Importing, Annotating and Filtering Variants with Bioconductor

June 18, 2014

Contents

1 Introduction

Calling variants from high-throughput sequencing data remains a challenge. There are multiple sources of error, including sample prep, sequencing and, perhaps most importantly, alignment. We need to generate diagnostics and remove or flag suspect variants. Once we trust a variant set, the problems shift to interpretation, where the workflow depends on the particulars of the experiment and its driving hypotheses. Often, we are interested in functional consequences and disease associations.

Both variant filtering and interpretation rely on annotation of the variants with information such as the genomic context and whether the variant represents a change in a protein. Exploratory analysis of these annotations may suggest additional filtering criteria, or at least guide our interpretation of the data.

We will demonstrate variant annotation and filtering using the Bioconductor VariantAnnotation and VariantFiltering packages, along with an ensemble of supporting packages.

2 Dataset

To demonstrate the variant annotation features in Bioconductor, we will analyze the genotypes of the human NA12878 cell line, the mother of the CEU HapMap trio. It seems that NA12878 has been sequenced and genotyped by more organizations than any other individual. Our analysis will focus on the variants published by the Illumina Platinum Genomes project. These data

were aligned with BWA and genotyped with GATK. For practical reasons, the data have been subset to chr20.

3 Importing variants from VCF

3.1 VCF: Variant Call Format

The Variant Call Format (VCF) is the standard file format for storing variant calls. It is a complex and flexible format, one applicable to a large number of use cases but convenient for none. Every VCF file consists of two parts: a header describing the format and provenance of the file, and the actual variant records. Each variant record pertains to one or more alternate alleles at a particular position in the genome. Each alternate allele might be a single or multi-base substitution, a short indel or a complex structural rearrangement. The record contains information at four different levels: the position, a particular alternate allele, a particular sample, and a particular combination of alternate allele and sample. This information will usually but not necessarily contain the genotype (wildtype, het, or hom) and some indicator of confidence in the genotype.

At a lower level, each VCF record consists of the following components:

`CHROM` The chromosome on which the variant is located,

`POS` The variant (start) position on `CHROM`,

`ID` A string identifier, such as the dbSNP ID,

`REF` The reference allele,

`ALT` The alternate allele,

`QUAL` Some notion of quality for entire record,

`FILTER` A list of filters that the variant failed to pass,

`INFO` A list of arbitrary fields describing the record or a specific alt allele,

`GENO` A set of columns, one per sample, each a list of sample-specific fields, and each field may itself be a list, perhaps with one value per alt.

3.2 Previewing a VCF file

To begin, we obtain the path to the demonstration VCF file:

```
library(CSAMA2014RangesAnnotationLab)
vcf.file <- NA12878_pg.chr20.vcf.bgz
```

The VariantAnnotation package in Bioconductor provides facilities for importing VCF data into R. Whenever working with a new VCF file, it is a good idea to explore its header, which describes the contents of the file. This is useful for knowing what to expect, and it allows us to optimize import by restricting the operation to the components of interest (see the next section).

```
header <- scanVcfHeader(vcf.file)
header
```

```
class: VCFHeader
samples(1): NA12878
meta(7): fileformat ApplyRecalibration ... source fileDate
fixed(1): FILTER
info(22): AC AF ... culprit set
geno(8): GT GQX ... PL VF
```

The above output is how the loaded *VCFHeader* object displays itself when printed at the R console. By convention, many objects in the Bioconductor sequence analysis infrastructure print a single line per component of the data structure, and line label indicates the name of the accessor for retrieving that component. For example, entering

```
samples(header)
```

```
[1] "NA12878"
```

returns the sample names, as in the previous output.

Our VCF file contains information on one sample: NA12878, the mother in the CEU trio. Here are the other components described in the header:

`meta` Arbitrary metadata values at the file scope,

`fixed` Essentially a list of filter descriptions,

`info` Descriptions of the per record/alt fields,

`geno` Descriptions of sample-level fields.

The most important information is usually the sample-specific values, which typically include the genotype, as in this case:

```
geno(header)
```

```
DataFrame with 8 rows and 3 columns
         Number        Type
    <character> <character>
GT            1      String
GQX           1     Integer
AD            .     Integer
DP            1     Integer
GQ            1       Float
MQ            1     Integer
PL            G     Integer
VF            1       Float
Description
<character>
GT  Genotype
GQX Minimum of {Genotype quality assuming variant, non-variant}
AD  Allelic depths for the ref and alt alleles in the order listed
DP  Approximate read depth (reads with MQ=255 or with bad mates are filtered)
GQ  Genotype Quality
MQ  RMS Mapping Quality
PL  Normalized, Phred-scaled likelihoods for genotypes...
VF  Variant Frequency, the ratio of the sum of the called variant depth...
```

We also notice something interesting in the `INFO` header:

```
info(header)["END",]
```

```
DataFrame with 1 row and 3 columns
         Number        Type                   Description
    <character> <character>                   <character>
END           1     Integer End position of the region...
```

The presence of the `END` `INFO` field indicates that we are actually dealing with a special type of VCF called a gVCF, where "g" stands for "genomic"; more later.

