Sequences, Genomes, and Genes in *R* / *Bioconductor*

Martin Morgan

October 21, 2013

Contents

Chapter 1

Introduction

The first part of today's activities provide an introduction to high-throughput sequence analysis, including key 'infrastructure' in *R* and *Bioconductor*. The main objectives are to arrive at a common language for discussing sequence analysis, and to become familiar with concepts in *R* and *Bioconductor* that are necessary for effective analysis and comprehension of high-throughput sequence data. An approximate schedule is in Table 1.1.

1.1 High-throughput workflows

Recent technological developments introduce high-throughput sequencing approaches. A variety of experimental protocols and analysis work flows address gene expression, regulation, encoding of genetic variants, and microbial community structure. 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.

1.1.1 Technologies

The most common 'second generation' technologies readily available to labs are

- Illumina single- and paired-end reads. Short ($100 - 150$ per end) and very numerous. Flow cell, lane, bar-code.
- Roche 454. 100's of nucleotides, 100,000's of reads.
- Life Technologies SOLiD. Unique 'color space' model.
- Complete Genomics. Whole genome sequence / variants / etc as a service; end user gets derived results.

Figure 1.1 illustrates Illumina and 454 sequencing. *Bioconductor* has good support for Illumina and derived data such as aligned reads or called variants, and some support for Roche 454 sequencing; use of SOLiD color space reads typically requires conversion to FASTQ files that undermine the benefit of the color space model.

Table 1.1: Partial agenda, Day 1.

Time	Topic
8:30	...
09:15	Introduction to high-throughput workflows, with R examples
10:15	Tea / coffee break
10:30	Sequences Long and Short
11:30	Genomes and Genes
12:30	Lunch
13:30	Genomes and Genes (continued)
14:00	...

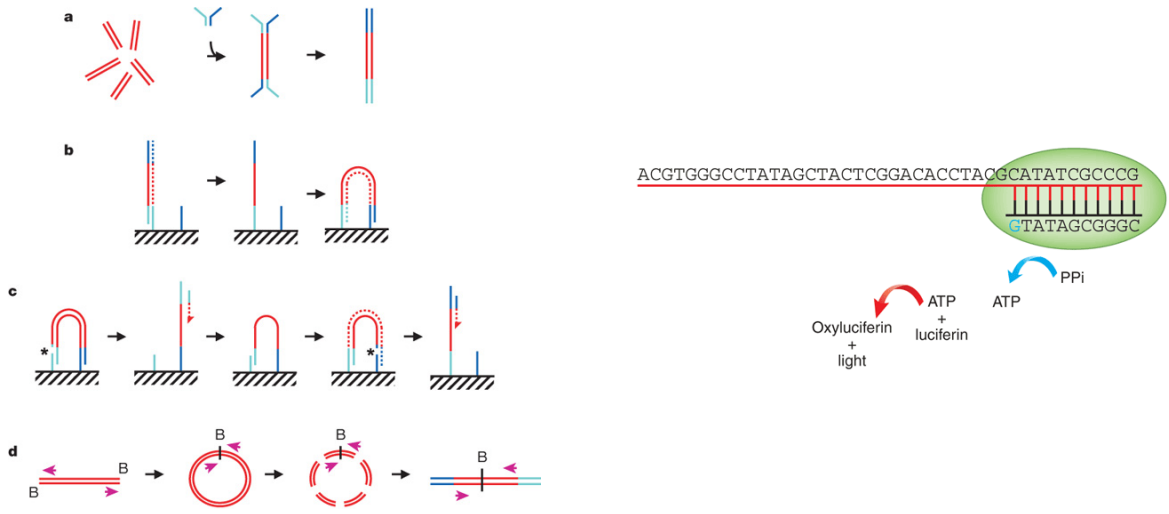

Figure 1.1: High-throughput sequencing. Left: Illumina bridge PCR [2]; mis-call errors. Right: Roche 454 [15]; homopolymer errors.

All second-generation technologies rely on PCR and other techniques to generate reads from samples that represent aggregations of many DNA molecules. 'Third-generation' technologies shift to single-molecule sequencing, with relevant players including Pacific Biosciences and IonTorent. This very exciting data (e.g., [16]) will not be discussed further.

1.1.2 Research questions

Sequence data can be derived from a tremendous diversity of research questions. Some of the most common include:

Variation DNA-Seq. Sequencing of whole or targeted (e.g., exome) genomic DNA. Common goals include *SNP* detection, *indel* and other large-scale structural polymorphisms, and *CNV* (copy number variation). DNA-seq is also used for *de novo* assembly, but *de novo* assembly is not an area where *Bioconductor* contributes. Key reference: [7, 11].

Expression RNA-seq. Sequencing of reverse-complemented mRNA from the entire expressed transcriptome, typically. Used for *differential expression* studies like micro-arrays, or for *novel transcript discovery*.

Regulation ChIP-seq. ChIP (chromatin immuno-precipitation) is used to enrich genomic DNA for regulatory elements, followed by sequencing and mapping of the enriched DNA to a reference genome. The initial statistical challenge is to identify regions where the mapped reads are enriched relative to a sample that did not undergo ChIP[12]; a subsequent task is to identify differential binding across a designed experiment, e.g., [14]. Survey of diversity of methods: [6].

Metagenomics Sequencing generates sequences from samples containing multiple species, typically microbial communities sampled from niches such as the human oral cavity. Goals include inference of *species composition* (when sequencing typically targets phylogenetically informative genes such as 16S) or *metabolic contribution*. [10, 5]

Special challenges Non-model organisms. Small budgets.

1.1.3 Analysis

Work flows RNA-seq to measure gene expression through assessment of mRNA abundance represents major steps in a typical high-throughput sequence work flow. Typical steps include;

1. Experimental design.
2. Wet-lab protocols for mRNA extraction and reverse transcription to cDNA.
3. Sequencing; QA.
4. Alignment of sequenced reads to a reference genome; QA.
5. Summarizing of the number of reads aligning to a region; QA.
6. Normalization of samples to accommodate purely technical differences in preparation.

Table 1.2: Common file types and *Bioconductor* packages used for input.

File	Description	Package
FASTQ	Unaligned sequences: identifier, sequence, and encoded quality score tuples	*ShortRead*
BAM	Aligned sequences: identifier, sequence, reference sequence name, strand position, cigar and additional tags	*Rsamtools*
VCF	Called single nucleotide, indel, copy number, and structural variants, often compressed and indexed (with *Rsamtools* `bgzip`, `indexTabix`)	*VariantAnnotation*
GFF, GTF	Gene annotations: reference sequence name, data source, feature type, start and end positions, strand, etc.	*rtracklayer*
BED	Range-based annotation: reference sequence name, start, end coordinates.	*rtracklayer*
WIG, bigWig	'Continuous' single-nucleotide annotation.	*rtracklayer*
2bit	Compressed FASTA files with 'masks'	

7. Statistical assessment, including specification of an appropriate error model.
8. Interpretation of results in the context of original biological questions; QA.

The central inference is that higher levels of gene expression translate to more abundant cDNA, and greater numbers of reads aligned to the reference genome.

Common file formats The 'big data' component of high-throughput sequence analyses seems to be a tangle of transformations between file types; common files are summarized in Table 1.2. FASTQ and BAM (sometimes CRAM) files are the primary formats for representing raw sequences and their alignments. VCF are used to summarize called variants in DNA-seq; BED and sometimes WIG files are used to represent ChIP and other regulatory peaks and 'coverage'. GTF / GFF files are important for providing feature annotations, e.g., of exons organization into transcripts and genes.

Third-party (non-R) tools Common analyses often use well-established third-party tools for initial stages of the analysis; some of these have *Bioconductor* counterparts that are particularly useful when the question under investigation does not meet the assumptions of other facilities. Some common work flows (a more comprehensive list is available on the SeqAnswers wiki[1]) include:

DNA-seq especially variant calling can be facilitated by software such as the GATK[2] toolkit.

RNA-seq In addition to the aligners mentioned above, RNA-seq for differential expression might use the *HTSeq*[3] python tools for counting reads within regions of interest (e.g., known genes) or a pipeline such as the *bowtie* (basic alignment) / *tophat* (splice junction mapper) / *cufflinks* (esimated isoform abundance) (e.g., [4]) or *RSEM*[5] suite of tools for estimating transcript abundance.

