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

Martin Morgan[*], Hervé Pagès[†]

August 15, 2011

Contents

[*]mtmorgan@fhcrc.org
[†]hpages@fhcrc.org

1 Introduction

1.1 *Bioconductor*

Bioconductor is a collection of *R* packages for the analysis and comprehension of high-throughput genomic data. *Bioconductor* started more than 10 years ago. It gained credibility for its statistically rigorous approach to microarray pre-preprocessing and designed experiments, and integrative and reproducible approaches to bioinformatic tasks. There are now more than 460 *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.

1.2 High-throughput sequence analysis

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

1.3 This workshop

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

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

2 *R* and *Bioconductor*

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

R has several unique features. It has a surprisingly 'old school' interface: users type commands into a console; scripts in plain text represent work flows; tools other than *R* are used for editing and other tasks. *R* is a flexible programming language, so while one person might use functions provided by *R* to accomplish advanced analytic tasks, another might implement their own functions for novel data. 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. Of course statistical analyses, especially exploratory, are very well-developed.

Bioconductor is a collection of *R* packages for the analysis and comprehension of high-throughput genomic data. *Bioconductor* started more than 10 years ago. It gained credibility for its statistically rigorous approach to microarray pre-preprocessing and designed experiments, and integrative and reproducible approaches to bioinformatic tasks. There are now more than 460 *Bioconductor* packages for expression and other microarrays, sequence analysis, flow cytometry, imaging, and other domains.

2.1 Statistical programming

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

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

Analysis of high-throughput data is necessarily statistical. The volume of data requires that it be appropriately summarized before any sort of comprehension is possible. The data are produced by advanced technologies, and these

introduce artifacts (e.g., probe-specific bias in microarrays; sequence or base calling bias in RNA-seq experiments) that need to be accommodated to avoid incorrect or inefficient inference. Data sets typically derive from designed experiments, requiring a statistical approach both to account for the design, and to correctly address the large number of observed values (e.g., gene expression or sequence tag counts) and small number of samples accessible in typical experiments.

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

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

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

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

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

R is ideally suited to addressing the statistical challenges of high-throughput data. Three examples include the development of the 'RMA' and other normalization algorithm for microarray pre-processing, use of moderated *t*-statistics for

assessing microarray differential expression, and development of approaches to estimating dispersion read counts necessary for appropriate analysis of RNAseq designed experiments.

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

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

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

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

2.2 *R* data types

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

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

[1] 5 4 3 2 1
```

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

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

```
> x[2:4]

[1] 4 3 2
```

R functions operate on variables. Functions are usually vectorized, acting on all elements of their argument and obviating the need for explicit iteration. Functions can generate warnings when performing suspect operations, or errors if evaluation cannot proceed; try `log(0)` or `log(-1)`.

```
> log(x)

[1] 1.61 1.39 1.10 0.69 0.00
```

Essential data types *R* has a number of standard data types, to represent `integer`, `numeric` (floating point), `complex`, `character`, `logical` (boolean), and `raw` (byte) data. It is possible to convert between data types, and to discover the type or mode of a variable.

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

[1] 1.1 1.2 1.3

> c(FALSE, TRUE, FALSE)     # logical

[1] FALSE  TRUE FALSE

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

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

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

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

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

[1] "double"

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

[1] "integer"

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

[1] "integer"
```

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

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

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

Lists, data frames, and matrices All of the vectors mentioned so far are homogenous, consisting of a single type of element. A `list` can contain a collection of different types of elements and, like all vectors, these elements can be named to create a key-value association.

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

$a
[1] 1 2 3

$b
[1] "foo" "bar"

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

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

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

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

$a
[1] 1 2 3
```

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

[1] 1 2 3
```

A `data.frame` is a list of equal-length vectors, representing a rectangular data structure not unlike a spread sheet. Each column of the data frame is a vector, so data types must be homogenous with a column. A `data.frame` can be subset by row or column, and columns can be accessed with `$` or `[[`.

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

  age    sex
1  27   Male
2  32 Female
3  19   Male

> df[c(1, 3),]

  age  sex
1  27 Male
3  19 Male

> df[df$age > 20,]

  age    sex
1  27   Male
2  32 Female
```

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

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

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

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

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

> m[, 3]

[1] 7 8 9
```

```
> m[, 3, drop=FALSE]
     [,1]
[1,]    7
[2,]    8
[3,]    9
```

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

S3 and S4 classes More complicated data structures are represented using the 'S3' or 'S4' object system. Objects are often created by functions (`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 interogated 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
    0.00543      1.00444

> anova(fit)

Analysis of Variance Table

Response: y
           Df Sum Sq Mean Sq F value Pr(>F)
x           1   1045    1045    4307 <2e-16 ***
Residuals 998    242       0
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.âĂŹ 0.1 âĂŸ âĂŹ 1

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

[1] 0.49

> class(fit)

[1] "lm"
```

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

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

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

[1] 1.61 1.39 1.10 0.69 0.00

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

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

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

[1] 2.3 2.0 1.6 1.0 0.0

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

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

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

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

[1] 2.3 2.0 1.6 1.0 0.0
```

