Statistics with R for Biologists

James H. Bullard
Kasper Daniel Hansen
Margaret Taub

Berkeley, California
July 7-11, 2008

R Programming

Review

- What does this code do:

```
> v <- sample(c(TRUE, FALSE),
+      size = 10, replace = TRUE)
> x <- v | FALSE == v
> v && FALSE
> g1 <- sample(c(TRUE, FALSE),
+      size = 10, replace = TRUE)
> g2 <- sample(c(TRUE, FALSE),
+      size = 10, replace = TRUE)
> g1 && g2
```

- What is the dimension of the following objects? if there is no dimension what is the length? Finally, and for massive extra credit how does R add dimensions to vectors without dimensions?

Review

```
> v <- 1:10
> dim(v %*% t(v))
> rowMeans(v %*% t(v))
> rowMeans(t(v) %*% v)
```

- My friend wants to know how many games of rock paper scissors we need to play in order to determine who is the better player. He tells me that he is 2 times as good as me and he wants to know whats is the minimum number of games necessary to play in order to be 99% sure that he is twice as good as me.

Background

- The R language provides a mix of features to support functional, procedural, and object oriented programming.
- R is both an environment for statistical computing as well as general purpose programming language.
- R has first-class functions, general data structures, international support, matrix operations, and can be extended via C and other languages.
- R does not have threads, has two systems of classes, but none with explicit syntactic support, R is untyped.
- R has built in support for statistical models incluiding a reasonably complicated formula language.

Functions: Introduction

- R is “mostly” a functional language, so we really want to understand functions - they are really the fundamental unit of code (this is true whether you call them or write them).
- In R functions are “first class” objects - this is demonstrated by how they are defined '<-'. What does this mean?
- R functions are more formally known as 'closures'.
- The last expression of a function is the default return value. Alternatively, we can return from functions using the `return` function.

Functions: Arguments

```
> x <- 2
> fxy <- function(x, y = rep(1,
+     length(x))) {
+     return(x^y)
+ }
> fxy(y = seq(2, 16, by = 2),
+     x = rep(2, 8))
> fxy(rep(2, 8), seq(2, 16, by = 2))
> fxy(rep(2, 8))
```

- All arguments to a function are “keyword” arguments
- R has “partial matching” which is something you should try to avoid depending on.

Functions: Arguments

- R has lazy evaluation, what is the length of y when the function is called? (**A million points: write a piece of code to demonstrate lazy evaluation**)
- R has call by value semantics which means that whatever arguments you pass to a function are passed by value - if you modify objects referred to by parameters then copies will be made.

Functions: Arguments

```
> collapse <- function(...) {
+     paste("(", paste(list(...),
+         collapse = ", "), ")",
+         sep = "")
+ }
```

- The special argument: '...' matches all remaining arguments.
- The function missing can be used to determine if an argument was passed in or not.

Functions: New Binary Operators

```
> "%r%" <- function(y, x) {
+     nn <- if (is.null(dim(x)))
+         rep(1, length(x))
+     else rep(1, nrow(x))
+     x <- cbind(nn, x)
+     solve(crossprod(x, x)) %*%
+         t(x) %*% y
+ }
> data(state.x77)
> state.x77[, "Murder"] %r% state.x77[,
+     "Income"]
```

Functions: New Binary Operators

- We can define new binary operators by using the special %name% syntax. Note the use of quotes around the defintion.
- Matrix multiplication and others are examples.
- What does my new binary operator do?

Functions: Anonymous

- In a vectorized language high-level 'mapping' operations are performed all of the time.
- Calculate some summary statistics on the columns of a matrix, process each element of a list, etc.
- In R it is widespread belief that we should "avoid the for loop"

```
> X <- matrix(rnorm(1e+05), nrow = 1000,
+     ncol = 1000)
> apply(X, 1, min)
> A <- apply(X, 1, function(row) {
+     sum(row > qnorm(0.975) |
+         row < qnorm(0.025))/length(row)
+ })
```

We could have done this with no loops! How?

