Package 'VariantAnnotation'

November 6, 2025

Type Package

Title Annotation of Genetic Variants

Description Annotate variants, compute amino acid coding changes, predict coding outcomes.

Version 1.57.0

License Artistic-2.0

Depends R (>= 4.0.0), methods, BiocGenerics (>= 0.37.0), MatrixGenerics, Seqinfo, GenomicRanges (>= 1.61.1), SummarizedExperiment (>= 1.39.1), Rsamtools (>= 2.25.1)

Imports utils, DBI, Biobase, S4Vectors (>= 0.27.12), IRanges (>= 2.23.9), XVector (>= 0.29.2), Biostrings (>= 2.77.2), AnnotationDbi (>= 1.27.9), rtracklayer (>= 1.69.1), BSgenome (>= 1.77.1), GenomicFeatures (>= 1.61.4), curl

Suggests GenomeInfoDb, RUnit, AnnotationHub, BSgenome.Hsapiens.UCSC.hg19, TxDb.Hsapiens.UCSC.hg19.knownGene, SNPlocs.Hsapiens.dbSNP144.GRCh37, SIFT.Hsapiens.dbSNP132, SIFT.Hsapiens.dbSNP137, PolyPhen.Hsapiens.dbSNP131, snpStats, ggplot2, BiocStyle, knitr, magick, jsonlite, httr, rjsoncons

LinkingTo S4Vectors, IRanges, XVector, Biostrings, Rhtslib (>= 2.99.0)

SystemRequirements GNU make

LazyLoad yes

biocViews DataImport, Sequencing, SNP, Annotation, Genetics, VariantAnnotation

Video https://www.youtube.com/watch?v=Ro0lHQ_J--I&list=UUqaMSQd_h-2EDGsU6WDiX0Q

RoxygenNote 7.3.1

VignetteBuilder knitr

git_url https://git.bioconductor.org/packages/VariantAnnotation

git_branch devel

git_last_commit 0da7ddb

git_last_commit_date 2025-10-29

Repository Bioconductor 3.23

Date/Publication 2025-11-05

Author Valerie Obenchain [aut],
Martin Morgan [aut],
Michael Lawrence [aut],
Stephanie Gogarten [ctb],
Bioconductor Package Maintainer [cre]

Maintainer Bioconductor Package Maintainer <maintainer@bioconductor.org>

Contents

filterVcf *Filter VCF files*

Description

Filter Variant Call Format (VCF) files from one file to another

Usage

```
## S4 method for signature 'character'
filterVcf(file, genome, destination, ..., verbose = TRUE,
    index = FALSE, prefilters = FilterRules(), filters = FilterRules(),
    param = ScanVcfParam())

## S4 method for signature 'TabixFile'
filterVcf(file, genome, destination, ..., verbose = TRUE,
    index = FALSE, prefilters = FilterRules(), filters = FilterRules(),
    param = ScanVcfParam())
```

Arguments

file	A character(1) file path or TabixFile specifying the VCF file to be filtered.
genome	A character(1) identifier
destination	A character(1) path to the location where the filtered VCF file will be written.
...	Additional arguments, possibly used by future methods.
verbose	A logical(1) indicating whether progress messages should be printed.
index	A logical(1) indicating whether the filtered file should be compressed and indexed (using bgzip and indexTabix).
prefilters	A FilterRules instance contains rules for filtering un-parsed lines of the VCF file.
filters	A FilterRules instance contains rules for filtering fully parsed VCF objects.
param	A ScanVcfParam instance restricting input of particular info or geno fields, or genomic locations. Applicable when applying a filter only. Prefiltering involves a grep of unparsed lines in the file; indexing is not used.

Details

This function transfers content of one VCF file to another, removing records that fail to satisfy prefilters and filters. Filtering is done in a memory efficient manner, iterating over the input VCF file in chunks of default size 100,000 (when invoked with character(1) for file) or as specified by the yieldSize argument of TabixFile (when invoked with TabixFile).

There are up to two passes. In the first pass, unparsed lines are passed to prefilters for filtering, e.g., searching for a fixed character string. In the second pass lines successfully passing prefilters are parsed into VCF instances and made available for further filtering. One or both of prefilter and filter can be present.

Filtering works by removing the rows (variants) that do not meet a criteria. Because this is a row-based approach and samples are column-based most genotype filters are only meaningful for single-sample files. If a single samples fails the criteria the entire row (all samples) are removed. The case where genotype filtering is effective for multiple samples is when the criteria is applied across samples and not to the individual (e.g., keep rows where all samples have DP > 10).

Value

The destination file path as a `character(1)`.

Author(s)

Martin Morgan and Paul Shannon

See Also

`readVcf`, `writeVcf`.

Examples

```
fl <- system.file("extdata", "chr22.vcf.gz", package="VariantAnnotation")

## ---------------------------------------------------------------------
## Filter for SNVs in a defined set of ranges:
## ---------------------------------------------------------------------

if (require(TxDb.Hsapiens.UCSC.hg19.knownGene)) {

  txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
  exons <- exons(txdb)
  exons22 <- exons[seqnames(exons) == "chr22"]

  library(GenomeInfoDb)
  seqlevelsStyle(exons22) <- "NCBI"  ## match chrom names in VCF file

  ## Range-based filter:
  withinRange <- function(rng)
      function(x) x

  ## The first filter identifies SNVs and the second applies the
  ## range restriction.
  filters <- FilterRules(list(
      isSNV = isSNV,
      withinRange = withinRange(exons22)))

  ## Apply
  ## Not run:
  filt1 <- filterVcf(fl, "hg19", tempfile(), filters=filters, verbose=TRUE)

## End(Not run)
}
```

```
## ------------------------------------------------------------------------
## Using a pre-filter and filter:
## ------------------------------------------------------------------------

## Low coverage exome snp filter:
lowCoverageExomeSNP = function(x) grepl("LOWCOV,EXOME", x, fixed=TRUE)

## The pre-filter identifies low coverage exome snps and the filter
## identifies variants with INFO variable VT = SNP.
pre <- FilterRules(list(lowCoverageExomeSNP = lowCoverageExomeSNP))
filt <- FilterRules(list(VTisSNP = function(x) info(x)$VT == "SNP"))

## Apply
filt2 <- filterVcf(fl, "hg19", tempfile(), prefilters=pre, filters=filt)

## Filtered results
vcf <- readVcf(filt2, "hg19")
```

---

genotypeToSnpMatrix *Convert genotype calls from a VCF file to a SnpMatrix object*

Description

Convert an array of genotype calls from the "GT", "GP", "GL" or "PL" FORMAT field of a VCF file to a SnpMatrix.

Usage

```
## S4 method for signature 'CollapsedVCF'
genotypeToSnpMatrix(x, uncertain=FALSE, ...)
## S4 method for signature 'array'
genotypeToSnpMatrix(x, ref, alt, ...)
```

Arguments

x	A CollapsedVCF object or a array of genotype data from the "GT", "GP", "GL" or "PL" FORMAT field of a VCF file. This array is created with a call to readVcf and can be accessed with geno(<VCF>).
uncertain	A logical indicating whether the genotypes to convert should come from the "GT" field (uncertain=FALSE) or the "GP", "GL" or "PL" field (uncertain=TRUE).
ref	A DNAStringSet of reference alleles.
alt	A DNAStringSetList of alternate alleles.
...	Additional arguments, passed to methods.

Details

genotypeToSnpMatrix converts an array of genotype calls from the "GT", "GP", "GL" or "PL" FORMAT field of a VCF file into a SnpMatrix. The following caveats apply,

- no distinction is made between phased and unphased genotypes
- variants with >1 ALT allele are set to NA
- only single nucleotide variants are included; others are set to NA
- only diploid calls are included; others are set to NA

In VCF files, 0 represents the reference allele and integers greater than 0 represent the alternate alleles (i.e., 2, 3, 4 would indicate the 2nd, 3rd or 4th allele in the ALT field for a particular variant). This function only supports variants with a single alternate allele and therefore the alternate values will always be 1. Genotypes are stored in the SnpMatrix as 0, 1, 2 or 3 where 0 = missing, 1 = "0/0", 2 = "0/1" or "1/0" and 3 = "1/1". In SnpMatrix terminology, "A" is the reference allele and "B" is the risk allele. Equivalent statements to those made with 0 and 1 allele values would be 0 = missing, 1 = "A/A", 2 = "A/B" or "B/A" and 3 = "B/B".

The genotype fields are defined as follows:

- GT : genotype, encoded as allele values separated by either of "/" or "|". The allele values are 0 for the reference allele and 1 for the alternate allele.
- GL : genotype likelihoods comprised of comma separated floating point log10-scaled likelihoods for all possible genotypes. In the case of a reference allele A and a single alternate allele B, the likelihoods will be ordered "A/A", "A/B", "B/B".
- PL : the phred-scaled genotype likelihoods rounded to the closest integer. The ordering of values is the same as for the GL field.
- GP : the phred-scaled genotype posterior probabilities for all possible genotypes; intended to store imputed genotype probabilities. The ordering of values is the same as for the GL field.

If uncertain=TRUE, the posterior probabilities of the three genotypes ("A/A", "A/B", "B/B") are encoded (approximately) as byte values. This encoding allows uncertain genotypes to be used in snpStats functions, which in some cases may be more appropriate than using only the called genotypes. The byte encoding conserves memory by allowing the uncertain genotypes to be stored in a two-dimensional raw matrix. See the snpStats documentation for more details.

Value

A list with the following elements,

genotypes	The output genotype data as an object of class "SnpMatrix". The columns are snps and the rows are the samples. See ?SnpMatrix details of the class structure.
map	A DataFrame giving the snp names and alleles at each locus. The ignore column indicates which variants were set to NA (see NA criteria in 'details' section).

Author(s)

Stephanie Gogarten and Valerie Obenchain

References

http://www.1000genomes.org/wiki/Analysis/Variant%20Call%20Format/vcf-variant-call-format-version-41

See Also

readVcf, VCF, SnpMatrix

Examples

```
## ------------------------------------------------------------------
## Non-probability based snp encoding using "GT"
## ------------------------------------------------------------------
fl <- system.file("extdata", "ex2.vcf", package="VariantAnnotation")
vcf <- readVcf(fl, "hg19")

## This file has no "GL" or "GP" field so we use "GT".
geno(vcf)

## Convert the "GT" FORMAT field to a SnpMatrix.
mat <- genotypeToSnpMatrix(vcf)

## The result is a list of length 2.
names(mat)

## Compare coding in the VCF file to the SnpMatrix.
geno(vcf)$GT
t(as(mat$genotype, "character"))

## The 'ignore' column in 'map' indicates which variants
## were set to NA. Variant rs6040355 was ignored because
## it has multiple alternate alleles, microsat1 is not a
## snp, and chr20:1230237 has no alternate allele.
mat$map

## ------------------------------------------------------------------
## Probability-based encoding using "GL", "PL" or "GP"
## ------------------------------------------------------------------
## Read a vcf file with a "GL" field.
fl <- system.file("extdata", "gl_chr1.vcf", package="VariantAnnotation")
vcf <- readVcf(fl, "hg19")
geno(vcf)

## Convert the "GL" FORMAT field to a SnpMatrix
mat <- genotypeToSnpMatrix(vcf, uncertain=TRUE)

## Only 3 of the 9 variants passed the filters.  The
## other 6 variants had no alternate alleles.
mat$map

## Compare genotype representations for a subset of
## samples in variant rs180734498.
## Original called genotype
```

```
geno(vcf)$GT["rs180734498", 14:16]

## Original genotype likelihoods
geno(vcf)$GL["rs180734498", 14:16]

## Posterior probability (computed inside genotypeToSnpMatrix)
GLtoGP(geno(vcf)$GL["rs180734498", 14:16, drop=FALSE])[1,]

## SnpMatrix coding.
t(as(mat$genotype, "character"))["rs180734498", 14:16]
t(as(mat$genotype, "numeric"))["rs180734498", 14:16]

## For samples NA11829 and NA11830, one probability is significantly
## higher than the others, so SnpMatrix calls the genotype.  These
## calls match the original coding: "0|1" -> "A/B", "0|0" -> "A/A".
## Sample NA11831 was originally called as "0|1" but the probability
## of "0|0" is only a factor of 3 lower, so SnpMatrix calls it as
## "Uncertain" with an appropriate byte-level encoding.
```

`getTranscriptSeqs` *Get transcript sequences*

Description

This function is defunct. Use GenomicFeatures::extractTranscriptSeqs() instead.

Extract transcript sequences from a BSgenome object or an FaFile.

Usage

```
  ## S4 method for signature 'GRangesList,BSgenome'
getTranscriptSeqs(query, subject, ...)
  ## S4 method for signature 'GRangesList,FaFile'
getTranscriptSeqs(query, subject, ...)
  ## S4 method for signature 'GRanges,FaFile'
getTranscriptSeqs(query, subject, ...)
```

Arguments

`query`	A GRangesList object containing exons or cds grouped by transcript.
`subject`	A BSgenome object or a FaFile from which the sequences will be taken.
`...`	Additional arguments

Details

`getTranscriptSeqs` is a wrapper for the `extractTranscriptSeqs` and `getSeq` functions. The purpose is to allow sequence extraction from either a BSgenome or FaFile. Transcript sequences are extracted based on the boundaries of the feature provided in the `query` (i.e., either exons or cds regions).

Value

A DNAStringSet instance containing the sequences for all transcripts specified in `query`.

Author(s)

Valerie Obenchain

See Also

predictCoding extractTranscriptSeqs getSeq

Examples

```
## See ?extractTranscriptSeqs in the GenomicFeatures package.
```

`GLtoGP` *Convert genotype likelihoods to genotype probabilities*

Description

Convert an array of genotype likelihoods to posterior genotype probabilities.

Usage

```
GLtoGP(gl)
PLtoGP(pl)
```

Arguments

`gl`	Array of genotype likelihoods (log10-scaled). The format can be a matrix of lists, or a three-dimensional array in which the third dimension corresponds to the probabilities for each genotype.
`pl`	Array of genotype likelihoods (phred-scaled, i.e. -10*log10). The format can be a matrix of lists, or a three-dimensional array in which the third dimension corresponds to the probabilities for each genotype.

Details

`GLtoGP` computes the probability of each genotype as `10^x / sum(10^x)`. `PLtoGP` first divides by -10 and then proceeds as in `GLtoGP`.

Value

An array of posterior genotype probabilities, in the same format as the input (matrix of lists or 3D array).

Author(s)

Stephanie Gogarten <sdmorris@u.washington.edu>

See Also

readVcf, genotypeToSnpMatrix

Examples

```
## Read a vcf file with a "GL" field.
vcfFile <- system.file("extdata", "gl_chr1.vcf", package="VariantAnnotation")
vcf <- readVcf(vcfFile, "hg19")

## extract genotype likelihoods as a matrix of lists
gl <- geno(vcf)$GL
class(gl)
mode(gl)

# convert to posterior probabilities
gp <- GLtoGP(gl)

## Read a vcf file with a "PL" field.
vcfFile <- system.file("extdata", "hapmap_exome_chr22.vcf.gz",
                       package="VariantAnnotation")
vcf <- readVcf(vcfFile, "hg19")

## extract genotype likelihoods as a matrix of lists
pl <- geno(vcf)$PL
class(pl)
mode(pl)

# convert to posterior probabilities
gp <- PLtoGP(pl)
```

`indexVcf` *Create index files for VCF files*

Description

`indexVcf()` creates an index file for compressed VCF files, if the index file does not exist.

Usage

```
## S4 method for signature 'character'
indexVcf(x, ...)
## S4 method for signature 'VcfFile'
indexVcf(x, ...)
## S4 method for signature 'VcfFileList'
indexVcf(x, ...)
```

Arguments

x	character(), VcfFile, or VcfFileList pointing to bgzf-compressed VCF files.
...	Additional arguments to indexTabix

Details

If x is a character vector, assumes they are the path(s) to bgzf-compressed VCF file(s). If an index does not exist, one is created. VCF files can be compreseed using bgzip. A VcfFile or VcfFileList is returned.

