A beginners guide to solving biological problems in R

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

Course material:
http://logic.sysbiol.cam.ac.uk/teaching/Rcourse/

Original slides by Ian Roberts and Robert Stojnić

Day 1 schedule

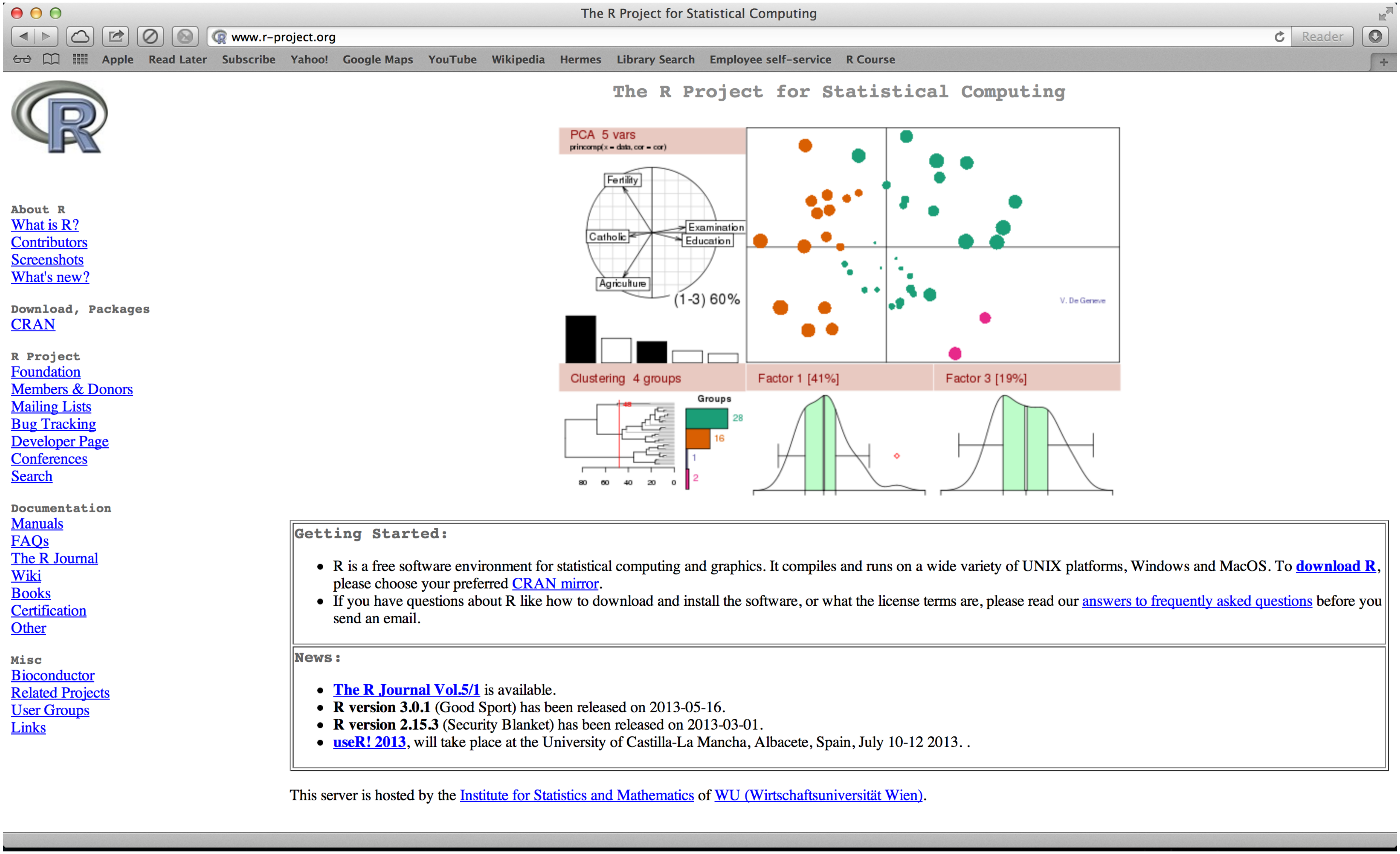
1. Introduction to R and its environment
2. Data structures
3. Data analysis example
4. Programming techniques
5. Statistics

Introduction to R and its environment

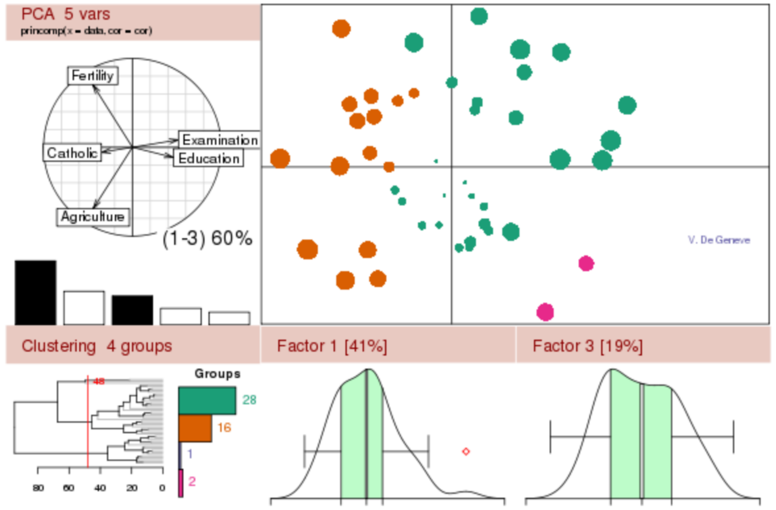
1

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 R Project for Statistical Computing



About R
What is R?
Contributors
Screenshots
What's new?

Download, Packages
CRAN

R Project
Foundation
Members & Donors
Mailing Lists
Bug Tracking
Developer Page
Conferences
Search

Documentation
Manuals
FAQs
The R Journal
Wiki
Books
Certification
Other

Misc
Bioconductor
Related Projects
User Groups
Links

Getting Started:

- R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To **download R**, please choose your preferred CRAN mirror.
- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

News:

- **The R Journal Vol.5/1** is available.
- **R version 3.0.1** (Good Sport) has been released on 2013-05-16.
- **R version 2.15.3** (Security Blanket) has been released on 2013-03-01.
- **useR! 2013**, will take place at the University of Castilla-La Mancha, Albacete, Spain, July 10-12 2013. .