3.2.1 Exercises

1. By convention, which accessor would you use to retrieve the `meta` component of the *VCFHeader* object?

2. What is the meaning of the `AD` and `DP` fields in the `geno()` component?

3.3 VCF import

Now that we understand the contents of the file, we can load the VCF data into R using `readVcf()`:

```
vcf <- readVcf(vcf.file, genome="hg19")
```

We need to pass a genome identifier as the `genome` argument for the sake of tracking provenance and ensuring data integrity.

Often, we want to avoid loading an entire VCF file. Perhaps we are only interested in a specific region, or we are iterating over the data to manage resource consumption. Recall that our file has been restricted to those variants on chr20. Without that restriction, the file would be 40X larger. The restricted form consumes this much memory:

```
print(object.size(vcf), unit="auto")
```

```
300 Mb
```

But the raw file would have consumed:

```
print(object.size(vcf) * 40L, unit="auto")
```

```
11.7 Gb
```

which is beyond the memory capacity of most laptops.

In order to restrict by genomic range, we pass the range to `readVcf()` via a *ScanVcfParam* object. The `readVcf()` function supports a large number of parameters, so the designers have encapsulated them in a special type of object for ease of management. *ScanVcfParam* is sometimes inconvenient for simple tasks, but it soon proves its worth, especially when writing reusable wrappers around `readVcf()` and related functions.

Although we have already subset the file to chr20, we repeat it here. The first step is to figure out a representative range for chr20. For that, we defer to the canonical representation of the hg19 genome in Bioconductor: the BSgenome.Hsapiens.UCSC.hg19 package. Calling `seqinfo()` on the

genome object returns a *Seqinfo* object, which is a special object for indicating the names, lengths and other attributes of chromosomes/contigs in a genome.

```
library(BSgenome.Hsapiens.UCSC.hg19)
seqinfo(Hsapiens)
```

```
Seqinfo of length 93
seqnames       seqlengths isCircular genome
chr1            249250621      FALSE   hg19
chr2            243199373      FALSE   hg19
chr3            198022430      FALSE   hg19
chr4            191154276      FALSE   hg19
chr5            180915260      FALSE   hg19
   ...
```

We can coerce the information for chr20 to a *GRanges* object, which is the primary object for representing genomic ranges in Bioconductor. A *GRanges* is a vector of ranges, where each range is described by its chromosome, start and end on the chromosome, and strand. The user can add arbitrary columns to a metadata table with one row per range. In this case, we have a bare range representing the entire length of chr20:

```
ranges.chr20 <- as(seqinfo(Hsapiens)["chr20"], "GRanges")
ranges.chr20
```

```
GRanges with 1 range and 0 metadata columns:
        seqnames        ranges strand
           <Rle>     <IRanges>  <Rle>
  chr20    chr20 [1, 63025520]      *
  ---
  seqlengths:
      chr20
   63025520
```

We are finally ready perform the restricted query with `readVcf`:

```
param <- ScanVcfParam(which=ranges.chr20)
vcf.chr20 <- readVcf(vcf.file, genome="hg19", param=param)
```

3.3.1 Exercises

1. How would we have restricted to chr19 instead of chr20?

2. Let us assume that we are not interested in any of the `info()` fields in the file. If we exclude them from the import operation, we can save valuable time and memory. See `?ScanVcfParam` and determine how to do this.

4 Filtering and reducing VCF data

4.1 Manipulating a *VCF* object

The `readVcf()` function returns a *VCF* object, a derivative of *SummarizedExperiment* that fully represents the complexity of the VCF file. This means that the *VCF* object needs to be quite general and complex, and it is not tailored for any particular use case. It is up to the user to reduce the *VCF* into an appropriate data structure for the task at hand. Before we can do that for our use case, we need to understand the structure of the object.