Functions: Objects

- It is important to be familiar with functions as objects, i.e. we can pass them as arguments, store them in lists, and do much more!
- Finally, why do we call functions closures?

```
> a <- list(f1 = function(x) {
+     tmp <- quantile(x, probs = seq(0,
+         1, length = 11))
+     mean(x[x > tmp[2] & x <
+         tmp[10]])
+ }, f2 = function(x) {
+     (x - mean(x))/sd(x)
+ })
> dta <- rnorm(1000)
> a[[1]](dta)
```

Functions: Scoping

- R is a lexically scoped languge like Python or Scheme.
- Quiz: What does this program print?

```
> a <- 2
> setA <- function() {
+     a <- 4
+     print(a)
+ }
> setA()
> print(a)
```

- What about this one:

Functions: Scoping

```
> a <- 2
> setA <- function() {
+     a <<- 3
+     print(a)
+ }
> setA()
> print(a)
```

- The $<<-$ is syntactic sugar for recursing up the “environment” looking for a binding for “a” and then setting it. We can do this manually by:

Functions: Scoping

```
> a <- 2
> setA <- function() {
+     thisEnv <- parent.env(new.env())
+     parentEnv <- parent.env(thisEnv)
+     assign("a", 4, parentEnv)
+     print(a)
+ }
> setA()
> print(a)
```

In a functional language we want to return values from functions. We call functions for the return values, not their side effects

Functions: Scoping

- This is pretty esoteric, but we can understand how R (and for that matter a number of lexically scoped programming languages) work. When we make a new function we are creating a new "environment" in this environment "a" has no binding, when we ask for the value of "a" we look it up in our environment and if we don't find it we follow the link from the current environment to the parent environment.
- The special environment refered to as the "workspace" is called the `.GlobalEnv`, we can query it, assign to it, and delete from it.
- Hard Question: How do we assign a variable to the .GlobalEnv
  1. assign, get, ls, rm, mget

Hashtables

- Environments can be used as hashtables and often are. Environments don't behave like normal R objects. First, lets look at how a typical R object behaves. What does the following code print and why?

```
> a <- list()
> a[["jim"]] <- 1
> b <- a
> b
> b[["joe"]] <- 2
> a
```

- Now what happens in the case of environments?

Hashtables

```
> a <- new.env(parent = emptyenv())
> assign("jim", 2, a)
> b <- a
> ls(b)

[1] "jim"

> assign("joe", 3, b)
> ls(a)

[1] "jim" "joe"
```

- Environments don't follow the pass by values semantics or the copy on assignment rule. The environment then can be used if you want a datastructure to pass around but you never want R to copy it (*think pointer*)

Hashtables

- Bioconductor tends to use environments a lot and in this context it is generally fine to just think about them as hashtables.
- We generally want to use lists to accomplish the same purposes as environments, but we need to know that these things exist in case we want to use them. To construct a hashtable we do the following:

```
> ht <- new.env(hash = TRUE,
+     parent = emptyenv())
> lst <- list()
> tmp <- sapply(letters, function(l) {
+     assign(l, rbinom(1, size = 10,
+         prob = 0.9), ht)
+ })
> ls(ht)
```

Hashtables

```
 [1] "a" "b" "c" "d" "e" "f" "g"
 [8] "h" "i" "j" "k" "l" "m" "n"
[15] "o" "p" "q" "r" "s" "t" "u"
[22] "v" "w" "x" "y" "z"
```

- What does this code print?

```
> sapply(letters, function(l) {
+     lst[[l]] <- rbinom(1, size = 10,
+         prob = 0.9)
+ })
> names(lst)
```

- Hard Question: What happens below

Hashtables

```
> ht <- new.env(parent = emptyenv())
> mystery <- function(a, v, h) {
+     assign(a, v, h)
+ }
> mystery("jim", "B-", ht)
> ls(ht)
```

Debugging

- We will spend much more time in our lives debugging than programming so we need to get a hang of it!
- Interpreted languages such as R are generally much nicer to program with because we can try things out interactively - contrast this with C where in order to determine the value of a variable at some point in the program we will often print it
- R has a number of debugging tools, but we are going to focus on just two key functions: debug, browser

We are going to debug the following function - you can find the entire source code in "src/foo.R"

Debugging

```
> foo <- function(a, b = 10) {
+     b <- seq(1 - min(a[, 1]),
+         1 + max(a[, 1]), length = b)
+     b <- cut(a[, 1], b)
+     b <- split(a, b, drop = FALSE)
+     res <- rep(0, length(b))
+     for (i in 1:length(b)) {
+         x <- b[[i]]
+         for (j in 1:(length(x) -
+             1)) {
+             for (k in (j +
+                 1):(length(x))) {
+                 res[i] <- res[i] +
+                   (x[i, 2] -
+                     x[j, 2])^2
```

Debugging

```
+              }
+          }
+      }
+      return(res/sapply(b, function(c) nrow(c)^2))
+      return(b)
+ }
> a <- foo(a = cbind(runif(100,
+      1, 100), rexp(100, 1/10)),
+      b = 10)
```

A great way to figure out how third party functions work in R is by debugging them and then step through line by line.

