An introduction to *R* and *Bioconductor*

Martin Morgan*

June 20-23, 2011

1 Introduction

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 cytometery, imaging, and other domains.

*mtmorgan@fhcrc.org

2 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.

2.1 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.

2.2 *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.

3 *R* data types and functions

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 interchangably) 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] 1.6094379 1.3862944 1.0986123 0.6931472 0.0000000
```

3.1 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 missinng values.

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

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

3.2 Lists, data frames, and matricies

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.

3.3 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
   -0.01908      0.97658

> anova(fit)

Analysis of Variance Table

Response: y
           Df Sum Sq Mean Sq F value    Pr(>F)
x           1 987.29  987.29  4074.1 < 2.2e-16 ***
Residuals 998 241.85    0.24
---
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] 0.4920244

> class(fit)

[1] "lm"
```

Find help on functions, e.g., the return value of `fit`, using the *R* help system.

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

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.

3.4 Functions

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

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

[1] 1.6094379 1.3862944 1.0986123 0.6931472 0.0000000

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

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

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

[1] 2.321928 2.000000 1.584963 1.000000 0.000000

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

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

Arguments can be matched by 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] 2.321928 2.000000 1.584963 1.000000 0.000000
```

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 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) 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.

```
> 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))
```

4 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

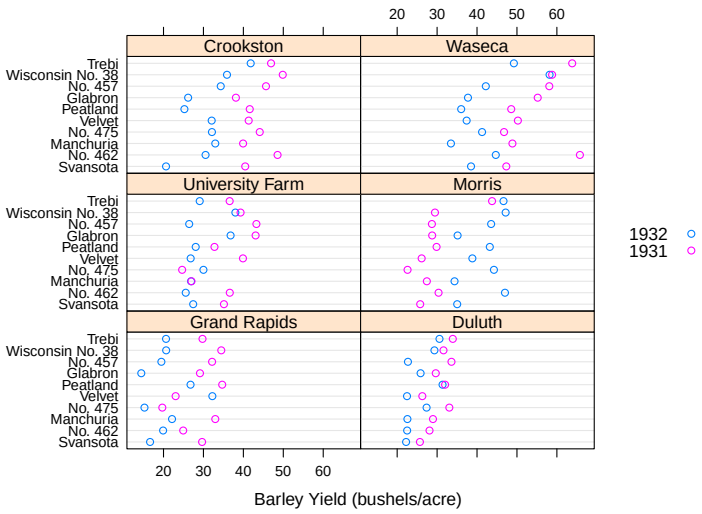

Figure 1: Variety yield conditional on site and grouped by year, for the `barley` data set.

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

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

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

4.1 *Bioconductor*

Bioconductor enhances *R* through packages for microarray, sequence, annotation, flow cytometry, imaging, and other high-throughput genomic analyses. Packages are contributed by academic and other groups from around the world; a core group at the Fred Hutchinson Cancer Research Center is funded by the US National Human Genome Research Institute to provide project infrastructure.

Classical *Bioconductor* work flows involve data input, pre-processing, quality assessment (before or after pre-processing), exploratory and downstream analysis. Downstream steps depends on the research question but might involve analysis of designed experiments (e.g., differential expression between groups), machine learning (clustering and classification), or per-sample processing (e.g., SNP calling, copy number estimation). Results are then subject to gene or pathway annotation and perhaps higher-level analysis (e.g., gene set enrichment). A more concrete work flow example illustrates use of *Bioconductor* packages for expression microarrays; it serves as an established model for other, nascent, technologies.

Input and pre-processing might begin with a vendor-specific package such as *affy*, *lumi*, or *limma* (for two-color arrays). The *oligo* package is useful for high density arrays, *GEOquery* and *ArrayExpress* provide facilities for retrieving data sets from the GEO and ArrayExpress repositories.

Typical pre-processing steps are background correction to address imaging and other technology-specific artifacts, normalization to standardize expression signals from different samples, and summarization of signals from multiple probes of a single feature (e.g., 'gene', used as a synonym for summarized expression values below). Packages used for input provide mechanisms for technology-specific pre-processing; *vsn* (variance stabilizing normalization, using a generalized log transform) is an example of a package that implements normalization methods meant for use on several different platforms. A common