This server is hosted by the Institute for Statistics and Mathematics of WU (Wirtschaftsuniversität Wien).

www.r-project.org

Various platforms supported

- Release 3.0.1 (May 2013)
  - Base package
  - Contributed packages (general purposes extras)
  - ~4700 available packages
- Download from http://www.stats.bris.ac.uk/R/
- Windows, Mac and Linux versions available

- Executed using command line, or a graphical user interface (GUI)
- On this course, we use the RStudio GUI (www.rstudio.com)
- Everything you need is installed on the training machines
- If you are using your own machine, download both R and 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

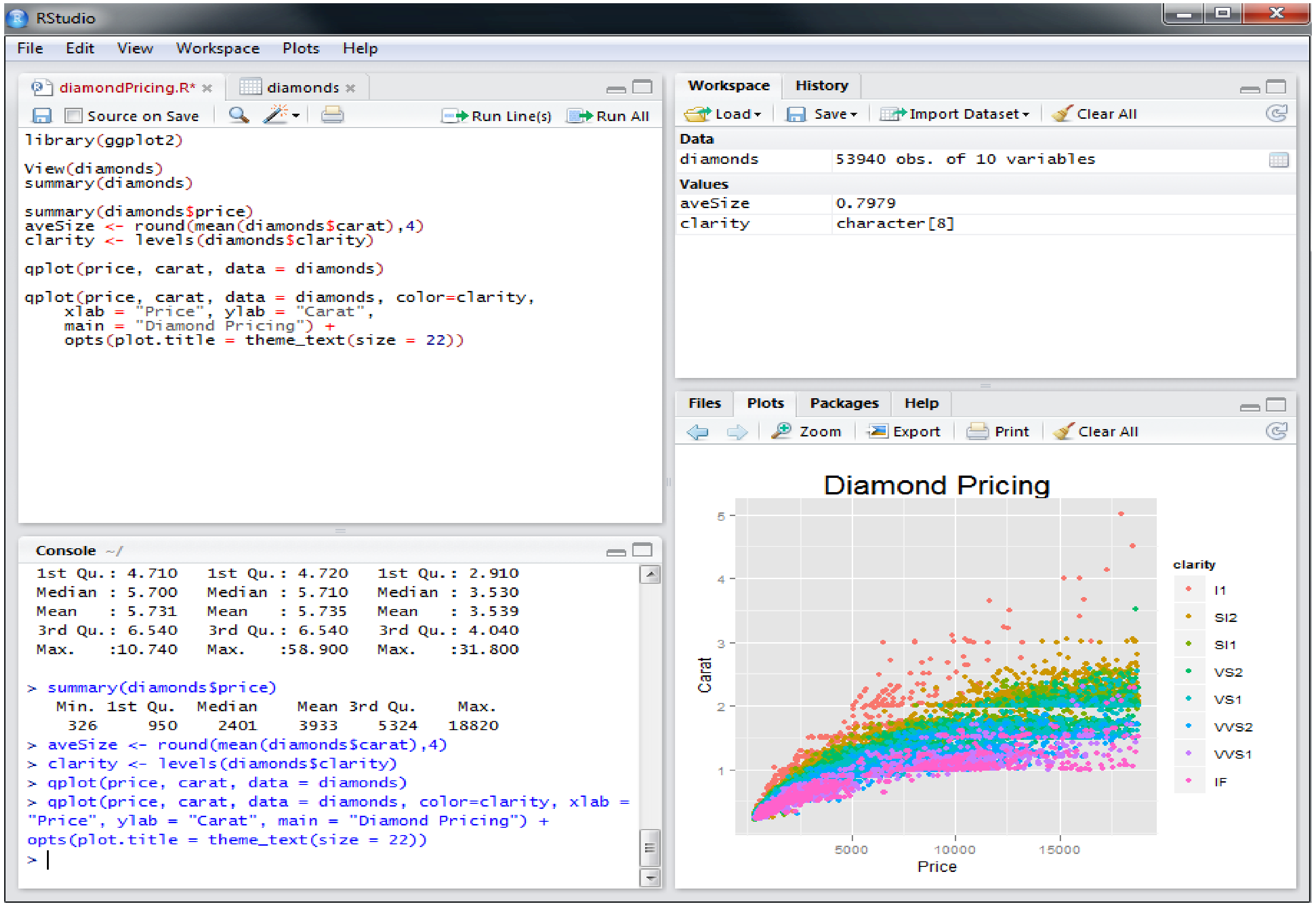
- To start R in Linux we need to enter the Linux console (also called Linux terminal and Linux shell)
- To start R, at the prompt simply type:

```
$ 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


RStudio
File Edit View Workspace Plots Help
diamondPricing.R* diamonds
Source on Save
Run Line(s) Run All
Workspace History
Load Save Import Dataset Clear All
Files Plots Packages Help
Zoom Export Print Clear All
Console ~/
Diamond Pricing
Carat
Price
clarity
I1
SI2
SI1
VS2
VS1
VVS2
VVS1
IF

```
library(ggplot2)

View(diamonds)
summary(diamonds)

summary(diamonds$price)
aveSize <- round(mean(diamonds$carat),4)
clarity <- levels(diamonds$clarity)

qplot(price, carat, data = diamonds)

qplot(price, carat, data = diamonds, color=clarity,
    xlab = "Price", ylab = "Carat",
    main = "Diamond Pricing") +
    opts(plot.title = theme_text(size = 22))
```

Data	
diamonds	53940 obs. of 10 variables
Values	
aveSize	0.7979
clarity	character[8]

```
 1st Qu.: 4.710   1st Qu.: 4.720   1st Qu.: 2.910
 Median : 5.700   Median : 5.710   Median : 3.530
 Mean   : 5.731   Mean   : 5.735   Mean   : 3.539
 3rd Qu.: 6.540   3rd Qu.: 6.540   3rd Qu.: 4.040
 Max.   :10.740   Max.   :58.900   Max.   :31.800

> summary(diamonds$price)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    326     950    2401    3933    5324   18820
> aveSize <- round(mean(diamonds$carat),4)
> clarity <- levels(diamonds$clarity)
> qplot(price, carat, data = diamonds)
> qplot(price, carat, data = diamonds, color=clarity, xlab =
"Price", ylab = "Carat", main = "Diamond Pricing") +
opts(plot.title = theme_text(size = 22))
> |
```

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 set one variable to equal the value of another variable:

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

- 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.
- Try these:

```
> sum(3, 4, 5, 6)
[1] 18
> max(3, 4, 5, 6)
[1] 6
> min(3, 4, 5, 6)
[1] 3
```

- Arguments can be named or unnamed, but if they are unnamed they must be ordered (we will see later how to find the right order).

