Variant Calling with R/Bioconductor

Michael Lawrence

July 28, 2014

Outline

Introduction

Calling variants with VariantTools

Diagnosing variant calls

Importing and manipulating VCF data

Comparing variant sets

Interpreting variants

Outline

Introduction

Calling variants with VariantTools

Diagnosing variant calls

Importing and manipulating VCF data

Comparing variant sets

Interpreting variants

Variant calls

Definition

- A variant call is a conclusion that there is a nucleotide difference vs. some reference at a given position in an individual genome or transcriptome,
- Usually accompanied by an estimate of variant frequency and some measure of confidence.

Use cases

DNA-seq: variants

- Genetic associations with disease
- Mutations in cancer
- Characterizing heterogeneous cell populations

RNA-seq: allele-specific expression

- Allelic imbalance, often differential
- Association with isoform usage (splicing QTLs)
- RNA editing (allele absent from genome)

ChIP-seq: allele-specific binding

Variant calls are more general than genotypes

Genotypes make additional assumptions

- A genotype identifies the set of alleles present at each locus.
- The number of alleles (the ploidy) is decided and fixed.
- Most genotyping algorithms output genotypes directly, under a blind diploid assumption and special consideration of SNPs and haplotypes.

Those assumptions are not valid in general

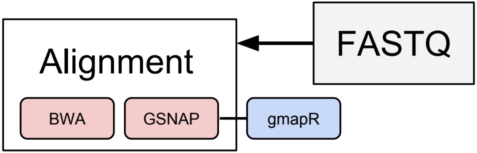
- Non-genomic input (RNA-seq) does not represent a genotype.
- Cancer genome samples are subject to:
  - Copy number changes
  - Tumor heterogeneity
  - Tumor/normal contamination

  So there is a mixture of potentially non-diploid genotypes, and there is no interpretable genotype for the sample

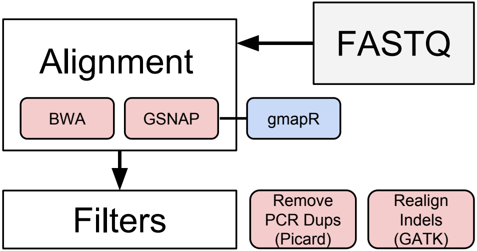
Typical variant calling workflow

FASTQ

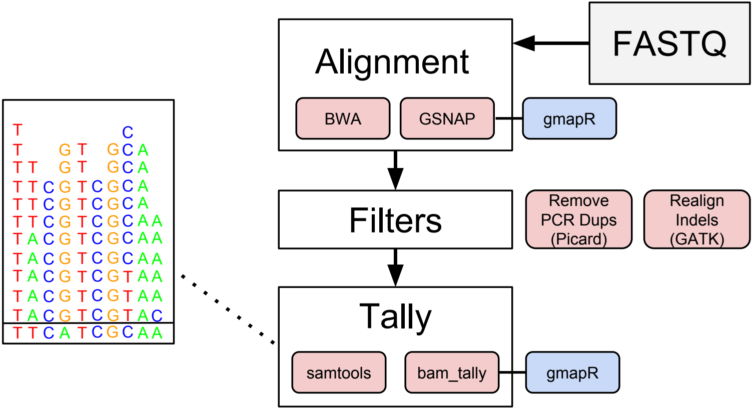
Typical variant calling workflow



Typical variant calling workflow



Typical variant calling workflow



```
T                   C
T       G T     G C A
T T     G T     G C A
T T C G T C G C A
T T C G T C G C A
T T C G T C G C A A
T A C G T C G C A A
T A C G T C G C A A
T A C G T C G T A A
T A C G T C G T A A
T A C G T C G T A C
T T C A T C G C A A
```

FASTQ

Alignment

BWA

GSNAP

gmapR

Filters

Remove PCR Dups (Picard)

Realign Indels (GATK)

