Annotations, Visualization, and Variants

Marc Carlson, Valerie Obenchain, Hervé Pagès, Paul Shannon, Dan Tenenbaum, Martin Morgan[1]

June 23 – 28, 2013

[1] mtmorgan@fhcrc.org

Contents

Part I

Annotation and Visualization (Intermediate)

Chapter 1

Gene-centric Annotation

Be sure to update your package to the current version by visiting the course web server 192.168.0.9 and installing (if neccessary) the current version of the *Morgan2013* package.

```
> PKG_OK <- packageDescription("Morgan2013")$Version >= "0.9.0"
> stopifnot(PKG_OK)
```

Bioconductor provides extensive annotation resources, some of which are summarized in Figure 1.1. These can be *gene*-, or *genome*-centric. Annotations can be provided in packages curated by *Bioconductor*, or obtained from web-based resources. Gene-centric *AnnotationDbi* packages include:

- Organism level: e.g. *org.Mm.eg.db*, *Homo.sapiens*.
- Platform level: e.g. *hgu133plus2.db*, *hgu133plus2.probes*, *hgu133plus2.cdf*.
- Homology level: e.g. *hom.Dm.inp.db*.
- System biology level: *GO.db*, *KEGG.db*, *Reactome.db*.

Examples of genome-centric packages include:

- *GenomicFeatures*, to represent genomic features, including constructing reproducible feature or transcript data bases from file or web resources.
- Pre-built transcriptome packages, e.g. *TxDb.Hsapiens.UCSC.hg19.knownGene* based on the *H. sapiens* UCSC hg19 knownGenes track.
- *BSgenome* for whole genome sequence representation and manipulation.
- Pre-built genomes, e.g., *BSgenome.Hsapiens.UCSC.hg19* based on the *H. sapiens* UCSC hg19 build.

Web-based resources include

- *biomaRt* to query biomart resource for genes, sequence, SNPs, and etc.
- *rtracklayer* for interfacing with browser tracks, especially the UCSC genome browser.
- *AnnotationHub* for retrieving common large genomic resources in formats that readily fit into *R* work flows.

1.1 Gene-centric annotations with *AnnotationDbi*

Organism-level ('org') packages contain mappings between a central identifier (e.g., Entrez gene ids) and other identifiers (e.g. GenBank or Uniprot accession number, RefSeq id, etc.). The name of an Rpackageorg package is always of the form *org.<Sp>.<id>.db* (e.g. *org.Sc.sgd.db*) where *<Sp>* is a 2-letter abbreviation of the organism (e.g. `Sc` for *Saccharomyces cerevisiae*) and *<id>* is an abbreviation (in lower-case) describing the type of central identifier (e.g. `sgd` for gene identifiers assigned by the *Saccharomyces* Genome Database, or `eg` for Entrez gene ids). The 'How to use the ".db" annotation packages' vignette in the *AnnotationDbi* package (org packages are only one type of '.db' annotation packages) is a key reference. The '.db' and most other *Bioconductor* annotation packages are updated every 6 months.

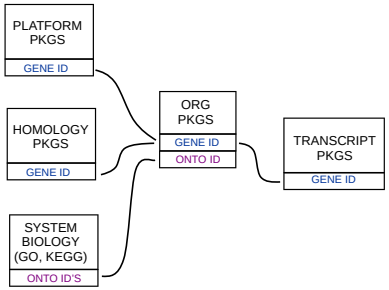


Figure 1.1: Annotation Packages: the big picture

Table 1.1: Common operations for retrieving and manipulating annotations.

Category	Function	Description
Discover	`cols`	List the kinds of columns that can be returned
	`keytypes`	List columns that can be used as keys
	`keys`	List values that can be expected for a given keytype
	`select`	Retrieve annotations matching `keys`, `keytype` and `cols`
Manipulate	`setdiff`, `union`, `intersect`	Operations on sets
	`duplicated`, `unique`	Mark or remove duplicates
	`%in%`, `match`	Find matches
	`any`, `all`	Are any `TRUE`? Are all?
	`merge`	Combine two different `data.frames` based on shared keys
*GRanges**	`transcripts`, `exons`, `cds`	Features (transcripts, exons, coding sequence) as *GRanges*.
	`transcriptsBy` , `exonsBy` `cdsBy`	Features group by gene, transcript, etc., as *GRangesList*.

Annotation packages contain an object named after the package itself. These objects are collectively called *AnnotationDb* objects, with more specific classes named *OrgDb*, *ChipDb* or *TranscriptDb* objects. Methods that can be applied to these objects include `cols`, `keys`, `keytypes` and `select`. Common operations for retrieving annotations are summarized in Table 1.1.

Exercise 1
What is the name of the org package for Drosophila*? Load it. Display the OrgDb object for the org.Dm.eg.db package. Use the* `cols` *method to discover which sorts of annotations can be extracted from it.*

Use the `keys` *method to extract UNIPROT identifiers and then pass those keys in to the* `select` *method in such a way that you extract the SYMBOL (gene symbol) and KEGG pathway information for each.*

Use `select` *to retrieve the ENTREZ and SYMBOL identifiers of all genes in the KEGG pathway 00310.*

Solution: The *OrgDb* object is named `org.Dm.eg.db`.

```
> library(org.Dm.eg.db)
> cols(org.Dm.eg.db)
```

```
 [1] "ENTREZID"     "ACCNUM"       "ALIAS"        "CHR"          "CHRLOC"
 [6] "CHRLOCEND"    "ENZYME"       "MAP"          "PATH"         "PMID"
[11] "REFSEQ"       "SYMBOL"       "UNIGENE"      "ENSEMBL"      "ENSEMBLPROT"
[16] "ENSEMBLTRANS" "GENENAME"     "UNIPROT"      "GO"           "EVIDENCE"
[21] "ONTOLOGY"     "GOALL"        "EVIDENCEALL"  "ONTOLOGYALL"  "FLYBASE"
[26] "FLYBASECG"    "FLYBASEPROT"
```



```
> keytypes(org.Dm.eg.db)
```

```
 [1] "ENTREZID"     "ACCNUM"       "ALIAS"        "CHR"          "CHRLOC"
 [6] "CHRLOCEND"    "ENZYME"       "MAP"          "PATH"         "PMID"
[11] "REFSEQ"       "SYMBOL"       "UNIGENE"      "ENSEMBL"      "ENSEMBLPROT"
[16] "ENSEMBLTRANS" "GENENAME"     "UNIPROT"      "GO"           "EVIDENCE"
[21] "ONTOLOGY"     "GOALL"        "EVIDENCEALL"  "ONTOLOGYALL"  "FLYBASE"
[26] "FLYBASECG"    "FLYBASEPROT"
```

```
> uniprotKeys <- head(keys(org.Dm.eg.db, keytype="UNIPROT"))
> cols <- c("SYMBOL", "PATH")
> select(org.Dm.eg.db, keys=uniprotKeys, cols=cols, keytype="UNIPROT")
```

```
  UNIPROT  SYMBOL  PATH
1  Q8IRZ0  CG3038  <NA>
2  Q95RP8  CG3038  <NA>
3  Q95RU8     G9a 00310
4  Q9W5H1 CG13377  <NA>
5  P39205     cin  <NA>
6  Q24312     ewg  <NA>
```

Selecting UNIPROT and SYMBOL ids of KEGG pathway `00310` is very similar:

```
> kegg <- select(org.Dm.eg.db, "00310", c("UNIPROT", "SYMBOL"), "PATH")
> nrow(kegg)
```

```
[1] 32
```

```
> head(kegg, 3)
```

```
   PATH UNIPROT  SYMBOL
1 00310  Q95RU8     G9a
2 00310  Q9W5E0 Hmt4-20
3 00310  Q9W3N9 CG10932
```

Exercise 2
For convenience, `lrTest`*, a DGEGLM object from an RNA-seq experiment, is included in the Morgan2013 package. The following code loads this data and creates a 'top table' of the ten most differentially represented genes. This top table is then coerced to a* `data.frame`*.*

```
> library(Morgan2013)
> library(edgeR)
> library(org.Dm.eg.db)
> data(lrTest)
> tt <- as.data.frame(topTags(lrTest))
```

Extract the Flybase gene identifiers (`FLYBASE`*) from the row names of this table and map them to their corresponding Entrez gene (*`ENTREZID`*) and symbol ids (*`SYMBOL`*) using* `select`*. Use* `merge` *to add the results of* `select` *to the top table.*

Solution:

Table 1.2: Selected packages querying web-based annotation services.