Browser

- Another useful function is: browser. This function can be inserted as a means to debug code that is misbehaving.
- browser is essentially breakpoint in other languages.

```
> X <- matrix(runif(10000), 10,
+     100)
> X[4, 21] <- NA
> apply(X, 1, function(row) {
+     E <- mean(row)
+     if (is.na(E)) {
+         browser()
+     }
+     sum((row - E)^2/E)
+ })
```

traceback

- When we commit an error we can look at the call stack by calling traceback.
- We will see an example below...

OOP

- Object oriented programming is a programming paradigm which has become very popular in recent years. Object oriented programming allows us to construct modular pieces of code which can be utilized as building blocks for large systems.
- R is not a particularly object oriented system, but support exists for programming in an object oriented style.
- The Bioconductor project has pushed this style and we will need to get familiar with the object system in R in order to work effectively with Bioconductor.
- Unfortunately R has two class systems known as S3 and S4. These two systems are quite different and don't play well together.

OOP

- In both R systems the object oriented system is much more method-centric than languages like Java and Python - R's system is very Lisp-like.

S3 Classes

First we will take a look at S3 classes as they are quite prevalent in day-to-day R programming and need to be introduced in order to get a handle on some of the tricky corners of R.

- An S3 class is constructed via the following code: `class(obj) <- "class.name"`
- Essentially, a class in this setting is nothing more than an attribute that is used by special functions to perform methods dispatch.

S3 Classes

- “The greatest use of object oriented programming in R is through print methods, summary methods and plot methods. These methods allow us to have one generic function call, plot say, that dispatches on the type of its argument and calls a plotting function that is specific to the data supplied.” – R Manual

```
> cdf <- ecdf(rnorm(1000))
> class(cdf)
> plot(cdf)
> plot(rnorm(1000))
> print
```

S3 Classes

An S3 method or generic is a method like print which when called dispatches on the class attribute of its first argument. If there is no class argument or if there is no matching function for the class then we call xxx.default.

```
> vec <- rnorm(100)
> class(vec)
> getS3method("plot", "numeric")
> class(vec) <- "density"
> plot(vec)
> traceback()
```

S3 Classes

```
> jim <- list(height = 2.54 *
+     12 * 6/100, weight = 180/2.2,
+     name = "James")
> class(jim) <- "person"
> class(jim)

[1] "person"

> print(jim)
```

S3 Classes

```
$height
[1] 1.8288

$weight
[1] 81.81818

$name
[1] "James"

attr(,"class")
[1] "person"
```

S3 Classes

```
> print.person <- function(x,
+     ...) {
+     cat("name:", x$name, "\n")
+     cat("height:", x$height,
+         "meters", "\n")
+     cat("weight:", x$weight,
+         "kilograms", "\n")
+ }
> print(jim)

name: James
height: 1.8288 meters
weight: 81.81818 kilograms
```

S3 Classes

Warning: There is an error below in two places, what is it? What is the object, and what is the class?
An S3 class is really just a list with an attribute class associated with it. In order to define a specialized method for our class (such as plot, or summary) we just define a new function with xxx.jim and then when xxx is called on an object with `class(object) == "jim"` then R calls our method automatically. This is very common and thus leads an R programmer to often just call plot or summary on about anything – generally with sensible results.

Useful S3 Method Functions

1. `getS3method("print","person")` : Gets the appropriate method associated with a class, useful to see how a method is implemented. Try: `getS3method("residuals", "lm")`
2. In emacs using ESS we can often use tab to determine what methods are available under a certain generic. Try typing "plot." and then hitting tab - hopefully we will see a list of possible completions. This can be quite useful for getting help on the specific method (we will see more of this later)
3. getAnywhere : `getAnywhere("lm")`
4. methods : `methods("print")`

```
> getS3method("residuals.HoltWinters")
> getAnywhere("residuals.HoltWinters")
```

S4 Classes

- Although S3 classes can be quite useful and powerful they do not facilitate the type of modularization and type safety that a true object oriented system intends.
- For this reason S4 classes were introduced. S4 classes are much more of an object oriented system with type checking, multiple-dispatch, and inheritance.
- Again, here we want to forget about the classes and center our attention on the methods.
- In the resources directory you'll find a document describing S4 classes – this document is highly recommended reading if you want to get a deeper understanding of the S4 system.

