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

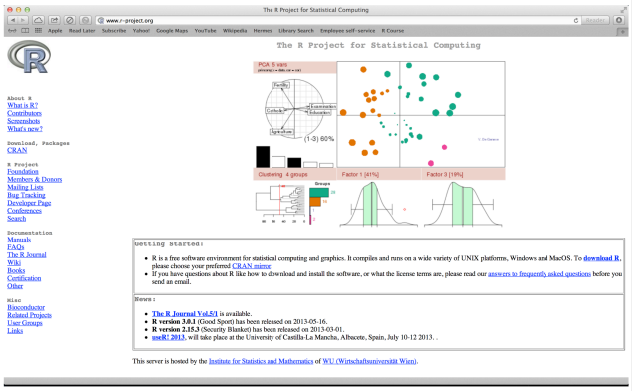
1. Introduction to R and its environment
2. Data structures
3. Data analysis example
4. Programming techniques
5. Statistics

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



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

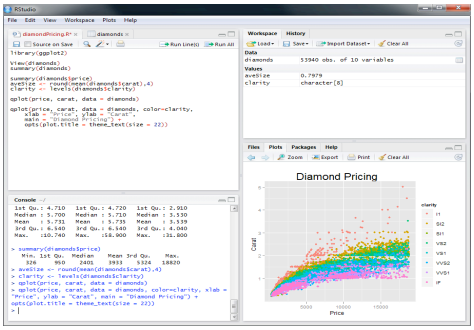
- To start R in Linux we need to enter the Linux console (also called Linux terminal and Linux shell)
- To start R, at the prompt simply type:

```
$ 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



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  (...?)
```

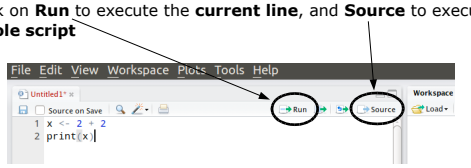
Writing scripts with Rstudio

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**



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   Female Male   Male   Female Male
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
```

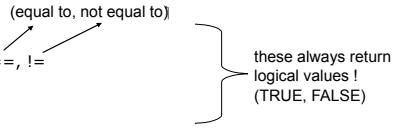
Advanced indexing

- 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
  +, -, *, /, ^
- comparison
  <, >, =<, >=, ==, !=
- logical
  !, &, |, xor



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")
```

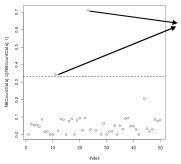
`?read.table` for a full list of arguments

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

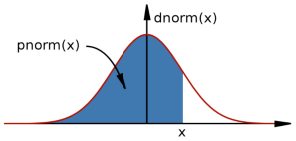
5

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
```

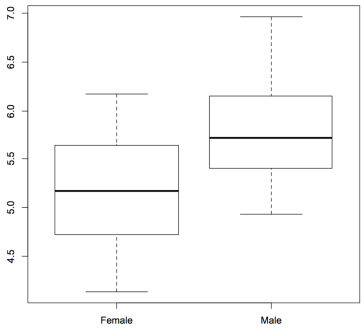
2. Comparison of 2 data sets
Student's t test

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

```
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

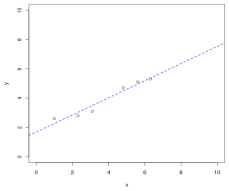
Typical linear regression analysis
Basic data analysis

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.

Writing custom scripts & running R batch mode analysis

1

The R scripting language
Scripting

- A script is a series of instructions that when executed sequentially automates a task
  - A script is a good solution to a repetitive problem
  - The art of good script writing is
    - understanding exactly what you want to do
    - expressing the steps as concisely as possible
    - making use of error checking
    - including descriptive comments
- R is a powerful scripting language, and embodies aspects found in most standard programming environments
  - procedural statements
  - loops
  - functions
  - conditional branching
- Scripts may be written in any standard text editor, e.g. notepad, gedit, kate
  - RGui (Mac and Windows) has a built-in text editor

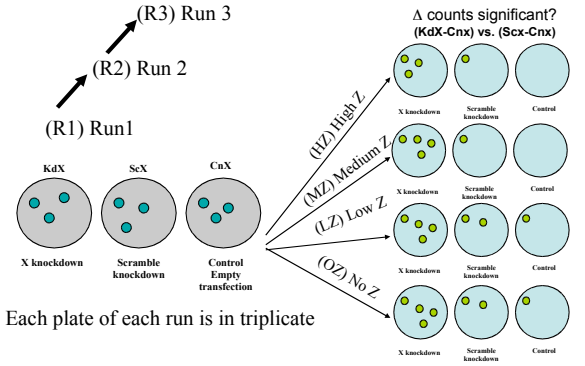
An example script
Scripting

- Colony forming assays provide a measure of cellular proliferation. They are used as read outs for various biological systems
  - A well controlled study may involve multiple samples, treatments and controls (probably replicated).
  - This produces a lot of 'count' data, ideally suited to routine script processing
- Encapsulating the analysis into an R script requires a clear understanding of the problem and data structure

CFA experimental design
Scripting

- Expression of gene X may prevent cells from proliferating in high concentrations of compound Z. The theory is tested by knocking down gene X and growing cells in varying concentrations of compound Z.
  - Three repeat runs (same cell line)
    - Gene X knockdown --> KdX
    - Scramble gene X knockdown control --> ScX
    - Control (transfect empty vector) --> CnX
  - 4 concentrations of compound Z
    - High (HZ), Medium (MZ), Low (LZ), None (OZ)
  - The experiment is replicated over 3 successive weeks
    - Run1 (R1), Run2 (R2) and Run3 (R3)
  - 108 counts in total

Colony forming assay experimental design



Each plate of each run is in triplicate

Preparing the calculation(s)
Scripting

- We need to make barplots of counts for the KDX-CNX and SCX-CNX for each concentration of Z.
- We will group the repeat runs & replicates, and take an average.
- A Wilcoxon Rank Sum test will tell us whether there is a significant level of protection for KDX in concentrations of Z
- We'll add in some data quality checks
  - Boxplots of repeat runs
  - Variance within replicates

We can copy & paste lines of code into a blank text document, try them out and keep the ones that work!