A 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
```

2.3 Packages

Packages provide functionality beyond that available in base *R*. There are over 3000 packages in CRAN (comprehensive *R* archive network) and more than 460 *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.

The *lattice* package is distributed with *R* but not loaded by default. It provides a very expressive way to visualize data. The following example plots

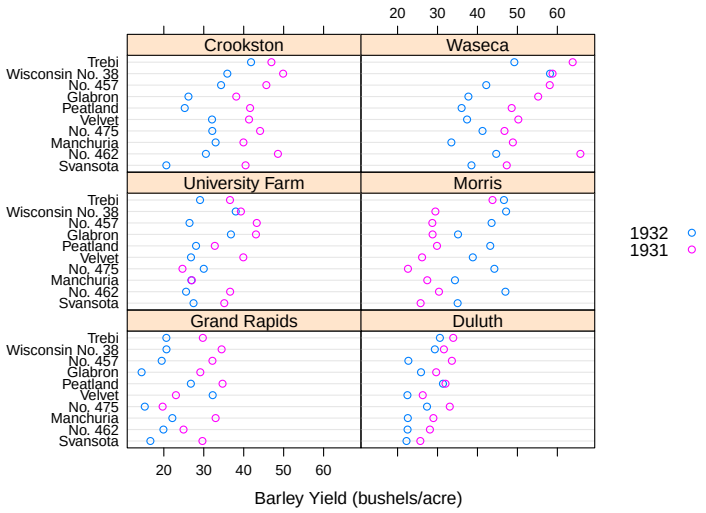

Figure 1: Variety yield conditional on site and grouped by year, for the `barley` data set.

yield for a number of barley varieties, conditioned on site and grouped by year. Figure 1 is read from the lower left corner. Note the common scales, efficient use of space, and not-too-pleasing default color palette. The Waseca sample appears to be mis-labelled for 'year', an apparent error in the original data. Find out about the built-in data set used in this example with `?barley`.

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

New packages can be added to an *R* installation using `install.packages`. A package is installed only once per *R* installation, but needs to be loaded (with `library`) in each session in which it is used. Loaded packages are displayed with `search`. The path returned by `search` represents the order in which the global environment (where commands entered at the prompt are evaluated) and attached packages are searched for symbols; it is possible for a package earlier in the search path to mask symbols later in the search path; these can be disambiguated using `::`.

```
> search()
```

```
 [1] ".GlobalEnv"
 [2] "package:useR2011"
```

```
 [3] "package:BSgenome.Dmelanogaster.UCSC.dm3"
 [4] "package:BSgenome"
 [5] "package:org.Dm.eg.db"
 [6] "package:RSQLite"
 [7] "package:DBI"
 [8] "package:AnnotationDbi"
 [9] "package:Biobase"
[10] "package:goseq"
[11] "package:geneLenDataBase"
[12] "package:BiasedUrn"
[13] "package:edgeR"
[14] "package:ShortRead"
[15] "package:Rsamtools"
[16] "package:lattice"
[17] "package:Biostrings"
[18] "package:GenomicFeatures"
[19] "package:GenomicRanges"
[20] "package:IRanges"
[21] "package:stats"
[22] "package:graphics"
[23] "package:grDevices"
[24] "package:utils"
[25] "package:datasets"
[26] "package:methods"
[27] "Autoloads"
[28] "package:base"
```

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

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

Exercise 1
Use the `library` *function to load the useR2011 package. Use the* `sessionInfo` *function to verify that you are using R version 2.13.1 and current packages, similar to those reported here.*

Solution:

```
> library(useR2011)
> sessionInfo()
```

```
R version 2.13.1 (2011-07-08)
Platform: x86_64-unknown-linux-gnu (64-bit)

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
 [5] LC_MONETARY=C              LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
```

```
attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
 [1] useR2011_0.1.0
 [2] BSgenome.Dmelanogaster.UCSC.dm3_1.3.17
 [3] BSgenome_1.20.0
 [4] org.Dm.eg.db_2.5.0
 [5] RSQLite_0.9-4
 [6] DBI_0.2-5
 [7] AnnotationDbi_1.14.1
 [8] Biobase_2.12.2
 [9] goseq_1.4.0
[10] geneLenDataBase_0.99.7
[11] BiasedUrn_1.04
[12] edgeR_2.2.5
[13] ShortRead_1.10.4
[14] Rsamtools_1.4.3
[15] lattice_0.19-31
[16] Biostrings_2.20.2
[17] GenomicFeatures_1.4.4
[18] GenomicRanges_1.4.7
[19] IRanges_1.10.6

loaded via a namespace (and not attached):
 [1] Matrix_0.9996875-3 RCurl_1.6-7        XML_3.4-2          biomaRt_2.8.1
 [5] grid_2.13.1        hwriter_1.3        limma_3.8.3        mgcv_1.7-6
 [9] nlme_3.1-102       rtracklayer_1.12.4 tools_2.13.1
```

2.4 Help

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

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

The 'Search Engine and Keywords' link is helpful in day-to-day use.

Manual pages Use manual pages to find detailed descriptions of the arguments and return values of functions, and the structure and methods of classes. Find help within an *R* session as

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

S3 methods can be queried interactively. For S3,

```
> methods(anova)
```

```
 [1] anova.MAList* anova.gam*    anova.glm     anova.glmlist anova.gls*
 [6] anova.lm      anova.lme*    anova.loess*  anova.mlm     anova.nls*

