Statistics with R for Biologists

James H. Bullard
Kasper Daniel Hansen
Margaret Taub

Berkeley, California
July 7-11, 2008

Introduction to R

Origins

- R is a version of the S programming language developed by John Chambers at Bell Labs in 1976 *to turn ideas into software, quickly and faithfully*.
- S was designed to allow people to do statistical analysis without having to write programs in a language like Fortran.
- R is an open source version of the S language described by Chambers et al. in the "blue book."
- R was written initially by Robert Gentleman and Ross Ihaka and released under the GPL in 1995.

Key features

- Being open source makes R a very dynamic language - people are developing new tools in R all the time, implementing the latest statistical methods. This means you should keep your version of R up-to-date: a new release is available every six months.
- Because R is a programming language, not just a program, you can really do anything you want and are capable of implementing, while also having a variety of pre-existing tools at your disposal.
- Additional functionality can easily be added to R through the use of packages.

Bioconductor

- A whole set of packages, designed specifically for working with biological data, is available through Bioconductor (`www.bioconductor.org/`).
- You will all probably be interested in installing packages from Bioconductor at some point.
- You can either install the packages from the command line, or use the GUI Package Installer. *If you use the GUI, be sure the "Install Dependencies" box is checked before you install.*

```
> source("http://bioconductor.org/biocLite.r")
> biocLite(c("affy", "ALL"))
```


GUI / UNIX command-line interface

- Can interact directly with command-line interface by typing commands at prompt
- Good for quick checks or simple calculations that you won't want to document or repeat
- Not good for multi-step analyses which you may want to reproduce in the future

Your "development environment"

- Want a way of documenting your work so that it is readily understandable by another person and reproducible
- Want an efficient way of interacting with R while working in a text-editing environment
- The important thing is to find something that works well for you, that makes you feel comfortable and efficient

For emacs lovers: ESS

- ESS (emacs speaks statistics) is the premier environment (according to Jim and Kasper) for working with and developing R: `ess.r-project.org/`
- "ESS provides a common, generic, and, useful interface, through emacs, to many statistical packages. It currently supports the S family, SAS, BUGS, Stata and XLisp-Stat with the level of support roughly in that order." - ESS manual
- ESS is a general environment for statistical computing in emacs. It can handle a number of other languages for statistical computing like `Stata`, `SAS`, and, `xlisp-stat`. However, it is predominantly used with R/S.

For Mac users: TextMate

- Available for free from `software-central.berkeley.edu/`
- A basic text editor with available application specific "bundles" to allow special functionality for the document you are editing
- The bundle for R allows for quick and easy interaction between your code and the R terminal.

Online resources: R website

- There are some very useful manuals on the R website, including "An Introduction to R" and "R Data Import/Export" : `cran.r-project.org/manuals.html`
- Also informative are the FAQ pages: `cran.r-project.org/faqs.html`
- There is an R-help mailing list, but be sure to read the instructions for posting first! `www.r-project.org/mail.html`
- A local copy of these materials is installed on your computer along with R and can be accessed through `help.start()`.

Other online resources

- A categorized list of R functions: `www.stat.berkeley.edu/~epurdom/RCommands/`
- R graphics gallery: `addictedtor.free.fr/graphiques/`
- R color chart: `research.stowers-institute.org/efg/R/Color/Chart/index.htm`

Getting help from within R

- help(lm): help for a specific function
- ?lm: an alternative way to call help
- help('for'): for certain functions, need quotes
- library(help = "stats"): gives you information on a whole package
- help(package = "stats"): another way to get info on a whole package
- help.search("multivariate normal"): search help-page keywords (not always useful)
- help.start(): launch browser-based help page
- RSiteSearch("multivariate normal"): search R mailing lists, help pages, manuals
- apropos("package"): searches your R workspace for objects with that string
- example(findInterval): prints example associated with the function

Example data sets in R

- There are many built-in data sets in R which have been packaged up and can be accessed by the user.
- These can be useful for understanding examples, or for testing out code.

```
> data()
> data(SpikeIn)
> `?`(SpikeIn)
> matplot(t(pm(SpikeIn)), type = "l")
```


Some exercises

Example

- Find the binary operator that performs modular arithmetic.
- Display an example color palette.
- Where is there documentation on reporting bugs in R?
- How do you set your working directory?