```
> seq(from=2, to=10, by=2)
[1] 2 4 6 8 10
> seq(2, 10, 2)
[1] 2 4 6 8 10
```

Basic concepts in R
vectors

- The basic data structure in R is a **vector** – an ordered collection of values. R even treats single values as 1-element 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 (the **index**). 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 (vector indexing):

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

- we can write:

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

- or we can use the **seq()** function, which returns a vector:

```
> 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  (...?)
```

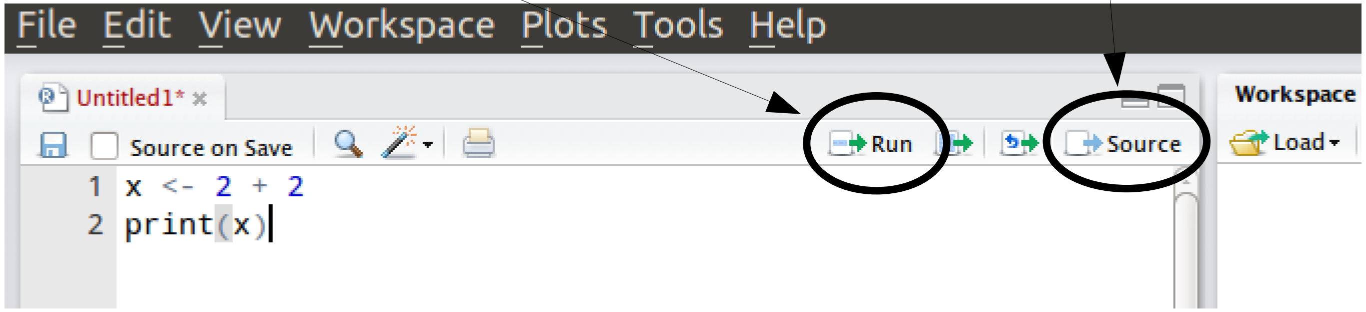
Writing scripts with Rstudio

Typing lots of commands directly to R can be tedious. A better way is to write the commands 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**



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

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. You can see the default order of arguments here. The help page also tells you which **package** it belongs to.
- There will typically be example usage, which you can test using the **example** function:

```
> example(seq)
```

- 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 line 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

- 4700+ 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/
- 670+ packages in Bioconductor:
  - Specialised in genomics:
    http://www.bioconductor.org/packages/release/bioc/
- **Other repositiories:**
- 1600+ projects on R-forge:
  - http://r-forge.r-project.org/
- R graphical manual:
  - http://rgm3.lab.nig.ac.jp/RGM

Bottomline: **always** first look if there is already an R package that does what you want before trying to implement it yourself

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

Data structures

2

R is designed to handle experimental data

- Although the basic unit of R is a vector, we usually handle data in **data frames**.
- A data frame is a set of observations of a set of variables – in other words, the outcome of an experiment.
- For example, we might want to analyse information about a set of patients. To start with, let's say we have ten patients and for each one we know their name, sex, age, weight and whether they give consent for their data to be made public.
- Load this data into a data frame called 'patients' in R:

```
source(“05_patients.R”)
```

The patients data frame

- The 'patients' data frame has ten rows (observations) and seven columns (variables). The columns must all be equal lengths.

```
> patients
   First_Name Second_Name      Full_Name    Sex Age Weight Consent
1        Adam       Jones     Adam Jones   Male  50   70.8    TRUE
2         Eve      Parker     Eve Parker Female  21   67.9    TRUE
3        John       Evans     John Evans   Male  35   75.3   FALSE
4        Mary       Davis     Mary Davis Female  45   61.9    TRUE
5       Peter       Baker    Peter Baker   Male  28   72.4   FALSE
6        Paul     Daniels   Paul Daniels   Male  31   69.9   FALSE
7      Joanna     Edwards Joanna Edwards Female  42   63.5   FALSE
8     Matthew       Smith  Matthew Smith   Male  33   71.5    TRUE
9       David     Roberts  David Roberts   Male  57   73.2   FALSE
10      Sally      Wilson   Sally Wilson Female  62   64.8    TRUE
```

- Let's see how we can construct this from scratch.

Character, numeric and logical data types

- Each column is a vector, like previous vectors we have seen:

```
   > firstName<- c("Adam", "Eve", "John", "Mary", "Peter", "Paul", "Joanna",
"Matthew", "David", "Sally")
   > secondName<-c("Jones", "Parker", "Evans", "Davis", "Baker", "Daniels",
"Edwards", "Smith", "Roberts", "Wilson")
   > age<-c(50, 21, 35, 45, 28, 31, 42, 33, 57, 62)
   > weight<-c(70.8, 67.9, 75.3, 61.9, 72.4, 69.9, 63.5, 71.5, 73.2, 64.8)
   > consent<-c(TRUE,TRUE,FALSE,TRUE,FALSE,FALSE,FALSE,TRUE,FALSE,TRUE)
```

- Each vector has a type, which we can see with the **mode** function:

```
> mode(firstName)
[ 1] "character"

> mode(age)
[ 1] "numeric"

> mode(weight)
[ 1] "numeric"

> mode(consent)
[ 1] "logical"
```

Factors

- Character vectors are fine for some variables, like names
- But sometimes we have categorical data and we want R to recognise this.
- A factor is R's data structure for categorical data.