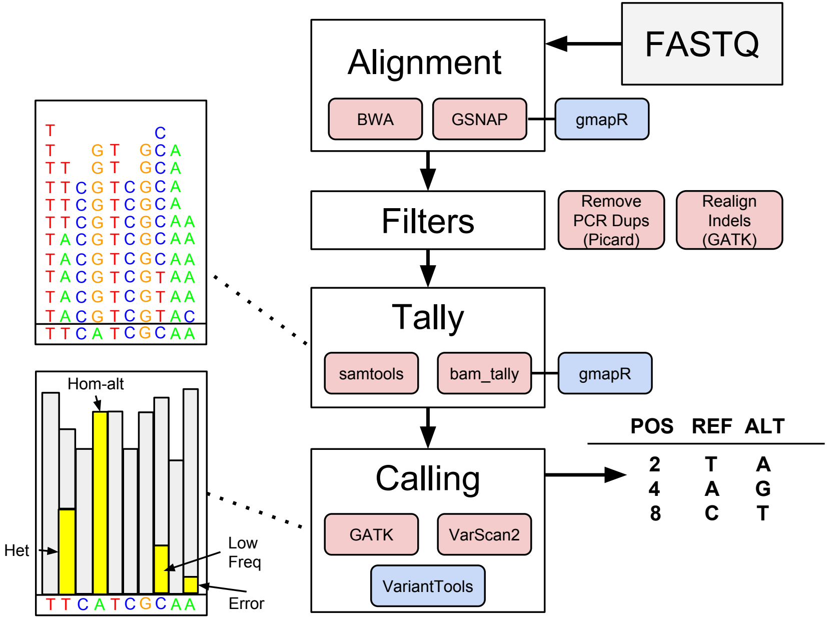
Tally

samtools

bam_tally

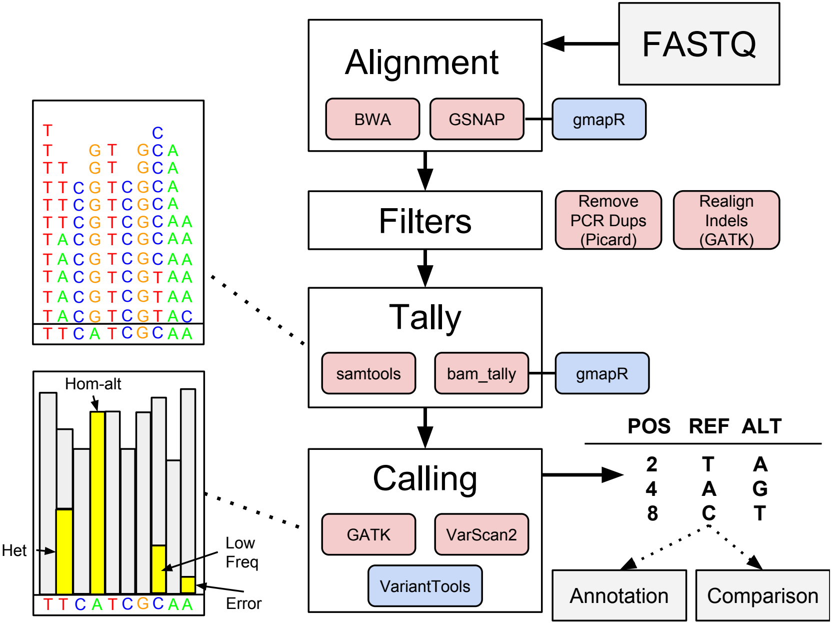
gmapR

Typical variant calling workflow



POS	REF	ALT
2	T	A
4	A	G
8	C	T

Typical variant calling workflow



POS	REF	ALT
2	T	A
4	A	G
8	C	T

Sources of technical error

Errors can occur at each stage of data generation:

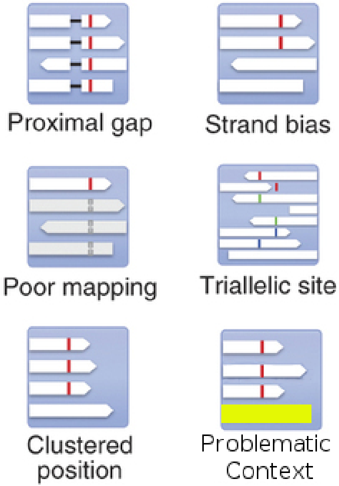
- Library prep
- Sequencing
- Alignment

Variant information for filtering

Information we know about each variant, and how it is useful:

Information	Utility
Base Qualities	Low quality indicates sequencing error
Read Positions	Bias indicates mapping issues
Genomic Strand	Bias indicates mapping issues
Genomic Position	PCR dupes; self-chain, homopolymers
Mapping Info	Aligner-dependent quality score/flags

Typical QC filters



10.1038/nbt.2514

These filters are heuristics that aim to reduce the FDR; however, they will also generate false negatives and are best applied as soft filters (annotations).

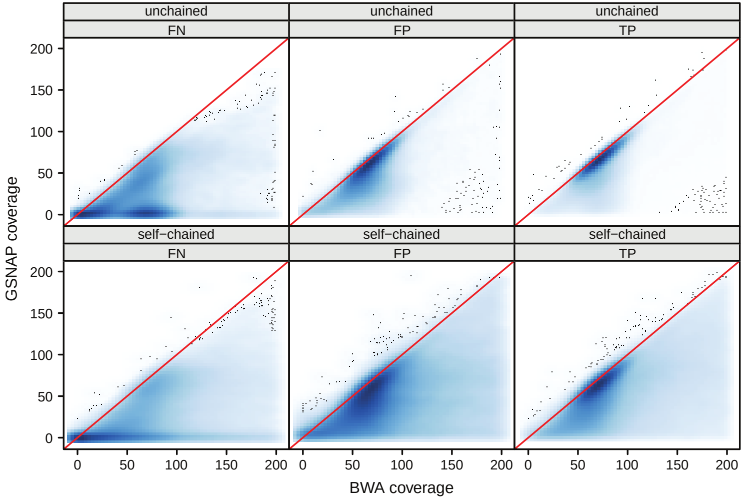
Whole-genome sequencing and problematic regions

- Many genomic regions are inherently difficult to interpret.
  - Including homopolymers, simple repeats
- These will complicate the analysis with little compensating benefit and should usually be excluded.

