A beginner's guide to solving biological problems in R

Day 1 Morning

Robert Stojnić (rs550) Laurent Gatto (lg390) Rob Foy (raf51)
John Davey (jd626)

Course materials:
`http://logic.sysbiol.cam.ac.uk/teaching/Rcourse/`
Slides by Ian Roberts and Robert Stojnić

Day 1 Schedule

- The R environment and basics
  - Where to get R
  - Brief introduction to essential R
  - R help, scripting and packages

Morning coffee

- Objects and data types
  - Learn how to input and manipulate data

Lunch

- Introduction to essential R commands
  - Base functions
  - Read and write data

Afternoon coffee

- R for data analysis
  - Statistical tests and maths support

What's R?

- A statistical programming environment
  - based on S
  - Suited to high level data analysis
- Open source & cross platform
- Extensive graphics capabilities
- Diverse range of add-on packages
- Active community of developers
- Thorough documentation

The environment and basics

Screenshot

Various platforms supported

- Release 2.15.0 (March 2012)
  - Base package
  - Contributed packages (general purpose extras)
  - Over 4000 packages available
- Windows
  - `http://www.stats.bris.ac.uk/R/bin/windows/base/R-2.11.1-win32.exe`
- Mac OS (10.2+)
  - `http://cran.r-project.org/bin/macosx/`
- Linux
  - `http://cran.r-project.org/bin/linux/`
- Executed using command line, or a graphical user interface
  - Will demonstrate both, and use all-platform GUI, RStudio

Getting Started

- R is a program which, once installed on your system, can be launched and is immediately ready to take input directly from the user
- There are two ways to launch R:
  1. From the command line (particularly useful if you're quite familiar with Linux)
  2. As an application called RStudio (very good for beginners)

Prepare to launch R

From command line

- To start R in Linux we need to enter the Linux console (also called Linux terminal and Linux shell)
- To start R, at the prompt simply type:

  ```
  $ R
  ```

- If R doesn't print the welcome message, call us to help!

Prepare to launch R

Using RStudio

- To launch RStudio, find the Rstudio icon in the menu bar on the left of the screen and double-click

The Working Directory (wd)

- Like many programs R has a concept of a working directory (wd)
- It is the place where R will look for files to execute and where it will save files, by default
- For this course we need to set the working directory to the location of the course scripts
- At the command prompt in the terminal or in RStudio console type:

  ```
  > setwd("R_course/Day_1_scripts")
  ```

- Alternatively in RStudio use the mouse and browse to the directory location
- Tools - Set Working Directory - Choose Directory...

Basic concepts in R

command line calculation

- The command line can be used as a calculator. Type:

```
> 2 + 2
[1] 4

> 20/5 - sqrt(25) + 3^2
[1] 8

>sin(pi/2)
[1] 1
```

- Note: The number in the square brackets is an indicator of the position in the output. In this case the output is a vector of length 1 (i.e. a single number). More on vectors coming up...

Basic concepts in R

variables

- A variable is a letter or word which takes (or contains) a value. We use the assignment 'operator', <-

```
> x <- 10
> x
[1] 10
> myNumber <- 25
> myNumber
[1] 25
```

- We can perform arithmetic on variables:

```
> sqrt(myNumber)
[1] 5
```

- We can add variables together:

```
> x + myNumber
[1] 35
```

Basic concepts in R

variables

- We can change the value of an existing variable:

```
> x <- 21
> x
[1] 21
```

- We can modify the contents of a variable:

```
> myNumber <- myNumber + sqrt(16)
[1] 29
```

Basic concepts in R

functions

- **Functions** in R perform operations on **arguments** (the input(s) to the function). We have already used **sin(x)** which returns the sine of **x**. In this case the function has one argument, **x**. Arguments are *always* contained in parentheses, i.e. curved brackets **()**, separated by commas.
- Some other common functions: **floor()**, **sum()**, **max()**, **mean()**
- Try these:

```
> floor(3.142)
[1] 3
> sum(3, 4, 5, 6)
[1] 18
> max(3, 4, 5, 6)
[1] 6
> mean(3, 4, 5, 6)
[1] 3
```

- Something has gone wrong with the last function. We need to understand more about vectors...

