useR2012!: High-throughput sequence analysis with *R* and *Bioconductor*

Valerie Obenchain, Martin Morgan[*]

June 2012

Abstract

DNA sequence analysis generates large volumes of data presenting challenging bioinformatic and statistical problems. This tutorial introduces *Bioconductor* packages and work flows for the analysis of sequence data. We learn about approaches for efficiently manipulating sequences and alignments, and introduce common work flows and the unique statistical challenges associated with 'RNA-seq', 'ChIP-seq' and variant annotation experiments. The emphasis is on exploratory analysis, and the analysis of designed experiments. The workshop assumes an intermediate level of familiarity with R, and basic understanding of biological and technological aspects of high-throughput sequence analysis. The workshop emphasizes orientation within the *Bioconductor* milieu; we will touch on the *Biostrings*, *ShortRead*, *GenomicRanges*, *edgeR*, and *DiffBind*, and *VariantAnnotation* packages, with short exercises to illustrate the functionality of each package. Participants should come prepared with a modern laptop with current *R* installed.

Contents

[*]mtmorgan@fhcrc.org

Table 1: Schedule.

Introduction	*Rstudio* and the AMI; *R* packages and help; *Bioconductor*.
Sequences and ranges	Representing and manipulating biological sequences; working with genomic coordinates.
Reads and alignments	High-throughput sequence reads (fastq files) and their aligned representation (bam files).
RNA-seq	A post-alignment workflow for differential representation.
ChIP-seq	Working with multiple ChIP-seq experiments
Annotation	Resources for annotation of gene and genomes; working the variants (VCF files).

1 Introduction

This workshop introduces use of *R* and *Bioconductor* for analysis of high-throughput sequence data. The workshop is structured as a series of short remarks followed by group exercises. The exercises explore the diversity of tasks for which *R* / *Bioconductor* are appropriate, but are far from comprehensive.

The goals of the workshop are to: (1) develop familiarity with *R* / *Bioconductor* software for high-throughput analysis; (2) expose key statistical issues in the analysis of sequence data; and (3) provide inspiration and a framework for further independent exploration. An approximate schedule is shown in Table 1.

1.1 *Rstudio*

Exercise 1
Log on to the Rstudio account set up for this course.

Visit the 'Packages' tab in the lower right panel, find the useR2012 package, and discover the vignette (i.e., this document).

Under the 'Files' tab, figure out how to upload and download (small) files to the server.

1.2 *R*

The following should be familiar to you. *R* has a number of standard data types that represent `integer`, `numeric` (floating point), `complex`, `character`, `logical` (boolean), and `raw` (byte) data. It is possible to convert between data types, and to discover the class (e.g., `class`) of a variable. All of the vectors mentioned so far are homogenous, consisting of a single type of element. A `list` can contain a collection of different types of elements and, like all vectors, these elements can be named to create a key-value association. A `data.frame` is a list of equal-length vectors, representing a rectangular data structure not unlike a spread sheet. Each column of the data frame is a vector, so data types must be homogenous within a column. A `data.frame` can be subset by row or column, and columns can be accessed with `$` or `[[`. A `matrix` is also a rectangular data structure, but subject to the constraint that all elements are the same type.

R has *object-oriented* ways of representing complicated data objects; *Bioconductor* makes extensive use of 'S4' objects. Objects are often created by functions (e.g., `GRanges`, below) with parts of the object extracted or assigned using *accessor* functions. Many operations on classes are implemented as *methods* that specialize a *generic* function for the particular class of objects used to invoke the function. For instance, `countOverlaps` is a generic that counts the number of times elements of its `query` argument overlaps elements of its `subject`; there are methods with slightly different behaviors when the arguments are *IRange* instances or *GRanges* instances (in the latter case, the `countOverlaps` method pays attention to whether ranges are on the same strand and chromosome, for instance).

Packages provide functionality beyond that available in base *R*. There are more than 500 *Bioconductor* packages. Packages are contributed by diverse members of the community; they vary in quality (many are excellent) and sometimes contain idiosyncratic aspects to their implementation. New packages (e.g., *ShortRead*, *VariantAnnotation* packages and their dependencies) can be added to an *R* installation using

```
> source("http://bioconductor.org/biocLite.R")
> biocLite(c("ShortRead", "VariantAnnotation")) # new packages
> biocLite(character())                         # update packages
```

`install.packages`. A package is installed only once per *R* installation, but needs to be loaded (with `library`) in each session in which it is used.

Find help using the *R* help system. Start a web browser with

```
> help.start()
```

The 'Search Engine and Keywords' link is helpful in day-to-day use. Use manual pages to find detailed descriptions of the arguments and return values of functions, and the structure and methods of classes. Find help within an *R* session using `?`; the package defining the help page must have been loaded (with `library`).

```
> library(ShortRead)
> ?readFastq
```

S4 classes and generics can be discovered with syntax like the following for the `complement` generic in the *Biostrings* package:

```
> library(Biostrings)
> showMethods(complement)
```

```
Function: complement (package Biostrings)
x="DNAString"
x="DNAStringSet"
x="MaskedDNAString"
x="MaskedRNAString"
x="RNAString"
x="RNAStringSet"
x="XStringViews"
```

Methods defined on the `DNAStringSet` class of *Biostrings* can be found with

```
> showMethods(class="DNAStringSet", where=getNamespace("Biostrings"))
```

Obtaining help on S4 classes and methods requires syntax such as

```
> class ? DNAStringSet
> method ? "complement,DNAStringSet"
```

Vignettes, especially in *Bioconductor* packages, provide an extensive narrative describing overall package functionality. Use

```
> vignette(package="useR2012")
```

to see a list of vignettes available in the *useR2012* package. Vignettes usually consist of text with embedded *R* code, a form of literate programming. The vignette can be read as a PDF document, while the *R* source code is present as a script file ending with extension `.R`. The script file can be sourced or copied into an *R* session to evaluate exactly the commands used in the vignette.

1.3 *Bioconductor*

Bioconductor is a collection of *R* packages for the analysis and comprehension of high-throughput genomic data. *Bioconductor* started more than 10 years ago. It gained credibility for its statistically rigorous approach to microarray pre-preprocessing and analysis of designed experiments, and integrative and reproducible approaches to bioinformatic tasks. There are now more than 500 *Bioconductor* packages for expression and other microarrays, sequence analysis, flow cytometry, imaging, and other domains. The *Bioconductor* web site provides installation, package repository, help, and other documentation.

The *Bioconductor* web site is at `bioconductor.org`. Features include:

- Introductory work flows.
- A manifest of *Bioconductor* packages arranged in BiocViews.
- Annotation (data bases of relevant genomic information, e.g., Entrez gene ids in model organisms, KEGG pathways) and experiment data (containing relatively comprehensive data sets and their analysis) packages.
- Mailing lists, including searchable archives, as the primary source of help.
- Course and conference information, including extensive reference material.
- General information about the project.
- Package developer resources, including guidelines for creating and submitting new packages.

Table 2: Selected *Bioconductor* packages for high-throughput sequence analysis.

Concept	Packages
Data representation	*IRanges*, *GenomicRanges*, *GenomicFeatures*, *Biostrings*, *BSgenome*, *girafe*.
Input / output	*ShortRead* (fastq), *Rsamtools* (bam), *rtracklayer* (gff, wig, bed), *VariantAnnotation* (vcf), *R453Plus1Toolbox* (454).
Annotation	*GenomicFeatures*, *ChIPpeakAnno*, *VariantAnnotation*.
Alignment	*Rsubread*, *Biostrings*.
Visualization	*ggbio*, *Gviz*.
Quality assessment	*qrqc*, *seqbias*, *ReQON*, *htSeqTools*, *TEQC*, *Rolexa*, *ShortRead*.
RNA-seq	*BitSeq*, *cqn*, *cummeRbund*, *DESeq*, *DEXSeq*, *EDASeq*, *edgeR*, *gage*, *goseq*, *iASeq*, *tweeDEseq*.
ChIP-seq, etc.	*BayesPeak*, *baySeq*, *ChIPpeakAnno*, *chipseq*, *ChIPseqR*, *ChIPsim*, *CSAR*, *DiffBind*, *MEDIPS*, *mosaics*, *NarrowPeaks*, *nucleR*, *PICS*, *PING*, *REDseq*, *Repitools*, *TSSi*.
Motifs	*BCRANK*, *cosmo*, *cosmoGUI*, *MotIV*, *seqLogo*, *rGADEM*.
3C, etc.	*HiTC*, *r3Cseq*.
Copy number	*cn.mops*, *CNAnorm*, *exomeCopy*, *seqmentSeq*.
Microbiome	*phyloseq*, *DirichletMultinomial*, *clstutils*, *manta*, *mcaGUI*.
Work flows	*ArrayExpressHTS*, *Genominator*, *easyRNASeq*, *oneChannelGUI*, *rnaSeqMap*.
Database	*SRAdb*.

High-throughput sequence analysis Table 2 enumerates many of the packages available for sequence analysis. The table includes packages for representing sequence-related data (e.g., *GenomicRanges*, *Biostrings*), as well as domain-specific analysis such as RNA-seq (e.g., *edgeR*, *DEXSeq*), ChIP-seq (e.g,. *ChIPpeakAnno*, *DiffBind*), and SNPs and copy number variation (e.g., *genoset*, *ggtools*, *VariantAnnotation*).

Exercise 2
Scavenger hunt. Spend five minutes tracking down the following information.

a. *The package containing the* `readFastq` *function.*

b. *The author of the* `alphabetFrequency` *function, defined in the* Biostrings *package.*

c. *A description of the GappedAlignments class.*

d. *The number of vignettes in the GenomicRanges package.*

e. *From the Bioconductor web site, instructions for installing or updating Bioconductor packages.*

f. *A list of all packages in the current release of Bioconductor.*

g. *The URL of the Bioconductor mailing list subscription page.*

Solution: Possible solutions are found with the following *R* commands

```
> ??readFastq
> library(Biostrings)
> ?alphabetFrequency
> class?GappedAlignments
> vignette(package="GenomicRanges")
```

and by visiting the *Bioconductor* web site, e.g., installation instructions[1] current software packages[2], and mailing lists[3].

