
DAY 2. A beginners guide to solving biological problems in R

Robert Stojnić (rs550), Laurent Gatto (lg390),
Rob Foy (raf51), John Davey (jd626) and Dávid Molnár (dm516)

Course material:

<http://logic.sysbiol.cam.ac.uk/teaching/Rcourse/>

Original slides by Ian Roberts and Robert Stojnić

Day 2

Schedule

1. Writing scripts
2. Writing functions
3. Data analysis examples
4. Graphics

Writing custom scripts for data 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
 - We will use RStudio

Colony forming experiment

- We have been asked by some collaborators to analyse some trial data to see if an experiment will work.
- We are interested in the behaviour of a gene, X, which is involved in a cell proliferation pathway.
- This pathway causes cells to proliferate in the presence of a compound, Z.
- Gene X turns the pathway off, reducing cell proliferation.

- Our collaborators want to test what happens when we knock down X in the presence of Z.
- To do this, they want to grow cell colonies in the presence of Z, with or without X, and count the number of colonies that result.

Initial trial

- Our collaborators have sent us a first batch of test data, growing colonies in different concentrations of compound Z.
- Does increasing concentration of Z have an effect on colony growth?
- We want to do the following:
 - Load the data into R
 - Plot the data to inspect it
 - Calculate an Analysis of Variance to see if growth is influenced by Z concentration
 - Calculate the mean growth for each level of Z concentration, to see the direction of change
 - (We will ignore full post hoc testing)

Initial trial exercise

- The initial trial data is in the file 2.1_colony_trial.xls. This is an Excel file and the data is not in the right format for R. Enter the data into a plain text file in a data frame format, and load it into R.
- Plot the data using a formula. Recall how we did this yesterday with linear modelling:

`plot(y~x)`

- Calculate an analysis of variance for the data. The R function for ANOVA is `aov()`, which works like `lm()` for linear modelling – recall this from yesterday:

`summary(lm(y~x))`

- There are four concentrations of Z, and each concentration has been replicated three times. What is the mean colony count for each concentration? See if you can figure out a way to calculate this with what we learned yesterday. You will need to use logical indexing and you may want to use a for loop.

Importing data

- In the Excel file, the data has this format:
- But this is not a data frame format, where columns are variables and rows are observations of those variables.
- There are three variables, Z, Replicate, and Count. We need to reshape the data with these three columns.
- We can do this in Excel, and then save the file in CSV format, or we can just type up the results in a CSV file ourselves.
- Once we have a CSV file, we can load it with read.csv:

```
colony<-read.csv("2.1_colony_trial.csv")
```

Replicate	1	2	3
No Z	150	180	223
Low Z	87	40	53
Medium Z	5	1	9
High Z	0	0	0

Z	Replicate	Count
None	1	150
None	2	180
None	3	223
Low	1	87
Low	2	40
Low	3	53
Medium	1	5
Medium	2	1
Medium	3	9
High	1	0
High	2	0
High	3	0

