file.

Details

The function **read.vcfR** reads in files in *.vcf (text) and *.vcf.gz (gzipped text) format and returns an object of class vcfR. The parameter 'limit' is an attempt to keep the user from trying to read in a file which contains more data than there is memory to hold. Based on the dimensions of the data matrix, an estimate of how much memory needed is made. If this estimate exceeds the value of 'limit' an error is thrown and execution stops. The user may increase this limit to any value, but is encourages to compare that value to the amout of available physical memory.

It is possible to input part of a VCF file by using the parameters nrows, skip and cols. The first eight columns (the fix region) are part of the definition and will always be included. Any columns beyond eight are optional (the gt region). You can specify which of these columns you would like to input by setting the cols parameter. If you want a usable vcfR object you will want to always include nine (the FORMAT column). If you do not include column nine you may experience reduced functionality.

According to the VCF specification **missing data** are encoded by a period ("."). Within the R language, missing data can be encoded as NA. The parameter 'convertNA' allows the user to either retain the VCF representation or the R representation of missing data. Note that the conversion only takes place when the entire value can be determined to be missing. For example, ".|.:48:8:51,51" would be retained because the missing genotype is accompanied by other delimited information. In contrast, ".|." should be converted to NA when `convertNA = TRUE`.

If file begins with http://, https://, ftp://, or ftps:// it is interpreted as a link. When this happens, file is split on the delimiter '/' and the last element is used as the filename. A check is performed to determine if this file exists in the working directory. If a local file is found it is used. If a local file is not found the remote file is downloaded to the working directory and read in.

The function **write.vcf** takes an object of either class vcfR or chromR and writes the vcf data to a vcf.gz file (gzipped text). If the parameter 'mask' is set to FALSE, the entire object is written to file. If the parameter 'mask' is set to TRUE and the object is of class chromR (which has a mask slot), this mask is used to subset the data. If an index is supplied as 'mask', then this index is used, and recycled as necessary, to subset the data.

Because vcfR provides the opportunity to manipulate VCF data, it also provides the opportunity for the user to create invalid VCF files. If there is a question regarding the validity of a file you have created one option is the VCF validator from VCF tools.

Value

read.vcfR returns an object of class `vcfR-class`. See the **vignette:** `vignette('vcf_data')`. The function write.vcf creates a gzipped VCF file.

See Also

CRAN: pegas::read.vcf, PopGenome::readVCF, data.table::fread

Bioconductor: VariantAnnotation::readVcf

Use: browseVignettes('vcfR') to find examples.

Examples

```
data(vcfR_test)
vcfR_test
head(vcfR_test)
# CRAN requires developers to us a tempdir when writing to the filesystem.
# You may want to implement this example elsewhere.
orig_dir <- getwd()
temp_dir <- tempdir()
setwd( temp_dir )
write.vcf( vcfR_test, file = "vcfR_test.vcf.gz" )
vcf <- read.vcfR( file = "vcfR_test.vcf.gz", verbose = FALSE )
vcf
setwd( orig_dir )
```

---

vcfR *Variant call format files processed with vcfR.*

Description

vcfR provides a suite of tools for input and output of variant call format (VCF) files, manipulation of their content and visualization.

Details

File input and output is facilitated with the functions `read.vcfR` and `write.vcf`. Input of vcf format data results in an S4 `vcfR-class` object. Objects of class vcfR can be manipulated with vcfR-method and `extract.gt`. Contents of the vcfR object can be visualized with the `plot` method. More complex visualizations can be created using a series of functions. See `vignette(topic="sequence_coverage")` for an example. Once manipulations are complete the object may be written to a *.vcf.gz format file using `write.vcf` or exported to objects supported by other R packages with `vcfR2genind` or `vcfR2loci`.

More complex visualization can be accomplished by converting a vcfR object to a `chromR-class` object. An example exists on the `create.chromR` man page.

A **complete list of functions** can be displayed with: library(help = vcfR).

Vignettes (documentation) can be listed with: `browseVignettes('vcfR')`.

Several example **datasets** are included in vcfR. **vcfR_test** comes from the VCF specification and provides a vcfR object with a diversity of examples in a small dataset. **vcfR_example** is a subset of the pinfsc50 dataset that includes VCF, GFF and FASTA data for moderate sized testing. The pinfsc50 dataset is available as a separate package and includes VCF, GFF and FASTA data for testing and benchmarking.