Vectors


```
> v1 <- 1:10
> v2 <- runif(10)
> v3 <- sample(c("A", "C", "G", "T"),
+     size = 10, replace = TRUE)
> v4 <- v3 %in% c("A", "G")
> v5 <- c("foo", 2, TRUE)
> v6 <- c(2, "3")
```

- *Atomic* vectors come in 6 different modes: logical, integer, double, complex, character, and, raw.
- An *atomic* vector contains only basic types, all such types must be the same.

$$\forall i, j \in 1, \ldots \text{length}(V) \qquad \text{mode}(i) == \text{mode}(j)$$

Vectors: modes and conversion

- A vector is the most basic entity in R. To understand R, what does this code do: `length(2)`?
- Everything is a vector!
- We can get and set the mode of vectors using mode, and, storage.mode.
- We can change the mode of vectors using as.*
- A character vector is not like a C character vector. What does `length("")` return? How about `length("unam")`?
- `NA` is special. What does `length(NA)` return?
- What is a length 0 object in R?

```
> as.numeric(v6)
> as.numeric(v5)
```

Some useful vector-related functions


```
> seq(1, 20, by = 2)
> seq(0, 20, along.with = c(1:101))
> seq(0, 20, length.out = 101)
> rep(1:5, 5)
> rep(1:5, 1:5)
> rep(1:5, each = 2)
> paste("chr", 1:23)
> paste(LETTERS[1:5], rep(1:5, each = 5),
+     sep = "")
> v1 <- c(1, 2, 6, 7, 4, 3)
> l1 <- LETTERS[1:6]
> l1[sort(v1, index.return = TRUE)$ix]
```


Some quick technical details

- In R, NA is generally used to represent missing data. It will often cause a whole arithmetic expression to be evaluated as NA.
- The values -Inf, Inf and NaN have real arithmetic meaning.
- Arithmetic on decimal numbers has limited precision - but it is not a bug! Please don't report it to the R bug reporter. (See R FAQ 7.31.)

```
> sum(c(2, 3, NA, 6))
> 5/0
> 0/0
> -5/0
> c(2, 3, NA, 0)/c(3, 0, 5, 0)
> 0 * Inf
> sqrt(2) * sqrt(2) == 2
```


Indexing

There are four types of vectors which can be used to index another vector, for either subsetting or assignment:

- A logical vector
- A vector of positive integers
- A vector of negative integers
- A vector of character strings (for named vectors)

Indexing is a critical skill to cultivate in R. By proper indexing one can often make computations much more efficient and save programmer time.

Indexing


```
> v3[v4]
> v1[seq(1, 9, by = 2)]
> v1[c(5, 1, 9)]
> v1[-c(2, 4)]
> names(v1) <- LETTERS[1:10]
> v1[c("A", "D", "F")] <- 20
> v1[v1 > 5] <- 5
> v1[LETTERS[1:6]][c(2, 4)]
```

Matrices

- Matrices can be formed from a vector using the function matrix.
- More fundamentally, matrices or multidimensional arrays are nothing more than vectors with a non NULL dimension vector.
- Matrices can be subsetted just like vectors.

```
> m1 <- matrix(1:10, nrow = 5, ncol = 2)
> print(m1)
> m2 <- matrix(1:10, nrow = 5, ncol = 2,
+     byrow = TRUE)
> print(m2)
```

What is the default orientation for storing a vector as a matrix?