```
   > sex<-c("Male", "Female", "Male", "Female", "Male", "Male", "Female", 
"Male", "Male", "Female")
   > sex
 [1] "Male"   "Female" "Male"   "Female" "Male"   "Male"   "Female" "Male"  
"Male"   "Female"
   > factor(sex)
 [1] Male   Female Male   Female Male   Male   Female Male   Male   Female
Levels: Female Male
```

- R has converted the strings of the sex character vector into two **levels**, which are the categories in the data.
- Note the values of this factor are not character strings, but levels.
- We can use this factor to compare data for males and females.

Creating a data frame (first attempt)

- We can construct a data frame from other objects:

```
> patients<-data.frame(firstName, secondName, paste(firstName,secondName),
     sex, age, weight, consent)
> patients
  firstName secondName paste.firstName..secondName.    sex age weight consent
1      Adam      Jones                   Adam Jones   Male  50   70.8    TRUE
2       Eve     Parker                   Eve Parker Female  21   67.9    TRUE
3      John      Evans                   John Evans   Male  35   75.3   FALSE
4      Mary      Davis                   Mary Davis Female  45   61.9    TRUE
5     Peter      Baker                  Peter Baker   Male  28   72.4   FALSE
6      Paul    Daniels                 Paul Daniels   Male  31   69.9   FALSE
7    Joanna    Edwards               Joanna Edwards Female  42   63.5   FALSE
8   Matthew      Smith                Matthew Smith   Male  33   71.5    TRUE
9     David    Roberts                David Roberts   Male  57   73.2   FALSE
10    Sally     Wilson                 Sally Wilson Female  62   64.8    TRUE
```

- The **paste** function joins character vectors together.
- We can access particular variables using the **dollar** operator:

```
> patients$age
 [1] 50 21 35 45 28 31 42 33 57 62
```

Naming data frame variables

- R has inferred the names of our data frame variables from the names of the vectors or the commands (eg the paste command).
- We can name the variables after we have created a data frame using the **names** function, and we can use the same function to see the names:
-
```
> names(patients)<-c("First_Name", "Second_Name", "Full_Name", "Sex",
"Age", "Weight", "Consent")
> names(patients)
[1] "First_Name"  "Second_Name" "Full_Name"   "Sex"         "Age"
"Weight"      "Consent"
```

- Or we can name the variables when we define the data frame:

```
> patients<-data.frame(First_Name=firstName, Second_Name=secondName,
Full_Name=paste(firstName,secondName), Sex=sex, Age=age, Weight=weight,
Consent=consent)
> names(patients)
[1] "First_Name"  "Second_Name" "Full_Name"   "Sex"         "Age"
"Weight"      "Consent"
```

Factors in data frames

- When creating a data frame, R assumes all character vectors should be categorical variables and converts them to factors. This is not always what we want:

```
> patients$firstName
 [1] Adam    Eve     John    Mary    Peter   Paul    Joanna  Matthew David   Sally
Levels: Adam David Eve Joanna John Mary Matthew Paul Peter Sally
```

- We can avoid this by asking R not to treat strings as factors, and then explicitly stating when we want a factor by using **factor**:

```
> patients<-data.frame(First_Name=firstName, Second_Name=secondName,
Full_Name=paste(firstName,secondName), Sex=factor(sex), Age=age,
Weight=weight, Consent=consent, stringsAsFactors=FALSE)
> patients$Sex
 [1] Male   Female Male   Female Male   Male   Female Male   Male   Female
Levels: Female Male
> patients$First_Name
 [1] "Adam"    "Eve"     "John"    "Mary"    "Peter"   "Paul"    "Joanna"
"Matthew" "David"   "Sally"
```

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.

      e.g.

      ```
      > 2 + "2"
      ```

      will fail as we cannot add a character string to integer!

Matrices
matrix(..., ncol=…, nrow=…)

- Data frames are R's speciality, but R also handles matrices:

```
> 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
> f ← matrix(1:10, nrow=2, ncol=5)
> f
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> f %*% e
     [,1] [,2]
[1,]   95  220
[2,]  110  260
```

The `%*%` operator is the matrix multiplication operator, not the standard multiplication operator.

Indexing data frames and matrices

Special cases:
a[*i*,] i-th row
a[,*j*] j-th column

- You can index multidimensional data structures like matrices and data frames using commas. If you don't provide an index for either rows or columns, all of the rows or columns will be returned.

```
 object [ rows , columns ]
```

```
> e[1,2]
[1] 6
> e[1,]
[1] 1 6
> patients[1,2]
[1] "Jones"
> patients[1,]
  First_Name Second_Name  Full_Name  Sex Age Weight Consent
1       Adam       Jones Adam Jones Male  50   70.8    TRUE
```

Advanced indexing

- As values in R are really vectors, so indices are actually vectors, and can be numeric or logical:

```
> s <- letters[1:5]
> s[c(1,3)]
[1] "a" "c"
> s[c(TRUE, FALSE, TRUE, FALSE, FALSE)]
[1] "a" "c"
> a<-1:5
> a<3
[1]  TRUE  TRUE FALSE FALSE FALSE
> s[a<3]
[1] "a" "b"
> s[a>1 & a<3]
[1] "b”
> s[a==2]
[1] "b"
```

Operators

- arithmetic

  +, -, *, /, ^

(equal to, not equal to)

- comparison

  <, >, =<, >=, ==, !=

- logical

  !, &, |, xor

these always return logical values ! (TRUE, FALSE)

Exercise

- Create the patients data frame using the instructions in the slides.
- Add three new variables to your data frame: country, continent, and height. Make up the data. Make country a character vector but continent a factor.
- Try the **summary** function on your data frame. What does it do? How does it treat vectors (numeric, character, logical) and factors? (What does it do for matrices?)
- Use logical indexing to select the following patients:
  - Patients under 40
  - Patients who give consent to share their data
  - Men who weight as much or more than the average European male (70.8 kg)

Logical indexing answers

- Patients under 40:

```
> patients[patients$Age<40,]
```

- Patients who give consent to share their data:

```
> patients[patients$Consent==TRUE,]
```

- Men who weigh as much or more than the average European male (70.8 kg):

```
> patients[patients$Sex=="Male" & patients$Weight<=70.8,]
```

R for data analysis

3

3 steps to Basic data analysis

1. Reading in data
   - read.table()
   - read.csv(), read.delim()

2. Analysis
   - Manipulating & reshaping the data
   - Any maths you like
   - Plotting the outcome
     - High level plotting functions (covered tomorrow)

3. Writing out results
   - write.table()
   - write.csv()

A simple walkthrough
Exemplifies 3 steps to R analysis

- 50 neuroblastoma patients were tested for NMYC gene copy number by interphase nuclei FISH
  - Amplification of NMYC correlates with worse prognosis
  - We have count data
    - Numbers of cells per patient assayed
      - For each we have NMYC copy number relative to base ploidy
- We need to determine which patients have amplifications
  - (i.e >33% of cells show NMYC amplification)

Step 1.
Read in the data

Patient	Nuclei	NB_Amp	NB_Nor	NB_Del
1	42	0	34	8
2	40	3	30	7
3	56	6	50	0
4	42	5	37	0
5	32	1	30	1
6	70	10	53	7
7	65	3	58	4
8	40	4	31	5
9	60	0	54	6
10	61	0	57	4
11	43	13	29	1

This data is a tab delimited text file
Each row is a record, each column is a field
Columns are separated by tabs in the text.

We need to read in the results table and assign it to an object (rawData)

