R / Bioconductor: A Short Course

James H. Bullard
Sandrine Dudoit
Division of Biostatistics, UC Berkeley
www.stat.berkeley.edu/~bullard
www.stat.berkeley.edu/~sandrine

Cuernevaca, Mexico
January 21-25, 2008

Introduction to R

Background

- 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.

Background

- R is both an environment for statistical computing as well as general purpose programming language.
- R has first-class functions, general data structures, lazy evaluation, 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.

An Environment for Programming

- A key component for working with R, or for that matter any programming language is to get a good development environment.
- ESS (emacs speaks statistics) is the premier environment for working and developing R: `http://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

An Environment for Programming

R / Bioconductor: A Short Course
Background
Using R (Emacs/ESS, GUI)
Getting help
Reading in Data
Types
Control Structures
Functions
Lists
Factors
Probability Distributions
Looping Constructs

- 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`.

Getting Started: Overview

R / Bioconductor: A Short Course
Background
Using R (Emacs/ESS, GUI)
Getting help
Reading in Data
Types
Control Structures
Functions
Lists
Factors
Probability Distributions
Looping Constructs

1. extreme Programming
2. download and setup ESS
3. write your first program
4. using ESS, evaluate each line in turn, evaluate the entire file

Getting Started: Extreme Programming

R / Bioconductor: A Short Course
Background
Using R (Emacs/ESS, GUI)
Getting help
Reading in Data
Types
Control Structures
Functions
Lists
Factors
Probability Distributions
Looping Constructs

- A software methodology: `www.extremeprogramming.org`
- We only want to use a couple of "ideas" here:
  - pair programming
  - rapid prototyping
  - writing test cases (or in statistics, simulating data)

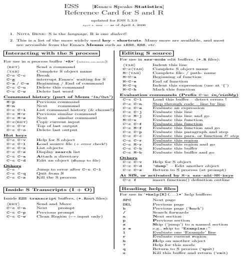
Getting Started: Installing ESS

- cd ; mkdir .emacs.d ; cd .emacs.d
- wget `ess.r-project.org/downloads/ess/ess-5.3.6.tgz`
- tar xzf ess-5.3.6.tgz
- emacs -nw ~/.emacs
- Add the line:
  `(load "~/.emacs.d/ess-5.3.6/lisp/ess-site")`

Program p1.R: Vectorized Hello World

```
> H <- rep("hello", 10)
> W <- rep("world!", 10)
> print(paste(H, W))
> X <- rnorm(100)
> Y <- rnorm(100)
> W <- X %*% Y
> Z <- X %*% t(Y)
> Q <- matrix(runif(100), nrow = 20,
+     ncol = 5)
> R <- Q %*% c(1, 2, 3, 4) + rnorm(100)
```

An ESS Reference Card



ESS [**E**macs **S**peaks **S**tatistics]
Reference Card for S and R

updated for ESS 5.3.0
April 4, 2006 — as of April 4, 2006

1. Nota Bene: S is the *language*, R is one *dialect*!
2. This is a list of the more widely used **key - shortcuts**. Many more are available, and most are accessible from the Emacs **Menus** such as iESS, ESS, etc.

Interacting with the S process

For use in a process buffer '*R*' (inferior-ess-mode):

⟨RET⟩	Send a command
⟨TAB⟩	Complete S object name
C-c C-c	Break
C-g	interrupt Emacs' waiting for S
C-a / C-e	Beginning / End of command
C-c C-u	Delete this command
C-c C-w	Delete last word

Command history (part of Menu 'In/Out')

M-p	Previous command
M-n	Next command
C-c C-l	List command history (& choose!)
C-c M-r	Previous similar command
C-c M-s	Next similar command
C-c⟨RET⟩	Copy current input
C-c C-r	Top of last output
C-c C-o	Delete last output

Hot keys

C-c C-v	Help for S object
C-c C-l	Load source file (+ error check!)
C-c C-x	List objects
C-c C-s	Display search list
C-c C-a	Attach a directory
C-c C-d	Edit an object (dump to file)

Others

C-c `	Jump to error after C-c C-l
C-c C-q	Quit from S
C-c C-z	Kill the S process

Inside S Transcripts (I + O)

Inside ESS transcript buffers, (*.Rout files):

⟨RET⟩	Send and Move
C-c C-n	Next prompt
C-c C-p	Previous prompt
C-c C-w	Clean Region (⟼ input only)

Editing S source

For use in ess-mode edit buffers, (*.R files):