Matrices

```
> m1[c(1, 3, 5), ]
> m2[m2[, 1] > 3, ]
> rownames(m1) <- LETTERS[1:5]
> m1[c("A", "B"), ]
> matrix(1:100, nrow = 10)[matrix(1:10,
+     nrow = 5)]
> V <- 1:100
> array(V, dim = c(5, 5, 4))
> dim(V) <- c(5, 5, 4)
> print(V)
> rbind(1:5, 11:15)
> cbind(1:5, 11:15)
```


Matrices

```
> m3 <- matrix(rnorm(50), 25, 2)
> m3[order(m3[, 1]), ]
> ii <- order(x <- c(1, 1, 3:1, 1:4,
+     3), y <- c(9, 9:1), z <- c(2,
+     1:9))
> rbind(x, y, z)
> rbind(x, y, z)[, ii]
```


Assignments in R

R style

You may have noticed R has two forms for assigning: =, and, <- (actually there are three, but lets keep it simple). The <- form is the traditional form and is really the assignment operator. We want to try to use that anywhere we are assigning a value to a variable. The = form can also be used as the general assignment operator, however it is the only form for passing in named arguments to functions and naming elements in vectors or lists. Therefore, although in the code below both lines are the same, it is preferable to use the 1st line.

```
> A <- c(a = 1, b = 2)["a"]
> A = c(a = 1, b = 2)["a"]
```

Attributes

- For the most part attributes exist behind the scenes. A good example of this is a matrix. We can use a matrix for a long time without realizing that the only thing that distinguishes a matrix from a vector is an attribute `dim`.
- dim, names, dimnames, colnames, length, class, attributes, attr.
- Attributes can be both determined, and assigned using these operators: e.g. length can be changed by doing `length(V) <- 10`.

Statistics with R for Biologists

Background
Using R: Programming environments
Getting help
Basic R data structures
Basic control flow and function definitions in R
Reading and writing data
Lists
String processing
Packages

Attributes: examples

```
> V <- rnorm(100)
> length(V) <- 10
> print(V)
> X <- matrix(rnorm(10), nrow = 2,
+     ncol = 5)
> attributes(X)
> colnames(X)
> rownames(X)
> colnames(X) <- paste("COLUMN-",
+     1:5, sep = "")
> attributes(X)
```


Statistics with R for Biologists

Background
Using R: Programming environments
Getting help
Basic R data structures
Basic control flow and function definitions in R
Reading and writing data
Lists
String processing
Packages

Recycling: A key to understanding vectorization

- In a vectorized language when we do, for example `x = 1:10; y = 11:20; x + y` we are really doing $x[i] + y[i], i \in 1, \ldots, 10$
- A natural question to ask is what happens when `length(x) != length(y)`
- Recycling happens!
- Recycling simply repeats elements from the smaller vector until it finishes with the bigger vector. When we do `1 + c(1,2,3)` we are really recycling the vector containing 1 3 times
- Try computing, for example `c(2,3) + c(3,4,5)`, and compare that to `c(2,3) + c(3,4,5,8)`.

Statistics with R for Biologists

Background
Using R: Programming environments
Getting help
Basic R data structures
Basic control flow and function definitions in R
Reading and writing data
Lists
String processing
Packages

Recycling: A key to understanding vectorization

Always pay attention to warnings which indicate you have added vectors with "non-matching" dimensions - 9 times out of 10 you have made an error. The rules for warnings are that if you have `(length(x) %% length(y)) == 0` no warnings will be given, and otherwise you will get a warning.

Working with vectorization and recycling


```
> H <- rep("hello", 10)
> W <- rep("world!", 5)
> print(paste(H, W))
> v1 <- 1:20
> v1[c(TRUE, FALSE)]
> matrix(v1, 5, 5)
```


Matrix algebra

- R can be used as a matrix algebra calculator.
- As we have seen `c(1,2,3) * c(1,2,3)` performs element-wise multiplication.
- In order to perform matrix multiplication we do: `c(1,2,3) %*% c(1,2,3)`.

```
> X <- rnorm(100)
> dim(X) <- c(10, 10)
> Y <- t(X) %*% X
> dim(Y[, 1] %*% X[, 1:5])
```