Package	Description
biomaRt	http://biomart.org, Ensembl and other annotations
rtracklayer	http://genome.ucsc.edu, genome tracks.
AnnotationHub	Ensembl, Encode, dbSNP, UCSC data objects
uniprot.ws	http://uniprot.org, protein annotations
KEGGREST	http://www.genome.jp/kegg, KEGG pathways
SRAdb	http://www.ncbi.nlm.nih.gov/sra, sequencing experiments.
GEOquery	http://www.ncbi.nlm.nih.gov/geo/, array and other data
ArrayExpress	http://www.ebi.ac.uk/arrayexpress/, array and other data

```
> fbids <- rownames(tt)
> cols <- c("ENTREZID", "SYMBOL")
> anno <- select(org.Dm.eg.db, fbids, cols, "FLYBASE")
> ttanno <- merge(tt, anno, by.x=0, by.y="FLYBASE")
> dim(ttanno)
```

```
[1] 10  8
```

```
> head(ttanno, 3)
```

```
    Row.names logConc logFC LR.statistic  PValue     FDR ENTREZID  SYMBOL
1 FBgn0000071     -11   2.8          183 1.1e-41 1.1e-38    40831     Ama
2 FBgn0024288     -12  -4.7          179 7.1e-41 6.3e-38    45039 Sox100B
3 FBgn0033764     -12   3.5          188 6.8e-43 7.8e-40     <NA>    <NA>
```

1.2 *biomaRt* and other web-based resources

A short summary of select *Bioconductor* packages enabling web-based queries is in Table 1.2.

1.2.1 Using biomaRt

The *biomaRt* package offers access to the online biomart resource. this consists of several data base resources, referred to as 'marts'. Each mart allows access to multiple data sets; the *biomaRt* package provides methods for mart and data set discovery, and a standard method `getBM` to retrieve data.

Exercise 3
WARNING: Internet connection required

Load the biomaRt package and list the available marts. Choose the ensembl *mart and list the datasets for that mart. Set up a mart to use the* ensembl *mart and the* hsapiens_gene_ensembl *dataset.*

A biomaRt dataset can be accessed via `getBM`*. In addition to the mart to be accessed, this function takes filters and attributes as arguments. Use* `filterOptions` *and* `listAttributes` *to discover values for these arguments. Call* `getBM` *using filters and attributes of your choosing.*

Solution:

```
> library(biomaRt)
> head(listMarts(), 3)                      ## list the marts
> head(listDatasets(useMart("ensembl")), 3) ## mart datasets
> ensembl <-                                ## fully specified mart
+     useMart("ensembl", dataset = "hsapiens_gene_ensembl")
> head(listFilters(ensembl), 3)             ## filters
> myFilter <- "chromosome_name"
> head(filterOptions(myFilter, ensembl), 3) ## return values
```



```
> myValues <- c("21", "22")
> head(listAttributes(ensembl), 3)          ## attributes
> myAttributes <- c("ensembl_gene_id","chromosome_name")
> ## assemble and query the mart
> res <- getBM(attributes =  myAttributes, filters =  myFilter,
+              values =  myValues, mart = ensembl)
```

Use `head(res)` to see the results.

Chapter 2

Genomic Annotation

2.1 *AnnotationHub*

WARNING: This section requires an internet connection, and has been removed from the lab.

2.2 Whole genome sequences

There are a diversity of packages and classes available for representing large genomes. Several include:

TxDb.* For transcript and other genome / coordinate annotation.

BSgenome For whole-genome representation. See `available.packages` for pre-packaged genomes, and the vignette 'How to forge a BSgenome data package' in the

Homo.sapiens For integrating *TxDb** and *org.** packages.

SNPlocs.* For model organism SNP locations derived from dbSNP.

`FaFile` (*Rsamtools*) for accessing indexed FASTA files.

SIFT.**, PolyPhen Variant effect scores.

2.3 Gene models

2.3.1 *TxDb.** packages for model organisms

Genome-centric packages are very useful for annotations involving genomic coordinates. It is straightforward, for instance, to discover the coordinates of coding sequences in regions of interest, and from these retrieve corresponding DNA or protein coding sequences. Other examples of the types of operations that are easy to perform with genome-centric annotations include defining regions of interest for counting aligned reads in RNA-seq experiments and retrieving DNA sequences underlying regions of interest in ChIP-seq analysis, e.g., for motif characterization.

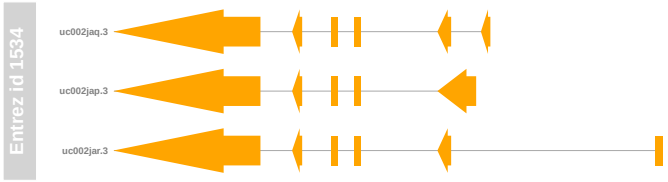


Figure 2.1: Gene model showing exons on three transcripts.

Exercise 4
Load the 'transcript.db' package relevant to the dm3 build of D. melanogaster. *Use* `select` *and friends to select the Flybase gene ids of the top table* `tt` *and the Flybase transcript names (TXNAME) and Entrez gene identifiers (GENEID).*

Use `cdsBy` *to extract all coding sequences, grouped by transcript. Subset the coding sequences to contain just the transcripts relevant to the top table. How many transcripts are there? What is the structure of the first transcript's coding sequence?*

Load the 'BSgenome' package for the dm3 build of D. melanogaster. *Use the coding sequences ranges of the previous part of this exercise to extract the underlying DNA sequence, using the* `extractTranscriptsFromGenome` *function. Use Biostrings'* `translate` *to convert DNA to amino acid sequences.*

Solution: The following loads the relevant Transcript.db package, and creates a more convenient alias to the *TranscriptDb* instance defined in the package.

```
> library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
> txdb <- TxDb.Dmelanogaster.UCSC.dm3.ensGene
```

We also need the data – flybase IDs from our differential expression analysis.

```
> library(Morgan2013)
> library(edgeR)
> data(lrTest)
> fbids <- rownames(topTags(lrTest))
```

We can discover available keys (using `keys`) and columns (`cols`) in `txdb`, and then use `select` to retrieve the transcripts associated with each differentially expressed gene. The mapping between gene and transcript is not one-to-one – some genes have more than one transcript.

```
> txnm <- select(txdb, fbids, "TXNAME", "GENEID")
> nrow(txnm)

[1] 19

> head(txnm, 3)

       GENEID      TXNAME
1 FBgn0039155 FBtr0084549
2 FBgn0039827 FBtr0085755
3 FBgn0039827 FBtr0085756
```

The *TranscriptDb* instances can be queried for data that is more structured than simple data frames, and in particular return *GRanges* or *GRangesList* instances to represent genomic coordinates. These queries are performed using `cdsBy` (coding sequence), `transcriptsBy` (transcripts), etc., where the function argument `by` specifies how coding sequences or transcripts are grouped. Here we extract the coding sequences grouped by transcript, returning the transcript names, and subset the resulting *GRangesList* to contain just the transcripts of interest to us. The first transcript is composed of 6 distinct coding sequence regions.

```
> cds <- cdsBy(txdb, "tx", use.names=TRUE)[txnm$TXNAME]
> length(cds)

[1] 19

> cds[1]

GRangesList of length 1:
$FBtr0084549
GRanges with 6 ranges and 3 metadata columns:
      seqnames              ranges strand |    cds_id    cds_name exon_rank
         <Rle>           <IRanges>  <Rle> | <integer> <character> <integer>
```

```
  [1]     chr3R [19970946, 19971592]      + |     39378        <NA>         2
  [2]     chr3R [19971652, 19971770]      + |     39379        <NA>         3
  [3]     chr3R [19971831, 19972024]      + |     39380        <NA>         4
  [4]     chr3R [19972088, 19972461]      + |     39381        <NA>         5
  [5]     chr3R [19972523, 19972589]      + |     39382        <NA>         6
  [6]     chr3R [19972918, 19973094]      + |     39383        <NA>         7

---
seqlengths:
     chr2L     chr2R     chr3L     chr3R ...   chrXHet   chrYHet chrUextra
  23011544  21146708  24543557  27905053 ...    204112    347038  29004656
```