As with the header, the textual display of *VCF* is quite descriptive and indicates how we can retrieve the various subcomponents:

```
vcf
```

```
class: CollapsedVCF
dim: 1091548 1
rowData(vcf):
  GRanges with 5 metadata columns: paramRangeID, REF, ALT, QUAL, FILTER
info(vcf):
  DataFrame with 22 columns: AC, AF, AN, DP, QD, BLOCKAVG_min30p3a, BaseQRan...
info(header(vcf)):
                     Number Type
   AC                A      Integer
   AF                A      Float
   AN                1      Integer
   DP                1      Integer
   QD                1      Float
   BLOCKAVG_min30p3a 0      Flag
   BaseQRankSum      1      Float
   DS                0      Flag
   Dels              1      Float
```

```
   END                1      Integer
   FS                 1      Float
   HRun               1      Integer
   HaplotypeScore     1      Float
   InbreedingCoeff    1      Float
   MQ                 1      Float
   MQ0                1      Integer
   MQRankSum          1      Float
   ReadPosRankSum     1      Float
   SB                 1      Float
   VQSLOD             1      Float
   culprit            1      String
   set                1      String
                     Description
   AC                Allele count in genotypes, for each ALT allele,...
   AF                Allele Frequency, for each ALT allele, in the s...
   AN                Total number of alleles in called genotypes
   DP                Approximate read depth; some reads may have bee...
   QD                Variant Confidence/Quality by Depth
   BLOCKAVG_min30p3a Non-variant site block. All sites in a block ar...
   BaseQRankSum      Z-score from Wilcoxon rank sum test of Alt Vs. ...
   DS                Were any of the samples downsampled?
   Dels              Fraction of Reads Containing Spanning Deletions
   END               End position of the region described in this re...
   FS                Phred-scaled p-value using Fisher's exact test ...
   HRun              Largest Contiguous Homopolymer Run of Variant A...
   HaplotypeScore    Consistency of the site with at most two segreg...
   InbreedingCoeff   Inbreeding coefficient as estimated from the ge...
   MQ                RMS Mapping Quality
   MQ0               Total Mapping Quality Zero Reads
   MQRankSum         Z-score From Wilcoxon rank sum test of Alt vs. ...
   ReadPosRankSum    Z-score from Wilcoxon rank sum test of Alt vs. ...
   SB                Strand Bias
   VQSLOD            Log odds ratio of being a true variant versus b...
   culprit           The annotation which was the worst performing i...
   set               Source VCF for the merged record in CombineVari...
geno(vcf):
  SimpleList of length 8: GT, GQX, AD, DP, GQ, MQ, PL, VF
geno(header(vcf)):
       Number Type    Description
```

```
GT  1      String  Genotype
GQX 1      Integer Minimum of {Genotype quality assuming variant p...
AD  .      Integer Allelic depths for the ref and alt alleles in t...
DP  1      Integer Approximate read depth (reads with MQ=255 or wi...
GQ  1      Float   Genotype Quality
MQ  1      Integer RMS Mapping Quality
PL  G      Integer Normalized, Phred-scaled likelihoods for genoty...
VF  1      Float   Variant Frequency, the ratio of the sum of the ...
```

The `rowData(vcf)` is a *GRanges* object indicating the location of each variant. The metadata columns of the *GRanges* hold the fixed VCF columns, including `REF`, `ALT`, `QUAL` and `FILTER`.

```
rowData(vcf)
```

```
GRanges with 1091548 ranges and 5 metadata columns:
                      seqnames              ranges strand   | paramRangeID
                         <Rle>           <IRanges>  <Rle>   |     <factor>
     chr20:60001_G/.    chr20       [60001, 60001]      *   |         <NA>
     chr20:60048_T/.    chr20       [60048, 60048]      *   |         <NA>
     chr20:60230_A/.    chr20       [60230, 60230]      *   |         <NA>
     chr20:60447_G/.    chr20       [60447, 60447]      *   |         <NA>
     chr20:60526_C/.    chr20       [60526, 60526]      *   |         <NA>
         ...
  chr20:62965490_T/.    chr20 [62965490, 62965490]      *   |         <NA>
  chr20:62965507_G/.    chr20 [62965507, 62965507]      *   |         <NA>
  chr20:62965509_A/.    chr20 [62965509, 62965509]      *   |         <NA>
  chr20:62965511_T/.    chr20 [62965511, 62965511]      *   |         <NA>
  chr20:62965514_G/.    chr20 [62965514, 62965514]      *   |         <NA>
                            REF                 ALT      QUAL        FILTER
                    <DNAStringSet> <DNAStringSetList> <numeric>  <character>
     chr20:60001_G/.              G                          <NA>         PASS
     chr20:60048_T/.              T                          <NA>         PASS
     chr20:60230_A/.              A                          <NA>         PASS
     chr20:60447_G/.              G                          <NA>         PASS
     chr20:60526_C/.              C                          <NA>         PASS
                 ...            ...                 ...       ...          ...
  chr20:62965490_T/.              T                          <NA>       LowGQX
  chr20:62965507_G/.              G                          <NA>       LowGQX
  chr20:62965509_A/.              A                          <NA> LowGQX;LowMQ
  chr20:62965511_T/.              T                          <NA> LowGQX;LowMQ
```

```
chr20:62965514_G/.              G                         <NA>       LowGQX
---
seqlengths:
     chr20      chr1     chr10     chr11 ...      chrM      chrX      chrY
  63025520 249250621 135534747 135006516 ...     16571 155270560  59373566
```

Several important attributes are located in the sample-specific `geno()` fields, which include the genotype (`GT`):

```
head(geno(vcf)$GT)
```

```
                NA12878
chr20:60001_G/. "0/0"
chr20:60048_T/. "0/0"
chr20:60230_A/. "0/0"
chr20:60447_G/. "0/0"
chr20:60526_C/. "0/0"
chr20:60641_A/. "0/0"
```

We can see that this particular VCF includes wildtype calls, while we are only interested in the variants, so we subset the object:

```
variants <- vcf[geno(vcf)$GT[,1] != "0/0",]
```

Note that we use matrix-style indexing, because *VCF* models the data as a variant by sample matrix.

