

# *R / Bioconductor* for High-Throughput Sequence Analysis

Martin Morgan<sup>1</sup> Nicolas Delhomme<sup>2</sup>

29-30 October, 2012

<sup>1</sup>[mtmorgan@fhcrc.org](mailto:mtmorgan@fhcrc.org)

<sup>2</sup>[nicolas.delhomme@plantphys.umu.se](mailto:nicolas.delhomme@plantphys.umu.se)

# Contents

<b>1</b>	<b>Introduction to <i>R</i> / <i>Bioconductor</i></b>	<b>2</b>
1.1	Introduction . . . . .	2
1.1.1	This workshop . . . . .	2
1.1.2	<i>Bioconductor</i> . . . . .	2
1.1.3	High-throughput sequence analysis . . . . .	3
1.1.4	Statistical programming . . . . .	3
1.1.5	<i>Bioconductor</i> for high-throughput sequence analysis . . . . .	5
1.2	<i>R</i> . . . . .	5
1.2.1	<i>R</i> data types . . . . .	6
1.2.2	Useful functions . . . . .	10
1.2.3	Packages . . . . .	14
1.2.4	Help . . . . .	15
1.2.5	Efficient scripts . . . . .	17
1.2.6	Warnings, errors, and debugging . . . . .	20
1.2.7	Resources . . . . .	20
<b>2</b>	<b>Sequences and Short Reads</b>	<b>21</b>
2.1	Ranges and Strings . . . . .	21
2.1.1	Genomic ranges . . . . .	21
2.1.2	Working with strings . . . . .	27
2.2	Reads and Alignments . . . . .	28
2.2.1	The <i>pasilla</i> data set . . . . .	28
2.2.2	Reads and the <i>ShortRead</i> package . . . . .	28
2.2.3	Alignments and the <i>Rsamtools</i> package . . . . .	32
2.2.4	Alignments and other <i>Bioconductor</i> packages . . . . .	38
2.2.5	Resources . . . . .	42
<b>3</b>	<b>Annotation of Genes and Genomes</b>	<b>43</b>
3.1	Annotation . . . . .	43
3.1.1	Gene-centric annotations with <i>AnnotationDbi</i> . . . . .	43
3.1.2	Genome-centric annotations with <i>GenomicFeatures</i> . . . . .	45
3.1.3	Using biomaRt . . . . .	47
<b>4</b>	<b>Estimating Expression over Genes and Exons</b>	<b>49</b>
4.1	Counting reads over known genes and exons . . . . .	49
4.1.1	The alignments . . . . .	49
4.1.2	The annotation . . . . .	50
4.1.3	Discovering novel transcribed regions . . . . .	53
4.2	Using easyRNASeq . . . . .	55
<b>5</b>	<b>Working with Called Variants</b>	<b>58</b>
5.1	Annotation of Variants . . . . .	58
5.1.1	Variant call format (VCF) files . . . . .	58
5.1.2	Coding consequences . . . . .	60

# Chapter 1

## Introduction to *R* / *Bioconductor*

### 1.1 Introduction

#### 1.1.1 This workshop

This portion of the workshop introduces use of *R* [35] and *Bioconductor* [11] for analysis of high-throughput sequence data. The workshop is structured as a series of short remarks followed by group exercises. The exercises explore the diversity of tasks for which *R* / *Bioconductor* are appropriate, but are far from comprehensive.

The goals of the workshop are to: (1) develop familiarity with *R* / *Bioconductor* software for high-throughput analysis; (2) expose key statistical issues in the analysis of sequence data; and (3) provide inspiration and a framework for further independent exploration. An approximate schedule is shown in Table 1.1.

#### 1.1.2 *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-processing and analysis of designed experiments, and integrative and reproducible approaches to bioinformatic tasks. There are now more than 600 *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](http://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.

Table 1.1: Tentative schedule.

---

<i>R</i> / <i>Bioconductor</i> Introduction and Short Reads
<i>R</i> data types & functions; help; objects; essential packages, efficient programming.
Working with strings, reads and ranges.
Annotation of Genes and Variants
Common work flows; variants in and around genes, amino acid and coding consequences.

---

- [Package developer](#) resources, including guidelines for creating and submitting new packages.

### Exercise 1

*Scavenger hunt. Spend five minutes tracking down the following information.*

- From the Bioconductor web site, instructions for installing or updating Bioconductor packages.*
- A list of all packages in the current release of Bioconductor.*
- 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), <http://bioconductor.org/help/mailling-list/> (mailing lists).

## 1.1.3 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 (tens of millions per sample) of short (e.g., 35-150, 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.

## 1.1.4 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 analysis 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 L<sup>A</sup>T<sub>E</sub>X document with embedded *R* commands. The *R* commands are evaluated when the document is built, thus reproducing the analysis. The use of L<sup>A</sup>T<sub>E</sub>X 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

Table 1.2: Selected *Bioconductor* packages for high-throughput sequence analysis.

Concept	Packages
Data representation	<i>IRanges</i> , <i>GenomicRanges</i> , <i>GenomicFeatures</i> , <i>Biostrings</i> , <i>BSgenome</i> , <i>girafe</i> [41].
Input / output	<i>ShortRead</i> [31] (fastq), <i>Rsamtools</i> (bam), <i>rtracklayer</i> (gff, wig, bed), <i>VariantAnnotation</i> (vcf), <i>R453Plus1Toolbox</i> [21] (454).
Annotation	<i>GenomicFeatures</i> , <i>ChIPpeakAnno</i> , <i>VariantAnnotation</i> .
Alignment	<i>Rsubread</i> , <i>Biostrings</i> .
Visualization	<i>ggbio</i> [44], <i>Gviz</i> .
Quality assessment	<i>qRQC</i> , <i>seqbias</i> [18], <i>ReQON</i> , <i>htSeqTools</i> , <i>TEQC</i> [29], <i>Rolexa</i> , <i>Short-Read</i> .
RNA-seq	<i>BitSeq</i> [12], <i>cqn</i> [16], <i>cummeRbund</i> , <i>DESeq</i> [1], <i>DEXSeq</i> [2], <i>EDASeq</i> [36], <i>edgeR</i> [37], <i>gage</i> [28] <i>goseq</i> [45], <i>iASeq</i> , <i>tweeDEseq</i> .
ChIP-seq, etc.	<i>BayesPeak</i> [5], <i>baySeq</i> , <i>ChIPpeakAnno</i> , <i>chipseq</i> , <i>ChIPseqR</i> , <i>ChIPsim</i> , <i>CSAR</i> ,[33] <i>DiffBind</i> [38], <i>MEDIPS</i> , <i>mosaics</i> , <i>NarrowPeaks</i> , <i>nucleR</i> [9], <i>PICS</i> [46], <i>PING</i> , <i>REDseq</i> , <i>Repitools</i> , <i>TSSi</i> .
Motifs	<i>BCRANK</i> , <i>cosmo</i> , <i>cosmoGUI</i> , <i>MotIV</i> , <i>seqLogo</i> , <i>rGADEM</i> .
3C, etc.	<i>HiTC</i> [40], <i>r3Cseq</i> .
Copy number	<i>cn.mops</i> [20], <i>CNAnorm</i> [14], <i>exomeCopy</i> , <i>segmentSeq</i> .
Microbiome	<i>phyloseq</i> ,[34] <i>DirichletMultinomial</i> [17], <i>clstutils</i> , <i>manta</i> , <i>mcaGUI</i> .
Work flows	<i>ArrayExpressHTS</i> , <i>Genominator</i> [4], <i>easyRNASeq</i> [8], <i>oneChannel-GUI</i> , <i>rnaSeqMap</i> [24].
Database	<i>SRADB</i> .

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.

### 1.1.5 *Bioconductor* for high-throughput sequence analysis

Table 1.2 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*).

## 1.2 *R*

*R* is an open-source statistical programming language. It is used to manipulate data, to perform statistical analysis, 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 analysis, 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 analysis functions are very well-developed.

### 1.2.1 *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. Sub-setting 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
```

Index values can be repeated, and if outside the domain of `x` return the special value `NA`. Negative index values remove elements from the vector. Logical and character vectors (described below) can also be used for sub-setting.

*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] 1.61 1.39 1.10 0.69 0.00
```

**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, sub-setting 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 homogeneous 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 sub-setting, *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 Homogeneous, 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
      0.006         1.004

> anova(fit)

Analysis of Variance Table

Response: y
      Df Sum Sq Mean Sq F value Pr(>F)
x         1    1004      1004    4104 <2e-16 ***
Residuals 998      244         0
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

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

[1] 0.49

> 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] 1.61 1.39 1.10 0.69 0.00

> 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] 2.3 2.0 1.6 1.0 0.0

> 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] 2.3 2.0 1.6 1.0 0.0
```

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.

## 1.2.2 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 Homogeneous 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`, `chisq.test` 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`);  $\chi^2$  tests.

`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`, `aggregate` Apply a function to each element of a list (`lapply`, `sapply`), to elements of several lists (`mapply`), or to elements of a list partitioned by one or more factors (`aggregate`).

`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 evaluation 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="EMBO2012", "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 sub-setting 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 sub-setting, 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		remission	
10005	: 1	1/15/1997	: 2	F	:42	Min.	: 5	B2	:36	CR	:99
1003	: 1	1/29/1997	: 2	M	:83	1st Qu.:	:19	B3	:23	REF	:15
1005	: 1	11/15/1997:	2	NA's:	3	Median	:29	B1	:19	NA's:	14
1007	: 1	2/10/1998	: 2			Mean	:32	T2	:15		
1010	: 1	2/10/2000	: 2			3rd Qu.:	:46	B4	:12		
11002	: 1	(Other)	:116			Max.	:58	T3	:10		
(Other):	122	NA's	: 2			NA's	:5	(Other):	13		
		CR				t.4.11.		t.9.22.			
CR		:96	11/11/1997:	3		Mode	:logical	Mode	:logical		
DEATH IN CR		: 3	1/21/1998	: 2		FALSE:	86	FALSE:	67		
DEATH IN INDUCTION:	7	10/18/1999:	2			TRUE	:7	TRUE	:26		
REF		:15	12/7/1998	: 2		NA's	:35	NA's	:35		