   Non-visible functions are asterisked
```

```
> methods(class="lm")
```

```
 [1] add1.lm*           alias.lm*          anova.lm           case.names.lm*
 [5] confint.lm*        cooks.distance.lm* deviance.lm*       dfbeta.lm*
 [9] dfbetas.lm*        drop1.lm*          dummy.coef.lm*     effects.lm*
[13] extractAIC.lm*     family.lm*         formula.lm*        hatvalues.lm
[17] influence.lm*      kappa.lm           labels.lm*         logLik.lm*
[21] model.frame.lm     model.matrix.lm    nobs.lm*           plot.lm
[25] predict.lm         print.lm           proj.lm*           qqnorm.lm*
[29] qr.lm*             residuals.lm       rstandard.lm       rstudent.lm
[33] simulate.lm*       summary.lm         variable.names.lm* vcov.lm*

   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.lme")
```

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; use `getAnywhere` to retrieve non-visible function definitions. We use `head` to look at the first six lines of the `head` method specialized for `matrix` objects.

```
> utils::head
```

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

```
> methods(head)
```

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

   Non-visible functions are asterisked
```

```
> head(head.matrix)
```

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

S4 classes and generics are queried in a similar way, as for the `complement` generic in the *Biostrings* package:

```
> showMethods(complement)
```

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

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

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

Obtaining help on S4 classes and methods requires syntax such as

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

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

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

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

```
> browseVignettes("Rsamtools")
```

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

2.5 The *Bioconductor* web site

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

- Brief introductory work flows.
- A manifest of all *Bioconductor* packages arranged alphabetically or as 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.
- Access to the mailing lists, including searchable archives, as the primary source of help.

- Course and conference information, including extensive reference material.
- General information about the project.
- Information for package developers, including guidelines for creating and submitting new packages.

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

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

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

c. *A description of the GappedAlignments class.*

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

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

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

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

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

```
> ?library
> library(Biostrings)
> ?alphabetFrequency
> class?GappedAlignments
> browseVignettes("GenomicRanges")
```

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

3 Ranges and strings

This section introduces two essential ways in which sequence data are manipulated. Ranges describe both aligned reads and features of interest on the genome. Sets of DNA strings represent the reads themselves and the nucleotide sequence of reference genomes.

3.1 Reads and genomic features as ranges

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

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

GRanges Instances of *GRanges* are used to specify genomic coordinates. Suppose we wished to represent two *D. melanogaster* genes. The first is located on the positive strand of chromosome 3R, from position 19967116 to 19973212. The second is on the minus strand of the X chromosome, with 'left-most' base at 18962305, and right-most base at 18962925. The coordinates are *1-based* (i.e., the first nucleotide on a chromosome is numbered 1, rather than 0), *left-most* (i.e., reads on the minus strand are defined to 'start' at the left-most coordinate, rather than the 5' coordinate), and *closed* (the start and end coordinates are included in the range; a range with identical start and end coordinates has width 1, a 0-width range is represented by the special construct where the end coordinate is one less than the start coordinate).

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

```
> genes <- GRanges(seqnames=c("3R", "X"),
+                  ranges=IRanges(
+                      start=c(19967116, 18962305),
+                      end=c(19973212, 18962925)),
+                  strand=c("+", "-"),
+                  seqlengths=c(`3R`=27905053L, `X`=22422827L))
```

The components of a gene 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 *D. melanogaster*. This data is displayed as

```
> genes
```

```
GRanges with 2 ranges and 0 elementMetadata values
    seqnames               ranges strand |
       <Rle>            <IRanges>  <Rle> |
```

```
[1]       3R [19967116, 19973212]      + |
[2]        X [18962305, 18962925]      - |

seqlengths
       3R        X
 27905053 22422827
```

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

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

```
> ?GRanges
```

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

```
> browseVignettes("GenomicRanges")
```

for a comprehensive introduction. A *GRanges* instance can be subset, with accessors for setting, updating, and querying information.

```
> genes[2]

GRanges with 1 range and 0 elementMetadata values
    seqnames               ranges strand |
       <Rle>            <IRanges>  <Rle> |
[1]        X [18962305, 18962925]      - |

seqlengths
       3R        X
 27905053 22422827

> strand(genes)

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

> width(genes)

[1] 6097  621

> length(genes)

[1] 2

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

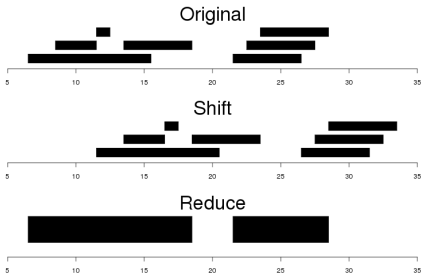

Figure 2: Ranges

```
GRanges with 2 ranges and 0 elementMetadata values
            seqnames               ranges strand |
               <Rle>            <IRanges>  <Rle> |
FBgn0039155       3R [19967116, 19973212]      + |
FBgn0085359        X [18962305, 18962925]      - |

seqlengths
       3R        X
 27905053 22422827
```

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

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

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

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

These are illustrated in the upper left panel of Figure 2.

Methods on ranges can be grouped as follows:

Intra-range methods act on each range independently. These include `flank`, `narrow`, `reflect`, `resize`, `restrict`, `shift`. An illustration is `shift`, which translates each range by the amount specified by the `shift` argument. Positive values shift to the right, negative to the left; `shift` can be a

vector, with each element of the vector shifting the corresponding element of the *IRanges* instance. Here we shift all ranges to the right by 5, with the result illustrated in the middle panel of Figure 2

