High-throughput sequence analysis with *R* and *Bioconductor*

Marc Carlson, Valerie Obenchain, Hervé Pagès, Paul Shannon,
Daniel Tenenbaum, Martin Morgan[*]

June 2012

Contents

[*]mcarlson,vobencha,hpages,pshannon,dtenenba,mtmorgan@fhcrc.org

Table 1: Tentative schedule.

R / *Bioconductor* for Sequence Analysis
R data types & functions; help; objects; essential packages, efficient programming (Section 2). Working with ranges, strings, reads and alignments. Quality assessment (Sections 3, 4).
RNA-Seq
Differential representation, gene set enrichment, annotation, exon use (Sections 5, 7).
ChIP-Seq
Peak calling (3rd party); collated experiments; motifs; annotation (Section 6, 7).
Variant Annotation
Common work flows; variants in and around genes, amino acid and coding consequences (Sections 8).

1 Introduction

1.1 This workshop

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.

1.2 *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.

Exercise 1
Scavenger hunt. Spend five minutes tracking down the following information.

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

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

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

Solution: Possible solutions from the *Bioconductor* web site are, e.g., `http://bioconductor.org/install/` (installation instructions), `http://bioconductor.org/packages/release/bioc/` (current software packages), and `http://bioconductor.org/help/mailing-list/` (mailing lists).

1.3 High-throughput sequence analysis

Recent technological developments introduce high-throughput sequencing approaches. A variety of experimental protocols and analysis work flows address gene expression, regulation, and encoding of genetic variants. Experimental protocols produce a large number (millions per sample) of short (e.g., 35-100, single or paired-end) nucleotide sequences. These are aligned to a reference or other genome. Analysis work flows use the alignments to infer levels of gene expression (RNA-seq), binding of regulatory elements to genomic locations (ChIP-seq), or prevalence of structural variants (e.g., SNPs, short indels, large-scale genomic rearrangements). Sample sizes range from minimal replication (e.g,. 2 samples per treatment group) to thousands of individuals.

1.4 Statistical programming

Many academic and commercial software products are available; why would one use *R* and *Bioconductor*? One answer is to ask about the demands high-throughput genomic data places on effective computational biology software.

Effective computational biology software High-throughput questions make use of large data sets. This applies both to the primary data (microarray expression values, sequenced reads, etc.) and also to the annotations on those data (coordinates of genes and features such as exons or regulatory regions; participation in biological pathways, etc.). Large data sets place demands on our tools that preclude some standard approaches, such as spread sheets. Likewise, intricate relationships between data and annotation, and the diversity of research questions, require flexibility typical of a programming language rather than a narrowly-enabled graphical user interface.

Analysis of high-throughput data is necessarily statistical. The volume of data requires that it be appropriately summarized before any sort of comprehension is possible. The data are produced by advanced technologies, and these introduce artifacts (e.g., probe-specific bias in microarrays; sequence or base calling bias in RNA-seq experiments) that need to be accommodated to avoid incorrect or inefficient inference. Data sets typically derive from designed experiments, requiring a statistical approach both to account for the design and to correctly address the large number of observed values (e.g., gene expression or sequence tag counts) and small number of samples accessible in typical experiments.

Research needs to be reproducible. Reproducibility is both an ideal of the scientific method, and a pragmatic requirement. The latter comes from the long-term and multi-participant nature of contemporary science. An analysis will be performed for the initial experiment, revisited again during manuscript preparation, and revisited during reviews or in determining next steps. Likewise, analyses typically involve a team of individuals with diverse domains of expertise. Effective collaborations result when it is easy to reproduce, perhaps with minor modifications, an existing result, and when sophisticated statistical or bioinformatic analyses can be effectively conveyed to other group members.

Science moves very quickly. This is driven by the novel questions that are the hallmark of discovery, and by technological innovation and accessibility. Rapidity of scientific development places significant burdens on software, which must also move quickly. Effective software cannot be too polished, because that requires that the correct analyses are 'known' and that significant resources of time and money have been invested in developing the software; this implies software that is tracking the trailing edge of innovation. On the other hand, leading-edge software cannot be too idiosyncratic; it must be usable by a wider audience than the creator of the software, and fit in with other software relevant to the analysis.

Effective software must be accessible. Affordability is one aspect of accessibility. Another is transparent implementation, where the novel software is sufficiently documented and source code accessible enough for the assumptions, approaches, practical implementation decisions, and inevitable coding errors to be assessed by other skilled practitioners. A final aspect of affordability is that the software is actually usable. This is achieved through adequate documentation, support forums, and training opportunities.

***Bioconductor* as effective computational biology software** What features of *R* and *Bioconductor* contribute to its effectiveness as a software tool?

Bioconductor is well suited to handle extensive data and annotation. *Bioconductor* 'classes' represent high-throughput data and their annotation in an integrated way. *Bioconductor* methods use advanced programming techniques or *R* resources (such as transparent data base or network access) to minimize memory requirements and integrate with diverse resources. Classes and methods coordinate complicated data sets with extensive annotation. Nonetheless, the basic model for object manipulation in *R* involves vectorized in-memory representations. For this reason, particular programming paradigms (e.g., block processing of data streams; explicit parallelism) or hardware resources (e.g., large-memory computers) are sometimes required when dealing with extensive data.

R is ideally suited to addressing the statistical challenges of high-throughput data. Three examples include the development of the 'RMA' and other normalization algorithm for microarray pre-processing, use of moderated t-statistics for assessing microarray differential expression, and development of negative binomial approaches to estimating dispersion read counts necessary for appropriate analysis of RNAseq designed experiments.

Many of the 'old school' aspects of *R* and *Bioconductor* facilitate reproducible research. An analysis is often represented as a text-based script. Reproducing the analysis involves re-running the script; adjusting how the analysis is performed involves simple text-editing tasks. Beyond this, *R* has the notion of a 'vignette', which represents an analysis as a LaTeX document with embedded *R* commands. The *R* commands are evaluated when the document is built, thus reproducing the analysis. The use of LaTeX means that the symbolic manipulations in the script are augmented with textual explanations and justifications for the approach taken; these include graphical and tabular summaries at appropriate places in the analysis. *R* includes facilities for reporting the exact version of *R* and associated packages used in an analysis so that, if needed, discrepancies between software versions can be tracked down and their importance evaluated. While users often think of *R* packages as providing new functionality, packages are also used to enhance reproducibility by encapsulating a single analysis. The package can contain data sets, vignette(s) describing the analysis, *R* functions that might have been written, scripts for key data processing stages, and documentation (via standard *R* help mechanisms) of what the functions, data, and packages are about.

The *Bioconductor* project adopts practices that facilitate reproducibility. Versions of *R* and *Bioconductor* are released twice each year. Each *Bioconductor* release is the result of development, in a separate branch, during the previous six months. The release is built daily against the corresponding version of *R* on Linux, Mac, and Windows platforms, with an extensive suite of tests performed. The `biocLite` function ensures that each release of *R* uses the corresponding *Bioconductor* packages. The user thus has access to stable and tested package versions. *R* and *Bioconductor* are effective tools for reproducible research.

R and *Bioconductor* exist on the leading portion of the software life cycle. Contributors are primarily from academic institutions, and are directly involved in novel research activities. New developments are made available in a familiar format, i.e., the *R* language, packaging, and build systems. The rich set of facilities in *R* (e.g., for advanced statistical analysis or visualization) and the extensive resources in *Bioconductor* (e.g., for annotation using third-party data such as Biomart or UCSC genome browser tracks) mean that innovations can be directly incorporated into existing work flows. The 'development' branches of *R* and *Bioconductor* provide an environment where contributors can explore new approaches without alienating their user base.

R and *Bioconductor* also fair well in terms of accessibility. The software is freely available. The source code is easily and fully accessible for critical evaluation. The *R* packaging and check system requires that all functions are documented. *Bioconductor* requires that each package contain vignettes to illustrate the use of the software. There are very active *R* and *Bioconductor* mailing lists for immediate support, and regular training and conference activities for professional development.

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*.

1.5 *Bioconductor* for 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*).

1.6 Resources

Dalgaard [4] provides an introduction to statistical analysis with *R*. Kabaloff [9] provides a broad survey of *R*. Matloff [15] 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.

The RStudio environment provides a nice, cross-platform environment for working in *R*.

2 *R*

R is an open-source statistical programming language. It is used to manipulate data, to perform statistical analyses, and to present graphical and other results. *R* consists of a core language, additional 'packages' distributed with the *R* language, and a very large number of packages contributed by the broader community. Packages add specific functionality to an *R* installation. *R* has become the primary language of academic statistical analyses, and is widely used in diverse areas of research, government, and industry.

R has several unique features. It has a surprisingly 'old school' interface: users type commands into a console; scripts in plain text represent work flows; tools other than *R* are used for editing and other tasks. *R* is a flexible programming language, so while one person might use functions provided by *R* to accomplish advanced analytic tasks, another might implement their own functions for novel data types. As a programming language, *R* adopts syntax and grammar that differ from many other languages: objects in *R* are 'vectors', and functions are 'vectorized' to operate on all elements of the object; *R* objects have 'copy on change' and 'pass by value' semantics, reducing unexpected consequences for users at the expense of less efficient memory use; common paradigms in other languages, such as the 'for' loop, are encountered much less commonly in *R*. Many authors contribute to *R*, so there can be a frustrating inconsistency of documentation and interface. *R* grew up in the academic community, so authors have not shied away from trying new approaches. Common statistical analyses are very well-developed.

2.1 *R* data types

Opening an *R* session results in a prompt. The user types instructions at the prompt. Here is an example:

```
> ## assign values 5, 4, 3, 2, 1 to variable 'x'
> x <- c(5, 4, 3, 2, 1)
> x
```

```
[1] 5 4 3 2 1
```

The first line starts with a `#` to represent a comment; the line is ignored by *R*. The next line creates a variable `x`. The variable is assigned (using `<-`, we could have used `=` almost interchangeably) a value. The value assigned is the result of a call to the `c` function. That it is a function call is indicated by the symbol named followed by parentheses, `c()`. The `c` function takes zero or more arguments, and returns a vector. The vector is the value assigned to `x`. *R* responds to this line with a new prompt, ready for the next input. The next line asks *R* to display the value of the variable `x`. *R* responds by printing `[1]` to indicate that the subsequent number is the first element of the vector. It then prints the value of `x`.

