Introduction to *VariantAnnotation*

Valerie Obenchain

August 28, 2012

Contents

1 Introduction

This vignette outlines a general workflow for annotating and filtering genetic variants using the *VariantAnnotation*package. Sample data are in VariantCall Format (VCF) and are a subset of chromosome 22 from 1000 Genomes, `ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/release/20110521/`. VCF is a text file format that contains meta-information lines, a header line with column names, data lines with information about a position in the genome, and optional genotype information on samples for each position. A full description of the VCF format can be found on the 1000 Genomes page, `http://www.1000genomes.org/wiki/Analysis/Variant%20Call%20Format/vcf-variant-call-format-version-41`

The sample dataare read in from a VCF file and variants are identified according to region such as `coding`, `intron`, `intergenic`, `spliceSite` etc. Amino acid coding changes are computed for the non-synonymous variants and SIFT and PolyPhen databases provide predictions of how severly the coding changes affect protein function. The end of the vignette covers other transformations of VCF data such as the creation of a `SnpMatrix` or a 'long form' `GRanges`.

2 Variant Call Format (VCF) files

2.1 Import complete files

Data are parsed into a `VCF` object with `readVcf`.

```
> library(VariantAnnotation)
> fl <- system.file("extdata", "chr22.vcf.gz", package="VariantAnnotation")
> vcf <- readVcf(fl, "hg19")
> vcf
```

```
class: VCF
dim: 10376 5
genome: hg19
exptData(1): header
fixed(4): REF ALT QUAL FILTER
info(22): LDAF AVGPOST ... VT SNPSOURCE
geno(3): GT DS GL
rownames(10376): rs7410291 rs147922003 ... rs144055359
  rs114526001
rowData values names(1): paramRangeID
colnames(5): HG00096 HG00097 HG00099 HG00100 HG00101
colData names(1): Samples
```

Extract the header information stored in the `exptData` slot

```
> hdr <- exptData(vcf)[["header"]]
> hdr
```

```
class: VCFHeader
samples(5): HG00096 HG00097 HG00099 HG00100 HG00101
meta(1): fileformat
fixed(1): ALT
info(22): LDAF AVGPOST ... VT SNPSOURCE
geno(3): GT DS GL
```

and explore it with the `fixed`, `info` and `geno` accessors. More information on this object can be found at ?`VCFHeader`.

```
> fixed(hdr)
```

```
SimpleDataFrameList of length 1
names(1): ALT
```

```
> head(info(hdr), 3)
```

```
DataFrame with 3 rows and 3 columns
             Number        Type
        <character> <character>
LDAF              1       Float
AVGPOST           1       Float
RSQ               1       Float
                                          Description
                                          <character>
```

```
LDAF             MLE Allele Frequency Accounting for LD
AVGPOST Average posterior probability from MaCH/Thunder
RSQ       Genotype imputation quality from MaCH/Thunder
```

The `GRanges` in the `rowData` slot is created from information in the the CHROM, POS, and ID fields of the VCF file. Values in the `paramRangeID` column are meaningful when ranges have been specified in the the `param` argument to `readVcf`. This is discussed further in the `Data Subsets` section.

```
> head(rowData(vcf))
```

```
GRanges with 6 ranges and 1 elementMetadata col:
              seqnames               ranges strand | paramRangeID
                 <Rle>            <IRanges>  <Rle> |     <factor>
    rs7410291       22 [50300078, 50300078]      * |         <NA>
  rs147922003       22 [50300086, 50300086]      * |         <NA>
  rs114143073       22 [50300101, 50300101]      * |         <NA>
  rs141778433       22 [50300113, 50300113]      * |         <NA>
  rs182170314       22 [50300166, 50300166]      * |         <NA>
  rs115145310       22 [50300187, 50300187]      * |         <NA>
  ---
  seqlengths:
   22
   NA
```

The REF, ALT, QUAL and FILTER fields can be accessed together with `fixed` accessor or individually with `ref`, `alt`, `qual` and `filt` accessors.

```
> head(fixed(vcf), 3)
```

```
GRanges with 3 ranges and 5 elementMetadata cols:
              seqnames               ranges strand | paramRangeID
                 <Rle>            <IRanges>  <Rle> |     <factor>
    rs7410291       22 [50300078, 50300078]      * |         <NA>
  rs147922003       22 [50300086, 50300086]      * |         <NA>
  rs114143073       22 [50300101, 50300101]      * |         <NA>
                       REF                ALT      QUAL      FILTER
              <DNAStringSet> <DNAStringSetList> <numeric> <character>
    rs7410291              A           ########       100        PASS
  rs147922003              C           ########       100        PASS
  rs114143073              G           ########       100        PASS
  ---
  seqlengths:
   22
   NA
```

```
> qual(vcf)[1:3,]
```

```
GRanges with 3 ranges and 2 elementMetadata cols:
              seqnames               ranges strand | paramRangeID
                 <Rle>            <IRanges>  <Rle> |     <factor>
    rs7410291       22 [50300078, 50300078]      * |         <NA>
  rs147922003       22 [50300086, 50300086]      * |         <NA>
  rs114143073       22 [50300101, 50300101]      * |         <NA>
```

```
                 QUAL
             <numeric>
   rs7410291       100
 rs147922003       100
 rs114143073       100
 ---
 seqlengths:
  22
  NA
```