1.4 Resources

Dalgaard [4] provides an introduction to statistical analysis with *R*. Matloff [11] introduces *R* programming concepts. Chambers [3] provides more advanced insights into *R*. Gentleman [5] emphasizes use of *R* for bioinformatic programming tasks. The *R* web site enumerates additional publications from the user community.

[1]`http://bioconductor.org/install/`
[2]`http://bioconductor.org/packages/release/bioc/`
[3]`http://bioconductor.org/help/mailing-list/`

Table 3: Selected *Bioconductor* packages for representing and manipulating ranges, strings, and other data structures.

Package	Description
IRanges	Defines important classes (e.g., *IRanges*, *Rle*) and methods (e.g., `findOverlaps`, `countOverlaps`) for representing and manipulating ranges of consecutive values. Also introduces *DataFrame*, *SimpleList* and other classes tailored to representing very large data.
GenomicRanges	Range-based classes tailored to sequence representation (e.g., *GRanges*, *GRangesList*), with information about strand and sequence name.
GenomicFeatures	Foundation for manipulating data bases of genomic ranges, e.g., representing coordinates and organization of exons and transcripts of known genes.
Biostrings	Classes (e.g., *DNAStringSet*) and methods (e.g., `alphabetFrequency`, `pairwiseAlignment`) for representing and manipulating DNA and other biological sequences.
BSgenome	Representation and manipulation of large (e.g., whole-genome) sequences.

2 Sequences and Ranges

Many *Bioconductor* packages are available for analysis of high-throughput sequence data. This section introduces two essential ways in which sequence data are manipulated. Sets of DNA strings represent the reads themselves and the nucleotide sequence of reference genomes. Ranges describe both aligned reads and features of interest on the genome. Key packages are summarized in Table 3.

2.1 *Biostrings*

The *Biostrings* package provides tools for working with DNA (and other biological) sequence data. The essential data structures are *DNAString* and *DNAStringSet*, for working with one or multiple DNA sequences. The *Biostrings* package contains additional classes for representing amino acid and general biological strings. The *BSgenome* and related packages (e.g., *BSgenome.Dmelanogaster.UCSC.dm3*) are used to represent whole-genome sequences. The following exercise explores these packages.

Fixme: Exercise: trimLRPattern

2.2 *GenomicRanges*

Next-generation sequencing data consists of a large number of short reads. These are, typically, aligned to a reference genome. Basic operations are performed on the alignment, asking e.g., how many reads are aligned in a genomic range defined by nucleotide coordinates (e.g., in the exons of a gene), or how many nucleotides from all the aligned reads cover a set of genomic coordinates. How is this type of data, the aligned reads and the reference genome, to be represented in *R* in a way that allows for effective computation?

The *IRanges*, *GenomicRanges*, and *GenomicFeatures* *Bioconductor* packages provide the essential infrastructure for these operations; we start with the *GRanges* class, defined in *GenomicRanges*.

GRanges Instances of *GRanges* are used to specify genomic coordinates. Suppose we wish to represent two *D. melanogaster* genes. The first is located on the positive strand of chromosome 3R, from position 19967117 to 19973212. The second is on the minus strand of the X chromosome, with 'left-most' base at 18962306, and right-most base at 18962925. The coordinates are *1-based* (i.e., the first nucleotide on a chromosome is numbered 1, rather than 0), *left-most* (i.e., reads on the minus strand are defined to 'start' at the left-most coordinate, rather than the 5' coordinate), and *closed* (the start and end coordinates are included in the range; a range with identical start and end coordinates has width 1, a 0-width range is represented by the special construct where the end coordinate is one less than the start coordinate).

A complete definition of these genes as *GRanges* is:

```
> genes <- GRanges(seqnames=c("3R", "X"),
+                  ranges=IRanges(
+                      start=c(19967117, 18962306),
+                      end=c(19973212, 18962925)),
+                  strand=c("+", "-"),
+                  seqlengths=c(`3R`=27905053L, `X`=22422827L))
```

The components of a *GRanges* object are defined as vectors, e.g., of `seqnames`, much as one would define a *data.frame*. The start and end coordinates are grouped into an *IRanges* instance. The optional `seqlengths` argument specifies the maximum size of each sequence, in this case the lengths of chromosomes 3R and X in the 'dm2' build of *D. melanogaster* genome. This data is displayed as

```
> genes
```

```
GRanges with 2 ranges and 0 elementMetadata cols:
      seqnames                 ranges strand
         <Rle>              <IRanges>  <Rle>
  [1]       3R [19967117, 19973212]      +
  [2]        X [18962306, 18962925]      -
  ---
  seqlengths:
         3R        X
   27905053 22422827
```

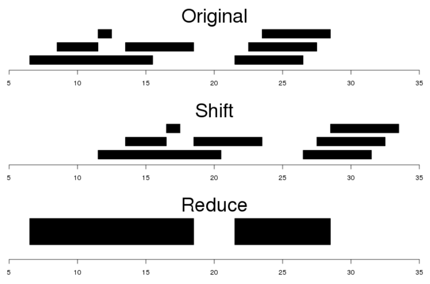

Figure 1: Ranges

For the curious, the gene coordinates and sequence lengths are derived from the *org.Dm.eg.db* package for genes with Flybase identifiers FBgn0039155 and FBgn0085359, using the annotation facilities described in section 6.

The *GRanges* class has many useful methods defined on it. Consult the help page

```
> ?GRanges
```

and package vignettes (especially 'An Introduction to *GenomicRanges*')

```
> vignette(package="GenomicRanges")
```

for a comprehensive introduction.

Operations on ranges The *GRanges* class has many useful methods from the *IRanges* class; some of these methods are illustrated here. We use *IRanges* to illustrate these operations to avoid complexities associated with strand and seqname, but the operations are comparable on *GRanges*. We begin with a simple set of ranges:

```
> ir <- IRanges(start=c(7, 9, 12, 14, 22:24),
+                end=c(15, 11, 12, 18, 26, 27, 28))
```

These and some common operations are illustrated in the upper panel of Figure 1 and summarized in Table 4.

`elementMetadata` (`values`) The *GRanges* class (actually, most of the data structures defined or extending those in the *IRanges* package) has two additional very useful data components. The `elementMetadata` function (or its synonym `values`) allows information on each range to be stored and manipulated (e.g., subset) along with the *GRanges* instance. The element metadata is represented as a *DataFrame*, defined in *IRanges* and acting like a standard *R data.frame* but with the ability to hold more complicated data structures as columns (and with element metadata of its own, providing an enhanced alternative to the *Biobase* class *AnnotatedDataFrame*).

Table 4: Common operations on *IRanges*, *GRanges* and *GRangesList*.

Category	Function	Description
Accessors	`start`, `end`, `width`	Get or s et the starts, ends and widths
	`names`	Get or set the names
	`elementMetadata`, `metadata`	Get or set metadata on elements or object
	`length`	Number of ranges in the vector
	`range`	Range formed from min start and max end
Ordering	`<`, `<=`, `>`, `>=`, `==`, `!=`	Compare ranges, ordering by start then width
	`sort`, `order`, `rank`	Sort by the ordering
	`duplicated`	Find ranges with multiple instances
	`unique`	Find unique instances, removing duplicates
Arithmetic	`r + x`, `r - x`, `r * x`	Shrink or expand ranges `r` by number `x`
	`shift`	Move the ranges by specified amount
	`resize`	Change width, ancoring on start, end or mid
	`distance`	Separation between ranges (closest endpoints)
	`restrict`	Clamp ranges to within some start and end
	`flank`	Generate adjacent regions on start or end
Set operations	`reduce`	Merge overlapping and adjacent ranges
	`intersect`, `union`, `setdiff`	Set operations on reduced ranges
	`pintersect`, `punion`, `psetdiff`	Parallel set operations, on each `x[i]`, `y[i]`
	`gaps`, `pgap`	Find regions not covered by reduced ranges
	`disjoin`	Ranges formed from union of endpoints
Overlaps	`findOverlaps`	Find all overlaps for each `x` in `y`
	`countOverlaps`	Count overlaps of each `x` range in `y`
	`nearest`	Find nearest neighbors (closest endpoints)
	`precede`, `follow`	Find nearest `y` that `x` precedes or follows
	`x %in% y`	Find ranges in `x` that overlap range in `y`
Coverage	`coverage`	Count ranges covering each position
Extraction	`r[i]`	Get or set by logical or numeric index
	`r[[i]]`	Get integer sequence from `start[i]` to `end[i]`
	`subsetByOverlaps`	Subset `x` for those that overlap in `y`
	`head`, `tail`, `rev`, `rep`	Conventional R semantics
Split, combine	`split`	Split ranges by a factor into a *RangesList*
	`c`	Concatenate two or more range objects

```
> elementMetadata(genes) <-
+     DataFrame(EntrezId=c("42865", "2768869"),
+               Symbol=c("kal-1", "CG34330"))
```

metadata allows addition of information to the entire object. The information is in the form of a list; any data can be provided.

```
> metadata(genes) <-
+     list(CreatedBy="A. User", Date=date())
```

GRangesList The GRanges class is extremely useful for representing simple ranges. Some next-generation sequence data and genomic features are more hierarchically structured. A gene may be represented by seven exons within it. An aligned read may be represented by discontinuous ranges of alignment to a reference. The *GRangesList* class represents this type of information. It is a list-like data structure, which each element of the list itself a *GRanges* instance. The gene FBgn0039155 contains several exons, and can be represented as a list of length 1, where the element of the list contains a *GRanges* object with 7 elements:

```
GRangesList of length 1:
$FBgn0039155
GRanges with 7 ranges and 2 elementMetadata cols:
      seqnames               ranges strand |   exon_id   exon_name
         <Rle>            <IRanges>  <Rle> | <integer> <character>
  [1]    chr3R [19967117, 19967382]      + |     64137        <NA>
  [2]    chr3R [19970915, 19971592]      + |     64138        <NA>
  [3]    chr3R [19971652, 19971770]      + |     64139        <NA>
  [4]    chr3R [19971831, 19972024]      + |     64140        <NA>
  [5]    chr3R [19972088, 19972461]      + |     64141        <NA>
  [6]    chr3R [19972523, 19972589]      + |     64142        <NA>
  [7]    chr3R [19972918, 19973212]      + |     64143        <NA>

---
seqlengths:
    chr3R
 27905053
```

The *GenomicFeatures* package Many public resources provide annotations about genomic features. For instance, the UCSC genome browser maintains the 'knownGene' track of established exons, transcripts, and coding sequences of many model organisms. The *GenomicFeatures* package provides a way to retrieve, save, and query these resources. The underlying representation is as sqlite data bases, but the data are available in *R* as *GRangesList* objects. The following exercise explores the *GenomicFeatures* package and some of the functionality for the *IRanges* family introduced above.