Importing data
Scripting

	R1						R2						R3					
Plate	KDX			SCX			CNX			KDX			SCX			CNX		
Replicate	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
OZ																					
LZ																					
MZ																					
HZ																					

Fine for humans, bad for R

Run	Plate	Treatment	Replicate	Count
R1	KDX	OZ	1	105
R1	KDX	OZ	2	101
R1	KDX	OZ	3	83
R1	KDX	LZ	1	47
R1	KDX	LZ	2	41
R1	KDX	LZ	3	44
R1	KDX	MZ	1	13
R1	KDX	MZ	2	12
R1	KDX	MZ	3	18
R1	KDX	HZ	1	11
R1	KDX	HZ	2	9
R1	KDX	HZ	3	19
R1	SCX	OZ	1	156
R1	SCX	OZ	2	96
R1	SCX	OZ	3	154
R1	SCX	LZ	1	75
R1	SCX	LZ	2	43
R1	SCX	LZ	3	52
R1	SCX	MZ	1	7
R1	SCX	MZ	2	9
R1	SCX	MZ	3	14
R1	SCX	HZ	1	11
R1	SCX	HZ	2	9
R1	SCX	HZ	3	19
R1	CNX	OZ	1	110
R1	CNX	OZ	2	166
R1	CNX	OZ	3	73
R1	CNX	LZ	1	14
R1	CNX	LZ	2	11
R1	CNX	LZ	3	20
R1	CNX	MZ	1	11
R1	CNX	MZ	2	8
R1	CNX	MZ	3	10
R1	CNX	HZ	1	5
R1	CNX	HZ	2	5
R1	CNX	HZ	3	11

Factors

Response

We will create a data frame of factors comprising Run, Plate, Treatment, Replicate
The response variable is counts.

We have 3 spreadsheets of data
Run1counts.csv
Run2counts.csv
Run3counts.csv

We will need to write a procedure that reads in the data
Note that our script will require a consistent data format

Prepare for raw data
Script walkthrough 1

- Open a blank text document, and prepare to write this script
  - The data is contained in three files:
    - 11_CFA_Run1Counts.csv
    - 11_CFA_Run2Counts.csv
    - 11_CFA_Run3Counts.csv
  - Load in the data and concatenate it into a single data frame

```
# load in the data from the three runs into three separate data frames t1,
      t2, t3
t1 = read.csv("11_CFA_Run1Counts.csv")
t2 = read.csv("11_CFA_Run2Counts.csv")
t3 = read.csv("11_CFA_Run3Counts.csv")

# concatenate the three data frames into a single data frame
colony = rbind(t1, t2, t3)

# (or use one of the loops from yesterday...)
```

Example code:
11_CFAcountData.R

Import raw data
Script walkthrough 2

- Data is by default read in as factors, i.e. all input strings are enumerated and stored as numbers
- The three separate data frame have no indication of which number they came from. We will add a column indicating this:

```
# add the missing Run column - factors are stored as numbers !
runNum <- factor( rep( 1:3, each=36 ), labels=c("Run1","Run2","Run3") )
colony <- cbind( "Run" = runNum, colony )

# reorder factor levels in their natural order (instead of alphabetical)
colony$Treatment <- factor(colony$Treatment, c("OZ", "LZ", "MZ", "HZ"))
colony$Plate <- factor(colony$Plate, c("KDX","SCX","CNX"))

# show the full table
colony
```

The tapply function
a brief digression

- Assume we have the following data for heights of 5 males and females:

```
    data <- data.frame(gender=c("Male", "Male", "Female",
                "Female", "Female"), height=c(6, 6.1, 5.8, 6, 5.95))
gender height
1   Male   6.00
2   Male   6.10
3 Female   5.80
4 Female   6.00
5 Female   5.95
```

- By calling mean() on the height column we can get the average of all 5 people, but how do we get average separately for males and females?
- `tapply()` lets us do exactly this
  - It applies a function to grouped data:
- `tapply( data$height, data$gender, mean )`
  data groups function

Undertake data analysis
Script walkthrough 3

- We need the means of the triplicate counts for each Run
  - Broken down by plate type (KDX,SCX,CNX) and Z treatment concentration (OZ,LZ,MZ,HZ)

```
### Part 2.  Investigating data ###
tapply(colony$Count, list(colony$Run, colony$Plate,
colony$Treatment), mean)
```

```
, , OZ

          KDX      SCX      CNX
Run1  98.33333 129.6667 108.3333
Run2 180.33333 206.0000 188.6667
Run3 282.33333 288.6667 265.6667

, , LZ

          KDX      SCX      CNX
Run1  75.0000  53.0000 21.66667
Run2 136.3333 103.6667 32.00000
Run3 157.0000 180.6667 46.66667

, , MZ

          KDX       SCX       CNX
Run1 29.66667  6.333333 0.3333333
Run2 47.00000 11.666667 2.0000000
Run3 73.00000 17.333333 3.6666667

, , HZ

          KDX       SCX       CNX
Run1  6.333333 1.3333333 0.3333333
Run2 12.000000 0.3333333 0.0000000
Run3 18.666667 0.3333333 0.0000000
```

We can plot a graph of this. It gives us the variation in counts per run

```
par(oma=c(4,2,2,2))
boxplot(Count~Run*Plate*Treatment, las=2, cex=0.2,
data=colony)
```

Better still, lets plot a grouped bar chart of mean counts per plate type per Z treatment

```
barplot(tapply(colony$Count, list(colony$Plate,
colony$Treatment), mean),beside=T)
```

Summarize & save the analysis
Script walkthrough 4

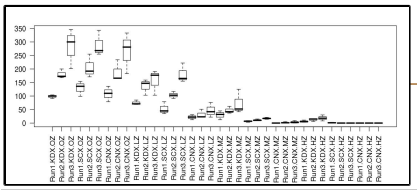
- we need a reshaped, background corrected, table of results on which to perform our tests
- for clarity where possible use dollar ($) notation (work only with data frames)

```
### Part 3.  Summarizing data ###
result <- tapply(colony$Count, list(colony$Treatment, colony$Plate), mean)
result <- data.frame(result) # result of tapply is matrix, convert to dataframe
result

# calculate kdx and scx values after background correction
kdx = result$KDX - result$CNX
scx = result$SCX - result$CNX

result <- cbind(kdx, scx)
# remove the OZ entry
result <- result[-1,]
barplot(result,beside=T)

wilcox.test(result[,1],result[,2],paired=T)
cor.test(result[,1],result[,2],paired=T)
write.csv(result,"CFAresults.csv")
```

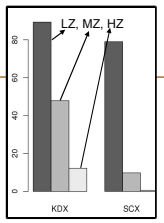
-ve subscripts mean 'delete'

We can plot the results as a barchart, and undetake an appropriate two sample classical test

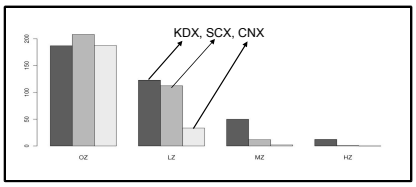
We find that the difference in means is not significant (would expect observation to occur 1:4 times), and that the scramble and knockdown counts have a 90% correlation



Boxplot shows variation in replicates. Run 3 had greatest count variation.