ChIP-seq ChIP-seq experiments typically use DNA sequencing to identify regions of genomic DNA enriched in prepared samples relative to controls. A central task is thus to identify peaks, with common tools including *MACS* and *PeakRanger*.

Programs such as those outlined the previous paragraph often rely on information about gene or other structure as input, or produce information about chromosomal locations of interesting features. The GTF and BED file formats are common representations of this information. Representing these files as *R* data structures is often facilitated by the *rtracklayer* package. We explore these files in Chapter 3.2.3. Variants are very commonly represented in VCF (Variant Call Format) files; these are explored in Chapter 3.2.4.

Common statistical issues Important statistical issues are summarized in Table 1.3. These will be discussed further in later parts of the course, but similar types of issues are relevant in all high-throughput sequence work flows. An important general point is that wet-lab protocols, sequencing reactions, and alignment or other technological processing

[1] `http://seqanswers.com/wiki/RNA-Seq`
[2] `http://www.broadinstitute.org/gatk/`
[3] `http://www-huber.embl.de/users/anders/HTSeq/doc/overview.html`
[4] `http://bowtie-bio.sourceforge.net/index.shtml`
[5] `http://deweylab.biostat.wisc.edu/rsem/`

Table 1.3: Common statistical issues in RNA-seq differential expression and other high-throughput experiments.

Analysis stage	Issues
Experimental design	Technical versus biological replication, sample size, complexity of design, feasibility of intended analysis
Batch effects	Known and unknown factors; technical artifacts.
Summary	Data reduction without loss of information, e.g., counts versus RPKM.
Normalization	Robust estimates of library size.
Differential expression	Appropriate error model (Negative Binomial, Poisson, ...); 'shrinkage' to balance accuracy of per-gene estimates with precision of experiment-wide estimates.
Testing	Filtering to reduce multiple comparisons & false discovery rate.

steps introduce artifacts that need to be acknowledged and, if possible, accommodated in down-stream analysis, e.g., through modeling or remediation of *batch effects*.

1.2 R and Bioconductor

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.

A first session 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`

Table 1.4: Essential aspects of the *R* language.

Category	Function	Description
Vectors	`integer`, `numeric`	Vectors of length $>= 0$ holding a single data type
	`complex`, `character`	
	`raw`, `factor`	
Statistical	`NA`, `factor`	Essential statistical concepts, integral to the language.
List-like	`list`	Arbitrary collections of elements
	`data.frame`	List of equal-length vectors
	`environment`	*Pass-by-reference* data storage; hash
Array-like	`data.frame`	Homogeneous columns; row- and column indexing
	`array`	0 or more dimensions
	`matrix`	Two-dimensional, homogeneous types
Classes	'S3'	List-like structured data; simple inheritance & dispatch
	'S4'	Formal classes, multiple inheritance & dispatch
Functions	'function'	A simple function with arguments, body, and return value
	'generic'	A (S3 or S4) function with associated *methods*
	'method'	A function implementing a generic for an S3 or S4 class

```
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] 1.6094 1.3863 1.0986 0.6931 0.0000
```

1.2.1 Essential R

Built-in (atomic) data types *R* has a number of built-in data types, summarized in Table 1.4. These 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"

class(x)                  # the number 5 is numeric

## [1] "numeric"
```

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 -- class isomorphism
```

```
## $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.

```
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
```

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[, 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
##      0.0104       0.9847

anova(fit)

## Analysis of Variance Table
##
## Response: y
##            Df Sum Sq Mean Sq F value Pr(>F)
## x           1   1023    1023    4141 <2e-16 ***
## Residuals 998    247       0
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

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

## [1] 0.4969

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 conceptually 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. A quick guide to using S4 methods is in Table 1.5.

Functions *R* has a very large number of functions; Table 1.6 provides a brief list of those that might be commonly used and particularly useful. See the help pages (e.g., `?lm`) and examples (`example(match)`) for more details on each of these 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] 1.6094 1.3863 1.0986 0.6931 0.0000

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] 2.322 2.000 1.585 1.000 0.000

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

Table 1.5: Using S4 classes and methods.

Best practices	
`gr <- GRanges()`	'Constructor'; create an instance of the *GRanges* class
`seqnames(gr)`	'Accessor', extract information from an instance
`countOverlaps(gr1, gr2)`	A *method* implementing a *generic* with useful functionality
Older packages	
`s <- new("MutliSet)`	A constructor
`s@annotation`	A 'slot' accessor
Help	
`class(gr)`	Discover class of instance
`getClass(gr)`	Display class structure, e.g., inheritance
`showMethods(findOverlaps)`	Classes for which methods of `findOverlaps` are implemented
`showMethods(class="GRanges", where=search())`	Generics with methods implemented for the *GRanges* class, limited to currently loaded packages.
`class?GRanges`	Documentation for the *GRanges* class.
`method?"findOverlaps,GRanges,GRanges"`	Documentation for the `findOverlaps` method when the two arguments are both *GRanges* instances.
`selectMethod(findOverlaps, c("GRanges", "GRanges"))`	View source code for the method, including method 'dispatch'

Table 1.6: A selection of *R* function.

`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`); χ^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`, `unlist` Split one vector by an equal length factor, cut a single vector into intervals encoded as levels of a factor, unlist (concatenate) list elements.

`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.

`biocLite`, `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.

Table 1.7: Selected base and contributed packages.

Package	Description
base	Data input and manipulation; scripting and programming.
stats	Essential statistical and plotting functions.
lattice, *ggplot2*	Approaches to advanced graphics.
methods	'S4' classes and methods.
parallel	Facilities for parallel evaluation.
Matrix	Diverse matrix representations
data.table	Efficient management of large data tables

```
## 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] 2.322 2.000 1.585 1.000 0.000
```

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 passes these arguments to the method it dispatches to.

Packages Packages provide functionality beyond that available in base *R*. There are over 4000 packages in CRAN (Comprehensive *R* Archive Network) and 749 *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 1.7 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. New packages (from *Bioconductor* or CRAN) can be added to an *R* installation using `biocLite()`:

```
source("http://bioconductor.org/biocLite.R")
biocLite(c("GenomicRanges", "ShortRead"))
```

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.

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

```
## [1] 37
```

```
head(search(), 3)
```

```
## [1] ".GlobalEnv"          "package:EMBO2013"    "package:rtracklayer"
```

It is possible for a package earlier in the search path to mask symbols later in the search path; these can be disambiguated using `::`.

```
pi <- 3.2    ## http://en.wikipedia.org/wiki/Indiana_Pi_Bill
base::pi
```

```
## [1] 3.142
```

```
rm(pi)       ## remove from the .GlobalEnv
```

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 provided 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
?anova.lm
```

S3 and S4 methods can be queried interactively. For S3,

```
methods(anova)
```

```
## [1] anova.glm     anova.glmlist anova.lm      anova.loess*  anova.mlm     anova.nls*
##
##    Non-visible functions are asterisked
```

```
methods(class="glm")
```

```
##  [1] add1.glm*           anova.glm           confint.glm*        cooks.distance.glm*
##  [5] deviance.glm*       drop1.glm*          effects.glm*        extractAIC.glm*
##  [9] family.glm*         formula.glm*        influence.glm*      logLik.glm*
## [13] model.frame.glm     nobs.glm*           predict.glm         print.glm
## [17] residuals.glm       rstandard.glm       rstudent.glm        summary.glm
## [21] vcov.glm*           weights.glm*
##
##    Non-visible functions are asterisked
```

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

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

For instance, the source code of a function is printed if the function is invoked without parentheses. Here we discover that the function `head` (which returns the first 6 elements of anything) defined in the *utils* package, is an S3 generic

(indicated by `UseMethod`) and has several methods. We use `head` to look at the first six lines of the `head` method specialized for `matrix` objects.

```
utils::head

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

methods(head)

## [1] head.data.frame* head.default*    head.ftable*     head.function*   head.matrix
## [6] head.table*      head.Vector
##
##    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))
```

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

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

to see a list of vignettes available in the *GenomicRanges* package; add the short name of the vignette to view in your web browser. 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. For *Bioconductor* packages, vignettes are available on the package 'landing page', e.g., for *GenomicRanges*.