R has many features to aid common operations. Entering sequences is a very common operation, and expressions of the form `2:4` create a sequence from `2` to `4`. Subsetting one vector by another is enabled with `[`. Here we create an integer sequence from 2 to 4, and use the sequence as an index to select the second, third, and fourth elements of `x`

```
> x[2:4]
```

```
[1] 4 3 2
```

R functions operate on variables. Functions are usually vectorized, acting on all elements of their argument and obviating the need for explicit iteration. Functions can generate warnings when performing suspect operations, or errors if evaluation cannot proceed; try `log(-1)`.

```
> log(x)
```

```
[1] 1.6094379 1.3862944 1.0986123 0.6931472 0.0000000
```

Essential data types *R* has a number of standard data types, to 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 type or mode of a variable.

```
> c(1.1, 1.2, 1.3)         # numeric
```

```
[1] 1.1 1.2 1.3
```

```
> c(FALSE, TRUE, FALSE)    # logical
```

```
[1] FALSE  TRUE FALSE
```

```
> c("foo", "bar", "baz")   # character, single or double quote ok
```

```
[1] "foo" "bar" "baz"
```

```
> as.character(x)          # convert 'x' to character
```

```
[1] "5" "4" "3" "2" "1"
```

```
> typeof(x)                # the number 5 is numeric, not integer
```

```
[1] "double"
```

```
> typeof(2L)               # append 'L' to force integer
```

```
[1] "integer"
```

```
> typeof(2:4)              # ':' produces a sequence of integers
```

```
[1] "integer"
```

R includes data types particularly useful for statistical analysis, including `factor` to represent categories and `NA` (used in any vector) to represent missing values.

```
> sex <- factor(c("Male", "Female", NA), levels=c("Female", "Male"))
> sex
```

```
[1] Male   Female <NA>
Levels: Female Male
```

Lists, data frames, and matrices All of the vectors mentioned so far are homogeneous, 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.

```
> lst <- list(a=1:3, b=c("foo", "bar"), c=sex)
> lst

$a
[1] 1 2 3

$b
[1] "foo" "bar"

$c
[1] Male   Female <NA>
Levels: Female Male
```

Lists can be subset like other vectors to get another list, or subset with `[[` to retrieve the actual list element; as with other vectors, subsetting can use names

```
> lst[c(3, 1)]             # another list

$c
[1] Male   Female <NA>
Levels: Female Male

$a
[1] 1 2 3

> lst[["a"]]               # the element itself, selected by name

[1] 1 2 3
```

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 `[[`.

```
> df <- data.frame(age=c(27L, 32L, 19L),
+                  sex=factor(c("Male", "Female", "Male")))
> df

  age    sex
1  27   Male
2  32 Female
3  19   Male

> df[c(1, 3),]

  age  sex
1  27 Male
3  19 Male
```

```
> df[df$age > 20,]

  age    sex
1  27   Male
2  32 Female
```

A `matrix` is also a rectangular data structure, but subject to the constraint that all elements are the same type. A matrix is created by taking a vector, and specifying the number of rows or columns the vector is to represent. On subsetting, *R* coerces a single column `data.frame` or single row or column `matrix` to a vector if possible; use `drop=FALSE` to stop this behavior.

```
> m <- matrix(1:12, nrow=3)
> m

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> m[c(1, 3), c(2, 4)]

     [,1] [,2]
[1,]    4   10
[2,]    6   12

> m[, 3]

[1] 7 8 9

> m[, 3, drop=FALSE]

     [,1]
[1,]    7
[2,]    8
[3,]    9
```

An `array` is a data structure for representing homogenous, rectangular data in higher dimensions.

S3 and S4 classes More complicated data structures are represented using the 'S3' or 'S4' object system. Objects are often created by functions (for example, `lm`, below), with parts of the object extracted or assigned using *accessor* functions. The following generates 1000 random normal deviates as `x`, and uses these to create another 1000 deviates `y` that are linearly related to `x` but with some error. We fit a linear regression using a 'formula' to describe the relationship between variables, summarize the results in a familiar ANOVA table, and access `fit` (an S3 object) for the residuals of the regression, using these as input first to the `var` (variance) and then `sqrt` (square-root) functions. Objects can be interrogated for their class.

```
> x <- rnorm(1000, sd=1)
> y <- x + rnorm(1000, sd=.5)
> fit <- lm(y ~ x)       # formula describes linear regression
> fit                    # an 'S3' object

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
    0.01043      1.00035

> anova(fit)

Analysis of Variance Table

Response: y
           Df Sum Sq Mean Sq F value    Pr(>F)
x           1 945.90  945.90  3756.4 < 2.2e-16 ***
Residuals 998 251.31    0.25
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

> sqrt(var(resid(fit)))  # residuals accessor and subsequent transforms

[1] 0.5015571

> class(fit)

[1] "lm"
```

Many *Bioconductor* packages implement S4 objects to represent data. S3 and S4 systems are quite different from a programmer's perspective, but fairly similar from a user's perspective: both systems encapsulate complicated data structures, and allow for methods specialized to different data types; accessors are used to extract information from the objects.

Functions *R* functions accept arguments, and return values. Arguments can be required or optional. Some functions may take variable numbers of arguments, e.g., the columns in a `data.frame`

```
> y <- 5:1
> log(y)

[1] 1.6094379 1.3862944 1.0986123 0.6931472 0.0000000

> args(log)        # arguments 'x' and 'base'; see ?log

function (x, base = exp(1))
NULL

> log(y, base=2)   # 'base' is optional, with default value
```

```
[1] 2.321928 2.000000 1.584963 1.000000 0.000000

> try(log())       # 'x' required; 'try' continues even on error
> args(data.frame) # ... represents variable number of arguments

function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE,
    stringsAsFactors = default.stringsAsFactors())
NULL
```

Arguments can be matched by name or position. If an argument appears after ..., it must be named.

```
> log(base=2, y)   # match argument 'base' by name, 'x' by position

[1] 2.321928 2.000000 1.584963 1.000000 0.000000
```

A function such as `anova` is a *generic* that provides an overall signature but dispatches the actual work to the *method* corresponding to the class(es) of the arguments used to invoke the generic. A generic may have fewer arguments than a method, as with the S3 function `anova` and its method `anova.glm`.

```
> args(anova)

function (object, ...)
NULL

> args(anova.glm)

function (object, ..., dispersion = NULL, test = NULL)
NULL
```

The ... argument in the `anova` generic means that additional arguments are possible; the `anova` generic hands these arguments to the method it dispatches to.

2.2 Useful functions

R has a very large number of functions. The following is a brief list of those that might be commonly used and particularly useful.

`dir`, `read.table` (and friends), `scan` List files in a directory, read spreadsheet-like data into *R*, efficiently read homogenous data (e.g., a file of numeric values) to be represented as a matrix.

`c`, `factor`, `data.frame`, `matrix` Create a vector, factor, data frame or matrix.

`summary`, `table`, `xtabs` Summarize, create a table of the number of times elements occur in a vector, cross-tabulate two or more variables.

`t.test`, `aov`, `lm`, `anova` Basic comparison of two (`t.test`) groups, or several groups via analysis of variance / linear models (`aov` output is probably more familiar to biologists), or compare simpler with more complicated models (`anova`).

`dist`, `hclust` Cluster data.

`plot` Plot data.

`ls, str, library, search` List objects in the current (or specified) workspace, or peak at the structure of an object; add a library to or describe the search path of attached packages.

`lapply, sapply, mapply` Apply a function to each element of a list (`lapply`, `sapply`) or to elements of several lists (`mapply`).

`with` Conveniently access columns of a data frame or other element without having to repeat the name of the data frame.

`match, %in%` Report the index or existence of elements from one vector that match another.

`split, cut` Split one vector by an equal length factor, cut a single vector into intervals encoded as levels of a factor.

`strsplit, grep, sub` Operate on character vectors, splitting it into distinct fields, searching for the occurrence of a patterns using regular expressions (see `?regex`, or substituting a string for a regular expression.

`install.packages` Install a package from an on-line repository into your *R*.

`traceback, debug, browser` Report the sequence of functions under evaluatino at the time of the error; enter a debugger when a particular function or statement is invoked.

See the help pages (e.g., `?lm`) and examples (`example(match)`) for each of these functions

Exercise 2
This exercise uses data describing 128 microarray samples as a basis for exploring R functions. Covariates such as age, sex, type, stage of the disease, etc., are in a data file `pData.csv`.

The following command creates a variable `pdataFiles` that is the location of a comma-separated value ('csv') file to be used in the exercise. A csv file can be created using, e.g., 'Save as...' in spreadsheet software.

```
> pdataFile <- system.file(package="SequenceAnalysisData", "extdata",
+                          "pData.csv")
```

Input the csv file using `read.table`, assigning the input to a variable `pdata`. Use `dim` to find out the dimensions (number of rows, number of columns) in the object. Are there 128 rows? Use `names` or `colnames` to list the names of the columns of `pdata`. Use `summary` to summarize each column of the data. What are the data types of each column in the data frame?

A data frame is a list of equal length vectors. Select the 'sex' column of the data frame using `[[` or `$`. Pause to explain to your neighbor why this subsetting works. Since a data frame is a list, use `sapply` to ask about the class of each column in the data frame. Explain to your neighbor what this produces, and why.

Use `table` to summarize the number of males and females in the sample. Consult the help page `?table` to figure out additional arguments required to include `NA` values in the tabulation.

The `mol.biol` column summarizes molecular biological attributes of each sample. Use `table` to summarize the different molecular biology levels in the sample. Use `%in%` to create a logical vector of the samples that are either `BCR/ABL` or `NEG`. Subset the original phenotypic data to contain those samples that are `BCR/ABL` or `NEG`.

After subsetting, what are the levels of the `mol.biol` column? Set the levels to be `BCR/ABL` and `NEG`, i.e., the levels in the subset.