Other matrix functions

- Other useful matrix functions are:
  - solve : X^{-1}
  - t : X^t
  - outer (%o%) : outer product of two vectors: xx^t
  - kronecker (%x%) : Kronecker product of two matrices
  - crossprod, tcrossprod : compute A^tX, compute AX^t
  - eigen : compute the eigen decomposition of a matrix

```
> Y[, 1] %o% Y[, 2]
> X <- rnorm(10)
> Y <- rnorm(10)
> W <- X %*% Y
> Z <- X %*% t(Y)
> Q <- matrix(runif(100), nrow = 20,
+     ncol = 5)
> R <- Q %*% c(1, 2, 3, 4) + rnorm(20)
```


if-else

- R offers the standard control structures `if`, and `else`.

```
> x <- 5
> if (x > 0) {
+     x <- x - 1
+     print(x)
+ } else {
+     x <- x + 1
+     print(x)
+ }
```

- What happens when we execute the following code?

```
> vec <- rnorm(10)
> if (abs(vec) > 2) {
+     1
+ }
```

if-else

- What did we expect would happen?
- In addition R offers the ifelse construct:

```
> ifelse(abs(vec) > 2, 1, 0)
```

- Also the R function switch can be useful.

```
> strand <- "add"
> ff <- switch(strand, add = function(...) {
+     Reduce("+", list(...))
+ }, subtract = function(...) {
+     Reduce("-", list(...))
+ })
> ff(1, 2, 3)
```


for and while loops

- As with most programming languages R has both a for loop and a while loop
- It used to be the case that the for loop was dreadfully inefficient and good R programming involved vectorizing everything
- We still want to vectorize as much as possible, however the for loop is not as bad as before

```
> for (i in 1:10) {
+     print(i)
+ }
> while (i > 5) {
+     print(i)
+     i <- i - 1
+ }
```


Other control-flow

- repeat, break, next
- ?Syntax

```
> i <- 1
> repeat {
+     if (i > 10)
+         break
+     print(i)
+     i <- i + 1
+ }
```


Some useful logical operators

- As in many other programming languages, comparison equality is done using ==.
- Single OR or AND operations are performed using || and &&, respectively.
- The functions any and all take the union or intersection, respectively, of a vector of booleans.
- Vectorized OR and AND operators are given by | and &.

```
> NULL || TRUE
> TRUE || NULL
> any(c(TRUE, FALSE, NULL))
> all(c(TRUE, NULL, NULL))
```

Functions: basic syntax


```
> fx <- function(x, y) {
+     x^y
+ }
> fx(1:10, 1:10)
> fx(4, 2)
> fx(1:10, 2:5)
```

- In R functions are objects - this is demonstrated by how they are defined, with the assignment operator <-.
- The last expression of a function is the default return value. Alternatively, we can return from functions using the `return` function.

Alternatives to loops in R

- Often, we want to perform a functional operation on all the rows, or all the columns, of a matrix. Rather than using a loop, the function apply is great for this.

```
> x <- cbind(x1 = 3, x2 = c(4:1,
+     2:5))
> dimnames(x)[[1]] <- letters[1:8]
> apply(x, 2, mean, trim = 0.2)
> col.median <- apply(x, 2, median)
> row.median <- apply(x, 1, function(x) {
+     median(x)
+ })
> rbind(cbind(x, Rmed = row.median),
+     Cmed = c(col.median, median(x)))
```


Reading from a file

- R can read data in a variety of different forms: csv, tab-delimited, stata, excel, relational databases, etc.
- read.table: generally a good function to start with, can be very flexible.
- readLines: good to try if you're having problems with read.table.
- scan: can be faster than read.table but more difficult to deal with multiple types.
- `help.search("read")`

For our purposes, data can be read right from the internet, e.g.: `scan("http://biostat-09.berkeley.edu/~bullard/courses/T-berkeley-08/data/jumbled.dta")`