Exercise 3
Load the TxDb.Dmelanogaster.UCSC.dm3.ensGene annotation package, and create an alias `txdb` pointing to the TranscriptDb object this class defines.

Extract all exon coordinates, organized by gene, using `exonsBy`. What is the class of this object? How many elements are in the object? What does each element correspond to? And the elements of each element? Use `elementLengths` and `table` to summarize the number of exons in each gene, for instance, how many single-exon genes are there?

Select just those elements corresponding to flybase gene ids FBgn0002183, FBgn0003360, FBgn0025111, and FBgn0036449. Use `reduce` to simplify gene models, so that exons that overlap are considered 'the same'.

Solution:

```
> library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
> txdb <- TxDb.Dmelanogaster.UCSC.dm3.ensGene # alias
> ex0 <- exonsBy(txdb, "gene")
> head(table(elementLengths(ex0)))

   1    2    3    4    5    6
3182 2608 2070 1628 1133  886

> ids <- c("FBgn0002183", "FBgn0003360", "FBgn0025111", "FBgn0036449")
> ex <- reduce(ex0[ids])
```

Exercise 4
The objective of this exercise is to calculate the GC content of the exons of a single gene, whose coordinates are specified by the `ex` object of the previous exercise.

Load the BSgenome.Dmelanogaster.UCSC.dm3 data package, containing the UCSC representation of D. melanogaster *genome assembly dm3.*

Extract the sequence name of the first gene of `ex`. Use this to load the appropriate D. melanogaster *chromosome.*

Use `Views` to create views on to the chromosome that span the start and end coordinates of all exons.

The useR2012 package defines a helper function `gcFunction` to calculate GC content. Use this to calculate the GC content in each of the exons.

Look at the helper function, and describe what it does.

Solution: Here we load the *D. melanogaster* genome, select a single chromosome, and create `Views` that reflect the ranges of the `FBgn0002183`.

```
> library(BSgenome.Dmelanogaster.UCSC.dm3)
> nm <- as.character(unique(seqnames(ex[[1]])))
> chr <- Dmelanogaster[[nm]]
> v <- Views(chr, start=start(ex[[1]]), end=end(ex[[1]]))
```

Using the `gcFunction` helper function, the subject GC content is

```
> gcFunction(v)
```

```
[1] 0.4767442 0.5555556 0.5389610 0.5513308 0.5351788 0.5441426 0.4933333
[8] 0.5189394 0.5110132
```

The `gcFunction` is really straight-forward: it invokes the function `alphabetFrequency` from the *Biostrings* package. This returns a simple matrix of exon × nuclotiede probabilities. The row sums of the `G` and `C` columns of this matrix are the GC contents of each exon.

```
> gcFunction
```

```
function (x)
{
    alf <- alphabetFrequency(x, as.prob = TRUE)
    rowSums(alf[, c("G", "C")])
}
<environment: namespace:useR2012>
```

2.3 Resources

There are extensive vignettes for *Biostrings* and *GenomicRanges* packages. A useful online resource is from Thomas Grike's group.

Table 5: Selected *Bioconductor* packages for sequence reads and alignments.

Package	Description
ShortRead	Defines the *ShortReadQ* class and functions for manipulating `fastq` files; these classes rely heavily on *Biostrings*.
GenomicRanges	*GappedAlignments* and *GappedAlignmentPairs* store single- and paired-end aligned reads.
Rsamtools	Provides access to BAM alignment and other large sequence-related files.
rtracklayer	Input and output of `bed`, `wig` and similar files

3 Reads and Alignments

The following sections introduce core tools for working with high-throughput sequence data; key packages for representating reads and alignments are summarized in Table 5. This section focus on the reads and alignments that are the raw material for analysis. Section 6 introduces resources for annotating sequences, while section 4 addresses statistical approaches to assessing differential representation in RNA-seq experiments. Section 5 outlines ChIP-seq analysis.

3.1 The *pasilla* Data Set

As a running example, we use the *pasilla* data set, derived from [2]. The authors investigate conservation of RNA regulation between *D. melanogaster* and mammals. Part of their study used RNAi and RNA-seq to identify exons regulated by Pasilla (*ps*), the *D. melanogaster* ortholog of mammalian NOVA1 and NOVA2. Briefly, their experiment compared gene expression as measured by RNAseq in S2-DRSC cells cultured with, or without, a 444bp dsRNA fragment corresponding to the *ps* mRNA sequence. Their assessment investigated differential exon use, but our worked example will focus on gene-level differences.

In this section we look at a subset of the *ps* data, corresponding to reads obtained from lanes of their RNA-seq experiment, and to the same reads aligned to a *D. melanogaster* reference genome. Reads were obtained from GEO and the Short Read Archive (SRA), and were aligned to the *D. melanogaster* reference genome *dm3* as described in the *pasilla* experiment data package.

3.2 Reads and the *ShortRead* Package

Short read formats The Illumina GAII and HiSeq technologies generate sequences by measuring incorporation of florescent nucleotides over successive PCR cycles. These sequencers produce output in a variety of formats, but *FASTQ* is ubiquitous. Each read is represented by a record of four components:

```
@SRR031724.1 HWI-EAS299_4_30M2BAAXX:5:1:1513:1024 length=37
GTTTTGTCCAAGTTCTGGTAGCTGAATCCTGGGGCGC
```

```
+SRR031724.1 HWI-EAS299_4_30M2BAAXX:5:1:1513:1024 length=37
IIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
```

The first and third lines (beginning with `@` and `+` respectively) are unique identifiers. The second and fourth lines of the FASTQ record are the nucleotides and qualities of each cycle in the read. This information is given in 5' to 3' orientation as seen by the sequencer. A letter `N` in the sequence is used to signify bases that the sequencer was not able to call. The fourth line of the FASTQ record encodes the quality (confidence) of the corresponding base call. The quality score is encoded following one of several conventions, with the general notion being that letters later in the visible ASCII alphabet

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

are of lower quality. Both the sequence and quality scores may span multiple lines. Technologies other than Illumina use different formats to represent sequences; 454 sequence input is supported in *R453Plus1Toolbox*; 'color space' is not supported.

FASTQ files can be read in to *R* using the `readFastq` function from the *ShortRead* package. Use this function by providing the path to a FASTQ file. There are sample data files available in the *useR2012* package, each consisting of 1 million reads from a lane of the Pasilla data set.

```
> fastqDir <- file.path(bigdata(), "fastq")
> fastqFiles <- dir(fastqDir, full=TRUE)
> fq <- readFastq(fastqFiles[1])
> fq
```

```
class: ShortReadQ
length: 1000000 reads; width: 37 cycles
```

The data are represented as an object of class *ShortReadQ*.

```
> head(sread(fq), 3)
```

```
  A DNAStringSet instance of length 3
    width seq
[1]    37 GTTTTGTCCAAGTTCTGGTAGCTGAATCCTGGGGCGC
[2]    37 GTTGTCGCATTCCTTACTCTCATTCGGGAATTCTGTT
[3]    37 GAATTTTTTGAGAGCGAAATGATAGCCGATGCCCTGA
```

```
> head(quality(fq), 3)
```

```
class: FastqQuality
quality:
  A BStringSet instance of length 3
    width seq
[1]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
[2]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
[3]    37 IIIIIIIIIIIIIIIIIIIII'IIIIIGBIIII2I+
```

There are many ways to manipulate these objects; the `alphabetByCycle` function summarizes use of nucleotides at each cycle in a (equal width) *ShortReadQ* or *DNAStringSet* instance.

```
> abc <- alphabetByCycle(sread(fq))
> abc[1:4, 1:8]
```

```
        cycle
alphabet   [,1]   [,2]   [,3]   [,4]   [,5]   [,6]   [,7]   [,8]
       A  78194 153156 200468 230120 283083 322913 162766 220205
       C 439302 265338 362839 251434 203787 220855 253245 287010
       G 397671 270342 258739 356003 301640 247090 227811 246684
       T  84833 311164 177954 162443 211490 209142 356178 246101
```

FASTQ files are getting larger. A very common reason for looking at data at this early stage in the processing pipeline is to explore sequence quality. In these circumstances it is often not necessary to parse the entire FASTQ file. Instead create a representative sample

```
> sampler <- FastqSampler(fastqFiles[1], 1000000)
> yield(sampler) # sample of 1000000 reads
```

```
class: ShortReadQ
length: 1000000 reads; width: 37 cycles
```

A second common scenario is to pre-process reads, e.g., trimming low-quality tails, adapter sequences, or artifacts of sample preparation. The *FastqStreamer* class can be used to 'stream' over the fastq files in chunks, processing each chunk independently.