```
> shift(ir, 5)
```

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

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

```
> reduce(ir)
```

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

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

```
> coverage(ir)
```

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

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

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

`countOverlaps` and `findOverlaps` also 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 overlap. Both arguments allow some flexibility in the definition of 'overlap'.

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

```
> elementMetadata(genes) <-
+     DataFrame(EntrezId=c("42865", "42865"),
+               Symbol=c("kal-1", "CG6173"))
```

`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="Martin Morgan", Date=date())
```

3.2 Genomic features

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 length 1 list, where the element of the list contains a *GRanges* object with 7 elements:

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


seqlengths
    chr3R
 27905053
```

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

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

Exercise 3
Use the helper function `bigdata` *and* `list.files` *to identify the path to a data base created by* `makeTranscriptDbFromUCSC`*.*

Load the saved TranscriptDb object using `loadFeatures`*.*

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

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

Solution:

```
> txdbFile <- list.files(bigdata(), "sqlite", full=TRUE)
> txdb <- loadFeatures(txdbFile)
> ex0 <- exonsBy(txdb, "gene")
> head(table(elementLengths(ex0)))

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

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

Exercise 4
(Independent) Create a TranscriptDb instance from UCSC, using `makeTranscriptDbFromUCSC`*.*

Solution:

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

3.3 Working with strings

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

Exercise 5
The objective of this exercise is to calculate the GC content of the exons of a single gene, whose coordinates are specified by the `ex` *object of the previous exercise.*

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

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

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

The useR2011 package defines a helper function `gcFunction` *(developed in a later exercise) to calculate GC content. Use this to calculate the GC content in each of the exons.*

Solution:

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

Here is the helper function, available in the *useR2011* package, to calculate GC content:

```
> gcFunction

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

The subject GC content is

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

4 Exploring sequence data: short reads and alignments

The following sections introduce core tools for working with high-throughput sequence data. This section focus on the reads and alignments that are the raw material for analysis. Section 5 addresses statistical approaches to assessing differential representation in RNA-seq experiments.

4.1 The *pasilla* data set

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

In this section we look at a subset of the *ps* data, corresponding to reads obtained from lanes of their RNA seq experiment, and to the same reads aligned to a *D. melanogaster* reference genome. Reads were obtained from GEO and the Short Read Archive (SRA); reads were aligned to *D. melanogaster* reference genome dm3 as described in the *pasilla* experiment data package (this package is available in the 'devel' version of *Bioconductor*; the devel version will be released in October, 2011).

4.2 Short reads

Sequencer technologies The Illumina GAII and HiSeq technologies generate sequencers by measuring incorporation of florescent nucleotides over successive PCR cycles. These sequencers produce 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. In the sample above the identifier produces 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 relative to batch effects. The spacial coordinates (lane, tile, x, y) are often used to identify optical duplicates (artifacts introduced when the sequencer falsely interprets reads in close proximity spatial as distinct); spacial coordinates can also be used during quality assessment to identify spatial artifacts of sequencing, 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. It is given in 5' to 3' orientation as seen by the sequencer. A letter `N` is used to signify bases that the sequencer was not able to call. The fourth line of the FASTQ record encodes the quality (confidence) of the corresponding base call. The quality score is encoded following one of several conventions, with the general notion being that letters later in the visible ASCII alphabet

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

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

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

Short reads in *R* FASTQ files can be read in to *R* using the `readFastq` function from the *ShortRead* package. Use this function by providing the path to a FASTQ file. There are sample data files available in the *useR2011* package, each consisting of 1 million reads from a lane of the Pasilla data set.

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

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

The data are represented as an object of class *ShortReadQ* ('short read and *q*uality').

```
> 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 IIIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
[2]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
[3]    37 IIIIIIIIIIIIIIIIIIIII'IIIIIGBIIII2I+
```

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

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

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

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

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

Slots:

Name:       quality        sread           id
Class: QualityScore DNAStringSet   BStringSet

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

Known Subclasses: "AlignedRead"
```

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

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

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

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

```
     37
1000000
```

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

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

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

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

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

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

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

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

A report from a larger subset of the experiment is available

```
> rpt <- system.file("GSM461176_81_qa_report", package="useR2011")
> browseURL(rpt)
```

Exercise 6

Use the helper function `bigdata` *(defined in the useR2011 package) and the* `file.path` *and* `list.files` *functions to locate two fastq files from [2] (the files were obtained as described in the appendix and pasilla experiment data package, available with the development versions of R and Bioconductor).*

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

Use `alphabetFrequency` *to summarize the GC content of all reads (hint: use the* `sread` *accessor to extract the reads, and the* `collapse=TRUE` *argument to the* `alphabetFrequency` *function). Using the helper function* `gcFunction` *from the useR2011 package, draw a histogram of the distribution of GC frequencies across reads.*