```

NA's          : 7   1/17/1997 : 1
                (Other) :87
                NA's     :31

cyto.normal    citog      mol.biol    fusion.protein  mdr
Mode :logical   normal    :24   ALL1/AF4:10   p190      :17   NEG :101
FALSE:69        simple alt. :15   BCR/ABL :37   p190/p210: 8   POS : 24
TRUE :24         t(9;22)    :12   E2A/PBX1: 5   p210      : 8   NA's: 3
NA's :35         t(9;22)+other:11   NEG      :74   NA's      :95
                complex alt. :10   NUP-98   : 1
                (Other)     :21   p15/p16  : 1
                NA's        :35

    kinet      ccr      relapse      transplant
dyploid:94    Mode :logical Mode :logical Mode :logical
hyperd.:27    FALSE:74     FALSE:35   FALSE:91
NA's : 7      TRUE :26     TRUE :65   TRUE :9
                NA's :28     NA's :28   NA's :28

```

```

                f.u      date.last.seen
REL             :61   1/7/1998 : 2
CCR             :23   12/15/1997: 2
BMT / DEATH IN CR: 4   12/31/2002: 2
BMT / CCR       : 3   3/29/2001 : 2
DEATH IN CR     : 2   7/11/1997 : 2
(Other)         : 7   (Other)   :83
NA's            :28   NA's      :35

```

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 = 4.8, df = 69, p-value = 8.401e-06
```

```
alternative hypothesis: true difference in means is not equal to 0
```

```
95 percent confidence interval:
```

```
 7.1 17.2
```

```
sample estimates:
```

```
mean in group BCR/ABL      mean in group NEG
              40              28
```

Table 1.3: Selected base and contributed packages.

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

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.

### 1.2.3 Packages

Packages provide functionality beyond that available in base *R*. There are over 4000 packages in CRAN (comprehensive *R* archive network) and more than 600 *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 1.3 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 1.1 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)
> plt <- dotplot(variety ~ yield | site, data = barley, groups = year,
+               xlab = "Barley Yield (bushels/acre)" , ylab=NULL,
+               key = simpleKey(levels(barley$year), space = "top",
+               columns=2),
+               aspect=0.5, layout = c(2,3))
> print(plt)
```

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] 10
```

```
> search()
```

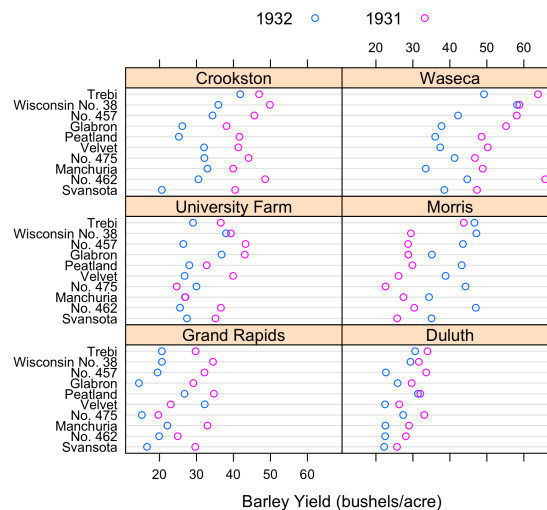


Figure 1.1: Variety yield conditional on site and grouped by year, for the `barley` data set.

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

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

```
[1] 0.00 0.69 1.10
```

### Exercise 3

Use the `library` function to load the `EMBO2012` package. Use the `sessionInfo` function to verify that you are using R version 2.15.1 and current packages, similar to those reported here. What other packages were loaded along with `EMBO2012`?

### Solution:

```
> library(EMBO2012)
> sessionInfo()
```

## 1.2.4 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")
```

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")
<bytecode: 0x103334f08>
<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"

```

(Most) methods defined on the `DNAStringSet` class of *Biostrings* and available on the current search path can be found with

```
> showMethods(class="DNAStringSet", where=search())
```

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="EMBO2012")
```

to see, in your web browser, vignettes available in the *EMBO2012* 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")

```

### 1.2.5 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  $100 \times 10$  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] 0.087 0.089 0.078 0.072 0.074

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

[1] 0.001 0.001 0.001 0.000 0.001
```

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.

## 1.2.6 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 `setBreakpoint`.

*Fixme: tryCatch*

## 1.2.7 Resources

Dalgaard [7] provides an introduction to statistical analysis with *R*. Kabaloff [19] provides a broad survey of *R*. Matloff [30] introduces *R* programming concepts. Chambers [6] provides more advanced insights into *R*. Gentleman [10] 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*.

## Chapter 2

# Sequences and Short Reads

### 2.1 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 2.1.

#### 2.1.1 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

Table 2.1: Selected *Bioconductor* packages for representing and manipulating ranges, strings, and other data structures.

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

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 metadata columns:
      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 3.1.

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 metadata columns:
      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): + - *
```

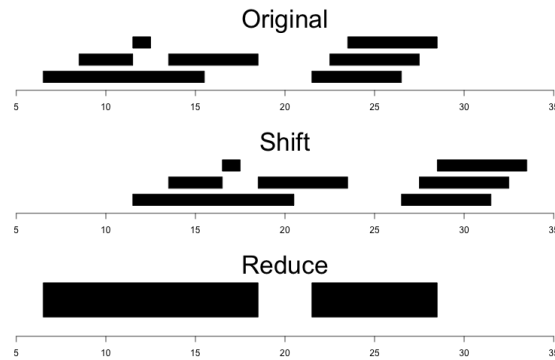


Figure 2.1: Ranges

```
> width(genes)

[1] 6096 620

> length(genes)

[1] 2

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

GRanges with 2 ranges and 0 metadata columns:
      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 `seqnames`, `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 seqnames, but the operations are comparable on *GRanges*. We begin with a simple set of ranges:

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

These and some common operations are illustrated in the upper panel of Figure 2.1 and summarized in Table 2.2.

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 2.1.

```
> 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 2.1.

```
> 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 2.2.

**mcols 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 `mcols` function allows information on each range to be stored and manipulated (e.g., subset) along with the *GRanges* instance. The element metadata is represented as a *DataFrame*, defined in *IRanges* and acting like a standard *R data.frame* but with the ability to hold more complicated data structures as columns (and with element metadata of its own, providing an enhanced alternative to the *Biobase* class *AnnotatedDataFrame*).

Table 2.2: Common operations on *IRanges*, *GRanges* and *GRangesList*.

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

```
> mcols(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 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 metadata columns:

	seqnames	ranges	strand	exon_id	exon_name
	<Rle>	<IRanges>	<Rle>	<integer>	<character>
[1]	chr3R	[19967117, 19967382]	+	45515	<NA>
[2]	chr3R	[19970915, 19971592]	+	45516	<NA>
[3]	chr3R	[19971652, 19971770]	+	45517	<NA>
[4]	chr3R	[19971831, 19972024]	+	45518	<NA>
[5]	chr3R	[19972088, 19972461]	+	45519	<NA>
[6]	chr3R	[19972523, 19972589]	+	45520	<NA>
[7]	chr3R	[19972918, 19973212]	+	45521	<NA>

---

seqlengths:

```
chr3R
27905053
```

The *GRangesList* object has methods one would expect for lists (e.g., `length`, sub-setting). 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")

```

## 2.1.2 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 *EMBO2012* 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 *EMBO2012* package, to calculate GC content:

```

> gcFunction

function (x)
{
  alf <- alphabetFrequency(x, as.prob = TRUE)
  rowSums(alf[, c("G", "C")])
}
<environment: namespace:EMBO2012>

```

Table 2.3: Selected *Bioconductor* packages for sequence reads and alignments.

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

The `gcFunction` is really straight-forward: it invokes the function `alphabetFrequency` from the [\*Biostrings\*](#) package. This returns a simple matrix of exon  $\times$  nucleotide 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)
```

## 2.2 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 2.3. This section focus on the reads and alignments that are the raw material for analysis. Section 3.1 introduces resources for annotating sequences.

### 2.2.1 The *pasilla* data set

As a running example, we use the *pasilla* data set, derived from [3]. 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.