Of the fixed columns, the most important is the `ALT` column, which stores the alternate allele(s) for each record. We can access it directly:

```
alt(variants)
```

```
DNAStringSetList of length 150402
[[1]] T
[[2]] CT
[[3]] C
[[4]] C
[[5]] T
[[6]] C
[[7]] T
[[8]] A
[[9]] A
[[10]] A
...
<150392 more elements>
```

From this output, we first notice that we have more than just SNVs in our alt alleles. For simplicity, we restrict to SNVs:

```
snvs <- variants[isSNV(variants),]
```

The reader may have expected the return value of `alt()` to be a *character* vector; however, there may be **multiple** alts per record, which means `ALT` is stored as the likely unfamiliar *CharacterList*. While the collapsed *List* structure may be appropriate for communicating position-oriented data, such as population-level polymorphisms, for our use case it is usually easier to reason on the data in the expanded form, where there is one alt per position and positions may be repeated. Thus, we `expand()` our *VCF* object

```
snvs <- expand(snvs)
head(alt(snvs))
```

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

and end up with a flat *DNAStringSet*, essentially a specialized character vector for DNA sequences.

4.1.1 Exercises

1. Obtain the `AD` and/or `DP` components of the `snvs` object.
2. One could calculate the allele fraction from the `AD` and `DP` components. However, some variant callers, including GATK, filter the `DP` component differently from the counts in `AD`, so the two values are incompatible. Luckily, this file contains the alt frequency as a special genotype field; which one is it?
3. How much memory have we saved through this filtering? Hint: see the usage of `object.size()` in the previous section.

4.2 Filtering a VCF file

Typically, we want to process an entire genome; there is nothing particularly interesting about chr20. We calculated that importing the variants for the entire genome would consume roughly 9GB of memory, which is infeasible. However, by filtering to the SNVs we have shrunk the object size from 230 MB to 39 MB (the answer to the last exercise). If we assumed the same reduction over the entire genome, we would only need about 1.5 GB to load all of the SNVs in the genome. But how can we do that if we cannot load the data in the first place? Do we need to resort to the UNIX shell, sacrifice reproducibility and probably reinvent the wheel, only to discover a bug the night before we submit our paper? Thankfully, no.

The solution to our quandary is the `filterVcf` function, which iterates over the data behind the scenes, apply a user-specified filter to each chunk before writing it back to disk. We will use `filterVcf` to perform the same filtering as in the previous section: restricting to SNVs.

The function accepts two types of filters:

`prefilters` Applied to the raw text of the file, and

`filters` Applied to the data parsed as a *VCF* file.

To avoid the overhead of parsing, we want to do as much work as is safe in the prefilters, but it is important to realize the risk inherent in implementing a prefilter: we are essentially writing our own VCF parser.

In this case, we are probably safe with a prefilter that excludes the 0/0 genotypes:

```
prefilters <- FilterRules(list(restrictToVariants=function(text) {
  !grepl("0/0", text, fixed=TRUE)
}))
```

In the above, we implement our filter as a simple function and wrap it in a *FilterRules* object, which `filterVcf` expects. Our function will be called on each chunk during the iteration.

We also need to restrict to SNVs, as opposed to indels or other changes. This is much more complicated and so we need to operate on a parsed *VCF* object:

```
filters <- FilterRules(list(restrictToSNVs=function(vcf) isSNV(vcf)))
```

Finally, we can invoke `filterVcf`:

```
filterVcf(vcf.file, genome="hg19", "snvs.vcf", index=TRUE,
          prefilters=prefilters, filters=filters,
          param=ScanVcfParam(info=NA))
```

```
 starting prefilter
prefiltering 1091548 records
prefiltered to /tmp/RtmpdU6cy6/file15fd729fc273
prefilter compressing and indexing '/tmp/RtmpdU6cy6/file15fd729fc273'
starting filter
filtering 150402 records
completed filtering
compressing and indexing 'snvs.vcf'
```

Above, we have written a new indexed and compressed VCF file, "snvs.vcf.gz" that contains only the SNVs. We also exclude the relatively uninteresting `INFO` fields (answer to a previous exercise) for a further reduction.

4.2.1 Exercises

1. What if we wanted to create a separate file with all of the indels? Hint: see `?isIndel`.

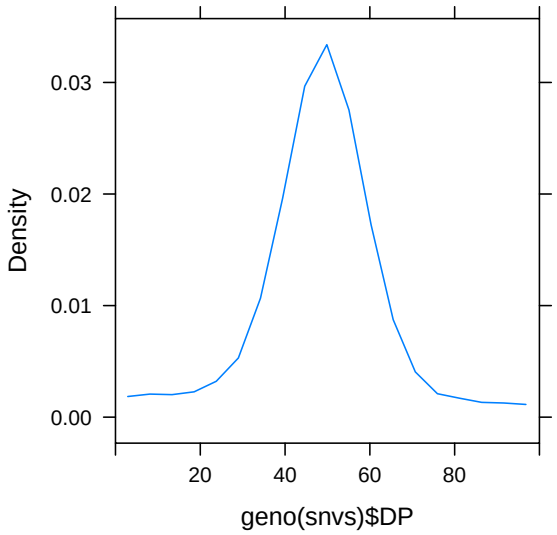
5 Diagnosing variants

5.1 QC of variant frequency and coverage

Upon import of an unfamiliar set of variants, we recommend generating and analyzing some diagnostic annotations to ascertain some notion of the quality of the data.