UCSC self-chain as indicator of mappability

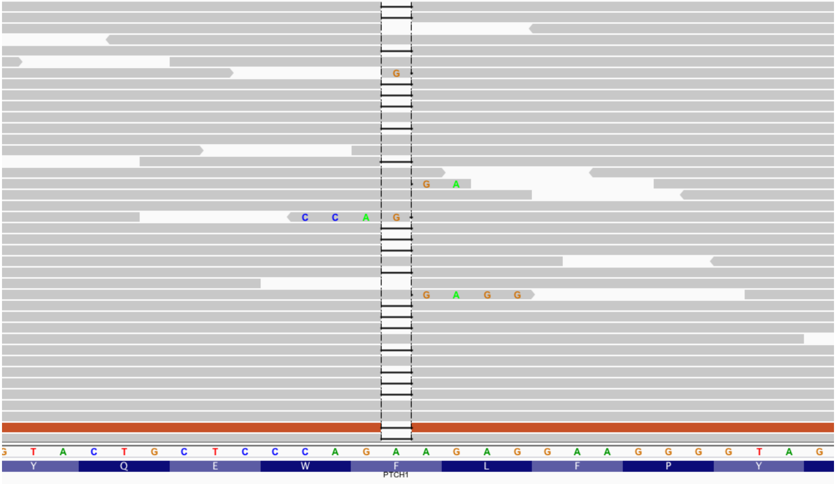
- UCSC publishes the self-chain score as a generic indicator of intragenomic similarity that is independent of any aligner
- About 6% of the genome fits this definition
- Virtually all (GSNAP) multi-mapping is in self-chains
- Lower unique coverage in self-chains

Aligner matters: coverage and mappability



Aligning indels is error prone

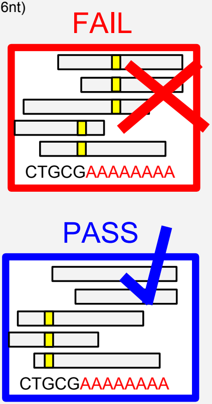
Resolved by indel realignment

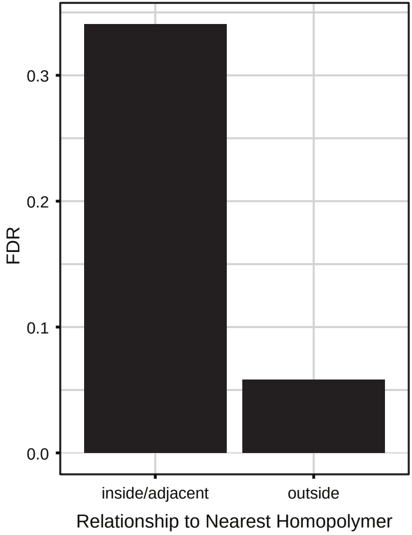


Homopolymers are problematic

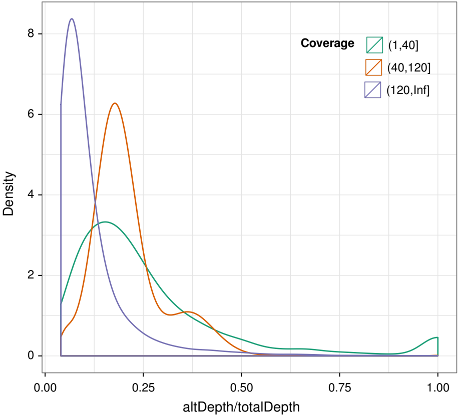
Discard variants over or next to homopolymers

(>6nt)