One would like covariates to be similar across groups of interest. Use `t.test` to assess whether `BCR/ABL` and `NEG` have individuals with similar age. To do this, use a `formula` that describes the response `age` in terms of the predictor `mol.biol`. If age is not independent of molecular biology, what complications might this introduce into subsequent analysis? Use

Solution: Here we input the data and explore basic properties.

```
> pdata <- read.table(pdataFile)
> dim(pdata)
```

```
[1] 128  21
```

```
> names(pdata)
```

```
 [1] "cod"            "diagnosis"      "sex"            "age"
 [5] "BT"             "remission"      "CR"             "date.cr"
 [9] "t.4.11."        "t.9.22."        "cyto.normal"    "citog"
[13] "mol.biol"       "fusion.protein" "mdr"            "kinet"
[17] "ccr"            "relapse"        "transplant"     "f.u"
[21] "date.last.seen"
```

```
> summary(pdata)
```

```
      cod            diagnosis     sex          age               BT
 10005   :  1   11/15/1997:  2   F    :42   Min.   : 5.00   B2     :36
 1003    :  1   1/15/1997 :  2   M    :83   1st Qu.:19.00   B3     :23
 remission                 CR           date.cr    t.4.11.
 CR  :99   CR              :96   11/11/1997: 3   Mode :logical
 REF :15   DEATH IN CR     : 3   10/18/1999: 2   FALSE:86
  t.9.22.        cyto.normal               citog        mol.biol
 Mode :logical   Mode :logical   normal       :24   ALL1/AF4:10
 FALSE:67        FALSE:69        simple alt.  :15   BCR/ABL :37
   fusion.protein   mdr          kinet         ccr            relapse
 p190     :17     NEG :101   dyploid:94   Mode :logical   Mode :logical
 p190/p210: 8     POS : 24   hyperd.:27   FALSE:74        FALSE:35
 transplant                     f.u        date.last.seen
 Mode :logical   REL              :61   12/15/1997: 2
 FALSE:91        CCR              :23   12/31/2002: 2
 [ reached getOption("max.print") -- omitted 5 rows ]
```

A data frame can be subset as if it were a matrix, or a list of column vectors.

```
> head(pdata[,"sex"], 3)
```

```
[1] M M F
Levels: F M
```

```
> head(pdata$sex, 3)
```

```
[1] M M F
Levels: F M
```

```
> head(pdata[["sex"]], 3)

[1] M M F
Levels: F M

> sapply(pdata, class)

           cod      diagnosis            sex            age             BT
      "factor"       "factor"       "factor"      "integer"       "factor"
     remission             CR        date.cr        t.4.11.        t.9.22.
      "factor"       "factor"       "factor"      "logical"      "logical"
   cyto.normal          citog       mol.biol fusion.protein            mdr
     "logical"       "factor"       "factor"       "factor"       "factor"
         kinet            ccr        relapse     transplant            f.u
      "factor"      "logical"      "logical"      "logical"       "factor"
date.last.seen
      "factor"
```

The number of males and females, including `NA`, is

```
> table(pdata$sex, useNA="ifany")

   F    M <NA>
  42   83    3
```

An alternative version of this uses the `with` function: `with(pdata, table(sex, useNA="ifany"))`.

The `mol.biol` column contains the following samples:

```
> with(pdata, table(mol.biol, useNA="ifany"))

mol.biol
ALL1/AF4  BCR/ABL E2A/PBX1      NEG   NUP-98  p15/p16
      10       37        5       74        1        1
```

A logical vector indicating that the corresponding row is either `BCR/ABL` or `NEG` is constructed as

```
> ridx <- pdata$mol.biol %in% c("BCR/ABL", "NEG")
```

We can get a sense of the number of rows selected via `table` or `sum` (discuss with your neighbor what `sum` does, and why the answer is the same as the number of `TRUE` values in the result of the `table` function).

```
> table(ridx)

ridx
FALSE  TRUE
   17   111

> sum(ridx)

[1] 111
```

The original data frame can be subset to contain only BCR/ABL or NEG samples using the logical vector ridx that we created.

```
> pdata1 <- pdata[ridx,]
```

The levels of each factor reflect the levels in the original object, rather than the levels in the subset object, e.g.,

```
> levels(pdata$mol.biol)

[1] "ALL1/AF4" "BCR/ABL"  "E2A/PBX1" "NEG"      "NUP-98"   "p15/p16"
```

These can be re-coded by updating the new data frame to contain a factor with the desired levels.

```
> pdata1$mol.biol <- factor(pdata1$mol.biol)
> table(pdata1$mol.biol)

BCR/ABL     NEG
     37      74
```

To ask whether age differs between molecular biology types, we use a formula age ~ mol.biol to describe the relationship ('age as a function of molecular biology') that we wish to test

```
> with(pdata1, t.test(age ~ mol.biol))

        Welch Two Sample t-test

data:  age by mol.biol
t = 4.8172, df = 68.529, p-value = 8.401e-06
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  7.13507 17.22408
sample estimates:
mean in group BCR/ABL     mean in group NEG
             40.25000              28.07042
```

This summary can be visualize with, e.g., the boxplot function

```
> ## not evaluated
> boxplot(age ~ mol.biol, pdata1)
```

Molecular biology seem to be strongly associated with age; individuals in the NEG group are considerably younger than those in the BCR/ABL group. We might wish to include age as a covariate in any subsequent analysis seeking to relate molecular biology to gene expression.

Table 3: Selected base and contributed packages.

Package	Description
base	Data input and essential manipulation; scripting and programming concepts.
stats	Essential statistical and plotting functions.
lattice, *ggplot2*	Approaches to advanced graphics.
methods	'S4' classes and methods.
parallel	Facilities for parallel evaluation.

2.3 Packages

Packages provide functionality beyond that available in base *R*. There are over 3000 packages in CRAN (comprehensive *R* archive network) and 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. Table 3 outlines key base packages and selected contributed packages; see a local CRAN mirror (including the task views summarizing packages in different domains) and *Bioconductor* for additional contributed packages.

The *lattice* package illustrates the value packages add to base *R*. *lattice* is distributed with *R* but not loaded by default. It provides a very expressive way to visualize data. The following example plots yield for a number of barley varieties, conditioned on site and grouped by year. Figure 1 is read from the lower left corner. Note the common scales, efficient use of space, and not-too-pleasing default color palette. The Morris sample appears to be mis-labeled for 'year', an apparent error in the original data. Find out about the built-in data set used in this example with `?barley`.

```
> library(lattice)
> dotplot(variety ~ yield | site, data = barley, groups = year,
+         key = simpleKey(levels(barley$year), space = "right"),
+         xlab = "Barley Yield (bushels/acre)",
+         aspect=0.5, layout = c(2,3), ylab=NULL)
```

New packages can be added to an *R* installation using `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. Loading a package also loads any package that it depends on. Packages loaded in the current session are displayed with `search`. The ordering of packages returned by `search` represents the order in which the global environment (where commands entered at the prompt are evaluated) and attached packages are searched for symbols; it is possible for a package earlier in the search path to mask symbols later in the search path; these can be disambiguated using `::`.

```
> length(search())
```

```
[1] 11
```

```
> search()
```

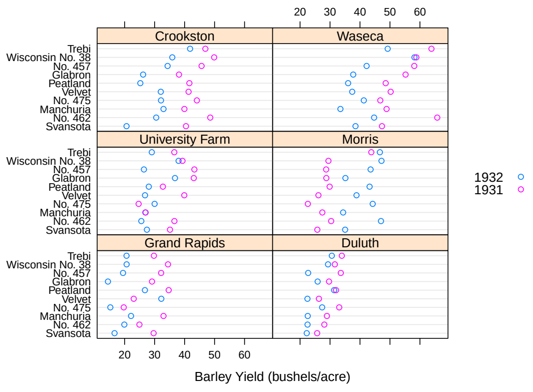


Figure 1: Variety yield conditional on site and grouped by year, for the `barley` data set.

```
 [1] ".GlobalEnv"              "package:lattice"         "package:BiocInstaller"
 [4] "package:stats"           "package:graphics"        "package:grDevices"
 [7] "package:utils"           "package:datasets"        "package:methods"
[10] "Autoloads"               "package:base"
```

```
> base::log(1:3)
```

```
[1] 0.0000000 0.6931472 1.0986123
```

Exercise 3
Use the `library` *function to load the SequenceAnalysis package. Use the* `sessionInfo` *function to verify that you are using R version 2.15.0 and current packages, similar to those reported here. What other packages were loaded along with SequenceAnalysis?*

Solution:

```
> library(SequenceAnalysis)
> sessionInfo()
```

2.4 Help

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.

Manual pages 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 as

```
> ?data.frame
> ?lm
> ?anova             # a generic function
> ?anova.lm          # an S3 method, specialized for 'lm' objects
```

S3 methods can be queried interactively. For S3,

```
> methods(anova)

[1] anova.glm     anova.glmlist anova.lm      anova.loess*  anova.mlm
[6] anova.nls*

   Non-visible functions are asterisked

> methods(class="glm")

 [1] add1.glm*           anova.glm           confint.glm*
 [4] cooks.distance.glm* deviance.glm*       drop1.glm*
 [7] effects.glm*        extractAIC.glm*     family.glm*
[10] formula.glm*        influence.glm*      logLik.glm*
[13] model.frame.glm     nobs.glm*           predict.glm
[16] print.glm           residuals.glm       rstandard.glm
[19] rstudent.glm        summary.glm         vcov.glm*
[22] weights.glm*

   Non-visible functions are asterisked
```

It is often useful to view a method definition, either by typing the method name at the command line or, for 'non-visible' methods, using `getAnywhere`:

```
> anova.lm
> getAnywhere("anova.loess")
```

For instance, the source code of a function is printed if the function is invoked without parentheses. Here we discover that the function `head` (which returns the first 6 elements of anything) defined in the *utils* package, is an S3 generic (indicated by `UseMethod`) and has several methods. We use `head` to look at the first six lines of the `head` method specialized for `matrix` objects.

```
> utils::head

function (x, ...)
UseMethod("head")
<environment: namespace:utils>

> methods(head)

[1] head.data.frame* head.default*    head.ftable*     head.function*
[5] head.matrix      head.table*

   Non-visible functions are asterisked
```

```
> head(head.matrix)

1 function (x, n = 6L, ...)
2 {
3     stopifnot(length(n) == 1L)
4     n <- if (n < 0L)
5         max(nrow(x) + n, 0L)
6     else min(n, nrow(x))
```

S4 classes and generics are queried in a similar way to S3 classes and generics, but with different syntax, as 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"
```

The specification of method and class in the latter must not contain a space after the comma. The definition of a method can be retrieved as

```
> selectMethod(complement, "DNAStringSet")
```

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

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

to see, in your web browser, vignettes available in the *SequenceAnalysis* 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.

Exercise 4
Scavenger hunt. Spend five minutes tracking down the following information.

a. *The package containing the* `library` *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.*

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

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

2.5 Efficient scripts

There are often many ways to accomplish a result in *R*, but these different ways often have very different speed or memory requirements. For small data sets these performance differences are not that important, but for large data sets (e.g., high-throughput sequencing; genome-wide association studies, GWAS) or complicated calculations (e.g., bootstrapping) performance can be important. There are several approaches to achieving efficient *R* programming.

Easy solutions Several common performance bottlenecks often have easy solutions; these are outlined here.

Text files often contain more information, for example 1000's of individuals at millions of SNPs, when only a subset of the data is required, e.g., during algorithm development. Reading in all the data can be demanding in terms of both memory and time. A solution is to use arguments such as `colClasses` to specify the columns and their data types that are required, and to use `nrow` to limit the number of rows input. For example, the following ignores the first and fourth column, reading in only the second and third (as type `integer` and `numeric`).

```
> ## not evaluated
> colClasses <-
+   c("NULL", "integer", "numeric", "NULL")
> df <- read.table("myfile", colClasses=colClasses)
```

R is vectorized, so traditional programming `for` loops are often not necessary. Rather than calculating 100000 random numbers one at a time, or squaring each element of a vector, or iterating over rows and columns in a matrix to calculate row sums, invoke the single function that performs each of these operations.