The following code loads the appropriate BSgenome package; the `Dmelanogaster` object refers to the whole genome sequence represented in this package. The remaining steps extract the DNA sequence of each transcript, and translates these to amino acid sequences. Issues of strand are handled such that gene regions on the minus strand are reverse compelemented, so returned in 5' to 3' orientation).

```
> library(BSgenome.Dmelanogaster.UCSC.dm3)
> txx <- extractTranscriptsFromGenome(Dmelanogaster, cds)
> length(txx)

[1] 19

> head(txx, 3)

  A DNAStringSet instance of length 3
    width seq                                               names
[1]  1578 ATGGGCAGCATGCAAGTGGCGCT...TGCAGATCAAGTGCAGCGACTAG FBtr0084549
[2]  2760 ATGCTGCGTTATCTGGCGCTTTC...TTGCTGCCCCATTCGAACTTTAG FBtr0085755
[3]  2217 ATGGCACTCAAGTTTCCCACAGT...TTGCTGCCCCATTCGAACTTTAG FBtr0085756

> head(translate(txx), 3)

  A AAStringSet instance of length 3
    width seq
[1]   526 MGSMQVALLALLVLGQLFPSAVANGSSSYSSTST...VLDDSRNVFTFTTPKCENFRKRFPKLQIKCSD*
[2]   920 MLRYLALSEAGIAKLPRPQSRCYHSEKGVWGYKP...YCGRCEAPTPATGIGKVHKREVDEIVAAPFEL*
[3]   739 MALKFPTVKRYGGEGAESMLAFFWQLLRDSVQAN...YCGRCEAPTPATGIGKVHKREVDEIVAAPFEL*
```

2.4 UCSC tracks

2.4.1 Easily creating *TranscriptDb* objects from GTF files

Exercise 5
WARNING: Internet connection required (Adapted from[1]) The 'track' curated annotations at UCSC are a great resource; this exercise creates a TranscriptDb instance from one such track.

a. *Load the GenomicFeatures and rtracklayer packages.*

b. *Discover available genomes with* `ucscGenomes`*, and available tables with the* `supportedUCSCtables`*.*

c. *Use the* `makeTranscriptDbFromUCSC` *from a suitable track, e.g., genome* `genome="ce2", tablename="refGene"`*. There are some warnings; are these something to be concerned about?*

d. *Exercise the object that you've created, e.g., exploring the basis of the warnings.*

[1] http://www.sph.emory.edu/~hwu/teaching/bioc/bios560R.html

e. *Save and load the TranscriptDb object you've created, illustrating how one can make these annotations convenient and reproducible.*

f. *What's the difference between* `makeTranscriptDbFromUCSC` *and* `makeFeatureDbFromUCSC`*? Where else can transcript and feature data bases be made from?*

Solution: Load the *GenomicFeatures* and *rtracklayer* packages and discover available genomes and tables:

```
> library(rtracklayer)
> library(GenomicFeatures)
> ## genomes
> gnms <- ucscGenomes()
> nrow(gnms)

[1] 147

> gnms[grep("elegans", gnms$species),]

      db    species      date              name
136 ce10 C. elegans Oct. 2010 WormBase v. WS220
137  ce6 C. elegans  May 2008 WormBase v. WS190
138  ce4 C. elegans Jan. 2007 WormBase v. WS170
139  ce2 C. elegans Mar. 2004 WormBase v. WS120

> ## tables
> tbls <- supportedUCSCtables()
> nrow(tbls)

[1] 25

> head(tbls)

                                  track        subtrack
knownGene                   UCSC Genes            <NA>
knownGeneOld3            Old UCSC Genes            <NA>
wgEncodeGencodeManualV3  Gencode Genes Genecode Manual
wgEncodeGencodeAutoV3    Gencode Genes   Genecode Auto
wgEncodeGencodePolyaV3   Gencode Genes  Genecode PolyA
ccdsGene                       CCDS            <NA>
```

Make the *TranscriptDb* object (this will take a minute)

```
> ## Not run
> txdb <- makeTranscriptDbFromUCSC("ce10", "refGene")
> saveDb(txdb, file="/path/to/file.sqlite")
```

The warnings during object creation are about unusual lengths for CDS (coding sequences should be in multiples of 3, since there are three nucleotide residues per amino acid residue).

```
1: In .extractUCSCCdsStartEnd(cdsStart[i], cdsEnd[i], ... :
  UCSC data anomaly in transcript NM_001129046: the cds cumulative
  length is not a multiple of 3
```

but we seem to have a useful object with relevant metadata information for reproducible research:

```
> txdb
```

```
TranscriptDb object:
| Db type: TranscriptDb
| Supporting package: GenomicFeatures
| Data source: UCSC
| Genome: ce10
| Organism: Caenorhabditis elegans
| UCSC Table: refGene
| Resource URL: http://genome.ucsc.edu/
| Type of Gene ID: Entrez Gene ID
| Full dataset: yes
| miRBase build ID: NA
| transcript_nrow: 48714
| exon_nrow: 152542
| cds_nrow: 129947
| Db created by: GenomicFeatures package from Bioconductor
| Creation time: 2013-02-10 10:33:50 -0800 (Sun, 10 Feb 2013)
| GenomicFeatures version at creation time: 1.11.8
| RSQLite version at creation time: 0.11.2
| DBSCHEMAVERSION: 1.0
```

Let us investigate the source of the warnings – what fraction of the CDS have lengths that are not a multiple of 3? To do this we need to determine the sum of the widths of the coding sequence exons in each transcript. Start by extracting all coding sequence exons, grouped by transcript; verify that this is a *GRangesList* with a reasonable number of entries.

```
> cdsByTx <- cdsBy(txdb, "tx", use.names=TRUE)
> length(cdsByTx)

[1] 26146

> cdsByTx[1:2]

GRangesList of length 2:
$NM_058259
GRanges with 3 ranges and 3 metadata columns:
      seqnames         ranges strand |    cds_id    cds_name exon_rank
         <Rle>      <IRanges>  <Rle> | <integer> <character> <integer>
  [1]     chrI [11641, 11689]      + |         1        <NA>         1
  [2]     chrI [14951, 15160]      + |         2        <NA>         2
  [3]     chrI [16473, 16585]      + |         3        <NA>         3

$NM_058264
GRanges with 5 ranges and 3 metadata columns:
      seqnames         ranges strand | cds_id cds_name exon_rank
  [1]     chrI [43733, 43961]      + |      4     <NA>         1
  [2]     chrI [44030, 44234]      + |      5     <NA>         2
  [3]     chrI [44281, 44324]      + |      6     <NA>         3
  [4]     chrI [44372, 44468]      + |      7     <NA>         4
  [5]     chrI [44521, 44677]      + |      8     <NA>         5

---
seqlengths:
     chrI    chrII   chrIII    chrIV     chrV     chrX     chrM
 15072423 15279345 13783700 17493793 20924149 17718866    13794
```

Extract the width of each each exon; this is an *IntegerList* instance with one element of the list for each transcript, and one integer value for each exon.


```
> wd <- width(cdsByTx)
> length(wd)

[1] 26146

> head(wd, 3)

CompressedIntegerList of length 3
[["NM_058259"]] 49 210 113
[["NM_058264"]] 229 205 44 97 157
[["NM_001026606"]] 139 163 183 166
```

Use `sum` to add up the widths within each list element; note that we're using the `sum,CompressedIntegerList-method`, and that this has been specialized to do the summation within list elements.

```
> head(sum(wd))

   NM_058259    NM_058264 NM_001026606    NM_058265 NM_001026607    NM_182094
         372          732          651         1341         1620         1221
```

We now have a standard *R* vector; a one-liner asking about the number of transcripts with exons that do not sum to 3 is

```
> table( (sum(width(cdsBy(txdb, "tx"))) %% 3) != 0 )

FALSE  TRUE
25676   470
```

Exercise 6
(Adapted from[2]) Suppose you have a list of transcription factor binding sites on hg19. How would you obtain (a) the GC content of each site and (b) the percentage of gene promoters covered by the binding sites?

Solution: As an outline of a solution, the steps for calculating GC content might be

a. Represent the list of transcription factor binding sites ('regions of interest') as a *GRanges* instance, `roi`.
b. Load *BSgenome* and the appropriate genome package, e.g., *BSgenome.Hsapiens.UCSC.hg19*.
c. Use `getSeq` to retrieve the sequences, `seqs <- getSeq(Hsapiens, gr)`.
d. Use `alphabetFrequency(seqs)` to summarize nucleotide use, and simple *R* functions to determine GC content of each region of interest.
e. Summarize these as density plots, etc. A meaningful extension of this exercise might compare the observed GC content to the expected content, where expectation is the product of the independent G and C frequencies.