See Also

More documentation for vcfR can be found at the vcfR documentation website.

`vcfR-class` *vcfR class*

Description

An S4 class for storing VCF data.

Details

Defines a class for variant call format data. A vcfR object contains three slots. The first slot is a character vector which holds the meta data. The second slot holds an eight column matrix to hold the fixed data. The third slot is a matrix which holds the genotype data. The genotype data is optional according to the VCF definition. When it is missing the gt slot should consist of a character matrix with zero rows and columns.

See `vignette('vcf_data')` for more information. See the VCF specification for the file specification.

Slots

`meta` character vector for the meta information

`fix` matrix for the fixed information

`gt` matrix for the genotype information

`vcfR2DNAbin` *Convert vcfR to DNAbin*

Description

Convert objects of class vcfR to objects of class ape::DNAbin

Usage

```
vcfR2DNAbin(
  x,
  extract.indels = TRUE,
  consensus = FALSE,
  extract.haps = TRUE,
  unphased_as_NA = TRUE,
  asterisk_as_del = FALSE,
  ref.seq = NULL,
  start.pos = NULL,
  verbose = TRUE
)
```

Arguments

x	an object of class chromR or vcfR
extract.indels	logical indicating to remove indels (TRUE) or to include them while retaining alignment
consensus	logical, indicates whether an IUPAC ambiguity code should be used for diploid heterozygotes
extract.haps	logical specifying whether to separate each genotype into alleles based on a delimiting character
unphased_as_NA	logical indicating how to handle alleles in unphased genotypes
asterisk_as_del	logical indicating that the asterisk allele should be converted to a deletion (TRUE) or NA (FALSE)
ref.seq	reference sequence (DNAbin) for the region being converted
start.pos	chromosomal position for the start of the ref.seq
verbose	logical specifying whether to produce verbose output

Details

Objects of class **DNAbin**, from the package ape, store nucleotide sequence information. Typically, nucleotide sequence information contains all the nucleotides within a region, for example, a gene. Because most sites are typically invariant, this results in a large amount of redundant data. This is why files in the vcf format only contain information on variant sites, it results in a smaller file. Nucleotide sequences can be generated which only contain variant sites. However, some applications require the invariant sites. For example, inference of phylogeny based on maximum likelihood or Bayesian methods requires invariant sites. The function vcfR2DNAbin therefore includes a number of options in attempt to accomodate various scenarios.

The presence of indels (insertions or deletions)in a sequence typically presents a data analysis problem. Mutation models typically do not accomodate this data well. For now, the only option is for indels to be omitted from the conversion of vcfR to DNAbin objects. The option **extract.indels** was included to remind us of this, and to provide a placeholder in case we wish to address this in the future.

The **ploidy** of the samples is inferred from the first non-missing genotype. All samples and all variants within each sample are assumed to be of the same ploid.

Conversion of **haploid data** is fairly straight forward. The options `consensus` and `extract.haps` are not relevant here. When vcfR2DNAbin encounters missing data in the vcf data (NA) it is coded as an ambiguous nucleotide (n) in the DNAbin object. When no reference sequence is provided (option `ref.seq`), a DNAbin object consisting only of variant sites is created. When a reference sequence and a starting position are provided the entire sequence, including invariant sites, is returned. The reference sequence is used as a starting point and variable sitees are added to this. Because the data in the vcfR object will be using a chromosomal coordinate system, we need to tell the function where on this chromosome the reference sequence begins.

Conversion of **diploid data** presents a number of scenarios. When the option `consensus` is TRUE and `extract.haps` is FALSE, each genotype is split into two alleles and the two alleles are converted into their IUPAC ambiguity code. This results in one sequence for each diploid sample. This may be an appropriate path when you have unphased data. Note that functions called downstream

of this choice may handle IUPAC ambiguity codes in unexpected manners. When extract.haps is set to TRUE, each genotype is split into two alleles. These alleles are inserted into two sequences. This results in two sequences per diploid sample. Note that this really only makes sense if you have phased data. The options ref.seq and start.pos are used as in halpoid data.