Plotting

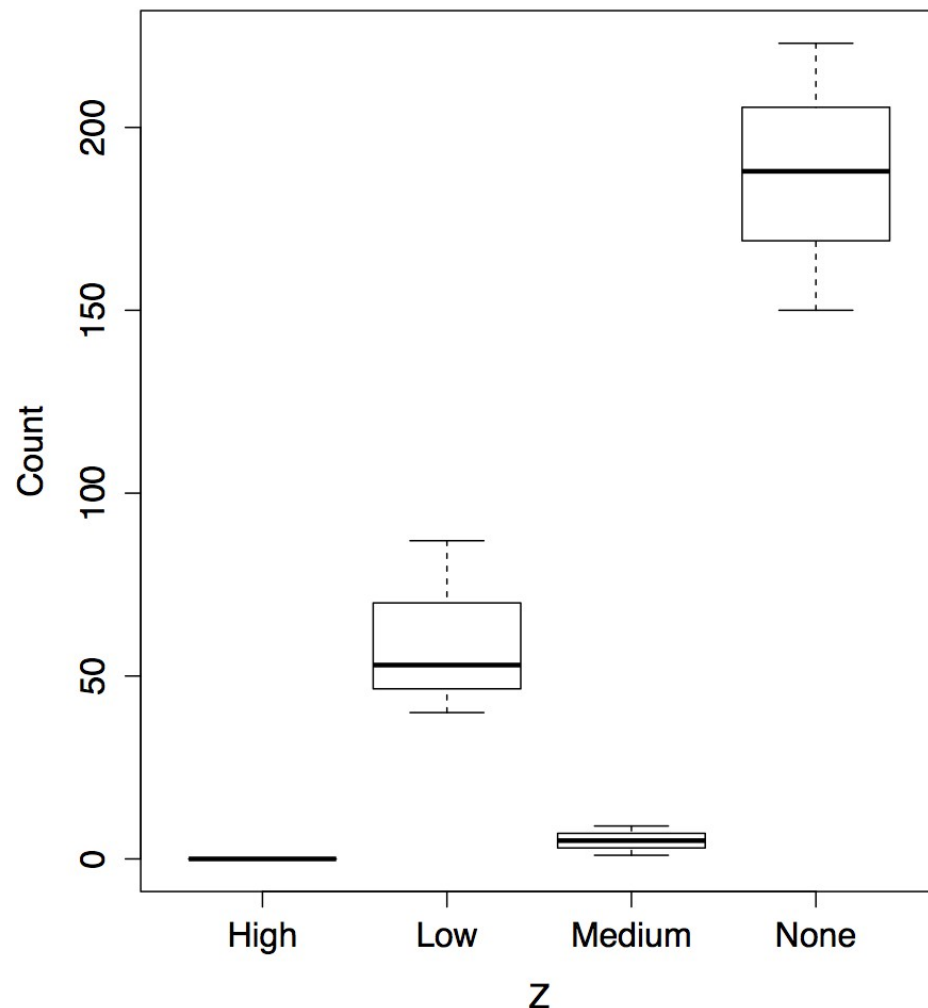
We want to plot the colony growth in response to changing Z concentration.

Z is the explanatory variable, and Count is the response variable.

We don't want to plot replicates separately here, but get R to summarise each Z concentration over all replicates.

We can call plot using the same formula syntax we learnt yesterday:

```
plot(colony$Count~colony$Z)
```