### 2.2.2 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: 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 *EMBO2012* 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: 657900 reads; width: 76 cycles
```

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

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

A DNASTringSet instance of length 3
width seq
[1] 76 AGGTCACCTTGCCCTTTATTATTCGCTGACGCGCG...TCGTGCTATCGGACGCNGGCGCTANCACACGCA
[2] 76 ACCCGTTATGACAAGATCTCTCTTGTGCCACCGTG...ACACCNTCTGTGCTACNAGGCGCGACGNTNCNG
[3] 76 AGCCACACAGCACACCCACAATGCACGCGTCA...GNTTCTGCACATACTTCNACGTCNCCGACACGC
```

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

class: FastqQuality
quality:
A BStringSet instance of length 3
width seq
[1] 76 1=A4@05;7;=.8;@B<A.8515<??;1.;(5(;...9@+B#####!#####!#####
[2] 76 5C#####!#####!#####!#####!#####!#####!#####!#####!#####!#####
[3] 76 .51B91.<7=B;8.%(10;(1(<@+B5' '<.;...#!#####!#####!#####!#####!#####
```

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

A BStringSet instance of length 3
width seq
[1] 37 SRR074430.1 HWUSI-EASXXX:3:2:43:774/1
[2] 37 SRR074430.2 HWUSI-EASXXX:3:2:52:582/1
[3] 38 SRR074430.3 HWUSI-EASXXX:3:2:52:1215/1
```

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

```

> getClass("ShortReadQ")

Class "ShortReadQ" [package "ShortRead"]

Slots:

Name:      quality      sread      id
Class: QualityScore DNABStringSet 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="package:ShortRead")

For instance, the width can be used to demonstrate that all reads consist of 76 nucleotides.

> table(width(fq))

```

```

      76
657900

```

The `alphabetByCycle` function summarizes use of nucleotides at each cycle in a (equal width) *ShortReadQ* or *DNABStringSet* instance.

```

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

```

	cycle							
alphabet	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]
A	63549	109402	124876	140919	155322	178279	82574	111608
C	303400	169804	212501	199667	173161	195572	191666	224396
G	167366	191188	164347	179011	173194	134686	126919	148676
T	115257	185342	153225	134014	153735	146956	254679	170428

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. 1 million reads is a decent number. As the file we are using only has about 660,000 reads, in the following example, we select only a 100,000.

```

> sampler <- FastqSampler(fastqFiles[1], 100000)
> yield(sampler) # sample of 100000 reads

```

```

class: ShortReadQ
length: 100000 reads; width: 76 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

```

> gas0 <- Map(function(fl, nm) {
+   fq <- FastqSampler(fl)
+   qa(yield(fq), nm)
+ }, fastqFiles,

```

```
+ sub("\\.fastq\\.gz", "", basename(fastqFiles)))
> qas <- do.call(rbind, qas0)
> rpt <- report(qas, dest=tempfile())
> browseURL(rpt)
```

Clearly these results are of poor quality. A report from the re-sequencing of the same sample (run accession SRR074431 and SRR074461) is available

```
> rpt <- system.file("SRR074431_SRR074461_qa_report", "index.html", package="EMBO2012")
> browseURL(rpt)
```

### Exercise 8

Use the helper function `bigdata` (defined in the `EMBO2012` package) and the `file.path` and `dir` functions to locate the subset of the SRR074431 fastq file from [3] (the file was obtained as described in the `pasilla` experiment data package and in section 2.2.4).

Input 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 `EMBO2012` 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"         "fastq_subset" "multiplex"
> 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] 0.55
```

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")
```



*Use quality to extract the quality scores of the short reads. Interpret the encoding qualitatively.*

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

```
> head(quality(fq))

class: FastqQuality
quality:
  A BStringSet instance of length 6
    width seq
[1] 76 1=A4@05;7;=.8;@B<A.8515<??;1.;(5(;...9@+B#####!#####!#####
[2] 76 5C#####...#####!#####!#####!#!##
[3] 76 .51B91.<7=B;8.%. (1@; (1(<@+B5' '<;...#!#####!#####!#####
[4] 76 36;9?CCA:8??%<<'%8<B8(=@@?'<%?BCC...#!###!#####!#####!#!#####
[5] 76 .; ;5=;%;) (9*0:C530;;#####...#####!#####!#####
[6] 76 0%1;AA;?=@A'%>B<C?%8'?@?<B>/):@>!...#!###!#####!#####!#!#####

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

[1] 657900      76

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

Most down-stream analysis of short read sequences is based on reads aligned to reference genomes. There are many aligners available, including **BWA** [27, 26], **Bowtie** / **Bowtie2** [22], 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.

```
> fl <- system.file("extdata", "ex1.sam", package="Rsamtools")  
> strsplit(readLines(fl, 1), "\t")[[1]]  
  
[1] "B7_591:4:96:693:509"  
[2] "73"  
[3] "seq1"  
[4] "1"  
[5] "99"  
[6] "36M"  
[7] "*"  
[8] "0"  
[9] "0"  
[10] "CACTAGTGGGCTCATTGTAAATGTGTGGTTTAACTCG"  
[11] "<<<<<<<<<<<<<<<<;<<<<<<<<5<<<<<;<;7"  
[12] "MF:i:18"  
[13] "Aq:i:73"
```

Table 2.4: 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	CIGAR	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

```
[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 2.4. 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 Matches 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*** As introduced - *c.f.* section 2.2 - there are three different packages to read alignments in *R*:

- [ShortRead](#)
- [GenomicRanges](#)
- [Rsamtools](#)

The last two will be described in more details in the next paragraphs.

**GenomicRanges** 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 metadata columns:

	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 -   +
 2L      192058 229262
 2LHet    1002  1119
 2R      141811 165304
 2RHet    21005  23100
 3L      142042 162718

```

3LHet	25865	34553
3R	136564	151778
3RHet	23264	28042
4	6374	5986
dmel_mitochondrion_genome	7807	5599
U	59733	68172
Uextra	212680	263574
X	124124	137632
XHet	1432	1888
YHet	3507	3911

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
```

One important issue when counting reads is to make sure that the reference names in both the annotation and the read files are identical.

### Exercise 11

*Check the reference name in both the `ex` and `aln`. If they are not similar, how could you correct them?*

**Solution:** The names of both references are different; the read ones lack the "chr" prefix and the mitochondrion is called either "dmel\_mitochondrion\_genome" or "M". It is essential to correct them to be able to proceed.

```
> names(seqlengths(ex))

[1] "chr2L"      "chr2LHet"   "chr2R"      "chr2RHet"   "chr3L"      "chr3LHet"
[7] "chr3R"      "chr3RHet"   "chr4"        "chrU"        "chrUextra"   "chrX"
[13] "chrXHet"    "chrYHet"    "chrM"

> names(seqlengths(aln))

[1] "2L"          "2LHet"
[3] "2R"          "2RHet"
[5] "3L"          "3LHet"
[7] "3R"          "3RHet"
[9] "4"           "U"
[11] "Uextra"      "X"
[13] "XHet"        "YHet"
[15] "dmel_mitochondrion_genome"

> levels(seqnames(aln)) <- paste("chr",
+                               sub("dmel_mitochondrion_genome", "M",
+                               levels(seqnames(aln))), sep="")
```

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
2380981	923	2	

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 and correcting the annotation is

```
> counter <-
+   function(filePath, range)
+ {
+   aln <- readGappedAlignments(filePath)
+   strand(aln) <- "*"
+   levels(seqnames(aln)) <- paste("chr",
+                                   sub("dmel_mitochondrion_genome", "M",
+                                       levels(seqnames(aln))), sep="")
+   hits <- countOverlaps(aln, range)
+   countOverlaps(range, aln[hits==1])
+ }
```

This can be applied to all files using `sapply`

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

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

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

**Rsamtools** 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 12

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

Use the *ScanBamParam* object to query a BAM file, and calculate the GC content of all aligned reads. Summarize the GC content as a histogram (Figure 2.2).

## Solution:

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

**Advanced *Rsamtools* usage** The *Rsamtools* package has more advanced functionalities:

1. function to count, index, filter, sort BAM files
2. function to access the header only
3. the possibility to access SAM attributes (tags)

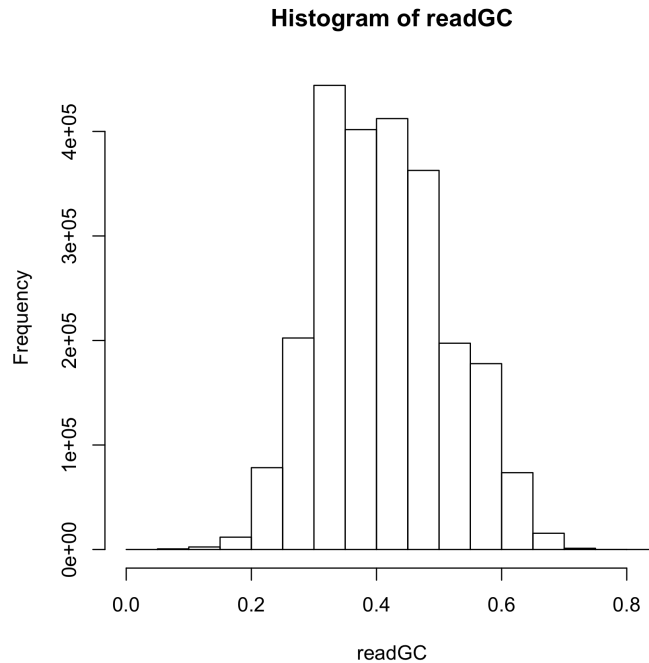


Figure 2.2: GC content in aligned reads

4. manipulate the CIGAR string
5. create BAM libraries to represent a study set (BamViews)
6. ...

### Exercise 13

Find out the function that permit to scan the BAM header and retrieve the header of the first BAM file in the `bigdata()` bam subfolder. What information does it contain?

**Solution:** It contains the reference sequence length and names as well as the name, version and command line of the tool used for performing the alignments.