Scripts Many users implement analyses as scripts that load packages and input data (including massaging raw data into formats that are conducive to down-stream analysis), and then perform one or several statistical analyses to generate output in the form of summary tables or figures. *R* scripts are plain text files, so easily shared.

Experienced users rapidly migrate to several 'best practices' for managing their scripts. (1) *R* has the notion of a *vignette* that integrates a textual description with the actual analysis code, a form of literate programming. Simple and elegant vignettes can be constructed using markdown and the *knitr* package (*RStudio* has great integration of these technologies); more elaborate vignettes are based on LaTeX `Sweave` documents. (2) *Version control*, including *git* running on your own computer or in *github* is really amazing, with a relatively moderate speed-bump to get going. Version control allows one to 'check in' a current working version of a script and data, and proceeed with modifications without worrying about arbitrary file naming conventions or corrupting a previously working version. (3) It is surprisingly easy to *create an R package* that coordinates scripts, specialized and potentially re-usable functions, data, and vignettes into an easy-to-share (with lab mates or more broadly) package.

1.2.2 Bioconductor for high-throughput analysis

Bioconductor is a collection of *R* packages for the analysis and comprehension of high-throughput genomic data. *Bioconductor* started more than 10 years ago, and is widely used (Figure 1.2). It gained credibility for its statistically rigorous approach to microarray pre-processing and analysis of designed experiments, and integrative and reproducible approaches

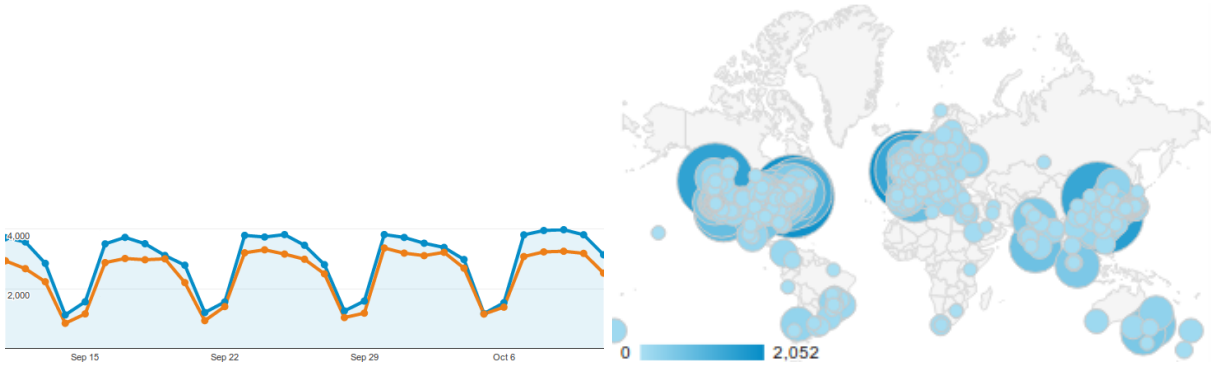

Figure 1.2: Bioconductor use, September-October 2012 (orange) and 2013 (blue).

to bioinformatic tasks. There are now 749 *Bioconductor* packages for expression and other microarrays, sequence analysis, flow cytometry, imaging, and other domains. The *Bioconductor* web site provides installation, package repository, help, and other documentation.

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

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

Table 1.8 enumerates some 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*), variants (e.g., *VariantAnnotation*, *VariantTools*), and SNPs and copy number variation (e.g., *genoset*, *ggtools*). [1] illustrate integration of *Bioconductor* packages into a typical high-throughput work flow, in this case for RNA-seq analysis.

1.2.3 Strategies for working with large data

Bioinformatics data is now very large; it is not reasonable to expect all of a FASTQ or BAM file, for instance, to fit into memory. How is this data to be processed? This challenge confronts us in whatever language or tool we are using. In *R* and *Bioconductor*, the main approaches are to: (1) write *efficient R* code; (2) *restrict* data input to an interesting subset of the larger data set; (3) *sample* from the large data, knowing that an appropriately sized sample will accurately estimate statistics we are interested in; (4) *iterate* through large data in chunks; and (5) use *parallel* evaluation on one or several computers.

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. Several approaches to achieving efficient *R* programming are summarized in Table 1.9; common tools used to help with assessing performance (including comparison of results from different implementations!) are in Table 1.10. Several common performance bottlenecks often have easy or moderate solutions; the most common of these are highlighted here.

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)
```

Table 1.8: Selected *Bioconductor* packages for high-throughput sequence analysis.

Concept	Packages
Data representation	*IRanges*, *GenomicRanges*, *Biostrings*, *BSgenome*, *VariantAnnotation*.
Input / output	*ShortRead* (FASTQ), *Rsamtools* (BAM), *rtracklayer* (GFF, WIG, BED), *VariantAnnotation* (VCF).
Annotation	*AnnotationHub*, *biomaRt*, *GenomicFeatures*, *TxDb.**, *org.**, *ChIPpeakAnno*, *VariantAnnotation*.
Alignment	*gmapR*, *Rsubread*, *Biostrings*.
Visualization	*ggbio*, *Gviz*.
Quality assessment	*qrqc*, *seqbias*, *ReQON*, *htSeqTools*, *TEQC*, *ShortRead*.
RNA-seq	*BitSeq*, *cqn*, *cummeRbund*, *DESeq2*, *DEXSeq*, *EDASeq*, *edgeR*, *gage*, *goseq*, *iASeq*, *tweeDEseq*.
ChIP-seq, etc.	*BayesPeak*, *baySeq*, *ChIPpeakAnno*, *chipseq*, *ChIPseqR*, *ChIPsim*, *CSAR*, *DiffBind*, *MEDIPS*, *mosaics*, *NarrowPeaks*, *nucleR*, *PICS*, *PING*, *REDseq*, *Repitools*, *TSSi*.
Variants	*VariantAnnotation*, *VariantTools*, *gmapR*
SNPs	*snpStats*, *GWASTools*, *SeqVarTools*, *hapFabia*, *GGtools*
Copy number	*cn.mops*, *genoset*, *fastseq*, *CNAnorm*, *exomeCopy*, *seqmentSeq*.
Motifs	*MotifDb*, *BCRANK*, *cosmo*, *MotIV*, *seqLogo*, *rGADEM*.
3C, etc.	*HiTC*, *r3Cseq*.
Microbiome	*phyloseq*, *DirichletMultinomial*, *clstutils*, *manta*, *mcaGUI*.
Work flows	*QuasR*, *ReportingTools*, *easyRNASeq*, *ArrayExpressHTS*, *oneChannelGUI*.
Database	*SRAdb*, *GEOquery*.

Table 1.9: Common ways to improve efficiency of *R* code.

Easy	Moderate
1. Selective input	1. Know relevant packages
2. Vectorize	2. Understand algorithm complexity
3. Pre-allocate and fill	3. Use parallel evaluation
4. Avoid expensive conveniences	4. Exploit libraries and `C++` code

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 during loop iteration.

```
result <- numeric(nrow(df))          ## pre-allocate
for (i in seq_len(nrow(df)))
  result[[i]] <- some_calc(df[i,])   ## fill
```

Table 1.10: Tools for measuring performance.

Function	Description
`identical`, `all.equal`	Compare content of objects.
`system.time`	Time required to evaluate an expression
`Rprof`	Time spent in each function; also `summaryRprof`.
`tracemem`	Indicate when memory copies occur (*R* must be configured to support this).
microbenchmark	Packages for standardizing speed measurment

Failure to pre-allocate and fill is the second cirle of *R* hell [4].

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

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

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

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

Restriction Just because a data file contains a lot of data does not mean that we are interested in all of it. In base *R*, one might use the `colClasses` argument to `read.delim` or similar function (e.g., setting some elements to `NULL`) to read only some columns of a large comma-separated value file.

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

In addition to the obvious benefit of using less memory than if all of the file had been read in, input will be substantially faster because less computation needs to be done to coerce values from their representation in the file to their representation in *R*'s memory.

A variation on the idea of restricting data input is to organize the data on disk into a representation that facilitates restriction. In *R*, large data might be stored in a relational data base like the *sqlite* data bases made available by the *RSQLite* package and used in the *AnnotationDbi Bioconductor* packages. In addition to facilitating restriction, these approaches are typically faster than parsing a plain text file, because the data base software has stored data in a way that efficiently transforms from on-disk to in-memory representation.

Restriction is such a useful concept that many *Bioconductor* high throughput sequence analysis functions enable doing the right thing. Functions such as `coverage,BamFile-method` or `readGAlignments` use restrictions to read in the specific data required for them to compute the statistic of interest. Most "higher level" functions have a `param=ScanBamParam()` argument to allow the user to specify additional fields if desired.

Sampling *R* is after all a statistical language, and it sometimes makes sense to draw inferences from a sample of large data. For instance, many quality assessment statistics summarize overall properties (e.g., GC content or per-nucleotide base quality of FASTQ reads) that don't require processing of the entire data. For these statistics to be valid, the sample from the file needs to be a random sample, rather than a sample of convenience.

There are two advantages to sampling from a FASTQ (or BAM) file. The sample uses less memory than the full data. And, because less data needs to be parsed from the on-disk to in-memory representation, the input is faster.

Iteration Restriction may not be enough to wrestle large data down to size, and sampling may be inappropriate for the task at hand. A solution is then to iterate through the file. An example in base *R* is to open a file connection, and then read and process successive chunks of the file, e.g., reading chunks of 10000 lines