```
> x <- runif(100000); x2 <- x^2
> m <- matrix(x2, nrow=1000); y <- rowSums(m)
```

This often requires a change of thinking, turning the sequence of operations 'inside-out'. For instance, calculate the log of the square of each element of a vector by calculating the square of all elements, followed by the log of all elements `x2 <- x^2; x3 <- log(x2)`, or simply `logx2 <- log(x^2)`.

It may sometimes be natural to formulate a problem as a `for` loop, or the formulation of the problem may require that a `for` loop be used. In these circumstances the appropriate strategy is to pre-allocate the `result` object, and to fill the result in during loop iteration.

```
> ## not evaluated
> result <- numeric(nrow(df))
> for (i in seq_len(nrow(df)))
+   result[[i]] <- some_calc(df[i,])
```

Some *R* operations are helpful in general, but misleading or inefficient in particular circumstances. An example is the behavior of `unlist` when the list is named – *R* creates new names that have been made unique. This can be confusing (e.g., when Entrez gene identifiers are 'mangled' to unintentionally look like other identifiers) and expensive (when a large number of new names need to be created). Avoid creating unnecessary names, e.g.,

```
> unlist(list(a=1:2)) # name 'a' becomes 'a1', 'a2'

a1 a2
 1  2

> unlist(list(a=1:2), use.names=FALSE)   # no names

[1] 1 2
```

Names can be very useful for avoiding book-keeping errors, but are inefficient for repeated look-ups; use vectorized access or numeric indexing.

Moderate solutions Several solutions to inefficient code require greater knowledge to implement.

Using appropriate functions can greatly influence performance; it takes experience to know when an appropriate function exists. For instance, the `lm` function could be used to assess differential expression of each gene on a microarray, but the *limma* package implements this operation in a way that takes advantage of the experimental design that is common to each probe on the microarray, and does so in a very efficient manner.

```
> ## not evaluated
> library(limma) # microarray linear models
> fit <- lmFit(eSet, design)
```

Using appropriate algorithms can have significant performance benefits, especially as data becomes larger. This solution requires moderate skills, because one has to be able to think about the complexity (e.g., expected number of operations) of an algorithm, and to identify algorithms that accomplish the same goal in fewer steps. For example, a naive way of identifying which of 100 numbers are in a set of size 10 might look at all 100×10 combinations of numbers (i.e., polynomial time), but a faster way is to create a 'hash' table of one of the set of elements and probe that for each of the other elements (i.e., linear time). The latter strategy is illustrated with

```
> x <- 1:100; s <- sample(x, 10)
> inS <- x %in% s
```

R is an interpreted language, and for very challenging computational problems it may be appropriate to write critical stages of an analysis in a compiled language like C or Fortran, or to use an existing programming library (e.g., the BOOST graph library) that efficiently implements advanced algorithms. *R* has a well-developed interface to C or Fortran, so it is 'easy' to do this. This places a significant burden on the person implementing the solution, requiring knowledge of two or more computer languages and of the interface between them.

Measuring performance When trying to improve performance, one wants to ensure (a) that the new code is actually faster than the previous code, and (b) both solutions arrive at the same, correct, answer.

The `system.time` function is a straight-forward way to measure the length of time a portion of code takes to evaluate. Here we see that the use of `apply` to calculate row sums of a matrix is much less efficient than the specialized `rowSums` function.

```
> m <- matrix(runif(200000), 20000)
> replicate(5, system.time(apply(m, 1, sum))[[1]])

[1] 0.176 0.168 0.168 0.164 0.168

> replicate(5, system.time(rowSums(m))[[1]])

[1] 0.000 0.004 0.000 0.000 0.000
```

Usually it is appropriate to replicate timings to average over vagaries of system use, and to shuffle the order in which timings of alternative algorithms are calculated to avoid artifacts such as initial memory allocation.

Speed is an important metric, but equivalent results are also needed. The functions `identical` and `all.equal` provide different levels of assessing equivalence, with `all.equal` providing ability to ignore some differences, e.g., in the names of vector elements.

```
> res1 <- apply(m, 1, sum)
> res2 <- rowSums(m)
> identical(res1, res2)

[1] TRUE

> identical(c(1, -1), c(x=1, y=-1))

[1] FALSE

> all.equal(c(1, -1), c(x=1, y=-1),
+           check.attributes=FALSE)

[1] TRUE
```

Two additional functions for assessing performance are `Rprof` and `tracemem`; these are mentioned only briefly here. The `Rprof` function profiles *R* code, presenting a summary of the time spent in each part of several lines of *R* code. It is useful for gaining insight into the location of performance bottlenecks when these are not readily apparent from direct inspection. Memory management, especially copying large objects, can frequently contribute to poor performance. The `tracemem` function allows one to gain insight into how *R* manages memory; insights from this kind of analysis can sometimes be useful in restructuring code into a more efficient sequence.

2.6 Warnings, errors, and debugging

R signals unexpected results through warnings and errors. Warnings occur when the calculation produces an unusual result that nonetheless does not preclude further evaluation. For instance `log(-1)` results in a value `NaN` ('not a number') that allows computation to continue, but at the same time signals an warning

```
> log(-1)
[1] NaN
Warning message:
In log(-1) : NaNs produced
```

Errors result when the inputs or outputs of a function are such that no further action can be taken, e.g., trying to take the square root of a character vector

```
> sqrt("two")
Error in sqrt("two") : Non-numeric argument to mathematical function
```

Warnings and errors occurring at the command prompt are usually easy to diagnose. They can be more enigmatic when occurring in a function, and exacerbated by sometimes cryptic (when read out of context) error messages.

An initial step in coming to terms with errors is to simplify the problem as much as possible, aiming for a 'reproducible' error. The reproducible error might involve a very small (even trivial) data set that immediately provokes the error. Often the process of creating a reproducible example helps to clarify what the error is, and what possible solutions might be.

Invoking `traceback()` immediately after an error occurs provides a 'stack' of the function calls that were in effect when the error occurred. This can help understand the context in which the error occurred. Knowing the context, one might use `debug` to enter into a browser (see `?browser`) that allows one to step through the function in which the error occurred.

It can sometimes be useful to use global options (see `?options`) to influence what happens when an error occurs. Two common global options are `error` and `warn`. Setting `error=recover` combines the functionality of `traceback` and `debug`, allowing the user to enter the browser at any level of the call stack in effect at the time the error occurred. Default error behavior can be restored with `options(error=NULL)`. Setting `warn=2` causes warnings to be promoted to errors. For instance, initial investigation of an error might show that the error occurs when one of the arguments to a function has value `NaN`. The error might be accompanied by a warning message that the `NaN` has been introduced, but because warnings are by default not reported immediately it is not clear where the `NaN` comes from. `warn=2` means that the warning is treated as an error, and hence can be debugged using `traceback`, `debug`, and so on.

Additional useful debugging functions include `browser`, `trace`, and `setBreak-point`.

Fixme: tryCatch

Table 4: 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.

3 Ranges and Strings

Many *Bioconductor* packages are available for analysis of high-throughput sequence data. This section introduces two essential ways in which sequence data are manipulated. Ranges describe both aligned reads and features of interest on the genome. Sets of DNA strings represent the reads themselves and the nucleotide sequence of reference genomes. Key packages are summarized in Table 4.

3.1 Genomic ranges

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 the *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
```

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 7.

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. A *GRanges* instance can be subset, with accessors for getting and updating information.

```
> genes[2]
```

```
GRanges with 1 range and 0 elementMetadata cols:
      seqnames               ranges strand
         <Rle>            <IRanges>  <Rle>
```

```
  [1]         X [18962306, 18962925]      -
  ---
  seqlengths:
        3R        X
   27905053 22422827

> strand(genes)

'factor' Rle of length 2 with 2 runs
  Lengths: 1 1
  Values : + -
Levels(3): + - *

> width(genes)

[1] 6096  620

> length(genes)

[1] 2

> names(genes) <- c("FBgn0039155", "FBgn0085359")
> genes  # now with names

GRanges with 2 ranges and 0 elementMetadata cols:
              seqnames               ranges strand
                 <Rle>            <IRanges>  <Rle>
  FBgn0039155       3R [19967117, 19973212]      +
  FBgn0085359        X [18962306, 18962925]      -
  ---
  seqlengths:
        3R        X
   27905053 22422827
```

`strand` returns the strand information in a compact representation called a *run-length encoding*, this is introduced in greater detail below. The 'names' could have been specified when the instance was constructed; once named, the *GRanges* instance can be subset by name like a regular vector.

As the `GRanges` function suggests, the *GRanges* class extends the *IRanges* class by adding information about `seqname`, `strand`, and other information particularly relevant to representing ranges that are on genomes. The *IRanges* class and related data structures (e.g., *RangedData*) are meant as a more general description of ranges defined in an arbitrary space. Many methods implemented on the *GRanges* class are 'aware' of the consequences of genomic location, for instance treating ranges on the minus strand differently (reflecting the 5' orientation imposed by DNA) from ranges on the plus strand.

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:

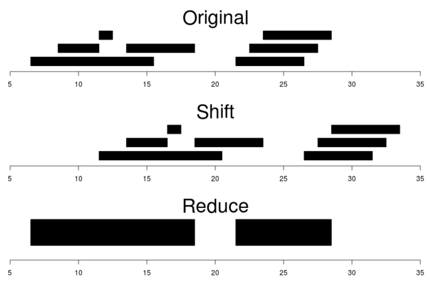


Figure 2: Ranges

```
> 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 2 and summarized in Table 5.

Methods on ranges can be grouped as follows:

Intra-range methods act on each range independently. These include `flank`, `narrow`, `reflect`, `resize`, `restrict`, and `shift`, among others. An illustration is `shift`, which translates each range by the amount specified by the `shift` argument. Positive values shift to the right, negative to the left; `shift` can be a vector, with each element of the vector shifting the corresponding element of the *IRanges* instance. Here we shift all ranges to the right by 5, with the result illustrated in the middle panel of Figure 2.

```
> shift(ir, 5)

IRanges of length 7
    start end width
[1]    12  20     9
[2]    14  16     3
[3]    17  17     1
[4]    19  23     5
[5]    27  31     5
[6]    28  32     5
[7]    29  33     5
```

Inter-range methods act on the collection of ranges as a whole. These include `disjoin`, `reduce`, `gaps`, and `range`. An illustration is `reduce`, which reduces overlapping ranges into a single range, as illustrated in the lower panel of Figure 2.