Basic concepts in R

vectors

- The function **c()** *combines* its arguments into a **vector**

```
> x <- c(3, 4, 5, 6)
> x
[1] 3 4 5 6
```

- As mentioned, the square brackets **[]** indicate position within the vector. We can extract individual elements by using the **[]** notation.

```
> x[1]
[1] 3
> x[4]
[1] 6
```

- We can even put a vector inside the square brackets.

```
> y <- c(2, 3)
> x[y]
[1] 4 5
```

- We can now solve the problem from the previous slide:

```
> mean(x)
[1] 4.5
```

Basic concepts in R

vectors

- There are a number of shortcuts to create a vector. Instead of:

```
> x <- c(3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
```

- write

```
> x <- 3:12
```

- Using the **seq()** function...

```
> x <- seq(2, 10, 2)
> x
[1] 2 4 6 8 10
> x <- seq(2, 10, length.out=7)
> x
[1] 2.00000 3.33333 4.66667 6.00000 7.33333 8.66667 10.00000
```

- or the **rep()** function

```
> y <- rep(3, 5)
> y
[1] 3 3 3 3 3
> y <- rep(1:3, 5)
> y
[1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

Basic concepts in R

vectors

- We have seen some ways of extracting elements of a vector. We can use these shortcuts to make things easier (or more complex!)

```
> x <- 3:12
> x[3:7]
[1] 5 6 7 8 9
> x[seq(2, 6, 2)]
[1] 4 6 8
> x[rep(3, 2)]
[1] 5 5
```

- We can add an element to a vector

```
> y <- c(x, 1)
> y
[1] 3 4 5 6 7 8 9 10 11 12 1
```

- We can glue vectors together

```
> z <- c(x, y)
> z
[1] 3 4 5 6 7 8 9 10 11 12 3 4 5 6 7 8 9 10 11 12 1
```

Basic concepts in R

vectors

- We can remove element(s) from a vector

```
> x <- 3:12
> x[-3]
[1] 3 4 6 7 8 9 10 11 12
> x[-(5:7)]
[1] 3 4 5 6 10 11 12
> x[-seq(2, 6, 2)]
[1] 3 5 7 9 10 11 12
```

- Finally, we can modify the contents of a vector

```
> x[6] <- 4
> x
[1] 3 4 5 6 7 4 9 10 11 12
> x[3:5] <- 1
> x
[1] 3 4 1 1 1 4 9 10 11 12
```

- Remember! **Square** brackets for indexing **[]**, **parentheses** for function arguments **()**.

Basic concepts in R

vector arithmetic

- When applying all standard arithmetic operations to vectors, application is element-wise.

```
> x <- 1:10
> y <- x*2
> y
[1] 2 4 6 8 10 12 14 16 18 20
> z <- x^2
> z
[1] 1 4 9 16 25 36 49 64 81 100
```

- Adding two vectors

```
> y + z
[1] 3 8 15 24 35 48 63 80 99 120
```

- Vectors don't have to be the same length (what's this?)...

```
> x + 1:2
[1] 2 4 4 6 6 8 8 10 10 12
```

- but that doesn't always work:

```
> x + 1:3 (...?)
```

Writing scripts with Rstudio

Typing lots of commands directly to R can be tedious. A better way is to write the command to a file and then load it into R.

- Click on **File** - **New** in RStudio
- Type in some R code, e.g.

```
x <- 2 + 2
print(x)
```

- Click on **Run** to execute the **current line**, and **Source** to execute the **whole script**.
- Sourcing can also be performed manually with `source("myScript.R")`

Getting Help

- To get help on any R function, type ? followed by the function name. For example:

  ```
  > ?seq
  ```

- This retrieves the syntax and arguments for the function. It also tells you which **package** it belongs to. There will typically be example usage.
- If you can't remember the exact name type ?? followed by your guess. R will return a list of possibles.