```
## NOT RUN
con <- file("<hypothetical-file>.txt")
open(f)
repeat {
    x <- readLines(f, n=10000)          # or other input function
    if (length(x) == 0)
        break
    ## work on character vector 'x'
}
close(f)
```

This paradigm extends to parsing FASTQ, BAM, VCF, and other files.

Parallel evaluation The preceding paragraphs emphasize that the starting point for analysis of large data is efficient, vectorized code. Performance differences between poorly written versus well written *R* code can easily span two orders of magnitude, whereas parallel processing can only increase throughput by an amount inversely proportional to the number of processing units (e.g., CPUs) available. The memory management techniques outlined earlier in this chapter are important in a parallel evaluation context. This is because we will typically be trying to exploit multiple processing cores on a single computer, and the cores will be competing for the same pool of shared memory. We thus want to arrange for the collection of processors to cooperate in dividing available memory between them, i.e., each processor needs to use only a fraction of total memory.

There are a number of ways in which *R* code can be made to run in parallel. The least painful and most effective will use 'multicore' functionality provided by the *parallel* package; the *parallel* package is installed by default with base *R*, and has a useful vignette [13].

Parallel evaluation on several cores of a single Linux or MacOS computer is particularly easy to achieve when the code is already vectorized. The solution on these operating systems is to use the functions `mclapply` or `pvec`. These functions allow the 'master' process to 'fork' processes for parallel evaluation on each of the cores of a single machine. The forked processes initially share memory with the master process, and only make copies when the forked process modifies a memory location ('copy on change' semantics). On Linux and MacOS, the `mclapply` function is meant to be a 'drop-in' replacement for `lapply`, but with iterations being evaluated on different cores.

The *parallel* package does not support fork-like behavior on Windows, where users need to more explicitly create a cluster of *R* workers and arrange for each to have the same data loaded into memory; similarly, parallel evaluation across computers (e.g., in a cluster) require more elaborate efforts to coordinate workers; this is typically done using `lapply`-like functions provided by the *parallel* package but specialized for simple ('snow') or more robust ('MPI') communication protocols between workers.

Data movement and *random numbers* are two important additional considerations in parallel evaluation. Moving data to and from cores to the manager can be expensive, so strategies that minimize explicit movement (e.g., passing file names data base queries rather than *R* objects read from files; reducing data on the worker before transmitting results to the manager) can be important. Random numbers need to be synchronized across cores to avoid generating the same sequences on each 'independent' computation.

Chapter 2

Sequences

2.1 Biostrings and GenomicRanges

Biostrings The *Biostrings* package provides tools for working with sequences. 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. Table 2.2 summarizes common operations.

```
library(Biostrings)
DNAString(c("ACACTTG"))

##   7-letter "DNAString" instance
## seq: ACACTTG

dna <- DNAStringSet(c("AACCAA", "GCCGTCGNM"))
dna

##   A DNAStringSet instance of length 2
##     width seq
## [1]     6 AACCAA
## [2]     9 GCCGTCGNM

alphabetFrequency(dna, baseOnly=TRUE)

##      A C G T other
## [1,] 4 2 0 0     0
## [2,] 0 3 3 1     2
```

Table 2.1: Selected *Bioconductor* packages for representing strings and reads.

Package	Description
Biostrings	Classes (e.g., *DNAStringSet*) and methods (e.g., `alphabetFrequency`, `pairwiseAlignment`) for representing and manipulating DNA and other biological sequences.
BSgenome	Representation and manipulation of large (e.g., whole-genome) sequences.
ShortRead	I/O and manipulation of FASTQ files.

Table 2.2: Operations on strings in the *Biostrings* package.

	Function	Description
Access	`length`, `names`	Number and names of sequences
	`[`, `head`, `tail`, `rev`	Subset, first, last, or reverse sequences
	`c`	Concatenate two or more objects
	`width`, `nchar`	Number of letters of each sequence
	`Views`	Light-weight sub-sequences of a sequence
Compare	`==`, `!=`, `match`, `%in%`	Element-wise comparison
	`duplicated`, `unique`	Analog to `duplicated` and `unique` on character vectors
	`sort`, `order`	Locale-independent sort, order
	`split`, `relist`	Split or relist objects to, e.g., *DNAStringSetList*
Edit	`subseq`, `subseq<-`	Extract or replace sub-sequences in a set of sequences
	`reverse`, `complement` `reverseComplement`	Reverse, complement, or reverse-complement DNA
	`translate`	Translate DNA to Amino Acid sequences
	`chartr`	Translate between letters
	`replaceLetterAt`	Replace letters at a set of positions by new letters
	`trimLRPatterns`	Trim or find flanking patterns
Count	`alphabetFrequency` `letterFrequency` `letterFrequencyInSlidingView`	Tabulate letter occurrence
	`consensusMatrix`	Nucleotide $\times$ position summary of letter counts
	`dinucleotideFrequency` `trinucleotideFrequency` `oligonucleotideFrequency`	2-mer, 3-mer, and k-mer counting
	`nucleotideFrequencyAt`	Nucleotide counts at fixed sequence positions
Match	`matchPattern`, `countPattern` `vmatchPattern`, `vcountPattern`	Short patterns in one or many (`v*`) sequences
	`matchPDict`, `countPDict` `whichPDict`, `vcountPDict` `vwhichPDict`	Short patterns in one or many (`v*`) sequences (mismatch only)
	`pairwiseAlignment`	Needleman-Wunsch, Smith-Waterman, etc. pairwise alignment
	`matchPWM`, `countPWM`	Occurrences of a position weight matrix
	`matchProbePair`	Find left or right flanking patterns
	`findPalindromes`	Palindromic regions in a sequence. Also `findComplementedPalindromes`
	`stringDist`	Levenshtein, Hamming, or pairwise alignment scores
I/0	`readDNAStringSet`	FASTA (or sequence only from FASTQ). Also `readBStringSet`, `readRNAStringSet`, `readAAStringSet`
	`writeXStringSet`	
	`writePairwiseAlignments`	Write `pairwiseAlignment` as "pair" format
	`readDNAMultipleAlignment`	Multiple alignments (FASTA, "stockholm", or "clustal"). Also `readRNAMultipleAlignment`, `readAAMultipleAlignment`
	`write.phylip`	

GenomicRanges Ranges describe both features of interest (e.g., genes, exons, promoters) and reads aligned to the genome. *Bioconductor* has very powerful facilities for working with ranges, some of which are summarized in Table 2.3. These are implemented in the *GenomicRanges* package; see [9] for a more comprehensive conceptual orientation.

The GRanges class 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

Table 2.3: Selected *Bioconductor* packages for representing and manipulating ranges, strings, and other data structures.

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

base at 18962925. The coordinates are *1-based* (i.e., the first nucleotide on a chromosome is numbered 1, rather than 0), *left-most* (i.e., reads on the minus strand are defined to 'start' at the left-most coordinate, rather than the 5' coordinate), and *closed* (the start and end coordinates are included in the range; a range with identical start and end coordinates has width 1, a 0-width range is represented by the special construct where the end coordinate is one less than the start coordinate). A complete definition of these genes as *GRanges* is:

```
genes <- GRanges(seqnames=c("chr3R", "chrX"),
                 ranges=IRanges(
                     start=c(19967117, 18962306),
                     end=c(19973212, 18962925)),
                 strand=c("+", "-"),
                 seqlengths=c(chr3R=27905053L, chrX=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]    chr3R [19967117, 19973212]      +
##   [2]     chrX [18962306, 18962925]      -
##   ---
##   seqlengths:
##       chr3R     chrX
##    27905053 22422827
```

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

```
?GRanges
```

and package vignettes

```
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]     chrX [18962306, 18962925]      -
##   ---
##   seqlengths:
##       chr3R     chrX
##    27905053 22422827

strand(genes)

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

width(genes)

## [1] 6096  620

length(genes)

## [1] 2

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

## GRanges with 2 ranges and 0 metadata columns:
##               seqnames              ranges strand
##                  <Rle>           <IRanges>  <Rle>
##   FBgn0039155    chr3R [19967117, 19973212]      +
##   FBgn0085359     chrX [18962306, 18962925]      -
##   ---
##   seqlengths:
##       chr3R     chrX
##    27905053 22422827
```

`strand` returns the strand information in a compact representation called a *run-length encoding*. 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. 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))
```

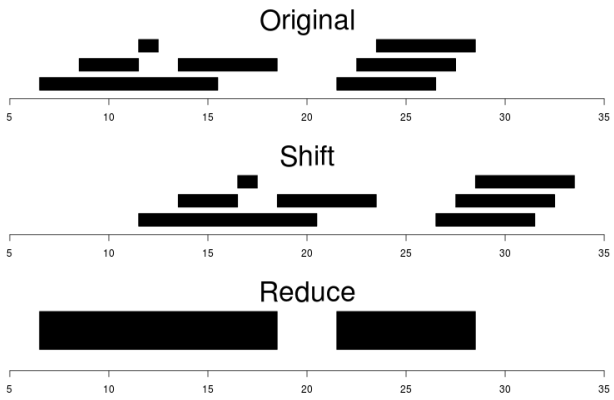

Figure 2.1: Ranges

These and some common operations are illustrated in the upper panel of Figure 2.1 and summarized in Table 2.4.

Common operations on ranges are summarized in Table 2.4. 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`,

Table 2.4: Common operations on *IRanges*, *GRanges* and *GRangesList*.

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

`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'.

`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*).

```
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())
```

The GRangesList class 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]      + |     50486        <NA>
##   [2]     chr3R [19970915, 19971592]      + |     50487        <NA>
##   [3]     chr3R [19971652, 19971770]      + |     50488        <NA>
##   [4]     chr3R [19971831, 19972024]      + |     50489        <NA>
##   [5]     chr3R [19972088, 19972461]      + |     50490        <NA>
##   [6]     chr3R [19972523, 19972589]      + |     50491        <NA>
##   [7]     chr3R [19972918, 19973212]      + |     50492        <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.

2.2 From whole genome to short read

2.2.1 Large and whole-genome sequences

There are three ways in which whole-genome sequences are represented in *Bioconductor*. For model organisms, the *BSgenome* package and suite of annotations (e.g., *BSgenome.Hsapiens.UCSC.hg19*) can be used to query genome coordinates and to load whole chromosomes into memory. The annotation packages contain optional 'masks', e.g., of repeat regions, and are explored in an exercise at the end of this section.

The *Rsamtools* package provides an interface to indexed FASTA files via the `FaFile` function; this can be used to input whole genomes or, more usefully, along with *GRanges* instances to input selected sequences. This is explored in an exercise related to the *AnnotationHub* package later today.

Finally, the *rtracklayer* package enables import of '2bit' FASTA format files.

2.2.2 Short reads

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

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

The first and third lines (beginning with `@` and + respectively) are unique identifiers. The 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 higher 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 `bigdata` folder

```
bigdata <- file.choose()
```

Each file consists of 1 million reads from a lane of the Pasilla data set.

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

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

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

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

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

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

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

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

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

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

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

```
## Class "ShortReadQ" [package "ShortRead"]
##
## Slots:
##
## Name:       quality        sread           id
## Class: QualityScore DNAStringSet   BStringSet
##
## Extends:
## Class "ShortRead", directly
## Class ".ShortReadBase", by class "ShortRead", distance 2
##
## Known Subclasses: "AlignedRead"
```

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

```
showMethods(class="ShortRead", where="package:ShortRead")
```

For instance, the `width` can be used to demonstrate that all reads are of the same width:

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

```
##
##      37
## 1000000
```

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

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

```
##         cycle
## alphabet   [,1]   [,2]   [,3]   [,4]   [,5]   [,6]   [,7]   [,8]
```

```
##        A  78194 153156 200468 230120 283083 322913 162766 220205
##        C 439302 265338 362839 251434 203787 220855 253245 287010
##        G 397671 270342 258739 356003 301640 247090 227811 246684
##        T  84833 311164 177954 162443 211490 209142 356178 246101
```

We mentioned sampling and iteration as strategies for dealing with large data. A very common reason for looking at FASTQ data is to explore sequence quality. In these circumstances it is often not necessary to parse the entire FASTQ file. Instead create a representative sample

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

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

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

Quality assessment *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

```
qa <- qa(dirname(fastqFiles), "*.fastq", type="fastq")
rpt <- report(qa, dest=tempfile())
browseURL(rpt)
```

A report from a larger subset of the experiment is available

```
rpt <- system.file("GSM461176_81_qa_report", "index.html",
                   package="EMBO2013")
browseURL(rpt)
```

2.3 Exercises

Exercise 1 *Develop a function `gcFunction` to calculate GC content; likely your function will use `alphabetFrequency`. Demonstrate its use on a DNAStringSet instance.*

Solution: Here is the `gcFunction` helper function to calculate GC content:

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

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. Here's a simple illustration:

```
gcFunction(dna)

## [1] 0.3333 0.6667
```

Exercise 2 *This exercise calculates the GC content of two genes. We make use of a BSgenome package, and DNAString and GRanges classes.*

Load the BSgenome.Dmelanogaster.UCSC.dm3 data package, containing the UCSC representation of D. melanogaster *genome assembly dm3. Discover the content of the package by evaluating* `Dmelanogaster`.

Create the `genes` *object (a GRanges instance), from a page or so ago.*

Use `getSeq` *to retrieve a DNAStringSet representing the sequence of each gene.*

Use `gcFunction` *to calculate the GC content of each gene.*

Compare this to the density of short reads, calcuated later.

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

```
library(BSgenome.Dmelanogaster.UCSC.dm3)
Dmelanogaster

## Fly genome
## |
## | organism: Drosophila melanogaster (Fly)
## | provider: UCSC
## | provider version: dm3
## | release date: Apr. 2006
## | release name: BDGP Release 5
## |
## | single sequences (see '?seqnames'):
## |   chr2L      chr2R      chr3L      chr3R      chr4       chrX       chrU       chrM
## |   chr2LHet   chr2RHet   chr3LHet   chr3RHet   chrXHet    chrYHet    chrUextra
## |
## | multiple sequences (see '?mseqnames'):
## |   upstream1000  upstream2000  upstream5000
## |
## | (use the '$' or '[[' operator to access a given sequence)

data(genes)
genes

## GRanges with 15682 ranges and 1 metadata column:
##               seqnames              ranges strand   |         gene_id
##                  <Rle>           <IRanges>  <Rle>   | <CharacterList>
##   FBgn0000003    chr3R [ 2648220,  2648518]      +   |     FBgn0000003
##   FBgn0000008    chr2R [18024494, 18060346]      +   |     FBgn0000008
##   FBgn0000014    chr3R [12632936, 12655767]      -   |     FBgn0000014
##   FBgn0000015    chr3R [12752932, 12797958]      -   |     FBgn0000015
##   FBgn0000017    chr3L [16615470, 16640982]      -   |     FBgn0000017
##           ...      ...                 ...    ... ...             ...
##   FBgn0264723    chr3L [12238610, 12239148]      -   |     FBgn0264723
##   FBgn0264724    chr3L [15327882, 15329271]      +   |     FBgn0264724
##   FBgn0264725    chr3L [12025657, 12026099]      +   |     FBgn0264725
##   FBgn0264726    chr3L [12020901, 12021253]      +   |     FBgn0264726
##   FBgn0264727    chr3L [22065469, 22065720]      +   |     FBgn0264727
##   ---
##   seqlengths:
##        chr2L     chr2R     chr3L     chr3R ...  chr3RHet   chrXHet   chrYHet chrUextra
##     23011544  21146708  24543557  27905053 ...   2517507    204112    347038  29004656

seq <- getSeq(Dmelanogaster, genes)
```

The subject GC content is

```
gc <- gcFunction(seq)
```

Exercise 3 *Use the file path `bigdata` and the `file.path` and `dir` functions to locate the fastq file from [3] (the file was obtained as described in the pasilla experiment data package).*

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` defined elsewhere in this document, 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.