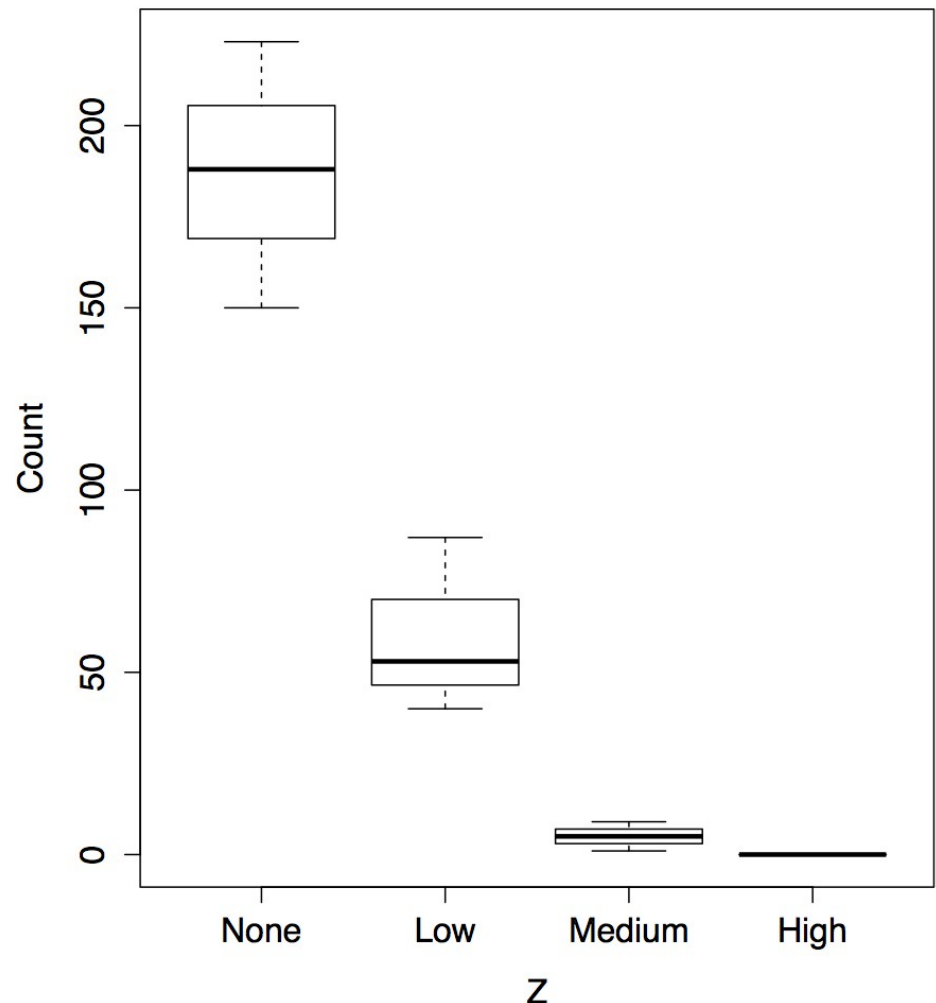
Plotting

We can improve on this. Firstly, we want to order the Z categories. Z is a factor, so we need to supply new levels to this factor in the colony data frame:

```
colony$Z<-factor(colony$Z,  
  levels=c("None", "Low", "Medium", "High"))
```

Second, we can tell the plot command which data frame to use, rather than using the dollar operator:

```
plot(Count~Z, data=colony)
```



Analysis of Variance

We can use the same formula syntax to calculate an analysis of variance:

```
colony.aov<-aov(Count~Z, colony)
```

```
summary(colony.aov)
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Z	3	68154	22718	46.89	2.02e-05 ***
Residuals	8	3876	484		

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This tells us what we can already see from the plot, that there is a highly significant relationship between Z concentration and colony growth.

We would like to investigate this relationship. For example, we might want to calculate the mean colony count for each concentration of Z.

Calculating group means

We can calculate a mean for a particular group like this:

```
> mean(colony[colony$Z=="None",]$Count)
```

```
[1] 187
```

```
> mean(colony[colony$Z=="Low",]$Count)
```

```
[1] 60
```

```
> mean(colony[colony$Z=="Medium",]$Count)
```

```
[1] 5
```

```
> mean(colony[colony$Z=="High",]$Count)
```

```
[1] 0
```

We could generalise this with a for loop:

```
for (z in levels(colony$Z)) {  
  print(mean(colony[colony$Z==z,]$Count))  
}
```

```
[1] 187
```

```
[1] 60
```

```
[1] 5
```

```
[1] 0
```

But there is a better way.

The tapply function

a brief digression

- The apply family of functions allow us to group data by variable and calculate something for each group.
- 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
```

- How can we get mean height of males and females separately?

`tapply()` lets us do exactly this:

- `tapply(data$height, data$gender, mean)`
 data groups function

Using tapply on colony

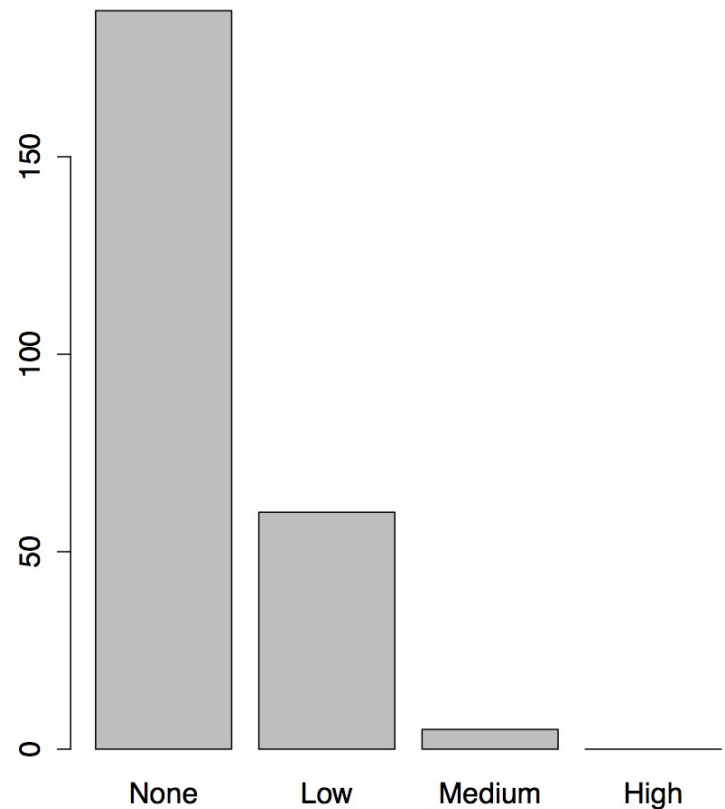
- We can use tapply to calculate group means on colony like this:

```
> colony.means<-tapply( colony$Count, colony$Z, mean )
```

```
> colony.means
```

None	Low	Medium	High
187	60	5	0

```
> barplot(colony.means)
```



A complete script

We now have a complete script to analyse this data:

```
# Load data, order Z and plot
colony<-read.csv("2.1_colony_trial.csv")
colony$Z<-factor(colony$Z,c("None","Low","Medium","High"))
plot(Count~Z,colony)

# Analysis of Variance
colony.aov<-aov(Count~Z,colony)
print(summary(colony.aov))

# Calculate group means
colony.means<-tapply(colony$Count,colony$Z,mean)
print(colony.means)
barplot(colony.means)
```

We need to print the results we want to see on screen, otherwise they will not be output.

Make sure you can source your commands (or the file 2.1_colony_1.R) from Rstudio and generate the results and plot.

Knocking down gene X: revising the script

As the trial worked, our collaborators have gone ahead with an experiment to knock down gene X in the same concentrations of Z. On our request, they have delivered the data in a data frame format in a CSV file: 2.1_colony_Run1Counts.csv.

They want us to see if knocking down X affects colony growth.

Because we saved our analysis in a script, we can rerun the same script to analyse the data, just by changing the name of the file we are loading.

Run your script on this new data file and confirm that you can calculate an ANOVA and group means for this new data set.