The `ALT` column is stored as a `DNAStringSetList` unless the file is a structural VCF, in which case it is stored as a `CharacterList`. Extract `ALT` from the `GRanges` and determine the number of elements in the list.

```
> alternate <- values(alt(vcf))[["ALT"]]
> alternate

DNAStringSetList of length 10376

> ## number of ALT values per variant
> unique(elementLengths(alternate))

[1] 1

> head(unlist(alternate))

  A DNAStringSet instance of length 6
    width seq
[1]     1 G
[2]     1 T
[3]     1 A
[4]     1 T
[5]     1 T
[6]     1 A
```

Data from the `INFO` field is accessed with the `info` accessor. Look at the heder information related to the `INFO` fields

```
> head(info(hdr), 4)

DataFrame with 4 rows and 3 columns
             Number        Type
        <character> <character>
LDAF              1       Float
AVGPOST           1       Float
RSQ               1       Float
ERATE             1       Float
                                          Description
                                          <character>
LDAF             MLE Allele Frequency Accounting for LD
AVGPOST Average posterior probability from MaCH/Thunder
RSQ       Genotype imputation quality from MaCH/Thunder
ERATE        Per-marker Mutation rate from MaCH/Thunder
```

and retrieve the first couple of the fields,

```
> info(vcf)[1:3, 2:3]
```

```
GRanges with 3 ranges and 2 elementMetadata cols:
              seqnames              ranges strand |      LDAF
                 <Rle>           <IRanges>  <Rle> | <numeric>
    rs7410291       22 [50300078, 50300078]      * |    0.3431
  rs147922003       22 [50300086, 50300086]      * |    0.0091
  rs114143073       22 [50300101, 50300101]      * |    0.0098
                AVGPOST
              <numeric>
    rs7410291     0.989
  rs147922003    0.9963
  rs114143073    0.9891
  ---
  seqlengths:
   22
   NA
```

Genotype data described in the `FORMAT` field are parsed into matrices or arrays and can be accessed with the `geno` accessor. These data are not returned with the `GRanges` from `rowData` because they are unique for each sample and the data structures can be multidimensional. This is in contrast to the `fixed` and `info` data which are the same for a each variant across all samples.

Extract the header information for the genotypes.

```
> geno(hdr)
```

```
DataFrame with 3 rows and 3 columns
        Number        Type                        Description
   <character> <character>                        <character>
GT            1      String                           Genotype
DS            1       Float Genotype dosage from MaCH/Thunder
GL            .       Float              Genotype Likelihoods
```

Elements of the genotype list can be accessed in the usual way.

```
> geno(vcf)
```

```
SimpleList of length 3
names(3): GT DS GL
```

```
> geno(vcf)$GT[1:3,1:5]
```

```
            HG00096 HG00097 HG00099 HG00100 HG00101
rs7410291   "0|0"   "0|0"   "1|0"   "0|0"   "0|0"
rs147922003 "0|0"   "0|0"   "0|0"   "0|0"   "0|0"
rs114143073 "0|0"   "0|0"   "0|0"   "0|0"   "0|0"
```

```
> geno(vcf)$DS[1:3,1:5]
```

```
            HG00096 HG00097 HG00099 HG00100 HG00101
rs7410291         0       0       1       0       0
rs147922003       0       0       0       0       0
rs114143073       0       0       0       0       0
```

2.2 Import data subsets

When working with large VCF files it may be more efficient to read in subsets of the data. Data can be subset by selecting genomic coordinates (ranges) or by selecting fields from the VCF file.

2.2.1 Genomic coordinates

Subset by genomic coordinates by creating a `GRanges`, `RangedData` or `RangesList`. To read in a portion of chromosome 22, we create a `GRanges` with the regions of interest.

```
> rng <- GRanges(seqnames="22",
+                ranges=IRanges(c(50301422, 50989541), c(50312106, 51001328)))
> names(rng) <- c("gene_79087", "gene_644186")
```

When ranges are specified, the VCF file must have an accompanying Tabix index file; if one does not exist it must be created. See ?`indexTabix` for help creating an index.

Once the index exists a `TabixFile` instance can be created, see ?`TabixFile`. This object creates a reference to the VCF and its index. Once opened, the reference remains open across calls to methods, avoiding costly index re-loading. An index file for our sample data is included in the package so the `TabixFile` can be created with,

```
> tab <- TabixFile(fl)
> tab

class: TabixFile
path: /private/tmp/Rtmp5zo5D2/Rinst3e6d62ce3979/VariantA.../chr22.vcf.gz
index: /private/tmp/Rtmp5zo5D2/Rinst3e6d62ce3979/Var.../chr22.vcf.gz.tbi
isOpen: FALSE
```