S4 declaring a class

Lets say we want to construct a class representation for alignments. What does an alignment contain? At a minimum we need the names of the species in the alignment, the length of the alignment, the sequences themselves, and whether we are dealing with nucleotide data or amino acid data.

```
> repr <- representation(species = "character",
+     sequences = "character",
+     length = "integer", type = "character")
> setClass("Alignment", repr)
[1] "Alignment"
> A <- new("Alignment")
> species <- c("tiger", "lion",
+     "bear")
> seqs <- sapply(1:length(species),
+     function(i) paste(sample(c("A",
+         "C", "G", "T"), size = 10,
```

Initialization

```
> setMethod("initialize", "Alignment",
+     function(.Object, species,
+         sequences) {
+         .Object@species <- species
+         .Object@sequences <- sequences
+         names(.Object@species) <- NULL
+         names(.Object@sequences) <- NULL
+         if (length(sequences) !=
+             length(species)) {
+             stop("length(sequences) != length(speci
+         }
+         .Object@length <- nchar(sequences[1])
+         names(.Object@length) <- NULL
+         ss <- do.call("c",
```

Initialization

```
+             strsplit(sequences,
+                 split = ""))
+         if (all(ss %in% c("A",
+             "C", "G", "T",
+             "-")))
+             .Object@type <- "nucleotide"
+         else if (all(ss %in%
+             c("G", "A", "L",
+                 "M", "F", "W",
+                 "K", "Q", "E",
+                 "S", "P", "V",
+                 "I", "C", "Y",
+                 "H", "R", "N",
+                 "D", "T", "-")))
+             .Object@type <- "amino-acid"
```

Initialization

```
+         else stop("Unknown character in alignment")
+         return(.Object)
+     })
[1] "initialize"
```

Methods

We have already seen our first method “inititialize”, this method is called immediately after the object is instantiated and allows the programmer to custumize the initialization of an object. The show method is the S4 analog of print. Now how we access variables '@'

```
> A <- new("Alignment", species = species,
+     sequences = seqs)
> setMethod("show", "Alignment",
+     function(object) {
+         cat("Alignment of length:",
+             .Object@length,
+             "with type:", .Object@type,
+             "\n")
+     })
> show(A)
```

Where is the bug in this code?

S4 Generics

In order to make a new generic we need to call the function setGeneric.

```
> f <- function(object) standardGeneric("summarize")
> setGeneric("summarize", f)
[1] "summarize"
> setMethod("summarize", "Alignment",
+     function(object) {
+         table(do.call("c",
+             strsplit(object@sequences,
+                 split = "")))
+     })
[1] "summarize"
> summarize(A)
 A  C  G  T
 8  7  5 10
```

S4 Nuances

- R has pass-by-value semantics what does that mean for the following code:

```
> deleteSpecies <- function(alignment,
+     species) {
+     a <- which(alignment@species ==
+         species)
+     alignment@species <- alignment@species[-a]
+     alignment@sequences <- alignment@sequences[-a]
+     alignment
+ }
> B <- deleteSpecies(A, "tiger")
> A@species
```

```
[1] "tiger" "lion"  "bear"
```

Useful S4 Functions

- `showMethods("summarize")`
- `getGeneric("+")`, `getGenerics()`

Example

We want to add a simple method to our alignment class so we can add alignments. Add a new method using setMethod to allow the user to perform the following: A1 + A2 which will construct a new alignment with the species from A1 and A2. Also, make sure that the alignments are of the same length.

Replacement Methods

- As we have already seen R has a somewhat strange type of function that allows us to modify objects in place.
- It is uncommon to define new replacement functions, however they are used quite frequently in day to day programming of R.
- Two examples are: names and colnames. Type "colnames" into the R window and hit "tab", notice the function "colnames<-"?

Replacement Methods

```
> a <- matrix(1:16, nrow = 4,
+     ncol = 4)
> colnames(a) <- paste("V", 1:4,
+     sep = ".")
> colnames(a)
> point <- list(x = 1, y = 2)
> x.val <- function(x, value) {
+     x$x <- value
+ }
> "x.val<-" <- function(x, value) {
+     x$x <- value
+     return(x)
+ }
> x.val(point, 10)
> print(point)
```

Replacement Methods

```
> x.val(point) <- 10
> print(point)
```

What does the first print statement print? What about the second?