```
> reduce(ir)

IRanges of length 2
    start end width
[1]     7  18    12
[2]    22  28     7
```

`coverage` is an inter-range operation that calculates how many ranges overlap individual positions. Rather than returning ranges, `coverage` returns a compressed representation (run-length encoding)

```
> coverage(ir)
```

```
'integer' Rle of length 28 with 12 runs
  Lengths: 6 2 4 1 2 3 3 1 1 3 1 1
  Values : 0 1 2 1 2 1 0 1 2 3 2 1
```

The run-length encoding can be interpreted as 'a run of length 6 of nucleotides covered by 0 ranges, followed by a run of length 2 of nucleotides covered by 1 range. . . '.

Between methods act on two (or sometimes more) *IRanges* instances. These include `intersect`, `setdiff`, `union`, `pintersect`, `psetdiff`, and `punion`.

The `countOverlaps` and `findOverlaps` functions operate on two sets of ranges. `countOverlaps` takes its first argument (the `query`) and determines how many of the ranges in the second argument (the `subject`) each overlaps. The result is an integer vector with one element for each member of `query`. `findOverlaps` performs a similar operation but returns a more general matrix-like structure that identifies each pair of query / subject overlaps. Both arguments allow some flexibility in the definition of 'overlap'.

Common operations on ranges are summarized in Table 5.

`elementMetadata` (`values`) and `metadata` 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*).

```
> 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 several 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

Table 5: 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, anchoring 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

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 *GRangesList* object has methods one would expect for lists (e.g., `length`, subsetting). Many of the methods introduced for working with *IRanges* are also available, with the method applied element-wise.

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 5
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 6
(Independent) Create a TranscriptDb instance from UCSC, using `makeTranscriptDbFromUCSC`*.*

Solution:

```
> txdb <- makeTranscriptDbFromUCSC("dm3", "ensGene")
> saveDb(txdb, "my.dm3.ensGene.txdb.sqlite")
```

3.2 Working with strings

Underlying the ranges of alignments and features are DNA sequences. The *Biostrings* package provides tools for working with this 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.

Exercise 7
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 SequenceAnalysis package defines a helper function `gcFunction` *(developed in a later exercise) 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]]))
```

Here is the helper function, available in the *SequenceAnalysis* package, to calculate GC content:

```
> gcFunction

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

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.

The subject GC content is

```
> subjectGC <- gcFunction(v)
```

3.3 Resources

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

Table 6: 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

4 Reads and Alignments

The following sections introduce core tools for working with high-throughput sequence data; key packages for representing reads and alignments are summarized in Table 6. This section focus on the reads and alignments that are the raw material for analysis. Section 5 addresses statistical approaches to assessing differential representation in RNA-seq experiments. Section 6 outlines ChIP-seq analysis. Section 7 introduces resources for annotating sequences.

4.1 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.

4.2 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
IIIIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
```

The first and third lines (beginning with `@` and `+` respectively) are unique identifiers. The identifier produced by the sequencer typically includes a machine id followed by colon-separated information on the lane, tile, x, and y coordinate of the read. The example illustrated here also includes the SRA accession number, added when the data was submitted to the archive. The machine identifier could potentially be used to extract information about batch effects. The spatial coordinates (lane, tile, x, y) are often used to identify optical duplicates; spatial coordinates can also be used during quality assessment to identify artifacts of sequencing, e.g., uneven amplification across the flow cell, though these spatial effects are rarely pursued.

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; this is developed further below. Both the sequence and quality scores may span multiple lines.

Technologies other than Illumina use different formats to represent sequences. Roche 454 sequence data is generated by 'flowing' labeled nucleotides over samples, with greater intensity corresponding to longer runs of A, C, G, or T. This data is represented as a series of 'flow grams' (a kind of run-length encoding of the read) in Standard Flowgram Format (SFF). The *Bioconductor* package *R453Plus1Toolbox* has facilities for parsing SFF files, but after quality control steps the data are frequently represented (with some loss of information) as FASTQ. SOLiD technologies produce sequence data using a 'color space' model. This data is not easily read in to *R*, and much of the error-correcting benefit of the color space model is lost when converted to FASTQ; SOLiD sequences are not well-handled by *Bioconductor* packages.

Short reads in *R* 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 *SequenceAnalysisData* 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+
```

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

```
  A BStringSet instance of length 3
    width seq
[1]    58 SRR031724.1 HWI-EAS299_4_30M2BAAXX:5:1:1513:1024 length=37
[2]    57 SRR031724.2 HWI-EAS299_4_30M2BAAXX:5:1:937:1157 length=37
[3]    58 SRR031724.4 HWI-EAS299_4_30M2BAAXX:5:1:1443:1122 length=37
```

The *ShortReadQ* class illustrates *class inheritance*. It extends the *ShortRead* class

```
> getClass("ShortReadQ")
```

```
Class "ShortReadQ" [package "ShortRead"]

Slots:

Name:       quality        sread           id
Class: QualityScore DNAStringSet   BStringSet

Extends:
Class "ShortRead", directly
Class ".ShortReadBase", by class "ShortRead", distance 2

Known Subclasses: "AlignedRead"
```

Methods defined on *ShortRead* are available for *ShortReadQ*.

```
> showMethods(class="ShortRead", where=getNamespace("ShortRead"))
```

For instance, the `width` can be used to demonstrate that all reads consist of 37 nucleotides.

```
> table(width(fq))
```

```
     37
1000000
```

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="SequenceAnalysis")
> browseURL(rpt)
```

Exercise 8
Use the helper function `bigdata` *(defined in the SequenceAnalysis 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.*

Use `alphabetFrequency` *to summarize the GC content of all reads (hint: use the* `sread` *accessor to extract the reads, and the* `collapse=TRUE` *argument to the* *`alphabetFrequency` function). Using the helper function `gcFunction` from the SequenceAnalysis 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:

```
> 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")
```

Exercise 9
Use `quality` to extract the quality scores of the short reads. Interpret the encoding qualitatively.

Convert the quality scores to a numeric matrix, using `as`. Inspect the numeric matrix (e.g., using `dim`) and understand what it represents.

Use `colMeans` to summarize the average quality score by cycle. Use `plot` to visualize this.

Solution:

```
> head(quality(fq))
```

```
class: FastqQuality
quality:
  A BStringSet instance of length 6
    width seq
[1]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
[2]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
[3]    37 IIIIIIIIIIIIIIIIIIIIIII'IIIIIGBIIII2I+
[4]    37 IIIIIIIIIIIIIIIIIIIIIIIII,II*E,&4HI++B
[5]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII&.$
[6]    37 III.IIIIIIIIIIIIIIIIIII%IIE(-EIH<IIII
```

```
> qual <- as(quality(fq), "matrix")
> dim(qual)
```

```
[1] 1000000      37
```

```
> plot(colMeans(qual), type="b")
```

4.3 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 [14, 13], Bowtie / Bowtie2 [12], 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"
```

Table 7: Fields in a SAM record. From http://samtools.sourceforge.net/samtools.shtml

Field	Name	Value
1	QNAME	Query (read) NAME
2	FLAG	Bitwise FLAG, e.g., strand of alignment
3	RNAME	Reference sequence NAME
4	POS	1-based leftmost POSition of sequence
5	MAPQ	MAPping Quality (Phred-scaled)
6	CIAGR	Extended CIGAR string
7	MRNM	Mate Reference sequence NaMe
8	MPOS	1-based Mate POSition
9	ISIZE	Inferred insert SIZE
10	SEQ	Query SEQuence on the reference strand
11	QUAL	Query QUALity
12+	OPT	OPTional fields, format TAG:VTYPE:VALUE

```
 [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"
```

Fields in a SAM file are summarized in Table 7. 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 argument, `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 10
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 5974 5402
   chrX  2283 2278
```

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])
+ }
```

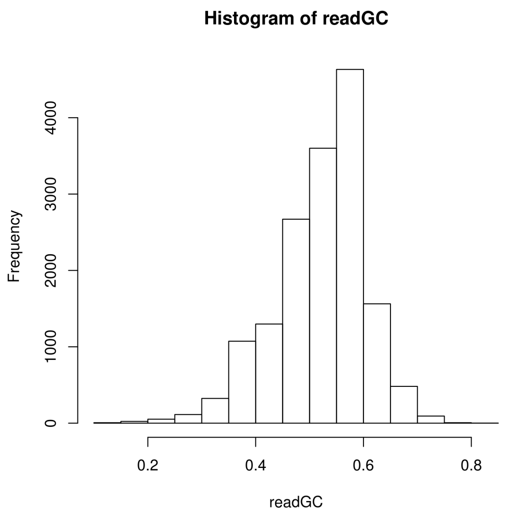


Figure 3: GC content in aligned reads

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 *GappedAlignments* class inputs only some of the fields of a BAM file, and may not be appropriate for all uses. In these cases the `scanBam` function in *Rsamtools* provides greater flexibility. The idea is to view BAM files as a kind of data base. Particular regions of interest can be selected, and the information in the selection restricted to particular fields. These operations are determined by the values of a *ScanBamParam* object, passed as the named `param` argument to `scanBam`.

Exercise 11
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 3).

Solution:

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

5 RNA-seq

5.1 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. The 'knownGenes' track of the UCSC genome browser represents one source of such data. A track like this describes, for each gene, the transcripts and exons that are expected based on current data. The *GenomicFeatures* package allows ready access to this information by creating a local database out of the track information. This data base of known genes is coupled with high throughput sequence data by counting reads overlapping known genes and modeling the relationship between treatment groups and counts.

A more ambitious approach to RNA-seq attempts to identify novel transcripts. This requires that sequenced reads be assembled into contigs that, presumably, correspond to expressed transcripts that are then located in the genome. Regions identified in this way may correspond to known transcripts, to novel arrangements of known exons (e.g., through alternative splicing), or to completely novel constructs. We will not address the identification of completely novel transcripts here, but will instead focus on the analysis of the designed experiments: do the transcript abundances, novel or otherwise, differ between experimental groups?

Bioconductor packages play a role in several stages of an RNA-seq analysis (Table 8; 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* [19] 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?

5.2 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 cap-

Table 8: 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`.
easyRNASeq	Workflow; also *ArrayExpressHTS*, *rnaSeqMap*, *oneChannelGUI*.
Rsubread	Alignment (Linux only); also *Biostrings* `matchPDict` for special-purpose alignments.

ture 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 12
The SequenceAnalysisData 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.

As a sanity check and to provide confidence that subsequent analysis is worthwhile, use `plotMDS` on the filtered reads to perform multi-dimensional scaling. Interpret the resulting plot.

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,]
```

Multi-dimensional scaling takes data in high dimensional space (in our case, the dimension is equal to the number of genes in the filtered *DGEList* instance) and reduces it to fewer (e.g., 2) dimensions, allowing easier visual assessment. The plot is shown in Figure 4; that the samples separate into distinct groups provides some reassurance that the data differ according to treatment. Nonetheless, there appears to be considerable heterogeneity within groups. Any guess, perhaps from looking at the quality report generated earlier, what the within-group differences are due to?