Use `gcFunction` to calculate the GC content in each gene.

Solution: Discovery:

```
dir(bigdata)

## [1] "bam"   "fastq"

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

Input:

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

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")
```

(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")
```

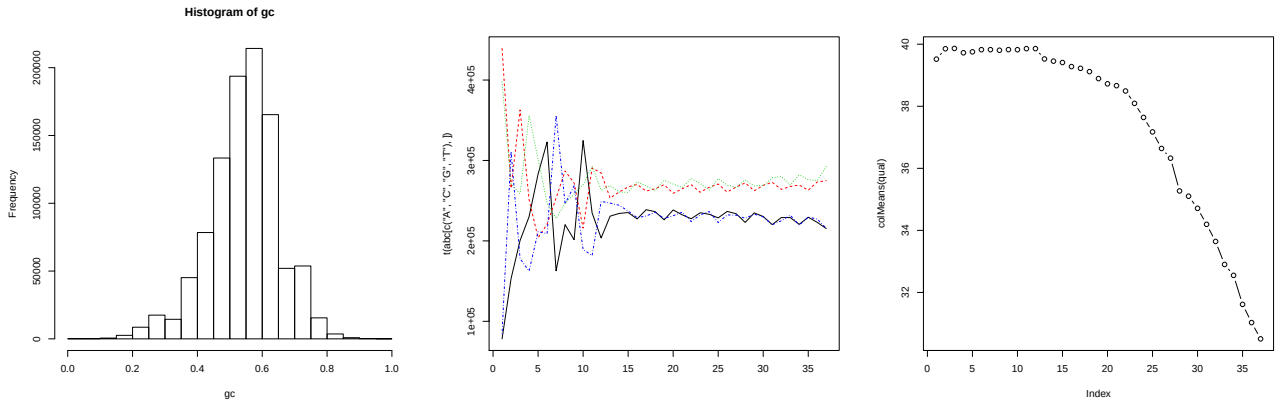

Figure 2.2: Short read GC content (left), alphabet-by-cycle (middle), and quality scores (right)

Exercise 4 *Use* `quality` *to extract the quality scores of the short reads. Interpret the encoding qualitatively.*

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

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

Solution:

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

```
## class: FastqQuality
## quality:
##   A BStringSet instance of length 6
##     width seq
## [1]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
## [2]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
## [3]    37 IIIIIIIIIIIIIIIIIIII'IIIIIGBIIII2I+
## [4]    37 IIIIIIIIIIIIIIIIIIIII,II*E,&4HI++B
## [5]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII&.$
## [6]    37 III.IIIIIIIIIIIIIIIII%IIE(-EIH<IIII
```

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

```
## [1] 1000000      37
```

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

Exercise 5 *As an independent exercise, visit the qrqc landing page and explore the package vignette. Use the qrqc package (you may need to install this) to generate base and average quality plots for the data, like those in the report generated by ShortRead.*

Chapter 3

Genes and Genomes

Bioconductor provides extensive annotation resources. 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 Gene annotation

3.1.1 Bioconductor data annotation packages

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`. Common operations for retrieving annotations are summarized in Table 3.1.

3.1.2 Internet resources

A short summary of select *Bioconductor* packages enabling web-based queries is in Table 3.2.

Table 3.1: Common operations for retrieving and manipulating annotations.

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

Table 3.2: Selected packages querying web-based annotation services.

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

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.

3.1.3 Exercises

Exercise 6 *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 `columns` method to discover which sorts of annotations can be extracted from it.*

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

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

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

```
library(org.Dm.eg.db)
columns(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, columns=cols, keytype="UNIPROT")

##   UNIPROT  SYMBOL  PATH
## 1  Q8IRZ0  CG3038  <NA>
## 2  Q95RP8  CG3038  <NA>
## 3  M9PGH7     G9a 00310
## 4  Q95RU8     G9a 00310
## 5  Q9W5H1 CG13377  <NA>
## 6  P39205     cin  <NA>
```

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

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

## Warning:  'select' resulted in 1:many mapping between keys and return rows

nrow(kegg)

## [1] 35

head(kegg, 3)

##    PATH UNIPROT  SYMBOL
## 1 00310  M9PGH7     G9a
## 2 00310  Q95RU8     G9a
## 3 00310  M9NE25 Hmt4-20
```

Exercise 7 *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.

Exercise 8 *As an exercise for later in this course, annotate the genes that are differentially expressed in Dr. Huber's lab, e.g., find the GENENAME associated with the five most differentially expressed genes. Do these make biological sense? Can you merge the annotation results with the 'top table' results to provide a statistically and biologically informative summary?*

3.2 Genome annotation

There are a diversity of packages and classes available for representing large genomes. Several include:
TxDb.* For transcript and other genome / coordinate annotation.
BSgenome For whole-genome representation. See `available.packages` for pre-packaged genomes, and the vignette 'How to forge a BSgenome data package' in the
Homo.sapiens For integrating *TxDb** and *org.** packages.
SNPlocs.* For model organism SNP locations derived from dbSNP.
`FaFile` (*Rsamtools*) for accessing indexed FASTA files.
SIFT.*, PolyPhen, ensemblVEP Variant effect scores.

3.2.1 Bioconductor transcript annotation packages

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

3.2.2 AnnotationHub

The *AnnotationHub* package makes it easier to access genome-scale resources. It consists of an *R* 'client' that queries an AnnotationHub server. There server contains lightly-curated versions of large genome resources such as UCSC or ENCODE tracks and Ensembl gtf or fasta files. There are a large number of resources available.

```
library(AnnotationHub)
hub <- AnnotationHub()
hub

## class: AnnotationHub
## length: 8674
## filters: none
## hubUrl: http://annotationhub.bioconductor.org/ah
## snapshotVersion: 2.13; snapshotDate: 2013-06-29
## hubCache: /home/mtmorgan/.AnnotationHub
```

The hub can be queried for metadata `metadata(hub)` or explored using tab completion `hub$ensembl<tab>`.

'Light curation' means that they have been transformed to data structures that are particularly easy to use from *Bioconductor*. Here we retrieve the complete sequence of *Pan troglodytes* (this is about 865M of data to download).

```
pan <-
  hub$ensembl.release.72.fasta.pan_troglodytes.dna.Pan_troglodytes.CHIMP2.1..4.72.dna.toplevel.fa.rz
```

pan is a razip-compressed file FaFile from the *Rsamtools* package. We can discover sequences the file contains

```
seqinfo(pan)
```

and retrieve arbitrary regions

```
roi <- GRanges("22", IRanges(c(30000000, 40000000), width=100000))
getSeq(pan, roi)
```

Retrievals from the AnnotationHub server are cached, so the next time access is fast It's cached, so the next time it's fast

```
system.time(
  hub$ensembl.release.72.fasta.pan_troglodytes.dna.Pan_troglodytes.CHIMP2.1.4.72.dna.toplevel.fa.rz
)
##   user  system elapsed
##  0.024   0.000   0.421
```

3.2.3 rtracklayer

rtracklayer

```
library(rtracklayer)
```

3.2.4 VariantAnnotation

A major product of DNASeq experiments are catalogs of called variants (e.g., SNPs, indels); recent scenarios use public consortium called variants to develop novel predictive filters of regulatory function [8]. 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.

Important operations on VCF files available with the *VariantAnnotation* package are summarized in Table 3.3.

Data input *VariantAnnotation* input of whole VCF files (`readVcf`), or more conveniently restriction to specific fields (e.g., `readGeno`, `readInfo`), samples or ranges (using `ScanVcfParam`). It is also straight-forward to iterate through large VCF files by using a `TabixFile` with a `yieldSize` specification, or to use `filterVcf` to transform a large VCF to a subset of variants relevant in a particular study.