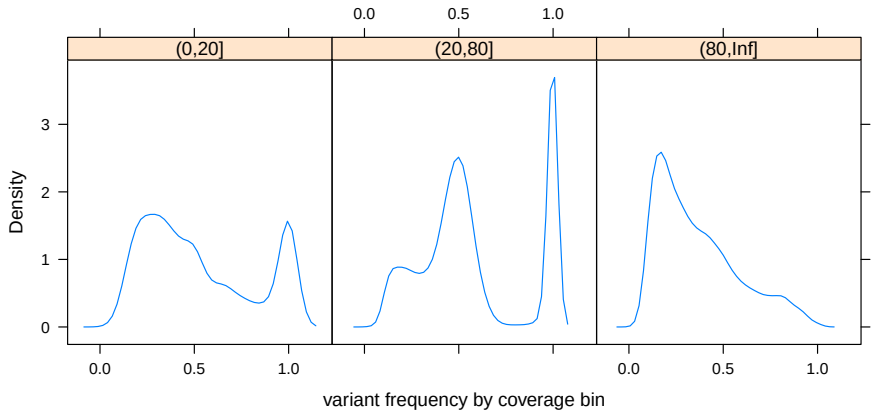
A useful diagnostic is the association between variant frequency and coverage. Given the diploid assumption, we expect variant frequencies to concentrate around 0.5 or 1.0. Any deviation from this is an indicator of artifact. For example, we suspect that outlying coverage values will present aberrant frequencies. To demonstrate, we first plot our coverage distribution to get a feel for the appropriate cutoffs:

```
library(lattice)
densityplot(~ geno(snvs)$DP, xlim=c(0, 2*median(geno(snvs)$DP, na.rm=TRUE)),
            plot.points=FALSE)
```



From this distribution, it seems appropriate to cut like this and plot the relationship to variant frequency, which we calculate from `AD` and `DP` (which are much more standard than `VF`):

```
rowData(snvs)$coverage.bin <- cut(geno(snvs)$DP, c(0, 20, 80, Inf))
rowData(snvs)$variant.freq <- geno(snvs)$AD[,,2]/geno(snvs)$DP[,1]
densityplot(~ variant.freq | coverage.bin,
            as.data.frame(rowData(snvs)),
            plot.points=FALSE, layout=c(3,1),
            xlab="variant frequency by coverage bin")
```



We notice some interesting trends. First, there is a mode, even in the (20,80] coverage range, that is at a lower frequency than the het mode. This mode is dominant at extremely low and high coverage. The corresponding variant calls are very likely artifacts. In the high coverage bin, the hom variants seem to have shifted to above 75% frequency.

Since we can assume that Illumina is good at sequencing, these problems are likely due to alignment. Alignment is a function of the read sequence and the reference sequence. Indels in the reads will give the aligner trouble. Genomic context, such as homopolymers and paralogs, are also problematic and tend to indicate regions where the invidual genome will differ structurally from the reference.

5.2 Applying included filters

Some useful annotations are already present in the VCF file. For example, we can summarize the `filt()` component, which is a semi-colon separated list of filter codes generated by the variant caller, where "PASS" indicates that all QC filters were passed. First, we list the descriptions of the filter codes from the header:

```
fixed(header(snvs))$FILTER
```

```
DataFrame with 9 rows and 1 column
Description
<character>
LowGQX              Locus GQX is less than 30.0000 or not present
LowQD               Locus QD is less than 2.0000
LowMQ               Site MQ is less than 20.0000
IndelConflict       Locus is in region with conflicting indel calls.
MaxDepth            Site depth is greater than 3.0x the mean chromosome...
SiteConflict        Site genotype conflicts with proximal indel call...
...                 ...
```

These filters point to common issues with SNV calls, in particular conflicts with indels (`IndelConflict` and `SiteConflict`) and abnormally high coverage (`MaxDepth`).

Next, a simple filter summary:

```
table(unlist(strsplit(filt(snvs), ";", fixed=TRUE)))
```

```
                      IndelConflict                                LowGQX
```

```
                                    35                                    11020
                                 LowMQ                                  LowQD
                                  1055                                  19583
                              MaxDepth                                   PASS
                                    25                                  68626
                           SiteConflict  TruthSensitivityTranche99.00to99.90
                                   785                                  17174
TruthSensitivityTranche99.90to100.00
                                 28793
```

It looks like a large number of variants have `FILT` values other than "PASS". For the purposes of this tutorial, we will restrict to the "PASS" variants.

```
snvs <- snvs[grep("PASS", filt(snvs), fixed=TRUE),]
```