Barchart shows KDX had greater viability in Z compared to SCX. Treatments are Low, Medium and High.



Barchart shows run average counts of various plate types in different Z treatments. KDX is the only plate type that had growth in high Z.

```
> wilcox.test(result[,1],result[,2],paired=TRUE)

    Wilcoxon signed rank test

data:  result[, 1] and result[, 2]
V = 6, p-value = 0.25
alternative hypothesis: true location shift is not equal
to 0
```

```
> cor.test(result[,1],result[,2],paired=TRUE)

    Pearson's product-moment correlation

data:  result[, 1] and result[, 2]
t = 2.5584, df = 1, p-value = 0.2372
alternative hypothesis: true correlation is not equal
to 0
sample estimates:
      cor
0.9313792
```

Would expect to see trend 1 in 4 times. There is a 93% correlation between the knockdown of gene X and scramble control and cell counts response when grown in compound Z.

Script steps review
Script walkthrough 5

- Excel formatted data needs to be exported as comma separated values text (or tab!)
- Get the data into R
  - `read.csv()` … to assign the data to an object
- Produce exploratory plots
  - `boxplot()`
  - `barplot()`
- Undertake statistical tests
  - `cor.test()`
    - Spearman's rank correlation test
  - `wilcox.test()`
    - Wilcoxon test with two sets of paired data … Mann-Whitney U test
- Write out the results
  - `write.csv()`
    - exports data as comma separated list
  - `save.image()`
    - could also save the R environment after analysis (we didn't do this)

Exercise
Colony forming assay script

- Enter the text of the count data script, and save the file.
  - To run the count data script in R, type
  - `source("filename") # the script is available as 11_countData.R`
- Each step of the script is executed, and the results displayed.
- We need to export the graphical output to a file, and the R objects also need to be saved.
  - Modify the script as follows:

Section 3, line directly after ***tapply*** command insert:

```
jpeg(file="fig1.jpg",width=1600,height=800,res=150)
par(oma=c(4,2,2,2))
… <boxplot commands in middle> …
dev.off()
jpeg(file="fig2.jpg",width=675,height=900,res=150)
… <barplot commands in middle> …
dev.off()
```

Section 4, line directly after ***result*** command insert:

```
jpeg(file="fig3.jpg",width=1600,height=800,res=150)
… <barplot commands in middle> …
dev.off()
```

Batch processing R scripts
Scripting

- Scripts can be run without ever launching R, using R CMD batch mode.

quit R and type the following in a linux terminal

```
R CMD BATCH -no-restore 11_CFAcountData.R
```

or if you write all of graphical output to files:

```
Rscript 11_CFAcountData.R
```

(works only with recent R versions)

Advanced example: multiple file handling
Reading files using loops and regular expressions

- Earlier we read in each of the three colony files one-by-one
- Yesterday, we saw how to load files using a loop
- But what if we didn't know how many files we had?
- We can use a regular expression to look up the list of available files

```
# look for patterns like 'Counts.csv'
this.pattern <- "*Counts.csv"

# find all filenames in the current directory containing this pattern
matching.filenames <- dir(pattern=glob2rx(this.pattern))

# see what has been found
matching.filenames

[1] "11_CFA_Run1Counts.csv" "11_CFA_Run2Counts.csv" "11_CFA_Run3Counts.csv"
```

Advanced example: multiple file handling
Reading files using loops and regular expressions

- Using '*' in the pattern is a 'wild card' – means we search for anything that ends in 'Counts.csv'
- glob2rx() is a function which translates this pattern into something the file system can recognise
- Now loop through the matching filenames and open each in turn

```
# start with an empty data frame
colony <- data.frame()

# loop through vector of file names
for(filename in matching.filenames){
      # open each file
      t <- read.csv(matching.filename)
      # append rows
      colony <- rbind(colony, t)
}
```

User functions

2

Introducing ...
User functions

- All R commands are functions.
- Functions perform calculations, possibly involving several arguments, then return a value to the calling statement.
- The calculation maybe any process, might or might not have return value
  - It need not be arithmetic
- User functions extend the capabilities of R by adapting or creating new tasks that are tailored to your specific requirements.
- User functions are a special kind of object

Defining a new function

- Parts of function definition: name, arguments, procedural steps, return value

```
sqXplusX <- function(x){
    x^2 + x
}
```

- **sqXplusX** is the function name
- **x** is the single argument to this function and it exists only within the function
- everything between brackets { } are procedural steps
- the **last** calculated value is the function return value
- after defining the function, we can use it:

```
> sqXplusX(10)
[1] 110
```

Named and default arguments

- Example of function with more than one named argument:

```
powXplusX <- function(x, power=2){
    x^power + x
}
```

- Now we have two arguments. The second argument has a default value of 2.
- Arguments without default value are required, those with default values are optional.

```
> powXplusX(10)
[1] 110
> powXplusX(10, 3)
[1] 1010
> powXplusX(x=10, power=3)
[1] 1010
```

arguments matched based on **position**

arguments matched based on **name**

Assignments with arguments
User functions

```
sqXplusX <- function(x){
   x^2 + x
}
```

You can use a blank document in gedit, nedit or other text editor to hold these commands for you, then copy / paste the instructions into R

- Now try this …

```
a <- matrix(1:100, ncol=10, byrow=T)  # make some dummy data
b <- sqXplusX(a)  # transform a by sqXplusX, assign result to b
b # to view the result
```

- sqXplusX user function is now an R object, check its arguments and list it in the current workspace

```
> args(sqXplusX)
> ls()
> sqXplusX
```

Don't add brackets to see the definition of sqXplusX

Assigned or anonymous …
User functions

- Functions may be assigned a name, or anonymously created within an operation
  - Anonymous functions are really useful in `apply()` style procedures

```
apply(object, margin, function)
```

  - E.g. I have a 10 x 10 matrix and want to square each item, and add the item to itself

```
a<-matrix(1:100, ncol=10, byrow=T)
a # to view new object
apply(a, c(1,2), function(x) x^2+x)
```

`x` is transiently assigned each item of `a`, and this is passed as an argument to the anonymous function

1 means by rows, 2 means by columns [1st or 2nd margin]
c(1,2) means do both rows and columns

Functions occupy their own space
User functions

- Objects created in functions are not available to the general environment unless returned.
  - they are said to be out of scope
    - Scope relates to the accessibility of an object.
- A function can only return one object.
- Custom functions disappear when R sessions end, unless the function object is saved in an Rdata file or sourced from a script.
  - A really useful function could be added to your .Rprofile file, and would always be ready for you at launch
- You could also make a package
  - Beyond the scope of the beginners course!!!!