Knocking down gene X: revising the script

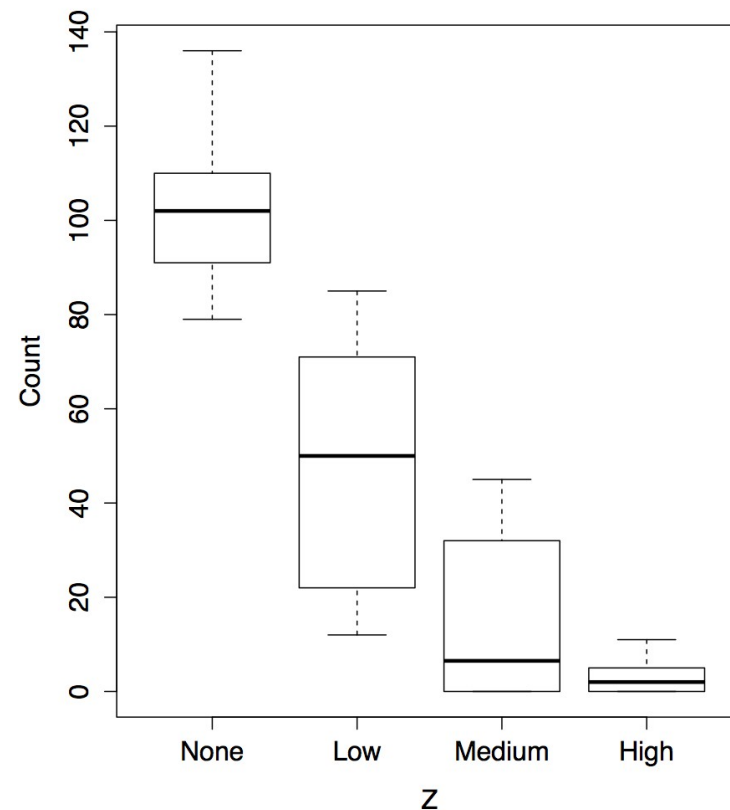
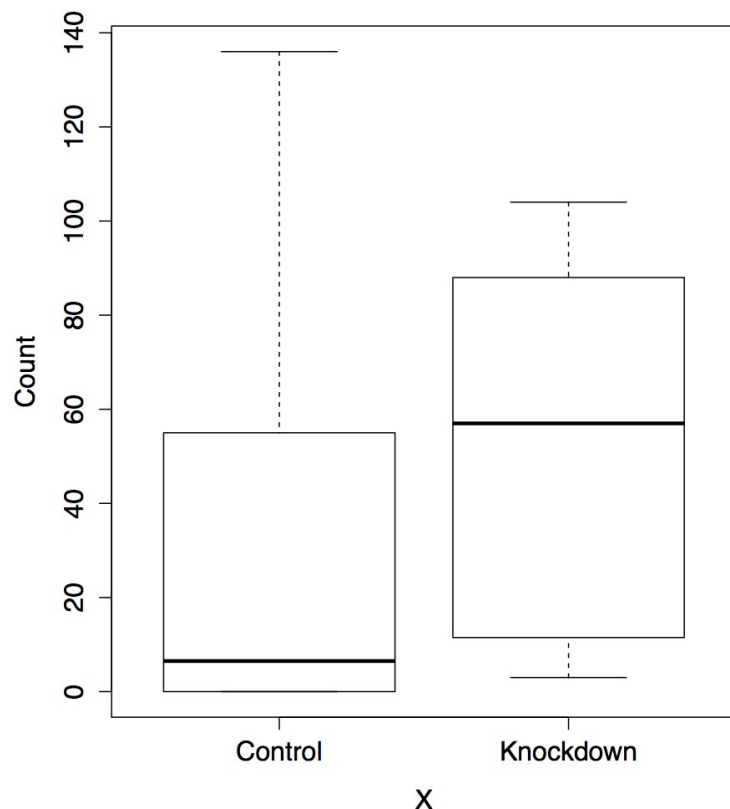
Our current script only analyses Z, not X. We need to modify it to include X and see how both X and Z influence colony growth.

1. We need to include X and the interaction between Z and X in our formulae for plotting and for ANOVA. Look up the 'Modelling formulae' slide from Day 1 to see how to do this.
2. What does **plot** do with a formula including both X and Z? Try using **boxplot** instead. What difference does it make if you change the order of X and Z?
3. We need to include both X and Z in our call to **tapply**. Modify the call to **tapply** by changing the second argument, which should be a list containing the data for both X and Z.
4. Plot the group means you calculated with **tapply** using **barplot**. Plot bars for different conditions *beside* each other, not on top of each other. Check the help page for an option to do this.

Plotting interactions

Including interactions in formulae is straightforward, but **plot** doesn't show us the interaction, only the main effects:

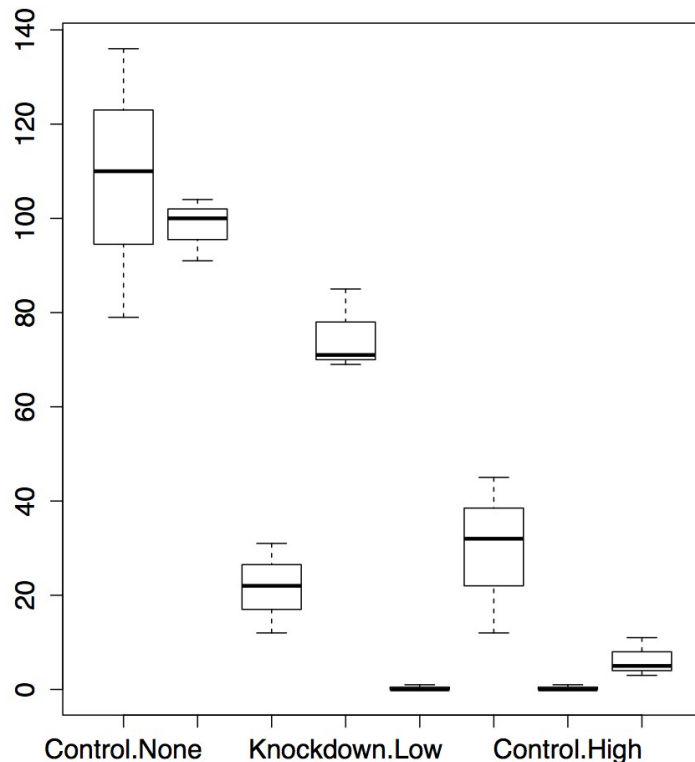
```
> plot(Count~X*Z,colony)
```



Plotting interactions

To get a sense of what's happening with the interactions, use **boxplot**:

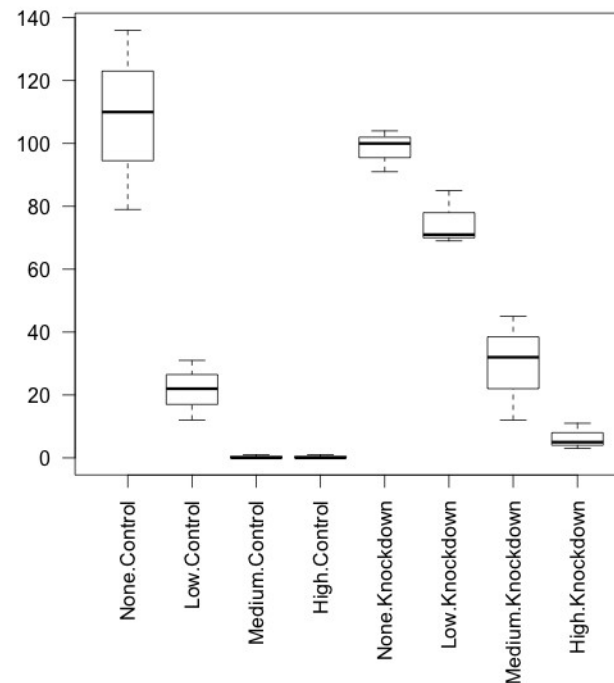
```
> boxplot(Count~X*Z,colony)
```



To make the labels visible, we'll use some graphics commands to increase the size of the lower margin and make the x-axis labels vertical (full details on this this afternoon):

```
> par(oma=c(6,2,2,2))
```

```
> boxplot(Count~X*Z,colony,las=2)
```



It looks like knocking down X increases colony growth, except when Z is completely absent.

Analysis of variance with interactions

Including interactions in the analysis of variance is straightforward:

```
> colony.aov<-aov(Count~X*Z,colony)
```

```
> print(summary(colony.aov))
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	
X	1	2321	2321	14.072	0.00174	**
Z	3	36150	12050	73.067	1.48e-09	***
X:Z	3	3441	1147	6.954	0.00329	**
Residuals	16	2639	165			

Signif. Codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Not only do X and Z have a significant effect on colony growth individually, but there is also a significant interaction between them.

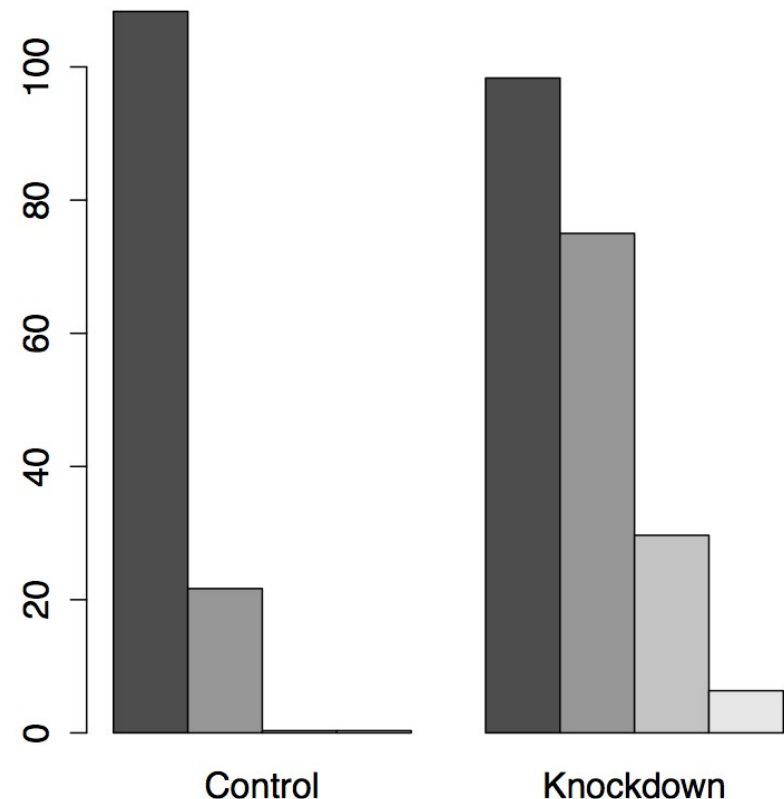
tapply with multiple variables

Including Z in the call to tapply is a little fiddly, but easy when you know how. Use the **beside** option in the call to **barplot**. (What happens if you put X first in the list?)