```
rawData <- read.delim("08_NBcountData.txt")
rawData[1:10,]     # View the first 10 rows to ensure import is OK
                   # Note data frame contains a patient index column
```

If the data had been comma separated values, then sep=","

```
read.csv("08_NBcountData.csv")
```

`?read.table` for a full list of arguments

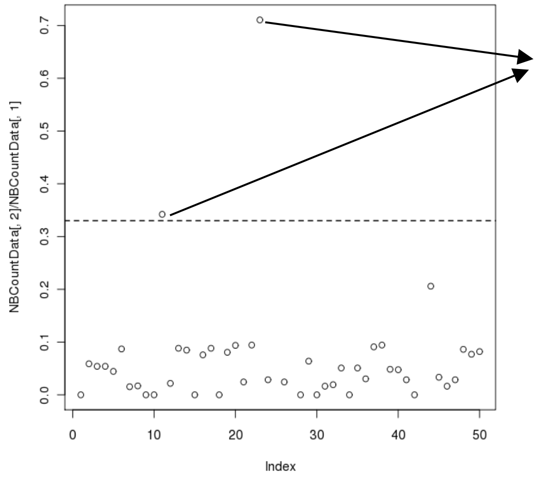
08_NBcountData.R
(script commands)

08_NBcountData.txt
(data file)

Step 2.
Analysis (reshaping data & maths)

- Our analysis involves identifying patients with > 33% NB amplification
  - `prop <- rawData$NB_Amp / rawData$Nuclei # create an index of results`
  - `amp <- which(prop > 0.33) # Get sample names of amplified patients`

- We can plot a simple chart of the % NB amplification
  - `plot(prop, ylim=c(0,1.2))`
  - `abline(h=0.33,lwd=1.5,lty=2)`



These 2 samples are amplified (11 & 23)

Step 3.
Outputting the results

- We write out a data frame of results (patients > 33% NB amplification) as a 'comma separated values' text file
  - `write.csv(rawData[amp,],file="selectedSamples.csv") # Export table, file name = selectedSamples.csv`
    - Files are directly readable by Excel and Calc

- Its often helpful to double check where the data has been saved
  - Use get working directory function
    - `getwd() # print working directory`

Data analysis exercise:
Which samples are near normal?

- Patients are near normal if:

(NB_Amp/Nuclei <0.33 & NB_Del ==0)

- Modify the condition in our previous code to find these patients

- Write out a results file of the samples that match these criteria, and open it in a spreadsheet program

08_NBcountData.R
(script commands)

Solution to NB normality test
Basic data analysis

```
> norm <- which( prop < 0.33 & rawData$NB_Del==0)
> norm

[1] 3  4  7 15 20 24 36 37 42 47


> write.csv(rawData[norm,],"My_NB_output.csv")
```

R programming techniques

4

Basic R 'Built-in' functions for working with objects

- R has many built-in functions for doing simple calculations on objects. Start with a random sample of 15 numbers from 1 to 100 and try the functions below.

```
> x<-sample(100,15)
```

- Arithmetic with vectors
  - Min / Max value number in a series

```
min(x) ; max(x)
```

  - Sum of values in a series

```
sum(x)
```

  - Average estimates (mean / median)

```
mean(x) ; median(x)
```

  - Range of values in a series

```
range(x)
```

  - Variance

```
var(x)
```

- Arithmetic with vectors
  - Rank ordering

```
rank(x)
```

  - Quantiles

```
quantile(x); boxplot(x)
```

  - Square Root

```
sqrt(x)
```

  - Standard deviation

```
sd(x)
```

  - Trigonometry functions

```
tan(x) ; cos(x) ; sin(x)
```

Basic R 'Built-in' functions for working with variables

- list & remove objects

```
ls(), rm()
rm(list=ls()) # get rid of everything
```

- Add rows or columns to a data frame, *df*. Row bind, column bind

```
rbind(df,...), cbind(df,...)
```

- Remove a row, or column, from a data frame.

```
df[-1,] # remove first row
df[,-1] # remove first column
```

- Names of objects

```
names(…)
colnames(...)
rownames(...)
```

- Return length of an object, number of rows or columns of a dataframe or matrix

```
length(…)
nrow(…)
ncol(…)
```

Sorting a vector with **sort**:

```
sort(patients$Second_Name)
[1] "Baker"   "Daniels" "Davis"   "Edwards" "Evans"   "Jones"   "Parker"  "Roberts" "Smith"
 "Wilson"
```

Sorting a data frame by one variable with **order**:

```
order(patients$Second_Name)
[1]  5  6  4  7  3  1  2  9  8 10
patients[order(patients$Second_Name),]
```

Looping - informal introduction

- What if we had 100 data files to load in, and we wanted to load them all into one data frame?
- We could do this:

```
> colony<-data.frame()     # Start with empty data frame
> colony<-rbind(colony, read.csv("11_CFA_Run1Counts.csv"))
> colony<-rbind(colony, read.csv("11_CFA_Run2Counts.csv"))
> colony<-rbind(colony, read.csv("11_CFA_Run3Counts.csv"))
...
> colony<-rbind(colony, read.csv("11_CFA_Run100Counts.csv"))
```

But this will be boring to type, difficult to change, and prone to error.

- As we are doing the same thing 100 times, but with a different file name each time, we can use a **loop** instead.

R language elements
Commands & flow control

- Looping
  - Iterate over a set of values (**for** loop)
  - or while a condition is met (**while** loop)

• Loops are very common in most programming languages, but are not as common in R. Because R can do vectorized calculations, there is no need to use loops to do most things – for example, to sum two vectors.

• Loops are multi-line commands. R will execute them only after the whole loop has been typed in. Use Rstudio editor to type it all in, don't do it in R console!

LOOPS
Commands & flow control

- We can generate a filename using **paste**:

```
paste("11_CFA_Run",1,"Counts.csv",sep="")
[1] "11_CFA_Run1Counts.csv"
```

- So we can load all the files using a **for** loop as follows:

```
colony<-data.frame()
for (f in 1:100) {
    t<-read.csv(paste("11_CFA_Run",f,"Counts.csv",sep=""))
    colony<-rbind(colony,t)
}
```

- Or we could use a **while** loop:

```
f <- 1
colony<-data.frame()
while ( f <= 100 ) {
    t<-read.csv(paste("11_CFA_Run",f,"Counts.csv",sep=""))
    colony<-rbind(colony,t)
    f <- f + 1
}
```

when this condition is false the loop stops

Loops with breaks

Commands & flow control

Suppose, for testing purposes, we only wanted to load the first 2 files in, to make sure our analysis worked on those before we load all the data in. We can use an **if** statement to check for a condition:

```
colony<-data.frame()
for (f in 1:100) {
    if (f<=2) {
        t<-read.csv(paste("11_CFA_Run",f,"Counts.csv",sep=""))
        colony<-rbind(colony,t)
    } else {
        warning(paste("Not loading past file ", f))
        break
    }
}
```

The break statement ends the loop on whichever iteration has been reached. The **warning** function prints out an error message, but carries on with the program (use **stop** if you want to output an error and quit).

Conditional branching
Commands & flow control

- Use an **if** statement for any kind of condition testing.

- Different outcomes can be selected based on a condition within brackets.

```
if (condition) {
… do this …
} else {
… do something else …
}
```

  - condition is any logical value, and can contain multiple conditions
    - e.g. `(a==2 & b <5),` this is a compound conditional argument

Code formatting avoids bugs!

- Code formatting is crucial for readability of loops

```
f<-26
while(f!=0){
print(letters[f])
f<-f-1 }
```

BAD!!!

```
f <- 26
while( f != 0 ){
    print(letters[f])
    f <- f-1
}
```

GOOD!

- The code between brackets {} **always** is indented, this clearly separates what is executed once, and what is run multiple times
- Trailing bracket } always alone on the line at the same indentation level as the initial bracket {
- Use white spaces to divide the horizontal space between units of your code, e.g. around assignments, comparisons

Exercise

1. Load in the **colony** data frame using a for loop. Three of the data files (but not the other 97!) are in the *Day_1_scripts* folder. Load all three files into **colony**.

2. How many observations do you have? Find out by counting the number of rows in **colony** using the **nrow** function.

3. You have calculated that you will have sufficient power for your analysis if you have at least 70 observations. Write a **while** loop that will continue to load files until you have loaded at least 70 observations into the **colony** data frame.

Answers to exercise

1. To load all three files, use the code from the first **for** loop slide, but only specify three files:

```
colony<-data.frame()
for (f in 1:3) {
   t<-read.csv(paste("11_CFA_Run",f,"Counts.csv",sep=""))
   colony<-rbind(colony,t)
}
```

2. Loading enough files to load 70 observations:

```
f <- 1
colony<-data.frame()
while ( nrow(colony)<=70 ) {
   t<-read.csv(paste("11_CFA_Run",f,"Counts.csv",sep=""))
   colony<-rbind(colony,t)
   f <- f + 1
}
```

Statistics

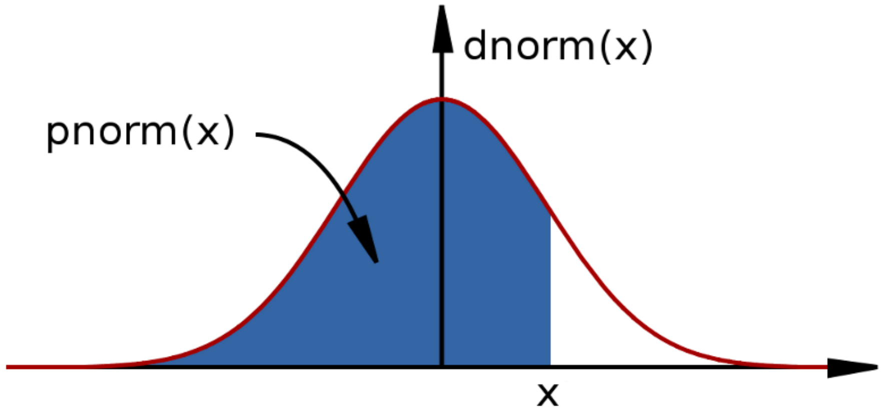
5

Built-in support for statistics

- R is a statistical programming language
  - Classical statistical tests are built-in
  - Statistical modeling functions are built-in
  - Regression analysis is fully supported
  - Additional mathematical packages are available
    - MASS, Waves, sparse matrices, etc

Pseudo-random numbers and distributions

- mostly commonly used distributions are built-in, functions have stereotypical names, e.g. for normal distribution:
  - pnorm - cumulative distribution for x
  - qnorm - inverse of pnorm (from probability gives x)
  - dnorm - distribution density
  - rnorm - random number from normal distribution



- available for variety of distributions: punif (uniform), pbinom (binomial), pnbinom (negative binomial), ppois (poisson), pgeom (geometric), phyper (hyper-geometric), pt (T distribution), pf (F distribution) ...

Two sample tests
Basic data analysis

- Comparing 2 variances
  - Fisher's F test

`var.test()`

- Comparing 2 sample means with normal errors
  - Student's t test

`t.test()`

- Comparing 2 means with non-normal errors
  - Wilcoxon's rank test

`wilcox.test()`

- Comparing 2 proportions
  - Binomial test

`prop.test()`

- Correlating 2 variables
  - Pearson's / Spearman's rank correlation

`cor.test()`

- Testing for independence of 2 variables in a contingency table
  - Chi-squared

`chisq.test()`

  - Fisher's exact test

`fisher.test()`

Comparison of 2 data sets example
Basic data analysis

- Men, on average, are taller than women.
  - The steps
    1. Determine whether variances in each data series are different
       - Variance is a measure of sampling dispersion, a first estimate in determining the degree of difference
         - *Fisher's F test*
    2. Comparison of the mean heights.
       - Determine probability that mean heights really are drawn from different sample populations
         - *Student's t test, Wilcoxon's rank sum test*

1. Comparison of 2 data sets
Fisher's F test

- Read in the data file into a new object, heightData

```
heightData<-read.csv("10_heightData.csv",header=T)
```

- **attach** the data frame so we don't have to refer to it by name all the time:

```
attach(heightData)
```

- Do the two sexes have the same variance?