To calculate the percentage of promoters covered by binding sites, we might

a. Load the reference genome *TxDb* package, *TxDb.Hsapiens.UCSC.hg19.knownGene*.
b. Query the package for promoters using the `promoters` function, or otherwise manipulating exon or transcript coordinates to get a *GRanges* or *GRangesList* representing genomic regions of interest, `groi`.
c. Use `countOverlaps(groi, roi)` to find how many transcription factor binding sites overlap each promoter, and from there use standard *R* functions to tally the number of promoters that have zero overlaps.

[2] http://www.sph.emory.edu/~hwu/teaching/bioc/bios560R.html

Chapter 3

Visualizing Sequence Data

R has some great visualization packages; essential references include [1] for a general introduction, Murrell [4] for base graphics, Sarkar [5] for *lattice*, and Wickham [6] for *ggplot2*. Here we take a quick tour of visualization facilities tailored for sequence data and using *Bioconductor* approaches. Our focus is on *Gviz*; an exercise explores interactive visualization using *shiny*.

3.1 *Gviz*

The *Gviz* package produces very elegant images organized in a more-or-less familiar 'track' format. The following exercises walk through the *Gviz* User guide Section 2.

Exercise 7
Load the Gviz package and sample GRanges containing genomic coordinates of CpG islands. Create a couple of variables with information on the chromosome and genome of the data (how can this information be extracted from the `cpgIslands` *object?).*

```
> library(Gviz)
> data(cpgIslands)
> chr <- "chr7"
> genome <- "hg19"
```

The basic idea is to create a track, perhaps with additional attributes, and to plot it. There are different types of track, and we create these one at a time. We start with a simple annotation track

```
> atrack <- AnnotationTrack(cpgIslands, name="CpG")
> plotTracks(atrack)
```

Then add a track that represents genomic coordinates. Tracks are combined when plotted, as a simple list. The vertical ordering of tracks is determined by their position in the list.

```
> gtrack <- GenomeAxisTrack()
> plotTracks(list(gtrack, atrack))
```

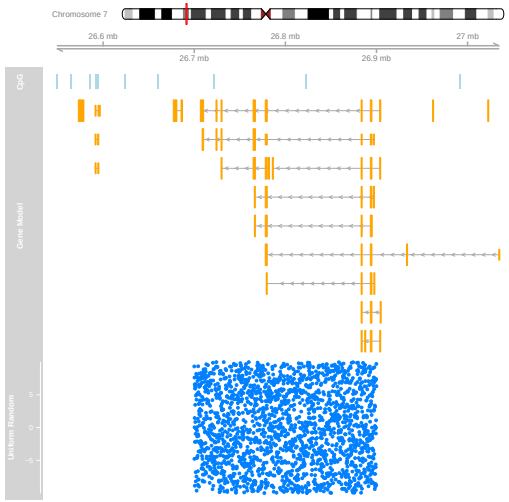
We can add an ideogram to provide overall orientation...

```
> itrack <- IdeogramTrack(genome=genome, chromosome=chr)
> plotTracks(list(itrack, gtrack, atrack))
```

and a more elaborate gene model, as an *data.frame* or *GRanges* object with specific columns of metadata.

```
> data(geneModels)
> grtrack <-
+     GeneRegionTrack(geneModels, genome=genome,
+                     chromosome=chr, name="Gene Model")
> tracks <- list(itrack, gtrack, atrack, grtrack)
> plotTracks(tracks)
```


Figure 3.1: *Gviz* ideogram, genome coordinate, annotation, and data tracks.



Zooming out changes the location box on the ideogram

```
> plotTracks(tracks, from=2.5e7, to=2.8e7)
```

When zoomed in we can add sequence data

```
> library(BSgenome.Hsapiens.UCSC.hg19)
> strack <- SequenceTrack(Hsapiens, chromosome=chr)
> plotTracks(c(tracks, strack), from=26450430, to=26450490, cex=.8)
```

As the *Gviz* vignette humbly says, 'so far we have replicated the features of a whole bunch of other genome browser tools out there'. We'd like to be able integrate our data into these plots, with a rich range of plotting options. The key is the `DataTrack` function, which we demonstrate with some simulated data; this final result is shown in Figure 3.1.

```
> ## some data
> lim <- c(26700000, 26900000)
> coords <- seq(lim[1], lim[2], 101)
> dat <- runif(length(coords) - 1, min=-10, max=10)
> ## DataTrack
> dtrack <-
+     DataTrack(data=dat, start=coords[-length(coords)],
+               end= coords[-1], chromosome=chr, genome=genome,
+               name="Uniform Random")
> plotTracks(c(tracks, dtrack))
```

Section 4.3 of the *Gviz* vignette illustrates flexibility of the data track.

3.2 *ggbio*

The *ggbio* package complements facilities in *Gviz*. It uses the central metaphors of the 'grammar of graphics' made popular by the *ggplot2* package, and integrates closely with the *Bioconductor* ranges

infrastructure. For example, the `autoplot` function applied to a *Rsamtools BamFile* instances results in a coverage plot. In addition to the grammar of graphics approach, the package offers a wider range of plots, for instance circular plots. The use of the package is covered in it's vignette.

3.3 *shiny* for easy interactive reports

As a final example of visualization, the *shiny* package and web site[1] has recently been introduced. It offers a new model for developing interactive, browser-based visualizations. These visualizations could be an excellent way to provide sophisticated exploratory or summary analysis in a very accessible way. The idea is to write a 'user interface' component that describes how a page is to be presented to users, and a 'server' that describes how the data are to be calculated or modified in responses to user choices. The programming model is 'reactive', where changes in a user choice automatically trigger re-calculations in the server. This reactive model is like in a spreadsheet with a formula, where adjusting a cell that the formula references triggers re-calculation of the formula. As with a spreadsheet, someone creating a *shiny* application does not have to work hard to make reactivity work.

Exercise 8
We explore shiny by creating a simple application based on the parathyroidSE package. Start by copying the directory returned by

```
> system.file(package="Morgan2013", "shiny")
```

to a convenient location on your disk, e.g., `/shiny`*. Start R and evaluate the commands*

```
> library(shiny)
> runApp("~/shiny/se-app0")
```

Where the path is correct for the location where you placed the `shiny` *folder. This should launch a web browser, with a simple shiny application.*

Add features to the user interface. Do this by adding commands to the file `/shiny/se-app0/ui.R` *in the* `sidebarPanel` *function. Add: (a) a* `checkboxInput` *to indicate that the user would like to see the experiment data; and (b) a* `checkboxInput` *to indicate that the user would like to see the column (sample) data. Add some output to* `mainPanel`*, in particular* `verbatimTextOutput` *for an overview of a SummarizedExperiment. Use the shiny help pages and* `/shiny/se-app1/ui.R` *as a guide to correct syntax.*

Re-load the browser page of the shiny app to check that the changes you have made are correct.

Add functionality to the `/shiny/se-app0/server.R`*. Start by loading the parathyroidSE data package. Then update the body of* `shinyServer` *by: (a) writing a* `reactive` *code chunk that evaluates* `data(parathyroidGenesSE)` *to load the data in to R, and then returns the* `parathyroidGenesSE` *object; (b) a* `renderPrint` *code chunk that assigns to the* `output` *list the result of* `print`*'ing the object returned by the reactive function; and (c) additional* `renderPrint` *and* `renderTable` *functions to finish the implementation of the server. Consult the shiny package and* `/shiny/se-app1/server` *file for the correct syntax.*

'Kill' the running app (cntrl-C in the R session in which is was created?), and reload the app with `runApp(" /shiny/se-app0")`*.*

Add additional functionality that would help the user to understand the data present in the `parathyroidGenesSE` *instance.*

[1] http://www.rstudio.com/shiny/

Part II

Variants (Advanced)

Chapter 4

Called Variants: Manipulation and Annotation

4.1 Variants

4.1.1 Work flows

The term 'variant' applies to a number of different possible analyses:

- Single nucleotide polymorphism
- Copy number change
- Structural variation
- Long-range interaction

Common steps in a variant analysis work flow include:

- Alignment. requires tools sensitive to variation, e.g., *GSNAP*, *BWA*; *Bowtie* not optimized for this.
- Variant calling, e.g., *GATK*[1] or *VariantTools*.
- Filtering and manipulation.
- Biological context – variant annotation.
- Integrative analysis, e.g., GWAS, genetical genomics, regulatory motifs.

This chapter explores filtering, manipulation and biological context of variants; subsequent chapters delve into variant calling work flows related to tumor / normal pairs (not covered in depth), and to regulatory motifs.