When a variant overlaps a deletion it may be encoded by an **asterisk allele (*)**. The GATK site covers this in a post on Spanning or overlapping deletions]. This is handled in vcfR by allowing the user to decide how it is handled with the paramenter `asterisk_as_del`. When `asterisk_as_del` is TRUE this allele is converted into a deletion ('-'). When `asterisk_as_del` is FALSE the asterisk allele is converted to NA. If `extract.indels` is set to FALSE it should override this decision.

Conversion of **polyploid data** is currently not supported. However, I have made some attempts at accomodating polyploid data. If you have polyploid data and are interested in giving this a try, feel free. But be prepared to scrutinize the output to make sure it appears reasonable.

Creation of DNAbin objects from large chromosomal regions may result in objects which occupy large amounts of memory. If in doubt, begin by subsetting your data and the scale up to ensure you do not run out of memory.

See Also

ape

Examples

```
library(ape)
data(vcfR_test)

# Create an example reference sequence.
nucs <- c('a','c','g','t')
set.seed(9)
myRef <- as.DNAbin(matrix(nucs[round(runif(n=20, min=0.5, max=4.5))], nrow=1))

# Recode the POS data for a smaller example.
set.seed(99)
vcfR_test@fix[,'POS'] <- sort(sample(10:20, size=length(getPOS(vcfR_test))))

# Just vcfR
myDNA <- vcfR2DNAbin(vcfR_test)
seg.sites(myDNA)
image(myDNA)

# ref.seq, no start.pos
myDNA <- vcfR2DNAbin(vcfR_test, ref.seq = myRef)
seg.sites(myDNA)
image(myDNA)

# ref.seq, start.pos = 4.
# Note that this is the same as the previous example but the variants are shifted.
myDNA <- vcfR2DNAbin(vcfR_test, ref.seq = myRef, start.pos = 4)
seg.sites(myDNA)
image(myDNA)
```

```
# ref.seq, no start.pos, unphased_as_NA = FALSE
myDNA <- vcfR2DNAbin(vcfR_test, unphased_as_NA = FALSE, ref.seq = myRef)
seg.sites(myDNA)
image(myDNA)
```

---

vcfR2migrate *Convert a vcfR object to MigrateN input file*

Description

The function converts a vcfR object to a text format that can be used as an infile for MigrateN.

Usage

```
vcfR2migrate(
  vcf,
  pop,
  in_pop,
  out_file = "MigrateN_infile.txt",
  method = c("N", "H")
)
```

Arguments

vcf	a vcfR object.
pop	factor indicating population membership for each sample.
in_pop	vector of population names indicating which population to include in migrate output file.
out_file	name of output file.
method	should 'N' or 'H' format data be generated?

Details

This function converts a vcfR object to a text file which can be used as input for MigrateN. The function will remove loci with missing data, indels, and loci that are not bialleleic (loci with more than two alleles). Thus, only SNP data analysed where the length of each locus (inmutational steps) is 1 (as opposed to microsatellites or indels).

The output file should contain Unix line endings ("\n"). Note that opening the output file in a Windows text editor (just to validate number of markers, individuals or populations) might change the end of line character (eol) to a Windows line ending ("\r\n"). This may produce an error running migrate-n. Because these are typically non-printing characters, this may be a difficult problem to troubleshoot. The easiest way to circumvent the problem is to transfer the output file to Unix machine and view it there. If you do introduce Windows line endings you can convert them back to Unix with a program such as 'dos2unix' or 'fromdos' to change the line endings.

Value

a text file that can be used as an input for MigrateN software (SNP format).

Author(s)

Shankar Shakya and Brian J. Knaus

See Also

Migrate-N website.

Examples

```
## Not run:
data(vcfR_example)
my_pop <- as.factor(paste("pop_", rep(c("A", "B", "C"), each = 6), sep = ""))
vcfR2migrate(vcf = vcf , pop = my_pop , in_pop = c("pop_A","pop_C"),
             out_file = "my2pop.txt", method = 'H')

## End(Not run)
```

---

vcfR_example *Example data for vcfR.*

Description

An example dataset containing parts of the *Phytophthora infestans* genome.