If a VcfFile or VcfFileList is given, the index file is checked, if it does not exist it will crete one. If the index file was NA or missing, the path of the associated VCF file is used as the index file path. An updated VcfFile or VcfFileList is returned.

Value

VcfFile or VcfFileList

Author(s)

Lori Shepherd

See Also

VcfFile

Examples

```
fl <- system.file(
    "extdata", "chr7-sub.vcf.gz", package="VariantAnnotation",
    mustWork=TRUE
)
vcf1 <- indexVcf(fl)
vcf1
```

---

isSNV *Identification of genomic variant types.*

Description

Functions for identifying variant types such as SNVs, insertions, deletions, transitions, and structural rearrangements.

Usage

```
## S4 method for signature 'VRanges'
isSNV(x, ...)
## S4 method for signature 'ExpandedVCF'
isSNV(x, ...)
## S4 method for signature 'CollapsedVCF'
isSNV(x, ..., singleAltOnly = TRUE)

## S4 method for signature 'VRanges'
isInsertion(x, ...)
## S4 method for signature 'ExpandedVCF'
isInsertion(x, ...)
## S4 method for signature 'CollapsedVCF'
isInsertion(x, ..., singleAltOnly = TRUE)

## S4 method for signature 'VRanges'
isDeletion(x, ...)
## S4 method for signature 'ExpandedVCF'
isDeletion(x, ...)
## S4 method for signature 'CollapsedVCF'
isDeletion(x, ..., singleAltOnly = TRUE)

## S4 method for signature 'VRanges'
isIndel(x, ...)
## S4 method for signature 'ExpandedVCF'
isIndel(x, ...)
## S4 method for signature 'CollapsedVCF'
isIndel(x, ..., singleAltOnly = TRUE)

## S4 method for signature 'VRanges'
isDelins(x, ...)
## S4 method for signature 'ExpandedVCF'
isDelins(x, ...)
## S4 method for signature 'CollapsedVCF'
isDelins(x, ..., singleAltOnly = TRUE)

## S4 method for signature 'VRanges'
isTransition(x, ...)
## S4 method for signature 'ExpandedVCF'
isTransition(x, ...)
## S4 method for signature 'CollapsedVCF'
isTransition(x, ..., singleAltOnly = TRUE)

## S4 method for signature 'VRanges'
isSubstitution(x, ...)
## S4 method for signature 'ExpandedVCF'
isSubstitution(x, ...)
## S4 method for signature 'CollapsedVCF'
```

```
isSubstitution(x, ..., singleAltOnly = TRUE)
```

Arguments

x	A VCF or VRanges object.
singleAltOnly	A logical only applicable when x is a CollapsedVCF class. When TRUE (default) only variants with a single alternate allele are evaluated; all multi-alt variants evaluate to FALSE. When singleAltOnly=FALSE all ref / alt pairs for each variant are evaluated. If any ref / alt pairs meet the test criteria a value of TRUE is returned for the variant; this may result in a value of TRUE for a variant with a mixture of alternate alleles, some that pass the criteria and some that do not. To retain single ref / alt pairs that pass the critera use expand on the CollapsedVCF and then apply the test.
...	Arguments passed to other methods.

Details

All functions return a logical vector the length of x. Variants in gvcf files with NON_REF alt alleles return TRUE; structural variants return FALSE.

- isSNV: Reference and alternate alleles are both a single nucleotide long.
- isInsertion: Reference allele is a single nucleotide and the alternate allele is greater (longer) than a single nucleotide and the first nucleotide of the alternate allele matches the reference.
- isDeletion: Alternate allele is a single nucleotide and the reference allele is greater (longer) than a single nucleotide and the first nucleotide of the reference allele matches the alternate.
- isIndel: The variant is either a deletion or insertion as determined by isDeletion and isInsertion.
- isDelins: The variant is a deletion followed by an insertion, either of them involving two or more nucleotides.
- isSubstition: Reference and alternate alleles are the same length (1 or more nucleotides long).
- isTransition: Reference and alternate alleles are both a single nucleotide long. The reference-alternate pair interchange is of either two-ring purines (A <-> G) or one-ring pyrimidines (C <-> T).

Value

A logical vector the same length as x.

Author(s)

Michael Lawrence, Valerie Obenchain and Robert Castelo

Examples

```
fl <- system.file("extdata", "ex2.vcf", package="VariantAnnotation")
## ------------------------------------------------------------------------
## VCF objects
## ------------------------------------------------------------------------
vcf <- readVcf(fl, "hg19")
```

```
DataFrame(ref(vcf), alt(vcf))

## This vcf has transitions in row 2 and 3. When 'singleAltOnly=TRUE'
## only the row 2 variant is identified:
isTransition(vcf)

## Both row 2 and 3 are identified when 'singleAltOnly=FALSE':
isTransition(vcf, singleAltOnly=FALSE)

## Expand the CollapsedVCF to ExpandedVCF
evcf <- expand(vcf)

## All ref / alt pairs are now expanded and there is no need to
## use 'singleAltOnly'. The return length is now 7 instead of 5:
transition <- isTransition(evcf)
transition
DataFrame(ref(evcf)[transition], alt(evcf)[transition])

## ---------------------------------------------------------------------
## VRanges objects
## ---------------------------------------------------------------------
## A VRanges object holds data from a VCF class in a completely
## 'flat' fashion. INFO and FORMAT variables for all subjects are
## 'repped' out such that each row is a unique combination of data.
vr <- as(vcf, "VRanges")
isSNV(vr, singleAltOnly=FALSE)
```

`locateVariants` *Locate variants*

Description

Variant location with respect to gene function

Usage

```
locateVariants(query, subject, region, ...)
## S4 method for signature 'VCF,TxDb,VariantType'
locateVariants(query, subject, region, ...,
    cache=new.env(parent=emptyenv()), ignore.strand=FALSE, asHits=FALSE)
## S4 method for signature 'GRanges,TxDb,VariantType'
locateVariants(query, subject, region, ...,
    cache=new.env(parent=emptyenv()), ignore.strand=FALSE, asHits=FALSE)
```

Arguments

`query`	A IntegerRanges, GRanges or VCF object containing the variants. Metadata columns are allowed but ignored. NOTE: Zero-width ranges are treated as width-1 ranges; start values are decremented to equal the end value.

subject	A TxDb or GRangesList object that serves as the annotation. GFF files can be converted to TxDb objects with makeTxDbFromGFF() in the txdbmaker package.
region	An instance of one of the 8 VariantType classes: CodingVariants, IntronVariants, FiveUTRVariants, ThreeUTRVariants, IntergenicVariants, SpliceSiteVariants, PromoterVariants, AllVariants. All objects can be instantiated with no arguments, e.g., CodingVariants() will create an object of CodingVariants. AllVariants, PromoterVariants and IntergenicVariants have upstream and downstream arguments. For PromoterVariants and IntergenicVariants these are single integer values >= 0. For AllVariants these are integer vectors of length 2 named 'promoter' and 'intergenic'. See ?upstream for more details. When using AllVariants, a range in query may fall in multiple regions (e.g., 'intergenic' and 'promoter'). In this case the result will have a row for each match. All data in the row will be equivalent except the LOCATION column.
...	Additional arguments passed to methods
cache	An environment into which required components of subject are loaded. Provide, and re-use, a cache to speed repeated queries to the same subject across different query instances.
ignore.strand	A logical indicating if strand should be ignored when performing overlaps.
asHits	A logical indicating if the results should be returned as a Hits object. Not applicable when region is AllVariants or IntergenicVariants.

Details

Range representation: The ranges in query should reflect the position(s) of the reference allele. For snps the range will be of width 1. For range insertions or deletions the reference allele could be a sequence such as GGTG in which case the width of the range should be 4.

Location: Possible locations are 'coding', 'intron', 'threeUTR', 'fiveUTR', 'intergenic', 'spliceSite', or 'promoter'.

Overlap operations for 'coding', 'intron', 'threeUTR', and 'fiveUTR' require variants to fall completely within the defined region to be classified as such.

To be classified as a 'spliceSite' the variant must overlap with any portion of the first 2 or last 2 nucleotides in an intron.

'intergenic' variants are ranges that do not fall within a defined gene region. 'transcripts by gene' are extracted from the annotation and overlapped with the variant positions. Variants with no overlaps are classified as intergenic. When available, gene IDs for the flanking genes are provided as PRECEDEID and FOLLOWID. upstream and downstream arguments define the acceptable distance from the query for the flanking genes. PRECEDEID and FOLLOWID results are lists and contain all genes that fall within the defined distance. See the examples for how to compute the distance from ranges to PRECEDEID and FOLLOWID.

'promoter' variants fall within a specified range upstream and downstream of the transcription start site. Ranges values can be set with the upstream and downstream arguments when creating the PromoterVariants() or AllVariants() classes.

Subject as GRangesList: The subject can be a TxDb or GRangesList object. When using a GRangesList the type of data required is driven by the VariantType class. Below is a description of the appropriate GRangesList for each VariantType.

CodingVariants: coding (CDS) by transcript

IntronVariants: introns by transcript

FiveUTRVariants: five prime UTR by transcript

ThreeUTRVariants: three prime UTR by transcript

IntergenicVariants: transcripts by gene

SpliceSiteVariants: introns by transcript

PromoterVariants: list of transcripts

AllVariants: no GRangeList method available

Using the cache: When processing multiple VCF files performance is enhanced by specifying an environment as the `cache` argument. This cache is used to store and reuse extracted components of the subject (TxDb) required by the function. The first call to the function (i.e., processing the first VCF file in a list of many) populates the cache; repeated calls to `locateVariants` will access these objects from the cache vs re-extracting the same information.

Value

A `GRanges` object with a row for each variant-transcript match. Strand of the output is from the `subject` hit except in the case of IntergenicVariants. For intergenic, multiple precede and follow gene ids are returned for each variant. When `ignore.strand=TRUE` the return strand is `*` because genes on both strands are considered and it is possible to have a mixture. When `ignore.strand=FALSE` the strand will match the `query` because only genes on the same strand are considered.

Metadata columns are `LOCATION`, `QUERYID`, `TXID`, `GENEID`, `PRECEDEID`, `FOLLOWID` and `CDSID`. Results are ordered by `QUERYID`, `TXID` and `GENEID`. Columns are described in detail below.

`LOCATION` Possible locations are 'coding', 'intron', 'threeUTR', 'fiveUTR', 'intergenic', 'spliceSite' and 'promoter'.

To be classified as 'coding', 'intron', 'threeUTR' or 'fiveUTR' the variant must fall completely within the region.

'intergenic' variants do not fall within a transcript. The 'GENEID' for these positions are `NA`. Lists of flanking genes that fall within the distance defined by `upstream` and `downstream` are given as 'PRECEDEID' and 'FOLLOWID'. By default, the gene ID is returned in the 'PRECEDEID' and 'FOLLOWID' columns. To return the transcript ids instead set `idType = "tx"` in the `IntergenicVariants()` constructor.

A 'spliceSite' variant overlaps any portion of the first 2 or last 2 nucleotides of an intron.

`LOCSTART, LOCEND` Genomic position in LOCATION-centric coordinates. If LOCATION is 'intron', these are intron-centric coordinates, if LOCATION is 'coding' then cds-centric. All coordinates are relative to the start of the transcript. SpliceSiteVariants, IntergenicVariants and PromoterVariants have no formal extraction 'by transcript' so for these variants LOCSTART and LOCEND are NA. Coordinates are computed with `mapToTranscripts`; see `?mapToTranscripts` in the GenomicFeatures package for details.

`QUERYID` The `QUERYID` column provides a map back to the row in the original `query`. If the `query` was a `VCF` object this index corresponds to the row in the `GRanges` object returned by the `rowRanges` accessor.

`TXID` The transcript id taken from the `TxDb` object.

`CDSID` The coding sequence id(s) taken from the `TxDb` object.

GENEID The gene id taken from the TxDb object.

PRECEDEID IDs for all genes the query precedes within the defined upstream and downstream distance. Only applicable for 'intergenic' variants. By default this column contains gene ids; to return transcript ids set idType = "tx" in the IntergenicVariants constructor.

FOLLOWID IDs for all genes the query follows within the defined upstream and downstream distance. Only applicable for 'intergenic' variants. By default this column contains gene ids; to return transcript ids set idType = "tx" in the IntergenicVariants constructor.

All ID values will be 'NA' for variants with a location of transcript_region or NA.

Author(s)

Valerie Obenchain

See Also

- The readVcf function.
- The predictCoding function.
- The promoters function on the intra-range-methods man page in the **GenomicRanges** package.

Examples

```
library(TxDb.Hsapiens.UCSC.hg19.knownGene)
txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene

## ------------------------------------------------------------------------
## Variants in all gene regions
## ------------------------------------------------------------------------
## Read variants from a VCF file.
fl <- system.file("extdata", "gl_chr1.vcf",
                  package="VariantAnnotation")
vcf <- readVcf(fl, "hg19")

## Often the seqlevels in the VCF file do not match those in the TxDb.
head(seqlevels(vcf))
head(seqlevels(txdb))
intersect(seqlevels(vcf), seqlevels(txdb))

## Rename seqlevels with renameSeqlevesl().
seqlevels(vcf) <- paste0("chr", seqlevels(vcf))

## Confirm.
intersect(seqlevels(vcf), seqlevels(txdb))

## Overlaps for all possible variant locations.
loc_all <- locateVariants(vcf, txdb, AllVariants())
table(loc_all$LOCATION)

## ------------------------------------------------------------------------
## Variants in intergenic regions
```

```
  ## ------------------------------------------------------------------------
  ## Intergenic variants do not overlap a gene range in the
  ## annotation and therefore 'GENEID' is always NA. Flanking genes
  ## that fall within the 'upstream' and 'downstream' distances are
  ## reported as PRECEDEID and FOLLOWID.
  region <- IntergenicVariants(upstream=70000, downstream=70000)
  loc_int <- locateVariants(vcf, txdb, region)
  mcols(loc_int)[c("LOCATION", "PRECEDEID", "FOLLOWID")]

  ## Distance to the flanking genes can be computed for variants that
  ## have PRECEDEID(s) or FOLLOWID(s). Each variant can have multiple
  ## flanking id's so we first expand PRECEDEID and the corresponding
  ## variant ranges.
  p_ids <- unlist(loc_int$PRECEDEID, use.names=FALSE)
  exp_ranges <- rep(loc_int,  elementNROWS(loc_int$PRECEDEID))

  ## Compute distances with the distance method defined in GenomicFeatures.
  ## Help page can be found at ?`distance,GenomicRanges,TxDb-method`.
  ## The method returns NA for ids that cannot be collapsed into a single
  ## range (e.g., genes with ranges on multiple chromosomes).
  distance(exp_ranges, txdb, id=p_ids, type="gene")

  ## To search for distance by transcript id set idType='tx' in the
  ## IntergenicVariants() constructor, e.g.,
  ## locateVariants(vcf, txdb, region=IntergenicVariants(idType="tx"))

  ## Unlist ids and expand ranges as before to get p_ids and exp_ranges.
  ## Then call distance() with type = "tx":
  ## distance(exp_ranges, txdb, id=p_ids, type="tx")


  ## ------------------------------------------------------------------------
  ## GRangesList as subject
  ## ------------------------------------------------------------------------
  ## When 'subject' is a GRangesList the GENEID is unavailable and
  ## will always be reported as NA. This is because the GRangesList
  ## objects are extractions of region-by-transcript, not region-by-gene.
  ## Not run:
  cdsbytx <- cdsBy(txdb)
  locateVariants(vcf, cdsbytx, CodingVariants())

  intbytx <- intronsByTranscript(txdb)
  locateVariants(vcf, intbytx, IntronVariants())

## End(Not run)

  ## ------------------------------------------------------------------------
  ## Using the cache
  ## ------------------------------------------------------------------------
  ## When processing multiple VCF files, the 'cache' can be used
  ## to store the extracted components of the TxDb
  ## (i.e., cds by tx, introns by tx etc.). This avoids having to
  ## re-extract these GRangesLists during each loop.
```

```
  ## Not run:
  myenv <- new.env()
  files <- list(vcf1, vcf2, vcf3)
  lapply(files,
      function(fl) {
          vcf <- readVcf(fl, "hg19")
          ## modify seqlevels to match TxDb
          seqlevels(vcf_mod) <- paste0("chr", seqlevels(vcf))
          locateVariants(vcf_mod, txdb, AllVariants(), cache=myenv)
      })

## End(Not run)

  ## ------------------------------------------------------------------------
  ## Parallel implmentation
  ## ------------------------------------------------------------------------
  ## Not run:
  library(BiocParallel)

  ## A connection to a TxDb object is established when
  ## the package is loaded. Because each process reading from an
  ## sqlite db must have a unique connection the TxDb
  ## object cannot be passed as an argument when running in
  ## parallel. Instead the package must be loaded on each worker.

  ## The overhead of the multiple loading may defeat the
  ## purpose of running the job in parallel. An alternative is
  ## to instead pass the appropriate GRangesList as an argument.
  ## The details section on this man page under the heading
  ## 'Subject as GRangesList' explains what GRangesList is
  ## appropriate for each variant type.

  ## A. Passing a GRangesList:

  fun <- function(x, subject, ...)
      locateVariants(x, subject, IntronVariants())

  library(TxDb.Hsapiens.UCSC.hg19.knownGene)
  grl <- intronsByTranscript(TxDb.Hsapiens.UCSC.hg19.knownGene)
  mclapply(c(vcf, vcf), fun, subject=grl)


  ## B. Passing a TxDb:

  ## Forking:
  ## In the case of forking, the TxDb cannot be loaded
  ## in the current workspace.
  ## To detach the NAMESPACE:
  ##     unloadNamespace("TxDb.Hsapiens.UCSC.hg19.knownGene")

  fun <- function(x) {
      library(TxDb.Hsapiens.UCSC.hg19.knownGene)
      locateVariants(x, TxDb.Hsapiens.UCSC.hg19.knownGene,
```

```
                    IntronVariants())
  }
  mclapply(c(vcf, vcf), fun)

  ## Clusters:
  cl <- makeCluster(2, type = "SOCK")
  fun <- function(query, subject, region) {
      library(VariantAnnotation)
      library(TxDb.Hsapiens.UCSC.hg19.knownGene)
      locateVariants(query, TxDb.Hsapiens.UCSC.hg19.knownGene, region)
  }
  parLapply(cl, c(vcf, vcf), fun, region=IntronVariants())
  stopCluster(cl)

## End(Not run)
```