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

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

Solution: Discovery:

```
> list.files(bigdata())
```

```
[1] "bam"                       "dm3.ensGene.txdb.sqlite"
[3] "fastq"
```

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

Input:

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

GC content:

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

```
[1] 0.55
```

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

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

Alphabet by cycle:

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

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

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

Exercise 7
Use `quality` *to extract the quality scores of the short reads. Interpret the encoding qualitatively.*

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

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

Solution:

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

```
class: FastqQuality
quality:
  A BStringSet instance of length 6
    width seq
[1]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
[2]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
[3]    37 IIIIIIIIIIIIIIIIIIIII'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")
```

4.3 Alignments

Most down-stream analysis of short read sequences is based on reads aligned to reference genomes. There are many aligners available, including BWA [5, 4], Bowtie [3], GSNAP, and Illumina's ELAND; merits of these are discussed in the literature. There are also alignment algorithms implemented in *Bioconductor* (e.g., `matchPDict` in the *Biostrings* package, and the *Rsubread* package); `matchPDict` is particularly useful for flexible alignment of moderately sized subsets of data.

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

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

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

Table 1: Fields in a SAM record. From http://samtools.sourceforge.net/samtools.shtml

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

```
[15] "UQ:i:0"
[16] "H0:i:1"
[17] "H1:i:0"
```

Fields in a SAM file are summarized in Table 1. We recognize from the FASTQ file the identifier string, read sequence and quality. The alignment is to a chromosome 'seq1' starting at position 1. The strand of alignment is encoded in the 'flag', field. The alignment record also includes a measuring of mapping quality and a CIGAR string describing the nature of the alignment. In this case, the CIGAR is 35M, indicating that the alignment consisted of 35 `M`atches or mismatches, with no indels or gaps; indels are represented by `I` and `D`; gaps (e.g., from alignments spanning introns) by `N`.

BAM files encode the same information as SAM files, but in a format that is more efficiently parsed by software; BAM files are the primary way in which aligned reads are imported in to *R*.

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

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

GappedAlignments of length 3
    rname strand cigar qwidth start end width ngap
[1]  seq1      +   36M     36     1  36    36    0
[2]  seq1      +   35M     35     3  37    35    0
[3]  seq1      +   35M     35     5  39    35    0

seqlengths
```

```
 seq1 seq2
 1575 1584
```

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

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

```
> table(strand(aln))

   +    -
1647 1624

> table(width(aln))

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

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

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

Exercise 8

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A simple function for counting reads is

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

This can be applied to all files using `sapply`

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

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

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

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

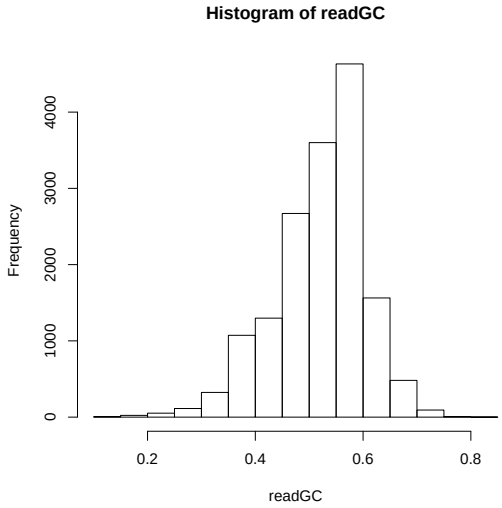

Figure 3: GC content in aligned reads

Exercise 9
Consult the help page for `ScanBamParam`*, and construct an object that restricts the information returned by a* `scanBam` *query to the aligned read DNA sequence. Your solution will use the* `what` *parameter to the* `ScanBamParam` *function.*

Use the `ScanBamParam` *object to query a BAM file, and calculate the GC content of all aligned reads. Summarize the GC content as a histogram (Figure 3).*

Solution:

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

5 RNA-seq

5.1 Varieties of RNA-seq

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

The most straight-forward RNA-seq experiments quantify abundance of known gene models. The known models are derived from reference databases, reflecting the accumulated wisdom of the community responsible for the data. The 'knownGenes' track of the UCSC genome browser represents one source of data. It contains, for each gene, the transcripts and exons that are thought through experimental or computational approaches to exist. The *GenomicFeatures* package allows ready access to this information, as we have seen. The data base of known genes is coupled with high throughput sequence data by counting or otherwise estimating the number of reads associated with each gene.

A more ambitious approach to RNA-seq attempts to identify novel transcripts. This requires that sequenced reads be assembled into contigs that, presumably, correspond to expressed transcripts that are then located in the genome. Transcripts identified in this way may correspond to known transcripts, to novel organization of known exons (e.g., through alternative splicing), or to completely novel constructs. We will not address the identification of completely novel transcripts here, but note that having quantified transcript abundances in several samples one is still interested in the analysis of designed experiments – do transcript abundances, novel or otherwise, differ between experimental groups?

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

5.2 Data preparation

Counting reads aligning to genes An essential step is to arrive at some measure of gene representation amongst the aligned reads. A straight-forward and commonly used approach is to count the number of times a read overlaps exons. Nuance arises when a read only partly overlaps an exon; when two exons overlap (and hence a read appears to be 'double counted'); when reads are aligned with gaps, and the gaps are inconsistent with known exon boundaries; etc. The `countGenomicOverlaps` function in the *GenomicRanges* package provides facilities for implementing different count strategies, using arguments such as

`type` to determine the nature of the exon / read overlap, and `resolution` to select strategies for counting reads overlapping multiple subjects. The behavior of the function is also influenced by whether the queried region is an *GRanges* or *GRangesList* instance.