```
> scanBamHeader(fls[1])
```

### Exercise 14

The SAM/BAM format contains a tag: "NH" that defines the total number of valid alignments reported for a read. How can you extract that information from the same first bam file and plot it as an histogram?

**Solution:**

```
> param <- ScanBamParam(tag="NH")
> nhs <- scanBam(fls[[1]], param=param)[[1]]$tag$NH
```

So it seems a majority of our reads have multiple alignments! Some filtering might be required.

### Exercise 15

The CIGAR string contains interesting information, in particular, whether or not a given match contain indels. Using the first bam file, can you get a matrix of all seven CIGAR operations? And find out the intron size distribution?

### Solution:

```
> param <- ScanBamParam(what="cigar")
> cigars <- scanBam(fls[[1]], param=param)[[1]]$cigar
> cigar.matrix <- cigarOpTable(cigars)
> intron.size <- cigar.matrix[, "N"]
> intron.size[intron.size>0]
> plot(density(intron.size[intron.size>0]))
> histogram(log10(intron.size[intron.size>0]), xlab="intron size (log10 bp)")
```

### Exercise 16

Look up the documentation for the *BamViews* and using the [leeBamViews](#), create a *BamViews* instance. Afterwards, use some of the accessors of that object to access e.g. to view the file paths or the sample names

### Solution:

```
> library(leeBamViews)
> bpaths = dir(system.file("bam", package="leeBamViews"), full=TRUE, patt="bam$")
> gt<-do.call(rbind, strsplit(basename(bpaths), "_"))[,1]
> geno<-substr(gt, 1, nchar(gt)-1)
> lane<-substr(gt, nchar(gt), nchar(gt))
> pd = DataFrame(geno=geno, lane=lane, row.names=paste(geno, lane, sep="."))
> bs1 = BamViews(bamPaths=bpaths, bamSamples=pd,
+   bamExperiment=list(annotation="org.Sc.sgd.db"))
> bamPaths(bs1)
> bamSamples(bs1)
```

### Exercise 17

Finally, extract the coverage for the chromosome “Scchr13” 861250:863000 locus for every sample in the *bs1* object

**Solution:** `sel <- GRanges(seqnames = "Scchr13", IRanges(start = 861250, end = 863000), strand="+")`  
`covex = RleList(lapply(bamPaths(bs1), function(x) coverage(readGappedAlignments(x))[[1]]))`

This offer an interesting way to process multiple sample at the same time when you're interested in a particular locus.

## 2.2.4 Alignments and other *Bioconductor* packages

In the following, an excerpt of additional functionalities offered by *Bioconductor* packages is presented. It is far from being a complete overview, and as such only aims at giving a feel for what's out there.

**Retrieving data using *SRADB*** Most journals require the raw data to be deposited in a public repository, such as [GEO](#), [SRA](#) or [ENA](#). The *SRADB* package offers the possibility to list the content of this archive, and to retrieve raw (fastq or sra) files.

### Exercise 18

Using the [pasilla](#) package, retrieve the submission accession of that dataset (check out that package vignette)

### Solution:

```
> vignette(package="pasilla")
> vignette("create_objects")
> geo.acc <- "GEO: GSE18508"
```

Now that we have the GEO ID, we need to convert it to an SRA ID. You can either use the [GEO](#), [SRA](#) or [ENA](#) website for this or if you are slightly familiar with SQL, just use the [SRADB](#) package.

### Exercise 19

*Look into the [SRADB](#) package vignette to figure out how to do this.*

**Solution:** Accessing the vignette and reading it tells us

```
> library(SRADb)
> vignette("SRADB")
```

- we need to download the [SRADB](#) sqlfile
- we need to create a connection to the locally downloaded database
- we need to query that database with our submission\_alias: "GEO: GSE18508" to retrieve the SRA submission\_accession.

The first step requires the download of a 150Mb compressed large file, so to limit the downloading time, you could do it in groups of 3-4 persons

```
> sqlfile <- getSRADBFile()
> sra_con <- dbConnect(SQLite(),sqlfile)
> sra_acc <- dbGetQuery(sra_con,paste("select submission_accession ",
+                                     "from submission ",
+                                     'where submission_alias = "',
+                                     geo_acc, '";', sep=""))
```

The retrieved `sra_acc` is: "SRA010243".

Now that we have that accession, the vignette tells us how to get every experiment, sample, run, ... associated with this submission accession.

### Exercise 20

*There are at least two possibilities to do so, one using an SQL query and the other one using a function of the packages. What would be that function?*

**Solution:** For those that like SQL:

```
> run_acc <- dbGetQuery(sra_con,paste("select run_accession ",
+                                     "from run ",
+                                     'where submission_accession = "',
+                                     sra_acc, '";', sep=""))$run_accession
```

For those that like functions:

```
> sraConvert(sra_acc,sra_con=sra_con)
> run_acc <- sraConvert(sra_acc,"run",sra_con=sra_con)$run
```

### Exercise 21

*Now that we've got the list of runs, it would be interesting to get more information about the corresponding fastq file.*

**Solution:**

```
> info <- getFASTQinfo(run_acc,srcType="ftp")
```

And the final step would be to download the fastq file(s) of interest.



## Exercise 22

Retrieve the shortest fastq file from that particular submission.

### Solution:

```
> getSRAfile(in_acc=info[which.min(info[, "run.read.count"]), "run"],
+           sra_con, destDir = getwd(),
+           fileType = 'fastq', srcType = 'ftp' )
```

Well, that's almost it. As we are tidy people, we clean after ourselves.

```
> dbDisconnect( sra_con )
```

**Demultiplexing using [easyRNASeq](#)** *Note: This section does not apply to all datasets but only to multiplexed ones. Since the data we loaded so far into R was not multiplexed we will use a different dataset here.*

Nowadays, NGS machines produces so many reads (e.g. 40M for Illumina GAIIx, 100M for ABI SOLiD4 and 160M for an Illumina HiSeq), that the coverage obtained per lane for the transcriptome of organisms with small genomes, is very high. Sometimes it's more valuable to sequence more samples with lower coverage than sequencing only one to very high coverage, so techniques have been optimised for sequencing several samples in a single lane using 4-6bp barcodes to uniquely identify the sample within the library[23]. This is called multiplexing and one can on average sequence 12 yeast samples at 30X coverage in a single lane of an Illumina GenomeAnalyzer GAIIx (100bp read, single end). This approach is very advantageous for researchers, especially in term of costs, but it adds an additional layer of pre-processing that is not as trivial as one would think. Extracting the barcodes would be fairly straightforward, but for the average 0.1-1 percent sequencing error rate that introduces a lot of multiplicity in the actual barcodes present in the samples. A proper design of the barcodes, maximising the Hamming distance [15] is an essential step for proper de-multiplexing.

The data we loaded into R in the previous section was not mutiplexed, so we now load a different FASTQ file where the 4 different samples sequenced were identified by the barcodes "ATGGCT", "TTGCGA", "ACACTG" and "ACTAGC".

```
> reads <- readFastq(file.path(bigdata(), "multiplex", "multiplex.fq.gz"))
> # filter out reads with more than 2 Ns
> filter <- nFilter(threshold=2)
> reads <- reads[filter(reads)]
> # access the read sequences
> seqs <- sread(reads)
> # this is the length of your adapters
> barcodeLength <- 6
> # get the first 6 bases of each read
> seqs <- narrow(seqs, end=barcodeLength)
> seqs
> length(table(as.character(seqs)))
```

So it seems we have 1953 barcodes instead of 6 ...

## Exercise 23

Which barcode is most represented in this library? Plot the relative frequency of the top 20 barcodes. Try:

- using the function `table` to count how many times each barcode occurs in the library, you can't apply this function to `seqs` directly you must convert it first to a character vector with the `as.character` function
- sort the counts object you just created with the function `sort`, use `decreasing=TRUE` as an argument for `sort` so that the elements are sorted from high to low (use `sort( ..., decreasing=TRUE )`)

- look at the first element of your sorted counts object to find out with barcode is most represented
- find out what the relative frequency of each barcode is by dividing your counts object by the total number of reads (the function sum might be useful)
- plot the relative frequency of the top 20 barcodes by adapting these function calls:

```
> # set up larger margins for the plot so we can read the barcode names
> par(mar=c(5, 5, 4, 2))
> barplot(..., horiz=T, las=1, col="orange" )
```

**Solution:**

```
> barcount = sort(table(as.character(seqs)), decreasing=TRUE)
> barcount[1:10] # TTGCGA
> barcount = barcount/sum(barcount)
> par( mar=c(5, 5, 4, 2))
> barplot(barcount[1:20], horiz=TRUE, las=1, col="orange" )
```

### Exercise 24

The designed barcodes ("ATGGCT", "TTGCGA", "ACACTG" and "ACTAGC") seem to be equally distributed, what is the percentage of reads that cannot be assigned to a barcode?

**Solution:**

```
> signif((1-sum(barcount[1:4]))*100,digits=2) # ~6.4%
```

We will now iterate over the 4 barcodes, split the reads between them and save a new fastq file for each:

```
> barcodes = c("ATGGCT", "TTGCGA", "ACACTG", "ACTAGC")
> # iterate through each of these top 10 adapters and write
> # output to fastq files
> for(barcode in barcodes) {
+   seqs <- sread(reads) # get sequence list
+   qual <- quality(reads) # get quality score list
+   qual <- quality(qual) # strip quality score type
+   mismatchVector <- 0 # allow no mismatches
+
+   # trim sequences looking for a 5' pattern
+   # gets IRanges object with trimmed coordinates
+   trimCoords <- trimLRPatterns(Lpattern=barcode,
+   subject=seqs, max.Lmismatch=mismatchVector, ranges=T)
+
+   # generate trimmed ShortReadQ object
+   seqs <- DNASTringSet(seqs, start=start(trimCoords),
+   end=end(trimCoords))
+   qual <- BStringSet(qual, start=start(trimCoords),
+   end=end(trimCoords))
+
+   # use IRanges coordinates to trim sequences and quality scores
+   qual <- SFASTQQuality(qual) # reapply quality score type
+   trimmed <- ShortReadQ(sread=seqs, quality=qual, id=id(reads))
+
+   # rebuild reads object with trimmed sequences and quality scores
```

```

+   # keep only reads which trimmed the full barcode
+   trimmed <- trimmed[start(trimCoords) == barcodeLength + 1]
+
+   # write reads to Fastq file
+   outputFileName <- paste(barcode, ".fq", sep="")
+   writeFastq(trimmed, outputFileName)
+
+   # export filtered and trimmed reads to fastq file
+   print(paste("wrote", length(trimmed),
+               "reads to file", outputFileName))
+ }

```

You should have four new FASTQ files: AACTG.fq, ACTAGC.fq ATGGCT.fq and TTGCGA.fq with the reads (the barcodes have been trimmed) corresponding to each multiplexed sample. The next step would be to align these reads against your reference genome.

### Aligning reads using *Rsubread*