Call `readVcf` with `TabixFile` and the ranges as the `param`. The dimension of the resulting `VCF` object shows 397 records overlaped with the specified ranges.

```
> vcf_rng <- readVcf(tab, "hg19", rng)
> vcf_rng

class: VCF
dim: 397 5
genome: hg19
exptData(1): header
fixed(4): REF ALT QUAL FILTER
info(22): LDAF AVGPOST ... VT SNPSOURCE
geno(3): DS GL GT
rownames(397): rs114335781 rs8135963 ... rs144055359
  rs114526001
rowData values names(1): paramRangeID
colnames(5): HG00096 HG00097 HG00099 HG00100 HG00101
colData names(1): Samples
```

The `paramRangesID` column now has meaning as it distinguishes which variant records came from which param range.

```
> rowData(vcf_rng)
```

```
GRanges with 397 ranges and 1 elementMetadata col:
               seqnames               ranges strand   | paramRangeID
                  <Rle>            <IRanges>  <Rle>   |     <factor>
  rs114335781       22 [50301422, 50301422]      *   |   gene_79087
    rs8135963       22 [50301476, 50301476]      *   |   gene_79087
  22:50301488       22 [50301488, 50301488]      *   |   gene_79087
  22:50301494       22 [50301494, 50301494]      *   |   gene_79087
  22:50301584       22 [50301584, 50301584]      *   |   gene_79087
  22:50301622       22 [50301622, 50301622]      *   |   gene_79087
    rs9627644       22 [50301664, 50301664]      *   |   gene_79087
  rs192916413       22 [50301719, 50301719]      *   |   gene_79087
  rs149477000       22 [50301774, 50301774]      *   |   gene_79087
          ...      ...                 ...    ... ...          ...
  rs138542635       22 [50999306, 50999306]      *   |  gene_644186
  rs184258531       22 [50999489, 50999489]      *   |  gene_644186
    rs9628177       22 [50999490, 50999490]      *   |  gene_644186
    rs9628212       22 [50999502, 50999502]      *   |  gene_644186
  rs187302552       22 [50999536, 50999536]      *   |  gene_644186
    rs9628178       22 [50999538, 50999538]      *   |  gene_644186
    rs5770892       22 [50999681, 50999681]      *   |  gene_644186
  rs144055359       22 [50999830, 50999830]      *   |  gene_644186
  rs114526001       22 [50999964, 50999964]      *   |  gene_644186
  ---
  seqlengths:
   22
   NA
```

2.2.2 VCF fields

Data can also be subset on the `fixed`, `info` and/or `geno` fields in the VCF file. Fields available for import are described in the header information. To view the header before reading in the data in use `ScanVcfHeader`.

```
> hdr <- scanVcfHeader(fl)
> hdr
```

```
class: VCFHeader
samples(5): HG00096 HG00097 HG00099 HG00100 HG00101
meta(1): fileformat
fixed(1): ALT
info(22): LDAF AVGPOST ... VT SNPSOURCE
geno(3): GT DS GL
```

The `info` and `geno` accessors return `DataFrames` containing descriptions of the fields, data type and number of values. A listing of all possible `info` or `geno` values is constructed by selecting the rownames of the `DataFrames`.

```
> ## INFO fields
> info_DF <- info(hdr)
> rownames(info_DF)
```

```
 [1] "LDAF"      "AVGPOST"   "RSQ"       "ERATE"     "THETA"
 [6] "CIEND"     "CIPOS"     "END"       "HOMLEN"    "HOMSEQ"
[11] "SVLEN"     "SVTYPE"    "AC"        "AN"        "AA"
```

```
[16] "AF"        "AMR_AF"    "ASN_AF"    "AFR_AF"    "EUR_AF"
[21] "VT"        "SNPSOURCE"
```

```
> ## FORMAT fields
> geno_DF <- geno(hdr)
> rownames(geno_DF)
```

```
[1] "GT" "DS" "GL"
```

We are interested in "LDAF" in INFO which is 'allele frequency accounting for linkage disequlibrium', and "GT" in FORMAT which is 'genotype'. Full descriptions of the elements can be seen in the header INFO and FORMAT `DataFrames`.

```
> info_DF[rownames(info_DF) == "LDAF", ]
```

```
DataFrame with 1 row and 3 columns
          Number        Type                              Description
     <character> <character>                              <character>
LDAF           1       Float MLE Allele Frequency Accounting for LD
```

```
> geno_DF[rownames(geno_DF) == "GT", ]
```

```
DataFrame with 1 row and 3 columns
        Number        Type Description
   <character> <character> <character>
GT           1      String    Genotype
```

To subset on "LDAF" and "GT" we specify them as `character` vectors in the `info` and `geno` arguments to `ScanVcfParam`. This creates a `ScanVcfParam` object which is used as the `param` argument to `readVcf`.

```
> ## Return "ALT" from 'fixed', "LAF" from 'info' and "GT" from 'geno'
> svp <- ScanVcfParam(fixed="ALT", info="LDAF", geno="GT")
> ## Return all 'fixed' fields, "LAF" from 'info' and "GT" from 'geno'
> svp <- ScanVcfParam(info="LDAF", geno="GT")
> svp
```

```
class: ScanVcfParam
vcfWhich: 0 elements
vcfFixed: character() [All]
vcfInfo: LDAF
vcfGeno: GT
```