Script / function tips
User functions

- If your script repeats the same style command more than twice, you should consider writing a function
- Writing functions makes your code more easily understandable because they encapsulate a procedure into a well-defined boundary with consistent input/output
- Functions should not be longer than one-to-two screens of code, keep functions clean and simple
- Look at other functions to get ideas for how to write your own …
  - Display function code by entering the function's name without brackets.

File commands for extending
scripts & user functions

Generic file commands

```
dir(...,pattern="txt")
```

Retrieve working directory file listing filtered by pattern. Note pattern is a regular expression, not a shell wildcard

```
glob2rx("*.txt")
```

Changes wildcards to regular expressions!

```
> glob2rx("*.txt")
[1] "^.*\\.txt$"
```

```
unlink(...)
```

Remove (permanently) a file from system

```
system(...)
```

Execute a shell command from within R
Result can not be coerced to an object, only available to linux R

Text manipulation for extending scripts & user functions

- Text manipulation and file name mangling ... that's a technical term

`grep( pattern, object )`

  - If pattern is not found, grep returns a 0 length object.
    - Test for null with `is.null()`

`sub( pattern, replacement, object )`

`gsub( pattern, replacement, object)`

  - Sub replaces first occurrence only, gsub does them all.

`cat( "...", file=...)`

  - Outputs text to a file, or prints it on screen if file=""
    - cat requires "\n" to be given for new lines ... try ...

```
cat("Hello World!") ; cat("Hello World!",sep="\n") ; cat("Hello
   World!",sep="\n",file="world.txt")
```

  - cat is extremely useful for writing scripts or generating reports on-the-fly

Error reporting for extending scripts & user functions

- Your code should report errors if inconsistency is detected.

`stop(...)`

  - Stops execution of a function and reports a custom error message

`is.family(...)`

  - Functions that can be used to test for a variety of conditions place them inside `if` structures to check that all is well

```
if( !is.numeric(x) ){ stop ("Non numeric value entered.  Cannot
   continue.") }
```

If the object x is non numeric (e.g. Text has been entered when numbers were required), then stop execution and report message

The `is.family`

Temperature conversion exercise
User functions

- Centigrade to Fahrenheit conversion is given by
  - F = 9/5 C +32

  - Write a function that converts between temperatures.
    - The function will need two named arguments
      - *temperature* **`(t)`** *is numeric*
      - *units* **`(unit)`** *is character*
      - *They will need default values, e.g t=0, unit="c"*
    - The function should report an appropriate error if inappropriate values are given

```
if( !is.numeric(t) ) { .... }
if( !(unit %in% c("c","f")) ){...}
```

    - The function should print out the temperature in F if given in C, and vica versa

Functions with named arguments are defined with the following syntax

```
myFunc<-function(arg=defaultValue,…)
```

- Why not add a third scale?
  K=C+273.15

Example code: 12_convTemp.R

Building the solution

- It is difficult to write large chunks of code, instead start with something that works and build upon it
- E.g. to solve the temperature conversion exercise:
  - start with the function powXplusX (from some slides ago)
  - modify the argument names
  - delete the old code, for now just print out the input arguments
  - save the function file, load it into R and try it out
  - now add the two lines for input checking from the previous slide
  - try it out, see if passing a character for temperature gives the expected error
  - now try to convert C into F and print out the result
  - when that works, add the conversion from F to C
- If you get stuck, call us to help you !

Temperature conversion script

```
convTemp<-function(t=0,unit="c"){ # convTemp is defined as a new user function requiring two
arguments, t and unit, the default values are 0 and "c", respectively.

     if( !is.numeric(t) ){
          stop("Non numeric temparture entered") # Exception error if character given for
temperature
     }

     if(!(unit %in% c("c","f","k"))){
          stop("Unrecognized temperature unit. \n Enter either (c)entigrade, (f)ahreneinheit
or (k)elvin") # Exception error if unrecognized units entered
     }
# Conversion for centigrade
     if(unit=="c"){
          fTemp <- 9/5 * t + 32
          kTemp <- t + 273.15
          output <- paste(t,"C is: \n",fTemp,"F \n",kTemp,"K \n")
          cat(output)
     }
# Conversion for Fahrenheit
     if(unit=="f"){
          cTemp <- 5/9 * (t-32)
          kTemp <- cTemp + 273.15
          output <- paste(t,"F is: \n",cTemp,"C \n",kTemp,"K \n")
          cat(output)
     }
# Conversion for Kelvin
     if(unit=="k"){
          cTemp <- t - 273.15
          fTemp <- 9/5 * cTemp + 32
          output <-paste(t,"K is: \n",cTemp,"C \n",fTemp,"F \n")
          cat(output)
     }
}
```

"\n" -> puts text on a new line

Units must be entered in quotes, as it's a character object

```
> convTemp(t=-273,unit="c")
-273 C is:
 -459.4 F
 0.149999999999977 K
```

Example code: 12_convTemp.R

Advanced data processing

3

Combining data from multiple sources
Gene clustering example

- R has powerful functions to combine heterogeneous data into a single data set
- Gene clustering example data:
  - five sets of differentially expressed genes from various experimental conditions
  - file with names of experimentally verified genes
- Gene clustering exercise:
  1. combine this dataset into a single table and cluster to see which conditions are similar
  2. repeat the clustering but only on a subset of experimentally verified genes

Combining gene tables

- input files have two columns: gene names and fold change
- we want to combine all five tables into a single table, with 0 for missing values

LpR2	3.5795
fs(1)h	3.1376
CG6954	2.7492
Psa	2.7012
zfh2	2.6247
Fur1	2.4413
ct	2.3804
S	2.3674
nux	2.3574
RhoBTB	2.26
CG14889	2.1735
oc	2.1421
pros	2.0882
Kr-h1	-2.0447
CG5149	-2.1521
tna	-2.2102
CG14888	-2.4346
CG31368	-2.4793
Trim9	-2.616
Awd	-3.0095