⟨TAB⟩	Indent this line
C-c⟨TAB⟩	Complete S object name
M-⟨TAB⟩	Complete file- / path- name
M-C-a	Beginning of function
M-C-e	End of function
M-C-q	Indent this expression (use at '{')
M-C-h	Mark this function

Evaluation commands (Prefix C-u: *in/visibly*)

C-c C-l	Load this buffer – detect errors !
C-c C-n	Step through code – line by line
C-c C-e	Evaluate an expression
C-c C-j	Evaluate this line
C-c M-j	Evaluate this line and go
M-C-x	Evaluate this function
C-c C-f	Evaluate this function
C-c M-f	Evaluate this function and go
C-c C-p	Evaluate this paragraph and step
C-c C-c	Evaluate this para. or function *& step*
C-c C-r	Evaluate this region
C-c M-r	Evaluate this region and go
C-c C-b	Evaluate this buffer
C-c M-b	Evaluate this buffer and go

Others

C-c C-v	Help for S object
C-c C-d	"dump" – Edit another object
C-c C-z	Return to S process (at prompt)

At SfS, or activated by M-x ess-add-MM-keys

C-c f	insert function() definition outline

Reading help files

For use in '*help[R](...)*' help buffers:

SPC	Next page
DEL	Previous page
b	Previous page ('**b**ack')
/	Search forwards
n	**N**ext section
p	**P**revious section
s	**S**kip ('jump') to a named section
s e	e.g., skip to "Examples:"
l	Evaluate one 'Example' line
r	Evaluate current **r**egion
h	**H**elp on another object
?	Help for this mode
q	Return to S process ('**q**uit)
x	Kill this buffer and return ('exit)

ESS-Emacs Useful Commands

- Emacs
  - Ctl-x Ctl-f : open file
  - Ctl-x Ctl-s : save file
  - Ctl-x b : switch to buffer
  - Ctl-x o : switch from one buffer to the next when you have more than one open (Ctl-x 2 to split screen)
  - Ctl-x k : kill buffer
  - Ctl-k : kill a line
  - M-w : copy
  - Ctl-w : cut
  - Ctl-y : paste
  - Ctl-spacebar : set mark
- ESS
  - Ctl-c Ctl-n : eval a line and then goto the next line
  - Ctl-c Ctl-v : help on a R function

Getting Help

- Within ESS C-c C-v
- Within R

```
> help("lm")
> `?`(help)
> help("for")
> library(help = "stats")
> help(package = "stats")
> help.search()
> help.start()
> RSiteSearch("multivariate normal")
> apropos("package")
> example(findInterval)
```

The R environment

```
> sessionInfo()
> .libPaths()
> options()
> R.version
```

- The definitive guide to this is: ?Startup.
- Essentially, we have two types of files:
  - environment files: set/unset environment variables of R (R_LIBS= /.R-packages)
  - R code: set various options, do things on startup/shutdown
- We need to set the R_LIBS environmental variable in .bashrc
  - .Renviron is read when you start R interactively.
  - .bashrc can be used to set environment variables and the R_LIBS environment variable is used when you do an "R CMD ..." from the shell.
- Let's install some bioconductor packages which we might need.

Data Sets

- R can read data in a variety of different forms: csv, tab-delimited, stata, excel, relational databases, etc.
- Data can also be packaged up and presented to the user in a data package:

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

- readLines
- scan
- read.table
- help.search("read")

Example

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.
- Write the data into a .csv file.
- Check that the .csv file is valid.
- Make sure that the new file has the appropriate column names added.
- Print the first couple lines and the last couple of lines (head, tail might be useful)

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)
```

Attributes

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

- 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."

Attributes

- dim, names, colnames, length, class, attributes, attr.
- length can be changed i.e. `length(V) = 10`.

R Style

Attributes

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 variable a value. 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"]
```

Matrices

- Matrices or multidimensional arrays are nothing more than vectors with a non NULL dimension vector.
- This affects things like printing and matrix algebra.

```
> V <- runif(100)
> dim(V) <- c(2, 5, 10)
> print(V)
> V2 <- array(V, dim = c(2, 5, 10))
> all(V2 == V)
> dim(V) <- NULL
> print(V)
```

Indexing

Key

- Indexing is a critical skill to cultivate in R.
- By proper indexing we can often make computations much more efficient as well as saving programmer time.

```
> V <- 1:100
> odds <- V[seq(1, 99, by = 2)]
> matrix(V, nrow = 10)[1, ]
> matrix(V, nrow = 10)[, 1]
> matrix(V, nrow = 10)[matrix(1:10,
+     nrow = 5)]
```

Indexing

Example

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. 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. Please output a file with 20 columns and 30 rows with the mean expression level for each of the probe sets.