```

> library(Rsubread)
> library(BSgenome.Dmelanogaster.UCSC.dm3)
> chr4 <- DNASTringSet(unmasked(Dmelanogaster[["chr4"]]))
> names(chr4) <- "chr4"
> writeXStringSet(chr4, file="dm3-chr4.fa")
> ## create the indexes
> dir.create("indexes")
> buildindex(basename=file.path("indexes", "dm3-chr4"),
+           reference="dm3-chr4.fa")
> ## align the reads
> sapply(dir(pattern="*\\.fq$"), function(fil){
+   ## align
+   align(index=file.path("indexes", "dm3-chr4"),
+         readfile1=sub("\\.fq$", "", fil),
+         nsubreads=2, TH1=1,
+         output_file=sub("\\.fq$", "\\sam", fil)
+   )
+
+   ## create bam files
+   asBam(file=sub("\\.fq$", "\\sam", fil),
+         destination=sub("\\.fq$", "", fil),
+         indexDestination=TRUE)
+ })

```

And that's it you have filtered, demultiplexed and aligned your reads!

## 2.2.5 Resources

There are extensive vignettes for *Biostrings* and *GenomicRanges* packages. A useful on-line resource is from [Thomas Girke's group](#).

## Chapter 3

# Annotation of Genes and Genomes

### 3.1 Annotation

*Bioconductor* provides extensive annotation resources, summarized in Figure 3.1. These can be *gene*-, or *genome*-centric. Annotations can be provided in packages curated by *Bioconductor*, or obtained from web-based resources. Gene-centric *AnnotationDbi* packages include:

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

Examples of genome-centric packages include:

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

Web-based resources include

- *biomaRt* to query *biomart* resource for genes, sequence, SNPs, and etc.
- *rtracklayer* for interfacing with browser tracks, especially the UCSC genome browser.

#### 3.1.1 Gene-centric annotations with *AnnotationDbi*

Organism-level (‘org’) packages contain mappings between a central identifier (e.g., Entrez gene ids) and other identifiers (e.g. GenBank or Uniprot accession number, RefSeq id, etc.). The name of an 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 25

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.

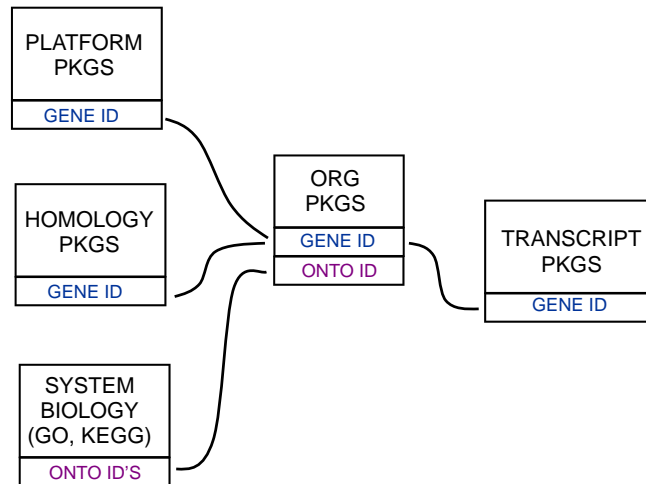


Figure 3.1: Annotation Packages: the big picture

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"	"GOALL"	"EVIDENCEALL"	"ONTOLOGYALL"	"FLYBASE"
[26]	"FLYBASECG"	"FLYBASEPROT"			

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

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

```
> uniprotKeys <- head(keys(org.Dm.eg.db, keytype="UNIPROT"))
```

```
> cols <- c("SYMBOL", "PATH")
```

```
> select(org.Dm.eg.db, keys=uniprotKeys, cols=cols, keytype="UNIPROT")
```

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

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

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

[1] 36

> head(kegg, 3)

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

### Exercise 26

For convenience, the *EMBO2012* package contains *lrTest*, an object representing the results of a RNA-seq gene-level differential expression analysis of the pasilla RNA-seq data using the *edgeR* package (you will create a similar object later in the course, using *DESeq* and *DEXSeq*). The following code loads this data and creates a ‘top table’ of the ten most differentially represented genes. This top table is then coerced to a *data.frame*.

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

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

**Solution:**

```
> 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>

### 3.1.2 Genome-centric annotations with *GenomicFeatures*

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

### Exercise 27

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 metadata columns:

	seqnames	ranges	strand	cds_id	cds_name	exon_rank
	<Rle>	<IRanges>	<Rle>	<integer>	<character>	<integer>
[1]	chr3R	[19970946, 19971592]	+	39378	<NA>	2
[2]	chr3R	[19971652, 19971770]	+	39379	<NA>	3
[3]	chr3R	[19971831, 19972024]	+	39380	<NA>	4
[4]	chr3R	[19972088, 19972461]	+	39381	<NA>	5
[5]	chr3R	[19972523, 19972589]	+	39382	<NA>	6
[6]	chr3R	[19972918, 19973094]	+	39383	<NA>	7

---

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

The following code loads the appropriate BSgenome package; the *Dmelanogaster* object refers to the whole genome sequence represented in this package. The remaining steps extract the DNA sequence of each transcript, and translates these to amino acid sequences. Issues of strand are handled 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...TTGCTGCCCCATTTCGAACTTTAG FBtr0085755
[3]  2217 ATGGCACTCAAGTTTCCACAGT...TTGCTGCCCCATTTCGAACTTTAG FBtr0085756

> head(translate(txx), 3)

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

### 3.1.3 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 28

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

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

#### Solution:

```
> library(biomaRt)
> head(listMarts(), 3) ## list the marts
> head(listDatasets(useMart("ensembl")), 3) ## mart datasets
> ensembl <- ## fully specified mart
+   useMart("ensembl", dataset = "hsapiens_gene_ensembl")
> head(listFilters(ensembl), 3) ## filters
> myFilter <- "chromosome_name"
> head(filterOptions(myFilter, ensembl), 3) ## return values
> myValues <- c("21", "22")
> head(listAttributes(ensembl), 3) ## attributes
```



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

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

## Chapter 4

# Estimating Expression over Genes and Exons

This chapter<sup>1</sup> describes a RNA-Seq analysis use-case. RNA-Seq [32] was introduced as a new method to perform Gene Expression Analysis, using the advantages of the high throughput of *Next-Generation Sequencing* (NGS) machines.

### 4.1 Counting reads over known genes and exons

The goal of this use-case is to generate a count table for the selected genic features of interest, *i.e.* exons, transcripts, gene models, *etc.*

To achieve this, we need to take advantage of all the steps performed previously in that document.

1. the alignments information has to be retrieved
2. the corresponding annotation need to be fetched
3. the read coverage per genic feature of interest determined

#### Exercise 29

Can you associate at least a Bioconductor package to every of these tasks?

**Solution:** There are numerous choices, as an example in the following we will go for the following set of packages:

- a. *Rsamtools*
- b. *BiomaRt*
- c. *GenomicRanges*

#### 4.1.1 The alignments

This was introduced in section 2.2.3, page 32. In this section we will import the data using the *GenomicRanges* `readGappedAlignments`. This will create a `GappedAlignments` object that contains only the reads that aligned to the genome.

#### Exercise 30

In the introduction of that chapter, we said we would be using the *Rsamtools*, why are we using *GenomicRanges* instead?

---

<sup>1</sup>The author want to thank Ângela Gonçalves for parts of the present chapter

**Solution:** Because [GenomicRanges](#) `readGappedAlignments` function uses the [Rsamtools](#) `scanBam` function internally and accept most of the parameters of that one.

### Exercise 31

Using what was introduced in section 2.2.3, read in the first bam file from the `bigdata()` bam folder. Remember that the protocol used was not strand-specific.

**Solution:** First we scan the bam directory:

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

Then we read the first file:

```
> library(GenomicRanges)
> aln <- readGappedAlignments(fls[1])
> strand(aln) <- "*"
```

As we have seen, many of these reads actually align to multiple locations. In a first basic analysis - to get a feel for the data - such reads could be ignored.

### Exercise 32

Filter the multiple alignment reads. Think of the “NH” tag.

```
> param <- ScanBamParam(tag="NH")
> nhs <- scanBam(fls[[1]], param=param)[[1]]$tag$NH
> aln <- aln[nhs==1,]
```

Now that we have our alignment, let’s get the corresponding genome annotation.

## 4.1.2 The annotation

To map the alignments to their respective features, we need to know the genome composition of the studied organism, in our case *D. melanogaster*. As introduced in section 3.1, page 43, there are again numerous possibilities to do this.

### Exercise 33

Can you list other Bioconductor packages than [biomaRt](#) for doing this?

**Solution:** There are e.g. [GenomicFeatures](#), [rtracklayer](#), ...

In this practical we will be using [biomaRt](#) to download the data from Ensembl. The [biomaRt](#) package provides an interface to a growing collection of databases such as Ensembl, Uniprot and HapMap. In this case we can use [EnsemblMetazoa14](#) to retrieve the annotation for our organism:

```
> library(biomaRt)
> ensembl <- useMart(biomart="metazoa_mart_14",
+                   dataset="dmelanogaster_eg_gene")
> fields = c("chromosome_name",
+ "strand",
+ "ensembl_gene_id",
+ "ensembl_exon_id",
+ "start_position",
+ "end_position",
+ "exon_chrom_start",
+ "exon_chrom_end")
> annot.df <- getBM(attributes=fields,
+                   mart=ensembl)
```

Now that we have retrieved the annotation, it is necessary to convert them into a format that we can use for summarizing the read counts.

### Exercise 34

Convert the obtained `data.frame` into a `GRanges` object. And do not forget to verify if the reference name needs to be edited.