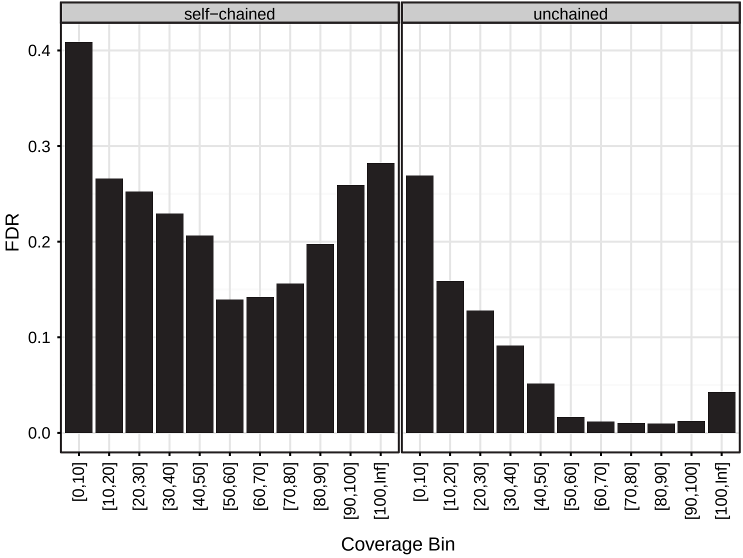


Effect of coverage extremes on frequencies

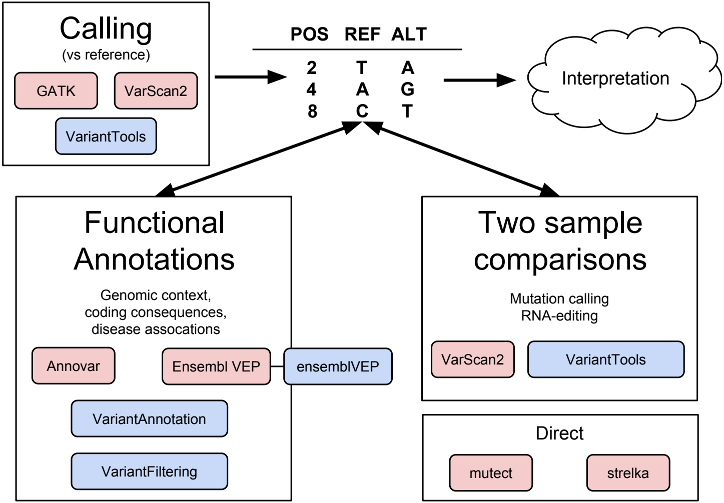


- Coverage sweet-spot (40-120) matches expected distribution.
- High coverage (>120) has much lower frequencies than expected; mapping error?
- Low coverage also different

Coverage extremes and self-chained regions



Downstream of variant calling



Calling

(vs reference)

GATK

VarScan2

VariantTools

POS	REF	ALT
2	T	A
4	A	G
8	C	T

Interpretation

Functional Annotations

Genomic context,
coding consequences,
disease assocations

Annovar

Ensembl VEP

ensemblVEP

VariantAnnotation

VariantFiltering

Two sample comparisons

Mutation calling
RNA-editing

VarScan2

VariantTools

Direct

mutect

strelka

Outline

Introduction

Calling variants with VariantTools

Diagnosing variant calls

Importing and manipulating VCF data

Comparing variant sets

Interpreting variants

Tutorial setup

Data

Alignments over chr20 from the De Pristo et. al. (GATK paper) dataset on the HapMap CEU individual NA12878.

Strategy

1. *Generate tallies from the BAM file*
2. Load pre-computer tallies (pileup) from the alignments.
3. Call/filter variants.
4. Execute basic dianostics, visualize variants in IGV.
5. Import published genotypes for NA12878 and check concordance.
6. Interpret variants (functional consequences)

Load the tutorial package

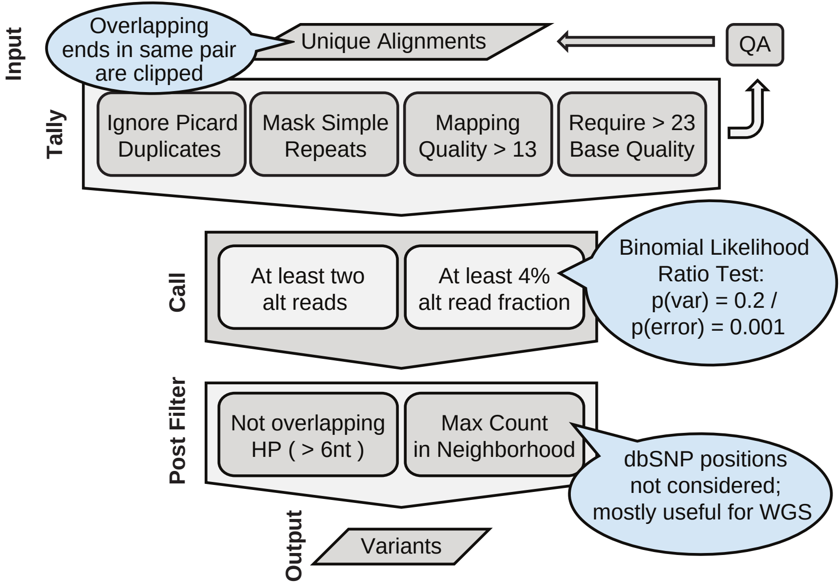
```
library(VariantCallingTutorial)
```

The VariantTools package

VariantTools is a set of utilities for:

- Tallying alignments (via gmapR)
- Annotating tallies
- Filtering tallies into variant calls
- Exporting tallies to VCF (actually VariantAnnotation)
- Wildtype calling (for a specific set of filters)
- Sample ID verification via rudimentary genotyping

Default VariantTools algorithm (WGS)


Input
Overlapping ends in same pair are clipped
Unique Alignments
QA
Tally
Ignore Picard Duplicates
Mask Simple Repeats
Mapping Quality > 13
Require > 23 Base Quality
Call
At least two alt reads
At least 4% alt read fraction
Binomial Likelihood Ratio Test: p(var) = 0.2 / p(error) = 0.001
Post Filter
Not overlapping HP (> 6nt)
Max Count in Neighborhood
dbSNP positions not considered; mostly useful for WGS
Output
Variants

How the tallies were (pre-)generated

We first prepare our parameters as a *TallyVariantsParam* object:

```
humanGmapGenome <- gmapR::GmapGenome("GRCh37")
tiles <- tileGenome(seqinfo(humanGmapGenome)["20"],
                    ntile=50)
param <- TallyVariantsParam(humanGmapGenome,
                            which = unlist(tiles),
                            mask = repeats,
                            indels = TRUE)
```

We mask out simple repeats and iterate over 50 tiles of chr20 (so as not to exhaust memory).