Writing to a file

- For writing a `data.frame` to a file, the easiest function to use is write.table.
- Also very useful is the ability to save R objects or sets of R objects using the function save. You can save your whole workspace using save.image.
- To load data that you have saved, use the function load. This can be much faster than reading it in with read.table, and can save you from repeating any data cleaning.

```
> my.data <- rnorm(100)
> save(my.data, file = "saved_data.rda")
> load("saved_data.rda")
```

Example problem: Indexing

Example (Jumbled data)

A colleague approaches you hoping you might be able to help with some "data cleaning" issues. The colleague has measurements from a microarray experiment, however, due to some post-processing issues all of the intensity values have been jumbled. In the file (data/jumbled.dta) you will find the results of 30 microarray experiments where every 30th number corresponds to one array, that is: element 1 and 31 are from the same chip. First, your colleague asks you to calculate array means for the data. Furthermore, your colleague asks if you can summarize the probe intensity values into probe-set means. Each experiment has 20 probesets of length 20 which are stored in sequential order ie. 1,...,20 are measurements for one probe set.

Example problem: Matrix operations

Example (Least Squares)

Based on your success with the last assignment your colleague asks if you can help him with another problem he is having. After converting the microarray experimental data to a matrix he wishes to fit a linear regression model of the form $Y_{i,j} = \alpha_j + \beta_j \text{casestatus}_{i,j} + \epsilon_{i,j}$. Here, j is an index over probesets and i is an index over microarray experiments. He tells us that each microarray corresponds to an experimental subject who was identified as either a case or a control. $Y_{i,j}$ is the mean expression level from the previous example. The case/control vector is located in (data/case-control.dta). Fit a linear regression model and estimate both α and β for the 20 probesets. What about standard errors? p-values?

data.frame

- Data frames are what you get when you do read.table.
- For all practical purposes a data.frame is a matrix, however it has a number of disadvantages and advantages as compared to matrices.
- In general, each row of a data.frame can be thought of as a single data record.
- The different columns of a data.frame can have different types, which allows your variables to be of different types (numeric, character, factor)

```
> bases <- sample(c("A", "C", "G",
+     "T"), 8, replace = TRUE)
> obs <- runif(8)
> dta <- data.frame(obs, bases)
> names(dta)
> dta$bases
```


What is a factor?

- Factors are used to represent categorical data.
- They are a discrete set of levels which are associated with vectors of objects.
- When you read in data with read.table anything that looks like a character gets read as a factor.
- Factors are useful for generating tabular data.
- Factors are enumerations / they are stored in a very efficient manner and when applicable they should be used instead of strings.

What is a factor?


```
> myColors <- colors()[sample(1:10,
+     size = 200, replace = TRUE)]
> write.table(data.frame(age = runif(200,
+     20, 40), colors = myColors),
+     file = "tmp.dta", row.names = F)
> dta <- read.table("tmp.dta", header = TRUE,
+     stringsAsFactors = TRUE)
> class(dta[, 2])
> table(dta[, 2])
> levels(dta[, 2])
```

Often we mistake numbers/strings for factors and vice-versa we will see examples of this throughout the week.

Example problem: Reading and writing data

Example (Mystery data)

A colleague sends you a data file saying that he can't open it and hopes that you might be able to convert it to a .csv file. He believes it contains the following columns: "age", "height", "weight", "personality", and, "died." The file is located in: (data/mystery.dta).

- Read in the data using either scan, read.table, or another of the read.* variants.
- Make sure that the `data.frame` has the appropriate column names added.
- Write the data into a .csv file.
- Check that the .csv file is valid.
- Print the first couple lines and the last couple lines (head, tail might be useful)

Example problem: Unknown data format

Example (Yeast data)

Your goal is to read in data from the files (data/saccharomyces_cerevisiae.gff) and which contain chromosomal features data for the S. cerevisiae genome, as obtained from www.yeastgenome.org. You may need to try multiple functions, and look at the different function arguments carefully in order to do this. You may particularly want to think about using readLines and grep.