**Solution:** The alignment file and the annotation file have a common subset of reference name, so let's proceed.

```
> annot <- GRanges(seqnames = Rle(annot.df$chromosome_name),
+                 ranges = IRanges(
+                   start=annot.df$exon_chrom_start,
+                   end = annot.df$exon_chrom_end),
+                 strand = Rle(annot.df$strand),
+                 exon = annot.df$ensembl_exon_id,
+                 gene = annot.df$ensembl_gene_id)
> annot
```

The experimental protocol that generated this dataset did not retain strand information; therefore we should set all strand locations in the `annot` object to the wildcard: `*` to be able to overlap reads with the annotation later on:

```
> # check how the strand information is encoded
> strand(annot)
> # this object is compressed to save space
> # but like this it's difficult to visualise it
> # we can look at the first 10 exons:
> as.vector(strand(annot))[1:10]
> # change strand information to *
> strand(annot) <- "*"
> as.vector(strand(annot))[1:10]
```

Now that we have the alignments (`aln` object) and the genome annotation (`annot` object), we can quantify gene expression by counting reads over all exons of a gene and summing them together. One thing to keep in mind is that special care must be taken in dealing with reads that overlap more than one feature (e.g. overlapping genes, isoforms), and thus might be counted several times in different features. To deal with this we can use any of the approaches summarised in Figure 4.1:

The [GenomicRanges](#) `summarizeOverlaps` offer different possibilities to summarize reads per features:

```
> counts1 <- summarizeOverlaps(annot, aln, mode="Union")
> counts2 <- summarizeOverlaps(annot, aln, mode="IntersectionStrict")
> counts3 <- summarizeOverlaps(annot, aln, mode="IntersectionNotEmpty")
```

### Exercise 35

Create a `data.frame` or a matrix of the results above and figure out if any differences can be observed. E.g check for difference in the row standard deviation (using the `apply` and `sd` functions).

**Solution:**

```
> exonCountsTable <- data.frame(
+   union = assays(counts1)$counts,
+   intStrict = assays(counts2)$counts,
+   intNotEmpty = assays(counts3)$counts)
> rownames(exonCountsTable) <- elementMetadata(annot)$exon
> sds <- apply(exonCountsTable,1,sd)
> sum(sds!=0)
> sum(sds!=0)/length(sds)
> exonCountsTable[which.max(sds),]
> annot[which.max(sds),]
```

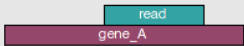
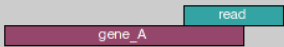


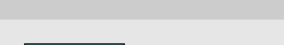
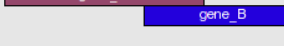
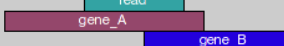
	union	intersection_strict	intersection_nonempty
	gene_A	gene_A	gene_A
	gene_A	no_feature	gene_A
	gene_A	no_feature	gene_A
	gene_A	gene_A	gene_A
	gene_A	gene_A	gene_A
	ambiguous	gene_A	gene_A
	ambiguous	ambiguous	ambiguous

Figure 4.1: Overlap modes; Image from the [HTSeq](#) package developed by Simon Anders.

So it appears that we have about 8,300 cases where these counting generate different results (11% of the total), and that the exon “FBgn0064225:2” shows the largest difference.

For a detailed analysis, it would be important to adequately choose one of the intersection modes above, however for the remainder of this section, we will use the “union” set. We can now finally sum the exons together to get a vector of read counts per gene:

```
> exonCounts <- exonCountsTable[, "union"]
> # look at the counts of the first few exons
> head(exonCounts)
> # assign the name of the corresponding gene to each exon
> names(exonCounts) <- elementMetadata(annot)$gene
> head(exonCounts)
> # create a list in which each element corresponds to
> # all exons of the same gene
> splitCounts <- split(exonCounts, names(exonCounts) )
> head(splitCounts)
> # sum the exons counts in each gene using the function sapply
> # sapply will sum the values in each element of the list
> geneCounts <- sapply( splitCounts, function(x) sum(x) )
> head(geneCounts)
```

As before for reads aligning to multiple places in the genome, choosing to take the union when reads overlap several features is a simplification we may not want to do. There are several methods that probabilistically estimate the expression of overlapping features [25, 42, 43].

This concludes that section on counting reads per known features. In the next section, we will look at how novel transcribed regions could be identified.

### 4.1.3 Discovering novel transcribed regions

One main advantage of RNA-seq experiments over microarrays is that they can be used to identify any transcribed molecule, including unknown transcripts and isoforms, as well as other regulatory transcribed elements. To identify such new elements, several methods are available to recreate and annotate transcripts, e.g. Cufflinks[42], Oases[39], Trinity[13], to mention some of them. We can use Bioconductor tools as well, to identify loci and quantify counts without prior annotation knowledge.

**Defining transcribed regions** The process begins with calculating the coverage, using the method from the *GenomicRanges* package:

```
> cover <- coverage(aln)

> cover
> # this object is compressed to save space
> # we can look at a section of chromosome say between 1000 and 3000
> # which gives us the number of read overlapping each of those bases
> as.vector(cover[[1]])[1000:3000]
```

Next, “islands” of expression can be formed using the `slice` function. The peak height for the islands can be found using the `viewMaxs` function and the island widths can be found using the `width` function:

```
> islands <- slice(cover, 1)
> islandPeakHeight <- viewMaxs(islands)
> islandWidth <- width(islands)
```

While some more sophisticated approaches can be used to find exons de novo, we can use a simple approach whereby we select islands whose maximum peak height is 2 or more and whose width is 114 bp (150% of the read size) or more to be candidate exons. The `elementLengths` function shows how many of these candidate exons appear on each chromosome:

```
> candidateExons <- islands[islandPeakHeight >= 2L & islandWidth >= 114L]
> candidateExons[[1]]
```

Remember that we used an aligner which is capable of mapping reads across splice junctions in the genome. For example:

```
> aln[94120,]
```

GappedAlignments with 1 alignment and 0 metadata columns:

	seqnames	strand	cigar	qwidth	start	end	width
	<Rle>	<Rle>	<character>	<integer>	<integer>	<integer>	<integer>
[1]	2L	*	24M304N52M	76	17601125	17601504	380
	ngap						
	<integer>						
[1]	1						
---							
seqlengths:							

```
                2L ... dmel_mitochondrion_genome
23011544 ...                               19517
```

has 24 bases aligned at coordinate 17,601,125 of chromosome 2L, then there is a gap of 304 bases and the remainder 52 bases map again at coordinate 17,601,452 (the CIGAR string is 24M304N52M). The *GenomicRanges* package is aware of this. Have a look at the coverage for that region:

```
> cover[["2L"]][17601125:17601504]
```

integer-Rle of length 380 with 3 runs

```
Lengths: 24 304 52
Values :  2  0  1
```

**Overlapping with known annotation** We now want to find out if our candidate exons overlap with any known annotation. For this we need to transform the candidateExons object into a GRanges object like we did for the annotation. We will concentrate only on the chromosome “4”

```
> candidateExonRanges <- GRanges( seqname=Rle("4"),  
+ ranges=candidateExons[["4"]], strand=Rle("*") )
```

We then use function countOverlaps (you could also use summarizeOverlaps) to find how many times each candidate exon overlaps with the annotation:

```
> chr4 <- annot[seqnames(annot) == "4",]  
> candidateOverlap <- countOverlaps(candidateExonRanges, chr4)  
> # select only exons with an overlap count equal to 0  
> nonoverlapExons <- GRanges( seqname=Rle("4"),  
+ ranges=candidateExons[["4"]][candidateOverlap == 0], strand=Rle("*") )  
> # compare the new list of exons with the previous  
> nonoverlapExons  
> candidateExonRanges
```

**Exporting and visualising the novel regions** The novel regions just defined are more conveniently visualised alongside current annotation using a genome browser. The rtracklayer package provides useful functions to import and export genomic annotation tracks in standard formats such as BED, GFF and WIG, that can be loaded into web-based genome browsers such as UCSC and Ensembl:

```
> library(rtracklayer)  
> # GRanges objects are easily exported to standard formats  
> # using the rtracklayer package  
> export(nonoverlapExons, "novelexons.bed")
```

The previous function produced a file novelexons.bed. Try visualising this file in EnsemblMetazoa by following these instructions (extracted from the [Ensembl help pages](#)<sup>2</sup>):

1. access the *Drosophila melanogaster* EnsemblMetazoa [website](#)<sup>3</sup>
2. click on [Manage your data] in the side menu
3. click on “Upload Data”
4. enter the name for the track (e.g. novelexons) in the “Name for this upload (optional)” text box
5. select “Data format: BED”
6. click [Browse...] behind “Upload file:”
7. select the novelexons.bed file just created
8. click [Upload]
9. click “Go to first region with data”

Your data should now be shown as a new track on the “Region in detail” page.

This concludes the section on summarizing counts. As you could realize, juggling with the different package for manipulating the alignment and annotation requires some coding. To facilitate this a number of “workflow” package are available at *Bioconductor*. The next section gives a brief introduction of *easyRNASeq* (a biased selection...)

---

<sup>2</sup>[http://metazoa.ensembl.org/info/website/tutorials/Ensembl\\_upload\\_exercises.pdf](http://metazoa.ensembl.org/info/website/tutorials/Ensembl_upload_exercises.pdf)

<sup>3</sup>[http://metazoa.ensembl.org/Drosophila\\_melanogaster/Info/Index](http://metazoa.ensembl.org/Drosophila_melanogaster/Info/Index)

## 4.2 Using easyRNASeq

Let us redo what was done in the previous section. Note that most of the *RNAseq* object slots are optional. However, it is advised to set them, especially the *readLength* and the *organismName*; to help having a proper documentation of your analysis. The *organismName* slot is actually mandatory if you want to get genomic annotation using *biomaRt*. In that case, you need to provide the name as specified in the corresponding *BSgenome* package, *i.e.* “*Dmelanogaster*” for the *BSgenome.Dmelanogaster.UCSC.dm3* package.

```
> ## load the library
> library("easyRNASeq")
> count.table <- easyRNASeq(filesDirectory=dirname(fls[1]),
+                           filenames=basename(fls),
+                           organism="Dmelanogaster",
+                           readLength=76L,
+                           annotationMethod="rda",
+                           annotationFile=system.file(
+                             "data",
+                             "gAnnot.rda",
+                             package="RnaSeqTutorial"),
+                           format="bam",
+                           gapped=TRUE,
+                           count="exons")

> head(count.table)
> dim(count.table)
```

That is all. In one command, you got the count table for your 2 samples!