Tallies are generated via the `tallyVariants` function:

```
bpp <- BiocParallel::MulticoreParam(2)
tallies <- tallyVariants(bam, param, BPPARAM = bpp)
```

Loading the cached tallies

The tallies were pre-generated and placed in the package.

```
data(tallies)
```

VRanges objects

- The tallies are represented as a *VRanges* object, defined by the VariantAnnotation package
- All VariantTools filters and utilities operate on *VRanges*
- *VRanges* is an extension of *GRanges* for more formally representing variant calls

VRanges components

- On top of *GRanges*, *VRanges* adds these fixed columns:

`ref`	ref allele
`alt`	alt allele
`totalDepth`	total read depth
`refDepth`	ref allele read depth
`altDepth`	alt allele read depth
`sampleNames`	sample identifiers
`softFilterMatrix`	*FilterMatrix* of filter results
`hardFilters`	*FilterRules* used to subset object

- Unused columns are filled with a single run of NAs (slots can be either vector or *Rle*)

VRanges features

- Rough, lossy, two-way conversion between *VCF* and *VRanges*
- Matching/set operations by position and alt (`match`, `%in%`)
- Recurrence across samples (`tabulate`)
- Provenance tracking of applied hard filters
- Convenient summaries of soft filter results (*FilterMatrix*)
- Lift-over across genome builds (`liftOver`)
- *VRangesList*, stackable into a *VRanges* by sample
- All of the features of *GRanges* (overlap, etc)

Configure filters

VariantTools implements its filters within the FilterRules framework from IRanges. The default variant calling filters are constructed by VariantCallingFilters:

```
calling.filters <- VariantCallingFilters()
```

Post-filters are filters that attempt to remove anomalies from the called variants:

```
post.filters <- VariantPostFilters()
```

Hard filter tallies into variant calls

We pass the filters to the `callVariants` function:

```
variants <- callVariants(tallies,
                         calling.filters,
                         post.filters)
```

Selecting variants by type

Extra the SNVs:

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

Extract the indels:

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

Other helpers exist: `isSV`, `isSubstitution`, etc.

Outline

Introduction

Calling variants with VariantTools

Diagnosing variant calls

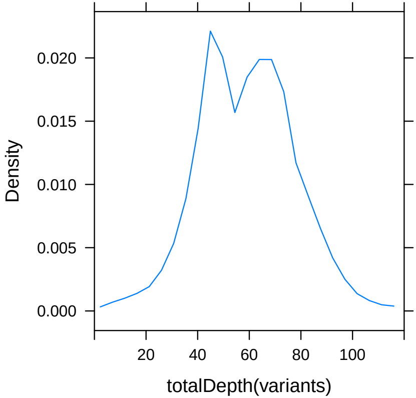
Importing and manipulating VCF data

Comparing variant sets

Interpreting variants

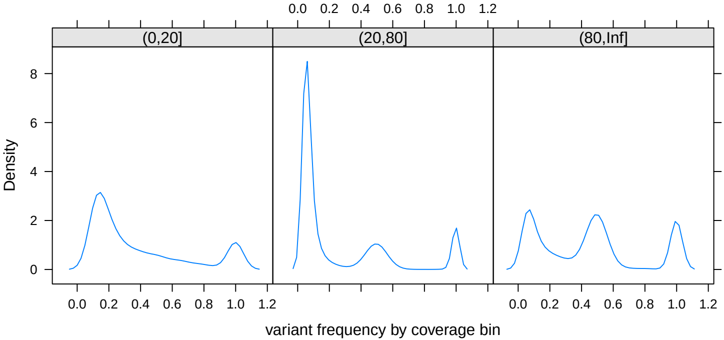
Checking the coverage

```
densityplot(~ totalDepth(variants),
            xlim=c(0, 2*median(totalDepth(variants))),
            plot.points=FALSE, n=200)
```



Checking the association of frequencies and coverage

```
variants$coverage.bin <- cut(totalDepth(variants), c(0, 20, 8
densityplot(~ altDepth/totalDepth | coverage.bin,
            as.data.frame(variants),
            plot.points=FALSE, layout=c(3, 1),
            xlab="variant frequency by coverage bin")
```



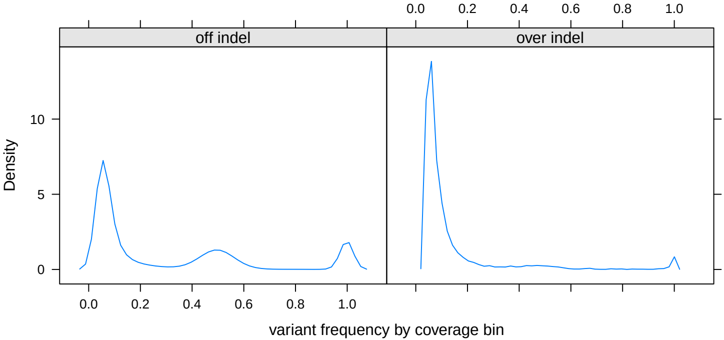
Indel proximity

Generate a window around the each indel:

```
indel.windows <- indels + 10
```

Find which SNVs overlap an indel window and summarize:

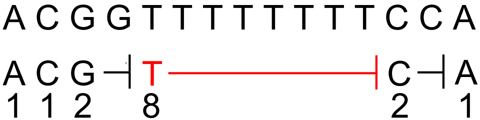
```
snvs$near.indel <- ifelse(snvs %over% indel.windows,
                          "over indel", "off indel")
```



Finding homopolymers

Find the homopolymers by forming an *Rle* on the chromosome sequence:

```
chr20.sequence <- getSeq(Hsapiens, "chr20")
chr20.hp <- ranges(Rle(as.raw(chr20.sequence)))
```



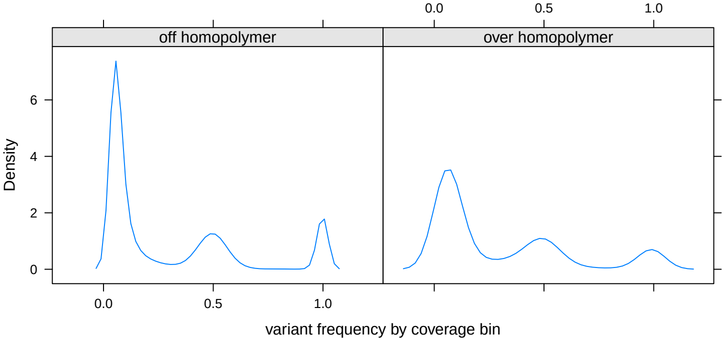
Homopolymer overlap

We consider homopolymers with length > 6:

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

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

```
snvs$over.hp <- ifelse(ranges(snvs) %over% chr20.hp,
                             "over homopolymer",
                             "off homopolymer")
```



Self-chain scores

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

```
data(selfChains, package="VariantCallingTutorial")
```

Exercise

- Find the overlap between the snvs object and the `selfChains`. Store the result on `snvs`. Summarize it somehow.

Self-chain score as soft filter

We can formally flag variants by self-chain overlap without discarding them from the dataset:

```
unchained.filter <-
  FilterRules(list(unchained = function(x) {
    x %outside% selfChains
  }))
variants <- softFilter(variants, unchained.filter)
summary(softFilterMatrix(variants))
```

```
<initial> unchained   <final>
   229272    213733    213733
```

Visualizing putative FPs: IGV

IGV is an effective tool for exploring alignment issues and other variant calling anomalies; SRAdb drives IGV from R.

To begin, we create a connection:

```
SRAdb::startIGV("lm")
sock <- SRAdb::IGVsocket()
```

Exporting our calls as VCF

IGV will display variant calls as VCF:

```
mcols(variants) <- NULL
sampleNames(variants) <- "NA12878"
vcf <- writeVcf(sort(variants),
                "variants.vcf")
vcf <- tools::file_path_as_absolute(vcf)
vcf_gz <- paste0(tools::file_path_sans_ext(vcf), ".gz")
indexTabix(bgzip(vcf, vcf_gz, overwrite=TRUE),
           format="vcf4")
```

We need to manually index the VCF, because *VariantAnnotation* uses the bgz extension, while IGV expects gz.

Creating an IGV session

Create an IGV session with our VCF, BAMs and custom p53 genome:

```
bam <- tools::file_path_as_absolute(bam)
session <- SRAdb::IGVsession(c(bam, vcf_gz),
                             "session.xml",
                             "hg19")
```

Load the session:

```
SRAdb::IGVload(sock, session)
```

Browsing regions of interest

IGV will (manually) load BED files as a list of bookmarks:

```
rtracklayer::export(variants, "roi.bed")
```

Outline

Introduction

Calling variants with VariantTools

Diagnosing variant calls

Importing and manipulating VCF data

Comparing variant sets

Interpreting variants

Goals/Motivation

- Assume we want to compare the VariantTools calls with those from the Illumina Platinum Genome for NA12878
- We need to:
  1. Import the Illumina calls from VCF (in a scalable fashion)
  2. Perform basic QC/EDA on the variants
  3. Filter out suspect and uninteresting variants

VCF: Variant Call Format

- The Variant Call Format (VCF) is the standard file format for storing variant calls.
- Every VCF file consists of two parts:
  - Header describing the format/provenance
  - Actual variant records, describing one or more alternate alleles (SNV, indel, etc) at a particular position in the genome.
- Each variant record contains information at four levels:
  - The position
  - A particular alternate allele at the position,
  - A particular sample at the position
  - A particular combination of alternate allele and sample (usually includes genotype).

VCF fields

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.