ShortRead contains facilities for quality assessment of FASTQ files. Here we generate a report from a sample of 1 million reads from each of our files and display it in a web browser

```
> qas0 <- Map(function(fl, nm) {
+     fq <- FastqSampler(fl)
+     qa(yield(fq), nm)
+ }, fastqFiles,
+    sub("_subset.fastq", "", basename(fastqFiles)))
> qas <- do.call(rbind, qas0)
> rpt <- report(qas, dest=tempfile())
> browseURL(rpt)
```

A report from a larger subset of the experiment is available

```
> rpt <- system.file("GSM461176_81_qa_report", "index.html",
+                    package="useR2012")
> browseURL(rpt)
```

Exercise 5
Use the helper function `bigdata` *(defined in the useR2012 package) and the* `file.path` *and* `dir` *functions to locate two fastq files from [2] (the files were obtained as described in the appendix and pasilla experiment data package.*

Input one of the fastq files using `readFastq` *from the ShortRead package.*

Using the helper function `gcFunction` *from the useR2012 package, draw a histogram of the distribution of GC frequencies across reads.*

Use `alphabetByCycle` *to summarize the frequency of each nucleotide, at each cycle. Plot the results using* `matplot`*, from the graphics package.*

As an advanced exercise, and if on Mac or Linux, use the parallel package and `mclapply` *to read and summarize the GC content of reads in two files in parallel.*

Solution: Discovery:

```
> dir(bigdata())
```

```
[1] "bam"   "fastq"
```

```
> fls <- dir(file.path(bigdata(), "fastq"), full=TRUE)
```

Input:

```
            used  (Mb) gc trigger   (Mb)  max used   (Mb)
Ncells   4071585 217.5    6193578  330.8   6193578  330.8
Vcells 125856983 960.3  279142478 2129.7 279142148 2129.7
```

```
> fq <- readFastq(fls[1])
```

GC content:

```
> alf0 <- alphabetFrequency(sread(fq), as.prob=TRUE, collapse=TRUE)
> sum(alf0[c("G", "C")])
```

```
[1] 0.5457237
```

A histogram of the GC content of individual reads is obtained with

```
> gc <- gcFunction(sread(fq))
> hist(gc)
```

Alphabet by cycle:

```
> abc <- alphabetByCycle(sread(fq))
> matplot(t(abc[c("A", "C", "G", "T"),]), type="l")
```

Advanced (Mac, Linux only): processing on multiple cores.

```
> library(parallel)
> gc0 <- mclapply(fls, function(fl) {
+   fq <- readFastq(fl)
+   gc <- gcFunction(sread(fq))
+   table(cut(gc, seq(0, 1, .05)))
+ })
> ## simplify list of length 2 to 2-D array
> gc <- simplify2array(gc0)
> matplot(gc, type="s")
```

3.3 Alignments and the *Rsamtools* Package

Most down-stream analysis of short read sequences is based on reads aligned to reference genomes. There are many aligners available, including BWA [10, 9], Bowtie [8], and GSNAP; merits of these are discussed in the literature. There are also alignment algorithms implemented in *Bioconductor* (e.g., `matchPDict` in the *Biostrings* package, and the *Rsubread* package); `matchPDict` is particularly useful for flexible alignment of moderately sized subsets of data.

Alignment formats Most main-stream aligners produce output in SAM (text-based) or BAM format. A SAM file is a text file, with one line per aligned read, and fields separated by tabs. Here is an example of a single SAM line, split into fields.

```
> fl <- system.file("extdata", "ex1.sam", package="Rsamtools")
> strsplit(readLines(fl, 1), "\t")[[1]]
```

```
 [1] "B7_591:4:96:693:509"
 [2] "73"
 [3] "seq1"
 [4] "1"
 [5] "99"
 [6] "36M"
 [7] "*"
 [8] "0"
 [9] "0"
[10] "CACTAGTGGCTCATTGTAAATGTGTGGTTTAACTCG"
[11] "<<<<<<<<<<<<<<<;<<<<<<<<<5<<<<<;:<;7"
[12] "MF:i:18"
[13] "Aq:i:73"
[14] "NM:i:0"
[15] "UQ:i:0"
[16] "H0:i:1"
[17] "H1:i:0"
```

The SAM specification summarizes these fields. We recognize from the FASTQ file the identifier string, read sequence and quality. The alignment is to a chromosome 'seq1' starting at position 1. The strand of alignment is encoded in the 'flag' field. The alignment record also includes a measure of mapping quality, and a CIGAR string describing the nature of the alignment. In this case, the CIGAR is 36M, indicating that the alignment consisted of 36 `M`atches or mismatches, with no indels or gaps; indels are represented by `I` and `D`; gaps (e.g., from alignments spanning introns) by `N`. BAM files encode the same information as SAM files, but in a format that is more efficiently parsed by software; BAM files are the primary way in which aligned reads are imported in to *R*.

Aligned reads in *R* The `readGappedAlignments` function from the *GenomicRanges* package reads essential information from a BAM file in to *R*. The result is an instance of the *GappedAlignments* class. The *GappedAlignments* class has been designed to allow useful manipulation of many reads (e.g., 20 million) under moderate memory requirements (e.g., 4 GB).

```
> alnFile <- system.file("extdata", "ex1.bam", package="Rsamtools")
> aln <- readGappedAlignments(alnFile)
> head(aln, 3)
```

```
GappedAlignments with 3 alignments and 0 elementMetadata cols:
      seqnames strand       cigar    qwidth     start       end     width
         <Rle>  <Rle> <character> <integer> <integer> <integer> <integer>
  [1]     seq1      +         36M        36         1        36        36
  [2]     seq1      +         35M        35         3        37        35
  [3]     seq1      +         35M        35         5        39        35
           ngap
      <integer>
  [1]         0
  [2]         0
  [3]         0
  ---
  seqlengths:
   seq1 seq2
   1575 1584
```

The `readGappedAlignments` function takes an additional parameter, `param`, allowing the user to specify regions of the BAM file (e.g., known gene coordinates) from which to extract alignments.

A *GappedAlignments* instance is like a data frame, but with accessors as suggested by the column names. It is easy to query, e.g., the distribution of reads aligning to each strand, the width of reads, or the cigar strings

```
> table(strand(aln))
```

```
   +    -
1647 1624
```

```
> table(width(aln))

  30   31   32   33   34   35   36   38   40 
   2   21    1    8   37 2804  285    1  112 

> head(sort(table(cigar(aln)), decreasing=TRUE))

    35M      36M      40M      34M      33M 14M4I17M 
   2804      283      112       37        6        4 
```

Exercise 6
Use `bigdata`, `file.path` *and* `dir` *to obtain file paths to the BAM files. These are a subset of the aligned reads, overlapping just four genes.*

Input the aligned reads from one file using `readGappedAlignments`*. Explore the reads, e.g., using* `table` *or* `xtabs`*, to summarize which chromosome and strand the subset of reads is from.*

The object `ex` *created earlier contains coordinates of four genes. Use* `countOverlaps` *to first determine the number of genes an individual read aligns to, and then the number of uniquely aligning reads overlapping each gene. Since the RNAseq protocol was not strand-sensitive, set the strand of* `aln` *to* `*`*.*

Write the sequence of steps required to calculate counts as a simple function, and calculate counts on each file. On Mac or Linux, can you easily parallelize this operation?

Solution: We discover the location of files using standard *R* commands:

```
> fls <- dir(file.path(bigdata(), "bam"), ".bam$", full=TRUE)
> names(fls) <- sub("_.*", "", basename(fls))
```

Use `readGappedAlignments` to input data from one of the files, and standard *R* commands to explore the data.

```
> ## input
> aln <- readGappedAlignments(fls[1])
> xtabs(~seqnames + strand, as.data.frame(aln))

        strand
seqnames    +    -
   chr3L 5402 5974
   chrX  2278 2283
```

To count overlaps in regions defined in a previous exercise, load the regions.

```
> data(ex)              # from an earlier exercise
```

Many RNA-seq protocols are not strand aware, i.e., reads align to the plus or minus strand regardless of the strand on which the corresponding gene is encoded. Adjust the strand of the aligned reads to indicate that the strand is not known.

```
> strand(aln) <- "*"   # protocol not strand-aware
```

For simplicity, we are interested in reads that align to only a single gene. Count the number of genes a read aligns to...

```
> hits <- countOverlaps(aln, ex)
> table(hits)
```

```
hits
    0     1     2 
  772 15026   139 
```

and reverse the operation to count the number of times each region of interest aligns to a uniquely overlapping alignment.

```
> cnt <- countOverlaps(ex, aln[hits==1])
```

A simple function for counting reads is

```
> counter <-
+     function(filePath, range)
+ {
+     aln <- readGappedAlignments(filePath)
+     strand(aln) <- "*"
+     hits <- countOverlaps(aln, range)
+     countOverlaps(range, aln[hits==1])
+ }
```

This can be applied to all files using `sapply`

```
> counts <- sapply(fls, counter, ex)
```

The counts in one BAM file are independent of counts in another BAM file. This encourages us to count reads in each BAM file in parallel, decreasing the length of time required to execute our program. On Linux and Mac OS, a straight-forward way to parallelize this operation is:

```
> if (require(parallel))
+     simplify2array(mclapply(fls, counter, ex))
```

The `summarizeOverlaps` function in the *GenomicRanges* package implements more appropraite counting strategies.

Exercise 7
Consult the help page for `ScanBamParam`, and construct an object that restricts the information returned by a `scanBam` query to the aligned read DNA sequence. Your solution will use the `what` parameter to the `ScanBamParam` function.

Use the `ScanBamParam` object to query a BAM file, and calculate the GC content of all aligned reads. Summarize the GC content as a histogram (Figure 2).

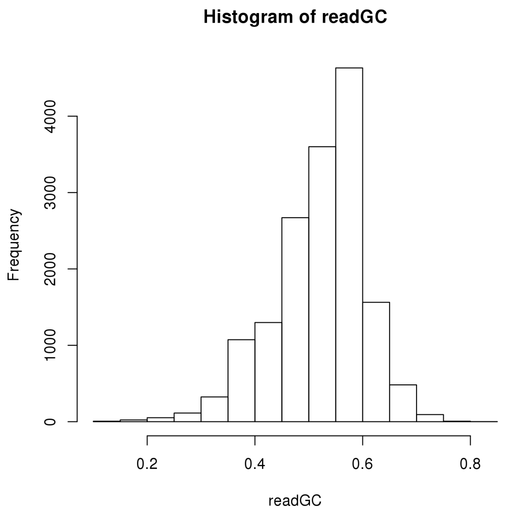

Figure 2: GC content in aligned reads

Solution:

```
> param <- ScanBamParam(what="seq")
> seqs <- scanBam(fls[[1]], param=param)
> readGC <- gcFunction(seqs[[1]][["seq"]])
> hist(readGC)
```

Table 6: Selected *Bioconductor* packages for RNA-seq analysis.