PolyPhenDb-class *PolyPhenDb objects*

Description

The PolyPhenDb class is a container for storing a connection to a PolyPhen sqlite database.

Details

PolyPhen (Polymorphism Phenotyping) is a tool which predicts the possible impact of an amino acid substitution on the structure and function of a human protein by applying empirical rules to the sequence, phylogenetic and structural information characterizing the substitution.

PolyPhen makes its predictions using UniProt features, PSIC profiles scores derived from multiple alignment and matches to PDP or PQS structural databases. The procedure can be roughly outlined in the following steps, see the references for complete details,

- sequence-based characterization of substitution site
- calculation of PSIC profile scores for two amino acid variants
- calculation of structural parameters and contacts
- prediction

PolyPhen uses empirically derived rules to predict that a non-synonymous SNP is

- probably damaging : it is with high confidence supposed to affect protein function or structure
- possibly damaging : it is supposed to affect protein function or structure
- benign : most likely lacking any phenotypic effect
- unknown : when in some rare cases, the lack of data do not allow PolyPhen to make a prediction

Methods

In the code below, x is a PolyPhenDb object.

`metadata(x)`: Returns x's metadata in a data frame.

`columns(x)`: Returns the names of the columns that can be used to subset the data columns. For column descriptions see ?PolyPhenDbColumns.

`keys(x)`: Returns the names of the keys that can be used to subset the data rows. The keys values are the rsid's.

`select(x, keys = NULL, columns = NULL, ...)`: Returns a subset of data defined by the character vectors keys and columns. If no keys are supplied, all rows are returned. If no columns are supplied, all columns are returned. See ?PolyPhenDbColumns for column descriptions.

`duplicateRSID(x)`: Returns a named list of duplicate rsid groups. The names are the keys, the list elements are the rsid's that have been reported as having identical chromosome position and alleles and therefore translating into the same amino acid residue substitution.

Author(s)

Valerie Obenchain

References

PolyPhen Home: http://genetics.bwh.harvard.edu/pph2/dokuwiki/

Adzhubei IA, Schmidt S, Peshkin L, Ramensky VE, Gerasimova A, Bork P, Kondrashov AS, Sunyaev SR. Nat Methods 7(4):248-249 (2010).

Ramensky V, Bork P, Sunyaev S. Human non-synonymous SNPs: server and survey. Nucleic Acids Res 30(17):3894-3900 (2002).

Sunyaev SR, Eisenhaber F, Rodchenkov IV, Eisenhaber B, Tumanyan VG, Kuznetsov EN. PSIC: profile extraction from sequence alignments with position-specific counts of independent observations. Protein Eng 12(5):387-394 (1999).

See Also

?PolyPhenDbColumns

Examples

```
library(PolyPhen.Hsapiens.dbSNP131)

## metadata
metadata(PolyPhen.Hsapiens.dbSNP131)

## available rsid's
head(keys(PolyPhen.Hsapiens.dbSNP131))

## column descriptions found at ?PolyPhenDbColumns
columns(PolyPhen.Hsapiens.dbSNP131)

## subset on keys and columns
```

```
subst <- c("AA1", "AA2", "PREDICTION")
rsids <- c("rs2142947", "rs4995127", "rs3026284")
select(PolyPhen.Hsapiens.dbSNP131, keys=rsids, columns=subst)

## retrieve substitution scores
subst <- c("IDPMAX", "IDPSNP", "IDQMIN")
select(PolyPhen.Hsapiens.dbSNP131, keys=rsids, columns=subst)

## retrieve the PolyPhen-2 classifiers
subst <- c("PPH2CLASS", "PPH2PROB", "PPH2FPR", "PPH2TPR", "PPH2FDR")
select(PolyPhen.Hsapiens.dbSNP131, keys=rsids, columns=subst)

## duplicate groups of rsid's
duplicateRSID(PolyPhen.Hsapiens.dbSNP131, c("rs71225486", "rs1063796"))
```

`PolyPhenDbColumns` *PolyPhenDb Columns*

Description

Description of the PolyPhen Sqlite Database Columns

Column descriptions

These column names are displayed when `columns` is called on a `PolyPhenDb` object.

- rsid : rsid

Original query :

- OSNPID : original SNP identifier from user input
- OSNPACC : original protein identifier from user input
- OPOS : original substitution position in the protein sequence from user input
- OAA1 : original wild type (reference) aa residue from user input
- OAA2 : original mutant (reference) aa residue from user input

Mapped query :

- SNPID : SNP identifier mapped to dbSNP rsID if available, otherwise same as o_snp_id. This value was used as the rsid column
- ACC : protein UniProtKB accession if known protein, otherwise same as o_acc
- POS : substitution position mapped to UniProtKB protein sequence if known, otherwise same as o_pos
- AA1 : wild type aa residue
- AA2 : mutant aa residue
- NT1 : wild type allele nucleotide

- NT2 : mutant allele nucleotide

PolyPhen-2 prediction :

- PREDICTION : qualitative ternary classification FPR thresholds

PolyPhen-1 prediction :

- BASEDON : prediction basis
- EFFECT : predicted substitution effect on the protein structure or function

PolyPhen-2 classifiers :

- PPH2CLASS : binary classifier outcome ("damaging" or "neutral")
- PPH2PROB : probability of the variation being dammaging
- PPH2FPR : false positive rate at the pph2_prob level
- PPH2TPR : true positive rate at the pph2_prob level
- PPH2FDR : false discovery rate at the pph2_prob level

UniProtKB-SwissProt derived protein sequence annotations :

- SITE : substitution SITE annotation
- REGION : substitution REGION annotation
- PHAT : PHAT matrix element for substitution in the TRANSMEM region

Multiple sequence alignment scores :

- DSCORE : difference of PSIC scores for two aa variants (Score1 - Score2)
- SCORE1 : PSIC score for wild type aa residue (aa1)
- SCORE2 : PSIC score for mutant aa residue (aa2)
- NOBS : number of residues observed at the substitution position in the multiple alignment (sans gaps)

Protein 3D structure features :

- NSTRUCT : initial number of BLAST hits to similar proteins with 3D structures in PDB
- NFILT : number of 3D BLAST hits after identity threshold filtering
- PDBID : protein structure identifier from PDB
- PDBPOS : position of substitution in PDB protein sequence
- PDBCH : PDB polypeptide chain identifier
- IDENT : sequence identity between query and aligned PDB sequences
- LENGTH : PDB sequence alignment length
- NORMACC : normalized accessible surface
- SECSTR : DSSP secondary structure assignment
- MAPREG : region of the phi-psi (Ramachandran) map derived from the residue dihedral angles
- DVOL : change in residue side chain volume

- DPROP : change in solvent accessible surface propensity resulting from the substitution
- BFACT : normalized B-factor (temperature factor) for the residue
- HBONDS : number of hydrogen sidechain-sidechain and sidechain-mainchain bonds formed by the residue
- AVENHET : average number of contacts with heteroatoms per residue
- MINDHET : closest contact with heteroatom
- AVENINT : average number of contacts with other chains per residue
- MINDINT : closest contact with other chain
- AVENSIT : average number of contacts with critical sites per residue
- MINDSIT : closest contact with a critical site

Nucleotide sequence features (CpG/codon/exon junction) :

- TRANSV : whether substitution is a transversion
- CODPOS : position of the substitution within the codon
- CPG : whether or not the substitution changes CpG context
- MINDJNC : substitution distance from exon/intron junction

Pfam protein family :

- PFAMHIT : Pfam identifier of the query protein

Substitution scores :

- IDPMAX : maximum congruency of the mutant aa residue to all sequences in multiple alignment
- IDPSNP : maximum congruency of the mutant aa residue to the sequence in alignment with the mutant residue
- IDQMIN : query sequence identity with the closest homologue deviating from the wild type aa residue

Comments :

- COMMENTS : Optional user comments

Author(s)

Valerie Obenchain

See Also

`?PolyPhenDb`

post_Hs_region *elementary vep/homo_sapiens/region call to ensembl VEP REST API*

Description

elementary vep/homo_sapiens/region call to ensembl VEP REST API

Usage

```
post_Hs_region(chr, pos, id, ref, alt)
```

Arguments

chr	character(1) ensembl chromosome identifier (e.g., "7")
pos	numeric(1) 1-based chromosome position
id	character(1) arbitrary identifier
ref	character(1) reference allele
alt	character(1) alternative allele

Value

Instance of 'response' defined in httr package.

Note

This function prepares a POST to rest.ensembl.org/vep/homo_sapiens/region endpoint.

Examples

```
chk = post_Hs_region("7", 155800001, "chk", "A", "T")
chk
res = jsonlite::fromJSON(jsonlite::toJSON(httr::content(chk)))
dim(chk)
```

predictCoding *Predict amino acid coding changes for variants*

Description

Predict amino acid coding changes for variants a coding regions

Usage

```
## S4 method for signature 'CollapsedVCF,TxDb,ANY,missing'
predictCoding(query, subject, seqSource, varAllele, ..., ignore.strand=FALSE)
## S4 method for signature 'ExpandedVCF,TxDb,ANY,missing'
predictCoding(query, subject, seqSource, varAllele, ..., ignore.strand=FALSE)
## S4 method for signature 'IntegerRanges,TxDb,ANY,DNAStringSet'
predictCoding(query, subject, seqSource, varAllele, ..., ignore.strand=FALSE)
## S4 method for signature 'GRanges,TxDb,ANY,DNAStringSet'
predictCoding(query, subject, seqSource, varAllele, ..., ignore.strand=FALSE)
## S4 method for signature 'VRanges,TxDb,ANY,missing'
predictCoding(query, subject, seqSource, varAllele, ..., ignore.strand=FALSE)
```

Arguments

query	A VCF, IntegerRanges, GRanges or VRanges instance containing the variants to be annotated. If query is a IntegerRanges or VRanges it is coerced to a GRanges. If a VCF is provided the GRanges returned by the rowRanges() accessor will be used. All metadata columns are ignored. When query is not a VCF object a varAllele must be provided. The varAllele must be a DNAStringSet the same length as the query. If there are multiple alternate alleles per variant the query must be expanded to one row per each alternate allele. See examples. NOTE: Variants are expected to conform to the VCF specs as described on the 1000 Genomes page (see references). Indels must include the reference allele; zero-width ranges are not valid and return no matches.
subject	A TxDb object that serves as the annotation. GFF files can be converted to TxDb objects with makeTxDbFromGFF() in the txdbmaker package.
seqSource	A BSgenome instance or an FaFile to be used for sequence extraction.
varAllele	A DNAStringSet containing the variant (alternate) alleles. The length of varAllele must equal the length of query. Missing values are represented by a zero width empty element. Ranges with missing varAllele values are ignored; those with an 'N' character are not translated. When the query is a VCF object the varAllele argument will be missing. This value is taken internally from the VCF with alt(<VCF>).
...	Additional arguments passed to methods. Arguments genetic.code and if.fuzzy.codon are supported for the translate function.
ignore.strand	A logical indicating if strand should be ignored when performing overlaps. When ignore.strand == TRUE the query strand is set to '*' and can overlap with any strand of the subject. The return GRanges reflects the strand of the subject hit, however, the positions and alleles reported are computed as if both were from the '+' strand. ignore.strand == FALSE requires the query and subject to have compatible strand. Again the return GRanges reports the strand of the subject hit but in this case positions and alleles are computed according to the strand of the subject.

Details

This function returns the amino acid coding for variants that fall completely 'within' a coding region. The reference sequences are taken from a fasta file or BSgenome. The width of the reference is determined from the start position and width of the range in the `query`. For guidance on how to represent an insertion, deletion or substitution see the 1000 Genomes VCF format (references).

Variant alleles are taken from the `varAllele` when supplied. When the `query` is a `VCF` object the `varAllele` will be missing. This value is taken internally from the `VCF` with `alt(<VCF>)`. The variant allele is substituted into the reference sequences and transcribed. Transcription only occurs if the substitution, insertion or deletion results in a new sequence length divisible by 3.

When the `query` is an unstranded (*) `GRanges` `predictCoding` will attempt to find overlaps on both the positive and negative strands of the `subject`. When the subject hit is on the negative strand the return `varAllele` is reverse complemented. The strand of the returned `GRanges` represents the strand of the subject hit.

Value

A GRanges with a row for each variant-transcript match. The result includes only variants that fell within coding regions. The strand of the output `GRanges` represents the strand of the `subject` hit.

At a minimum, the metadata columns (accessible with `mcols`) include,

`varAllele` Variant allele. This value is reverse complemented for an unstranded `query` that overlaps a `subject` on the negative strand.

`QUERYID` Map back to the row in the original query

`TXID` Internal transcript id from the annotation

`CDSID` Internal coding region id from the annotation

`GENEID` Internal gene id from the annotation

`CDSLOC` Variant location in coding region-based coordinates. This position is relative to the start of the coding (cds) region defined in the `subject` annotation.

`PROTEINLOC` Variant codon triplet location in coding region-based coordinates. This position is relative to the start of the coding (cds) region defined in the `subject` annotation.

`CONSEQUENCE` Possible values are 'synonymous', 'nonsynonymous', 'frameshift', 'nonsense' and 'not translated'. Variant sequences are translated only when the codon sequence is a multiple of 3. The value will be 'frameshift' when a sequence is of incompatible length. 'not translated' is used when `varAllele` is missing or there is an 'N' in the sequence. 'nonsense' is used for premature stop codons.

`REFCODON` The reference codon sequence. This range is typically greater than the width of the range in the `GRanges` because it includes all codons involved in the sequence modification. If the reference sequence is of width 2 but the alternate allele is of width 4 then at least two codons must be included in the `REFSEQ`.

`VARCODON` This sequence is the result of inserting, deleting or replacing the position(s) in the reference sequence alternate allele. If the result of this substitution is not a multiple of 3 it will not be translated.