Note that subsetting by the VCF fields does not affect the number of ranges read in. Instead the results of the filtering are reflected in the names of the elements returned from the `info` and `geno` accessors.

```
> vcf_flds <- readVcf(fl, "hg19", svp)
> geno(vcf_flds)
```

```
SimpleList of length 1
names(1): GT
```

```
> head(info(vcf_flds), 3)
```

```
GRanges with 3 ranges and 2 elementMetadata cols:
              seqnames               ranges strand | paramRangeID
                 <Rle>            <IRanges>  <Rle> |     <factor>
    rs7410291       22 [50300078, 50300078]      * |         <NA>
  rs147922003       22 [50300086, 50300086]      * |         <NA>
  rs114143073       22 [50300101, 50300101]      * |         <NA>
                 LDAF
              <numeric>
    rs7410291    0.3431
  rs147922003    0.0091
  rs114143073    0.0098
  ---
  seqlengths:
   22
   NA
```

In the previous section we saw that a Tabix index file must exist when data are subset by genomic coordinates (i.e., ranges). This is not the case when subsetting on INFO and FORMAT elements. An index file is only needed when subsetting by ranges.

2.2.3 Subset on both genomic coordinates and VCF fields

To subset on both genomic coordinates and INFO and FORMAT fields the `ScanVcfParam` object must contain both. Our previous `ScanVcfParam` did not have ranges associated with it so we create a new instance with the ranges and INFO and FORMAT fields.

```
> svp_all <- ScanVcfParam(info="LDAF", geno="GT", which=rng)
> svp_all

class: ScanVcfParam
vcfWhich: 1 elements
vcfFixed: character() [All]
vcfInfo: LDAF
vcfGeno: GT
```

The subsetting here involves genomic coordinates so we need to use the Tabix index file we created.

```
> readVcf(tab, "hg19", svp_all)

class: VCF
dim: 397 5
genome: hg19
exptData(1): header
fixed(4): REF ALT QUAL FILTER
info(1): LDAF
geno(1): GT
rownames(397): rs114335781 rs8135963 ... rs144055359
  rs114526001
rowData values names(1): paramRangeID
colnames(5): HG00096 HG00097 HG00099 HG00100 HG00101
colData names(1): Samples
```

2.3 Adjusting chromosome names

When functions involve the comparision of ranges by overlaps. For overlap methods to work properly the chromosome names (seqlevels) must be compatible.

The VCF data chromosome names are represented by number, i.e. '22',

```
> rowdat <- rowData(vcf)
> seqlevels(rowdat)

[1] "22"
```

but the TxDb chromosome names are preceded with 'chr'.

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

[1] "chr1" "chr2" "chr3" "chr4" "chr5" "chr6"
```

Chromosome names can be modified with the `renameSeqlevels` function. Seqlevels are modified at the `GRanges` level in the `rowData` slot of the `VCF` which means all future data extractions from this `VCF` will have the new seqlevels. If the data are read in from the file again, however, the seqlevels will need to be adjusted again. See ?`VCF` and ?`renameSeqlevels` for examples with `VCF` and `GRanges` objects.

```
> ## rename variant seqlevels in the VCF object
> vcf <- renameSeqlevels(vcf, c("22"="chr22"))
> ## extract the rowData with modified seqlevels
> rd <- rowData(vcf)
> ## confirm seqlevels are the same
> intersect(seqlevels(rd), seqlevels(txdb))

[1] "chr22"
```

To subset a `VCF` or `GRanges` by chromosome use `keepSeqlevels`. As an example we extract transcripts for all chromosomes in TxDb.Hsapiens.UCSC.hg19.knownGene then keep only 'chr21' and 'chr22'. See ?`VCF` and ?`keepSeqlevels` for details.

```
## initially there are 93 chromosomes
> rngs <- transcripts(txdb)
> length(seqlevels(rngs))
[1] 93
## keep only chr21 and chr22
> rngs <- keepSeqlevels(rngs, c("chr21", "chr22"))
> seqlevels(rngs)
[1] "chr21" "chr22"
```

3 Variant location

`locateVariants` identifies where the ranges in `query` fall with respect to the annotation supplied in `subject`. Regions are specified in the `region` argument and can be one of the following constructors: CodingVariants, IntronVariants, FiveUTRVariants, ThreeUTRVariants, IntergenicVariants, or SpliceSiteVariants. Location definitions are shown in Table 1.

When the `query` is a `VCF` the variant ranges are taken from the `rowData` slot. If `query` is a `GRanges` it can have additional elementMetadata columns but they are ignored. As an alternative to a `TranscriptDb`,

Location	Details
coding	falls *within* a coding region
fiveUTR	falls *within* a 5' untranslated region
threeUTR	falls *within* a 3' untranslated region
intron	falls *within* an intron region
intergenic	does not fall *within* a transcript associated with a gene
spliceSite	overlaps any portion of the first 2 or last 2 nucleotides of an intron

Table 1: Variant locations

the `subject` can be a `GRangesList` of the appropriate type. `CodingVariants` would require coding regions by transcript, for `IntronVariants` introns by transcripts would be necessary, etc. See ?`locateVariants` man page for details.

Identify the coding variants,