**Warnings** As you could see when running the previous example, warnings were emitted and quite rightly so.

1. about the annotation: The annotation we are using here is redundant and this at two levels. First, some exons overlap. These are alternative exons from different transcript isoforms. Second, the annotation contains the information about all the possible different transcript isoforms. This means that some exons are duplicated. Therefore counting by exons or transcripts using these annotation will result in counting some of the reads several times. There might be reasons one might want to do that, but as it is probably not what you want when performing an RNA-Seq analysis, the warning is emitted. As this can be a very significant source of error, all the examples here will emit this warning. The ideal solution is to provide an annotation object that contains no overlapping features. The *disjoin* function from the *IRanges* package offers a way to achieve this.
2. about potential naming issue in the input file: It is (sadly) very frequent that the sequencing facilities use different naming conventions for the chromosomes they report in the alignment files. It is therefore very frequent that the annotation provided to *easyRNASeq* uses different chromosome names than the alignment file. These warnings are there to inform you about this issue.

**Details** The *easyRNASeq* function currently accepts the following *annotationMethods*:

- “*biomaRt*” use *biomaRt* to retrieve the annotation
- “*env*” use a *RangedData* or *GRanges* class object present in the environment
- “*gff*” reads in a gff version 3 file
- “*gtf*” reads in a gtf file
- “*rda*” load an RData object. The object needs to be named *gAnnot* and of class *RangedData* or *GRanges*.



The reads can be read in from BAM files or any format supported by *ShortRead*.  
The reads can be summarized by:

- exons
- features (any features such as introns, enhancers, *etc.*)
- transcripts
- geneModels (a geneModel is the set of non overlapping loci (*i.e.* synthetic exons) that represents all the possible exons and UTRs of a gene. Such geneModels are essential when counting reads as they ensure that no reads will be accounted for several times. *E.g.*, a gene can have different isoforms, using different exons, overlapping exons, in which case summarizing by exons might result in counting a read several times, once per overlapping exon. *N.B.* Assessing differential expression between transcripts, based on synthetic exons is something possible since the release 2.14 of R, using the *DEXSeq* package available from Bioconductor.

The results can be exported in five different formats:

- count table (the default, a n (features) x m (samples) **matrix**).
- a *DESeq* [1] *countDataSet* class object. Useful to perform further analyses using the *DESeq* package.
- an *edgeR* [37] *DGEList* class object. Useful to perform further analyses using the *edgeR* package.
- an *RNAseq* class object. Useful for performing additional pre-processing without re-loading the reads and annotations.

The obtained results can optionally be corrected as *Reads per Kilobase of feature per Million reads in the library* (RPKM, [32]) or normalized using the *DESeq* or *edgeR* packages.

For more details and a complete overview of the *easyRNASeq* package capabilities, have a look at the *easyRNASeq* vignette.

```
> vignette("easyRNASeq")
```

### Exercise 36

From the same input files and annotations, generate an object of class *SummarizedExperiment*.

**Solution:**

```
> sumExp <- easyRNASeq(filesDirectory=dirname(fls[1]),
+                      filenames=basename(fls),
+                      organism="Dmelanogaster",
+                      readLength=76L,
+                      annotationMethod="rda",
+                      annotationFile=system.file(
+                        "data",
+                        "gAnnot.rda",
+                        package="RnaSeqTutorial"),
+                      format="bam",
+                      gapped=TRUE,
+                      count="exons",
+                      outputFormat="SummarizedExperiment")
```

See the *GenomicRange* package *SummarizedExperiment* class for more details on last three accessors used in the following.

```
> ## the counts
> assays(sumExp)
> ## the sample info
> colData(sumExp)
> ## the 'features' info
> rowData(sumExp)
```

**Caveats** `easyRNASeq` is still under active development and as such still lacks some essential data processing (*e.g.* strand specific sequencing is not yet supported). Have a look at the vignette for more details.

## Chapter 5

# Working with Called Variants

### 5.1 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.

#### 5.1.1 Variant call format (VCF) files

Data are read from a VCF file and variants identified according to region such as `coding`, `intron`, `inter-genic`, `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 37

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

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

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

*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 metadata column:

	seqnames	ranges	strand	paramRangeID
	<Rle>	<IRanges>	<Rle>	<factor>
rs7410291	22	[50300078, 50300078]	*	<NA>
rs147922003	22	[50300086, 50300086]	*	<NA>
rs114143073	22	[50300101, 50300101]	*	<NA>

---

```
seqlengths:
22
NA
```

Rename chromosome levels:

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

Discover whether SNPs are located in dbSNP:

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

```
inDbSNP
FALSE TRUE
6126 4250
```

Create a data frame summarizing SNP quality and dbSNP membership:

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

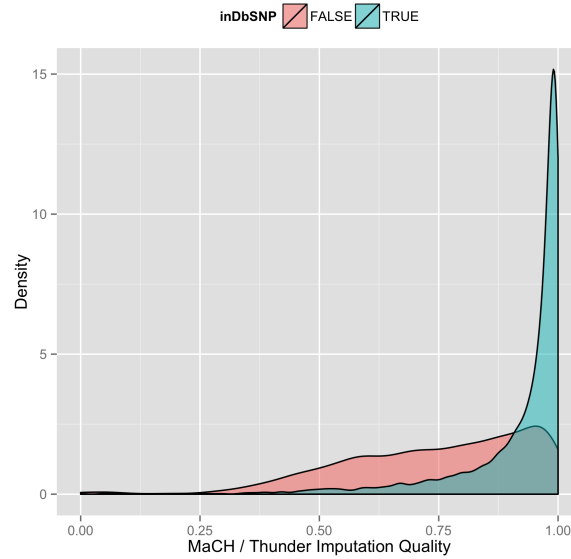


Figure 5.1: Quality scores of variants in dbSNP, compared to those not in dbSNP.

Table 5.1: 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

Finally, visualize the data, e.g., using `ggplot2` (Figure 5.1).

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

### 5.1.2 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 5.1.

#### Exercise 38

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

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

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

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

```
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
> fl <- system.file("extdata", "chr22.vcf.gz", package="VariantAnnotation")
> vcf <- readVcf(fl, "hg19")
> 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 metadata columns:

	seqnames	ranges	strand	LOCATION	QUERYID	TXID
	<Rle>	<IRanges>	<Rle>	<factor>	<integer>	<integer>
[1]	chr22	[50301422, 50301422]	*	coding	24	73482
[2]	chr22	[50301476, 50301476]	*	coding	25	73482
[3]	chr22	[50301488, 50301488]	*	coding	26	73482

	CDSID	GENEID
	<integer>	<character>
[1]	217009	79087
[2]	217009	79087
[3]	217009	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(loc$GENEID, loc$QUERYID)
> table(sapply(splt, function(x) length(unique(x)) > 1))
```

```
FALSE TRUE
956    15
```

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

```
113730 1890 23209
22     15     30
```

**Amino acid coding changes** `predictCoding` computes amino acid coding changes for non-synonymous variants. Only ranges in `query` that overlap with a coding region in `subject` are considered. Reference sequences are retrieved from either a `BSgenome` or fasta file specified in `seqSource`. Variant sequences are constructed by substituting, inserting or deleting values in the `varAllele` column into the reference sequence. Amino acid codes are computed for the variant codon sequence when the length is a multiple of 3.

The `query` argument to `predictCoding` can be a `GRanges` or `VCF`. When a `GRanges` is supplied the `varAllele` argument must be specified. In the case of a `VCF` object, the alternate alleles are taken from `alt(<VCF>)` and the `varAllele` argument is not specified.

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

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

GRanges with 5 ranges and 13 metadata columns:

	seqnames	ranges	strand	paramRangeID
	<Rle>	<IRanges>	<Rle>	<factor>
22:50301584	chr22	[50301584, 50301584]	-	<NA>
rs114264124	chr22	[50302962, 50302962]	-	<NA>
rs149209714	chr22	[50302995, 50302995]	-	<NA>
22:50303554	chr22	[50303554, 50303554]	-	<NA>
rs12167668	chr22	[50303561, 50303561]	-	<NA>

	varAllele	CDSLOC	PROTEINLOC	QUERYID
	<DNAStringSet>	<IRanges>	<CompressedIntegerList>	<integer>
22:50301584	A	[777, 777]	259	28
rs114264124	A	[698, 698]	233	57
rs149209714	C	[665, 665]	222	58
22:50303554	G	[652, 652]	218	73
rs12167668	A	[645, 645]	215	74

	TXID	CDSID	GENEID	CONSEQUENCE	REFCODON
	<character>	<integer>	<character>	<factor>	<DNAStringSet>
22:50301584	73482	217009	79087	synonymous	CCG
rs114264124	73482	217010	79087	nonsynonymous	CGG
rs149209714	73482	217010	79087	nonsynonymous	GGA
22:50303554	73482	217011	79087	nonsynonymous	ATC
rs12167668	73482	217011	79087	synonymous	CCG

	VARCODON	REFAA	VARAA
	<DNAStringSet>	<AAStringSet>	<AAStringSet>
22:50301584	CCA	P	P
rs114264124	CAG	R	Q
rs149209714	GCA	G	A
22:50303554	GTC	I	V
rs12167668	CCA	P	P

---

```
seqlengths:
chr22
NA
```

Using variant rs114264124 as an example, we see `varAllele` A has been substituted into the `refCodon` CGG to produce `varCodon` CAG. The `refCodon` is the sequence of codons necessary to make the variant allele substitution and therefore often includes more nucleotides than indicated in the range (i.e. the range is 50302962, 50302962, width of 1). Notice it is the second position in the `refCodon` that has been substituted. This position in the codon, the position of substitution, corresponds to genomic position 50302962. This genomic position maps to position 698 in coding region-based coordinates and to triplet 233 in the protein. This is a non-synonymous coding variant where the amino acid has changed from R (Arg) to Q (Gln).