Lists

- As mentioned before vectors (and hence matrices) can store only "raw" values of the same type.
- Quiz: What happens here:

  ```
  > c(2, "jim", TRUE)
  ```

- R also offers the list data structure which can be used to save objects of different types and different sizes or even different dimensions.

Lists

```
> lst <- list(name = "jim", age = 29,
+     chol = rnorm(10, 160, 10),
+     test.mat = matrix(1:100, 5,
+         20))
> class(lst[1])
> names(lst)
> class(lst[[1]])
> class(lst[[4]])
```

`lst[i]` always returns a list, whereas `lst[[i]]` returns the ith element no matter what the class!

Statistics with R for Biologists

Background
Using R: Programming environments
Getting help
Basic R data structures
Basic control flow and function definitions in R
Reading and writing data
Lists
String processing
Packages

Further avoiding loops with *apply

This slide is very important. The apply family of functions are used everywhere and good R programmers rely on them heavily.

- In addition to apply we have:
  - lapply : traverses a vector or list producing a new list by applying FUN to each of its components
  - sapply : similar to lapply, however sapply does some "s" implification which often gives you results which you didn't expect (or ones which are easier to work with)
  - mapply : applies a function to a set of arguments
  - tapply : apply a function to data grouped by a particular index factor
- Also, recently R introduced some higher-order functions found in Common Lisp: Map, Filter, and, Reduce.

Statistics with R for Biologists

Background
Using R: Programming environments
Getting help
Basic R data structures
Basic control flow and function definitions in R
Reading and writing data
Lists
String processing
Packages

More Lists

```
> lst <- lapply(runif(10), function(r) {
+     if (r > 0.5)
+         rnorm(100)
+     else rnorm(100, 2)
+ })
> mat <- do.call("cbind", lst)
```

Can we do without the lapply? Try to generate the same data using ifelse. What does the call to do.call do?

Statistics with R for Biologists

Background
Using R: Programming environments
Getting help
Basic R data structures
Basic control flow and function definitions in R
Reading and writing data
Lists
String processing
Packages

Strings

- R is not the best language for string processing, however a number of natural functions are available to handle strings.
- strsplit, grep, charmatch, substr, nchar, paste
- To build strings we have:
  1. paste : vectorized function for building strings, try `paste("chr", 1:23)`
  2. sprintf
  3. as.character
  4. toString

```
> sprintf("%10.20g", 1.10001)
> sprintf("%10.1000g", pi)
> toString(1:10)
```

Working with biological strings

- Bioconductor offers the Biostrings package which has a number of functions for taking reverse-complements, complements, and a number of other functions for processing sequences of nucleotides.

Example (Mismatch probes)

In the directory "data/pm.fasta" there is a fasta file with perfect match probes (A perfect match probe perfectly targets the gene of interest, i.e. if our gene of interest is: "ACG", then our perfect match probe will be: "TGC"). Our colleague wants us to construct a new fasta file where we have both the perfect match and the mismatch probes next to one another. A mismatch probe is identical to the perfect match probe but the middle base has been changed (from our previous example, we would have: "TCC" as our mismatch probe).

More on R packages

- We will cover packages in much greater detail in a future lecture but it is important to understand them operationally.
- The R system is essentially broken down into a number of core or base packages and a runtime environment.
- We can see what we have currently in our R session using sessionInfo.
- There are two main repositories for R packages - CRAN and Bioconductor:
  - cran.r-project.org/src/contrib/PACKAGES.html
  - bioconductor.org/packages/release/BiocViews.html
- It should be stressed that the quality of many of these packages is quite low, however there are a number of great third party packages as well: XML and MASS to name two.

Packages: Seeing what's available

- In order to see what packages we have installed we can use the installed.packages.
- To see what packages are available at a CRAN mirror we can do something like available.packages.

```
> install.package("xtable")
```

For Bioconductor it is a bit different

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("GO")
```

What does the source do?