```
> colony.means<-tapply(colony$Count,list(colony$Z,colony$X),mean)
> print(colony.means)
```

	Control	Knockdown
None	108.3333333	98.3333333
Low	21.6666667	75.0000000
Medium	0.3333333	29.6666667
High	0.3333333	6.3333333

```
> barplot(colony.means,beside=TRUE)
```



Complete script: first revision

Our script now looks like this (see 2.1_colony_2.R):

```
# Load data, order Z and plot
```

```
colony<-read.csv("2.1_colony_Run1Counts.csv")
```

```
colony$Z<-factor(colony$Z,c("None","Low","Medium","High"))
```

```
par(oma=c(6,2,2,2))
```

```
boxplot(Count~Z*X,colony,las=2)
```

```
# Analysis of Variance
```

```
colony.aov<-aov(Count~X*Z,colony)
```

```
print(summary(colony.aov))
```

```
# Calculate group means
```

```
colony.means<-tapply(colony$Count,list(colony$Z,colony$X),mean)
```

```
print(colony.means)
```

```
barplot(colony.means,beside=TRUE)
```

Incorporating multiple runs: second revision

As it looks like there is an interaction between Z and X, our collaborators have repeated the experiment twice more, to increase the sample size. They have delivered two more data files, one for each extra run, in the files 2.1_colony_Run2Counts.csv and 2.1_colony_Run3Counts.csv.

1. Modify your script to load in the two new files. You may wish to use the looping code you wrote yesterday. Combine all three data sets into one **colony** data frame.
2. Now we have an additional variable, the **Run** each observation came from. Create a **Run** vector to record this, and add it to your colony data frame with **cbind**. You will need to use the **rep** function with its **each** argument – look it up to see what it does.
3. Modify your boxplot and analysis of variance to include **Run**. How does this data set look? Would you trust an analysis of it?

Loading multiple files

This should look familiar from yesterday:

```
# Load data
colony.files<-dir(pattern="Counts.csv")
colony<-data.frame()
for (cf in colony.files) {
  colony<-rbind(colony,read.csv(cf))
}
```

We *could* just call **read.csv** three times, but this would be very brittle. What if we were given fifty more files? What if the filenames changed? This code will handle these cases – we would only have to change the pattern in the first line, not every call to **read.csv**.

Creating the Run variable

If we inspect the data files, we can see there are 24 observations in each file. So we can hardcode a Run variable like this:

```
Run<-rep(1:3,each=24)  
colony<-cbind(Run,colony)
```

But this is brittle in the same way as calling **read.csv** three times is brittle. It would be better to get R to calculate **Run** from what it knows about the data.

We can count the observations in **colony** with **nrow**, and the number of runs (number of files) by getting the **length** of **colony.files**.