```
> plotMDS(dge)
```

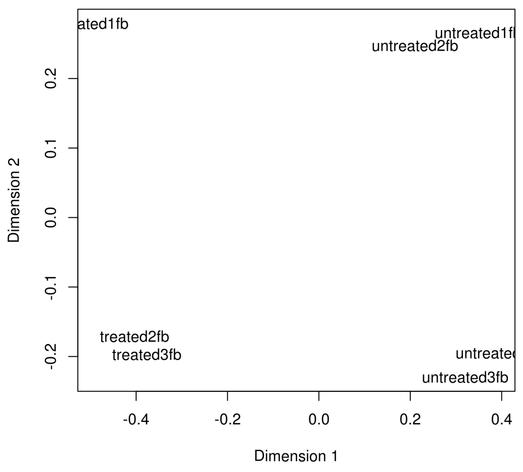


Figure 4: MDS plot of lanes from the Pasilla data set.

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 13
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] 0.18
```

This approach estimates a dispersion parameter that is specific to each tag. 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 14
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  -4.7    6.0 542 5.9e-120 4.7e-116
FBgn0039827  -4.3    4.6 247  1.0e-55  4.0e-52
FBgn0029167  -2.2    8.2 211  6.6e-48  1.8e-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         382
treated            64          31        1483          36
```

5.3 Additional steps in RNA-seq work flows

Annotation It is very easy to add annotations (mapping Entrez identifiers to gene names, KEGG or GO pathways, etc) to top tables; see Section 7.

Gene set enrichment Gene set enrichment approaches ask whether sets of genes are associated with a treatment group. Sets are typically defined to represent genes in particular biochemical or other pathways (e.g., from the KEGG or GO ontologies). Sets might be more generally defined, e.g., to represent chromosomal regions; the MSigDB (Molecular Signals data base) is one collection of diverse gene sets. Motivation for gene set analysis might be that pathways offer greater biological relevance or are more interpretabe than individual genes, or that the cumulative effect of several genes in a pathway may be statistically meaningful even though individual gene contributions are not.

Gene set enrichment was first developed for microarray data, where a variety of statistical approaches have been adopted; some of these are implemented in *Bioconductor*, e.g., under the 'Pathways' and 'GO' BiocViews terms, or in the *limma* package. One approach (implemented in the *GOstats* package) dichotomizes genes as 'significant' or not, and then uses a hypergeometric test (perhaps correcting for the hierarchical nature of some gene sets, e.g., in the GO classification) to assess whether significant genes are over-represented in each set. Another approach, developed in the *Category* package vignette, calculates the average of t-statistics of genes within a gene set and compares this with an appropriate null distribution.

Application of gene set enrichment approaches to studies of RNA-seq differential representation is conceptually similar to gene set enrichment in microarray analysis, but there are several important considerations [22]. Perhaps the most important is that long or highly expressed genes receive many hits, and hence are associated with greater statistical power. The *goseq* package uses a data base of known gene lengths to address this problem, as explored in the following exercise.

Exercise 15
Explore gene set analysis in the context of RNA-seq data through the goseq vignette. Can you think of alternative ways to address differences in statistical power associated with gene length or expression? Are there other nuances of RNA-seq data sets that should be taken into consideration?

In discussing gene set tests, Goeman and Buhlmann [6] distinguish between 'competitive' and 'self-contained' null hypotheses, and between 'gene' and 'subject' sampling to calculate P values.

A competitive test compares differential expression of a gene set by comparison to the complement of that gene set (e.g., hypergeommetric tests on $2x2$ tables of 'differentially expressed' versus 'not differentially expressed' genes and 'in gene set' versus 'not in gene set'). A self-contained test compares differential expression to a fixed standard independent of genes outside the gene set (e.g., sum of t statistics for genes in a set). Goeman and Buhlmann argue that self-contained tests have greater biological relevance and provide a natural extension of single-gene tests in a way that competitive tests do not.

Gene sampling assesses significance by permuting labels attached to genes (as in the hypergeommetric test), whereas subject sampling permutes subject labels and is much more consistent with standard statistical practice.

The *goseq* test is an example of a competitive test using gene sampling to assess significance. Alternative approaches for RNA-seq data are not well-developed. One possibility suggested by Smyth[1] is to use *limma*'s `voom` function to transform data to an approximate continuous scale, and use results in more familiar gene set analysis.

Exercise 16
As an advanced exercise, explore use of limma's `voom` transformation and the `roast` and `camera` gene set test.

Clustering and classification An obstacle to applying clustering and classification to sequence data is the choice of a distance metric appropriate for count data. The *edgeR* illustrates one approach. In the `plotMDS` method for objects of class *DGEList* uses tagwise dispersion estimates as a measure of biological variation. Tagwise dispersion across all samples is used to identify the most differentially expressed genes. The square root of the tagwise common dispersion of pairs of samples is then used to identify the most differentially expressed genes. The method is described on `?plotMDS.DGEList`; a similar distance metric could be used in other clustering and classification algorithms.

Specific applications may suggest more refined solutions. For instance, Holmes et al. [8] study microbial communities, where the raw data are read counts of taxonomic groups × biological samples. A useful approach is to fit a mixture model to the count data, where the mixture is of a finite number of Dirichlet probability vectors. This is illustrated in the *DirichletMultinomial* package.

Differential exon usage The RNA-seq analysis outlined here has focused on perhaps the most straight-forward research question – assessment of gene-level differential expression. Sequence data can deliver higher-resolution insight into gene expression. For instance, one might hope to gain understanding of transcript-level differential representation, or identify differential representation of novel transcripts. Novel transcript identification has received a great deal of attention, but poses significant challenges beyond the scope of this workshop. Approaches to transcript differential representation have often tried to combine estimation of transcript abundance with assessment of differential representation. The approach explored here is different and, in some ways, more straight-forward; it is based on the *DEXSeq* package.

DEXSeq starts with known gene models, rather than trying to quantify abundance of unknown transcripts. The gene models consist of exons grouped into transcripts, and transcripts grouped into genes; they can be retrieved from, e.g., the UCSC genome browser or ENSEMBL (via biomaRt). The gene models are simplified to represent disjoint (non-overlapping) exonic regions. Such regions may belong to one or several transcripts. Reads aligning to each region are counted, and the counts used in an analysis that is similar to the gene-level analysis of *edgeR* or *DESeq*. The analysis is modified, though, to incorporate the gene model. Specifically, one asks whether there is a significant interaction between treatment and exon use – do treatments differ in how exons are represented? An affirmative answer provides indirect evidence that, since a particular exon is also represented differently between treatments, the transcript to which the exon belongs is represented differently. The approach uses the same

[1]`https://stat.ethz.ch/pipermail/bioconductor/2012-April/044899.html`

statistical machinery as the *edgeR* or *DESeq* packages, so makes efficient use of available data with appropriate assumptions about the error model.

Exercise 17
Explore exon usage through the DEXSeq vignette. Compare the merits and challenges of this approach with, e.g., direct estimation of transcript abundance and differential representation. How straight-forward is it to interpret results of a DEXSeq analysis, in terms of differential transcript use? Under what experimental circumstances might this approach be most profitably employed? Are there any avenues for simplifying the analysis, e.g., in simplifying known gene models to capture just the splicing events differentiating transcripts (a tough question; see [21]).

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') occurring.

5.4 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 9: 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*.

6 ChIP-seq

6.1 Varieties of 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 [17, 7]; there is a large collection of peak-calling software, some features of which are summarized in Pepke et al. [18].

Initial stages in a ChIP-seq analysis differ from RNA-seq in several important ways. The ChIP protocol is more complicated and idiosyncratic than RNA-seq protocols, and the targets of ChIP more variable in terms of sequence and other characteristics. While RNA- and ChIP-seq use reads aligned to a reference genome, ChIP-seq protocols require that the aligned reads be processed to identify peaks, rather than simply counted in known gene regions.

Many early ChIP-seq studies focused on characterizing one or a suite of transcription factor binding sites across a small number of samples from one or two groups. The main challenge 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 [10, 16]. 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 9; a more comprehensive list is under the ChIPseq and HighThroughputSequencing BiocViews terms). The *ShortRead* package can provide a quality assessment report of reads. Following alignment, the *chipseq* package can be used, in conjunction with *ShortRead* and *GenomicRanges*, to identify enriched regions in a statistically informed and flexible way. *DiffBind* provides facilities for comprehensive analysis of experiments with multiple ChIP-seq samples. The *ChIPpeakAnno* package assists in annotating peaks in terms of known genes and other genomic features. Pattern matching in *Biostrings* and specialized packages such as *MotIV* can assist in motif identification.

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. An 'advanced' section then illustrates how exploratory or novel ChIP-seq algorithms can be implemented in *Bioconductor*. The chapter concludes with more downstream analysis, including motif discovery and annotation.

6.2 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.

Quality assessment

Exercise 18
As a precursor to analysis, it is prudent to perform an overall quality assessment of the data; The SequenceAnalysisData package contains a quality assessment report generated from the BAM files. View this report. Are there indications of batch or other systematic effects in the data?

Solution: Here we visit the QA report.

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

Samples 1-15 correspond to replicate 1, 16-26 to replicate 2, and 27 through 41 the 'input' samples. Notice that overall nucleotide frequencies fall into three distinct groups, and that samples 1-11 differ from the other input samples. The 'Depth of Coverage' portion of the report is particularly relevant for an early assessment of ChIP-seq experiments.

Alignment and peak calling (3rd party) The main computational stages in the original work flow involved alignment using Bowtie, followed by peak identification using an algorithm ('HotSpots', [20]) 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="SequenceAnalysisData")
> 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
```

Data exploration

Exercise 19
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 
```

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 5.

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

6.2.1 Peak calling with *R* / *Bioconductor* (advanced)

The following illustrates basic ChIP-seq in *R*. It is likely that these would be used either in an exploratory way, or as foundations for developing work flows tailored to particular ChIP experiments. We work through this section by developing functions for different parts of the work flow. Functions are applied to examples in an exercise at the end of this section.

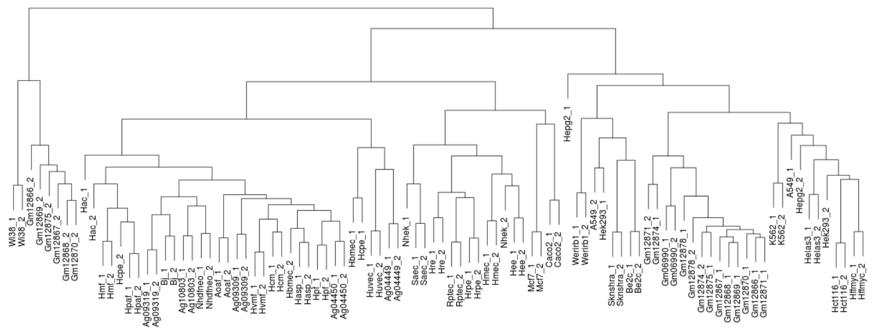


Figure 5: Hierarchical clustering of ENCODE samples.

Data input and pre-processing Here we develop our own function, named `chipPreprocess`, to pre-process a single sample lane. The work flow starts with data input. We suppose an available `bamFile`, with a *ScanBamParam* object `param` defined to select regions we are interested in; thee `bamFile` and `param` objects are defined later.