  ```
  > ??rint
  ```

Interacting with the R console

- R console symbols
  - end of line
    - Enables multiple commands to be placed on one line of text
  - # comment
    - indicates text is a comment and not executed
  - + command line wrap
    - R is waiting for you to complete an expression
- **Ctrl-c** or **escape** to clear input and try again
- **Ctrl-l** to clear window
- Press q to leave help (using R from the terminal)
- Use the **TAB key** for command auto completion
- Use **up and down arrows** to scroll through the command history

R packages

- R comes ready loaded with various libraries of functions called **packages**, e.g. the function **sum()** is in the **base** package and **sd()**, which calculates the standard deviation of a vector, is in the **stats** package
- There are 1000s of additional packages provided by third parties, and the packages can be found in numerous server locations on the web called **repositories**
- The two repositories you will come across the most are
  - **The Comprehensive R Archive Network (CRAN)**
  - Bioconductor
- CRAN is mirrored in many locations. Set your local mirror in RStudio using Tools - Options, and choose a CRAN mirror
- Set the Bioconductor package download tool by typing:

  ```
  > source ("http://bioconductor.org/biocLite.R")
  ```
- Bioconductor packages are then loaded with the **biocLite()** function:

  ```
  > biocLite("PackageName")
  ```

R packages

- 3900+ packages on CRAN:
  - Use CRAN search to find functionality you need:
    `http://cran.r-project.org/search.html`
  - Or, look for packages by theme:
    `http://cran.r-project.org/web/views/`
- 550+ packages in Bioconductor:
  - Specialised in genomics:
    `http://www.bioconductor.org/packages/release/bioc/`
- **Other repositories:**
- 1000+ projects on R-forge:
  `http://r-forge.r-project.org/`
- R graphical manual:
  `http://bg9.imslab.co.jp/Rhelp/`

Exercise: Install Packages Matrix and aCGH

- Matrix is a CRAN extras package
  - Use **install.packages()** function...
    `install.packages("Matrix")`
  - or in RStudio goto Tools - Install Packages... and type the package name
- aCGH is a Bioconductor package (`www.bioconductor.org`)
  - Use **biocLite()** function
    `biocLite("aCGH")`
- R needs to be told to use the new functions from the installed packages
  - Use **library()** function to load the newly installed features

    ```
    library("Matrix") # loads matrix functions

    library("aCGH") # loads aCGH functions
    ```
  - `library()`
    - Lists all the packages you've got installed locally

MORNING COFFEE

Morning coffee

OBJECTS

Objects

R stores different types of data

- Types of data
  - Logical, integer, character, floating point
- Data is stored in objects
  - Vectors, data frames, matrices, arrays, lists
  - Vector is the most basic object
- Most appropriate type and value is determined by R syntax

```
a <- 10     # takes the value of number 10
a <- "10"   # takes the value of characters "10"
a <- b      # takes the value of variable b
a <- "b"    # takes the value of character "b"
```

Data types and storage modes

- R creates appropriately sized object variables to hold data
- Objects are vectors ... they have a length dimension ...
  - Vectors support vector arithmetic, which is R's big thing
- The 'mode of storage' used is determined by the 'data type'
- *Mode* is one of
  - logical, numeric, or character
- Standard data *types*
  - **Logical**
    - (TRUE or FALSE)
  - **Integer**
    - (e.g. whole numbers -2 / +2 Billion)
  - **Double / floating point**
    - (e.g. fractions, scientific expressions)
  - **Character string - always in quotes!**
    - (e.g. 4 is a number, "4" or "abc" are characters)

ADD BOX

Exercise

Data types & storage modes

- Create vectors `i`, `l`, `s` and `d` of length 20 for each *data type* integer, logical, character and double
- Examine their storage mode & confirm data type
  - `mode(...)`
  - `typeof(...)`
- Tips:
  - you can do this by manually assigning values, or by generating them (see the functions below)
  - `sample(20)` will create a vector of 20 random integer values
  - R special character object `letters` is 'builtin', you can subset it
  - `runif(20)` will generate 20 random uniformly distributed 'double' values between 0 and 1
  - relationshps are 'tests' and return logical values (e.g. $2 > 1$ is TRUE, but $5 < 1$ is FALSE)
    - `1 <- i >= 10 # test if 'i' is greater or equal to 10`

Solution