Packages

- R is broken down into a number of core or base packages and runtime environment.
- How would we see what packages are installed? What packages are available on the system?

Package Contents

A package consists of the following directories and files:

1. *DESCRIPTION* : A file describing the contents and version of the package
2. *NAMESPACE* : A description of what R and C code will be available to the loaded package
3. *R* : R source code which will be available when the package is installed depending on the NAMESPACE file
4. *src* : A directory containing C or C++ source code which can be called directly from the R code in the package
5. *man* : A directory containing the help files for a given package, these can be easily created using `prompt()`
6. *data* A directory containing data sets available once the package is loaded

Package Contents

In addition to the directories listed above packages can also contain the following directories: *demo*, *exec*, *inst*, *po*, and *tests*.
In addition the following files can be in the top level directory: *INDEX*, *configure*, *cleanup*, *LICENSE*, *LICENCE*, and *COPYING*.
These directories and files are less often used but sometimes important, to make a simple package we need only *DESCRIPTION* and *R*

DESCRIPTION

```
Package: HMM
Title: An HMM for a K state model
Version: 1.0
Author: James Bullard
Depends: R (>= 2.6.1)
Description: A package for the HMM parsing
             of tiling array data
Maintainer: <bullard@stat.berkeley.edu>
License: LGPL
```

The exact rules for this file can be found at: DESCRIPTION

Example: Making a Simple Package

We now want to build and install a simple R package to contain the code from this lecture. It is overkill to develop a package for a simple analysis, but some reasons to develop a package are:

1. keep data, documentation, and code together
2. Develop code in other languages for use in R (C, C++)
3. Easily distribute the work we have done to collaborators
4. Have a standard structure for maintaining our work

One caveat is that sometimes building the package can be more trying because we need additional tools in order to get things to work. In Mac OS X we generally just need XCode and for windows we need the R tools found at: `www.murdoch-sutherland.com/Rtools`.

Example: Making a Simple Package

1. First, see if you can install a package from source, download the following URL to a local directory and try to install the package using the command line or the GUI. xtable
2. Construct a simple package containing one function and one data set.
3. Install the package using R CMD INSTALL.
4. Test that we can access our functions and data sets using require or library.
5. use prompt to create a helpfile for one of the functions you have created.

The R Workspace

- Like most programming environments the startup of R can be controlled by your environment (e.g. environment variables, or startup files in your home directory).
- The most important environment variable is *R_LIBS*. This environment variable dictates where packages are installed, so if you are on a shared system, or a system where you do not have admin rights then you want to use this variable to control where packages are installed.
- This variable shoulb be set in your .bashrc file (or in your environment variables widget in windows)
- You can check where R will check for packages by using the .libPaths function. This function additionally allows you to add directories to the search path.

The R Workspace

- .Platform, .Machine
- Additionally, you have a file called .Rprofile which can be used to set up some initial code.

```
> dirname(.libPaths())
[1] "/Library/Frameworks/R.framework/Resources"
> basename(.libPaths())
[1] "library"
```

Examining the R session

- Often we want to know what packages / capabilities / options R is using. There are a number of relevant functions for examining the R session

```
> sessionInfo()

R version 2.7.1 RC (2008-06-20 r45965)
i386-apple-darwin9.3.0

locale:
en_US.UTF-8/en_US.UTF-8/C/C/en_US.UTF-8/en_US.UTF-8

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

Examining the R session

```
> capabilities()

    jpeg      png    tcltk
    TRUE     TRUE     TRUE
     X11     aqua http/ftp
    TRUE     TRUE     TRUE
 sockets   libxml     fifo
    TRUE     TRUE     TRUE
  cledit    iconv      NLS
   FALSE     TRUE     TRUE
 profmem    cairo
   FALSE     TRUE

> options()[c("pkgType", "device")]
```

Examining the R session

```
$pkgType
[1] "mac.binary"

$device
[1] "pdf"

> R.version

               _
platform       i386-apple-darwin9.3.0
arch           i386
os             darwin9.3.0
system         i386, darwin9.3.0
status         RC
major          2
minor          7.1
```

Examining the R session

```
year           2008
month          06
day            20
svn rev        45965
language       R
version.string R version 2.7.1 RC (2008-06-20 r45965)
```