`REFAA` The reference amino acid column contains the translated `REFSEQ`. When translation is not possible this value is missing.

`VARAA` The variant amino acid column contains the translated `VARSEQ`. When translation is not possible this value is missing.

Author(s)

Michael Lawrence and Valerie Obenchain

References

http://www.1000genomes.org/wiki/analysis/variant-call-format/ http://vcftools.sourceforge.net/specs.html

See Also

readVcf, locateVariants, refLocsToLocalLocs getTranscriptSeqs

Examples

```
library(BSgenome.Hsapiens.UCSC.hg19)
library(TxDb.Hsapiens.UCSC.hg19.knownGene)
txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene

## ----------------------------
## VCF object as query
## ----------------------------
## Read variants from a VCF file
fl <- system.file("extdata", "chr22.vcf.gz", package="VariantAnnotation")
vcf <- readVcf(fl, "hg19")

## Rename seqlevels in the VCF object to match those in the TxDb.
seqlevels(vcf) <- "chr22"
## Confirm common seqlevels
intersect(seqlevels(vcf), seqlevels(txdb))

## When 'query' is a VCF object the varAllele argument is missing.
coding1 <- predictCoding(vcf, txdb, Hsapiens)
head(coding1, 3)

## Exon-centric or cDNA locations:
exonsbytx <- exonsBy(txdb, "tx")

cDNA <- mapToTranscripts(coding1, exonsbytx)
mcols(cDNA)$TXID <- names(exonsbytx)[mcols(cDNA)$transcriptsHits]
cDNA <- cDNA[mcols(cDNA)$TXID == mcols(coding1)$TXID[mcols(cDNA)$xHits]]

## Make sure cDNA is parallel to coding1
stopifnot(identical(mcols(cDNA)$xHits, seq_along(coding1)))

coding1$cDNA <- ranges(cDNA)

## ----------------------------
## GRanges object as query
## ----------------------------
## A GRanges can also be used as the 'query'. The seqlevels in the VCF
## were adjusted in previous example so the GRanges extracted with
## has the correct seqlevels.
```

```
rd <- rowRanges(vcf)

## The GRanges must be expanded to have one row per alternate allele.
## Variants 1, 2 and 10 have two alternate alleles.
altallele <- alt(vcf)
eltROWS <- elementNROWS(altallele)
rd_exp <- rep(rd, eltROWS)

## Call predictCoding() with the expanded GRanges and the unlisted
## alternate allele as the 'varAllele'.
coding2 <- predictCoding(rd_exp, txdb, Hsapiens, unlist(altallele))
```

`probabilityToSnpMatrix` *Convert posterior genotype probability to a SnpMatrix object*

Description

Convert a matrix of posterior genotype probabilites P(AA), P(AB), P(BB) to a SnpMatrix.

Usage

```
probabilityToSnpMatrix(probs)
```

Arguments

`probs`	Matrix with three columns for the posterior probabilities of the three genotypes: "P(A/A)", "P(A/B)", "P(B/B)". Each row must sum to 1.

Details

`probabilityToSnpMatrix` converts a matrix of posterior probabilites of genotype calls into a SnpMatrix.

Value

An object of class `"SnpMatrix"` with one row (one sample). Posterior probabilities are encoded (approximately) as byte values, one per SNP. See the help page for SnpMatrix for complete details of the class structure.

Author(s)

Stephanie Gogarten <sdmorris@u.washington.edu>

See Also

genotypeToSnpMatrix, SnpMatrix

Examples

```
probs <- matrix(c(1,0,0,
                  0,1,0,
                  0,0,1,
                  NA,NA,NA),
                  ncol=3, byrow=TRUE,
                  dimnames=list(1:4,c("A/A","A/B","B/B")))
sm <- probabilityToSnpMatrix(probs)
as(sm, "character")
```

PROVEANDb-class *PROVEANDb objects*

Description

The PROVEANDb class is a container for storing a connection to a PROVEAN sqlite database.

Details

The SIFT tool is no longer actively maintained. A few of the orginal authors have started the PROVEAN (Protein Variation Effect Analyzer) project. PROVEAN is a software tool which predicts whether an amino acid substitution or indel has an impact on the biological function of a protein. PROVEAN is useful for filtering sequence variants to identify nonsynonymous or indel variants that are predicted to be functionally important.

See the web pages for a complete description of the methods.

- PROVEAN Home: http://provean.jcvi.org/index.php/
- SIFT Home: http://sift.jcvi.org/

Though SIFT is not under active development, the PROVEAN team still provids the SIFT scores in the pre-computed downloads. This package, SIFT.Hsapiens.dbSNP137, contains both SIFT and PROVEAN scores. One notable difference between this and the previous SIFT database package is that keys in SIFT.Hsapiens.dbSNP132 are rs IDs whereas in SIFT.Hsapiens.dbSNP137 they are NCBI dbSNP IDs.

Methods

In the code below, x is a PROVEANDb object.

`metadata(x)`: Returns x's metadata in a data frame.

`columns(x)`: Returns the names of the columns that can be used to subset the data columns.

`keys(x, keytype="DBSNPID", ...)`: Returns the names of the keys that can be used to subset the data rows. For SIFT.Hsapiens.dbSNP137 the keys are NCBI dbSNP ids.

`keytypes(x)`: Returns the names of the columns that can be used as keys. For SIFT.Hsapiens.dbSNP137 the NCBI dbSNP ids are the only keytype.

`select(x, keys = NULL, columns = NULL, keytype = "DBSNPID", ...)`: Returns a subset of data defined by the character vectors keys and columns. If no keys are supplied, all rows are returned. If no columns are supplied, all columns are returned.

Author(s)

Valerie Obenchain

References

The PROVEAN tool has replaced SIFT: http://provean.jcvi.org/about.php

Choi Y, Sims GE, Murphy S, Miller JR, Chan AP (2012) Predicting the Functional Effect of Amino Acid Substitutions and Indels. PLoS ONE 7(10): e46688.

Choi Y (2012) A Fast Computation of Pairwise Sequence Alignment Scores Between a Protein and a Set of Single-Locus Variants of Another Protein. In Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine (BCB '12). ACM, New York, NY, USA, 414-417.

Kumar P, Henikoff S, Ng PC. Predicting the effects of coding non-synonymous variants on protein function using the SIFT algorithm. Nat Protoc. 2009;4(7):1073-81

Ng PC, Henikoff S. Predicting the Effects of Amino Acid Substitutions on Protein Function Annu Rev Genomics Hum Genet. 2006;7:61-80.

Ng PC, Henikoff S. SIFT: predicting amino acid changes that affect protein function. Nucleic Acids Res. 2003 Jul 1;31(13):3812-4.

Examples

```
if (require(SIFT.Hsapiens.dbSNP137)) {
    ## metadata
    metadata(SIFT.Hsapiens.dbSNP137)

    ## keys are the DBSNPID (NCBI dbSNP ID)
    dbsnp <- keys(SIFT.Hsapiens.dbSNP137)
    head(dbsnp)
    columns(SIFT.Hsapiens.dbSNP137)

    ## Return all columns. Note that the key, DBSNPID,
    ## is always returned.
    select(SIFT.Hsapiens.dbSNP137, dbsnp[10])
    ## subset on keys and cols
    cols <- c("VARIANT", "PROVEANPRED", "SIFTPRED")
    select(SIFT.Hsapiens.dbSNP137, dbsnp[20:23], cols)
}
```

---

readVcf *Read VCF files*

Description

Read Variant Call Format (VCF) files

Usage

```
  ## S4 method for signature 'TabixFile,ScanVcfParam'
readVcf(file, genome, param,
      ..., row.names=TRUE)
  ## S4 method for signature 'TabixFile,IntegerRangesList'
readVcf(file, genome, param,
      ..., row.names=TRUE)
  ## S4 method for signature 'TabixFile,GRanges'
readVcf(file, genome, param,
      ..., row.names=TRUE)
  ## S4 method for signature 'TabixFile,GRangesList'
readVcf(file, genome, param,
      ..., row.names=TRUE)
  ## S4 method for signature 'TabixFile,missing'
readVcf(file, genome, param,
      ..., row.names=TRUE)
  ## S4 method for signature 'character,ANY'
readVcf(file, genome, param,
      ..., row.names=TRUE)
  ## S4 method for signature 'character,missing'
readVcf(file, genome, param,
      ..., row.names=TRUE)
  ## S4 method for signature 'character,missing'
readVcf(file, genome, param,
      ..., row.names=TRUE)

## Lightweight functions to read a single variable
readInfo(file, x, param=ScanVcfParam(), ..., row.names=TRUE)
readGeno(file, x, param=ScanVcfParam(), ..., row.names=TRUE)
readGT(file, nucleotides=FALSE, param=ScanVcfParam(), ..., row.names=TRUE)

## Import wrapper
## S4 method for signature 'VcfFile,ANY,ANY'
import(con, format, text, ...)
```

Arguments

file	A VcfFile (synonymous with TabixFile) instance or character() name of the VCF file to be processed. When ranges are specified in param, file must be a VcfFile. Use of the VcfFile methods are encouraged as they are more efficient than the character() methods. See ?VcfFile, and ?indexVcf for help creating a VcfFile.
genome	A character or Seqinfo object. • character: Genome identifier as a single string or named character vector. Names of the character vector correspond to chromosome names in the file. This identifier replaces the genome information in the VCF Seqinfo (i.e.,

seqinfo(vcf)). When not provided, genome is taken from the VCF file header.

- Seqinfo: When genome is provided as a Seqinfo it is propagated to the VCF output. If seqinfo information can be obtained from the file, (i.e., seqinfo(scanVcfHeader(fl)) is not empty), the output Seqinfo is a product of merging the two.
  If a param (i.e., ScanVcfParam) is used in the call to readVcf, the seqlevels of the param ranges must be present in genome.

param	An instance of ScanVcfParam, GRanges, GRangesList or IntegerRangesList. VCF files can be subset on genomic coordinates (ranges) or elements in the VCF fields. Both genomic coordinates and VCF elements can be specified in a ScanVcfParam. See ?ScanVcfParam for details.
x	character name of single info or geno field to import. Applicable to readInfo and readGeno only.
row.names	A logical specifying if rownames should be returned. In the case of readVcf, rownames appear on the GRanges returned by the rowRanges accessor.
nucleotides	A logical indicating if genotypes should be returned as nucleotides instead of the numeric representation. Applicable to readGT only.
con	The VcfFile object to import.
format, text	Ignored.
...	Additional arguments, passed to methods. For import, the arguments are passed to readVcf.

Details

Data Import: **VCF object:** readVcf imports records from bzip compressed or uncompressed VCF files. Data are parsed into a VCF object using the file header information if available. To import a subset of ranges the VCF must have an index file. An index file can be created with bzip and indexVcf functions.

The readInfo, readGeno and readGT functions are lightweight versions of readVcf and import a single variable. The return object is a vector, matrix or CompressedList instead of a VCF class.

readVcf calls scanVcf, the details of which can be found with ?scanVcf.

Header lines (aka Meta-information): readVcf() reads and parses fields according to the multiplicity and data type specified in the header lines. Fields without header lines are skipped (not read or parsed). To see what fields are present in the header use scanVcfHeader(). See ?VCFHeader for more details.

Passing verbose = TRUE to readVcf() prints the fields with header lines that will be parsed by readVcf.

Data type: CHROM, POS, ID and REF fields are used to create the GRanges stored in the VCF object and accessible with the rowRanges accessor.

REF, ALT, QUAL and FILTER are parsed into the DataFrame in the fixed slot. Because ALT can have more than one value per variant it is represented as a DNAStringSetList. REF is a DNAStringSet, QUAL is numeric and FILTER is a character. Accessors include fixed, ref, alt, qual, and filt.

Data from the INFO field can be accessed with the `info` accessor. Genotype data (i.e., data immediately following the FORMAT field in the VCF) can be accessed with the `geno` accessor. INFO and genotype data types are determined according to the 'Number' and 'Type' information in the file header as follows:

'Number' should only be 0 when 'Type' is 'flag'. These fields are parsed as logical vectors.

If 'Number' is 1, 'info' data are parsed into a `vector` and 'geno' into a `matrix`.

If 'Number' is >1, 'info' data are parsed into a `DataFrame` with the same number of columns. 'geno' are parsed into an `array` with the same dimensions as 'Number'. Columns of the 'geno' matrices are the samples.

If 'Number' is '.', 'A' or 'G', both 'info' and 'geno' data are parsed into a `matrix`.

When the header does not contain any 'INFO' lines, the data are returned as a single, unparsed column.

Missing data: Missing data in VCF files on disk are represented by a dot ("."). `readVcf` retains the dot as a character string for data type character and converts it to `NA` for data types numeric or double.

Because the data are stored in rectangular data structures there is a value for each `info` and `geno` field element in the `VCF` class. If the element was missing or was not collected for a particular variant the value will be `NA`.

In the case of the ALT field we have the following treatment of special characters / missing values:

- '.' true missings become empty characters
- '*' are treated as missing and become empty characters
- 'I' are treated as undefined and become '.'

Efficient Usage: Subsets of data (i.e., specific variables, positions or samples) can be read from a VCF file by providing a `ScanVcfParam` object in the call to `readVcf`. Other lightweight options are the `readGT`, `readInfo` and `readGeno` functions which return data as a matrix instead of the `VCF` class.

Another option for handling large files is to iterate through the data in chunks by setting the `yieldSize` parameter in a `VcfFile` object. Iteration can be through all data fields or a subset defined by a `ScanVcfParam`. See example below, 'Iterating through VCF with yieldSize'.

Value

`readVcf` returns a `VCF` object. See ?`VCF` for complete details of the class structure. `readGT`, `readInfo` and `readGeno` return a `matrix`.

rowRanges: The CHROM, POS, ID and REF fields are used to create a `GRanges` object. Ranges are created using POS as the start value and width of the reference allele (REF). By default, the IDs become the rownames ('row.names = FALSE' to turn this off). If IDs are missing (i.e., '.') a string of CHROM:POS_REF/ALT is used instead. The `genome` argument is stored in the seqinfo of the `GRanges` and can be accessed with `genome(<VCF>)`.

One metadata column, `paramRangeID`, is included with the `rowRanges`. This ID is meaningful when multiple ranges are specified in the `ScanVcfParam` and distinguishes which records match each range.

fixed: REF, ALT, QUAL and FILTER fields of the VCF are parsed into a `DataFrame`.

REF is returned as a DNAStringSet.

ALT is a CharacterList when it contains structural variants and a DNAStringSetList otherwise. See also the 'Details' section for 'Missing data'.

info: Data from the INFO field of the VCF is parsed into a `DataFrame`.

geno: If present, the genotype data are parsed into a list of `matrices` or `arrays`. Each list element represents a field in the FORMAT column of the VCF file. Rows are the variants, columns are the samples.

colData: This slot contains a `DataFrame` describing the samples. If present, the sample names following FORMAT in the VCF file become the row names.

metadata: Header information present in the file is put into a `list` in `metadata`.

See references for complete details of the VCF file format.

Author(s)

Valerie Obenchain>

References

http://vcftools.sourceforge.net/specs.html outlines the VCF specification.

http://samtools.sourceforge.net/mpileup.shtml contains information on the portion of the specification implemented by `bcftools`.

http://samtools.sourceforge.net/ provides information on `samtools`.