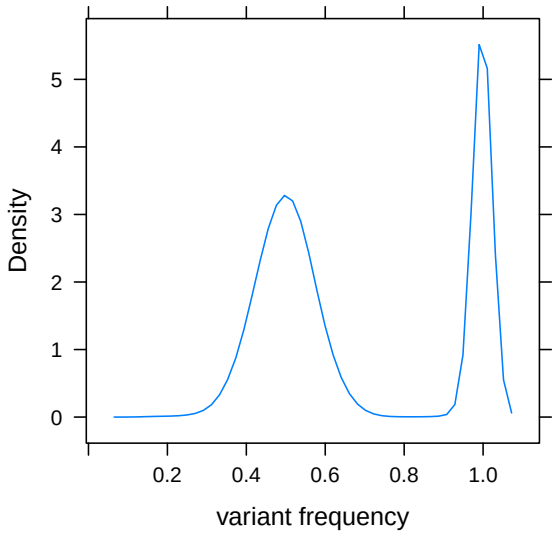
After that filter, there are very few positions with outlying coverage values:

```
table(rowData(snvs)$coverage.bin)
```

```
(0,20]  (20,80] (80,Inf]
    63    68528       35
```

And the frequency distribution matches our expectations:

```
densityplot(~ variant.freq, as.data.frame(rowData(snvs)),
            plot.points=FALSE,
            xlab="variant frequency")
```



5.2.1 Exercises

1. Use the patterns presented above to determine which specific filters were most responsible for removing the group of low frequency variants.

5.3 Indel proximity

We look first at indel proximity. The `FILT` component of the VCF already has some filters related to indels, but this will not be present in general, so we will repeat the analysis here. First, we find the indels:

```
indels <- variants[isIndel(variants)]
```

Then, we generate a window around the each indel, say 10bp on either side.

```
indel.windows <- rowData(indels) + 10
```

> *Sponsored message from IRanges, Inc:*
>
> The usage of `+` above may be surprising. It adds 10bp to either side of the indels in the `rowData`, which we recall is a *GRanges* object. There are many such operations defined for range objects in Bioconductor. The `+` convenience is based on the `resize`

operation, an example of an intra-range operation, because each range is treated separately. See `?resize` for the list of intra-range functions. The inter-range-operations consider all of the ranges at once and are mostly ways to summarize ranges. See `¿inter-range-methods‘` for a list.

The next step is to find which SNVs overlap an indel window:

```
rowData(snvs)$near.indel <- snvs %over% indel.windows
```

Sponsored message from IRanges, Inc:

The usage of `%over%` is one of the convenience functions in the overlap detection framework implemented for *SummarizedExperiment*, *GRanges* and other range objects in Bioconductor. The `%over%` function returns `TRUE` or `FALSE` for each range in its left argument, depending on whether there is any overlap with a range in its right argument. It is just the tip of the iceberg: see `?findOverlaps`.

As with the coverage bins and variant frequency, we have taken advantage of the metadata support in *GRanges* to store the indel overlap status along with the SNVs. It is a good idea to do this whenever sensible, because it organizes data in a way that promotes data integrity and keeps the workspace clean.

A simple summary reveals that very few SNVs are near an indel:

```
xtabs(~ near.indel, mcols(rowData(snvs)))
```

```
near.indel
FALSE  TRUE
68415   211
```

5.4 Homopolymer overlap

Variant calling is often problematic in regions of homopolymers. They are difficult regions, both for alignment and interpretation.

To determine which variants overlap homopolymers, we need find the homopolymers in the reference. First, we load the reference sequence for chr20:

```
chr20.sequence <- getSeq(Hsapiens, "chr20")
```

Then, we convert the DNA sequence into a run-length encoding, which Bioconductor represents with the *Rle* class.

```
chr20.hp <- ranges(Rle(as.raw(chr20.sequence)))
```

> *Sponsored message from IRanges, Inc:*
>
> An *Rle* object acts like an ordinary vector in R, but it is internally represented as a vector of runs of identical values, each with a value and length. This representation is more efficient, both in terms of run-time and memory consumption, for data with long runs of identical values, including sparse data with long runs of zeros. We often use it to represent coverage, especially when the coverage is sparse due to enrichment. In this case, we are interested in the runs themselves and extract the corresponding ranges with `ranges()`.

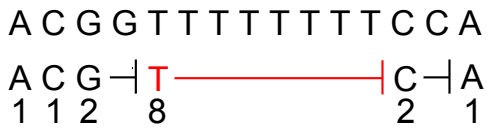


Figure 1: Representing homopolymers in a sequence (top) with a run-length encoding (bottom).

We somewhat arbitrarily decide that homopolymers become interesting after they exceed the length of 6:

```
chr20.hp <- chr20.hp[width(chr20.hp) > 6L]
```

Finally, we find the variants that overlap a homopolymer and summarize the counts:

```
rowData(snvs)$over.hp <- ranges(snvs) %over% chr20.hp
xtabs(~ over.hp, mcols(rowData(snvs)))
```

```
over.hp
FALSE  TRUE
67874   752
```

5.5 Self-chain scores

The UCSC genome browser has published a track that indicates whether a region shares similarity with another region in the genome. Related positions are termed "self-chained" and tend to be problematic to align due to ambiguity. Due to differences between the individual genome and the reference, aligners often pile up reads in one of the ambiguous regions. This leads to non-diploid frequencies and interpretation challenges.