+

Psa	3.8529
vnd	3.6457
ct	3.201
fs(1)h	3.1489
btd	3.1229
zfh2	2.8421
RhoBTB	2.6022
pros	2.5679
CG1124	2.5475
S	2.5424
oc	2.5111
Fur1	2.43
PHDP	2.304
CG31241	2.2602
nux	2.2232
CG14889	2.1752
CG31163	2.1606
HmgZ	2.0795
svp	-2.0404
TER94	-2.1807
corto	-2.3481
olf413	-2.4404
brat	-2.7256
CG31368	-2.7293
mub	-2.9555
Awd	-3.1413
lola	-3.8882

+

lola	3.0121
CG31368	2.8063
Kr-h1	2.7262
svp	2.7055
mub	2.6475
CG5149	2.5248
run	2.4759
tna	2.4302
CG6954	2.4235
CG11153	2.3045
Awd	2.2295
CG6919	2.1324
CG14888	2.067
Psa	-2.0276
nux	-2.093
fs(1)h	-2.141
CG1124	-2.155
Fur1	-2.1588
S	-2.2539
corto	-2.2618
oc	-2.3017
CG14889	-2.4393
zfh2	-2.5884
HmgZ	-3.6328
btd	-3.7627
brat	-3.7716

+

lola	3.3019
CG6919	2.9965
CG31368	2.817
CG5149	2.7675
Kr-h1	2.7647
TER94	2.6286
tna	2.5748
CG11153	2.4795
run	2.3831
CG14888	2.0938
S	-2.0243
nux	-2.0668
oc	-2.3437
corto	-2.5556
fs(1)h	-2.6211
brat	-2.9904
ct	-3.3404
zfh2	-4.4947
CG6954	-4.7244

+

brat	5.2812
ct	4.828
CG31163	4.3345
LpR2	3.6882
vnd	3.6866
zfh2	3.5314
pros	3.4307
Psa	3.3998
fs(1)h	3.3869
CG31241	2.9973
HmgZ	2.9226
Fur1	2.7469
RhoBTB	2.7189
oc	2.6543
Toll-7	2.6161
nux	2.5975
CG14889	2.3054
S	2.2324
CG1124	2.0216
Kr-h1	-2.1439
tna	-2.1793
CG5149	-2.1892
run	-2.2194
Trim9	-2.251
olf413	-2.3821
btd	-3.0293
CG6919	-3.3719

Gene clustering
Script walkthrough 1

- To make the big table we first need to find out all the genes present in at least one of the files
- Make sure not to use factors in read.delim()

```
# start with en empty collection of genes
genes <- c()
for( fileNum in 1:5 ){
   # load in files 13_DiffGenes1.tsv ...
   t <- read.delim(paste("13_DiffGenes", fileNum, ".tsv", sep=""),
                   as.is=TRUE, header=FALSE)
   # label the input columns to help code readability
   names(t) <- c("gene", "expression")
   genes <- union(genes, t$gene)
}

# for tidiness order our genes by name
genes <- sort(genes)

genes # show all genes
```

when loading in character data use **as.is=T** to prevent it being converted to factors!

union() is a set operation, combines two vectors by eliminating duplicates. There are also intersect() and setdiff()

Example code: 13_geneClustering.R

Gene clustering
Script walkthrough 2

- Using the complete list of genes, we can create the big table and fill in the values:

```
# make the destination table [rows = unique genes, cols = file numbers]
values <- matrix(0, nrow=length(genes), ncol=5)
rownames(values) <- genes # name the rows with the complete gene names

for(fileNum in 1:5){
  # read in the file again
     t <- read.delim(paste("13_DiffGenes", fileNum, ".tsv", sep=""),
           as.is=T, header=F)
     names(t) <- c("gene", "expression")

     # match the names of the genes to the rows in our big table
     index <- match(t$gene, rownames(values))
     # copy the expression levels
     values[index,fileNum] <- t$expression
}
```

match() returns the index of first argument in the second, i.e. index of input file genes in the big table

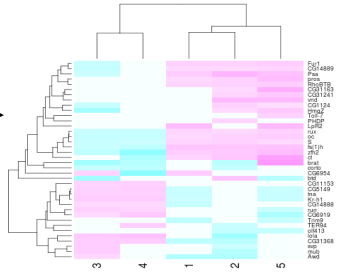
we use index to pick the rows in such way that they match the gene order in the input file

Gene clustering
Script walkthrough 3

- Now we can do hierarchical clustering:

```
heatmap(values, scale="none", col = cm.colors(256))
```

Values from the matrix are colour-coded. Rows and columns are re-arranged according to similarity



Gene clustering
Script walkthrough 4

• In a second part of our analysis, we want to produce the same heatmap but only based on a list of experimentally verified genes

• The problem is data is not formatted in the most convenient way:

genes	citation
oc,run,RhoBTB,CG5149,CG11153,S,Fur1	Segal et al, Development 2001
tna,Kr-h1,rux	Krejci et al, Development 2002

Gene clustering
Script walkthrough 5

• We load in this table, and only extract the gene names, then we use them to select a subset of values matrix

```
# load in the tab-delimited file with genes and citations
t.exp <- read.delim("13_ExperimentalGenes.tsv", as.is=T)
# split all gene names by "," and then flatten it out into a single vector
experim.genes <- unlist( strsplit(t.exp$genes, ",") )
```

unlist() flattens out a nested list into a single vector

strsplit() splits a vector of strings by a custom split character (","), the results is a list of split values for each element of input vector

```
# redo the heatmap by using just the genes in the experimentally verified set
is.experimental <- rownames(values) %in% experim.genes
heatmap(values[ is.experimental, ], scale="none", col = cm.colors(256))
```

Gene clustering review

• We load in the five tables twice - first to collect gene names, then to load expression values

• Based on expression table (values) we construct a clustered heatmap first on the whole set of genes, then on a selected subset

• Go through the code, try it out it and understand it

• Try answering the following questions:
  - what is rownames(values) ?
  - why is rownames(values)[index] and t$gene giving the same output?
  - what is a difference between rownames(values) %in% experim.genes and experim.genes %in% rownames(values)

Example code: 13_geneClustering.R

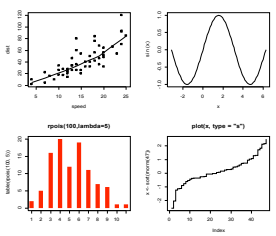
Graphics

4

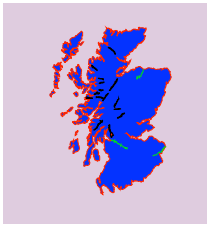
Starting out with R graphics
Graphics

- R provides several mechanisms for producing graphical output
  - Functionality depends on the level at which the user seeks interaction with R
    - graphics systems, packages, devices & engines
- High level graphics
  - Functions compute an appropriate chart based up on the information provided. Optional arguments may tailor the chart as required
    - Interaction is at traditional graphics system level. The user isn't required to know much about anything
- Low level graphics
  - The user interacts with the drawing device to build up a picture of the chart piece by piece.
    - This fine granular control is only required if you seek to do something exceptional
- R graphics produces plots using a painter's model
  - Elements of the graph are added to the canvas one layer at a time, and the picture built up in levels. Lower levels are obscured by higher levels, allowing for blending, masking and overlaying of objects.