See Also

indexVcf, VcfFile, indexTabix, TabixFile, scanTabix, scanBcf, expand,CollapsedVCF-method

Examples

```
fl <- system.file("extdata", "ex2.vcf", package="VariantAnnotation")
vcf <- readVcf(fl, "hg19")
## vcf <- readVcf(fl, c("20"="hg19"))  ## 'genome' as named vector

## --------------------------------------------------------------------
## Header and genome information
## --------------------------------------------------------------------
vcf

## all header information
hdr <- header(vcf)

## header information for 'info' and 'fixed' fields
info(hdr)
fixed(hdr)

## --------------------------------------------------------------------
## Accessors
## --------------------------------------------------------------------
## fixed fields together
```

```
head(fixed(vcf), 5)

## fixed fields separately
filt(vcf)
ref(vcf)

## info data
info(hdr)
info(vcf)
info(vcf)$DP

## geno data
geno(hdr)
geno(vcf)
head(geno(vcf)$GT)

## genome
unique(genome(rowRanges(vcf)))

## ---------------------------------------------------------------------
## Data subsets with lightweight read* functions
## ---------------------------------------------------------------------

## Import a single 'info' or 'geno' variable
DP <- readInfo(fl, "DP")
HQ <- readGeno(fl, "HQ")

## Import GT as numeric representation
GT <- readGT(fl)
## Import GT as nucleotides
GT <- readGT(fl, nucleotides=TRUE)

## ---------------------------------------------------------------------
## Data subsets with ScanVcfParam
## ---------------------------------------------------------------------

## Subset on genome coordinates:
## 'file' must have an index
rngs <- GRanges("20", IRanges(c(14370, 1110000), c(17330, 1234600)))
names(rngs) <- c("geneA", "geneB")
param <- ScanVcfParam(which=rngs)
compressVcf <- bgzip(fl, tempfile())
tab <- indexVcf(compressVcf)
vcf <- readVcf(tab, "hg19", param)

## When data are subset by range ('which' argument in ScanVcfParam),
## the 'paramRangeID' column provides a map back to the original
## range in 'param'.
rowRanges(vcf)[,"paramRangeID"]
vcfWhich(param)

## Subset on samples:
## Consult the header for the sample names.
```

```
samples(hdr)
## Specify one or more names in 'samples' in a ScanVcfParam.
param <- ScanVcfParam(samples="NA00002")
vcf <- readVcf(tab, "hg19", param)
geno(vcf)$GT

## Subset on 'fixed', 'info' or 'geno' fields:
param <- ScanVcfParam(fixed="ALT", geno=c("GT", "HQ"), info=c("NS", "AF"))
vcf_tab <- readVcf(tab, "hg19", param)
info(vcf_tab)
geno(vcf_tab)

## No ranges are specified in the 'param' so tabix file is not
## required. Instead, the uncompressed VCF can be used as 'file'.
vcf_fname <- readVcf(fl, "hg19", param)

## The header will always contain information for all variables
## in the original file reguardless of how the data were subset.
## For example, all 'geno' fields are listed in the header
geno(header(vcf_fname))

## but only 'GT' and 'HQ' are present in the VCF object.
geno(vcf_fname)

## Subset on both genome coordinates and 'info', 'geno' fields:
param <- ScanVcfParam(geno="HQ", info="AF", which=rngs)
vcf <- readVcf(tab, "hg19", param)

## When any of 'fixed', 'info' or 'geno' are omitted (i.e., no
## elements specified) all records are retrieved. Use NA to indicate
## that no records should be retrieved. This param specifies
## all 'fixed fields, the "GT" 'geno' field and none of 'info'.
ScanVcfParam(geno="GT", info=NA)

## ---------------------------------------------------------------------
## Iterate through VCF with 'yieldSize'
## ---------------------------------------------------------------------
fl <- system.file("extdata", "chr22.vcf.gz", package="VariantAnnotation")
param <- ScanVcfParam(fixed="ALT", geno=c("GT", "GL"), info=c("LDAF"))
tab <- VcfFile(fl, yieldSize=4000)
open(tab)
while (nrow(vcf_yield <- readVcf(tab, "hg19", param=param)))
    cat("vcf dim:", dim(vcf_yield), "\n")
close(tab)

## ---------------------------------------------------------------------
## Debugging with 'verbose'
## ---------------------------------------------------------------------
## readVcf() uses information in the header lines to parse the data to
## the correct number and type. Fields without header lines are skipped.
## If a call to readVcf() results in no info(VCF) or geno(VCF) data the
## file may be missing header lines. Set 'verbose = TRUE' to get
## a listing of fields found in the header.
```

```
## readVcf(myfile, "mygenome", verbose=TRUE)

## Header fields can also be discovered with scanVcfHeader().
hdr <- scanVcfHeader(fl)
geno(hdr)
```

---

scanVcf *Import VCF files*

Description

Import Variant Call Format (VCF) files in text or binary format

Usage

```
scanVcfHeader(file, ...)
## S4 method for signature 'character'
scanVcfHeader(file, ...)

scanVcf(file, ..., param)
## S4 method for signature 'character,ScanVcfParam'
scanVcf(file, ..., param)
## S4 method for signature 'character,missing'
scanVcf(file, ..., param)
## S4 method for signature 'connection,missing'
scanVcf(file, ..., param)

## S4 method for signature 'TabixFile'
scanVcfHeader(file, ...)
## S4 method for signature 'TabixFile,missing'
scanVcf(file, ..., param)
## S4 method for signature 'TabixFile,ScanVcfParam'
scanVcf(file, ..., param)
## S4 method for signature 'TabixFile,GRanges'
scanVcf(file, ..., param)
## S4 method for signature 'TabixFile,IntegerRangesList'
scanVcf(file, ..., param)
```

Arguments

file	For scanVcf and scanVcfHeader, the character() file name, TabixFile, or class connection (file() or bgzip()) of the 'VCF' file to be processed.
param	A instance of ScanVcfParam influencing which records are parsed and the 'INFO' and 'GENO' information returned.
...	Additional arguments for methods

Details

The argument `param` allows portions of the file to be input, but requires that the file be bgzip'd and indexed as a `TabixFile`.

`scanVcf` with `param="missing"` and `file="character"` or `file="connection"` scan the entire file. With `file="connection"`, an argument `n` indicates the number of lines of the VCF file to input; a connection open at the beginning of the call is open and incremented by `n` lines at the end of the call, providing a convenient way to stream through large VCF files.

The INFO field of the scanned VCF file is returned as a single 'packed' vector, as in the VCF file. The GENO field is a list of matrices, each matrix corresponds to a field as defined in the FORMAT field of the VCF header. Each matrix has as many rows as scanned in the VCF file, and as many columns as there are samples. As with the INFO field, the elements of the matrix are 'packed'. The reason that INFO and GENO are returned packed is to facilitate manipulation, e.g., selecting particular rows or samples in a consistent manner across elements.

Value

`scanVcfHeader` returns a `VCFHeader` object with header information parsed into five categories, `samples`, `meta`, `fixed`, `info` and `geno`. Each can be accessed with a 'getter' of the same name (e.g., info(<VCFHeader>)). If the file header has multiple rows with the same name (e.g., 'source') the row names of the DataFrame are made unique in the usual way, 'source', 'source.1' etc.

`scanVcf` returns a list, with one element per range. Each list has 7 elements, obtained from the columns of the VCF specification:

rowRanges `GRanges` instance derived from `CHROM`, `POS`, `ID`, and the width of `REF`

REF reference allele

ALT alternate allele

QUAL phred-scaled quality score for the assertion made in ALT

FILTER indicator of whether or not the position passed all filters applied

INFO additional information

GENO genotype information immediately following the FORMAT field in the VCF

The `GENO` element is itself a list, with elements corresponding to those defined in the VCF file header. For `scanVcf`, elements of GENO are returned as a matrix of records x samples; if the description of the element in the file header indicated multiplicity other than 1 (e.g., variable number for "A", "G", or "."), then each entry in the matrix is a character string with sub-entries comma-delimited.

Author(s)

Martin Morgan and Valerie Obenchain>

References

`http://vcftools.sourceforge.net/specs.html` outlines the VCF specification.

`http://samtools.sourceforge.net/mpileup.shtml` contains information on the portion of the specification implemented by `bcftools`.

`http://samtools.sourceforge.net/` provides information on `samtools`.

See Also

readVcf BcfFile TabixFile

Examples

```
fl <- system.file("extdata", "ex2.vcf", package="VariantAnnotation")
scanVcfHeader(fl)
vcf <- scanVcf(fl)
## value: list-of-lists
str(vcf)
names(vcf[[1]][["GENO"]])
vcf[[1]][["GENO"]][["GT"]]
```

ScanVcfParam-class *Parameters for scanning VCF files*

Description

Use ScanVcfParam() to create a parameter object influencing which records and fields are imported from a VCF file. Record parsing is based on genomic coordinates and requires a Tabix index file. Individual VCF elements can be specified in the 'fixed', 'info', 'geno' and 'samples' arguments.

Usage

```
ScanVcfParam(fixed=character(), info=character(), geno=character(),
             samples=character(), trimEmpty=TRUE, which, ...)

## Getters and Setters

vcfFixed(object)
vcfFixed(object) <- value
vcfInfo(object)
vcfInfo(object) <- value
vcfGeno(object)
vcfGeno(object) <- value
vcfSamples(object)
vcfSamples(object) <- value
vcfTrimEmpty(object)
vcfTrimEmpty(object) <- value
vcfWhich(object)
vcfWhich(object) <- value
```

Arguments

fixed	A character() vector of fixed fields to be returned. Possible values are ALT, QUAL and FILTER. The CHROM, POS, ID and REF fields are needed to create the GRanges of variant locations. Because these are essential fields there is

no option to request or omit them. If not specified, all fields are returned; if `fixed=NA` only REF is returned.

`info` A character() vector naming the 'INFO' fields to return. `scanVcfHeader()` returns a vector of available fields. If not specified, all fields are returned; if `info=NA` no fields are returned.

`geno` A character() vector naming the 'GENO' fields to return. `scanVcfHeader()` returns a vector of available fields. If not specified, all fields are returned; if `geno=NA` no fields are returned and requests for specific samples are ignored.

`samples` A character() vector of sample names to return. `samples(scanVcfHeader())` returns all possible names. If not specified, data for all samples are returned; if either `samples=NA` or `geno=NA` no fields are returned. Requests for specific samples when `geno=NA` are ignored.

`trimEmpty` A logical(1) indicating whether 'GENO' fields with no values should be returned.

`which` A `GRanges` describing the sequences and ranges to be queried. Variants whose `POS` lies in the interval(s) `[start, end]` are returned. If `which` is not specified all ranges are returned.

`object` An instance of class `ScanVcfParam`.

`value` An instance of the corresponding slot, to be assigned to `object`.

`...` Arguments passed to methods.

Objects from the Class

Objects can be created by calls of the form `ScanVcfParam()`.

Slots

`which`: Object of class `"IntegerRangesList"` indicating which reference sequence and coordinate variants must overlap.

`fixed`: Object of class `"character"` indicating fixed fields to be returned.

`info`: Object of class `"character"` indicating portions of 'INFO' to be returned.

`geno`: Object of class `"character"` indicating portions of 'GENO' to be returned.

`samples`: Object of class `"character"` indicating the samples to be returned.

`trimEmpty`: Object of class `"logical"` indicating whether empty 'GENO' fields are to be returned.

Functions and methods

See 'Usage' for details on invocation.

Constructor:

ScanVcfParam: Returns a `ScanVcfParam` object. The `which` argument to the constructor can be one of several types, as documented above.

Accessors:

vcfFixed, vcfInfo, vcfGeno, vcfSamples, vcfTrimEmpty, vcfWhich: Return the corresponding field from `object`.

Methods:

show Compactly display the object.

Author(s)

Martin Morgan and Valerie Obenchain

See Also

readVcf

Examples

```
ScanVcfParam()

fl <- system.file("extdata", "structural.vcf", package="VariantAnnotation")
compressVcf <- bgzip(fl, tempfile())
idx <- indexTabix(compressVcf, "vcf")
tab <- TabixFile(compressVcf, idx)
## ---------------------------------------------------------------------
## 'which' argument
## ---------------------------------------------------------------------
## To subset on genomic coordinates, supply an object
## containing the ranges of interest. These ranges can
## be given directly to the 'param' argument or wrapped
## inside ScanVcfParam() as the 'which' argument.

## When using a list, the outer list names must correspond to valid
## chromosome names in the vcf file. In this example they are "1"
## and "2".
gr1 <- GRanges("1", IRanges(13219, 2827695, name="regionA"))
gr2 <- GRanges(rep("2", 2), IRanges(c(321680, 14477080),
               c(321689, 14477090), name=c("regionB", "regionC")))
grl <- GRangesList("1"=gr1, "2"=gr2)
vcf <- readVcf(tab, "hg19", grl)

## Names of the ranges are in the 'paramRangeID' metadata column of the
## GRanges object returned by the rowRanges() accessor.
rowRanges(vcf)

## which can be used for subsetting the VCF object
vcf[rowRanges(vcf)$paramRangeID == "regionA"]

## When using ranges, the seqnames must correspond to valid
## chromosome names in the vcf file.
gr <- unlist(grl, use.names=FALSE)
vcf <- readVcf(tab, "hg19", gr)

## ---------------------------------------------------------------------
```

```
## 'fixed', 'info', 'geno' and 'samples' arguments
## ------------------------------------------------------------------------
## This param specifies the "GT" 'geno' field for a single sample
## and the subset of ranges in 'which'. All 'fixed' and 'info' fields
## will be returned.
ScanVcfParam(geno="GT", samples="NA00002", which=gr)

## Here two 'fixed' and one 'geno' field are specified
ScanVcfParam(fixed=c("ALT", "QUAL"), geno="GT", info=NA)

## Return only the 'fixed' fields
ScanVcfParam(geno=NA, info=NA)
```

`seqinfo` *Get seqinfo for VCF file*

Description

Get seqinfo for VCF file

Usage

```
## S4 method for signature 'VcfFile'
seqinfo(x)
## S4 method for signature 'VcfFileList'
seqinfo(x)
```

Arguments

x	Either character(), `VcfFile`, or `VcfFileList`

Details

If a `VcfFile`The file header is scanned an appropriate seqinfo object in given. If a `VcfFileList` is given, all file headers are scanned, and appropriate combined seqinfo object is given.

Value

Seqinfo object

Author(s)

Lori Shepherd

See Also

`VcfFile`, `Seqinfo`

Examples

```
fl <- system.file("extdata", "chr7-sub.vcf.gz", package="VariantAnnotation",
                  mustWork=TRUE)
vcf <- VcfFile(fl)
seqinfo(vcf)
```

SIFTDb-class *SIFTDb objects*

Description

The SIFTDb class is a container for storing a connection to a SIFT sqlite database.

Details

SIFT is a sequence homology-based tool that sorts intolerant from tolerant amino acid substitutions and predicts whether an amino acid substitution in a protein will have a phenotypic effect. SIFT is based on the premise that protein evolution is correlated with protein function. Positions important for function should be conserved in an alignment of the protein family, whereas unimportant positions should appear diverse in an alignment.

SIFT uses multiple alignment information to predict tolerated and deleterious substitutions for every position of the query sequence. The procedure can be outlined in the following steps,

- search for similar sequences
- choose closely related sequences that may share similar function to the query sequence
- obtain the alignment of the chosen sequences
- calculate normalized probabilities for all possible substitutions from the alignment.

Positions with normalized probabilities less than 0.05 are predicted to be deleterious, those greater than or equal to 0.05 are predicted to be tolerated.

Methods

In the code below, `x` is a `SIFTDb` object.

`metadata(x)`: Returns `x`'s metadata in a data frame.

`columns(x)`: Returns the names of the `columns` that can be used to subset the data columns.

`keys(x)`: Returns the names of the `keys` that can be used to subset the data rows. The `keys` values are the rsid's.

`select(x, keys = NULL, columns = NULL, ...)`: Returns a subset of data defined by the character vectors `keys` and `columns`. If no `keys` are supplied, all rows are returned. If no `columns` are supplied, all columns are returned. For column descriptions see `?SIFTDbColumns`.

Author(s)

Valerie Obenchain

References

SIFT Home: http://sift.jcvi.org/

Kumar P, Henikoff S, Ng PC. Predicting the effects of coding non-synonymous variants on protein function using the SIFT algorithm. Nat Protoc. 2009;4(7):1073-81

Ng PC, Henikoff S. Predicting the Effects of Amino Acid Substitutions on Protein Function Annu Rev Genomics Hum Genet. 2006;7:61-80.

Ng PC, Henikoff S. SIFT: predicting amino acid changes that affect protein function. Nucleic Acids Res. 2003 Jul 1;31(13):3812-4.