Package	Description
EDASeq	Exploratory analysis and QA; also *qrqc*, *ShortRead*.
edgeR, *DESeq*	Generalized Linear Models using negative binomial error.
DEXSeq	Exon-level differential representation.
goseq	Gene set enrichment tailored to RNAseq count data; also *limma*'s `roast` or `camera` after transformation with `voom` or *cqn*.
easyRNASeq	Workflow; also *ArrayExpressHTS*, *rnaSeqMap*, *oneChannelGUI*.
Rsubread	Alignment (Linux only); also *Biostrings* `matchPDict` for special-purpose alignments.

4 RNA-seq

Varieties of RNA-seq RNA-seq experiments typically ask about differences in trancription of genes or other features across experimental groups. The analysis of designed experiments is statistical, and hence an ideal task for *R*. The overall structure of the analysis, with tens of thousands of features and tens of samples, is reminiscent of microarray analysis; some insights from the microarray domain will apply, at least conceptually, to the analysis of RNA-seq experiments.

The most straight-forward RNA-seq experiments quantify abundance for known gene models. The known models are derived from reference databases, reflecting the accumulated knowledge of the community responsible for the data. A more ambitious approach to RNA-seq attempts to identify novel transcripts; this is beyond the scope of today's tutorial.

Bioconductor packages play a role in several stages of an RNA-seq analysis (Table 6; a more comprehensive list is under the RNAseq and HighThroughput-Sequencing BiocViews terms). The *GenomicRanges* infrastructure can be effectively employed to quantify known exon or transcript abundances. Quantified abundances are in essence a matrix of counts, with rows representing features and columns samples. The *edgeR* [15] and *DESeq* [1] packages facilitate analysis of this data in the context of designed experiments, and are appropriate when the questions of interest involve between-sample comparisons of relative abundance. The *DEXSeq* package extends the approach in *edgeR* and *DESeq* to ask about within-gene, between group differences in exon use, i.e., for a given gene, do groups differ in their exon use?

4.1 Differential Expression with the *edgeR* Package

RNA-seq differential representation experiments, like classical microarray experiments, consist of a single statistical design (e.g, comparing expression of samples assigned to 'Treatment' versus 'Control' groups) applied to each feature for which there are aligned reads. While one could naively perform simple tests (e.g., t-tests) on all features, it is much more informative to identify important aspects of RNAseq experiments, and to take a flexible route through this part of the work flow. Key steps involve formulation of a model matrix to capture the experimental design, estimation of a test static to describe differences between groups, and calculation of a P value or other measure as a statement of statistical significance.

Counting and filtering An essential step is to arrive at some measure of gene representation amongst the aligned reads. A straight-forward and commonly used approach is to count the number of times a read overlaps exons. Nuance arises when a read only partly overlaps an exon, when two exons overlap (and hence a read appears to be 'double counted'), when reads are aligned with gaps and the gaps are inconsistent with known exon boundaries, etc. The `summarizeOverlaps` function in the *GenomicRanges* package provides facilities for implementing different count strategies, using the argument `mode` to determine the counting strategy. The result of `summarizeOverlaps` can easily be used in subsequent steps of an RNA-seq analysis. Software other than *R* can also be used to summarize count data. An important point is that the desired input for downstream analysis is often raw count data, rather than normalized (e.g., reads per kilobase of gene model per million mapped reads) values. This is because counts allow information about uncertainty of estimates to propagate to later stages in the analysis.

The following exercise illustrates key steps in counting and filtering reads overlapping known genes.

Exercise 8
The useR2012 package contains a data set `counts` *with pre-computed count data. Use the* `data` *command to load it. Create a variable* `grp` *to define the groups associated with each column, using the column names as a proxy for more authoritative metadata.*

Create a `DGEList` *object (defined in the edgeR package) from the count matrix and group information. Calculate relative library sizes using the* `calcNormFactors` *function.*

A lesson from the microarray world is to discard genes that cannot be informative (e.g., because of lack of variation), regardless of statistical hypothesis under evaluation. Filter reads to remove those that are represented at less than 1 per million mapped reads, in fewer than 2 samples.

Solution: Here we load the data (a matrix of counts) and create treatment group names from the column names of the counts matrix.

```
> data(counts)
> dim(counts)
```

```
[1] 14470     7
```

```
> grps <- factor(sub("[1-4].*", "", colnames(counts)),
+                levels=c("untreated", "treated"))
> pairs <- factor(c("single", "paired", "paired",
+                   "single", "single", "paired", "paired"))
> pData <- data.frame(Group=grps, PairType=pairs,
+                     row.names=colnames(counts))
```

We use the *edgeR* package, creating a *DGEList* object from the count and group data. The `calcNormFactors` function estimates relative library sizes for use as offsets in the generalized linear model.

```
> library(edgeR)
> dge <- DGEList(counts, group=pData$Group)
> dge <- calcNormFactors(dge)
```

To filter reads, we scale the counts by the library sizes and express the results on a per-million read scale. This is done using the `sweep` function, dividing each column by it's library size and multiplying by `1e6`. We require that the gene be represented at a frequency of at least 1 read per million mapped (`m > 1`, below) in two or more samples (`rowSums(m > 1) >= 2`), and use this criterion to subset the *DGEList* instance.

```
> m <- sweep(dge$counts, 2, 1e6 / dge$samples$lib.size, `*`)
> ridx <- rowSums(m > 1) >= 2
> table(ridx)                        # number filtered / retained

ridx
FALSE  TRUE
 6476  7994

> dge <- dge[ridx,]
```

Experimental design In *R*, an experimental design is specified with the `model.matrix` function. The function takes as its first argument a `formula` describing the independent variables and their relationship to the response (counts), and as a second argument a `data.frame` containing the (phenotypic) data that the formula describes. A simple formula might read `~ 1 + Group`, which says that the response is a linear function involving an intercept (1) plus a term encoded in the variable `Group`. If (as in our case) `Group` is a factor, then the first coefficient (column) of the model matrix corresponds to the first level of `Group`, and subsequent terms correspond to *deviations* of each level from the first. If `Group` were *numeric* rather than *factor*, the formula would represent linear regressions with an intercept. Formulas are very flexible, allowing representation of models with one, two, or more factors as main effects, models with or without interaction, and with nested effects.

Exercise 9
To be more concrete, use the `model.matrix` function and a formula involving `Group` to create the model matrix for our experiment.

Solution: Here is the experimental design; it is worth discussing with your neighbor the interpretation of the `design` instance.

```
> (design <- model.matrix(~ Group, pDATA))
```

```
             (Intercept) Grouptreated
treated1fb             1            1
treated2fb             1            1
treated3fb             1            1
untreated1fb           1            0
untreated2fb           1            0
untreated3fb           1            0
untreated4fb           1            0
attr(,"assign")
[1] 0 1
attr(,"contrasts")
attr(,"contrasts")$Group
[1] "contr.treatment"
```

The coefficient (column) labeled 'Intercept' corresponds to the first level of `Group`, i.e., 'untreated'. The coefficient 'Grouptreated' represents the deviation of the treated group from untreated. Eventually, we will test whether the second coefficient is significantly different from zero, i.e., whether samples with a '1' in the second column are, on average, different from samples with a '0'. On the one hand, use of `model.matrix` to specify experimental design implies that the user is comfortable with something more than elementary statistical concepts, while on the other it provides great flexibility in the experimental design that can be analyzed.

Negative binomial error RNA-seq count data are often described by a negative binomial error model. This model includes a 'dispersion' parameter that describes biological variation beyond the expectation under a Poisson model. The simplest approach estimates a dispersion parameter from all the data. The estimate needs to be conducted in the context of the experimental design, so that variability between experimental factors is not mistaken for variability in counts. The square root of the estimated dispersion represents the coefficient of variation between biological samples. The following *edgeR* commands estimate dispersion.

```
> dge <- estimateTagwiseDisp(dge)
> mean(sqrt(dge$tagwise.dispersion))
```

```
[1] 0.1778359
```

This approach assumes that a common dispersion parameter is shared by all genes. A different approach, appropriate when there are more samples in the study, is to estimate a dispersion parameter that is specific to each tag (using `estimateTagwiseDisp` in the *edgeR* package). As another alternative, Anders and Huber [1] note that dispersion increases as the mean number of reads per gene decreases. One can estimate the relationship between dispersion and mean using `estimateGLMTrendedDisp` in *edgeR*, using a fitted relationship across all genes to estimate the dispersion of individual genes. Because in our case sample sizes (biological replicates) are small, gene-wise estimates of dispersion are likely imprecise. One approach is to moderate these estimates by calculating a weighted average of the gene-specific and common dispersion; `estimateGLMTagwiseDisp` performs this calculation, requiring that the user provides an *a priori* estimate of the weight between tag-wise and common dispersion.

Differential representation The final steps in estimating differential representation are to fit the full model; to perform the likelihood ratio test comparing the full model to a model in which one of the coefficients has been removed; and to summarize, from the likelihood ratio calculation, genes that are most differentially represented. The result is a 'top table' whose row names are the Flybase gene ids used to label the elements of the `ex` *GRangesList*.

Exercise 10
Use `glmFit` to fit the general linear model. This function requires the input data `dge`, the experimental design `design`, and the estimate of dispersion.

Use `glmLRT` to form the likelihood ratio test. This requires the original data `dge` and the fitted model from the previous part of this question. Which coefficient of the design matrix do you wish to test?

Create a 'top table' of differentially represented genes using `topTags`.

Solution: Here we fit a generalized linear model to our data and experimental design, using the tagwise dispersion estimate.

```
> fit <- glmFit(dge, design)
```

The fit can be used to calculate a likelihood ratio test, comparing the full model to a reduced version with the second coefficient removed. The second coefficient captures the difference between treated and untreated groups, and the likelihood ratio test asks whether this term contributes meaningfully to the overall fit.

```
> lrTest <- glmLRT(dge, fit, coef=2)
```

Here the `topTags` function summarizes results across the experiment.

```
> tt <- topTags(lrTest, n=10)
> tt[1:3,]
```

```
Coefficient:  Grouptreated
                 logFC   logCPM       LR        PValue           FDR
FBgn0039155 -4.698051 6.030423 542.3132 5.918302e-120 4.731091e-116
FBgn0039827 -4.275280 4.591903 247.2889  1.012761e-55  4.048005e-52
FBgn0029167 -2.233973 8.246355 211.4650  6.580299e-48  1.753430e-44
```

As a 'sanity check', summarize the original data for the first several probes, confirming that the average counts of the treatment and control groups are substantially different.

```
> sapply(rownames(tt$table)[1:4],
+        function(x) tapply(counts[x,], pData$Group, mean))

          FBgn0039155 FBgn0039827 FBgn0029167 FBgn0034736
untreated        1576         554    6447.000      382.25
treated            64          31    1482.667       36.00
```

4.2 Additional Steps in RNA-seq Work Flows

The forgoing provides an elementary work flow. There are many interesting additional opportunities, including:

Annotation Standard *Bioconductor* facilities, e.g., the `select` method from the *AnnotationDbi* package applied to packages such as *org.Dm.eg.db* can provide biological context (e.g., gene name, KEGG or GO pathway membership) for interpretting genes at the top of a top table. Packages using *GenomicFeatures*, e.g., *TxDb.Dmelanogaster.UCSC.dm3.ensGene*, can provide information on genome structure, e.g., genomic coordinates of exons, and relationship between exons, coding sequences, transcripts, and genes.