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

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 3.4.

Table 3.3: Working with *VCF* files and data.

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

Table 3.4: 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

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(VariantAnnotation)
library(TxDb.Hsapiens.UCSC.hg19.knownGene)
txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
library(BSgenome.Hsapiens.UCSC.hg19)
fl <- system.file("extdata", "chr22.vcf.gz", package="VariantAnnotation")
vcf <- readVcf(fl, "hg19")
seqlevels(vcf, force=TRUE) <- c("22"="chr22")
coding <- predictCoding(vcf, txdb, seqSource=Hsapiens)
coding[5:9]

## GRanges with 5 ranges and 17 metadata columns:
##                   seqnames              ranges strand | paramRangeID            REF
##                      <Rle>           <IRanges>  <Rle> |     <factor> <DNAStringSet>
##   22:50301584_C/T    chr22 [50301584, 50301584]      - |         <NA>              C
##       rs114264124    chr22 [50302962, 50302962]      - |         <NA>              C
```

```
##       rs149209714    chr22 [50302995, 50302995]      - |         <NA>              C
##   22:50303554_T/C    chr22 [50303554, 50303554]      - |         <NA>              T
##        rs12167668    chr22 [50303561, 50303561]      - |         <NA>              C
##                                  ALT      QUAL      FILTER      varAllele     CDSLOC
##                   <DNAStringSetList> <numeric> <character> <DNAStringSet>  <IRanges>
##   22:50301584_C/T                  T       100        PASS              A [777, 777]
##       rs114264124                  T       100        PASS              A [698, 698]
##       rs149209714                  G       100        PASS              C [665, 665]
##   22:50303554_T/C                  C       100        PASS              G [652, 652]
##        rs12167668                  T       100        PASS              A [645, 645]
##                      PROTEINLOC   QUERYID        TXID     CDSID      GENEID   CONSEQUENCE
##                   <IntegerList> <integer> <character> <integer> <character>      <factor>
##   22:50301584_C/T           259        28       75253    218562       79087    synonymous
##       rs114264124           233        57       75253    218563       79087 nonsynonymous
##       rs149209714           222        58       75253    218563       79087 nonsynonymous
##   22:50303554_T/C           218        73       75253    218564       79087 nonsynonymous
##        rs12167668           215        74       75253    218564       79087    synonymous
##                         REFCODON       VARCODON         REFAA         VARAA
##                   <DNAStringSet> <DNAStringSet> <AAStringSet> <AAStringSet>
##   22:50301584_C/T            CCG            CCA             P             P
##       rs114264124            CGG            CAG             R             Q
##       rs149209714            GGA            GCA             G             A
##   22:50303554_T/C            ATC            GTC             I             V
##        rs12167668            CCG            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 2 ranges and 17 metadata columns:
##                       seqnames              ranges strand | paramRangeID            REF
##                          <Rle>           <IRanges>  <Rle> |     <factor> <DNAStringSet>
##   22:50317001_G/GCACT    chr22 [50317001, 50317001]      + |         <NA>              G
##   22:50317001_G/GCACT    chr22 [50317001, 50317001]      + |         <NA>              G
##                                    ALT      QUAL      FILTER      varAllele     CDSLOC
##                       <DNAStringSetList> <numeric> <character> <DNAStringSet>  <IRanges>
##   22:50317001_G/GCACT              GCACT       233        PASS          GCACT [808, 808]
##   22:50317001_G/GCACT              GCACT       233        PASS          GCACT [628, 628]
##                          PROTEINLOC   QUERYID        TXID     CDSID      GENEID
##                       <IntegerList> <integer> <character> <integer> <character>
##   22:50317001_G/GCACT           270       359       74357    216303       79174
##   22:50317001_G/GCACT           210       359       74358    216303       79174
```

```
##                       CONSEQUENCE       REFCODON       VARCODON          REFAA
##                          <factor> <DNAStringSet> <DNAStringSet> <AAStringSet>
##   22:50317001_G/GCACT  frameshift             GCC            GCC             A
##   22:50317001_G/GCACT  frameshift             GCC            GCC             A
##                             VARAA
##                       <AAStringSet>
##   22:50317001_G/GCACT
##   22:50317001_G/GCACT
##   ---
##   seqlengths:
##    chr22
##       NA
```

Annotation with ensemblVEP *fixme: Material for ensemblVEP*

3.2.5 Exercises

Exercise 9 *Load the 'transcript.db' package relevant to the dm3 build of* D. melanogaster. *Use* `select` *and friends to select the Flybase gene ids of the top table* `tt` *and the Flybase transcript names (TXNAME) and Entrez gene identifiers (GENEID).*

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

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

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

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

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

```
library(EMBO2013)
data(topTable)
fbids <- rownames(topTable)
```

We can discover available keys (using `keys`) and columns (`columns`) 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")
```

```
## Warning:  'select' resulted in 1:many mapping between keys and return rows
```

```
nrow(txnm)
```

```
## [1] 207
```

```
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 <- txnm[txnm$TXNAME %in% names(cds),]
cds <- cds[txnm$TXNAME]
length(cds)
```

```
## [1] 202
```

```
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]      + |     41058        <NA>         2
##   [2]     chr3R [19971652, 19971770]      + |     41059        <NA>         3
##   [3]     chr3R [19971831, 19972024]      + |     41060        <NA>         4
##   [4]     chr3R [19972088, 19972461]      + |     41061        <NA>         5
##   [5]     chr3R [19972523, 19972589]      + |     41062        <NA>         6
##   [6]     chr3R [19972918, 19973094]      + |     41063        <NA>         7
##
## ---
## seqlengths:
##      chr2L     chr2R     chr3L     chr3R ...  chr3RHet   chrXHet   chrYHet chrUextra
##   23011544  21146708  24543557  27905053 ...   2517507    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] 202
```

```
head(txx, 3)
```

```
##   A DNAStringSet instance of length 3
##     width seq                                                 names
## [1]  1578 ATGGGCAGCATGCAAGTGGCGCTGCTGG...CAAGCTGCAGATCAAGTGCAGCGACTAG FBtr0084549
## [2]  2760 ATGCTGCGTTATCTGGCGCTTTCGGAGG...GATTGTTGCTGCCCCATTCGAACTTTAG FBtr0085755
## [3]  2217 ATGGCACTCAAGTTTCCCACAGTCAAGC...GATTGTTGCTGCCCCATTCGAACTTTAG FBtr0085756
```

```
head(translate(txx), 3)
```

```
##   A AAStringSet instance of length 3
##     width seq
## [1]   526 MGSMQVALLALLVLGQLFPSAVANGSSSYSSTSTSASNQ...SSPNSVLDDSRNVFTFTTPKCENFRKRFPKLQIKCSD*
## [2]   920 MLRYLALSEAGIAKLPRPQSRCYHSEKGVWGYKPIAQRE...GQQLHYCGRCEAPTPATGIGKVHKREVDEIVAAPFEL*
## [3]   739 MALKFPTVKRYGGEGAESMLAFFWQLLRDSVQANIEHVV...GQQLHYCGRCEAPTPATGIGKVHKREVDEIVAAPFEL*
```

Exercise 10 *The objective of this exercise is to compare the quality of called SNPs that are located in dbSNP, versus those that are novel.*

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

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

Data resource often adopt different naming conventions for sequences. For instance, dbSNP uses abbreviations such as `ch22` *to represent chromosome 22, whereas our VCF file uses 22. Use* `rowData` *and* `seqlevels<-` *to extract the row data of the variants, and rename the chromosomes.*

Solution: Explore the header:

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

## class: VCFHeader
## samples(5): HG00096 HG00097 HG00099 HG00100 HG00101
## meta(1): fileformat
## fixed(1): ALT
## info(22): LDAF AVGPOST ... VT SNPSOURCE
## geno(3): GT DS GL

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

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

Input the data and peak at their locations:

```
vcf <- readVcf(fl, "hg19")
head(rowData(vcf), 3)

## GRanges with 3 ranges and 5 metadata columns:
##               seqnames              ranges strand | paramRangeID            REF
##                  <Rle>           <IRanges>  <Rle> |     <factor> <DNAStringSet>
##     rs7410291       22 [50300078, 50300078]      * |         <NA>              A
##   rs147922003       22 [50300086, 50300086]      * |         <NA>              C
##   rs114143073       22 [50300101, 50300101]      * |         <NA>              G
##                             ALT      QUAL      FILTER
##               <DNAStringSetList> <numeric> <character>
##     rs7410291                  G       100        PASS
##   rs147922003                  T       100        PASS
##   rs114143073                  A       100        PASS
##   ---
##   seqlengths:
##    22
##    NA
```

Rename chromosome levels:

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

Exercise 11 *Load the TxDb.Hsapiens.UCSC.hg19.knownGene annotation package, and read in the `chr22.vcf.gz` example file from the VariantAnnotation package.*

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

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

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

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

The next lines locate coding variants.