Examples

```
if (interactive()) {
    library(SIFT.Hsapiens.dbSNP132)

    ## metadata
    metadata(SIFT.Hsapiens.dbSNP132)

    ## available rsid's
    head(keys(SIFT.Hsapiens.dbSNP132))

    ## for column descriptions see ?SIFTDbColumns
    columns(SIFT.Hsapiens.dbSNP132)

    ## subset on keys and columns
    rsids <- c("rs2142947", "rs17970171", "rs8692231", "rs3026284")
    subst <- c("RSID", "PREDICTION", "SCORE")
    select(SIFT.Hsapiens.dbSNP132, keys=rsids, columns=subst)
    select(SIFT.Hsapiens.dbSNP132, keys=rsids[1:2])
}
```

`SIFTDbColumns` *SIFTDb Columns*

Description

Description of the SIFT Sqlite Database Columns

Column descriptions

These column names are displayed when `columns` is called on a `SIFTDb` object.

- RSID : rsid
- PROTEINID : NCBI RefSeq protein ID
- AACHANGE : amino acid substitution; reference aa is preceeding, followed by the position and finally the snp aa
- METHOD : method of obtaining related sequences using PSI-BLAST

- AA : either the reference or snp residue amino acid
- PREDICTION : SIFT prediction
- SCORE : SIFT score (range 0 to 1)
  - TOLERATED : score is greater than 0.05
  - DAMAGING : score is less than or equal to 0.05
  - NOT SCORED : no prediction is made if there are less than 2 homologous sequences that have an amino acid at the position of the given SNP or if the SIFT prediction is not available
- MEDIAN : diversity measurement of the sequences used for prediction (range 0 to 4.32)
- POSITIONSEQS : number of sequences with an amino acide at the position of prediction
- TOTALSEQS : total number of sequences in alignment

Author(s)

Valerie Obenchain

See Also

`?SIFTDb`

`snpSummary` *Counts and distribution statistics for SNPs in a VCF object*

Description

Counts and distribution statistics for SNPs in a VCF object

Usage

```
  ## S4 method for signature 'CollapsedVCF'
snpSummary(x, ...)
```

Arguments

`x`	A CollapsedVCF object.
`...`	Additional arguments to methods.

Details

Genotype counts, allele counts and Hardy Weinberg equilibrium (HWE) statistics are calculated for single nucleotide variants in a CollapsedVCF object. HWE has been established as a useful quality filter on genotype data. This equilibrium should be attained in a single generation of random mating. Departures from HWE are indicated by small p values and are almost invariably indicative of a problem with genotype calls.

The following caveats apply:

- No distinction is made between phased and unphased genotypes.
- Only diploid calls are included.
- Only 'valid' SNPs are included. A 'valid' SNP is defined as having a reference allele of length 1 and a single alternate allele of length 1.

Variants that do not meet these criteria are set to NA.

Value

The object returned is a `data.frame` with seven columns.

g00 Counts for genotype 00 (homozygous reference).

g01 Counts for genotype 01 or 10 (heterozygous).

g11 Counts for genotype 11 (homozygous alternate).

a0Freq Frequency of the reference allele.

a1Freq Frequency of the alternate allele.

HWEzscore Z-score for departure from a null hypothesis of Hardy Weinberg equilibrium.

HWEpvalue p-value for departure from a null hypothesis of Hardy Weinberg equilibrium.

Author(s)

Chris Wallace <cew54@cam.ac.uk>

See Also

genotypeToSnpMatrix, probabilityToSnpMatrix

Examples

```
fl <- system.file("extdata", "ex2.vcf", package="VariantAnnotation")
vcf <- readVcf(fl, "hg19")

## The return value is a data.frame with genotype counts
## and allele frequencies.
df <- snpSummary(vcf)
df

## Compare to ranges in the VCF object:
rowRanges(vcf)

## No statistics were computed for the variants in rows 3, 4
## and 5. They were omitted because row 3 has two alternate
## alleles, row 4 has none and row 5 is not a SNP.
```

summarizeVariants *Summarize variants by sample*

Description

Variants in a VCF file are overlapped with an annotation region and summarized by sample. Genotype information in the VCF is used to determine which samples express each variant.

Usage

```
## S4 method for signature 'TxDb,VCF,CodingVariants'
summarizeVariants(query, subject, mode, ...)
## S4 method for signature 'TxDb,VCF,FiveUTRVariants'
summarizeVariants(query, subject, mode, ...)
## S4 method for signature 'TxDb,VCF,ThreeUTRVariants'
summarizeVariants(query, subject, mode, ...)
## S4 method for signature 'TxDb,VCF,SpliceSiteVariants'
summarizeVariants(query, subject, mode, ...)
## S4 method for signature 'TxDb,VCF,IntronVariants'
summarizeVariants(query, subject, mode, ...)
## S4 method for signature 'TxDb,VCF,PromoterVariants'
summarizeVariants(query, subject, mode, ...)
## S4 method for signature 'GRangesList,VCF,VariantType'
summarizeVariants(query, subject, mode, ...)
## S4 method for signature 'GRangesList,VCF,function'
summarizeVariants(query, subject, mode, ...)
```

Arguments

query	A TxDb or GRangesList object that serves as the annotation. GFF files can be converted to TxDb objects with makeTxDbFromGFF() in the txdbmaker package.
subject	A VCF object containing the variants.
mode	mode can be a VariantType class or the name of a function. When mode is a VariantType class, counting is done with locateVariants and counts are summarized transcript-by-sample. Supported VariantType classes include CodingVariants, IntronVariants, FiveUTRVariants, ThreeUTRVariants, SpliceSiteVariants or PromoterVariants. AllVariants() and IntergenicVariants are not supported. See ?locateVariants for more detail on the variant classes. mode can also be the name of any counting function that outputs a Hits object. Variants will be summarized by the length of the GRangesList annotation (i.e., 'length-of-GRangesList'-by-sample).
...	Additional arguments passed to methods such as **ignore.strand** A logical indicating if strand should be igored when performing overlaps.

Details

summarizeVariants uses the genotype information in a VCF file to determine which samples are positive for each variant. Variants are overlapped with the annotation and the counts are summarized annotation-by-sample. If the annotation is a GRangesList of transcripts, the count matrix will be transcripts-by-sample. If the GRangesList is genes, the count matrix will be gene-by-sample.

- Counting with locateVariants() :

  Variant counts are always summarized transcript-by-sample. When query is a GRangesList, it must be compatible with the VariantType-class given as the mode argument. The list below specifies the appropriate GRangesList for each mode.

  CodingVariants : coding (CDS) by transcript

  IntronVariants : introns by transcript

  FiveUTRVariants : five prime UTR by transcript

  ThreeUTRVariants : three prime UTR by transcript

  SpliceSiteVariants : introns by transcript

  PromoterVariants : list of transcripts

  When query is a TxDb, the appropriate region-by-transcript GRangesList listed above is extracted internally and used as the annotation.

- Counting with a user-supplied function :

  subject must be a GRangesList and mode must be the name of a function. The count function must take 'query' and 'subject' arguments and return a Hits object. Counts are summarized by the outer list elements of the GRangesList.

Value

A RangedSummarizedExperiment object with count summaries in the assays slot. The rowRanges contains the annotation used for counting. Information in colData and metadata are taken from the VCF file.

Author(s)

Valerie Obenchain

See Also

readVcf, predictCoding, locateVariants

Examples

```
library(TxDb.Hsapiens.UCSC.hg19.knownGene)
txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene

## Read variants from VCF.
fl <- system.file("extdata", "chr22.vcf.gz", package="VariantAnnotation")
vcf <- readVcf(fl, "hg19")
## Rename seqlevels to match TxDb; confirm the match.
seqlevels(vcf) <- paste0("chr", seqlevels(vcf))
intersect(seqlevels(vcf), seqlevels(txdb))
```

```
## ----------------------------------------
## Counting with locateVariants()
## ----------------------------------------
## TxDb as the 'query'
coding1 <- summarizeVariants(txdb, vcf, CodingVariants())
colSums(assays(coding1)$counts)

## GRangesList as the 'query'
cdsbytx <- cdsBy(txdb, "tx")
coding2 <- summarizeVariants(cdsbytx, vcf, CodingVariants())

stopifnot(identical(assays(coding1)$counts, assays(coding2)$counts))

## Promoter region variants summarized by transcript
tx <- transcripts(txdb)
txlst <- splitAsList(tx, seq_len(length(tx)))
promoter <- summarizeVariants(txlst, vcf,
                              PromoterVariants(upstream=100, downstream=10))
colSums(assays(promoter)$counts)

## ----------------------------------------
## Counting with findOverlaps()
## ----------------------------------------

## Summarize all variants by transcript
allvariants <- summarizeVariants(txlst, vcf, findOverlaps)
colSums(assays(allvariants)$counts)
```

VariantAnnotation-defunct

Defunct Functions in Package VariantAnnotation

Description

The functions or variables listed here are no longer part of VariantAnnotation.

Details

Removed

- refLocsToLocalLocs has been replaced by mapToTranscripts and pmapToTranscripts.
- readVcfLongForm has been replaced by expand.
- dbSNPFilter and regionFilter have been replaced by filterVcf.
- regionFilter has been replaced by filterVcf.
- MatrixToSnpMatrix has been replaced by genotypeToSnpMatrix.
- getTranscriptSeqs has been replaced by extractTranscriptSeqs in the GenomicFeatures package.
- VRangesScanVcfParam has been replaced by ScanVcfParam.
- restrictToSVN has been replaced by isSNV.

usage

Removed

- refLocsToLocalLocs()
- readVcfLongForm()
- dbSNPFilter()
- regionFilter()
- MatrixToSnpMatrix()
- getTranscriptSeqs()
- VRangesScanVcfParam()
- restrictToSNV()

Author(s)

Valerie Obenchain

See Also

- expand
- filterVcf
- genotypeToSnpMatrix
- mapToTranscripts
- extractTranscriptSeqs
- isSNV
- ScanVcfParam

VariantType-class *VariantType subclasses*

Description

VariantType subclasses specify the type of variant to be located with locateVariants.

Usage

```
CodingVariants()
IntronVariants()
FiveUTRVariants()
ThreeUTRVariants()
SpliceSiteVariants()
IntergenicVariants(upstream = 1e+06L, downstream = 1e+06L,
                   idType=c("gene", "tx"))
PromoterVariants(upstream = 2000L, downstream = 200L)
AllVariants(promoter = PromoterVariants(),
            intergenic = IntergenicVariants())
```

Arguments

upstream, downstream	Single integer values representing the number of base pairs upstream of the 5'-end and downstream of the 3'-end. Used in contructing PromoterVariants() and IntergenicVariants() objects only.
idType	character indicating if the ids in the PRECEDEID and FOLLOWID metadata columns should be gene ids ("gene") or transcript ids ("tx"). Applicable to IntergenicVariants() objects only.
promoter	PromoterVariants object with appropriate upstream and downstream values. Used when constructing AllVariants objects only.
intergenic	IntergenicVariants object with appropriate upstream and downstream values. Used when constructing AllVariants objects only.

Details

VariantType is a virtual class inherited by the CodingVariants, IntronVariants, FiveUTRVariants, ThreeUTRVariants, SpliceSiteVariants, IntergenicVariants and AllVariants subclasses.

The subclasses are used as the region argument to locateVariants. They designate the type of variant (i.e., region of the annotation to match) when calling locateVariants.

The majority of subclasses have no slots and require no arguments for an instance to be created. PromoterVariants and IntergenicVariants and accept upstream and downstream arguments that define the number of base pairs upstream from the 5'-end and downstream from the 3'-end of the transcript region. See the ?locateVariants man page for details. IntergenicVariants also accepts a idType that controls what IDs are returned in the PRECEDEID and FOLLOWID metadata columns.

AllVariants accepts promoter and intergenic arguments which are PromoterVariants() and IntergenicVariants() objects with the appropriate upstream and downstream values.

Accessors

In the following code, x is a PromoterVariants or a AllVariants object.

- upstream(x), upstream(x) <- value: Gets or sets the number of base pairs defining a range upstream of the 5' end (excludes 5' start value).
- downstream(x), downstream(x) <- value: Gets or sets the number of base pairs defining a range downstream of the 3' end (excludes 3' end value).
- idType(x), idType(x) <- value: Gets or sets the character() which controls the id returned in the PRECEDEID and FOLLOWID output columns. Possible values are "gene" and "tx".
- promoters(x), promoters(x) <- value: Gets or sets the PromoterVariants in the AllVariants object.
- intergenic(x), intergenic(x) <- value: Gets or sets the IntergenicVariants in the AllVariants object.

Author(s)

Valerie Obenchain

See Also

- The promoters function on the intra-range-methods man page in the GenomicRanges package.

Examples

```
CodingVariants()
SpliceSiteVariants()
PromoterVariants(upstream=1000, downstream=10000)

## Default values for PromoterVariants and IntergenicVariants
AllVariants()
## Modify 'upstream' and 'downstream' for IntergenicVariants
AllVariants(intergenic=IntergenicVariants(500, 100))
## Reset PromoterVariants on existing AllVariants object
av <- AllVariants()
av
promoter(av) <- PromoterVariants(100, 50)
av
```

---

variant_body *helper function to construct inputs for VEP REST API*

Description

helper function to construct inputs for VEP REST API

Usage

```
variant_body(chr, pos, id, ref, alt)
```

Arguments

chr	character(1)
pos	numeric(1)
id	character(1)
ref	character(1)
alt	character(1)

Note

Produces a string used as an example in VEP API documentation.

VCF-class *VCF class objects*

Description

The VCF class is a virtual class extended from RangedSummarizedExperiment. The subclasses, CompressedVCF and ExtendedVCF, are containers for holding data from Variant Call Format files.

Details

The VCF class is a virtual class with two concrete subclasses, CollapsedVCF and ExtendedVCF.

Slots unique to VCF and subclasses,

- fixed: A DataFrame containing the REF, ALT, QUAL and FILTER fields from a VCF file.
- info: A DataFrame containing the INFO fields from a VCF file.

Slots inherited from RangedSummarizedExperiment,

- metadata: A list containing the file header or other information about the overall experiment.
- rowRanges: A GRanges-class instance defining the variant ranges and associated metadata columns of REF, ALT, QUAL and FILTER. While the REF, ALT, QUAL and FILTER fields can be displayed as metadata columns they cannot be modified with rowRanges<-. To modify these fields use fixed<-.
- colData: A DataFrame-class instance describing the samples and associated metadata.
- geno: The assays slot from RangedSummarizedExperiment has been renamed as geno for the VCF class. This slot contains the genotype information immediately following the FORMAT field in a VCF file. Each element of the list or SimpleList is a matrix or array.

It is expected that users will not create instances of the VCF class but instead one of the concrete subclasses, CollapsedVCF or ExpandVCF. CollapsedVCF contains the ALT data as a DNAStringSetList allowing for multiple alleles per variant. The ExpandedVCF ALT data is a DNAStringSet where the ALT column has been expanded to create a flat form of the data with one row per variant-allele combination. In the case of strucutral variants, ALT will be a CompressedCharacterList or character in the collapsed or expanded forms.

Constructors

readVcf(file, genome, param, ..., row.names=TRUE)

VCF(rowRanges = GRanges(), colData = DataFrame(), exptData = list(header = VCFHeader()), fixed = DataFrame(), info = DataFrame(), geno = SimpleList(), ..., collapsed=TRUE, verbose = FALSE) Creates CollapsedVCF when collapsed = TRUE and an ExpandedVCF when collapsed = FALSE.

This is a low-level constructor used internally. Most instances of the VCF class are created with readVCF.

Accessors

In the following code snippets x is a CollapsedVCF or ExpandedVCF object.

`rowRanges(x, ..., fixed = TRUE)`, `rowRanges(x) <- value`: Gets or sets the rowRanges. The CHROM, POS, ID, POS and REF fields are used to create a `GRanges` object. The start of the ranges are defined by POS and the width is equal to the width of the reference allele REF. The IDs become the rownames. If they are missing (i.e., '.') a string of CHROM:POS_REF/ALT is used instead. The `genome` argument is stored in the seqinfo of the `GRanges` and can be accessed with `genome(<VCF>)`.

When `fixed = TRUE`, REF, ALT, QUAL and FILTER metadata columns are displayed as metadata columns. To modify the `fixed` fields, use the `fixed<-` setter.