Matrix Algebra

- R can be used as a matrix algebra calculator.
- As we have seen `c(1, 2, 3) * c(1, 2, 3)` performs elementwise 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])
> Y[, 1] %o% Y[, 2]
```

What order did it turn X into a matrix. Try the following code and try to understand the rule.

```
> x <- 1:16
> dim(x) <- c(4, 4)
```

- 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

Example (Least Squares)

Based on your success with the last two assignments 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 20 probesets. What about standard errors? p-values?

if-else

What happens when we execute the following code?

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

- What did we expect would happen?
- R offers the standard control structures `if`, and `else`.
- In addition R offers the ifelse construct:

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

- Also the R function switch can be useful.

if-else

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

Vectorization

if-else

Many R functions are vectorized. However, there are a number of exceptions. For certain functions it wouldn't make sense, and for others there good reasons why it is not vectorized. The "if ... else" construct in R is not a function call (although you might say that it kind of looks like one), and in this regard it is important to keep in mind that when you are calling a function you are getting what you expect.

Recycling: A key to understanding vectorization

- In a vectorized language when we do `x = 2; y = 3; x + y` we are really doing $x[i] + y[i], i \in 1, \dots \max\{|x|, |y|\}$
- 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
- Compare that to `c(2,3) + c(3,4,5)` and compare that to: `c(2,3) + c(3,4,5,8)`

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 the lengths `(length(x) %% length(y)) == 0` no warnings, otherwise warning.

Functions: Syntax

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

- In R functions are "first class" objects - this is demonstrated by how they are defined '<-'. A "first class" function is not like a function in C or perl. Loosely, when we talk about first class functions it means that they can be treated more like data, i.e. passed into functions, stored in data structures, and returned from functions.

Functions: Syntax

- The last expression of a function is the default return value. Alternatively, we can return from functions using the `return` special form - NB: `return` is called like a function not like a special form such as `break`.

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 lazy evaluation, what is the length of y when the function is called?
- What R does is match the arguments - you can see match.arg for more details. The matching is done partially, this is really a bad design choice/fact of life with R and thus I would suggest never counting on partial matching.

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"]
```

- 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

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(10000), nrow = 100,
+     ncol = 100)
> 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!

Functions: Wrapup

- 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!

```
> 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)
```

*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
  - mapply : like map in scheme, less used than lapply, apply
- Also, recently R introduced some higher-order functions found in Common Lisp: Map, Filter, and, Reduce.

Searching for Outliers

Example

In our previous analysis we fit a linear regression model to each of the patients to search for important genes. We are concerned about the presence of outliers however, and we wish to remove them and re-fit our linear models to see if our results change. Write a function which takes a data-set with the same dimensions as the result of exercise 2, i.e. $n_{\text{people}} \times n_{\text{genes}}$ and removes outliers. Define an outlier as any probeset which is in the lower 5% or upper 5% quantile.

Lists

- As mentioned before vectors can store only "raw" values of the same type.
- R also offers the list data structure which can be used to save objects of different types.
- lists inherit a lisp-like style from some of the roots of R so it is important to keep this in mind when accessing the elements of the list.

```
> lst <- list(name = "jim", age = 29,
+     chol = rnorm(10, 160, 10))
> class(lst[1])
> class(lst[[1]])
```

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

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

data.frame

- A matrix-like extension to lists is the data.frame.
- Data frames are what you get when you do read.table
- Really a data.frame is a list of columns that can be accessed like a matrix - for all practical purposes it is is a matrix, however it has a number of disadvantages and advantages as compared to matrices.

```
> bases <- c("A", "C", "G", "T")[1 +
+     rbinom(100, prob = 0.5, size = 3)]
> dta <- data.frame(runif(100), bases,
+     stringsAsFactors = TRUE)
```

What is a Factor?

- Factors are loosely like enumerations in other language.
- 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, we will also want to explore the function cut.