Previewing a VCF file

- For reading a VCF, we rely on the VariantAnnotation package.
- Always start by checking the header of an unfamiliar VCF file, so that we can:
  - Check the integrity of the data and
  - Determine which parts need to be imported

```
vcf.file <- NA12878_pg.chr20.vcf.bgz
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
```

Inspecting the VCF header

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    Description
    <character> <character>    <character>
GT            1      String    GT  Genotype
GQX           1     Integer    GQX Min genotype quality...
AD            .     Integer    AD  Allelic depths
DP            1     Integer    DP  Approximate read depth
GQ            1       Float    GQ  Genotype Quality
MQ            1     Integer    MQ  RMS Mapping Quality
PL            G     Integer    PL  Likelihoods for each alt
VF            1       Float    VF  Variant frequency
```

Detecting gVCF

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.

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?

Full VCF import

- We now load the VCF data into R using `readVcf()`:

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

- We pass a genome identifier to track provenance and ensure data integrity.
- Loading the full file consumes a lot of memory, much of which may be wasted if we are only interested in one region, or some subset of the fields
- This is the size for chr20:

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

```
300 Mb
```

- But for the genome, this would be closer to 12 GB, which is beyond the memory capacity of most laptops

Restricted VCF import

- Restricted import is useful when:
  - We need only a subset of the data
  - We want to process all of the data in an interative fashion
- `readVcf` supports multiple modes of restriction:
  - Range-based for selecting a region (`vcfWhich`)
  - Field-based for skipping irrelevant fields (`vcfWhat`)
  - Row chunk-based for streaming (`yieldSize`)

Restricting by genomic range

- Assume we had a VCF for the whole genome and we want to restrict it to chr20
- First, we obtain the chr20 range:

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

- And pass the range of interest to readVcf():

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

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.

VCF objects

The `readVcf()` function returns a *VCF* object, a derivative of *SummarizedExperiment* that fully and formally represents the complexity of the VCF file.

`rowData(vcf)` *GRanges* object holding the positions and fixed, position-level columns

```
colnames(mcols(rowData(vcf)))
```

```
[1] "paramRangeID" "REF" "ALT" "QUAL" "FILTER"
```

`info(vcf)` *DataFrame* of position- and alt-level fields

```
head(colnames(info(vcf)))
```

```
[1] "AC" "AF" "AN"
[4] "DP" "QD" "BLOCKAVG_min30p3a"
```

`geno(vcf)` Sample-specific values, including the genotype

```
names(geno(vcf))
```

```
[1] "GT" "GQX" "AD" "DP" "GQ" "MQ" "PL" "VF"
```

VCF subsetting/extraction

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

```
illumina_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.

Selection by variant type

Of the fixed columns, the most important is the ALT column, which stores the alternate allele(s) for each record.

```
head(alt(illumina_variants))
```

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

Restrict to SNVs using the isSNV helper:

```
selectSNVs <- isSNV(illumina_variants)
illumina_snvs <- subset(illumina_variants, selectSNVs)
```

Expanding the *VCF*: one alt per record

VCF supports multiple alts per row:

```
class(alt(illumina_snvs))
```

```
[1] ”DNAStringSetList”
attr(,”package”)
[1] ”Biostrings”
```

But it is easier to reason on data with one alt per row, which we can achieve with expand:

```
illumina_snvs <- expand(illumina_snvs)
class(alt(illumina_snvs))
```

```
[1] ”DNAStringSet”
attr(,”package”)
[1] ”Biostrings”
```

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.

Filtering a VCF file: filterVcf

- After filtering, the *VCF* object is much smaller

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

  ```
  37.2 Mb
  ```

- The reduction is sufficient for us to operate on the calls from the entire genome, not just chr20.
- In general, we want to apply the filter to the entire file, without loading all of the data into memory.
- The `filterVcf` function steams over a VCF file, writing out the records that pass two types of filters:

  `prefilters` Applied to the raw text of this file (faster but riskier), and

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

Filtering for the called SNVs

We filter out non-variant sites using a prefilter:

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

We also need to restrict to SNVs, which is sufficiently complicated to warrant filtering after parsing:

```
filters <- FilterRules(list(onlySNVs=isSNV))
```

We combine both of the filters and drop the unneeded `info` columns in a single step:

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

Exercises

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

Accessing included QC filters

- The `filt()` component marks which QC filters failed a variant
- Variants that pass all QC filters are labeled "PASS"
- Here, we list the descriptions of the filter codes from the header:

```
fixed(header(illumina_snvs))$FILTER
```

```
DataFrame with 9 rows and 1 column
Description
<character>
LowGQX               Locus GQX < 30.0000 or not present
LowQD                Locus QD < than 2.0000
LowMQ                Site MQ < than 20.0000
IndelConflict        Region has conflicting indel calls
MaxDepth             Site depth > 3.0x the mean depth
SiteConflict         Genotype conflicts with proximal indel
...                  ...
```

Summarizing the QC filters

```
table(unlist(strsplit(filt(illumina_snvs), ";")))
```

```
IndelConflict       LowGQX
           35        11020
        LowMQ        LowQD
         1055        19583
     MaxDepth         PASS
           25        68626
 SiteConflict          ...
          785          ...
```

Restricting by the QC filters

For the purposes of this tutorial, we will restrict to the "PASS" variants:

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

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

```
rowData(illumina_snvs)$coverage.bin <-
  cut(geno(illumina_snvs)$DP, c(0, 20, 80, Inf))
table(rowData(illumina_snvs)$coverage.bin)
```