Software other than *R* can also be used to summarize count data. An important point is that the desired input is often raw count data, rather than normalized (e.g., reads per kilobase of (gene) model per million mapped reads) values. This is because counts allow information about uncertainty of estimates to propagate to later stages in the analysis.

Object creation and filtering The following exercise illustrates key steps in data preparation.

Exercise 10
The user2011 package contains a data set with pre-computed count data. Use the `data` *command to load it. Create a variable* `grp` *to define the groups associated with each column, using the column names as a proxy for more authoritative metadata.*

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

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

Use `plotMDS` *on the filtered reads to perform multi-dimensional scaling. Interpret the resulting plot.*

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

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

[1] 14470     7

> grp <- factor(sub("[1-4].*", "", colnames(counts)),
+               levels=c("untreated", "treated"))
```

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

```
> library(edgeR)
> dge <- DGEList(counts, group=grp)
> dge <- calcNormFactors(dge)
```

To filter reads, we scale the counts by the library sizes and express the results on a per-million read scale. We require that the gene be represented at a frequency of at least 1 read per million mapped in two or more of each sample, and use this criterion to subset the *DGEList* instance.

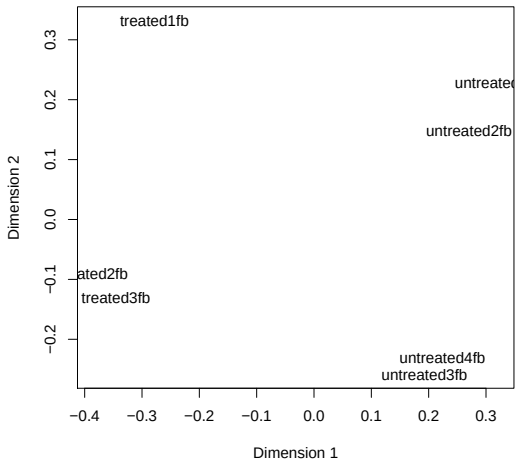

Figure 4: MDS plot of lanes from the Pasilla data set.

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

```
ridx
FALSE  TRUE
 6476  7994
```

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

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

```
> plotMDS.dge(dge)
```

```
Using grid search to estimate tagwise dispersion.
```

5.3 Differential representation

RNA-seq differential representation experiments, like classical microarray experiments, consist of a single statistical design (e.g, comparing expression of samples assigned to 'Treatment' versus 'Control' groups) applied to each feature for which there are aligned reads. While one could naively perform simple

tests (e.g., t-tests) on all features, it is much more informative to identify important aspects of RNAseq experiments, and to take a flexible route through this part of the work flow. Key steps involve formulation of a model matrix to capture the experimental design, estimation of a test static to describe differences between groups, and calculation of a P value or other measure as a statement of statistical significance.

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

Exercise 11
To be a more concrete, use the `model.matrix` function and a formula involving `grp` to create the model matrix for our experiment.

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

```
> (design <- model.matrix( ~ grp ))
```

```
  (Intercept) grptreated
1           1          1
2           1          1
3           1          1
4           1          0
5           1          0
6           1          0
7           1          0
attr(,"assign")
[1] 0 1
attr(,"contrasts")
attr(,"contrasts")$grp
[1] "contr.treatment"
```

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

on the other it provides great flexibility in the type of experiment that can be analyzed.

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

```
> dge <- estimateCommonDisp(dge, design)
> sqrt(dge$common.dispersion)
```

```
[1] 0.78
```

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

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

Exercise 12
Use `glmFit` to fit the general linear model. This function requires the input data `dge`, the experimental design `design`, and as estimate of dispersion.

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

Finally, create a 'top table' of differentially represented genes using `topTags`.

Solution: Here we fit a glm to our data and experimental design, using the common dispersion estimate.

```
> fit <- glmFit(dge, design, dispersion=dge$common.dispersion)
```

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

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

Here `topTags` function summarizes results across the experiment.

```
> (tt <- topTags(lrTest))
```

```
Coefficient:  grptreated
            logConc logFC LR P.Value   FDR
FBgn0039155   -9.6  -4.7 20 9.6e-06 0.056
FBgn0085359  -12.3  -4.7 19 1.5e-05 0.056
FBgn0024288  -12.4  -4.7 18 2.1e-05 0.056
FBgn0039827  -10.6  -4.2 17 4.3e-05 0.086
FBgn0034434  -11.4  -4.0 15 1.0e-04 0.165
FBgn0033764  -12.1   3.5 14 1.4e-04 0.188
FBgn0034736  -11.0  -3.5 12 4.6e-04 0.502
FBgn0033065  -13.0   3.3 12 5.1e-04 0.502
FBgn0037290  -12.0   3.1 12 6.2e-04 0.502
FBgn0035189  -11.0   3.1 12 6.3e-04 0.502
```

As a 'sanity check', summarize the original data for the first several probes

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