output from pre-processing is an S4 *ExpressionSet* object. This object coordinates expression values, sample metadata (e.g., sample X belongs to treatment group Y, and was processed on day Z), and gene annotation in a way that reduces book-keeping and other common errors associated with managing large data with complicated relationships.

The *arrayQualityMetrics* package produces a comprehensive HTML summary of expression microarray quality, based either on raw or pre-processed data from a variety of platforms. Exploratory analysis of pre-processed data sets often involve a combination of *Bioconductor* packages and standard *R* tools. One might use the *genefilter* package to remove unannotated genes and to identify genes that are highly variable across all samples; genes passing filter form a reduced data set for exploratory or perhaps downstream analysis. Exploratory analysis often involves standard *R* functions, e.g., `heatmap` to cluster samples and look for association with un-intended covariates such as batch processing date.

The next work flow step depends on research question, experimental design, and project-specific objectives. The *limma* package provides an excellent illustration of the opportunities provided by *Bioconductor* for analysis of designed experiments. A typical starting point is the *ExpressionSet* of pre-processed data, and a (statistical) model matrix describing experimental design. The functions `lmFit`, `eBayes` , and `topTable` are then used to fit the linear model implied by the design matrix to each gene, to assess differential expression, and to summarize results in a convenient `data.frame`. This step requires that the user is comfortable with more-than-elementary statistical concepts (e.g., formulating a model matrix and contrasts appropriate for their experiment) and is willing to accept the nuanced statistical reasoning (e.g., about pooled variance estimates and priors) that *limma* implements. In exchange, one gains great flexibility in the type of experiment that can be analyzed, and access to an approach that considerably improves on simple per-gene analyses. The *limma* vignette is an excellent resource.

Bioconductor annotation facilities help place results in context. *AnnotationDbi* is at the base of a large suite of packages that provide gene-level mappings between different identifiers. For instance, *org.Hs.eg.db* is an organism-specific annotation package for *H. sapiens*. It uses Entrez gene identifers (the `eg` in the package name) as a key, from which chromosomal location, gene ontology or KEGG pathway, and other information can be obtained. The *BSgenome* packages povides whole-genome sequences (see `BSgenome::available.genomes()`). The *biomaRt* package allows users to query the rich data resources of Biomart; *rtracklayer* (for the UCSC genome browser) and *GenomicFeatures* (for UCSC and Biomart) provide access to track and other annotation resources. Several packages (e.g., *Category*, *GOstats*, *topGO*, *GSEAlm*, *limma*, *GSEABase*) provide gene set enrichment style analyses.

5 Summary

High-throughput genomic data analysis requires software tools that are capable of dealing with extensive and inter-related data sets. The software must necessarily be statistical to accommodate large data volume, technological artifacts, and experimental design. Scientific researchers work on long term projects in collaborative groups, and require reproducible results and easily modified analyses. Software on the leading edge will not be blemish-free, but can fit in to existing work flows, be well-documented, and easily distributed to diverse users. *R* and *Bioconductor* have many attributes that make them very suitable for genomic data analysis.

R is a statistical programming language. Its vectorized data types can be incorporated into data structures and objects to represent complicated, inter-related data sets or results. The *R* packaging system means that the language is extensible by a wide community; it is the tool of choice for academic and other leading edge research groups.

The *Bioconductor* project builds on *R*'s foundations to provide a large suite of packages for high-throughput analysis. These packages are combined into work flows. The microarray differential expression work flow summarized above provides a template relevant to other domains, highlights many key packages for microarray analysis, and illustrates the sophisticated statistical approaches and rewards made available to *Bioconductor* users.