Gene Set Enrichment Care needs to be taken because statistical signficance of genes is proportional to the number of reads aligning to the gene (e.g., due to gene length or GC content); see, e.g., *goseq*.

Exon-level Differential Representation The *DEXSeq* package takes an interesting approach to within-gene differential expression, testing for interaction between exon use and treatment. The forthcoming *SpliceGraph* package takes this a step further by summarizing gene models into graphs, with 'bubbles' representing alternative splicing events; this reduces the number of statistical tests (increasing count per edge and statistical power) while providing meaningful insight into the types of events (e.g., 'exon skip', 'alternative acceptor') occuring.

4.3 Resources

The *edgeR*, *DESeq*, and *DEXSeq* package vignettes provide excellent, extensive discussion of issues and illustration of methods for RNA-seq differential expression analysis.

Table 7: Selected *Bioconductor* packages for RNA-seq analysis.

Package	Description
qrqc	Quality assessment; also *ShortRead*, *chipseq*.
PICS	Peak calling, also *mosaics*, *chipseq*, *ChIPseqR*, *BayesPeak*, *nucleR* (nucleosome positioning).
ChIPpeakAnno	Peak annotation.
DiffBind	Multiple-experiment analysis.
MotIV	Motif identification and validation; also *rGADEM*.

5 ChIP-seq

ChIP-seq and similar experiments combine chromosome immuno-precipitation (ChIP) with sequence analysis. The idea is that the ChIP protocol enriches genomic DNA for regions of interest, e.g., sites to which transcription factors are bound. The regions of interest are then subject to high throughput sequencing, the reads aligned to a reference genome, and the location of mapped reads ('peaks') interpreted as indicators of the ChIP'ed regions. Reviews include those by Park and colleagues [13, 6]; there is a large collection of peak-calling software, some features of which are summarized in Pepke et al. [14].

The main challenge in early ChIP-seq studies was to develop efficient peak-calling software, often tailored to the characteristics of the peaks of interest (e.g., narrow and well-defined CTCF binding sites, vs. broad histone marks). More comprehensive studies draw from multiple samples, e.g., in the ENCODE project [7, 12]. Decreasing sequence costs and better experimental and data analytic protocols mean that these larger-scale studies are increasingly accessible to individual investigators. Peak-calling in this kind of study represents an initial step, but interpretting analyses derived from multiple samples present significant analytic challenges. *Bioconductor* packages play a role in several stages of a ChIP-seq analysis. (Table 7; a more comprehensive list is under the ChIPseq and HighThroughputSequencing BiocViews terms).

Our attention is on analyzing multiple samples from a single experiment, and identifying and annotating peaks. We start with a typical work flow re-iterating key components in an exploration of data from the ENCODE project, and continue with down-stream analysis including motif discovery and annotation.

5.1 Initial Work Flow

We use data from GEO accession GSE30263, representing ENCODE CTCF binding sites. CTCF is a zinc finger transcription factor. It is a sequence specific DNA binding protein that functions as an insulator, blocking enhancer activity, and possibly the spread of chromatin structure. The original analysis involved Illumina ChIP-seq and matching 'input' lanes of 1 or 2 replicates from many cell lines. The GEO accession includes BAM files of aligned reads, in addition to tertiary files summarizing identified peaks. We focus on 15 cell lines aligned to hg19.

As a precursor to analysis, it is prudent to perform an overall quality assesssement of the data; an example is available:

```
> rpt <- system.file("GSE30263_qa_report", "index.html",
+            package="useR2012", mustWork=TRUE)
> if (interactive())
+     browseURL(rpt)
```

The main computational stages in the original work flow involved alignment using Bowtie, followed by peak identification using an algorithm ('HotSpots', [16]) originally developed for lower-throughput methodologies. We collated the output files from the original analysis with a goal of enumerating all peaks from all files, but collapsing the coordinates of sufficiently similar peaks to a common location. The *DiffBind* package provides a formalism with which to do these operations. Here we load the data as an *R* object `stam` (an abbreviation for the lab generating the data).

```
> stamFile <- system.file("data", "stam.Rda", package="useR2012")
> load(stamFile)
> stam
```

```
class: SummarizedExperiment
dim: 369674 96
exptData(0):
assays(2): Tags PVals
rownames: NULL
rowData values names(0):
colnames(96): A549_1 A549_2 ... Wi38_1 Wi38_2
colData names(10): CellLine Replicate ... PeaksDate PeaksFile
```

Exercise 11
Explore `stam`*. Tabulate the number of peaks represented 1, 2, . . . , 96 times. We expect replicates to have similar patterns of peak representation; do they?*

Solution: Load the data and display the *SummarizedExperiment* instance. The `colData` summarizes information about each sample, the `rowData` about each peak. Use `xtabs` to summarize `Replicate` and `CellLine` representation within `colData(stam)`.

```
> head(colData(stam), 3)
```

```
DataFrame with 3 rows and 10 columns
              CellLine Replicate   TotTags  TotPeaks      Tags     Peaks
           <character>  <factor> <integer> <integer> <numeric> <numeric>
A549_1            A549         1   1857934     50144   1569215     43119
A549_2            A549         2   2994916     77355   2881475     73062
Ag04449_1      Ag04449         1   5041026     81855   4730232     75677
```

```
           FastqDate FastqSize  PeaksDate
              <Date> <numeric>     <Date>
A549_1    2011-06-25       463 2011-06-25
A549_2    2011-06-25       703 2011-06-25
Ag04449_1 2010-10-22       368 2010-10-22
                                                   PeaksFile
                                                 <character>
A549_1       wgEncodeUwTfbsA549CtcfStdPkRep1.narrowPeak.gz
A549_2       wgEncodeUwTfbsA549CtcfStdPkRep2.narrowPeak.gz
Ag04449_1 wgEncodeUwTfbsAg04449CtcfStdPkRep1.narrowPeak.gz
```

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

```
GRanges with 3 ranges and 0 elementMetadata cols:
      seqnames         ranges strand
         <Rle>      <IRanges>  <Rle>
  [1]     chr1 [10100, 10370]      *
  [2]     chr1 [15640, 15790]      *
  [3]     chr1 [16100, 16490]      *
  ---
  seqlengths:
        chr1      chr2      chr3      chr4 ...     chr22      chrX      chrY
   249250621 243199373 198022430 191154276 ...  51304566 155270560  59373566
```

```
> xtabs(~Replicate + CellLine, colData(stam))[,1:5]
```

```
         CellLine
Replicate A549 Ag04449 Ag04450 Ag09309 Ag09319
        1    1       1       1       1       1
        2    1       1       1       1       1
```

Extract the Tags matrix from the assays. This is a standard *R matrix*. Test which matrix elements are non-zero, tally these by row, and summarize the tallies. This is the number of times a peak is detected, across each of the samples

```
> m <- assays(stam)[["Tags"]] > 0 # peaks detected...
> peaksPerSample <- table(rowSums(m))
> head(peaksPerSample)
```

```
     1      2      3      4      5      6
174574  35965  18939  12669   9143   7178
```

```
> tail(peaksPerSample)
```

```
   91    92    93    94    95    96
 1226  1285  1542  2082  2749 14695
```

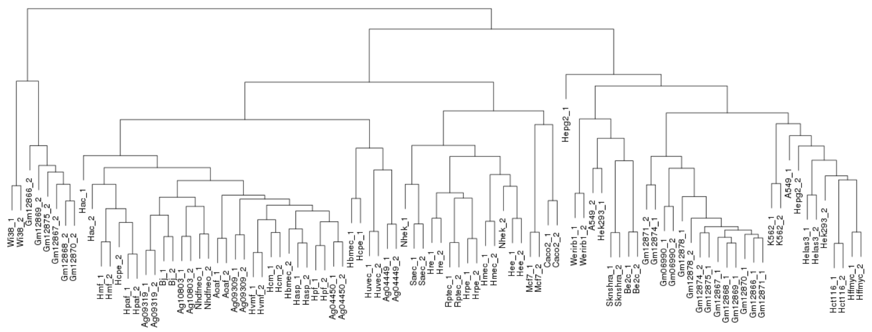

Figure 3: Hierarchical clustering of ENCODE samples.

To explore similarity between replicates, extract the matrix of counts. Transform the counts using the `asinh` function (a log-like transform, except near 0; are there other methods for transformation?), and use the 'correlation' distance (`cor.dist`, from *bioDist*) to measure similarity. Cluster these using a hierarchical algorithm, via the `hclust` function.

```
> library(bioDist)                    # for cor.dist
> m <- asinh(assays(stam)[["Tags"]]) # transformed tag counts
> d <- cor.dist(t(m))                 # correlation distance
> h <- hclust(d)                      # hierarchical clustering
```

Plot the result, as in Figure 3.

```
> plot(h, cex=.8, ann=FALSE)
```

5.2 Motifs

Transcription factors and other common regulatory elements often target specific DNA sequences ('motifs'). These are often well-characterized, and can be used to help identify, *a priori*, regions in which binding is expected. Known binding motifs may also be used to identify promising peaks identified using *de novo* peak discovery methods like *MACS*. This section explores use of known binding motifs to characterize peaks; packages such as *MotIV* can assist in motif discovery.