4.1.2 *Bioconductor* software

Selected *Bioconductor* software relevant to DNA-seq work flows are summarized in Table 4.1. The *gmapR* package provides access to the well-respected *GSNAP* aligner. *VariantTools* is an emerging tool that works with aligned reads to call single-sample and sample-specific variants; we will work with *VariantTools* as part of this course. Packages such as *cn.mops* and *exomeCopy* identify copy number variants from high-throughput sequence data.

VariantAnnotation provides facilities for manipulated called variants stored in VCF files; the facilities are very flexible, including simple range-based filtering, look-up in dbSNP, and coding and effect prediction using standard data bases. *VariantAnnotation* plays well with *ensemblVEP* to feed data through the Ensembl Variant Effect Predictor perl script, and to *Bioconductor* facilities for SNP analysis and genetical genomics like *snpStats* and *GGtools*.

[1] http://www.broadinstitute.org/gatk/

Table 4.1: Selected *Bioconductor* packages for DNA-seq analysis.

Package	Description
VariantAnnotation	Manipulating and annotating VCF files
ensemblVEP	Interface to the Ensembl Variant Effect Predictor
VariantTools	Single-sample and tumor specific variant calls
gmapR	Alignment (Linux only)
deepSNV	Sub-clonal SNVs in deep sequencing experiments
cn.mops	Mixture of Poissons copy number variation estimates
exomeCopy	Hidden Markov copy number variation estimates
snpStats	Down-stream GWAS; also *GWAStools*, *GGtools*

Table 4.2: Working with *VCF* files and data.

Category	Function	Description
Read	`scanVcfHeader`	Retrieve file header information
	`scanVcfParam`	Select fields to read in
	`readVcf`	Read VCF file into a VCF class
	`scanVcf`	Read VCF file into a list
	`readInfo`∗	Read a single 'info' column
	`readGeno`∗	Read a single 'geno' matrix
	`readGT`∗	Read 'GT' (genotype) fields as matrix
Filter	`filterVcf`	Filter a VCF from one file to another
Write	`writeVcf`	Write a VCF file to disk
Annotate	`locateVariants`	Identify where variant overlaps a gene annotation
	`predictCoding`	Amino acid changes for variants in coding regions
	`summarizeVariants`	Summarize variant counts by sample
SNPs	`genotypeToSnpMatrix`	Convert genotypes to a SnpMatrix
	`GLtoGP`	Convert genotype likelihoods to genotypes
	`snpSummary`	Counts and distribution statistics for SNPs
Manipulate	`expand`	Convert CompressedVCF to ExpandedVCF
	`cbind`, `rbind`	Combine by column or row

∗: available in the 'devel' version of *Bioconductor*

4.2 *VariantAnnotation* and VCF Files

A major product of DNASeq experiments are catalogs of called variants (e.g., SNPs, indels). We will use the *VariantAnnotation* package to explore this type of data. Sample data included in the package are a subset of chromosome 22 from the 1000 Genomes project. Variant Call Format (VCF, see[2]) text files contain 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.

Important operations on VCF files available with the *VariantAnnotation* package are summarized in Table 4.2.

Exercise 9

The objective of this exercise is to compare the quality of called SNPs that are located in dbSNP, versus those that are novel.

Locate the sample data in the file system. Explore the metadata (information about the content of the file) using `scanVcfHeader`*. Discover the 'info' fields* `VT` *(variant type), and* `RSQ` *(genotype imputation quality).*

Input the sample data using `readVcf`*. You'll need to specify the genome build (*`genome="hg19"`*) on which the variants are annotated. Take a peak at the* `rowData` *to see the genomic locations of each variant.*

Data resources often adopt different naming conventions for sequences. For instance, dbSNP uses

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

abbreviations such as `ch22` to represent chromosome 22, whereas our VCF file uses 22. Use `rowData` and `seqlevels<-` to extract the row data of the variants, and rename the chromosomes.

Solution: Explore the header:

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

```
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
```

```
> info(hdr)[c("VT", "RSQ"),]
```

```
DataFrame with 2 rows and 3 columns
         Number        Type                                        Description
    <character> <character>                                        <character>
VT            1      String indicates what type of variant the line represents
RSQ           1       Float      Genotype imputation quality from MaCH/Thunder
```

Input the data and peak at their locations:

```
> (vcf <- readVcf(fl, "hg19"))
```

```
class: CollapsedVCF
dim: 10376 5
rowData(vcf):
  GRanges with 5 metadata columns: paramRangeID, REF, ALT, QUAL, FILTER
info(vcf):
  DataFrame with 22 columns: LDAF, AVGPOST, RSQ, ERATE, THETA, CIEND, CIPOS,...
info(header(vcf)):
            Number Type    Description
   LDAF     1      Float   MLE Allele Frequency Accounting for LD
   AVGPOST  1      Float   Average posterior probability from MaCH/Thunder
   RSQ      1      Float   Genotype imputation quality from MaCH/Thunder
   ERATE    1      Float   Per-marker Mutation rate from MaCH/Thunder
   THETA    1      Float   Per-marker Transition rate from MaCH/Thunder
   CIEND    2      Integer Confidence interval around END for imprecise va...
   CIPOS    2      Integer Confidence interval around POS for imprecise va...
   END      1      Integer End position of the variant described in this r...
   HOMLEN   .      Integer Length of base pair identical micro-homology at...
   HOMSEQ   .      String  Sequence of base pair identical micro-homology ...
   SVLEN    1      Integer Difference in length between REF and ALT alleles
   SVTYPE   1      String  Type of structural variant
   AC       .      Integer Alternate Allele Count
   AN       1      Integer Total Allele Count
   AA       1      String  Ancestral Allele, ftp://ftp.1000genomes.ebi.ac....
   AF       1      Float   Global Allele Frequency based on AC/AN
 [ reached getOption("max.print") -- omitted 6 rows ]
geno(vcf):
  SimpleList of length 3: GT, DS, GL
geno(header(vcf)):
     Number Type   Description
   GT 1      String Genotype
   DS 1      Float  Genotype dosage from MaCH/Thunder
   GL .      Float  Genotype Likelihoods
```



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

```
GRanges with 3 ranges and 5 metadata columns:
             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                 G       100        PASS
 rs147922003              C                 T       100        PASS
 rs114143073              G                 A       100        PASS
 ---
 seqlengths:
  22
  NA
```

Rename chromosome levels:

```
> seqlevels(vcf, force=TRUE)  <- c("22"="ch22")
```

4.3 Large-scale filtering

This section was developed from initial work by Paul Shannon.

One source of VCF files is from whole-genome sequencing and variant calling; an example data set is from Complete Genomics, and a subset is available in the `BigData` folder accompanying this course

```
> vcfDir <- "~/BigData/CompleteGenomics"
> dir(vcfDir)
> vcfFile <- dir(vcfDir, ".gz$", full=TRUE)
> stopifnot(length(vcfFile) == 1)
```

The data were retrieved from the Complete Genomics web site, the first 750,000 of about 14 million variants selected, and the resulting file compressed and indexed. Indexing makes them accessible to fast queries using the `which` argument to `ScanVcfParam`.

Exercise 10
The objective of this exercise is to filter the larger VCF file to a subset of interesting variants that we might wish to study in depth at a later date. We use the `filterVcf` function of the VariantAnnotation package to perform the filtering.

Start by taking a look at the (complicated) header information

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

We'll be paying attention to the `SS` INFO field, and the `AD` GENO field. Determine the data types and possible values for these fields, using commands like

```
> info(hdr)
> geno(hdr)
```

What data type would you use to represent the `SS` field for a single or several VCF records in R? What about the `AD` field, across all samples?

This VCF file is big. While we can read this into memory all at once, we will often want to 'chunk' through a file, reading many (e.g., a million) records at a time. We do this by creating a TabixFile and specifying a `yieldSize` representing the size of the chunk that we would like to read at each iteration. Create a TabixFile with `yieldSize=100000` and verify with a simple loop that the entire file appears to be processed in chunks of the specified size, along the lines of...

```
> tbx <- TabixFile(vcfFile, yieldSize=10000)
> open(tbx)
> while (len <- nrow(readVcf(tbx, "hg19")))
+     cat("read", len, "rows\n")
```

As you can see from the chunking exercise, it takes quite a bit of time to process these lines. To filter 14 million variants effectively, it can pay to do a cheaper 'pre-filter'. Specifically, we're interested in variants tagged 'Germline'. If we were to represents each VCF record as an element of a character vector `x`*, then we could write a one-liner that returned* `TRUE` *if the line contained the word 'Germline':*

```
> grepl("Germline", x, fixed=TRUE)
```

This would be fast to read in to R, and fast to perform the filter. The `filterVcf` *function allows us to specify a pre-filter that works just like this. The filters are constructed using FilterRules in IRanges, by translating our one-liner into a simple function call, and placing the function call into a list.*

```
> isGermline <- function(x)
+     grepl("Germline", x, fixed=TRUE)
> filters <- FilterRules(list(isGermline=isGermline))
```

The idea is that several filters are chained together. Each filter returns a logical vector indicating the subset of data to be processed by the next filter.