```
> i <- sample(20)
> i
[1]  6  1 14  8  5 10 18 12  2 11 16  3  4 20  9  7 19 13 15 17
> s <- letters[i]]
> s
[1] "f" "a" "n" "h" "e" "j" "r" "l" "b" "k" "p" "c" "d" "t" "i" "g" "s" "m" "o"
[20] "q"
> l <- i >= 10
> l
[1] FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE
[13] FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
> d <- runif(20)
> d
 [1] 0.292480127 0.586057481 0.812791710 0.189530051 0.634724719 0.882981055
 [7] 0.358151866 0.003088843 0.900198085 0.905947217 0.032231749 0.330297215
[13] 0.398237377 0.832774598 0.048503020 0.965920822 0.357567181 0.688554482
[19] 0.728918707 0.477420883
> mode(i)
[1] "numeric
> typeof(i)
[1] "integer"
> mode(l)
[1] "logical"
> typeof(l)
[1] "logical"
> mode(s)
[1] "character"
> typeof(s)
[1] "character"
> mode(d)
[1] "numeric"
> typeof(d)
[1] "double"
```

ADD COMMENTS

Storage modes & data types

- Data types - why care?
  - May get an undesired result if calculations are between numbers stored as different types
  - R will coerce data types when calculations between differing types are forced
    - If the operation is inappropriate, the calculation will fail:
      `> 2 + "2"`
      will fail as we cannot add a character string to an integer!

R Objects - vectors

`c(...)` or `vector(..., mode=...,length=...)`

- Vectors
  - one-dimensional sequence of data items of only one data type

```
> a <- c(1, 2, 3, 4, 5)
> typeof(a)
[1] "double"
> aa <- c(1, "2", 3)
> typeof(aa)

[1] "character"
```

- Vector arithmetic is a fundamental R concept

```
> a <- 1:10 ; b <- 101:110 ; c <- a + b
> a
[1]  1  2  3  4  5  6  7  8  9 10
> b
[1] 101 102 103 104 105 106 107 108 109 110
> c
[1] 102 104 106 108 110 112 114 116 118 120
```

- Arithmetic operation occurs between corresponding items of the two vectors. If vectors are different lengths, shortest is recycled.

R Objects

`data.frame(var1=..., var2=..., etc.)`

- Data frames
  - multiple columns of vectors that form a table
  - columns can be of different data types

```
> a <- 1:10  # shortcut for numbers 1 to 10
> b <- letters[1:10]  # shortcut for letters
> c <- month.name[1:10] # shortcut for month names
> d <- data.frame(a, b, c, stringsAsFactors=FALSE) # data frame assignment
> d
   a    b   c
1      1    a   January
2      2    b  February
3      3    c     March
4      4    d     April
5      5    e       May
6      6    f      June
7      7    g      July
8      8    h    August
9      9    i September
10    10    j   October
```

- Column names can be changed
  `colnames(d) <- c("a", "b", "c")`
- Address columns with $ notation
  `d$a # returns only the column a (as a vector)`

R Objects

`matrix(..., ncol=..., nrow=...)`

- Matrices
  - Like data frames, but for a single data type

```
> e <- matrix(1:10, nrow=5, ncol=2)
> e
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
```

  - Matrices are usually associated with numeric data, and usual operations $+$, $-$, $*$, ... work on whole matrices as well

Indexing data frames, matrices & arrays

- Works just like vectors, only in 2 dimensions
  `object [ rows , columns ]`

```
a <- data.frame(1:10, letters[1:10], month.name[1:10], stringsAsFactors=F
names(a) <- c("numbers","letters","months")
a
   numbers letters    months
1        1       a   January
2        2       b  February
3        3       c     March
4        4       d     April
5        5       e       May
6        6       f      June
7        7       g      July
8        8       h    August
9        9       i September
10      10       j   October
```

ADD BOX, FIX WRAP

R Objects

`list(name1=obj1,name2=obj2,...)`

- Lists
  - Store collections of R objects
    - List members (items) can be of any R object, or data type.