We have included a *GRanges* of the self-chained regions for chr20 in the tutorial package:

```
data(selfChains)
```

5.5.1 Exercises

1. Find the overlap between the `snvs` object and the `selfChains`. Store the result on the `rowData()`. Summarize it somehow.

6 Interpreting variants

Having generated sufficient diagnostics, we turn our attention to annotations aimed at determining the consequences of a variant. The VariantAnnotation package provides several convenient functions for the common operations.

6.1 Genomic context

The `locateVariants` function annotates variants with their genomic context. It requires the variants, the gene models and a variant category:

```
gene.models <- TxDb.Hsapiens.UCSC.hg19.knownGene
locations <- locateVariants(snvs, gene.models, CodingVariants())
```

The return value, `locations`, is a *GRanges* with these columns:

```
colnames(mcols(locations))
```

```
[1] "LOCATION"  "QUERYID"   "TXID"      "CDSID"     "GENEID"    "PRECEDEID"
[7] "FOLLOWID"
```

Most of them identify the overlapping genomic feature for any variant that overlaps the coding region. See `?locateVariants` for details.

Notice that the `locations` object is only annotating a subset of the input variants: only those that are in coding regions. If we want to map these

annotations back into our input, substituting NA for the non-coding variants, we need to take advantage of the QUERYID column:

```
rowData(snvs)$coding.tx <- NA_integer_
rowData(snvs)$coding.tx[locations$QUERYID] <- locations$TXID
```

When interpreting hits at the gene level, we often want to see the gene symbols. This mapping is provided by the AnnotationDbi infrastructure:

```
syms <- unlist(mget(locations$GENEID[!is.na(locations$GENEID)],
                    org.Hs.egSYMBOL,
                    ifnotfound=NA))
locations$SYMBOL[!is.na(locations$GENEID)] <- syms
```

6.1.1 Exercises

1. Merge the coding$SYMBOL back into the original snvs object.
2. We found the variants that overlap a coding region; how would we find those inside a promoter?

6.2 Coding consequences

The predictCoding function predicts coding consequences. We need to pass it the variants, the gene models, and the genomic sequence:

```
coding <- predictCoding(snvs, gene.models, Hsapiens)
```

The returned object, coding, is a *GRanges* object with a number of metadata columns:

```
colnames(mcols(coding))
```

```
 [1] "paramRangeID" "coverage.bin" "variant.freq" "near.indel"   "over.hp"
 [6] "coding.tx"    "REF"          "ALT"          "QUAL"         "FILTER"
[11] "varAllele"    "CDSLOC"       "PROTEINLOC"   "QUERYID"      "TXID"
[16] "CDSID"        "GENEID"       "CONSEQUENCE"  "REFCODON"     "VARCODON"
[21] "REFAA"        "VARAA"
```

These include columns from the input *VCF*, the affected gene/transcript/CDS, locations relative to the CDS and protein, and the consequences, including the codon and amino acid changes. See ?predictCoding for a detailed description of the columns.

As an example summary, we tabulate the consequence codes:

```
table(coding$CONSEQUENCE)
```

```
nonsense nonsynonymous    synonymous
       1           502           809
```

6.2.1 Exercises

1. Cross tabulate the ref and alt amino acids.
2. Find the variant that occurred in the SOX12 gene.

6.3 Genetic disorders

The VariantFiltering package annotates and filters variants from the perspective of genetic disorders among related or unrelated individuals. Within related individuals, it is capable of modeling inheritance. It integrates annotations from a number of sources, including VariantAnnotation, Polyphen, SIFT, and UCSC phastCon conservation scores. We will apply the package to find autosomal recessive homozygous variants within the CEU trio. Most of this example was taken directly from the package vignette.

The first step is to construct a parameter object for the analysis, given the CEU trio VCF file (from the 1000 Genomes) and a PED file representing the pedigree:

```
CEUvcf <- file.path(system.file("extdata", package="VariantFiltering"),
                    "CEUtrio.vcf.bgz")
CEUped <- file.path(system.file("extdata", package="VariantFiltering"),
                    "CEUtrio.ped")
param <- VariantFilteringParam(vcfFilenames=CEUvcf, pedFilename=CEUped)
```

```
Error: could not find function "VariantFilteringParam"
```

There are many analyses available in the package. Here we perform the autosomal recessive analysis.

```
reHo <- autosomalRecessiveHomozygous(param)
reHo
```

We can filter the variants interactively using a web app:

```
aim <- reportVariants(reHo)
```

There are various possible operations on the result object, such as restricting the populations used for MAF filtering:

```
maxMAF(reHo) <- 0.05
MAFmask <- MAFpop(reHo)
MAFmask
```

```
MAFpop(reHo) <- !MAFmask
MAFpop(reHo, "ASN_AFKG") <- TRUE
MAFpop(reHo)
```

```
minCRYP5ss(reHo) <- 0
reHo
```

After filtering, we end up with a single variant:

```
filteredVariants(reHo)
```

7 Comparing variant sets

It is often of interest to compare two or more sets of variants. For our use case, we are interested in checking concordance between the Platinum genome calls and those published with the GATK paper for the same individual (NA12878).