Here is our pre-filter in action:

```
> destination <- tempfile()               # temporary location
> filterVcf(vcfFile, "hg19", destination, prefilters=filters)
```

This is pretty fast, and drops the number of variants under consideration quite substantially, to about 110000.

Our next filter is more challenging to write. We're interested in allelic depth, a summary of the evidence for the variant summarized in the `AD` *GENO field. There are many variants, each sample has two values of AD, and there are two samples. This means that* `AD` *is a three-dimensional array. Our filter criteria is that the ratio of ('alternate allele' of the tumor sample* or *the 'reference allele' of the the normal sample) to total reads is greater than* 0.1*. Here is a function implementing this:*

```
> allelicDepth <- function(x)
+ {
+     ##  ratio of AD of the 'alternate allele' for the tumor sample
+     ##  OR 'reference allele' for normal samples to total reads for
+     ##  the sample should be greater than some threshold (say 0.1,
+     ##  that is: at least 10% of the sample should have the allele
+     ##  of interest)
+     ad <- geno(x)[["AD"]]
+     tumorPct <- ad[,1,2,drop=FALSE] / rowSums(ad[,1,,drop=FALSE])
+     normPct <- ad[,2,1, drop=FALSE] / rowSums(ad[,2,,drop=FALSE])
+     test <- (tumorPct > 0.1) | (normPct > 0.1)
+     !is.na(test) & test
+ }
```

We can add it to our list of filters

```
> filters <- FilterRules(list(isGermline=isGermline,
+                             allelicDepth=allelicDepth))
```

To use this filter, we actually need to fully parse the VCF file into the VCF instance; the pre-filter trick used for germline filtering is not enough. `filterVcf` *allows us to perform a filter on the VCF instance, too, and does so after pre-filtering*

```
> destination <- tempfile()
> filterVcf(vcfFile, "hg19", destination, prefilters=filters[1],
+           filters=filters[2])
```


And finally input our interesting variants, confirming that we've done our filtering as desired.

```
> vcf <- readVcf(destination, "hg19")
> all(info(vcf)$SS == "Germline")
> table(allelicDepth(vcf))
```

4.4 SNP Annotation

This section is derived from a vignette by Valerie Obenchain in the *VariantAnnotation* package.

Variants can be easily identified according to region such as `coding`, `intron`, `intergenic`, `spliceSite` etc. Amino acid coding changes are computed for the non-synonymous variants. SIFT and PolyPhen databases provide predictions of how severely the coding changes affect protein function. Additional annotations are easily crafted using the *GenomicRanges* and *GenomicFeatures* software in conjunction with *Bioconductor* and broader community annotation resources.

Exercise 11

The SNPlocs.Hsapiens.dbSNP.20101109 contains information about SNPs in a particular build of dbSNP. Load the package.

Review the short helper function `isInDbSNP`*, in the Morgan2013 package, to query whether SNPs are in the data base (a version of this function will be introduced to VariantAnnotation before the next release)*

```
> library(Morgan2013)
> isInDbSNP

function (vcf, seqname, rsid = TRUE)
{
    snpLocs <- getSNPlocs(seqname)
    idx <- ((seqnames(vcf) == seqname) & (width(rowData(vcf)) ==
        1L))
    idx <- as.vector(idx)
    snps <- rowData(vcf)[idx]
    result <- rep(NA, nrow(vcf))
    result[idx] <- if (rsid) {
        sub("rs", "", names(snps)) %in% snpLocs[["RefSNP_id"]]
    }
    else {
        start(snps) %in% snpLocs[["loc"]]
    }
    result
}
<environment: namespace:Morgan2013>
```

Create a data frame containing the dbSNP membership status and imputation quality of each SNP. Create a density plot to illustrate the results.

Solution: Discover whether SNPs are located in dbSNP, using our helper function.

```
> library(SNPlocs.Hsapiens.dbSNP.20101109)
> inDbSNP <- .isInDbSNP(vcf, "ch22")
> table(inDbSNP)
```

Create a data frame summarizing SNP quality and dbSNP membership:

```
> metrics <-
+     data.frame(inDbSNP=inDbSNP, RSQ=info(vcf)$RSQ)
```

Finally, visualize the data, e.g., using *ggplot2* (Figure 4.1).

Figure 4.1: Quality scores of variants in dbSNP, compared to those not in dbSNP.

Table 4.3: Variant locations

Location	Details
`coding`	Within a coding region
`fiveUTR`	Within a 5' untranslated region
`threeUTR`	Within a 3' untranslated region
`intron`	Within an intron region
`intergenic`	Not within a transcript associated with a gene
`spliceSite`	Overlaps any of the first or last 2 nucleotides of an intron

```
> library(ggplot2)
> ggplot(metrics, aes(RSQ, fill=inDbSNP)) +
+     geom_density(alpha=0.5) +
+     scale_x_continuous(name="MaCH / Thunder Imputation Quality") +
+     scale_y_continuous(name="Density") +
+     theme(legend.position="top")
```

4.4.1 Variants in and around genes

Variant location with respect to genes can be identified with the `locateVariants` function. Regions are specified in the `region` argument and can be one of the following constructors: `CodingVariants()`, `IntronVariants()`, `FiveUTRVariants()`, `ThreeUTRVariants()`, `IntergenicVariants()`, `SpliceSiteVariants()`, or `AllVariants()`. Location definitions are shown in Table 4.3.

Exercise 12
Load the TxDb.Hsapiens.UCSC.hg19.knownGene annotation package, and read in the `chr22.vcf.gz` example file from the VariantAnnotation package.

Remembering to re-name sequence levels, use the `locateVariants` function to identify coding variants.

Summarize aspects of your data, e.g., did any coding variants match more than one gene? How many coding variants are there per gene ID?

Solution: Here we open the known genes data base, and read in the VCF file.

```
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
> fl <- system.file("extdata", "chr22.vcf.gz", package="VariantAnnotation")
> vcf <- readVcf(fl, "hg19")
> seqlevels(vcf, force=TRUE) <- c("22"="chr22")
```

The next lines locate coding variants.

```
> rd <- rowData(vcf)
> loc <- locateVariants(rd, txdb, CodingVariants())
> head(loc, 3)
```

```
GRanges with 3 ranges and 7 metadata columns:
      seqnames              ranges strand | LOCATION   QUERYID      TXID
         <Rle>           <IRanges>  <Rle> | <factor> <integer> <integer>
  [1]    chr22 [50301422, 50301422]      * |   coding        24     73482
  [2]    chr22 [50301476, 50301476]      * |   coding        25     73482
  [3]    chr22 [50301488, 50301488]      * |   coding        26     73482
         CDSID      GENEID   PRECEDEID    FOLLOWID
      <integer> <character> <character> <character>
```



```
  [1]    217009       79087       <NA>        <NA>
  [2]    217009       79087       <NA>        <NA>
  [3]    217009       79087       <NA>        <NA>
  ---
  seqlengths:
   chr22
      NA
```

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

```
> ## Did any coding variants match more than one gene?
> splt <- split(loc$GENEID, loc$QUERYID)
> table(sapply(splt, function(x) length(unique(x)) > 1))

FALSE  TRUE
  956    15

> ## Summarize the number of coding variants by gene ID
> splt <- split(loc$QUERYID, loc$GENEID)
> head(sapply(splt, function(x) length(unique(x))), 3)

113730   1890  23209
    22     15     30
```

4.4.2 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 `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.