```
> loc <- locateVariants(rd, txdb, CodingVariants())
> head(loc, 4)

GRanges with 4 ranges and 5 elementMetadata cols:
              seqnames               ranges strand | location   queryID
                 <Rle>            <IRanges>  <Rle> | <factor> <integer>
  rs114335781    chr22 [50301422, 50301422]      * |   coding        24
    rs8135963    chr22 [50301476, 50301476]      * |   coding        25
  22:50301488    chr22 [50301488, 50301488]      * |   coding        26
  22:50301494    chr22 [50301494, 50301494]      * |   coding        27
                   txID     cdsID      geneID
              <integer> <integer> <character>
  rs114335781     76833    225251       79087
    rs8135963     76833    225251       79087
  22:50301488     76833    225251       79087
  22:50301494     76833    225251       79087
  ---
  seqlengths:
   chr22
      NA
```

`SpliceSiteVariants` are those overlapping the first 2 or last 2 nucleotides of an intron.

```
> head(locateVariants(rd, txdb, SpliceSiteVariants()), 4)

GRanges with 4 ranges and 5 elementMetadata cols:
             seqnames               ranges strand |   location
                <Rle>            <IRanges>  <Rle> |   <factor>
  rs35683648    chr22 [50754200, 50754202]      * | spliceSite
    rs140524    chr22 [50960682, 50960682]      * | spliceSite
    rs140524    chr22 [50960682, 50960682]      * | spliceSite
    rs140524    chr22 [50960682, 50960682]      * | spliceSite
               queryID      txID     cdsID      geneID
             <integer> <integer> <integer> <character>
  rs35683648      6618     76889        <NA>      414918
    rs140524      9740     76909        <NA>       29781
```

```
    rs140524      9740     76910      <NA>       29781
    rs140524      9740     76911      <NA>       29781
  ---
  seqlengths:
   chr22
      NA
```

To locate variants in all regions use the `AllVariants()` constructor,

```
> allvar <- locateVariants(rd, txdb, AllVariants())
```

The `GRanges` output of `locateVariants` includes only the ranges that fell in the specified region. Each row is a variant-transcript match which may result in multiple rows for each variant. `elementMetadata` columns returned include `location`, `queryID`, `txID`, `cdsID`, and `geneID`. In the case of `IntergenicVariants` columns for `precedesID` and `followsID` are also included. The `queryID` column maps back to the row number in the original `query`.

To answer gene-centric questions data can be summarized by gene reguardless of transcript.

```
> ## Did any coding variants match more than one gene?
> table(sapply(split(values(loc)[["geneID"]], values(loc)[["queryID"]]),
+     function(x) length(unique(x)) > 1))

FALSE  TRUE
  956    15

> ## Summarize the number of coding variants by gene ID
> idx <- sapply(split(values(loc)[["queryID"]], values(loc)[["geneID"]]), unique)
> sapply(idx, length)

113730   1890  23209  23654  29781 400935 414918 415116 440836  54456
    22     15     30     87     44     15     33     11      5     82
 55586   5600  56666   6300   6305 644186  79087  79174  79924  80305
    24     16     19     38     56      5     25     50      4     26
 83642  83933  85378  91289   9701   9997
    55     50    147     29     68     15
```

4 Amino acid coding changes

`predictCoding` computes amino acid coding changes for non-synonymous variants. Only ranges in `query` that overlap with a coding region in the `subject` are considered. Reference sequences are retrieved from either a `BSgenome` or fasta file specified in `seqSource`. Variant sequences are constructed by substituting, inserting or deleting values in the `varAllele` column into the reference sequence. Amino acid codes are computed for the variant codon sequence when the length is a multiple of 3. Examples of coding situations are shown in Table 2.

The `query` argument to `predictCoding` can be a `GRanges` or `VCF`. When a `GRanges` is supplied the `varAllele` argument must be specified. In the case of a `VCF`, the alternate alleles are taken from `values(alt(<VCF>))[["ALT"]]` and the `varAllele` argument is not specified.

The result is a modified `query` containing only variants that fall within coding regions. Each row represents a variant-transcript match so more than one row per original variant is possible.