```
> aln <- readGappedAlignments(bamFile, param=param)
> seqlevels(aln) <-  names(bamWhich(param))
> aln <- as(aln, "GRanges")
```

We use `readGappedAlignments` followed by coercion to a *GRanges* object as a convenient way to retrieve a minimal amount of data from the BAM file, and to manage reads whose alignments include indels; `Rsamtools::scanBam` is a more flexible alternative. The `seqlevels` are adjusted to contain just the levels we are interested in, rather than all levels in the BAM file (the default returned by `readGappedAlignments`).

Sequence work flows typically filter reads to remove those that are optical duplicates or otherwise flagged as invalid by the manufacturer. Many work flows do not handle reads aligning to multiple locations in the genome. ChIP-seq experiments often eliminate reads that are duplicated in the sense that more than one read aligns to the same chromosome, strand, and start position; this acknowledges artifacts of sample preparation. These filters are handled by different stages in a typical work flow – flagging optical duplicates and otherwise suspect reads by the manufacturer or upstream software (illustrated in an exercise, below); discarding multiply aligning reads by the aligner (in our case, using the `-m` and `-n` options in *Bowtie*); and discarding duplicates as a pre-processing step. Simple alignment de-duplication is

```
> aln <- aln[!duplicated(aln)]
```

It is common to estimate fragment length (e.g., via the 'correlation' method [11], implemented in the *chipseq* package) and extend the 5' tags by the estimated length.

```
> fraglen <- estimate.mean.fraglen(aln, method="correlation")
> aln <- resize(aln, width=fraglen)
```

The end result can be summarized as a 'coverage vector' describing the number of (extended) reads at each location in the genome; a run length encoding is an efficient representation of this.

```
> coverage(aln)
```

These pre-processing steps can be summarized as a simple work-flow.

```
> chipPreprocess <- function(bamFile, param) {
+     aln <- readGappedAlignments(bamFile, param = param)
+     seqlevels(aln) <- names(bamWhich(param))
+     aln <- as(aln, "GRanges")
+     aln <- aln[!duplicated(aln)]
+     fraglen <- estimate.mean.fraglen(aln, method = "correlation")
+     aln <- resize(aln, width = fraglen)
+     coverage(aln)
+ }
```

Peak identification We now develop a function, `findPeaks`, to identify peaks. The coverage vector is a very useful representation of the data, and numerous peak discovery algorithms can be implemented on top of it. The *chipseq* package implements a straight-forward approach. The first step uses the distribution of singleton and doubleton islands to estimate a background Poisson noise distribution, and hence to identify a threshold island elevation above which peaks can be called at a specified false discovery rate.

```
> cutoff <- round(peakCutoff(cvg, fdr.cutoff=0.001))
```

Peaks are easily identified using `slice`

```
> slice(cvg, lower = cutoff)
```

resulting in a peak-finding work flow

```
> findPeaks <- function(cvg) {
+     cutoff <- round(peakCutoff(cvg, fdr.cutoff = 0.001))
+     slice(cvg, lower = cutoff)
+ }
```

Exercise 20
Walk through the work flow, from BAM file to called peaks, using the provided BAM files. These are from the Ag09319 cell line, CTCF replicate 1 and input lanes, filtered to include only reads from chromosome 6. Compare peaks found in the ChIP and Input lanes, and in the MACS analysis. It is possible to pick up the analysis after pre-processing by loading the `cvgs` *object. It can be very helpful to explore the data along the way; see the chipseq vignette for ideas.*

Solution: Specify the location of the BAM files, and the location where the coverage vectors will be saved.

```
> bamDir <- character()        # TODO: read BAM file from...
> cvgsSaveFile <- character()  # TODO: save coverage file to...
```

Storing the coverage vectors represents a check-pointing strategy, making it easy to resume an analysis if interrupted.

```
> library(GenomicRanges)
> bamFiles <- c(ChIP=file.path(bamDir,
+                  "wgEncodeUwTfbsAg09319CtcfStdAlnRep1.bam"),
+                Input=file.path(bamDir,
+                  "wgEncodeUwTfbsAg09319InputStdAlnRep1.bam"))
> stopifnot(all(file.exists(bamFiles)))
```

Create a `ScanBamParam` object specifying the regions of interest and other restrictions on reads to be input.

```
> chr6len <- scanBamHeader(bamFiles)[[1]][["targets"]][["chr6"]]
> param <- ScanBamParam(which=GRanges("chr6", IRanges(1, chr6len)),
+                        what=character(),
+                        flag=scanBamFlag(isDuplicate=FALSE,
+                          isValidVendorRead=TRUE))
```

Process each BAM file using `lapply`, and save the result.

```
> cvgs <- lapply(bamFiles, chipPreprocess, param)
> save(cvgs, cvgsSaveFile)
```

Load the saved coverage file, and find peaks using the simple approach outlined above.

```
> library(chipseq)
> cvgsFile <- system.file("data", "chipseq_chr6_cvgs.Rda",
+                          package="SequenceAnalysisData")
> stopifnot(file.exists(cvgsFile))
> load(cvgsFile)                             # previously saved
> peaks <- lapply(cvgs, findPeaks)
```

Compare the peaks using *GRanges* commands (e.g. convert the peaks to `IRanges` instances and use `countOverlaps` to identify peaks in common between the ChIP and Input lanes), and the `diffPeakSummary` function from the *chipseq* package. Compare the peaks to those found in the `stam` object.

```
> chip <- as(peaks[["ChIP"]][["chr6"]], "IRanges")
> inpt <- as(peaks[["Input"]][["chr6"]], "IRanges")
> table(countOverlaps(inpt, chip))

  0   1   2
635  19   3

> table(countOverlaps(chip, inpt))

   0    1    2
5534   23    1

> stamFile <- system.file("data", "stam.Rda",
+                  package="SequenceAnalysisData")
> load(stamFile)
> stam0 <- stam[,"Ag09319_1"]
> idx <- seqnames(rowData(stam0)) == "chr6" &
+            assays(stam0)[["Tags"]] != 0
> rng <- ranges(rowData(stam0))[as.logical(idx)]
> table(countOverlaps(chip, rng))
```

```
   0    1    2    3
 811 4689   56    2
```

Our naive analysis finds many of the peaks identified by a more comprehensive algorithm.

6.3 Comparison of multiple experiments: *DiffBind*

Exercise 21
Explore a ChIP-seq work flow through the DiffBind vignette.

6.4 Working with called peaks

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.

The JASPAR data base curates known binding motifs obtained from the literature. A binding motif is summarized as a *position weight matrix* PWM or *position frequency matrix* PFM. 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. A PFM is a similar representation, but with entries corresponding to a count of the times a nucleotide is observed.

Exercise 22
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.*

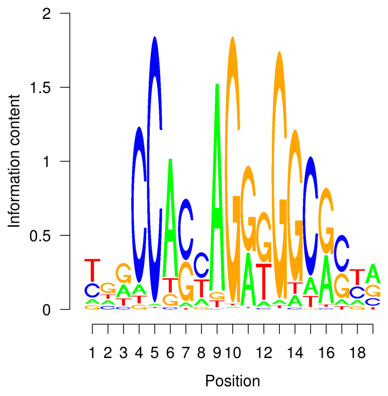
Retrieve the PWM for CTCF, with JASPAR id MA0139.1.pfm, using the helper function `getJASPAR` *defined in the SequenceAnalysis package. Visualize the PWM using* `seqLogo`*.*

Solution: Load the packages:

```
> library(Biostrings)
> library(BSgenome.Hsapiens.UCSC.hg19)
> library(seqLogo)
> library(lattice)
```

Retrieve the position weight matrix for CTCF, and display the PWM:

```
> pwm <- getJASPAR("MA0139.1.pfm") # SequenceAnalysis::getJASPAR
> seqLogo(scale(pwm, FALSE, colSums(pwm)))
```



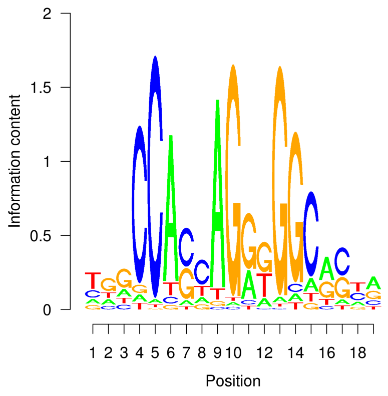


Figure 6: CTCF position weight matrix from JASPAR (left) and on the plus strand of chr1 (hits within 80% of maximum score, right).

Exercise 23
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: 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 matrix; this can be plotted as a logo, with the result in Figure 6. Reassuringly, the found sequences have a logo very similar to the expected.

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

Exercise 24
We might expect that peaks found using de novo *techniques like MACS would be enriched for motifs identified for the known PWM. What fraction of our high-scoring positions are in the peaks in the `stam` object? What technical and biological issues might cloud this result?*

Solution:

```
> roi <- GRanges(chrid, ranges(hits), "+")
> seqinfo(roi) <- seqinfo(Hsapiens)
> sum(roi %in% rowData(stam)) / length(roi)

[1] 0.55
```

Annotation For an introduction to annotation resources in *Bioconductor*, see Section 7; the *ChIPpeakAnno* contributed package provides convenient ways to annotate ChIP-seq experiments.

Exercise 25
Annotating ChIP peaks is straight-forward. 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="SequenceAnalysisData")
> 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

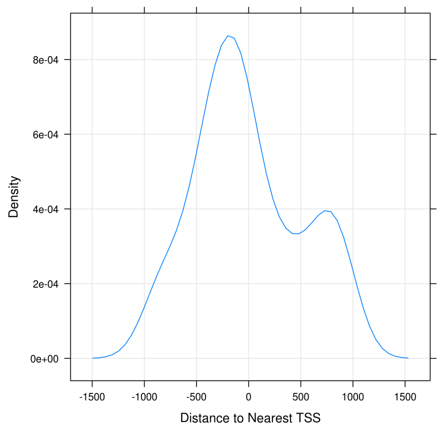


Figure 7: Distance to nearest TSS amongst conserved peaks

```
> 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 7.