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` object, the alternate alleles are taken from `alt(<VCF>)` 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]
```

```
GRanges with 5 ranges and 17 metadata columns:
             seqnames               ranges strand | paramRangeID
                <Rle>            <IRanges>  <Rle> |     <factor>
  22:50301584    chr22 [50301584, 50301584]      - |         <NA>
                    REF                ALT      QUAL      FILTER
             <DNAStringSet> <DNAStringSetList> <numeric> <character>
  22:50301584              C                 T       100        PASS
               varAllele     CDSLOC    PROTEINLOC   QUERYID        TXID
             <DNAStringSet>  <IRanges> <IntegerList> <integer> <character>
  22:50301584              A [777, 777]           259        28       73482
                CDSID      GENEID   CONSEQUENCE       REFCODON       VARCODON
             <integer> <character>      <factor> <DNAStringSet> <DNAStringSet>
  22:50301584    217009       79087    synonymous            CCG            CCA
                   REFAA         VARAA
             <AAStringSet> <AAStringSet>
```

```
  22:50301584             P             P
 [ reached getOption("max.print") -- omitted 4 rows ]
  ---
  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.

```
> coding[coding$CONSEQUENCE == "frameshift"]

GRanges with 1 range and 17 metadata columns:
             seqnames               ranges strand | paramRangeID
                <Rle>            <IRanges>  <Rle> |     <factor>
  22:50317001    chr22 [50317001, 50317001]      + |         <NA>
                    REF                 ALT      QUAL      FILTER
             <DNAStringSet> <DNAStringSetList> <numeric> <character>
  22:50317001              G              GCACT       233        PASS
                  varAllele     CDSLOC    PROTEINLOC   QUERYID        TXID
             <DNAStringSet>  <IRanges> <IntegerList> <integer> <character>
  22:50317001          GCACT [808, 808]           270       359       72592
                     CDSID      GENEID CONSEQUENCE       REFCODON       VARCODON
             <integer> <character>    <factor> <DNAStringSet> <DNAStringSet>
  22:50317001    214765       79174  frameshift            GCC            GCC
                   REFAA          VARAA
             <AAStringSet> <AAStringSet>
  22:50317001            A
  ---
  seqlengths:
   chr22
      NA
```

4.4.3 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 <- 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), 3)

[1] "rs47"  "rs268" "rs298"

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

 [1] "RSID"        "PROTEINID"   "AACHANGE"    "METHOD"      "AA"
 [6] "PREDICTION"  "SCORE"       "MEDIAN"      "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, 3)

         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
```

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.

4.5 Annotation with *ensemblVEP*

WARNING: This section requires an internet connection and additional software, and has been removed from the lab. As an alternative, consider working through the *ensemblVEP* vignette, available from the *Bioconductor* web site[3]. This section requires that *perl* is installed, and that the Ensembl VEP perl script is available; see instructions in the *ensemblVEP* README[4]. The necessary preconditions are tested by the following code; `VEP_OK` should be `TRUE`.

```
> CONNECTION_OK <- tryCatch({
+     con <- url("http://google.com", "r"); close(con); TRUE
+ }, error=function(...) FALSE)
> VEP_OK <- CONNECTION_OK && require(ensemblVEP) && tryCatch({
+     file.exists(ensemblVEP:::.getVepPath())
+ }, error=function(...) FALSE)
> stopifnot(VEP_OK)
```

Functionality of the Ensembl VEP is given on-line[5] and in [3].

[3] http://bioconductor.org/packages/release/bioc/html/ensemblVEP.html
[4] http://bioconductor.org/packages/release/bioc/readmes/ensemblVEP/README
[5] http://www.ensembl.org/info/docs/variation/vep/

Chapter 5

Structural Variants: Tumor / Normal Pairs

WARNING: This chapter requires Linux and has been removed from the lab. See the *VariantTools* package vignette for a tumor / normal pair work flow.

Chapter 6

Regulatory Variants: BAM to Motif

6.1 Introduction

This work flow was originally constructed by Paul Shannon and Valerie Obenchain.

This portion of the lab represents a work flow from aligned reads through variant calls to consequences for regulatory motif binding sites. The work flow reproduces part of Huang *et al*'s 'Highly recurrent TERT promoter mutations in human melanoma' [2]. Briefly, previous sequencing studies identified mutations in the TERT promoter locus; Huang et al. validated these in additional cell lines, and noted that the mutation introduced a binding site for the E-twenty-six transcription factor. We reproduce some of these results, notably calling variants in the TERT region across cell lines, and matching a data base of binding motifs to the original and modified promoter regions.

If you are not using windows, load the *parallel* package and set the number of cores to use to the number of cores on your machine.

```
> library(parallel); options(mc.cores=detectCores())
```

We'll use annotation resources for humans, specifically the gene-centric *org.Hs.eg.db* for basic information about TERT, the *TxDb.Hsapiens.UCSC.hg19.knownGene* package for information about exon and transcript structure of TERT, and the *BSgenome.Hsapiens.UCSC.hg19* package for sequence information.

```
> library(org.Hs.eg.db)
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> library(BSgenome.Hsapiens.UCSC.hg19)
```

Huang *et al* provide BAM files of reads aligning in the neighborhood of TERT for several cell lines; a subset of these are available for this lab. Create a variable `path` that points to the directory containing these files. For me, the path is

```
> path <- "~/BigData/TERT/bam"
```

As a sanity check, confirm that there are 22 files in this directory, 11 BAM and 11 index files.

```
> stopifnot(length(dir(path, "bam$")) == 11)
> stopifnot(length(dir(path, "bai$")) == 11)
```

A common challenge in real-world analysis is that the sequence names used in one part of a work flow (e.g., alignment) differ from sequence names in another part of the work flow, e.g., annotation. Here the BAM files are aligned using NCBI notation ("7") but the annotations follow UCSC notation ("chr7"). We need to make these consistent, and it's easier to change the behavior of our annotation packages than to change the bam files.

```
> seqnameStyle(BSgenome.Hsapiens.UCSC.hg19) <-
+     seqnameStyle(TxDb.Hsapiens.UCSC.hg19.knownGene) <- "NCBI"
```

It is important that we think carefully before re-naming sequence names – the underlying genome coordinates used for alignment need to match exactly the genome coordinates used for annotation. As a final preparation, let's make some convenient abbreviations for our genome and txdb, and figure out what the Entrez ID is for the gene symbol TERT.

```
> bsgenome <- BSgenome.Hsapiens.UCSC.hg19
> txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
> geneid <- select(org.Hs.eg.db, "TERT", "ENTREZID", "SYMBOL")
```

6.2 Tallying variants

WARNING: This section requires Linux to tally variants with the gmapR package. If you are not using Linux, then read through the steps and load the `called` object at the start of the next section. The necessary precondition to evaluate the code blocks in this section are tested by the following.

```
> VARIANTTOOLS_OK <- local({
+     sys <- !Sys.info()["sysname"] %in% c("Darwin", "Windows")
+     sys && require(VariantTools) && require(gmapR)
+ })
> stopifnot(VARIANTTOOLS_OK)
```

The main goal of this section is to tally variants in our region of interest. We'll use an algorithm implemented in the `callVariants` function of the *VariantTools* package. This in turn relies on the *gmapR* package, which provides an *R* interface to the GMAP aligner, *gmapR* needs a reference genome to do its work. A neat feature is that the reference genome can be created on the fly from arbitrary nucleotide sequences.

TERT is located on chromosome 5 (how would we discover that?) so we make an indexed mini-reference genome of chr5. We get the coordinates and actual sequence for this chromosome from the *BSgenome* package.

```
> roi <- GRanges("5", IRanges(1, seqlengths(bsgenome)[["5"]]))
> chr5seq <- getSeq(bsgenome, roi)
> names(chr5seq) <- "5"
```

and then make the reference genome in *gmapR* (*gmapR* maintains a cache of sequences, so this is only 'expensive' the first time; it is important to use a descriptive name, because this is the key used to retrieve a previously built genome from the cache).

```
> library(gmapR)
> genome.5 <- GmapGenome(chr5seq, name="hg19_5", create=TRUE)
```

We want to call variants in the promoter region of TERT, so query the `txdb` for transcripts grouped by gene, and then select the TERT gene from the result.

```
> rng <- transcriptsBy(txdb, "gene")[[geneid$ENTREZID]]
```

There are three unique start sites.

```
> unique(start(rng))
```

```
[1] 1253287 1255402 1278756
```

For simplicity we treat the gene as a single range (using `range`), use an *ad hoc* definition of promoter as 330 nt upstream of the transcription start site, and define the appropriate ranges using `promoter`. We restrict the seqlevels to just chromosome five (seqlevels are acting in some ways like factor levels, they persist even if there are no elements with the corresponding level).

```
> unique(start(rng))
```

```
[1] 1253287 1255402 1278756
```



```
> pregion <- promoters(range(rng), upstream=330, downstream=0)
> seqlevels(pregion) <- "5"
```

Our objective is to use the *VariantTools* package to call variants. Variant calling is influenced by parameters specified in creation of a *VariantTallyParam* object; details of the arguments to this function, and how they influence variant calling, are present on the `VariantTallyParam` help page and in the *VariantTools* vignette.