```
          FBgn0039155 FBgn0085359 FBgn0024288 FBgn0039827
untreated        1576       118.2       102.5         554
treated            64         4.7         4.3          31
```

6 Annotation

6.1 Major types of annotation in *Bioconductor*

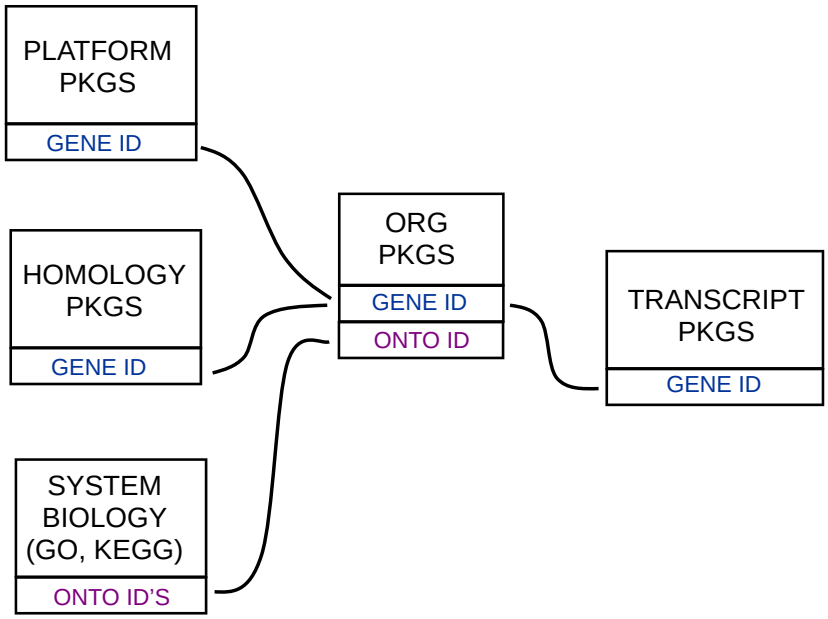

Figure 5: Annotation Packages: the big picture

Gene centric *AnnotationDbi* packages:

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

Genome centric *GenomicFeatures* packages:

- Transcriptome level: e.g. *TxDb.Hsapiens.UCSC.hg19.knownGene*
- Generic genome features: Can generate via *GenomicFeatures*

biomaRt:

- Query web-based 'biomart' resource for genes, sequence, SNPs, and etc.

6.2 Organism level packages

An organism level package (aka "org package") is organized around a central gene id (e.g. Entrez Gene id) and contains a collection of mappings between this central id and other kinds of ids (e.g. GenBank or Uniprot accession number, RefSeq id, etc...). The name of an org package is always of the form *org.<Ab>.<efg>.db* (e.g. *org.Sc.sgd.db*) where *<Ab>* is a 2-letter abbreviation of the organism (e.g. *Sc* for Saccharomyces cerevisiae) and *<efg>* is an abbreviation (in lower-case) describing the type of central gene id (e.g. *sgd* for gene ids assigned by the Saccharomyces Genome Database people or *eg* for Entrez Gene ids).

The document of reference for using org packages is the *How to use the ".db" annotation packages* vignette in the *AnnotationDbi* package (org packages are only one type of ".db" annotation packages).

Like almost all the annotation packages in *Bioconductor*, the ".db" annotation packages are updated every 6 months (i.e. at every new *Bioconductor* release).

Exercise 13

What's the name of the org package for Fly? Load it.

Use `ls('package:<pkgname>')` *to display the list of all the symbols defined in this package. Explore a few of them by looking at their man page, at their class, and by extracting a few samples from them with* `sample(map, 5)`.

Turn a map into a data frame with `toTable` *(use* `head` *to only display the first rows). What are the left keys? What are the right keys?*

Most maps can be reversed with `revmap`. *Reverse a map and extract a few samples from the reversed map.*

Note that reversing a map does NOT switch the left and right keys. You can check this with the `Lkeys` *and* `Rkeys` *accessors.*

Solution:

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

```
 [1] "org.Dm.eg"                "org.Dm.egACCNUM"
 [3] "org.Dm.egACCNUM2EG"       "org.Dm.egALIAS2EG"
 [5] "org.Dm.egCHR"             "org.Dm.egCHRLENGTHS"
 [7] "org.Dm.egCHRLOC"          "org.Dm.egCHRLOCEND"
 [9] "org.Dm.egENSEMBL"         "org.Dm.egENSEMBL2EG"
[11] "org.Dm.egENSEMBLPROT"     "org.Dm.egENSEMBLPROT2EG"
[13] "org.Dm.egENSEMBLTRANS"    "org.Dm.egENSEMBLTRANS2EG"
[15] "org.Dm.egENZYME"          "org.Dm.egENZYME2EG"
[17] "org.Dm.egFLYBASE"         "org.Dm.egFLYBASE2EG"
[19] "org.Dm.egFLYBASECG"       "org.Dm.egFLYBASECG2EG"
[21] "org.Dm.egFLYBASEPROT"     "org.Dm.egFLYBASEPROT2EG"
[23] "org.Dm.egGENENAME"        "org.Dm.egGO"
[25] "org.Dm.egGO2ALLEGS"       "org.Dm.egGO2EG"
[27] "org.Dm.egMAP"             "org.Dm.egMAP2EG"
[29] "org.Dm.egMAPCOUNTS"       "org.Dm.egORGANISM"
[31] "org.Dm.egPATH"            "org.Dm.egPATH2EG"
```