One metadata column, `paramRangeID`, is included with the `rowRanges`. This ID is meaningful when multiple ranges are specified in the `ScanVcfParam` and distinguishes which records match each range.

The metadata columns of a VCF object are accessed with the following:

- `ref(x)`, `ref(x) <- value`: Gets or sets the reference allele (REF). `value` must be a `DNAStringSet`.
- `alt(x)`, `alt(x) <- value`: Gets or sets the alternate allele data (ALT). When `x` is a CollapsedVCF, `value` must be a `DNAStringSetList` or `CompressedCharacterList`. For ExpandedVCF, `value` must be a `DNAStringSet` or `character`.
- `qual(x)`, `qual(x) <- value`: Returns or sets the quality scores (QUAL). `value` must be an `numeric(1L)`.
- `filt(x)`, `filt(x) <- value`: Returns or sets the filter data. `value` must be a `character(1L)`. Names must be one of 'REF', 'ALT', 'QUAL' or 'FILTER'.

`mcols(x)`, `mcols(x) <- value`: These methods behave the same as `mcols(rowRanges(x))` and `mcols(rowRanges(x)) <- value`. This method does not manage the fixed fields, 'REF', 'ALT', 'QUAL' or 'FILTER'. To modify those columns use `fixed<-`.

`fixed(x)`, `fixed(x) <- value`: Gets or sets a DataFrame of REF, ALT, QUAL and FILTER only. Note these fields are displayed as metadata columns with the rowRanges() data (set to fixed = FALSE to suppress).

`info(x, ..., row.names = TRUE)`, `info(x) <- value`: Gets or sets a DataFrame of INFO variables. Row names are added if unique and `row.names=TRUE`.

`geno(x, withDimnames=TRUE)`, `geno(x) <- value`: oets a SimpleList of genotype data. `value` is a SimpleList. To replace a single variable in the SimpleList use `geno(x)$variable <- value`; in this case `value` must be a matrix or array. By default row names are returned; to override specify `geno(vcf, withDimnames=FALSE)`.

`metadata(x)`: Gets a `list` of experiment-related data. By default this list includes the 'header' information from the VCF file. See the use of `header()` for details in extracting header information.

`colData(x)`, `colData(x) <- value`: Gets or sets a `DataFrame` of sample-specific information. Each row represents a sample in the VCF file. `value` must be a `DataFrame` with rownames representing the samples in the VCF file.

`genome(x)`: Extract the `genome` information from the `GRanges` object returned by the `rowRanges` accessor.

`seqlevels(x)`: Extract the `seqlevels` from the `GRanges` object returned by the `rowRanges` accessor.

`strand(x)`: Extract the `strand` from the `GRanges` object returned by the `rowRanges` accessor.

`header(x)`, `header(x)<- value`: Get or set the VCF header information. Replacement value must be a `VCFHeader` object. To modify individual elements use `info<-`, `geno<-` or `meta<-` on a 'VCFHeader' object. See ?`VCFHeader` man page for details.

- `info(header(x))`
- `geno(header(x))`
- `meta(header(x))`
- `samples(header(x))`

`vcfFields(x)` Returns a `CharacterList` of all available VCF fields, with names of `fixed`, `info`, `geno` and `samples` indicating the four categories. Each element is a character() vector of available VCF field names within each category.

Subsetting and combining

In the following code `x` is a VCF object, and ... is a list of VCF objects.

`x[i, j]`, `x[i, j] <- value`: Gets or sets rows and columns. `i` and `j` can be integer or logical vectors. `value` is a replacement `VCF` object.

`subset(x, subset, select, ...)`: Restricts `x` by evaluating the `subset` argument in the scope of `rowData(x)` and `info(x)`, and `select` in the context of `colData(x)`. The `subset` argument restricts by rows, while the `select` argument restricts by column. The `...` are passed to the underlying `subset()` calls.

`cbind(...)`, `rbind(...)`: `cbind` combines objects with identical ranges (`rowRanges`) but different samples (columns in `assays`). The colnames in `colData` must match or an error is thrown. Columns with duplicate names in `fixed`, `info` and `mcols(rowRanges(VCF))` must contain the same data.

`rbind` combines objects with different ranges (`rowRanges`) and the same subjects (columns in `assays`). Columns with duplicate names in `colData` must contain the same data. The 'Samples' columns in `colData` (created by `readVcf`) are renamed with a numeric extension ordered as they were input to `rbind` e.g., "Samples.1, Samples.2, ..." etc.

`metadata` from all objects are combined into a `list` with no name checking.

expand

In the following code snippets `x` is a CollapsedVCF object.

`expand(x, ..., row.names = FALSE)`: Expand (unlist) the ALT column of a CollapsedVCF object to one row per ALT value. Variables with Number='A' have one value per alternate allele and are expanded accordingly. The 'AD' genotype field (and any variables with 'Number' set to 'R') is expanded into REF/ALT pairs. For all other fields, the rows are replicated to match the elementNROWS of ALT.

The output is an ExpandedVCF with ALT as a `DNAStringSet` or `character` (structural variants). By default rownames are NULL. When `row.names=TRUE` the expanded output has duplicated rownames corresponding to the original `x`.

genotypeCodesToNucleotides(vcf, ...)

This function converts the 'GT' genotype codes in a VCF object to nucleotides. See also ?readGT to read in only 'GT' data as codes or nucleotides.

SnpMatrixToVCF(from, seqSource)

This function converts the output from the read.plink function to a VCF class. from must be a list of length 3 with named elements "map", "fam" and "genotypes". seqSource can be a BSgenome or an FaFile used for reference sequence extraction.

Variant Type

Functions to identify variant type include isSNV, isInsertion, isDeletion, isIndel, isSubstitution and isTransition. See the ?isSNV man page for details.

Arguments

geno A list or SimpleList of matrix elements, or a matrix containing the genotype information from a VCF file. If present, these data immediately follow the FORMAT field in the VCF.

Each element of the list must have the same dimensions, and dimension names (if present) must be consistent across elements and with the row names of rowRanges, colData.

info A DataFrame of data from the INFO field of a VCF file. The number of rows must match that in the rowRanges object.

fixed A DataFrame of REF, ALT, QUAL and FILTER fields from a VCF file. The number of rows must match that of the rowRanges object.

rowRanges A GRanges instance describing the ranges of interest. Row names, if present, become the row names of the VCF. The length of the GRanges must equal the number of rows of the matrices in geno.

colData A DataFrame describing the samples. Row names, if present, become the column names of the VCF.

metadata A list describing the header of the VCF file or additional information for the overall experiment.

... For cbind and rbind a list of VCF objects. For all other methods ... are additional arguments passed to methods.

collapsed A logical(1) indicating whether a CollapsedVCF or ExpandedVCF should be created. The ALT in a CollapsedVCF is a DNAStringSetList while in a ExpandedVCF it is a DNAStringSet.

verbose A logical(1) indicating whether messages about data coercion during construction should be printed.

Author(s)

Valerie Obenchain

See Also

GRanges, DataFrame, SimpleList, RangedSummarizedExperiment, readVcf, writeVcf isSNV

Examples

```
## readVcf() parses data into a VCF object:

fl <- system.file("extdata", "structural.vcf", package="VariantAnnotation")
vcf <- readVcf(fl, genome="hg19")

## ---------------------------------------------------------------
## Accessors
## ---------------------------------------------------------------
## Variant locations are stored in the GRanges object returned by
## the rowRanges() accessor.
rowRanges(vcf)

## Suppress fixed fields:
rowRanges(vcf, fixed=FALSE)

## Individual fields can be extracted with ref(), alt(), qual(), filt() etc.
qual(vcf)
ref(vcf)
head(info(vcf))

## All available VCF field names can be contracted with vcfFields().
vcfFields(vcf)

## Extract genotype fields with geno(). Access specific fields with
## '$' or '[['.
geno(vcf)
identical(geno(vcf)$GQ, geno(vcf)[[2]])

## ---------------------------------------------------------------
## Renaming seqlevels and subsetting
## ---------------------------------------------------------------
## Overlap and matching operations require that the objects
## being compared have the same seqlevels (chromosome names).
## It is often the case that the seqlevesls in on of the objects
## needs to be modified to match the other. In this VCF, the
## seqlevels are numbers instead of preceded by "chr" or "ch".

seqlevels(vcf)

## Rename the seqlevels to start with 'chr'.
vcf2 <- vcf
seqlevels(vcf2) <- paste0("chr", seqlevels(vcf2))
seqlevels(vcf2)

## Dropping some of the seqlevels can have the effect of reducing
## the length of the VCF object by keeping only its elements that
## belong to the remaining seqlevels.
vcf3 <- vcf2
length(vcf3)
seqlevels(vcf3, pruning.mode="coarse") <- "chr2"
seqlevels(vcf3)
```

```
length(vcf3)

## ------------------------------------------------------------------
## Header information
## ------------------------------------------------------------------

## Header data can be modified in the 'meta', 'info' and 'geno'
## slots of the VCFHeader object. See ?VCFHeader for details.

## Current 'info' fields.
rownames(info(header(vcf)))

## Add a new field to the header.
newInfo <- DataFrame(Number=1, Type="Integer",
                     Description="Number of Samples With Data",
                     row.names="NS")
info(header(vcf)) <- rbind(info(header(vcf)), newInfo)
rownames(info(header(vcf)))

## ------------------------------------------------------------------
## Collapsed and Expanded VCF
## ------------------------------------------------------------------
## readVCF() produces a CollapsedVCF object.
fl <- system.file("extdata", "ex2.vcf",
                  package="VariantAnnotation")
vcf <- readVcf(fl, genome="hg19")
vcf

## The ALT column is a DNAStringSetList to allow for more
## than one alternate allele per variant.
alt(vcf)

## For structural variants ALT is a CharacterList.
fl <- system.file("extdata", "structural.vcf",
                  package="VariantAnnotation")
vcf <- readVcf(fl, genome="hg19")
alt(vcf)

## ExpandedVCF is the 'flattened' counterpart of CollapsedVCF.
## The ALT and all variables with Number='A' in the header are
## expanded to one row per alternate allele.
vcfLong <- expand(vcf)
alt(vcfLong)

## Also see the ?VRanges class for an alternative form of
## 'flattened' VCF data.

## ------------------------------------------------------------------
## isSNV()
## ------------------------------------------------------------------
## isSNV() returns a subset VCF containing SNVs only.

vcf <- VCF(rowRanges = GRanges("chr1", IRanges(1:4*3, width=c(1, 2, 1, 1))))
```

```
alt(vcf) <- DNAStringSetList("A", c("TT"), c("G", "A"), c("TT", "C"))
ref(vcf) <- DNAStringSet(c("G", c("AA"), "T", "G"))
fixed(vcf)[c("REF", "ALT")]

## SNVs are present in rows 1 (single ALT value), 3 (both ALT values)
## and 4 (1 of the 2 ALT values).
vcf[isSNV(vcf, singleAltOnly=TRUE)]
vcf[isSNV(vcf, singleAltOnly=FALSE)] ## all 3 SNVs
```

`VcfFile` *Manipulate Variant Call Format (Vcf) files.*

Description

Use `VcfFile()` to create a reference to a Vcf file (and its index). Once opened, the reference remains open across calls to methods, avoiding costly index re-loading.

`VcfFileList()` provides a convenient way of managing a list of `VcfFile` instances.

usage

```
## Constructors

VcfFile(file, index = paste(file, "tbi", sep="."), ..., yieldSize=NA_integer_)

VcfFileList(..., yieldSize=NA_integer_)

## Accessors

index(object)

path(object, ...)

isOpen(con, rw="")

yieldSize(object, ...)

yieldSize(object, ...) <- value

show(object)

## Opening / closing

open(con, ...)

close(con, ...)

## method

vcfFields(object)
```

arguments

- conAn instance of `VcfFile`.
- fileA character(1) vector to the Vcf file path; can be remote (http://, ftp://).
- indexA character(1) vector of the Vcf file index (.tbi file).
- yieldSizeNumber of records to yield each time the file is read from using `scanVcf` or `readVcf`.
- ...Additional arguments. For `VcfFileList`, this can either be a single character vector of paths to Vcf files, or several instances of `VcfFile` objects.
- rwcharacter() indicating mode of file.

Objects from the Class

Objects are created by calls of the form `VcfFile()`.

Fields

`VcfFile` and `VcfFileList` classes inherit fields from the `TabixFile` and `TabixFileList` classes.

Functions and methods

`VcfFile` and `VcfFileList` classes inherit methods from the `TabixFile` and `TabixFileList` classes.

Opening / closing:

open Opens the (local or remote) `path` and `index`. Returns a `VcfFile` instance. `yieldSize` determines the number of records parsed during each call to `scanVcf` or `readVcf`; `NA` indicates that all records are to be parsed.

close Closes the `VcfFile` `con`; returning (invisibly) the updated `VcfFile`. The instance may be re-opened with `open.VcfFile`.

Accessors:

path Returns a character(1) vector of the Vcf path name.

index Returns a character(1) vector of Vcf index (tabix file) name.

yieldSize, yieldSize<- Return or set an integer(1) vector indicating yield size.

Methods:

vcfFields Returns a `CharacterList` of all available VCF fields, with names of `fixed`, `info`, `geno` and `samples` indicating the four categories. Each element is a character() vector of available VCF field names within each category. It works for both local and remote vcf file.

Author(s)

Valerie Obenchain

Examples

```
fl <- system.file("extdata", "chr7-sub.vcf.gz", package="VariantAnnotation",
                  mustWork=TRUE)
vcffile <- VcfFile(fl)
vcffile
vcfFields(fl)
vcfFields(vcffile)

param <- GRanges("7", IRanges(c(55000000,  55900000), width=10000))
vcf <- readVcf(vcffile, "hg19", param=param)
dim(vcf)

## `vcfFields` also works for remote vcf filepath.
## Not run:
chr22url <- "ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/ALL.chr22.phase3_shapeit2_mvncall_integr
vcfFields(chr22url)

## End(Not run)
```

VCFHeader-class *VCFHeader instances*

Description

The VCFHeader class holds Variant Call Format (VCF) file header information and is produced from a call to scanVcfHeader.

Details

The VCFHeader class holds header information from a VCF file.

Slots :

reference character() vector

sample character() vector

header DataFrameList-class

Constructor

```
VCFHeader(reference = character(), samples = character(), header = DataFrameList(),
    ...)
```

Accessors

In the following code snippets x is a VCFHeader object.

samples(x): Returns a character() vector of names of samples.

header(x): Returns all information in the header slot which includes meta, info and geno if present.

`meta(x)`, `meta(x)<- value`: The getter returns a `DataFrameList`. Each `DataFrame` represents a unique "key" name in the header file. Multiple header lines with the same "key" are parsed into the same `DataFrame` with the "ID" field as the row names. Simple header lines have no "ID" field in which case the "key" is used as the row name.

NOTE: In VariantAnnotation <= 1.27.5, the `meta()` extractor returned a `DataFrame` called "META" which held all simple key-value header lines. The VCF 4.3 specs allowed headers lines with key name "META" which caused a name clash with the pre-existing "META" `DataFrame`.

In `VariantAnnotation` >=1.27.6 the "META" `DataFrame` was split and each row became its own separate `DataFrame`. Calling `meta()` on a `VCFHeader` object now returns a list of `DataFrames`, one for each unique key name in the header.

`fixed(x)`, `fixed(x)<- value`: Returns a `DataFrameList` of information pertaining to any of 'REF', 'ALT', 'FILTER' and 'QUAL'. Replacement value must be a `DataFrameList` with one or more of the following names, 'QUAL', 'FILTER', 'REF' and 'ALT'.

`info(x)`, `info(x)<- value`: Gets or sets a `DataFrame` of 'INFO' information. Replacement value must be a `DataFrame` with 3 columns named 'Number', 'Type' and 'Description'.

`geno(x)`, `geno(x)<- value`: Returns a `DataFrame` of 'FORMAT' information. Replacement value must be a `DataFrame` with 3 columns named 'Number', 'Type' and 'Description'.

`reference(x)`: Returns a character() vector with names of reference sequences. Not relevant for `scanVcfHeader`.