```
> library(VariantTools)
> vtparam <- VariantTallyParam(genome.5, readlen=101L,
+     which=pregion, indels=TRUE)
```

The next step is to call variants. The basic approach is to invoked the `callVariants` function on each bam file, using our `vtparam` object to influence how variants are called.

Exercise 13
(Linux only) Consult section 2.2 of the VariantTools vignette for a brief description of how variants are called.

The following two lines calls variants in each of our files.

```
> fls <- dir(path, "bam$", full=TRUE)
> called <- lapply(fls, callVariants, tally.param=vtparam)
```

The results can be cleaned up a little. We currently have one `GRanges` object for each BAM file, but it's convenient to create a single `GRanges` object, adding a column ('id') to remember which range came from which file.

```
> len <- elementLengths(called)
> called <- do.call(c, called)
> id <- sub(".TERT.bam", "", basename(fls))
> called$id <- factor(rep(id, len), levels=unique(id))
```

We'll explore the `called` object in the next section.

6.3 Manipulation

Here we pick up from the previous section, loading a saved version of the data those of us on Linux were able to create.

```
> data(called, package="Morgan2013")
```

Exercise 14
What class of object is `called`? How many rows does it have? How many metadata columns?

Solution: The object is a *GRanges* object with length 88 and 25 metadata columns.

Each row represents a genomic location for which some evidence of a variant was discovered. The `ref` and `alt` columns contain the reference and alternate sequence observed at that location. From the *VariantTools* package, some of the important columns are:

```
 ncycles: The number of unique cycles at which the alternate allele was
          observed, 'NA' for the reference allele row.
   count: The number of reads with the alternate allele, 'NA' for the
          reference allele row.
high.quality: The number of reads for the alternate allele that were
          above 'high_quality_cutoff', 'NA' for the reference allele
          row.
mean.quality: The mean mapping quality for the alternate allele, 'NA'
          for the reference allele row.
```

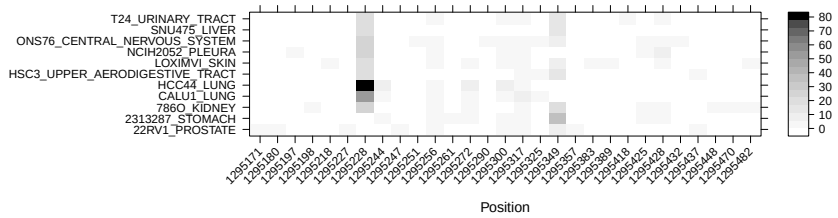


Figure 6.1: Called variants across samples in regulatory regions of TERT

```
count.pos: The number of positive strand reads for the alternate
          allele, 'NA' for the reference allele row.
cycleCount: An additional column is present for each bin formed by the
          'cycle_breaks' parameter, with the read count for that bin.
```

Additional columns are also appended with '.ref', '.total', '.pos', '.neg', etc., to indicate reference, total, positive strand and negative strand. The idea is that the metadata columns are providing further guidance on the nature of evidence supporting each called variant. It is useful to get a sense of the evidence supporting the called variants by exploring `called` using standard *R* commands, e.g., plotting `count` and `count.ref` columns against one another. The called variants depend critically on parameters used in `VariantTallyParam`, so we would want to spend some time giving calling and quality assessment parameters careful consideration.

A quick assessment of called variants, sufficient for reproducing results in [2], can be obtained by plotting the number of reads supporting each call, across samples. We focus on the highest-quality read cycles 10 to 91. The result is shown in Figure 6.1.

```
> library(lattice)
> altCounts <-
+     xtabs(cycleCount.10.91 ~ start(called) + id, mcols(called))
> plt <- levelplot(altCounts, xlab="Position", ylab=NULL,
+           scales=list(x=list(rot=45)), aspect="fill",
+           col.regions=rev(gray.colors(100, 0, 1)))
```

6.4 Regulatory signatures

Supposing that we have identified, as Huang *et al.* did, a mutation occurring at position 1295228 across samples or cell lines. How can we relate this variant to regulatory signature?

We start by extracting the sequence surrounding the SNP in the reference. We do this by creating a `GRanges` instance at the SNP, then selecting flanking sequence of say 10 nt.

```
> snp <- GRanges("5", IRanges(1295228, width=1))
> snp <- flank(snp, 10, both=TRUE)
```

We then retrieve the sequence from a data source, e.g., the `BSgenome`, `gmapR`, or *Rsamtools*' *FaFile* on which alignments were based.

```
> refSeq <- getSeq(bsgenome, snp)
```

Alternate sequences are a little more difficult; the helper function `variantSequences` has been defined to do the work. It takes the called variants, our SNP regions of interest, and the reference sequence as arguments. It subsets our called variants to include only those that overlap our region of interest using `subsetByOverlaps`. It then injects the variants into the reference sequence using `replaceLetterAt`.


```
> library(Morgan2013)
> variantSequences

function (x, roi, refSeq, ...)
{
    stopifnot(length(roi) == 1L)
    x <- subsetByOverlaps(x, roi)
    at <- split(start(x) - start(roi) + 1L, x$id)
    alt <- split(x$alt, x$id)
    alts <- Map(replaceLetterAt, at = at, letter = alt, refSeq)
    DNAStringSet(alts)
}
<environment: namespace:Morgan2013>

> altSeq <- variantSequences(called, snp, refSeq)
```

We can perform various operations on the alt sequences; here we suppose a common signal and create a consensus string.

```
> altConsensus <- DNAStringSet(consensusString(altSeq))
```

We can see the C228T transition in the middle of a DNAStringSet containing first the (complements) of the reference and consensus alternate sequence.

```
> complement(c(refSeq, altConsensus))

  A DNAStringSet instance of length 2
    width seq
[1]    20 TCCCGGGCCTCCCCCGACCC
[2]    20 TCCCGGGCCTTCCCCGACCC
```

The *MotifDb* package contains a data base of protein binding motifs. Here we load the data base and query it for all motifs present in humans and derived from the well-respected JASPAR core[1] data set.

```
> library(MotifDb)
> idx <- with(mcols(MotifDb),
+             organism=="Hsapiens" & dataSource == "JASPAR_CORE")
> jasparHumanPWMs <- MotifDb[idx]
```

We now run all motifs over the regulatory region, searching for possible similarity. We do this for the the reference sequence and for the alternative consensus.

```
> ## matchPWMs: helper in Morgan2013
> minScore = "90%"             # high min. matching score: 1 base change
> (refHits <- matchPWMs(jasparHumanPWMs, reverseComplement(refSeq)[[1]],
+                       minScore))

DataFrame with 1 row and 5 columns
                              PWM     score     start       end
                         <character> <numeric> <integer> <integer>
1 Hsapiens-JASPAR_CORE-SP1-MA0079.2  7.142857         1        10
            seq
  <DNAStringSet>
1     CCCAGCCCCC

> (altHits <- matchPWMs(jasparHumanPWMs, reverseComplement(altConsensus)[[1]],
+                       minScore))
```

[1] http://jaspar.cgb.ki.se/

```
DataFrame with 1 row and 5 columns
                                PWM     score     start       end
                          <character> <numeric> <integer> <integer>
1 Hsapiens-JASPAR_CORE-ETS1-MA0098.1     4.325         9        14
            seq
  <DNAStringSet>
1         CTTCCG
```

Comparison of results shows that the consensus sequence contains a novel high-scoring binding site for ETS1, as reported in Huang et al.

6.5 Conclusions

This work flow has provided brief tour through key steps in calling regulatory variants, using previous results by Huang et al., to motivate identification of a novel binding motif associated with introduction of a called variant. Major components of this work flow include the *VariantTools*, *Biostrings*, and *MotifDb* packages.

References

[1] W. Chang. *R Graphics Cookbook*. O'Reilly Media, Incorporated, 2012.

[2] F. W. Huang, E. Hodis, M. J. Xu, G. V. Kryukov, L. Chin, and L. A. Garraway. Highly recurrent tert promoter mutations in human melanoma. *Science*, 339(6122):957–959, 2013.

[3] W. McLaren, B. Pritchard, D. Rios, Y. Chen, P. Flicek, and F. Cunningham. Deriving the consequences of genomic variants with the ensembl api and snp effect predictor. *Bioinformatics*, 26(16):2069–2070, 2010.

[4] P. Murrell. *R graphics*. Chapman & Hall/CRC, 2005.

[5] D. Sarkar. *Lattice: multivariate data visualization with R*. Springer, 2008.

[6] H. Wickham. *ggplot2: elegant graphics for data analysis*. Springer Publishing Company, Incorporated, 2009.