```
> library(BSgenome.Hsapiens.UCSC.hg19)
> coding <- predictCoding(vcf, txdb, seqSource=Hsapiens)
> coding[5:9]
```

Type	refAllele	varAllele	refCodon	varCodon	translation possible
substitution	G	T	aag	aaT	yes
substitution	G	TG	tga	tTGa	no
substitution	G	TGCG	gtc	TGCGtc	yes
insertion	‘’	G	cgg	Gcgg	no
insertion	‘’	TTG	gaa	gaTTGa	yes
deletion	A	‘’	atc	tc	no
deletion	GGCCTA	‘’	acggcctaa	aca	yes

Table 2: Amino acid coding

```
GRanges with 5 ranges and 13 elementMetadata cols:
              seqnames              ranges strand | paramRangeID
                 <Rle>           <IRanges>  <Rle> |     <factor>
  22:50301584    chr22 [50301584, 50301584]      - |         <NA>
  rs114264124    chr22 [50302962, 50302962]      - |         <NA>
  rs149209714    chr22 [50302995, 50302995]      - |         <NA>
  22:50303554    chr22 [50303554, 50303554]      - |         <NA>
   rs12167668    chr22 [50303561, 50303561]      - |         <NA>
                   varAllele     cdsLoc              proteinLoc
              <DNAStringSet>  <IRanges> <CompressedIntegerList>
  22:50301584              A [777, 777]                     259
  rs114264124              A [698, 698]                     233
  rs149209714              C [665, 665]                     222
  22:50303554              G [652, 652]                     218
   rs12167668              A [645, 645]                     215
                queryID        txID     cdsID      geneID   consequence
              <integer> <character> <integer> <character>      <factor>
  22:50301584        28       76833    225251       79087    synonymous
  rs114264124        57       76833    225252       79087 nonsynonymous
  rs149209714        58       76833    225252       79087 nonsynonymous
  22:50303554        73       76833    225253       79087 nonsynonymous
   rs12167668        74       76833    225253       79087    synonymous
                 refCodon       varCodon         refAA         varAA
              <DNAStringSet> <DNAStringSet> <AAStringSet> <AAStringSet>
  22:50301584            CCG            CCA             P             P
  rs114264124            CGG            CAG             R             Q
  rs149209714            GGA            GCA             G             A
  22:50303554            ATC            GTC             I             V
   rs12167668            CCG            CCA             P             P
  ---
  seqlengths:
   chr22
      NA
```

Using variant rs114264124 as an example, we see varAllele `A` has been substituted into the `refCodon CGG` to produce `varCodon CAG`. The `refCodon` is the sequence of codons necessary to make the variant allele substitution and therefore often includes more nucleotides than indicated in the range (i.e. the range is 50302962, 50302962, width of 1). Notice it is the second position in the `refCodon` that has been substituted. This position in the codon, the position of substitution, corresponds to genomic position 50302962. This

genomic position maps to position 698 in coding region-based coordinates and to triplet 233 in the protein. This is a non-synonymous coding variant where the amino acid has changed from `R` (Arg) to `Q` (Gln).

When the resulting `varCodon` is not a multiple of 3 it cannot be translated. The consequence is considered a `frameshift` and `varAA` will be missing.

```
> ## consequence is 'frameshift' where translation is not possible
> coding[values(coding)[["consequence"]] == "frameshift"]

GRanges with 1 range and 13 elementMetadata cols:
              seqnames              ranges strand | paramRangeID
                 <Rle>           <IRanges>  <Rle> |     <factor>
  22:50317001    chr22 [50317001, 50317001]      + |         <NA>
                   varAllele     cdsLoc              proteinLoc
              <DNAStringSet>  <IRanges> <CompressedIntegerList>
  22:50317001          GCACT [808, 808]                     270
                queryID        txID     cdsID      geneID consequence
              <integer> <character> <integer> <character>    <factor>
  22:50317001       359       76834    225263       79174  frameshift
                    refCodon       varCodon         refAA         varAA
              <DNAStringSet> <DNAStringSet> <AAStringSet> <AAStringSet>
  22:50317001            GCC            ACC             A
  ---
  seqlengths:
   chr22
      NA
```

5 SIFT and PolyPhen Databases

From `predictCoding` we identified the amino acid coding changes for the non-synonymous variants. For this subset we can retrieve predictions of how damaging these coding changes may be. SIFT (Sorting Intolerant From Tolerant) and PolyPhen (Polymorphism Phenotyping) are methods that predict the impact of amino acid substitution on a human protein. The SIFT method uses sequence homology and the physical properties of amino acids to make predictions about protein function. PolyPhen uses sequence-based features and structural information characterizing the substitution to make predictions about the structure and function of the protein.

Collated predictions for specific dbSNP builds are available as downloads from the SIFT and PolyPhen web sites. These results have been packaged into *SIFT.Hsapiens.dbSNP132.db* and *PolyPhen.Hapiens.dbSNP131.db* and are designed to be searched by rsid. Variants that are in dbSNP can be searched with these database packages. When working with novel variants, SIFT and PolyPhen must be called directly. See references for home pages.

Identify the non-synonymous variants and obtain the rsids.

```
> nms <- names(coding)
> idx <- values(coding)[["consequence"]] == "nonsynonymous"
> nonsyn <- coding[idx]
> names(nonsyn) <- nms[idx]
> rsids <- unique(names(nonsyn)[grep("rs", names(nonsyn), fixed=TRUE)])
```

Detailed descriptions of the database columns can be found with `?SIFTDbColumns` and `?PolyPhenDbColumns`. Variants in these databases often contain more than one row per variant. The variant may have been reported by multiple sources and therefore the source will differ as well as some of the other variables.