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

## GRanges with 3 ranges and 7 metadata columns:
##       seqnames              ranges strand | LOCATION   QUERYID      TXID     CDSID
##          <Rle>           <IRanges>  <Rle> | <factor> <integer> <integer> <integer>
##   [1]    chr22 [50301422, 50301422]      * |   coding        24     75253    218562
##   [2]    chr22 [50301476, 50301476]      * |   coding        25     75253    218562
##   [3]    chr22 [50301488, 50301488]      * |   coding        26     75253    218562
##            GENEID       PRECEDEID        FOLLOWID
##       <character> <CharacterList> <CharacterList>
##   [1]       79087
##   [2]       79087
##   [3]       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
##   965    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
```

3.3 Visualization

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

Load the *Gviz* package and sample *GRanges* containing genomic coordinates of CpG islands. Create a couple of variables with information on the chromosome and genome of the data.

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

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

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

(This track could be plotted with `plotTrack(atrack)`). Then add a track that represents genomic coordinates. Tracks are combined when plotted, as a simple list. The vertical ordering of tracks is determined by their position in the list.

```
gtrack <- GenomeAxisTrack()
```

(Plot this with `plotTracks(list(gtrack, atrack))`). We can add an ideogram to provide overall orientation...

```
itrack <- IdeogramTrack(genome=genome, chromosome=chr)
```

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

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

Plot this as

```
plotTracks(tracks)
```

Zoom out to change the location box on the ideogram

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

When zoomed in we can add sequence data

```
library(BSgenome.Hsapiens.UCSC.hg19)
strack <- SequenceTrack(Hsapiens, chromosome=chr)
```

```
plotTracks(c(tracks, strack), from=26450430, to=26450490, cex=.8)
```

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

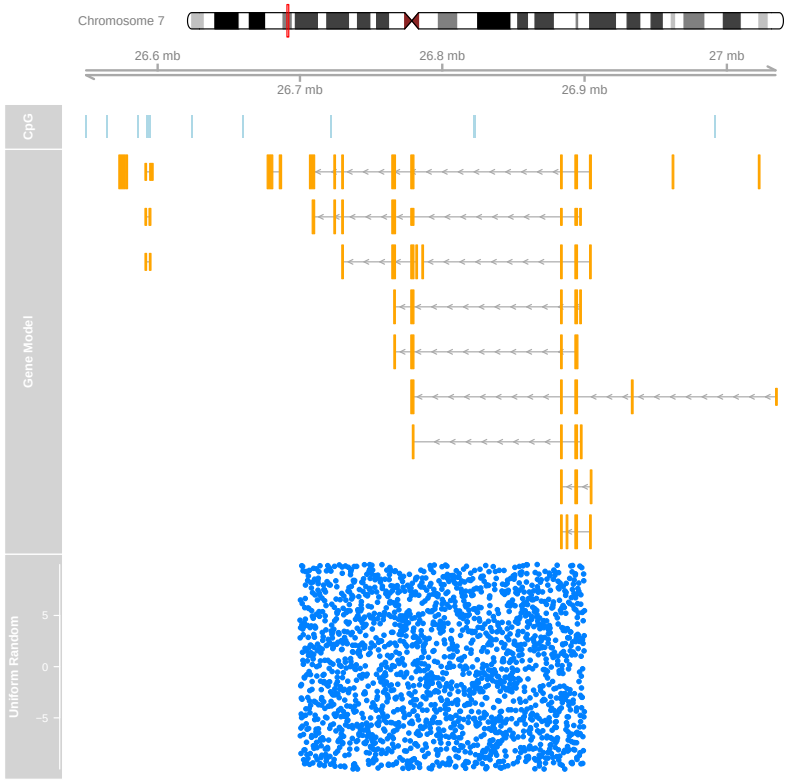

Figure 3.1: *Gviz* ideogram, genome coordinate, annotation, and data tracks.

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

```
plotTracks(c(tracks, dtrack))
```

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

Bibliography

[1] S. Anders, D. J. McCarthy, Y. Chen, M. Okoniewski, G. K. Smyth, W. Huber, and M. D. Robinson. Count-based differential expression analysis of RNA sequencing data using R and Bioconductor. *Nat Protoc*, 8(9):1765–1786, Sep 2013.

[2] D. Bentley, S. Balasubramanian, H. Swerdlow, G. Smith, J. Milton, C. Brown, K. Hall, D. Evers, C. Barnes, H. Bignell, et al. Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–59, 2008.

[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] P. Burns. The R inferno. Technical report, 2011.

[5] J. G. Caporaso, C. L. Lauber, W. A. Walters, D. Berg-Lyons, J. Huntley, N. Fierer, S. M. Owens, J. Betley, L. Fraser, M. Bauer, et al. Ultra-high-throughput microbial community analysis on the illumina hiseq and miseq platforms. *The ISME journal*, 6(8):1621–1624, 2012.

[6] T. S. Furey. Chip–seq and beyond: new and improved methodologies to detect and characterize protein–dna interactions. *Nature Reviews Genetics*, 13(12):840–852, 2012.

[7] D. B. Goldstein, A. Allen, J. Keebler, E. H. Margulies, S. Petrou, S. Petrovski, and S. Sunyaev. Sequencing studies in human genetics: design and interpretation. *Nat. Rev. Genet.*, 14(7):460–470, Jul 2013.

[8] E. Khurana, Y. Fu, V. Colonna, X. J. Mu, H. M. Kang, T. Lappalainen, A. Sboner, L. Lochovsky, J. Chen, A. Harmanci, J. Das, A. Abyzov, S. Balasubramanian, K. Beal, D. Chakravarty, D. Challis, Y. Chen, D. Clarke, L. Clarke, F. Cunningham, U. S. Evani, P. Flicek, R. Fragoza, E. Garrison, R. Gibbs, Z. H. Gümüş, J. Herrero, N. Kitabayashi, Y. Kong, K. Lage, V. Liluashvili, S. M. Lipkin, D. G. MacArthur, G. Marth, D. Muzny, T. H. Pers, G. R. S. Ritchie, J. A. Rosenfeld, C. Sisu, X. Wei, M. Wilson, Y. Xue, F. Yu, . G. P. Consortium, E. T. Dermitzakis, H. Yu, M. A. Rubin, C. Tyler-Smith, and M. Gerstein. Integrative annotation of variants from 1092 humans: Application to cancer genomics. *Science*, 342(6154), 2013.

[9] M. Lawrence, W. Huber, H. Pagès, P. Aboyoun, M. Carlson, R. Gentleman, M. T. Morgan, and V. J. Carey. Software for computing and annotating genomic ranges. *PLoS Comput Biol*, 9(8):e1003118, 08 2013.

[10] B. A. Methe et al. A framework for human microbiome research. *Nature*, 486(7402):215–221, Jun 2012.

[11] J. C. Mwenifumbo and M. A. Marra. Cancer genome-sequencing study design. *Nat. Rev. Genet.*, 14(5):321–332, May 2013.

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

[13] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. ISBN 3-900051-07-0.

[14] C. Ross-Innes, R. Stark, A. Teschendorff, K. Holmes, H. Ali, M. Dunning, G. Brown, O. Gojis, I. Ellis, A. Green, et al. Differential oestrogen receptor binding is associated with clinical outcome in breast cancer. *Nature*, 481(7381):389–393, 2012.

[15] J. Rothberg and J. Leamon. The development and impact of 454 sequencing. *Nature biotechnology*, 26(10):1117–1124, 2008.

[16] A. Sottoriva, I. Spiteri, D. Shibata, C. Curtis, and S. Tavaré. Single-molecule genomic data delineate patient-specific tumor profiles and cancer stem cell organization. *Cancer research*, 73(1):41–49, 2013.