High level vs. Low level plotting
Graphics



High level plotting
`example(plot)`

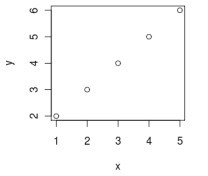


Low level plotting
(Scotland by blighty package)

Essential plotting - plot()

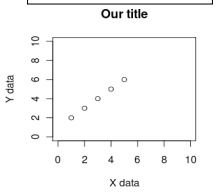
• plot() is the main function for plotting, it takes x,y values to plot and also lots of graphical parameters (see **?par** for all of them)

default plotting




```
x <- 1:5
y <- 2:6
plot(x,y)
```

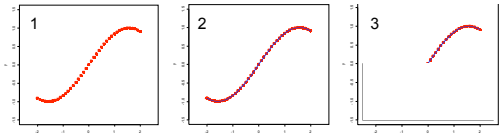
custom plotting




```
x <- 1:5
y <- 2:6
plot(x,y, xlab="X data", ylab="Y
data", xlim=c(0,10), ylim=c(0,10),
main="Our title")
```

R graphics uses a painter's model

```
x <- seq(-2, 2, 0.1)
y <- sin(x)
```




```
plot(y~x, ylim=c(-1.5,1.5),
xlim=c(-2.5,2.5),
col="red" ,pch=16, cex=1.4)
```

```
lines(y~x, ylim=c(-1.5,1.5),
xlim=c(-2.5,2.5), col="blue",
lty=1,lwd=2)
```

```
rect(-2.5,0,2.5,-1.5,
col="white", border="white")
```

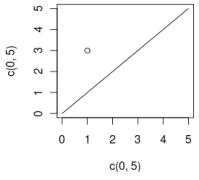
xlim, ylim = axis limits
col = line colour
pch = plotting character [`example(points)`]
cex = character expansion [scaling factor]

lty = line type
lwd = line width
rect = rectangle

Example code:
14_painterModel.R

Plotting x,y data - plot(), points(), lines()

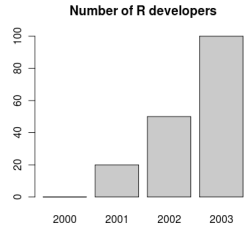
• **plot()** is used to start a new plot, accepts x,y data, but also data from some objects (like linear regression). Use the parameter **type** to draw points, lines, etc (see **?plot**)
• **points()** is used to add points to an existing plot
• **lines()** is used to add lines to an existing plot




```
plot(c(0, 5), c(0, 5), type="l") # draw as line from (0,0) to (5,5)
points(1, 3) # add a point at 1,3
```

Making bar plots - barplot()

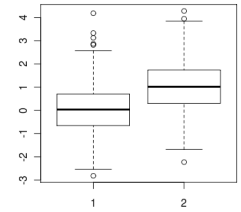
• visualizing a vector of data can be done with bar plots, using function **barplot()**




```
data <- c("2000"=0, "2001"=20, "2002"=50, "2003"=100)
barplot(data, main="Number of R developers")
```

Making box plots - boxplot()

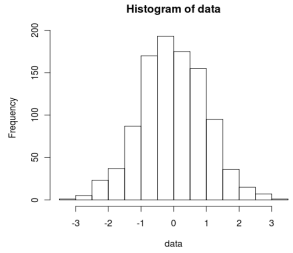
• when a spread of data needs to be visualised, we can use boxplots with function **boxplot()**




```
data1 <- rnorm(1000, mean=0)
data2 <- rnorm(1000, mean=1)
boxplot(data1, data2)
```

Making histograms - hist()

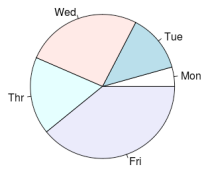
• when we need to look at the distribution of data, we can visualize it using histograms with function hist()




```
data <- rnorm(1000)
hist(data)
```

Pie charts - pie()

• to visualise percentages or parts of a whole we can use pie charts with function **pie()**




```
data <- c("Mon"=1, "Tue"=3, "Wed"=6, "Thr"=4, "Fri"=9)
pie(data)
```

Typical plotting workflow

- Set the plot layout and style - `par()`
  - Set the number of plots you want per page
  - Set the outer margins of the figure region
    - The distance between the edge of the page and the figure region, or between adjacent plots if there are multiple figures per page
  - Set the inner margins of the plot
    - The distance between the plot axes and the labels & titles
  - Set the styles for the plot
    - Colours, fonts, line styles and weights
- Draw the plot - `plot(x,y, …)`

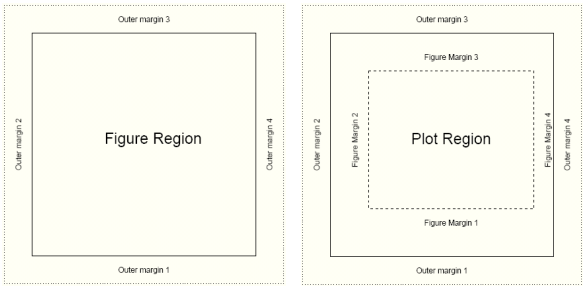
Setting graphics layout and style - par()

`par()` Top level graphics function

- parameter specifies various page settings. These are inherited by subordinate functions, if no other styles are set.
  - Specific colours and styles may be set globally with `par`, but changed ad hoc in plotting commands
  - The global setting will remain unchanged, and reused in future plotting calls.
- `par` sets the size of page and figure margins
  - Margin spacing is in 'lines'
- `par` is responsible for controlling the number of figures that are plotted on a page
- `par` may set global colouring of axes, text, background, foreground, line styles (solid/dashed), if figures should be boxed or open etc. etc.

type `par()` to get a list of top down settings which may be set globally

Page settings with `par`
Graphics



`par(mfrow=c(1,1))`
one figure on page
`par(oma=c(2,2,2,2))`
equal outer margins

`par(mar=c(5,4,4,2))`
Sets space for x & y labels, a main title, and a thin margin on the right