7.1 Exercise: Loading the Broad/GATK callset

Please obtain the file "CEUTrio.HiSeq.WGS.b37.bestPractices.phased.b37.vcf.gz" and its associated index from the ScalableGenomicsTutorial directory on the USB stick. It represents the best-practice variant calls from the GATK paper, made available as part of the GATK bundle. It contains genotypes for all three members of the CEU trio.

As an exercise,

1. Load the variants on chr20 for only NA12878 and
2. Filter them to the SNVs, where the genotype is not "0/0".
3. Note that the Broad calls use a different chromosome naming convention ("NCBI" instead of "UCSC"), so it is necessary to convert the style of the `ranges.chr20` when forming the query, and the style of the loaded object must be converted in the opposite direction to match those of the Illumina calls. See `?seqlevelsStyle`.

4. Since the mitochondrial chromosome differs between NCBI and UCSC, it is best to use `dropSeqlevels` to remove "chrM" from `snvs`.

7.2 Intersecting variant sets

To check whether two variant calls are identical, we need to check the chromosome, start, end (for non-SNVs) and alt allele. When finding exact matches between *GRanges* objects, the alt allele is not considered. Thus, we need a special derivative of *GRanges* called *VRanges*. Below, we coerce each *VCF* object to a *VRanges* and then check which sites in the Illumina calls match a call in the Broad set.

```
broad.vr <- as(broad.snvs, "VRanges")
illumina.vr <- as(snvs, "VRanges")
illumina.vr$in.broad <- illumina.vr %in% broad.vr
mean(illumina.vr$in.broad)
```

```
[1] 0.9905429
```

We find that 99% of the Illumina variants were also found by the Broad.

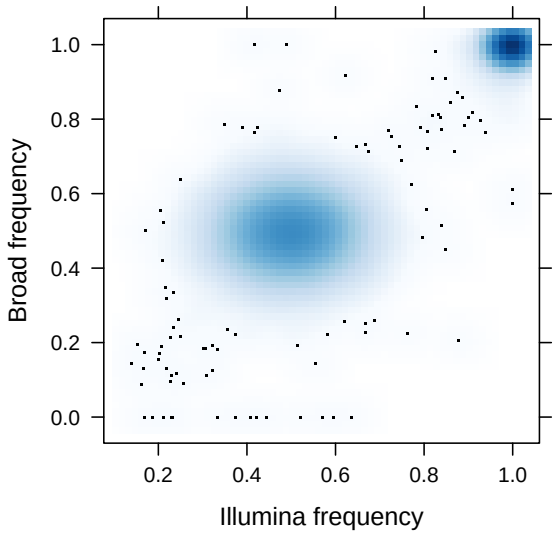
To compare the variant frequencies, we can merge the variant frequencies from the Broad set into the Illumina set:

```
illumina.vr$broad.freq <- altFraction(broad.vr)[match(illumina.vr, broad.vr)]
```

Note the use of the `altFraction` convenience function that operates on a *VRanges*. We can do this with *VRanges* since the coercion to *VRanges* asserts that the `AD` and `DP` fields conform to the informal conventions of GATK and related tools.

Now we can make a scatterplot:

```
xyplot(broad.freq ~ altFraction(illumina.vr),
       as.data.frame(illumina.vr),
       panel=panel.smoothScatter,
       xlab="Illumina frequency", ylab="Broad frequency")
```



We find that the frequencies agree, for the most part, although there is a number of cases with zero frequencies in the Broad set, but non-zero frequencies from Illumina.

7.2.1 Exercises

1. See `?softFilterMatrix` and `?called` to subset `broad.vr` to the variants that passed all filters.

2. Make a density plot of the variant frequencies from `broad.vr`.

3. What percentage of the Broad calls were not called by Illumina?

7.3 Manipulating gVCF runs

This VCF file is a special type of VCF file for whole genome genotyping called gVCF, where the "g" stands for genomic. It contains summarized runs for non-variant regions, which is more efficient than storing a record for every position in the genome. These regions may be considered wildtype or no-call, depending on the indicated quality of the 0/0 genotype.

The combination of `CHROM`, `POS` and the `END INFO` field specify the range of the run, and the primary value of interest is the genotype quality (`GQX`). We can manipulate the *VCF* object so that the ranges represent the runs:

```
runs <- vcf[!is.na(info(vcf)$END),]
end(rowData(runs)) <- info(runs)$END
```

Then we might check how much of chr20 is covered by a run:

```
sum(width(runs)) / seqlengths(runs)["chr20"]
```

```
    chr20
0.9403234
```

7.3.1 Exercises

1. Assuming that a "PASS" value for `filt(runs)` indicates wildtype and no-call otherwise, what percentage of the genome was callable?

2. For the unique Broad variants found in a previous exercise, how many of them were called wildtype vs. no-call by Illumina?