Known binding motifs The JASPAR data base curates known binding motifs obtained from the literature. A binding motif is summarized as a *position weight matrix* PWM. Rows of a PWM correspond to nucleotides, columns to positions, and entries to the probability of the nucleotide at that position. Each start position in a reference sequence can be compared and scored for similarity to the PWM, and high-scoring positions retained.

Exercise 12
The objective of this exercise is to identify occurrences of the CTCF motif on chromosome 1 of H. sapiens*.*

Load needed packages. Biostrings can represent a PWM and score a reference sequence. The BSgenome.Hsapiens.UCSC.hg19 package contains the hg19 build of H. sapiens*, retrieved from the UCSC genome browser. seqLogo and lattice are used for visualization.*

Retrieve the PWM for CTCF, with JASPAR id MA0139.1.pfm, using the helper function `getJASPAR` defined in the useR2012 package.

Use `matchPWM` to score the plus strand of chr1 for the CTCF PWM. Visualize the distribution of scores using, e.g., `densityplot`, and summarize the high-scoring matches (using `consensusMatrix`) as a `seqLogo`.

As an additional exercise, work up a short code segment to apply the PWM to both strands (see `?PWM` for some hints) and to all chromosomes.

Solution: Here we load the required packages and retrieve the position weight matrix for CTCF.

```
> library(Biostrings)
> library(BSgenome.Hsapiens.UCSC.hg19)
> library(seqLogo)
> library(lattice)
> pwm <- getJASPAR("MA0139.1.pfm") # useR2012::getJASPAR
```

Chromosome 1 can be loaded with `Hsapiens[["chr1"]]`; `matchPWM` returns a 'view' of the high-scoring locations matching the PWM. Scores are retrieved from the PWM and hits using `PWMscoreStartingAt`.

```
> chrid <- "chr1"
> hits <-matchPWM(pwm, Hsapiens[[chrid]]) # '+' strand
> scores <- PWMscoreStartingAt(pwm, subject(hits), start(hits))
```

The distribution of scores can be visualized with, e.g., `densityplot` from the *lattice* package.

```
> densityplot(scores, xlim=range(scores), pch="|")
```

`consensusMatrix` applied to the views in `hits` returns a position frequency amtrix; this can be plotted as a logo, with the result in Figure 4. Reassuringly, the found sequences have a logo very similar to the expected.

```
> cm <- consensusMatrix(hits)[1:4,]
> seqLogo(makePWM(scale(cm, FALSE, colSums(cm))))
```

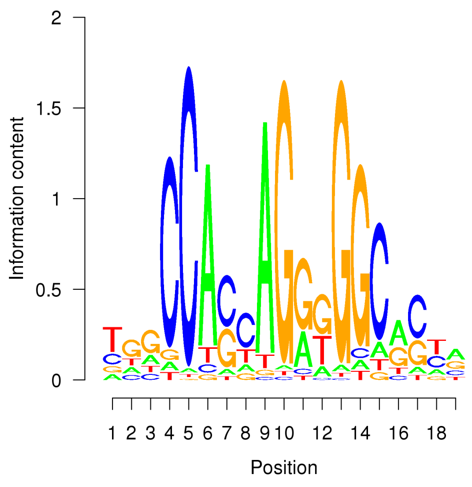

Figure 4: CTCF position weight matrix of found sites on the plus strand of chr1 (hits within 80% of maximum score).

6 Annotation

Bioconductor provides extensive annotation resources, summarized in Figure 5. 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*.
- 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.

6.1 Gene-Centric Annotations with *AnnotationDbi*

Organism-level ('org') packages uses a central gene identifier (e.g. Entrez Gene id) and contain mappings between this identifier and other kinds of identifiers (e.g. GenBank or Uniprot accession number, RefSeq id, etc.). The name of an org package is always of the form *org.<Ab>.<efg>.db* (e.g. *org.Sc.sgd.db*) where *<Ab>* is a 2-letter abbreviation of the organism (e.g. `Sc` for *Saccharomyces cerevisiae*) and *<efg>* 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.

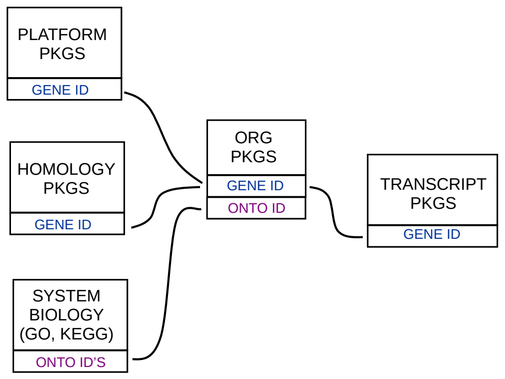

Figure 5: Annotation Packages: the big picture

Exercise 13
What is the name of the org package for Drosophila*? Load it.*

Use `ls("package:<pkgname>")` *to display the list of all symbols defined in this package. Explore a few of the symbols by looking at their man page, at their class, and by viewing their* `head` *with* `toTable`*.*

Most maps can be reversed with `revmap`*. Reverse the* `org.Dm.egUNIPROT` *map and extract a few identifiers from the reversed map.*

Solution:

```
> library(org.Dm.eg.db)
> head(ls('package:org.Dm.eg.db'), 3)
```

```
[1] "org.Dm.eg"       "org.Dm.eg.db"    "org.Dm.egACCNUM"
```

```
> org.Dm.egUNIPROT
```

```
UNIPROT map for Fly (object of class "AnnDbBimap")
```

```
> class(org.Dm.egUNIPROT)
```

```
[1] "AnnDbBimap"
attr(,"package")
[1] "AnnotationDbi"
```

```
> toTable(head(org.Dm.egUNIPROT, 3))
```

```
  gene_id uniprot_id
1   30970     Q8IRZ0
2   30970     Q95RP8
3   30971     Q95RU8
4   30972     Q9W5H1
```

Each map consists of left keys and right keys. The left keys are the Entrez gene ids and the right keys the Uniprot accession numbers. For all maps in an org package the left key is always the central gene id.

```
> toTable(head(revmap(org.Dm.egUNIGENE), 3))

  gene_id unigene_id
1   30970    Dm.6474
2   30971       Dm.9
3   30972   Dm.12271

> identical(Lkeys(org.Dm.egUNIGENE), Lkeys(revmap(org.Dm.egUNIGENE)))

[1] TRUE
```

Recent versions of many annotation packages allow a simpler way of extracting annotations. Annotation packages supporting these new methods 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`.

Exercise 14
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`.

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

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

> keytypes(org.Dm.eg.db)

 [1] "ENTREZID"     "ACCNUM"       "ALIAS"        "CHR"          "ENZYME"
 [6] "MAP"          "PATH"         "PMID"         "REFSEQ"       "SYMBOL"
[11] "UNIGENE"      "FLYBASE"      "FLYBASECG"    "FLYBASEPROT"  "UNIPROT"
[16] "ENSEMBL"      "ENSEMBLPROT"  "ENSEMBLTRANS" "GO"
```

```
> 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 Suv4-20
3 00310  Q9W3N9 CG10932
```

6.2 Genome-Centric Annotations with *GenomicFeatures*

Genome-centric packages are very useful for annotations involving genomic coordinates. It is straight-forward, 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 (Section 4) and retrieving DNA sequences underlying regions of interest in ChIP-seq analysis (Section 5), e.g., for motif characterization.

Exercise 15
The objective of this exercise is to characterize the distance between identified peaks and nearest transcription start site.

Load the ENCODE summary data, select the peaks found in all samples, and use the center of these peaks as a proxy for the true ChIP binding site.

Use the transcript data base for the UCSC Known Genes track of hg19 as a source for transcripts and transcription start sites (TSS).

Use `nearest` to identify the TSS that is nearest each peak, and calculate the distance between the peak and TSS; measure distance taking account of the strand of the transcript, so that peaks 5' of the TSS have negative distance.

Summarize the locations of the peaks relative to the TSS.

Solution: Read in the ENCODE ChIP peaks for all cell lines.

```
> stamFile <- system.file("data", "stam.Rda", package="useR2012")
> load(stamFile)
```

Identify the rows of `stam` that have non-zero counts for all cell lines, and extract the corresponding ranges:

```
> ridx <- rowSums(assays(stam)[["Tags"]] > 0) == ncol(stam)
> peak <- rowData(stam)[ridx]
```

Select the center of the ranges of these peaks, as a proxy for the ChIP binding site:

```
> peak <- resize(peak, width=1, fix="center")
```

Obtain the TSS from the *TxDb.Hsapiens.UCSC.hg19.knownGene* using the `transcripts` function to extract coordinates of each transcript, and `resize` to a width of 1 for the TSS; does this do the right thing for transcripts on the plus and on the minus strand?

```
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> tx <- transcripts(TxDb.Hsapiens.UCSC.hg19.knownGene)
> tss <- resize(tx, width=1)
```

The `nearest` function returns the index of the nearest `subject` to each `query` element; the distance between peak and nearest TSS is thus

```
> idx <- nearest(peak, tss)
> sgn <- as.integer(ifelse(strand(tss)[idx] == "+", 1, -1))
> dist <- (start(peak) - start(tss)[idx]) * sgn
```

Here we summarize the distances as a simple table and density plot, focusing on binding sites within 1000 bases of a transcription start site; the density plot is in Figure 6.