When the resulting `varCodon` is not a multiple of 3 it cannot be translated. The consequence is considered a `frameshift` and `varAA` will be missing.

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

GRanges with 1 range and 13 metadata columns:

	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> <DNAStrngSet>
22:50317001      72592      214765      79174 frameshift      GCC
              VARCHODON      REFAA      VARAA
              <DNAStrngSet> <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 <- 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.

# Bibliography

- [1] S. Anders and W. Huber. Differential expression analysis for sequence count data. *Genome Biology*, 11:R106, 2010.
- [2] S. Anders, A. Reyes, and W. Huber. Detecting differential usage of exons from rna-seq data. *Genome Research*, 2012.
- [3] 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.
- [4] J. Bullard, E. Purdom, K. D. Hansen, and S. Dudoit. Evaluation of statistical methods for normalization and differential expression in mrna-seq experiments. *BMC Bioinformatics*, 11, 2010. R package version 1.12.0.
- [5] J. Cairns, C. Spyrou, R. Stark, M. L. Smith, A. G. Lynch, and S. Tavaré. Bayespeak - an r package for analysing chip-seq data, bioinformatics. *Bioinformatics*, 27(5):714–714, 2011.
- [6] J. M. Chambers. *Software for Data Analysis: Programming with R*. Springer, New York, 2008.
- [7] P. Dalgaard. *Introductory Statistics with R*. Springer, 2nd edition, 2008.
- [8] N. Delhomme, I. Padiou, E. E. Furlong, and L. M. Steinmetz. easyrnaseq: a bioconductor package for processing rna-seq data. *Bioinformatics*, in press:in press, 2012.
- [9] O. Flores and M. Orozco. nucler: a package for non-parametric nucleosome positioning. *Bioinformatics*, 27:2149–2150, 2011.
- [10] R. Gentleman. *R Programming for Bioinformatics*. Computer Science & Data Analysis. Chapman & Hall/CRC, Boca Raton, FL, 2008.
- [11] R. C. Gentleman et al. Bioconductor: open software development for computational biology and bioinformatics. *Genome Biology* 2010 11:202, 5(10):R80, Jan 2004.
- [12] Glaus, Peter, Honkela, Antti, Rattray, and Magnus. Identifying differentially expressed transcripts from rna-seq data with biological variation. *Bioinformatics*, 28(13):1721–1728, 2012.
- [13] M. G. Grabherr, B. J. Haas, M. Yassour, J. Z. Levin, D. A. Thompson, I. Amit, X. Adiconis, L. Fan, R. Raychowdhury, Q. Zeng, Z. Chen, E. Mauceli, N. Hacohen, A. Gnirke, N. Rhind, F. D. Palma, B. W. Birren, C. Nusbaum, K. Lindblad-Toh, N. Friedman, and A. Regev. Full-length transcriptome assembly from rna-seq data without a reference genome. *Nat Biotechnol*, 29(7):644–652, May 2011.
- [14] A. Gusnanto, H. M. Wood, Y. Pawitan, P. Rabbitts, and S. Berri. Correcting for cancer genome size and tumour cell content enables better estimation of copy number alterations from next-generation sequence data. *Bioinformatics*, 28(1):40–7, Jan 2012.
- [15] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, XXIX(2):1–14, Nov 1950.
- [16] K. D. Hansen, R. A. Irizarry, and Z. Wu. Removing technical variability in RNA-seq data using conditional quantile normalization. *Biostatistics*, 13(2):204–216, 2012.

- [17] I. Holmes, K. Harris, and C. Quince. Dirichlet multinomial mixtures: Generative models for microbial metagenomics. *PLoS ONE*, 7(2):e30126, 02 2012.
- [18] D. C. Jones, W. L. Ruzzo, X. Peng, and M. G. Katze. A new approach to bias correction in rna-seq. *Bioinformatics*, 28:921–928, 2012.
- [19] R. Kabacoff. *R in Action*. Manning, 2010.
- [20] G. Klambauer, K. Schwarzbauer, A. Mayr, A. Mitterecker, D.-A. Clevert, U. Bodenhofer, and S. Hochreiter. cn.mops: Mixture of poisson for discovering copy number variations in next generation sequencing data with a low false discovery rate. *Nucleic Acids Research*, 40:e69, 2012.
- [21] H.-U. Klein, C. Bartenhagen, A. Kohlmann, V. Grossmann, C. Ruckert, T. Haerlach, and M. Dugas. R453plus1toolbox: an r/bioconductor package for analyzing roche 454 sequencing data. *Bioinformatics*, 27(8):1162–1163, 2011.
- [22] 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.
- [23] P. Lefrançois, G. M. Euskirchen, R. K. Auerbach, J. Rozowsky, T. Gibson, C. M. Yellman, M. Gerstein, and M. Snyder. Efficient yeast chip-seq using multiplex short-read dna sequencing. *BMC genomics*, 10(1):37, Jan 2009.
- [24] A. Leśniewska and M. J. Okoniewski. rnaseqmap: a bioconductor package for rna sequencing data exploration. *BMC Bioinformatics*, 12:200, Jan 2011.
- [25] B. Li, V. Ruotti, R. M. Stewart, J. A. Thomson, and C. N. Dewey. Rna-seq gene expression estimation with read mapping uncertainty. *Bioinformatics*, 26(4):493–500, Feb 2010.
- [26] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25:1754–1760, Jul 2009.
- [27] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26:589–595, Mar 2010.
- [28] Luo, Weijun, Friedman, Michael, Shedden, Kerby, Hankenson, Kurt, Woolf, and Peter. Gage: generally applicable gene set enrichment for pathway analysis. *BMC Bioinformatics*, 10:161, 2009.
- [29] E. L. M. Hummel, S. Bonnin and G. Roma. Teqc: an r-package for quality control in target capture experiments. *Bioinformatics*, 2011.
- [30] N. Matloff. *The Art of R Programming*. No Starch Press, 2011.
- [31] M. Morgan, S. Anders, M. Lawrence, P. Aboyoun, H. Pagès, and R. Gentleman. Shortread: a bioconductor package for input, quality assessment and exploration of high-throughput sequence data. *Bioinformatics*, 25:2607–2608, 2009.
- [32] A. Mortazavi et al. Mapping and quantifying mammalian transcriptomes by rna-seq. *Nature Methods*, 5(7):621–8, Jul 2008.
- [33] J. Muino, K. Kaufmann, R. van Ham, G. Angenent, and P. Krajewski. Chip-seq analysis in r (csar): An r package for the statistical detection of protein-bound genomic regions. *Plant Methods*, 7(1), 2011.
- [34] Pacific Symposium on Biocomputing. *phyloseq: A Bioconductor Package for Handling and Analysis of High-Throughput Phylogenetic Sequence Data*, volume 17, 2011.
- [35] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2009. ISBN 3-900051-07-0.
- [36] D. Risso, K. Schwartz, G. Sherlock, and S. Dudoit. GC-Content Normalization for RNA-Seq Data. *BMC Bioinformatics*, 12(1):480, 2011.

- [37] 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.
- [38] C. S. Ross-Innes et al. Differential oestrogen receptor binding is associated with clinical outcome in breast cancer. *Nature*, 481:389–393, 2012.
- [39] M. H. Schulz, D. R. Zerbino, M. Vingron, and E. Birney. Oases: robust de novo rna-seq assembly across the dynamic range of expression levels. *Bioinformatics*, 28(8):1086–92, Apr 2012.
- [40] N. Servant, B. R. Lajoie, E. P. Nora, L. Giorgetti, C.-J. Chen, E. Heard, J. Dekker, and E. Barillot. Hic: Exploration of high-throughput 'c' experiments. *Bioinformatics*, 2012.
- [41] J. Toedling, C. Ciaudo, O. Voinnet, E. Heard, and E. Barillot. girafe - an R/Bioconductor package for functional exploration of aligned next-generation sequencing reads. *Bioinformatics*, 26:2902–2903, 2010.
- [42] C. Trapnell, B. A. Williams, G. Pertea, A. Mortazavi, G. Kwan, M. J. van Baren, S. L. Salzberg, B. J. Wold, and L. Pachter. Transcript assembly and quantification by rna-seq reveals unannotated transcripts and isoform switching during cell differentiation. *Nat Biotechnol*, 28(5):511–5, May 2010.
- [43] E. Turro, S.-Y. Su, Â. Gonçalves, L. J. M. Coin, S. Richardson, and A. Lewin. Haplotype and isoform specific expression estimation using multi-mapping rna-seq reads. *Genome Biol*, 12(2):R13, Jan 2011.
- [44] T. Yin, D. Cook, and M. Lawrence. ggbio: an r package for extending the grammar of graphics for genomic data. *Genome Biology*, 13(8):R77, 2012.
- [45] 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.
- [46] X. Zhang, G. Robertson, M. Krzywinski, K. Ning, A. Droit, S. Jones, and R. Gottardo. Pics: Probabilistic inference for chip-seq. *Biometrics*, 66, 2010.