```
> library(SIFT.Hsapiens.dbSNP132)
> ## rsids in the package
> head(keys(SIFT.Hsapiens.dbSNP132))

[1] "rs10000692" "rs10001580" "rs10002700" "rs10003238" "rs10003369"
[6] "rs10004"

> ## list available columns
> cols(SIFT.Hsapiens.dbSNP132)

 [1] "RSID"        "PROTEINID"   "AACHANGE"    "METHOD"
 [5] "AA"          "PREDICTION"  "SCORE"       "MEDIAN"
 [9] "POSTIONSEQS" "TOTALSEQS"

> ## select a subset of columns
> ## a warning is thrown when a key is not found in the database
> subst <- c("RSID", "PREDICTION", "SCORE", "AACHANGE", "PROTEINID")
> sift <- select(SIFT.Hsapiens.dbSNP132, keys=rsids, cols=subst)
> head(sift)

         RSID PROTEINID AACHANGE PREDICTION SCORE
1 rs114264124 NP_077010    R233Q  TOLERATED  0.59
2 rs114264124 NP_077010    R233Q  TOLERATED  1.00
3 rs114264124 NP_077010    R233Q  TOLERATED  0.20
4 rs149209714      <NA>     <NA>       <NA>  <NA>
5 rs144665682      <NA>     <NA>       <NA>  <NA>
6 rs117687848 NP_077010    R120Q  TOLERATED  0.20
```

Next we query the PolyPhen database for information on these variants. PolyPhen provides predictions using two different training datasets and has considerable information about 3D protein structure. See `?PolyPhenDbColumns` or the PolyPhen web site listed in the references for more details.

Query the PolyPhen database with the rsids found in SIFT,

```
> library(PolyPhen.Hsapiens.dbSNP131)
> inSIFT <- unique(sift$RSID[!is.na(sift$PREDICTION)])
> pp <- select(PolyPhen.Hsapiens.dbSNP131, keys=inSIFT,
+              cols=c("TRAININGSET", "PREDICTION", "PPH2PROB"))
> head(pp[!is.na(pp$PREDICTION), ])

        RSID TRAININGSET        PREDICTION PPH2PROB
4  rs8139422      humdiv possibly damaging    0.228
5  rs8139422      humvar possibly damaging    0.249
6 rs74510325      humdiv possibly damaging    0.475
7 rs74510325      humvar possibly damaging    0.335
8 rs73891177      humdiv            benign    0.001
9 rs73891177      humvar            benign    0.005
```

6 Other operations

6.1 Create a SnpMatrix

The 'GT' element in the `FORMAT` field of the VCF represents the genotype. These data can be converted into a `snpMatrix` object which can then be used with the functions offered in *snpStats* and other packages making use of the `SnpMatrix` class.

The MatrixToSnpMatrix function converts the genotype calls in geno to a SnpMatrix. No dbSNP package is used in this computation. The return value is a named list where 'genotypes' is a SnpMatrix and 'map' is a DataFrame with SNP names and alleles at each loci. The ignore column in 'map' indicates which variants were set to NA (missing) because they met one or more of the following criteria,

- only diploid calls are included; others are set to NA
- only single nucleotide variants are included; others are set to NA
- variants with >1 ALT allele are set to NA

See ?MatrixToSnpMatrix for more details.

```
> calls <- geno(vcf)$GT
> a0 <- values(ref(vcf))[["REF"]]
> a1 <- values(alt(vcf))[["ALT"]]
> res <- MatrixToSnpMatrix(calls, a0, a1)
> res
```

```
$genotypes
A SnpMatrix with  5 rows and  10376 columns
Row names:  HG00096 ... HG00101
Col names:  rs7410291 ... rs114526001

$map
DataFrame with 10376 rows and 4 columns
        snp.names       allele.1           allele.2    ignore
      <character> <DNAStringSet> <DNAStringSetList> <logical>
1       rs7410291              A           ########     FALSE
2     rs147922003              C           ########     FALSE
3     rs114143073              G           ########     FALSE
4     rs141778433              C           ########     FALSE
5     rs182170314              C           ########     FALSE
6     rs115145310              G           ########     FALSE
7     rs186769856              T           ########     FALSE
8      rs77627744              G           ########     FALSE
9     rs193230365              G           ########     FALSE
...           ...            ...                ...       ...
10368 rs138542635              G           ########     FALSE
10369 rs184258531              C           ########     FALSE
10370   rs9628177              G           ########     FALSE
10371   rs9628212              G           ########     FALSE
10372 rs187302552              A           ########     FALSE
10373   rs9628178              A           ########     FALSE
10374   rs5770892              A           ########     FALSE
10375 rs144055359              G           ########     FALSE
10376 rs114526001              G           ########     FALSE
```

The ALT value in the 'map' DataFrame will be a CharacterList if the VCF was for structural variants or a DNAStringSetList otherwise. The column is not clearly visable inside the DataFrame but can be extracted and inspected as follows,

```
> allele2 <- res$map[["allele.2"]]
> ## number of alternate alleles per variant
> unique(elementLengths(allele2))
```