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

```
 -1   1
669 417
```

```
> print(densityplot(dist[ok], type="g", 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 26
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.

```
> 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, type="g",
+     auto.key=list(
+       columns=2,
+       title="Has Position Weight Matrix?",
+       cex.title=1),
+     xlab="Distance to Nearest Tss"))
```

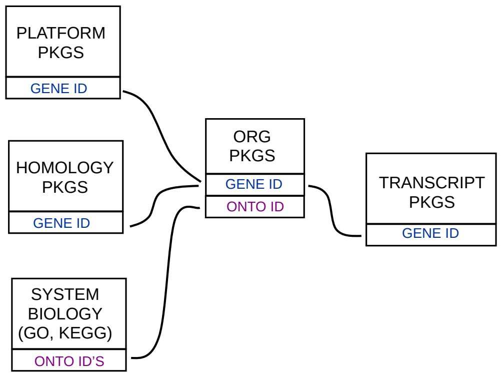


Figure 8: Annotation Packages: the big picture

7 Annotation

Bioconductor provides extensive annotation resources, summarized in Figure 8. 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.

7.1 Gene-centric annotations with *AnnotationDbi*

Organism-level ('org') packages contain mappings between a central identifier (e.g., Enterz gene ids) and other identifiers (e.g. GenBank or Uniprot accession number, RefSeq id, etc.). The name of an org 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.

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`.

Exercise 27
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`.

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

Exercise 28
For convenience, `lrTest`, a DGEGLM object from the RNA-seq chapter, is included in the SequenceAnalysisData package. The following code loads this data and create a 'top table' of the ten most differentially represented genes. This top table is then cast as a `data.frame`.

```
> 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:

```
> 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>
```

7.2 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 5) and retrieving DNA sequences underlying regions of interest in ChIP-seq analysis (Section 6), e.g., for motif characterization.

Exercise 29
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 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 a 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 elementMetadata cols:
      seqnames               ranges strand |    cds_id    cds_name exon_rank
         <Rle>            <IRanges>  <Rle> | <integer> <character> <integer>
  [1]    chr3R [19970946, 19971592]      + |     55167        <NA>         2
  [2]    chr3R [19971652, 19971770]      + |     55168        <NA>         3
  [3]    chr3R [19971831, 19972024]      + |     55169        <NA>         4
  [4]    chr3R [19972088, 19972461]      + |     55170        <NA>         5
  [5]    chr3R [19972523, 19972589]      + |     55171        <NA>         6
  [6]    chr3R [19972918, 19973094]      + |     55172        <NA>         7

---
seqlengths:
     chr2L  chr2LHet     chr2R  chr2RHet ...   chrXHet   chrYHet      chrM
  23011544    368872  21146708   3288761 ...    204112    347038     19517
```

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 correctly.

```
> 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*
```

7.3 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 30
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.

8 Annotation of Variants

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.

8.1 Variant call format (VCF) files

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.

Data exploration

Exercise 31
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:

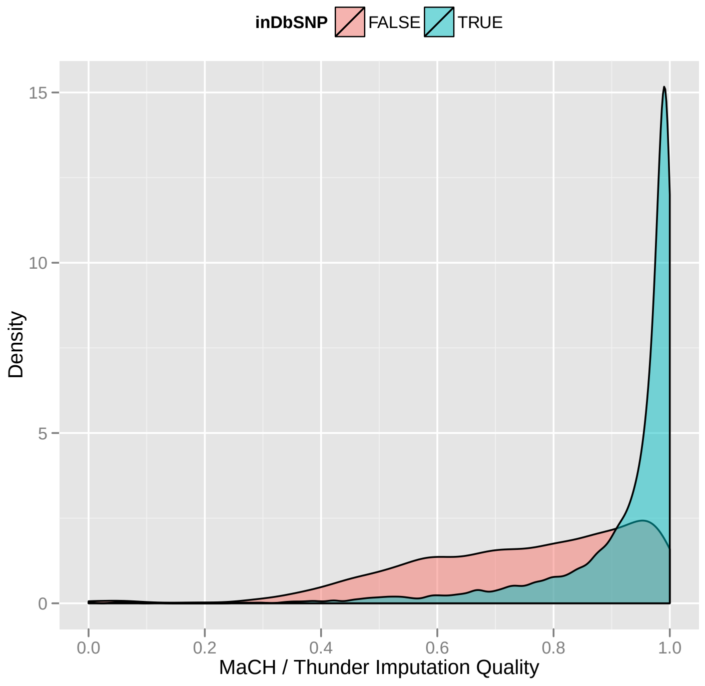


Figure 9: Quality scores of variants in dbSNP, compared to those not in dbSNP.

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

Finally, visualize the data, e.g., using *ggplot2* (Figure 9).

```
> 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")
```

8.2 Coding consequences

Locating 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 10.

Exercise 32
Load the TxDb.Hsapiens.UCSC.hg19.knownGene annotation package, and read in the `chr22.vcf.gz` example file from the VariantAnnotation package.

Table 10: 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

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")
> vcf <- renameSeqlevels(vcf, 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 5 elementMetadata cols:
      seqnames               ranges strand | LOCATION   QUERYID      TXID
         <Rle>            <IRanges>  <Rle> | <factor> <integer> <integer>
  [1]    chr22 [50301422, 50301422]      * |   coding        24     76833
  [2]    chr22 [50301476, 50301476]      * |   coding        25     76833
  [3]    chr22 [50301488, 50301488]      * |   coding        26     76833
          CDSID      GENEID
      <integer> <character>
  [1]    225251       79087
  [2]    225251       79087
  [3]    225251       79087
  ---
  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(values(loc)$GENEID, values(loc)$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(values(loc)$QUERYID, values(loc)$GENEID)
> head(sapply(splt, function(x) length(unique(x))), 3)

113730   1890  23209
    22     15     30
```

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`, the alternate alleles are taken from `values(alt(<VCF>))$ALT` and the `varAllele` argument is not specified.

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

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

GRanges with 5 ranges and 13 elementMetadata cols:
             seqnames               ranges strand | paramRangeID
                <Rle>            <IRanges>  <Rle> |     <factor>
  22:50301584    chr22 [50301584, 50301584]      - |         <NA>
                  varAllele     CDSLOC              PROTEINLOC   QUERYID
             <DNAStringSet>  <IRanges> <CompressedIntegerList> <integer>
  22:50301584             A [777, 777]                     259        28
                   TXID     CDSID      GENEID  CONSEQUENCE       REFCODON
             <character> <integer> <character>     <factor> <DNAStringSet>
  22:50301584       76833    225251       79087   synonymous            CCG
                  VARCODON         REFAA         VARAA
             <DNAStringSet> <AAStringSet> <AAStringSet>
  22:50301584           CCA             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[values(coding)$CONSEQUENCE == "frameshift"]

GRanges with 1 range and 13 elementMetadata cols:
              seqnames                 ranges strand | paramRangeID
                 <Rle>              <IRanges>  <Rle> |     <factor>
  22:50317001    chr22 [50317001, 50317001]      + |         <NA>
                   varAllele     CDSLOC             PROTEINLOC   QUERYID
              <DNAStringSet>  <IRanges> <CompressedIntegerList> <integer>
  22:50317001          GCACT [808, 808]                     270       359
                    TXID     CDSID      GENEID CONSEQUENCE       REFCODON
              <character> <integer> <character>    <factor> <DNAStringSet>
  22:50317001       76834    225263       79174  frameshift            GCC
                    VARCODON          REFAA          VARAA
              <DNAStringSet> <AAStringSet> <AAStringSet>
  22:50317001            ACC             A
  ---
  seqlengths:
   chr22
      NA
```

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

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

Identify the non-synonymous variants and obtain the rsids.

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

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

```
> library(SIFT.Hsapiens.dbSNP132)
> ## rsids in the package
> head(keys(SIFT.Hsapiens.dbSNP132), 3)

[1] "rs10000692" "rs10001580" "rs10002700"

> ## 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.

A Appendix: data retrieval

A.1 RNA-seq data retrieval

The following script was used to retrieve a portion of the Pasilla data set from the short read archive. The data is very large; extraction relies on installation of the SRA SDK, available from the Short Read Archive.

```
> library(RCurl)
> srasdk <- "/home/mtmorgan/bin/sra_sdk-2.0.1" # local installation
> sra <- "ftp://ftp-trace.ncbi.nih.gov/sra/sra-instant/reads/ByExp/sra"
> expt <- "SRX/SRX014/SRX014458/"
> url <- sprintf("%s/%s", sra, expt)
> acc <- strsplit(getURL(url, ftplistonly=TRUE), "\n")[[1]]
> urls <- sprintf("%s%s/%s.sra", url, acc, acc)
> for (fl in urls)
+     system(sprintf("wget %s",  fl), wait=FALSE, ignore.stdout=TRUE)
> app <- sprintf("%s/bin64/fastq-dump", srasdk)
> for (fl in file.path(wd, basename(urls)))
+      system(sprintf("%s %s", app, fl), wait=FALSE)
```

A.2 ChIP-seq data retrieval and MACS analysis

BAM and called peak files were obtained from http://hgdownload.cse.ucsc.edu/goldenPath/hg19/encodeDCC/wgEncodeUwTfbs. The script used to process called peak data into the `stam` object is at

```
> file.path("script", "chipseq-stam-called-peaks.R",
+           package="SequenceAnalysisData")

[1] "script/chipseq-stam-called-peaks.R/SequenceAnalysisData"
```

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. J. Goeman and P. Buhlmann. Analyzing gene expression data in terms of gene sets: methodological issues. *Bioinformatics*, 23(8):980–987, Apr 2007.

[7] 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].

[8] I. Holmes, K. Harris, and C. Quince. Dirichlet multinomial mixtures: Generative models for microbial metagenomics. *PLoS ONE*, 7(2):e30126, 02 2012.

[9] R. Kabacoff. *R in Action*. Manning, 2010.

[10] 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].

[11] P. V. Kharchenko, M. Y. Tolstorukov, and P. J. Park. Design and analysis of ChIP-seq experiments for DNA-binding proteins. *Nat. Biotechnol.*, 26:1351–1359, Dec 2008. [PubMed Central:PMC2597701] [DOI:10.1038/nbt.1508] [PubMed:19029915].

[12] 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.

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

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

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

[16] 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].

[17] 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].

[18] 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].

[19] 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.

[20] 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.

[21] M. Sammeth. Complete alternative splicing events are bubbles in splicing graphs. *J. Comput. Biol.*, 16:1117–1140, Aug 2009.

[22] M. D. Young, M. J. Wakefield, G. K. Smyth, and A. Oshlack. Gene ontology analysis for rna-seq: accounting for selection bias. *Genome Biology*, 11:R14, 2010.