Numbering: bottom, left, top, right

Page layout plot exercise
Graphics

`par(mfrow=c(2,2))`
- 2 x 2 figures per page

`par(oma=c(1,0,1,0))`
- 1 line spacing top and bottom

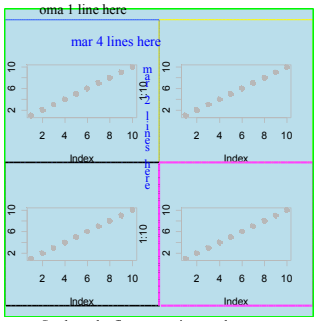
`par(mar=c(4,2,4,2))`
- 4 lines at bottom & top
- 2 lines left & right

`par(bg="lightblue",fg="darkgrey")`
- light blue background
- dark grey spots

`par(pch=16,cex=1.4)`
- Large circles for spots
- Execute 4 times with different colors:

```
plot(1:10)
box("figure",lty=3,col="blue")
```
- Draw a blue dashed line around plot

`box("outer",lty=1,lwd=3, col="green")`
- Draw a green solid line around figure



See how the figure margins overlap
Using painter's model

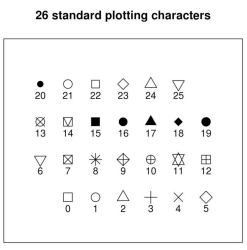
15_parExample.R

Plotting characters for plot()
size and orientation

`pch= ...`
Sets one of the 26 standard plotting character used.
Can also use characters, such as "."

`cex= ...`
Character expansion. Sets the scaling factor of the printing character

`las= ...`
Axes label style. 1 normal, 2 rotated 90º
4 styles (0-3)



Plotting characters exercise
Graphics

16_plottingChars.R

```
xCounter<-1
yCounter<-1
plotChar<-0

plot(NULL, xlim=c(0,8),
ylim=c(0,5),xaxt="n",
    yaxt="n",ylab="",xlab=""
    ,main="26 standard plotting
characters")

while (plotChar < 26){
  if(xCounter < 7){
    xCounter <- xCounter+1
  } else {
    xCounter <- 1
    yCounter <- yCounter+1
  }

  points(xCounter,yCounter,pch=plotChar,
cex=2)
  text(xCounter,(yCounter-0.3),plotChar)
  plotChar <- plotChar+1
}
```

X-Y coordinates,
Plotting character index counter

Sets up an empty plotting area.
Axis scale limits, xlim, ylim
Don't draw axis ticks, xaxt, yaxt="n"
Don't annotate axis, xlab, ylab=""
Set a main title, main

We want to print the characters in a 7 x 4 grid. The if statement sets up the character plotting coordinates such that each time x =7, make it 1 again and increment the y axis by 1 at the same time

While loop counts up to 25
(0 to 25 = 26 iterations)
And cycles through each pch available

Annotating the plot

- plot accepts main title, subtitle, X label, Y label as standard arguments

```
plot(x, y, main="...", sub="...", xlab="...", ylab="...")
```

`mtext(text="…", side= …)`
- allows text to be written directly into the margin of a plot

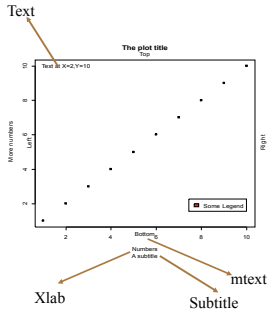
`text(x,y,labels="…")`
- allows text to be written in the plot at x,y

`legend(x,y, legend=…)`
- produces a legend for the plot

Appreciating drawing coordinates

- How do we know where to place items within the plot region when building up our customized graphs?
- Most of the time we can specify X,Y coordinates.
  - R calculates sensible pixel coordinates of plots from the data we provide. We don't need to worry about pixels, centimetre distances etc.
- `locator(…)`
  - Returns x,y coordinates from a mouse click within a plot
  - good for working out where to place legend items
- `identify(…)`
  - provides an id tag for the closest plotted point to a mouse click
  - useful if you want to label points on a chart
- `xy.coords(…)`
  - translates x,y coordinates into pixel coordinates
- Margin spacing is in lines
  - The exact distance is a factor of font family, style and size
    - Text may appear bunched or squashed if sufficient distance is not left between the axes and the caption

Building up a plot
Graphics



17_buildingAplot.R

R code

```
par(mfrow=c(1,1))
par(bg="white",fg="black",cex=1)
par(oma=c(1,1,1,1))
par(mar=c(5,4,4,2)+0.1)

plot(1:10,main="The plot title",
sub="A subtitle", xlab="Numbers",
        ylab="More numbers")

mtext(c("Bottom", "Left", "Top",
"Right"), c(1,2,3,4),     line=.5)

text(2,10,"Text at X=2,Y=10")
legend(locator(1),"Some
Legend",fill="red")
```

Adding legend …
Don't forget to mouse click!

align text left, right & centre with adj=(i,j) i.e centre is adj=(0.5,0.5), left is adj=(1,0) and right is adj=(0,1)

Plots with custom axes
Graphics

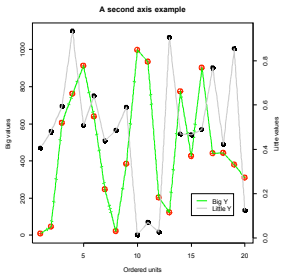
- R `plot` doesn't support multiple Y axis by default
  - You have to make additional axes yourself!
- Adding custom axis

```
axis(side=, at=, labels=, …)
```

- If you want to specify custom axes, make sure you turn off the automatic axes in the plot / points call

```
plot( …, axes=F)
```

Adding a second Y axis
Graphics



The trick

1. plot first Y series
2. use `par(new=T)` to overlay a second figure region
3. plot second series without axes
4. `axis(side=4, …)` to add second Y axis
5. `mtext(side=4, …)` to label second Y