```
[1] 1
```

```
> unlist(allele2)
```

```
  A DNAStringSet instance of length 10376
        width seq
    [1]     1 G
    [2]     1 T
    [3]     1 A
    [4]     1 T
    [5]     1 T
    [6]     1 A
    [7]     1 C
    [8]     1 A
    [9]     1 A
    ...   ... ...
[10368]     1 A
[10369]     1 T
[10370]     1 A
[10371]     1 A
[10372]     1 G
[10373]     1 G
[10374]     1 G
[10375]     1 A
[10376]     1 C
```

6.2 Long form GRanges

The `readVcfLongForm` function reads data from a VCF file in the same manner as `readVcf` but outputs a long form `GRanges` instead of a *VCF* class. This format is driven by the fact that the alternate allele (ALT) in the VCF file often has more than one value per record. In the long form `GRanges`, the rows of the `GRanges` are replicated to match the length of the 'unlisted' alternate allele. This format provides access to each possible REF, ALT pair for each variant.

Input arguments and data subsetting is the same for `readVcfLongForm` as for `readVcf`. The `fixed` and `info` fields are included as elementMetadata columns. Currently no `geno` information is included.

`info` information was previously collected from the file header. We import 'HOMSEQ' and 'ALT'.

```
> rownames(info_DF)
```

```
 [1] "LDAF"      "AVGPOST"   "RSQ"       "ERATE"     "THETA"
 [6] "CIEND"     "CIPOS"     "END"       "HOMLEN"    "HOMSEQ"
[11] "SVLEN"     "SVTYPE"    "AC"        "AN"        "AA"
[16] "AF"        "AMR_AF"    "ASN_AF"    "AFR_AF"    "EUR_AF"
[21] "VT"        "SNPSOURCE"
```

```
> param <- ScanVcfParam(fixed="ALT", info="HOMSEQ")
> gr <- readVcfLongForm(fl, "hg19", param)
> head(gr)
```

```
GRanges with 6 ranges and 5 elementMetadata cols:
      seqnames               ranges strand | paramRangeID          ID
         <Rle>            <IRanges>  <Rle> |     <factor> <character>
  [1]       22 [50300078, 50300078]      * |         <NA>   rs7410291
```

```
  [2]       22 [50300086, 50300086]      * |         <NA> rs147922003
  [3]       22 [50300101, 50300101]      * |         <NA> rs114143073
  [4]       22 [50300113, 50300113]      * |         <NA> rs141778433
  [5]       22 [50300166, 50300166]      * |         <NA> rs182170314
  [6]       22 [50300187, 50300187]      * |         <NA> rs115145310
                 REF             ALT                    HOMSEQ
      <DNAStringSet> <DNAStringSet> <CompressedCharacterList>
  [1]              A              G                          NA
  [2]              C              T                          NA
  [3]              G              A                          NA
  [4]              C              T                          NA
  [5]              C              T                          NA
  [6]              G              A                          NA
  ---
  seqlengths:
   22
   NA
```

6.3 Write out VCF files

A VCF file can be written out from data stored in a `VCF` class. Methods to write out from more general structures are in progress.

```
> 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")
> identical(in2, in3)
```

```
[1] TRUE
```

7 References

Wang K, Li M, Hakonarson H, (2010), ANNOVAR: functional annotation of genetic variants from high-throughput sequencing data. Nucleic Acids Research, Vol 38, No. 16, e164.

McLaren W, Pritchard B, RiosD, et. al., (2010), Deriving the consequences of genomic variants with the Ensembl API and SNP Effect Predictor. Bioinformatics, Vol. 26, No. 16, 2069-2070.

SIFT home page : `http://sift.bii.a-star.edu.sg/`

PolyPhen home page : `http://genetics.bwh.harvard.edu/pph2/`

8 Session Information

```
R version 2.15.1 (2012-06-22)
Platform: i386-apple-darwin9.8.0/i386 (32-bit)
```

```
locale:
[1] C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods
[7] base

other attached packages:
 [1] PolyPhen.Hsapiens.dbSNP131_1.0.2
 [2] SIFT.Hsapiens.dbSNP132_1.0.2
 [3] RSQLite_0.11.1
 [4] DBI_0.2-5
 [5] BSgenome.Hsapiens.UCSC.hg19_1.3.17
 [6] BSgenome_1.24.0
 [7] TxDb.Hsapiens.UCSC.hg19.knownGene_2.7.1
 [8] GenomicFeatures_1.8.3
 [9] AnnotationDbi_1.18.1
[10] Biobase_2.16.0
[11] VariantAnnotation_1.2.11
[12] Rsamtools_1.8.6
[13] Biostrings_2.24.1
[14] GenomicRanges_1.8.13
[15] IRanges_1.14.4
[16] BiocGenerics_0.2.0

loaded via a namespace (and not attached):
 [1] Matrix_1.0-7       RCurl_1.91-1       XML_3.9-4
 [4] biomaRt_2.12.0     bitops_1.0-4.1     grid_2.15.1
 [7] lattice_0.20-10    rtracklayer_1.16.3 snpStats_1.6.0
[10] splines_2.15.1     stats4_2.15.1      survival_2.36-14
[13] tools_2.15.1       zlibbioc_1.2.0
```