```
var.test(Female,Male)
```

```
F test to compare two variances

data:  Female and Male
F = 1.0073, num df = 99, denom df = 99, p-value = 0.9714
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.6777266 1.4970241
sample estimates:
ratio of variances
           1.00726
```

2. Comparison of 2 data sets
Student's t test

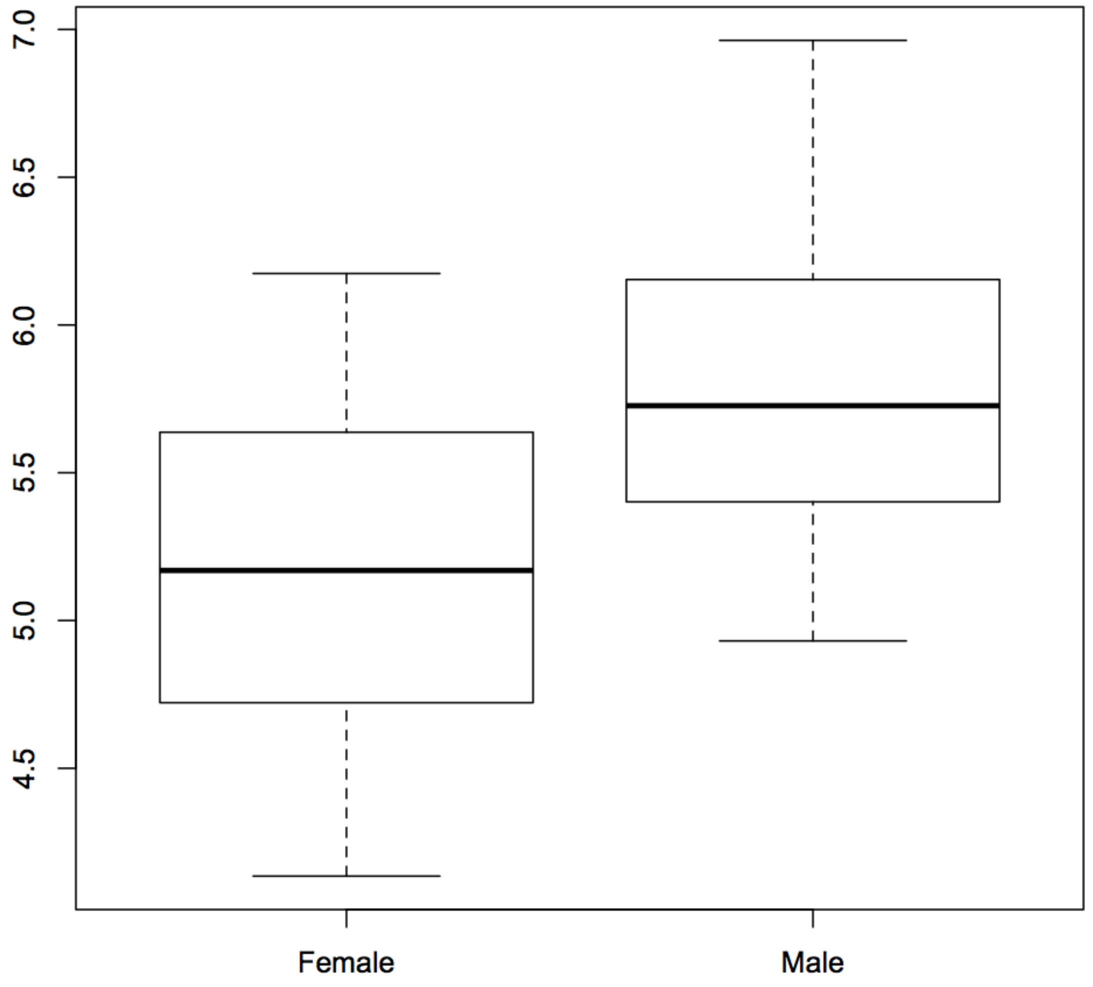
- Student's t test is appropriate for comparing the difference in mean height in our data.
  - Remember a t test = $\dfrac{\text{difference in two sample means}}{\text{standard error of the difference of the means}}$

```
t.test(Female,Male)

Welch Two Sample t-test

data:  Female and Male
t = -8.4508, df = 197.997, p-value = 6.217e-15
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.7788497 -0.4841288
sample estimates:
mean of x mean of y
 5.168725  5.800214
```

3. Comparison of 2 data sets

Review findings

```
> boxplot(heightData)
```



Linear regression
Basic data analysis

- Linear modeling is supported by the function `lm()`
    - `example(lm)` # the output assumes you know a fair bit about the subject

- lm is really useful for plotting lines of best fit to XY data in order to determine, intercept, gradient & Pearson's correlation coefficient
    - This is very easy in R

- Three steps to plotting with a best fit line
    - Plot XY scatter-plot data
    - Fit a linear model
    - Add bestfit line data to plot with abline() function

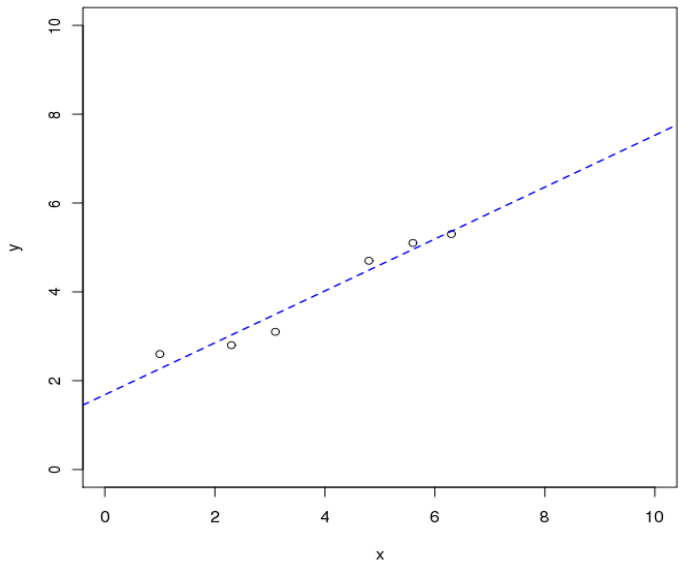
Typical linear regression analysis
Basic data analysis

X	Y
1.0	2.6
2.3	2.8
3.1	3.1
4.8	4.7
5.6	5.1
6.3	5.3