```
a <- 1:10
b <- matrix(runif(100),ncol=10,nrow=10)
c <- data.frame(a,month.name[1:10])

myList<-list( ls.obj.1=a, ls.obj.2=b, ls.obj.3=c)
summary(myList)
names(myList)
```

  - Using dollar notation($), we can address list items directly, by name, as with columns of a data frame `myList$ls.obj.1`
  - Alternatively, `myList[[1]]` to get first item in the list

R Objects

factor(obj, levels=..., labels=...)

- Factors
  - Factors store categorical data
  - categorical data is usually expressed in levels
    - They are especially useful where repetition is found
    - Exercise may take place on any day of the week
      `The factor is exercise, and levels are week day names`
    - A gene may be deleted, lost, normal, gained or amplified
      `The factor is gene copy number, and levels are -2, -1, 0, 1, 2`
  - Factors require a good understanding of data dependency
  - Experiments often have multiple explanatory variables, and it is interesting to observe interactions between them
  - Factors are similar to 'enumerated data types' in other languages

Operators

- arithmetic
  `+, -, *, /,`
- comparison
  `<, >, =<, >=, ==, !=`
- logical
  `!, &, |, xor`
  (not, and, or, exclusive or)

ADD POWER SYMBOL

Advanced indexing / subscripting vectors

- Indexing / subscripting
  - The process of referencing a particular data value stored in an R object
- Indexing can be achieved either with numbers or logicals, e.g.:

```
> s <- letters[1:5]
> s[c(1,3)]
[1] "a" "c"
> s[c(TRUE, FALSE, TRUE, FALSE, FALSE)]
[1] "a" "c"
```

- Or, with expression resulting in either numbers or logicals:

```
> a <- 1:5
> s[a[1:4]] # range expression, answer is a to d
> s[a<3] # first 2 items, returns a, b
> s[a>1 & a<3] # answer is 2, returns b
```

Exercise: objects & indexing

FIX OVERFLOW

Construct your phonebook from 5 vectors (make up the data).
First name
Character vector: `firstName`
Second name
Character vector: `secondName`
Telephone number
Numeric vector: `telNumber`
Listed in directory
Logical vector: `notListed`
`i.e.  TRUE or FALSE`
Full name
Character vector: `fullName`

```
fullName <-
```

Use your phonebook to extract the following data items

1. Telephone numbers of people with second names beginning with letters below M
   `phoneBook$secondName < "M"`
2. Telephone numbers of people not listed in a directory
   `phoneBook$notListed == TRUE`
3. Telephone numbers of all odd row people

One solution

```
> firstName<-c("Adam","Eve","John","Mary","Peter","Paul","Luke",
+ "Matthew","David","Sally")
> length(firstName)
[1] 10
> secondName<-c("Tiny","Large","Small","Davis","Thumb","Daniels",
+ "Edwards","Smith","Howkins","Dutch")
> length(secondName)
[1] 10
> telNumber<-c(111111,222222,333333,444444,555555,666666,777777,888888,121212,232323)
> length(telNumber)
[1] 10
> notListed<-c(TRUE,FALSE,TRUE,FALSE,TRUE,FALSE,TRUE,FALSE,TRUE,FALSE)
> length(notListed)
[1] 10
> phoneBook<-data.frame(firstName,secondName,paste(firstName,secondName),telNumber,notListed,
                        stringsAsFactors=FALSE)
> names(phoneBook)<-c("First_Name","Second_Name","Full_Name","Tel_Number","Not_Listed")
> phoneBook
   First_Name Second_Name     Full_Name Tel_Number Listed
1        Adam        Tiny     Adam Tiny     111111   TRUE
2         Eve       Large      Eve Large     222222  FALSE
3        John       Small     John Small     333333   TRUE
4        Mary       Davis     Mary Davis     444444  FALSE
5       Peter       Thumb    Peter Thumb     555555   TRUE
6        Paul     Daniels   Paul Daniels     666666  FALSE
7        Luke     Edwards   Luke Edwards     777777   TRUE
8     Matthew       Smith Matthew Smith     888888  FALSE
9       David     Howkins David Howkins     121212   TRUE
10      Sally       Dutch    Sally Dutch     232323  FALSE
```

ADD BOXES

LUNCH

Lunch

R Commands & flow control

Commands