Format

A DNAbin object, a data.frame and a vcfR object

Details

- dna DNAbin object
- gff gff format data.frame
- vcf vcfR object

This data is a subset of the pinfsc50 dataset. It has been subset to positions between 500 and 600 kbp. The coordinate systems of the vcf and gff file have been altered by subtracting 500,000. This results in a 100 kbp section of supercontig_1.50 that has positional data ranging from 1 to 100 kbp.

Note that it is encouraged to keep package contents small to facilitate easy downloading and installation. This is why a mitochondrion was chosen as an example. In practice I've used this package on supercontigs. This package was designed for much larger datasets in mind than in this example.

Examples

```
data(vcfR_example)
```

---

vcfR_test *Test data for vcfR.*

Description

A test file containing a diversity of examples intended to test functionality.

Format

A vcfR object

Details

- vcfR_test vcfR object

This data set began as the example (section 1.1) from The Variant Call Format Specification VCFv4.3 . This data consisted of 3 samples and 5 variants. As I encounter examples that challenge the code in vcfR they can be added to this data set.

Examples

```
data(vcfR_test)


## Not run:
  # When I add data it can be saved with this command.
  save(vcfR_test, file="data/vcfR_test.RData")

## End(Not run)
```

vep *Example data from the Variant Effect Predictor (VEP).*

Description

Example data to use with unit tests.

Format

A vcfR object

Details

- vep vcfR object

Output from the VEP may include values with multiple equals signs. This does not appear to conform with the VCF specification (at the time of writing this VCF v4.3). But it appears fairly easy to accomodate. This example data can be used to make unit tests to validate functionality.

Examples

```
data(vep)
vcfR2tidy(vep, info_only = TRUE)$fix
```

Windowing *Create window summaries of data*

Description

Create windows of non-overlapping data and summarize.

Usage

```
NM2winNM(x, pos, maxbp, winsize = 100L, depr = TRUE)

z.score(x)

windowize.NM(x, pos, starts, ends, summary = "mean", depr = TRUE)
```

Arguments

x	A NumericMatrix
pos	A vector of chromosomal positions for each row of data (variants)
maxbp	Length of chromosome
winsize	Size (in bp) for windows
depr	logical (T/F), this function has been deprecated, set to FALSE to override.
starts	integer vector of starting positions for windows
ends	integer vector of ending positions for windows
summary	string indicating type of summary (mean, median, sum)

Details

The numeric matrix where samples are in columns and variant data are in rows. The windowing process therefore occurs along columns of data. This matrix could be created with `extract.gt`.

The chromosome is expected to contain positions 1 though maxbp. If maxbp is not specified this can be inferred from the last element in pos.

`write.fasta` *Create fasta format output*

Description

Generate fasta format output

Usage

```
write.fasta(
  x,
  file = "",
  rowlength = 80,
  tolower = TRUE,
  verbose = TRUE,
  APPEND = FALSE,
  depr = TRUE
)
```

Arguments

x	object of class chromR
file	name for output file
rowlength	number of characters each row should not exceed
tolower	convert all characters to lowercase (T/F)
verbose	should verbose output be generated (T/F)
APPEND	should data be appended to an existing file (T/F)
depr	logical (T/F), this function has been deprecated, set to FALSE to override.

Details

The function **write_fasta** takes an object of class chromR and writes it to a fasta.gz (gzipped text) format file. The sequence in the seq slot of the chromR object is used to fill in the invariant sites. The parameter 'tolower', when set to TRUE, converts all the characters in teh sequence to lower case. This is important because some software, such as ape::DNAbin, requires sequences to be in lower case.

---

write.var.info *Write summary tables from chromR objects*

Description

Write summary tables from chromR objects.

Usage

```
write.var.info(x, file = "", mask = FALSE, APPEND = FALSE)

write.win.info(x, file = "", APPEND = FALSE)
```

Arguments

x	An object of class chromR
file	A filename for the output file
mask	logical vector indicating rows to use
APPEND	logical indicating whether to append to existing file (omitting the header) or write a new file

Details

The function **write.var.info** takes the variant information table from a chromR object and writes it as a comma delimited file.

The function **write.win.info** takes the window information table from a chromR object and writes it as a comma delimited file.

See Also

write.vcf

Index