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

Distributions

- We want to do statistics!
- We need to be able to generate random numbers according to distributions, compute probabilities, quantiles, densities.

d{distribution}	density
p{distribution}	probability
r{distribution}	random variates
q{distribution}	quantiles

In addition to these functions we have one of the most important functions sample which draws from a multinomial distribution with or without replacement.

```
> x <- rchisq(100, df = 10)
> pchisq(x, df = 1)
> dchisq(x, df = 10)
> qchisq(seq(0, 1, length = 10),
+     df = 1)
```

Random Numbers

Example (Simulating an alignment)

We would like to simulate a “hypothetical” two sequence alignment from humans and chimpanzees. It is estimated that humans and chimpanzees diverged aproximately X years ago. The model of of evolution will be the Jukes-Cantor model of evolution - specified entirely by the instantaneous rate matrix Q. Simulate a two species alignment for a number of different times X. How can we ensure that we have done the correct thing? The best way to do this is to write a function which takes as two arguments: t and N for time and number of bases to simulate - the function should return a matrix with 2 rows and N columns.

Random Numbers

$$\mathbf{Q} = \begin{pmatrix} -.75 & .25 & .25 & .25 \\ .25 & -.75 & .25 & .25 \\ .25 & .25 & -.75 & .25 \\ .25 & .25 & .25 & -.75 \end{pmatrix}$$

We will define the matrix exponential as in equation (1). However if we can write Q as $Q = UDU^{-1}$ i.e. if the matrix Q is diagonalizable then we can compute the matrix exponential more directly; as shown in: (2).

$$P(t) = e^{Qt} = \sum_{n=0}^{\infty} \frac{Q^n t}{n!} \quad (1)$$

$$e^{Qt} = Ue^{Dt}U^{-1} \quad (2)$$

Random Numbers

- As a reminder: Here U is a matrix of the eigen vectors and D is a matrix with the eigen values along the diagonals. What we have is a function of time for the conditional probability of a base given an ancestral base. That is, we plug in a time and we get a new probability distribution - this is all we need to simulate.
- There a lot of technical details here - we will ignore them. Our goal is to simulate data. Firstly, we can check quite quickly that the matrix is indeed diagonalizable by simply trying to compute the eigen vectors and eigen values (hint: solve).

for and while loop

- 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 inneficient and good R programming involved vectorizing everything.
- We still want to vectorize as much as possible, however the for loop is not as bad in newer versions of R.

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

Other control-flow

- `repeat`, `break`, `next`
- ?Syntax

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

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)
```

Biostrings

- 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

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).

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 in all cases.
- 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.

- 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 method dispatch.
- “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

S3 Classes

```
> class(ecdf(rnorm(1000)))
> plot(ecdf(rnorm(1000)))
> plot(rnorm(1000))
> print
```

An S3 method or generic is a method like print which when called dispatches on the class attribute of its first argument.

S3 Classes

```
> jim <- list(height = 2.54 * 12 *
+     6/100, weight = 180/2.2, name = "James")
> class(jim) <- "person"
> class(jim)
> print(jim)
> print.person <- function(x, ...) {
+     cat("name:", x$name, "\n")
+     cat("height:", x$height, "meters",
+         "\n")
+     cat("weight:", x$weight, "kilograms",
+         "\n")
+ }
> print(jim)
```

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 generally enforces.
- 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 two documents describing S4 classes. These should be looked at during the week.

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", representation = repr)

[1] "Alignment"

> A <- new("Alignment")
```

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("n species must = n sequences.")
+         .Object@length <- nchar(sequences[1])
+         names(.Object@length) <- NULL
+         ss <- do.call("c", strsplit(sequences,
+             split = ""))
```

Initialization

```
+         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"
+         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", names(seqs[1:10]),
+     seqs[1:10])
> print(A)
> setMethod("show", "Alignment",
+     function(object) {
+         cat("Alignment of length:",
+             .Object@length, "with type:",
+             .Object@type, "\n")
+     })
```

Where is the bug in this code?

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, "pm-2")
```

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?

Debugging

- We will spend much more time in our lives debugging than programming we need to get good at 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
- Often functions will come from a number of interactive operations which we do frequently enough to warrant a name
- R has a number of debugging tools, but we are going to focus on just two key functions: debug and browser

Debugging

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

```
> 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] +
```

Debugging

```
+                   (x[i, 2] - x[j,
+                     2])^2
+             }
+         }
+     }
+     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)
```

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.
- As we have seen before we can see what we have currently in our R session using sessionInfo.
- There are two main repositories for R packages - CRAN and Bioconductor:
  - http://cran.r-project.org/src/contrib/PACKAGES.html
  - http://bioconductor.org/packages/release/BiocViews.html
- It should be stressed that the quality of many of these package is quite low, however there are a number of great third party packages as well: XML and MASS to name two.

Packages: Seeing Whats 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?