```
[33] "org.Dm.egPMID"            "org.Dm.egPMID2EG"
[35] "org.Dm.egREFSEQ"          "org.Dm.egREFSEQ2EG"
[37] "org.Dm.egSYMBOL"          "org.Dm.egSYMBOL2EG"
[39] "org.Dm.egUNIGENE"         "org.Dm.egUNIGENE2EG"
[41] "org.Dm.egUNIPROT"         "org.Dm.eg_dbInfo"
[43] "org.Dm.eg_dbconn"         "org.Dm.eg_dbfile"
[45] "org.Dm.eg_dbschema"
```

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

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

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

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

```
> sample(org.Dm.egUNIPROT, 5)
```

```
$`248456`
[1] NA

$`249386`
[1] NA

$`32896`
[1] "Q8SZV7" "Q9VWM1"

$`250382`
[1] NA

$`34939`
[1] "Q9VJM9"
```

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

```
  gene_id uniprot_id
1   30970     Q8IRZ0
2   30970     Q95RP8
3   30971     Q95RU8
4   30972     Q9W5H1
5   30973     P39205
6   30975     Q24312
```

The left keys are the Entrez Gene ids and the right keys the Uniprot accession numbers. Note that for all the maps in an org package the left key is always the central gene id.

```
> revmap(org.Dm.egUNIGENE)
```

```
revmap(UNIGENE) map for Fly (object of class "AnnDbBimap")
```

```
> sample(revmap(org.Dm.egUNIGENE), 5)

$Dm.12274
[1] "37523"

$Dm.6045
[1] "246595"

$Dm.29799
[1] "42606"

$Dm.6489
[1] "33183"

$Dm.12224
[1] "31295"

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

[1] TRUE
```

Exercise 14
For convenience, `lrTest`, the DGEGLM object obtained in the previous section with `glmLRT`, has been included to the user2011 package. Load it and create again the 'top table' of differentially represented genes with `topTags`.

Extract the Flybase gene ids from this table and map them to their corresponding Entrez Gene id (create a named character vector with names the Flybase gene ids and values the Entrez Gene ids).

Finally, add 2 columns to the `table` component of the TopTags object created previously: one for the Entrez Gene ids and one for their corresponding gene symbols.

Solution:

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

> fbids <- rownames(tt$table)
> egids <- unlist(mget(fbids, revmap(org.Dm.egFLYBASE), ifnotfound=NA))
> egids

FBgn0039155 FBgn0039827 FBgn0034434 FBgn0034736 FBgn0035189 FBgn0085359
    "42865"     "43689"     "37219"     "37572"     "38124"   "2768869"
FBgn0033764 FBgn0000071 FBgn0024288 FBgn0037290
         NA     "40831"     "45039"     "40613"
```

Because `unlist` can do strange things when the list to unlist has duplicated names, a better way to do this is:

```
> fbids <- rownames(tt$table)
> map <- org.Dm.egFLYBASE
> fbids <- intersect(mappedRkeys(map), fbids)
> egids <- as.character(revmap(map)[fbids])
> egids

FBgn0034434 FBgn0034736 FBgn0035189 FBgn0037290 FBgn0000071 FBgn0039155
    "37219"     "37572"     "38124"     "40613"     "40831"     "42865"
FBgn0039827 FBgn0024288 FBgn0085359
    "43689"     "45039"   "2768869"
```

To add the 2 columns to `tt$table`, we proceed in 3 steps: (1) merge the 2 mappings in a single data frame `anno0`, (2) align the rows in `anno0` with the rows in `tt$table` (by redordering them), and (3) cbind `tt$table` with the 2 new columns:

```
> eg2fb <- toTable(org.Dm.egFLYBASE[egids])
> eg2sym <- toTable(org.Dm.egSYMBOL[egids])
> (anno0 <- merge(eg2fb, eg2sym))

  gene_id  flybase_id  symbol
1 2768869 FBgn0085359 CG34330
2   37219 FBgn0034434    Rgk1
3   37572 FBgn0034736  CG6018
4   38124 FBgn0035189  CG9119
5   40613 FBgn0037290  CG1124
6   40831 FBgn0000071     Ama
7   42865 FBgn0039155   kal-1
8   43689 FBgn0039827  CG1544
9   45039 FBgn0024288 Sox100B

> (anno0 <- anno0[match(rownames(tt$table), anno0$flybase_id), ])

   gene_id  flybase_id  symbol
7    42865 FBgn0039155   kal-1
8    43689 FBgn0039827  CG1544
2    37219 FBgn0034434    Rgk1
3    37572 FBgn0034736  CG6018
4    38124 FBgn0035189  CG9119
1  2768869 FBgn0085359 CG34330
NA    <NA>        <NA>    <NA>
6    40831 FBgn0000071     Ama
9    45039 FBgn0024288 Sox100B
5    40613 FBgn0037290  CG1124

> anno <- cbind(tt$table, anno0[ , c("gene_id", "symbol")])
```

A Appendix: data retrieval

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

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

References

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

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

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

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

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

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