`vcfFields(x)`: Returns a `CharacterList` of all available VCF fields, with names of `fixed`, `info`, `geno` and `samples` indicating the four categories. Each element is a character() vector of available VCF field names within each category.

Arguments

reference A character() vector of sequences.

sample A character() vector of sample names.

header A `DataFrameList` of parsed header lines (preceeded by "##") present in the VCF file.

... Additional arguments passed to methods.

Author(s)

Valerie Obenchain

See Also

`scanVcfHeader`, `DataFrameList`

Examples

```
fl <- system.file("extdata", "structural.vcf", package="VariantAnnotation")
hdr <- scanVcfHeader(fl)

fixed(hdr)
info(hdr)
geno(hdr)
vcfFields(hdr)
```

vep_by_region *Use the VEP region API on variant information in a VCF object as defined in VariantAnnotation.*

Description

Use the VEP region API on variant information in a VCF object as defined in VariantAnnotation.

Usage

```
vep_by_region(vcfobj, snv_only = TRUE, chk_max = TRUE)
```

Arguments

vcfobj	instance of VCF class; note the difference between the CollapsedVCF and ExpandedVCF instances.
snv_only	logical(1) if TRUE filter the VCF to information about single nucleotide addresses
chk_max	logical(1) requests to ensembl VEP API are limited to 200 positions; if TRUE and the request involves more than 200 positions, an error is thrown by this function.

Value

instance of 'response' from httr package

Examples

```
fl <- system.file("extdata", "chr22.vcf.gz", package="VariantAnnotation")
r22 = readVcf(fl)
dr = which(width(rowRanges(r22))!=1)
r22s = r22[-dr]
res = vep_by_region(r22[1:100], snv_only=FALSE, chk_max=FALSE)
ans = jsonlite::fromJSON(jsonlite::toJSON(httr::content(res)))
```

VRanges-class *VRanges objects*

Description

The VRanges class is a container for variant calls, including SNVs and indels. It extends GRanges to provide special semantics on top of a simple vector of genomic locations. While it is not as expressive as the VCF object, it is a simpler alternative that may be convenient for variant calling/filtering and similar exercises.

Details

VRanges extends GRanges to store the following components. Except where noted, the components are considered columns in the dataset, i.e., their lengths match the number of variants. Many columns can be stored as either an atomic vector or an Rle.

`ref` (`character`), the reference allele. The range (start/end/width) should always correspond to this sequence.

`alt` (`character/Rle`), the alternative allele (NA allowed). By convention there is only a single alt allele per element (row) of the VRanges. Many methods, like `match`, make this assumption.

`refCount` (`integer/Rle`), read count for the reference allele (NA allowed)

`altCount` (`integer/Rle`), read count for the alternative allele (NA allowed)

`totalCount` (`integer/Rle`), total read count at the position, must be at least `refCount+altCount` (NA allowed)

`sampleNames` (`factor/Rle`), name of the sample - results from multiple samplse can be combined into the same object (NA allowed)

`softFilterMatrix` (`matrix/FilterMatrix`), variant by filter matrix, `TRUE` where variant passed the filter; use a `FilterMatrix` to store the actual `FilterRules` object that was applied

`hardFilters` (`FilterRules`) record of hard filters applied, i.e., only the variants that passed the filters are present in this object; this is the only component that is not a column, i.e., its length does not match the number of variants

Except in the special circumstances described here, a `VRanges` may be treated like a `GRanges`. The range should span the sequence in `ref`. Indels are typically represented by the VCF convention, i.e., the start position is one upstream of the event. The strand is always constrained to be positive (+).

Indels, by convention, should be encoded VCF-style, with the upstream reference base prepended to the indel sequence. The ref/alt for a deletion of GCGT before A might be AGCGT/A and for an insertion might be A/AGCGT. Since the range always matches the `ref` sequence, this means a deletion will be the width of the deletion + 1, and an insertion is always of width 1.

VRanges and the VCF class: The VRanges and VCF classes encode different types of information and are semantically incompatible. While methods exist for converting a VCF object to a VRanges and vice versa, information is lost in the transformation. There is no way to collapse multiple rows of a VRanges at the same genomic position and accurately represent missing data. For this reason, it is not reasonable to assume that an object resulting from multiple conversions (VRanges -> VCF -> VRanges) will be equivalent to the original.

Constructors

`VRanges(seqnames = Rle(), ranges = IRanges(), ref = character(), alt = NA_character_, totalDepth = NA_integer_, refDepth = NA_integer_, altDepth = NA_integer_, ..., sampleNames = NA_character_, softFilterMatrix = FilterMatrix(matrix(nrow = length(gr), ncol = 0L), FilterRules()), hardFilters = FilterRules())`: Creates a VRanges object.

`seqnames` Rle object, character vector, or factor containing the sequence names.

`ranges` IRanges object containing the ranges.

`ref` character vector, containing the reference allele.

`alt` character vector/Rle, containing the alternative allele (NA allowed).

`totalDepth` integer vector/Rle, containing the total read depth (NA allowed).

`refDepth` integer vector/Rle, containing the reference read depth (NA allowed).

`altDepth` integer vector/Rle, containing the reference read depth (NA allowed).

`...` Arguments passed to the `GRanges` constructor.

`sampleNames` character/factor vector/Rle, containing the sample names (NA allowed).

`softFilterMatrix` a matrix (typically a `FilterMatrix`) of dimension variant by filter, with logical values indicating whether a variant passed the filter.

`hardFilters` a `FilterRules`, containing the filters that have already been applied to subset the object to its current state.

`makeVRangesFromGRanges(gr, ref.field="ref", alt.field="alt", totalDepth.field="totalDepth", refDepth.field="refDepth", altDepth.field="altDepth", sampleNames.field="sampleNames", keep.extra.columns=TRUE)`: Creates a VRanges object from a GRanges.

`gr` A GenomicRanges object.

`ref.field` The `character(1)` name of the GRanges metadata column to be used as the VRanges `ref` field.

`alt.field` The `character(1)` name of the GRanges metadata column to be used as the VRanges `alt` field.

`totalDepth.field` The `character(1)` name of the GRanges metadata column to be used as the VRanges `totalDepth` field.

`refDepth.field` The `character(1)` name of the GRanges metadata column to be used as the VRanges `refDepth` field.

`altDepth.field` The `character(1)` name of the GRanges metadata column to be used as the VRanges `altDepth` field.

`sampleNames.field` The `character(1)` name of the GRanges metadata column to be used as the VRanges `sampleNames` field.

`keep.extra.columns` `TRUE` (the default) or `FALSE`. If `TRUE`, then the columns in `gr` that are not used to form the VRanges are retained as metadata columns. Otherwise, they will be ignored.

Coercion

These functions/methods coerce objects to and from `VRanges`:

`asVCF(x, info = character(), filter = character(), meta = character())`: Creates a VCF object from a VRanges object. The following gives the mapping from VRanges components to VCF:

seqnames(x) CHROM column

start(x) POS column

names(x) ID column

ref(x) REF column

alt(x) ALT column

totalDepth(x) DP in FORMAT column

altDepth(x), refDepth(x) AD in FORMAT column

sampleNames(x) Names the sample columns

softFilterMatrix(x) FT in FORMAT column, except filters named in `filter` argument, which are considered per-position and placed in the FILTER column

hardFilters(x) Not yet exported

mcols(x) Become fields in the FORMAT column; unless they are named in the `info` argument, in which case they are considered per-position and placed in the INFO column

metadata(x) If named in the `meta` argument, output in the VCF header; a component is required to be coercible to a character vector of length one.

Note that `identical(x, as(as(x, "VCF"), "VRanges"))` generally return `FALSE`. During coercion to VCF, the "geno" components are reshaped into matrix form, with NAs filling the empty cells. The reverse coercion will not drop the NA values, so rows are added to the new VRanges. All logical values will become integers in VCF, and there is no automatic way of regenerating the logical column with the reverse coercion. There are many other cases of irreversibility.

`as(from, "VCF")`: Like calling `asVCF(from)`.

`as(from, "VRanges")`: When `from` is a `VCF` this coercion is essentially the inverse of `asVCF`. Information missing in the VCF is imputed as NA.

When `from` is a `GRanges`, metadata columns of `ref`, `alt`, `refDepth`, `altDepth`, `totalDepth` and `sampleNames` are transfered to the `VRanges` object. Additional metadata columns in the `GRanges` can be retained or dropped with `keep.extra.columns`. See also `makeVRangesFromGRanges`.

Accessors

In addition to all of the `GRanges` accessors, `VRanges` provides the following, where `x` is a VRanges object.

`alt(x)`, `alt(x) <- value`: Get or set the alt allele (character).

`ref(x)`, `ref(x) <- value`: Get or set the ref allele (character).

`altDepth(x)`, `altDepth(x) <- value`: Get or set the alt allele read depth (integer).

`refDepth(x)`, `refDepth(x) <- value`: Get or set the ref allele read depth (integer).

`totalDepth(x)`, `totalDepth(x) <- value`: Get or set the total read depth (integer).

`altFraction(x)`: Returns `altDepth(x)/totalDepth(x)` (numeric).

`sampleNames(x)`, `sampleNames(x) <- value`: Get or set the sample names (character/factor).

`softFilterMatrix(x)`, `softFilterMatrix(x) <- value`: Gets or sets the soft filter matrix (any matrix, but ideally a `FilterMatrix`).

`resetFilter(x)`: Removes all columns from `softFilterMatrix`.

`called(x)`: Returns whether all filter results in `softFilterMatrix(x)` are `TRUE` for each variant.

`hardFilters(x)`, `hardFilters(x) <- value`: Gets or sets the hard filters (those applied to yield the current subset).

Utilities and Conveniences

`match(x)`: Like GRanges `match`, except matches on the combination of chromosome, start, width, and **alt**.

`tabulate(bin)`: Finds `unique(bin)` and counts how many times each unique element occurs in `bin`. The result is stored in `mcols(bin)$sample.count`.

`softFilter(x, filters, ...)`: applies the `FilterRules` in `filters` to `x`, storing the results in `softFilterMatrix`.

Input/Output to/from VCF

`writeVcf(obj, filename, ...)`: coerces to a VCF object and writes it to a file; see `writeVcf`.

`readVcfAsVRanges(x, genome, param = ScanVcfParam(), ...)`: Reads a VCF `x` directly into a `VRanges`; see `readVcf` for details on the arguments. `readVcfAsVRanges` is an alternative syntax to

```
as(readVcf(), "VRanges")
```

NOTE: By default all INFO and FORMAT fields are read in with `ScanVcfParam()`. The minimal information needed to create the `VRanges` can be specified as follows:

```
ScanVcfParam(fixed = "ALT", info = NA, geno = "AD"))
```

Variant Type

Functions to identify variant type include isSNV, isInsertion, isDeletion, isIndel, isSubstitution and isTransition. See the ?`isSNV` man page for details.

Author(s)

Michael Lawrence. `makeVRangesFromGRanges` was contributed by Thomas Sandmann.

See Also

VRangesList, a list of `VRanges`; `bam_tally` in the gmapR package, which generates a `VRanges`.

Examples

```
## construction
vr <- VRanges(seqnames = c("chr1", "chr2"),
              ranges = IRanges(c(1, 10), c(5, 20)),
              ref = c("T", "A"), alt = c("C", "T"),
              refDepth = c(5, 10), altDepth = c(7, 6),
              totalDepth = c(12, 17), sampleNames = letters[1:2],
              hardFilters =
                FilterRules(list(coverage = function(x) totalDepth > 10)),
              softFilterMatrix =
                FilterMatrix(matrix = cbind(depth = c(TRUE, FALSE)),
                             FilterRules(depth = function(x) altDepth(x) > 6)),
              tumorSpecific = c(FALSE, TRUE))

## simple accessors
ref(vr)
alt(vr)
altDepth(vr)
vr$tumorSpecific
called(vr)

## coerce to VCF and write
vcf <- as(vr, "VCF")
## writeVcf(vcf, "example.vcf")
## or just
```

```
## writeVcf(vr, "example.vcf")

## other utilities
match(vr, vr[2:1])
```

VRangesList-class *VRangesList objects*

Description

VRangesList is a virtual class representing a list of VRanges objects and should behave much like any other derivative of List. It has both a simple and compressed implementation. VRangesList provides conveniences for manipulating sets of VRanges objects.

Constructor

VRangesList(...): Creates a VRangesList object from VRanges objects in ....

Accessors

alt(x): Returns a CharacterList or RleList, effectively by calling alt(x[[i]]) on each element of x.

ref(x): Returns a CharacterList, effectively by calling ref(x[[i]]) on each element of x.

Utilities

stackSamples(x): Concentrates the elements in x, using names(x) to appropriately fill sampleNames in the result.

Author(s)

Michael Lawrence

Examples

```
## construction
example(VRanges)
vrl <- VRangesList(sampleA = vr, sampleB = vr)
stackSamples(vrl)
```

writeVcf *Write VCF files*

Description

Write Variant Call Format (VCF) files to disk

Usage

```
## S4 method for signature 'VCF,character'
writeVcf(obj, filename, index = FALSE, ...)
## S4 method for signature 'VCF,connection'
writeVcf(obj, filename, index = FALSE, ...)
```

Arguments

obj	Object containing data to be written out. At present only accepts VCF.
filename	The character() name of the VCF file, or a connection (e.g., file()), to be written out. A connection opened with open = "a" will have header information written only if the file does not already exist.
index	Whether to bgzip the output file and generate a tabix index.
...	Additional arguments, passed to methods. • nchunk: Integer or NA. When provided this argument overrides the default chunking behavior of writeVcf, see Details section. An integer value specifies the number of records in each chunk; NA disables chunking.

Details

A VCF file can be written out from data in a VCF object. More general methods to write out from other objects may be added in the future.

writeVcf writes out the header fields in a VCF object 'as-is' with the exception of these key-value pairs:

- fileformat: When missing, a line is added at the top of the file with the current supported version. VariantAnnotation >=1.27.6 supports VCFv4.3.
- fileDate: When missing, a line is added with today's date. If the key-value pair exists, the date is overwritten with today's date.
- contig: When missing, VariantAnnotation attempts to use the Seqinfo of the VCF object to determine the contig information.

Large VCF files (i.e., > 1e5 records) are written out in chunks; VCF files with < 1e5 records are not chunked. The optimal number of records per chunk depends on both the number of records and complexity of the data. Currently writeVcf determines records per chunk based on the total number of records only. To override this behavior or experiment with other values use nchunk as an integer or NA. An integer value represents the number of records per chunk regardless of the size of the VCF; NA disables all chunking.

- writeVcf(vcf, tempfile()) ## default chunking
- writeVcf(vcf, tempfile(), nchunk = 1e6) ## chunk by 1e6
- writeVcf(vcf, tempfile(), nchunk = NA) ## no chunking

Value

VCF file

Note

NOTE: `VariantAnnotation` >= 1.27.6 supports VCFv4.3. See the NOTE on the `?VCFHeader` man page under the `meta()` extractor for a description of how header parsing has changed to accommodate the new header lines with key name of 'META'.

Author(s)

Valerie Obenchain and Michael Lawrence

References

http://vcftools.sourceforge.net/specs.html outlines the VCF specification.

http://samtools.sourceforge.net/mpileup.shtml contains information on the portion of the specification implemented by `bcftools`.

http://samtools.sourceforge.net/ provides information on `samtools`.

See Also

readVcf

Examples

```
fl <- system.file("extdata", "ex2.vcf", package="VariantAnnotation")

out1.vcf <- tempfile()
out2.vcf <- tempfile()
in1 <- readVcf(fl, "hg19")
writeVcf(in1, out1.vcf)
in2 <- readVcf(out1.vcf, "hg19")
writeVcf(in2, out2.vcf)
in3 <- readVcf(out2.vcf, "hg19")
stopifnot(all(in2 == in3))

## write incrementally
out3.vcf <- tempfile()
con <- file(out3.vcf, open="a")
writeVcf(in1[1:2,], con)
writeVcf(in1[-(1:2),], con)
close(con)
readVcf(out3.vcf, "hg19")
```

Index