Example: The second Y series
Graphics

18_secondYaxis.R

```
x1<-1:20
y1<-sample(1000,20)
y2<-runif(20)
y2axis<-seq(0,1,.2)
```
Demo data

```
par(mar=c(4,4,4,4))
```
Set up equivalent figure margins

```
plot(x1,y1,type="p",pch=10,cex=2,col="red",
     main="A second axis example",
     ylab="Big values",ylim=c(0,1100),
     xlab="Ordered units")
```
Plot and label first Y series

```
points(x1,y1,type="l",lty=3,lwd=2,col="green")
```
Connect dots with a line

```
par(new=T)
```
Overlay a second plot region

```
plot(x1,y2,type="p",pch=20,cex=2,col="black",axes=FALSE,bty="n",xlab="",ylab="")
points(x1,y2,type="l",lty=2,lwd=2,col="grey")
```
Plot second Y series, but suppress labels

```
axis(side=4,at=pretty(y2axis))
mtext("Little values",side=4,line=2.5)
```
Anotate second Y axis

```
legend(15,0.2,c("Big Y","Little Y"),lty=1,lwd=2,col=c("green","grey"))
```
Add legend, note X,Y is on second Y axis scale

Use of colour in R
Graphics

- Colour is usually expressed as a hexadecimal code of Red, Green, and Blue counterparts
  - No good for humans.
- R supports numerous colour palettes which are available through several "colour" functions.
  - `colours()` # get inbuilt names of known colours
    - RGB primaries may take on a decimal intensity value of 0 to 255
      - 255 is #FF in hexidecimal
        - White is #FF FF FF
        - Black is #00 00 00
  - `rgb()` # converts red green blue intensities to colour
    - Strangely, likes decimalized intensities (ie. 0 is black, 1 is white)

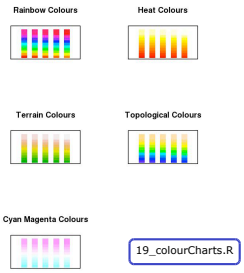
```
> rgb(1,1,1)
[1] "#FFFFFF"

> par(mfrow=c(2,2))
> plot(1:10,col="#FF00FF")
> plot(1:10,col=rgb(1,0,1))
> plot(1:10,col="magenta")
```

Colour Ramps & Palettes
Graphics

•Heatmaps use colour depth to convey data values. Cold colours are typically low values, and light colours are high state values. This is a colour ramp.

•R supports numerous graded colour charts. Specify *n*, to set the number of gradations required in the palette

```
rainbow(n)
heat.colors(n)
terrain.colors(n)
topo.colors(n)
cm.colors(n)
```



19_colourCharts.R

You can specify a user defined palette of indexed colours:

```
palette(rainbow(7)) # creates 7 indexed colours (1:7) based on
                    # rainbow palette  R O Y G B I V !!!
```

Colour packages: RColorBrewer
Graphics

- This add on package provides a series of well defined colour palettes. The colours in these palettes are selected to permit maximum visual discrimination
- Access the RColorBrewer library functions ...

```
library("RColorBrewer")
```

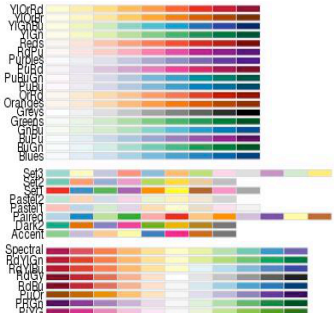
- Check out the available palettes

```
display.brewer.all(n=NULL, type="all", select=NULL, exact.n=TRUE)
```

- Define your own palette based on one of RColorBrewers'

```
myCol<-brewer.pal(n,"...") # n=number of colours, "..." is the palette name
```

RColorBrewer named palettes
Graphics



Saving plots to files

- Unless specified, R plots all graphics to the screen
- To send plots to a file, you need to set up an appropriate graphics device …

```
postscript(file="a_name.ps", …)
pdf(file="…pdf", …)
jpeg(file=" …jpg", …)
png(file=" ….png", …)
```

- Each graphics device will have a specific set of arguments that dictate characteristics of the outputted file
  - `height=, width=, horizontal=, res=, paper=`
    - Top tip: jpg, A4 @ 300 dpi, portrait, size in pixels
    - `jpg(file="my_Figure.jpg", height=3510, width=2490, res=300)`
    - Postscript & pdf work in inches by default, A4 = 8.3" x 11.7"
- Graphics devices need closing when printing is finished

```
dev.off()
```

for example:

```
png("tenPoints.png", width=300, height=300)
plot(1:10)
dev.off()
```

Thoughts when plotting to a file
Graphics

- Its very tempting to send all graphical output to a pdf file. Caution!
  - For high resolution publication quality images you need postscript. Set up postscript file capture with the following function

```
postscript("a_file.ps",paper="a4")
```

    - postscript images can be converted to JPEG using ghostscript (free to download) for low resolution lab book photos and talks
  - PDF images will grow too large for acrobat to render if plots contain many data points (e.g. Affymetrix MA plots)
  - Automatically send multiple page outputs to separate image files using …`file="somename%02d.jpg"`
  - Don't forget to close graphics devices (i.e. the file) by using
    - `dev.off()`

Plotting exercise
Graphics

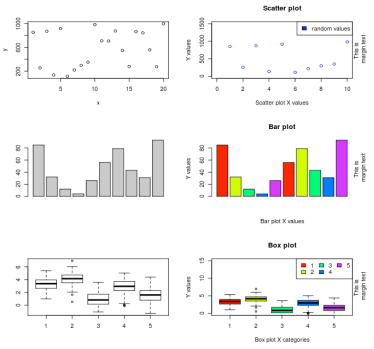
- Exercise:
  - Make a full A4 page figure comprising of 6 plots: 2 each of XY plot (`plot()`), barchart (`barplot()`) and box plots (`boxplot()`)
  - The two version of each plots should consistent of: the default plot and a customised plot (change for instance colours, range, captions...)
  - Output the completed 6-panel figure to: screen, jpeg, postscript and pdf file

- Suggested route to solution:

  1. Generate some plotting data appropriate for each type of plot
  2. Write the code to produce the six plots, once plotting the data by using default plotting, one with some customisations you want
  3. To output the plot to screen, jpeg, postscript and pdf you will need to redo the plot multiple times - create a function to do a plotting and call it by redirecting graphical output to screen, jpeg file, poscript file and pdf file

20_6PanelPlotScript.R

6 Panel plots exercise
Graphics



References

- Official documentation on:
  - http://cran.r-project.org/manuals.html
- A good repository of R recipes:
  - Quick-R: http://www.statmethods.net/
- Don't forget that many packages come with tutorials (vignettes)
- Website of this course:
  - http://logic.sysbiol.cam.ac.uk/teaching/Rcourse/
- R forums (stackoverflow & official):
  - http://stackoverflow.com/questions/tagged/r
  - http://news.gmane.org/gmane.comp.lang.r.general
- Plenty of textbooks to choose from, comprehensive list + reviews:
  - http://www.r-project.org/doc/bib/R-books.html