```
> x<-c(1, 2.3, 3.1, 4.8, 5.6, 6.3)
> y<-c(2.6, 2.8, 3.1, 4.7, 5.1, 5.3)
> plot(y~x, xlim=c(0,10),ylim=c(0,10))
```

Note formula notation
(y is given by x)

```
> myModel<-lm(y~x)
> abline(myModel,lty=2,lwd=1.5,col="blue")
```



Get the coefficients of the fit from:
`summary.lm(myModel)` **and**
`coef(myModel)`
`resid(myModel)`
`fitted(myModel)`

Get QC of fit from
`plot(myModel)`

Find out about the fit data from
`names(myModel)`

The linear model object

Basic data analysis

- Summary data describing the linear fit is given by
  - summary.lm(myModel)

```
> summary.lm(myModel)

Call:
lm(formula = y ~ x)↰

Residuals:
       1        2        3        4        5        6 
 0.33159 -0.22785 -0.39520  0.21169  0.14434 -0.06458 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)   
(Intercept)  1.68422    0.29056   5.796   0.0044 **
x            0.58418    0.06786   8.608   0.0010 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 

Residual standard error: 0.3114 on 4 degrees of freedom
Multiple R-squared: 0.9488,   Adjusted R-squared: 0.936 
F-statistic:  74.1 on 1 and 4 DF,  p-value: 0.001001
```

Y intercept

Gradient

Good fit: would happen 1 in 1000 by chance

R^2 , with pValue

Modelling formulae

- It is very easy to extend R model formulas to do multiple regressions, ANOVAs, include interactions...
- Suppose we had two explanatory variables **x** and **z** and one response variable **y**.

```
> y~x       # If x is continuous, this is linear regression
> y~x       # If x is categorical, this is ANOVA
> y~x+z     # If x and z are continuous, this is multiple regression
> y~x+z     # If x and z are categorical this is a two-way ANOVA
> y~x+z+x:z # : is the symbol for the interaction term
> y~x*z     # * is a shorthand for the above: x+z and their interaction
```

Exercise
The coin toss

To learn how the distribution functions work, try simulating tossing a fair coin 100 times and then show that it is fair.

1) We can model a coin toss using the binomial distribution. Use the **rbinom** function to generate a sample of 100 coin tosses. Look up the binomial distribution help page to find out what arguments this function needs.
2) How many heads or tails were there in your sample? You can do this in two ways; either select the number of successes using indices, or convert your sample to a factor and get a summary of the factor.
3) If we toss a coin 50 times, what is the probability that we get exactly 25 heads? What about 25 heads or less? Use **dbinom** and **pbinom** to find out.
4) The argument to **pbinom** is a vector, so try calculating the probabilities for getting any number of coin tosses from 0 to 50 in fifty trials using **dbinom**. Plot these probabilities using **plot**. Does this plot remind you of anything?

Coin toss answers

- To simulate a coin toss, give **rbinom** a number of observations, the number of trials for each observation, and a probability of success:

```
> coin.toss<-rbinom(100, 1, 0.5)
```

- Because we only specified one trial per observation, we either have an outcome of 0 or 1 successes. To get the number of successes, use indices or a factor to look up the number of 1s in the **coin.toss** vector (your numbers will vary):

```
> length(coin.toss[coin.toss==1])
[1] 50
> summary(factor(coin.toss))
 0  1
50 50
```

Coin toss answers

The probability of getting exactly 25 heads from 50 observations of a fair coin:

```
> dbinom(25, 50, 0.5)
```

The probability of getting 25 heads or less from 50 observations of a fair coin:

```
> pbinom(25, 50, 0.5)
```

The probabilities for getting all numbers of coin tosses from 0 to 50 in fifty trials:

```
> dbinom(0:50, 50, 0.5)
```

To plot this distribution, which should resemble a normal distribution:

```
> plot(dbinom(0:50, 50, 0.5))
```

Exercise
Linear modelling example

Mice have varying numbers of babies in each litter. Does the size of the litter affect the average brain weight of the offspring? We can use linear modelling to find out. (This example is taken from John Maindonald and John Braun's book *Data Analysis and Graphics Using R* (CUP, 2003), p140-143.)

1) Install and load the **DAAG** package. The **litters** data frame is part of this package. Take a look at it. How many variables and observations does it have? Does **summary** tell you anything useful? What about **plot**?
2) Are any of the variables correlated? Look up the **cor.test** function and use it to test for relationships.
3) Use **lm** to calculate the regression of brain weight on litter size, brain weight on body weight, and brain weight on litter size and body weight together.
4) Look at the coefficients in your models. How is brain weight related to litter size on its own? What about in the multiple regression? How would you interpret this result?

Linear modelling answers

- To install and load the package and look at **litters**:

```
> install.packages("DAAG")
> library(DAAG)
> litters
> summary(litters)
> plot(litters)
```

- To calculate correlations between variables:

```
> attach(litters)
> cor.test(brainwt, lsize)
> cor.test(bodywt, lsize)
> cor.test(brainwt, bodywt)
```

Linear modelling answers

- To calculate the linear models:

```
> lm(brainwt~lsize)
Call:
lm(formula = brainwt ~ lsize)

Coefficients:
(Intercept)        lsize
   0.447000    -0.004033
```

```
> lm(brainwt~bodywt)
Call:
lm(formula = brainwt ~ bodywt)

Coefficients:
(Intercept)       bodywt
    0.33555      0.01048
```

```
> lm(brainwt~lsize+bodywt)
Call:
lm(formula = brainwt ~ lsize + bodywt)

Coefficients:
(Intercept)        lsize       bodywt

    0.17825      0.00669      0.02431
```

Interpretation: brain weight decreases as litter size increases, but brain weight increases proportional to body weight (when bodywt is held constant, the lsize coefficient is positive – 0.00669). This is called 'brain sparing'; although the offspring get smaller as litter size increases, the brain does not shrink as much as the body.