```
> bound <- 1000
> ok <- abs(dist) < bound
> dist <- dist[ok]
> table(sign(dist))

  -1    0    1 
1262    4  707 

> griddensityplot <-
+     function(...)
+     ## 'panel' function to plot a grid underneath density
+ {
+     panel.grid()
+     panel.densityplot(...)
```

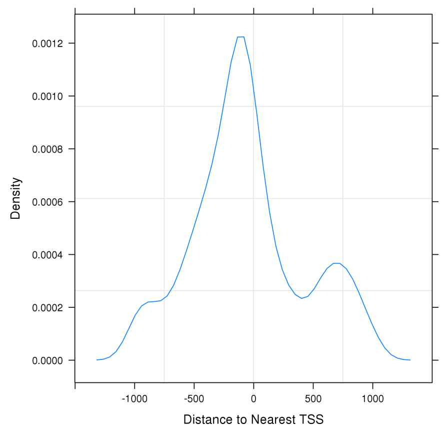

Figure 6: Distance to nearest TSS amongst conserved peaks

```
+ }
> print(densityplot(dist[ok], plot.points=FALSE,
+     panel=griddensityplot,
+     xlab="Distance to Nearest TSS"))
```

The distance to transcript start site is a useful set of operations, so let's make it a re-usable function

```
> distToTss <-
+     function(peak, tx)
+ {
+     peak <- resize(peak, width=1, fix="center")
+     tss <- resize(tx, width=1)
+     idx <- nearest(peak, tss)
+     sgn <- as.numeric(ifelse(strand(tss)[idx] == "+", 1, -1))
+     (start(peak) - start(tss)[idx]) * sgn
+ }
```

Exercise 16
As an additional exercise, extract the sequences of all conserved peaks on 'chr6'. Do this using the BSgenome.Hsapiens.UCSC.hg19 package and `getSeq` function. Use `matchPWM` to find sequences with a strong match to the JASPAR CTCF PWM motif, and plot the density of distances to nearest transcription start site for those with and without a match. What strategies are available for motif discovery?

Solution: Here we select peaks on chromosome 6, and extract the DNA sequences corresponding to these peaks.

```
> library(BSgenome.Hsapiens.UCSC.hg19)
> ridx <- rowSums(assays(stam)[["Tags"]] > 0) == ncol(stam)
> ridx <- ridx & (seqnames(rowData(stam)) == "chr6")
> pk6 <- rowData(stam)[ridx]
> seqs <- getSeq(Hsapiens, pk6, as.character=FALSE)
> head(seqs, 3)
```

```
  A DNAStringSet instance of length 3
    width seq
[1]   311 CAGGGAGACTTGGGAAGGCTTCACGAAGGAGGGT...ACCCAACTCCTAAGCGTCACACATATAATCCTG
[2]   331 GCTAATAATTTACCATGAAGTAACAACTTTTCAC...TTTCCTAGGCAGCGAATTTAAGGGTAATGATCA
[3]   751 GTAAAGAATGGACTGACTTAAAGGCAGATGGAAT...AATCAAACAAGACAAAGAATCTTCGTGGCCACA
```

`matchPWM` operates on one DNA sequence at a time, so we arrange to search for the PWM on each sequence using `lapply`. We identify sequences with a match by testing the length of the returned object, and use this to create a density plot.

```
> pwm <- getJASPAR("MA0139.1.pfm") # useR2012::getJASPAR
> hits <- lapply(seqs, matchPWM, pwm=pwm)
> hasPwmMatch <- sapply(hits, length) > 0
> dist <- distToTss(pk6, tx)
> ok <- abs(dist) < bound
> df <- data.frame(Distance = dist[ok], HasPwmMatch = hasPwmMatch[ok])
> print(densityplot(~Distance, group=HasPwmMatch, df,
+     plot.points=FALSE, panel=griddensityplot,
+     auto.key=list(
+       columns=2,
+       title="Has Position Weight Matrix?",
+       cex.title=1),
+     xlab="Distance to Nearest Tss"))
```

6.3 Exploring Annotated Variants: *VariantAnnotation*

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; full description) 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.

Data are read from a VCF file and variants 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.

Exercise 17
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 sample data in 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.

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

The SNPlocs.Hsapiens.dbSNP.20101109 contains information about SNPs in a particular build of dbSNP. Load the package, use the `dbSNPFilter` function to create a filter, and query the row data of the VCF file for membership.

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

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: VCF
dim: 10376 5
genome: hg19
exptData(1): header
```

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

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

```
GRanges with 3 ranges and 1 elementMetadata col:
              seqnames               ranges strand | paramRangeID
                 <Rle>            <IRanges>  <Rle> |     <factor>
    rs7410291       22 [50300078, 50300078]      * |         <NA>
  rs147922003       22 [50300086, 50300086]      * |         <NA>
  rs114143073       22 [50300101, 50300101]      * |         <NA>
  ---
  seqlengths:
   22
   NA
```

Rename chromosome levels:

```
> rowData(vcf) <- renameSeqlevels(rowData(vcf), c("22"="ch22"))
```

Discover whether SNPs are located in dbSNP:

```
> library(SNPlocs.Hsapiens.dbSNP.20101109)
> snpFilt <- dbSNPFilter("SNPlocs.Hsapiens.dbSNP.20101109")
> inDbSNP <- snpFilt(rowData(vcf), subset=FALSE)
> table(inDbSNP)
```

```
inDbSNP
FALSE  TRUE
 6126  4250
```

Create a data frame summarizing SNP quality and dbSNP membership:

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

Finally, visualize the data, e.g., using *ggplot2* (Figure 7).

```
> 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") +
+     opts(legend.position="top")
```

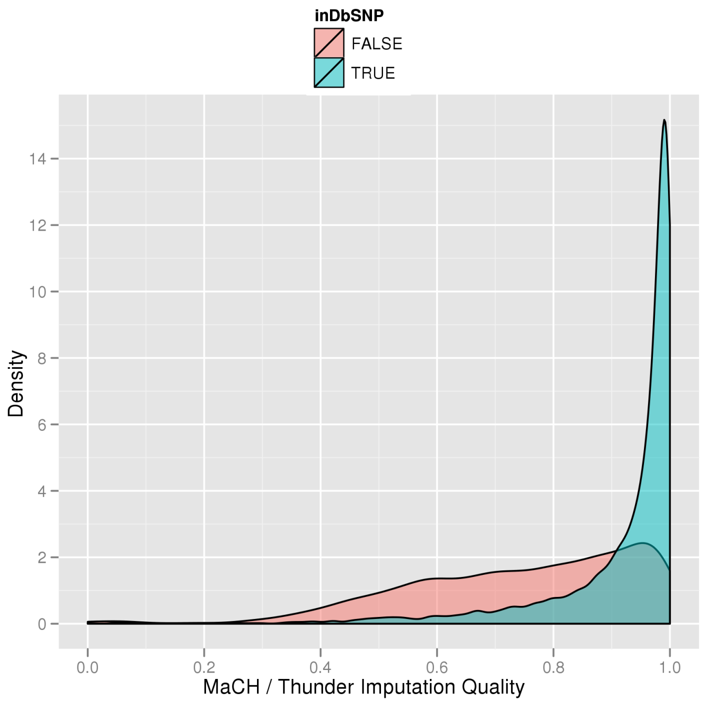

Figure 7: Quality scores of variants in dbSNP, compared to those not in dbSNP.

References

[1] S. Anders and W. Huber. Differential expression analysis for sequence count data. *Genome Biology*, 11:R106, 2010.

[2] A. N. Brooks, L. Yang, M. O. Duff, K. D. Hansen, J. W. Park, S. Dudoit, S. E. Brenner, and B. R. Graveley. Conservation of an RNA regulatory map between Drosophila and mammals. *Genome Research*, pages 193–202, 2011.

[3] J. M. Chambers. *Software for Data Analysis: Programming with R*. Springer, New York, 2008.

[4] P. Dalgaard. *Introductory Statistics with R*. Springer, 2nd edition, 2008.

[5] R. Gentleman. *R Programming for Bioinformatics*. Computer Science & Data Analysis. Chapman & Hall/CRC, Boca Raton, FL, 2008.

[6] J. W. Ho, E. Bishop, P. V. Karchenko, N. Negre, K. P. White, and P. J. Park. ChIP-chip versus ChIP-seq: lessons for experimental design and data analysis. *BMC Genomics*, 12:134, 2011. [PubMed Central:PMC3053263] [DOI:10.1186/1471-2164-12-134] [PubMed:21356108].

[7] P. V. Kharchenko, A. A. Alekseyenko, Y. B. Schwartz, A. Minoda, N. C. Riddle, J. Ernst, P. J. Sabo, E. Larschan, A. A. Gorchakov, T. Gu, D. Linder-Basso, A. Plachetka, G. Shanower, M. Y. Tolstorukov, L. J. Luquette, R. Xi, Y. L. Jung, R. W. Park, E. P. Bishop, T. K. Canfield, R. Sandstrom, R. E. Thurman, D. M. MacAlpine, J. A. Stamatoyannopoulos, M. Kellis, S. C. Elgin, M. I. Kuroda, V. Pirrotta, G. H. Karpen, and P. J. Park. Comprehensive analysis of the chromatin landscape in Drosophila melanogaster. *Nature*, 471:480–485, Mar 2011. [PubMed Central:PMC3109908] [DOI:10.1038/nature09725] [PubMed:21179089].

[8] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.*, 10:R25, 2009.

[9] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25:1754–1760, Jul 2009.

[10] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26:589–595, Mar 2010.

[11] N. Matloff. *The Art of R Programming*. No Starch Pess, 2011.

[12] R. M. Myers, J. Stamatoyannopoulos, M. Snyder, ..., and P. J. Good. A user's guide to the encyclopedia of DNA elements (ENCODE). *PLoS Biol.*, 9:e1001046, Apr 2011. [PubMed Central:PMC3079585] [DOI:10.1371/journal.pbio.1001046] [PubMed:21526222].

[13] P. J. Park. ChIP-seq: advantages and challenges of a maturing technology. *Nat. Rev. Genet.*, 10:669–680, Oct 2009. [PubMed Central:PMC3191340] [DOI:10.1038/nrg2641] [PubMed:19736561].

[14] S. Pepke, B. Wold, and A. Mortazavi. Computation for ChIP-seq and RNA-seq studies. *Nat. Methods*, 6:22–32, Nov 2009. [DOI:10.1038/nmeth.1371] [PubMed:19844228].

[15] M. D. Robinson, D. J. McCarthy, and G. K. Smyth. edgeR: a Bioconductor package for differential expression analysis of digital gene expression data. *Bioinformatics*, 26:139–140, Jan 2010.

[16] P. J. Sabo, M. Hawrylycz, J. C. Wallace, R. Humbert, M. Yu, A. Shafer, J. Kawamoto, R. Hall, J. Mack, M. O. Dorschner, M. McArthur, and J. A. Stamatoyannopoulos. Discovery of functional noncoding elements by digital analysis of chromatin structure. *Proc. Natl. Acad. Sci. U.S.A.*, 101:16837–16842, Nov 2004.