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

Exercises

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

Outline

Introduction

Calling variants with VariantTools

Diagnosing variant calls

Importing and manipulating VCF data

Comparing variant sets

Interpreting variants

Goals

- Compare the output from VariantTools from the Illumina "platinum" genotypes for the same individual (NA12878)
- Intersect the variant sets using *VRanges* to determine the FP and FN calls for VariantTools
- Compare the variants in terms of their frequencies

Coercing to *VCF* to *VRanges*

Coercion

```
illumina_vr <- as(illumina_snvs, "VRanges")
```

Rectify differences in reference genome

```
seqlevelsStyle(illumina_vr) <- "NCBI"
illumina_vr <- dropSeqlevels(illumina_vr, "MT")
genome(illumina_vr) <- "GRCh37"
```

Intersecting variant sets

Sensitivity

```
illumina_vr$in.vt <- illumina_vr %in% snvs
mean(illumina_vr$in.vt)
```

```
[1] 0.9654358
```

Specificity

```
snvs$in.illumina <- snvs %in% illumina_vr
mean(snvs$in.illumina)
```

```
[1] 0.3085657
```

Compare variant frequencies

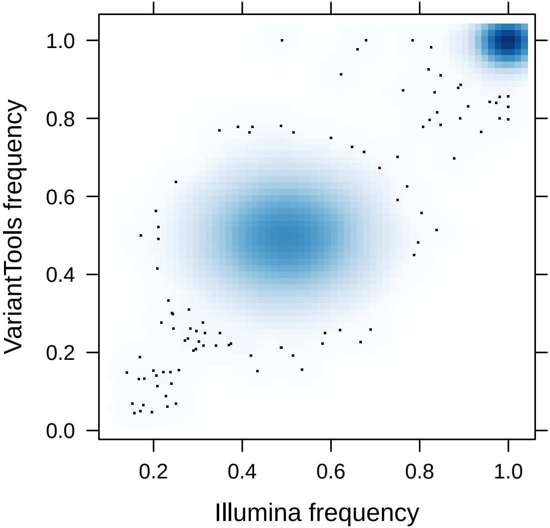
We merge the variant frequencies from the VariantTools set into the Illumina set:

```
illumina_vr$vt.freq <-
  altFraction(snvs)[match(illumina_vr, snvs)]
```

Now we can make a scatterplot:

```
xyplot(vt.freq ~ altFraction(illumina_vr),
       as.data.frame(illumina_vr),
       panel=panel.smoothScatter,
       xlab="Illumina frequency",
       ylab="VariantTools frequency")
```

Compare variant frequencies



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?

Manipulating gVCF runs

- This file is actually a gVCF file, where the "g" stands for genotype
- The extended format supports storing runs that indicate confidence in the WT genotype
- For the unique VariantTools calls, we can retrieve Illumina's confidence that the positions are indeed wildtype
- We begin by converting the positions in the *VCF* to runs

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

Exercises

1. Assuming that a "PASS" value for `filt(runs)` indicates wildtype and no-call otherwise, what percentage of chr20 was callable?
2. For the unique VariantTools variants found in a previous exercise, how many of them were called wildtype vs. no-call by Illumina?

False negatives: which filter to blame?

Apply the calling filters to our FN and summarize the results:

```
calling.filters <- hardFilters(snvs)[3:5]
tallies <- resetFilter(tallies)
tallies <- softFilter(tallies, calling.filters,
                      serial=TRUE)
fn <- tallies[tallies %in% subset(illumina_vr, !in.vt)]
t(summary(softFilterMatrix(fn)))
```

<initial>	readCount	likelihoodRatio	avgNborCount	<final>
1052	1036	1035	0	0

Outline

Introduction

Calling variants with VariantTools

Diagnosing variant calls

Importing and manipulating VCF data

Comparing variant sets

Interpreting variants

Genomic context

`locateVariants()` annotates variants with overlapping genes.

```
gene.models <- TxDb.Hsapiens.UCSC.hg19.knownGene
snvs <- keepSeqlevels(snvs, "20")
seqlevelsStyle(snvs) <- "UCSC"
genome(snvs) <- "hg19"
locations <- locateVariants(snvs, gene.models,
                            CodingVariants())
```

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

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

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

Munging the genomic context

- The QUERYID column maps each row in locations to a row in the input (only a subset of the variants are over a gene):

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

- We can use annotations to retrieve the gene symbols:

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

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?

Coding consequences

predictCoding() predicts coding consequences:

```
coding <- predictCoding(snvs, gene.models, Hsapiens,
                        varAllele = DNAStringSet(alt(snvs)))
```

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

```
setdiff(colnames(mcols(coding)), colnames(mcols(snvs)))
```

```
 [1] "varAllele"   "CDSLOC"      "PROTEINLOC"  "QUERYID"     "
 [6] "CDSID"       "GENEID"      "CONSEQUENCE" "REFCODON"    "
[11] "REFAA"       "VARAA"
```

Summarizing the consequences

We tabulate the consequence codes:

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

```
nonsense nonsynonymous    synonymous
      66          3651          1259
```

Exercises

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