```
nruns<-length(colony.files)  
Run<-rep(1:nruns,each=nrow(colony)/nruns)  
colony<-cbind(Run,colony)
```

Analysis of variance with Run variable

Adding the Run variable to our analysis of variance is easy:

```
> colony.aov<-aov(Count~X*Z*Run,colony)
```

```
> print(summary(colony.aov))
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	
X	1	24939	24939	32.454	4.71e-07	***
Z	3	355524	118508	154.221	< 2e-16	***
Run	1	48197	48197	62.721	1.05e-10	***
X:Z	3	21967	7322	9.529	3.51e-05	***
X:Run	1	3485	3485	4.535	0.0376	*
Z:Run	3	49513	16504	21.478	2.19e-09	***
X:Z:Run	3	805	268	0.349	0.7897	
Residuals	56	43032	768			

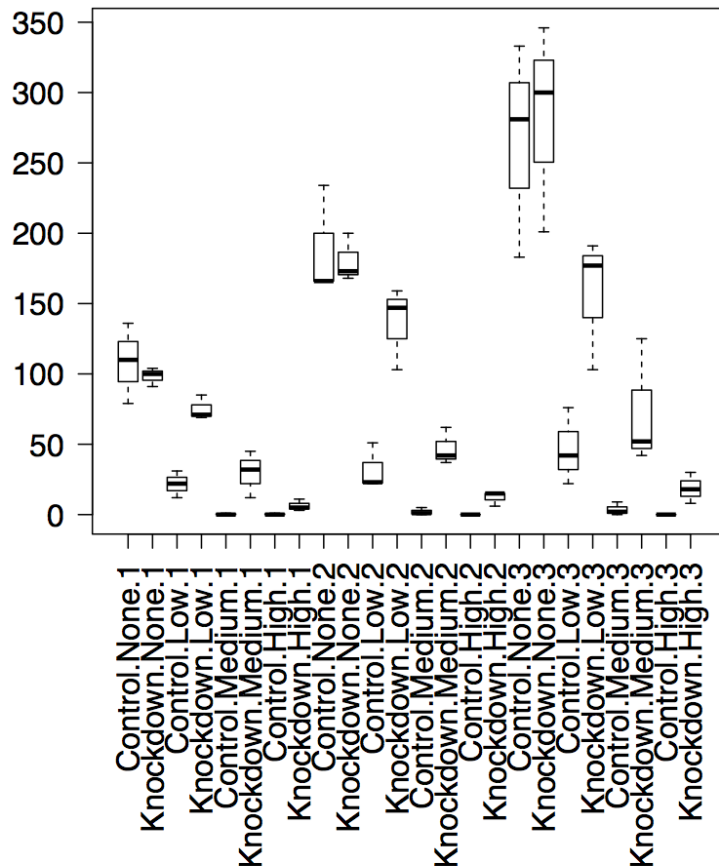
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

But the result is troubling – why should Run be correlated with colony growth? Let's have a look at the plot.

Plotting the Run variable

It is also easy to add Run to our boxplot:

```
boxplot(Count~X*Z*Run, colony, las=2)
```



Unfortunately, this shows that something has gone wrong during the experiments. The later runs have much higher colony growth than the first.

We probably can't rely on this data to tell us something about X and Z. At the very least, we will have to control for Run during the analysis.

Scripting summary

You can find the final version of the script in `2.1_colony_3.R`.

We have seen the value of writing a reusable script, and how we can modify scripts as new data comes in.

We have also seen how easy it is to incorporate new variables into R, and to repeat complicated analyses with single commands.

Finally, we have learnt a little about generalising our code to cope with variations in current and future input. Now we'll go on to learn how to define our own functions in R, to help us to generalise even further.

User functions

2

Introducing ...

User functions

- All R commands are function calls.
- Functions take some input, perform calculations on that input, and return some output.
- EG **sqrt** is a function that takes a value, calculates the square root of the value, and returns the square root.
- **aov** takes a formula referring to some data, calculates the analysis of variance for that data, and returns the model it calculated.
- We can define our own functions. User functions extend the capabilities of R by adapting or creating new tasks that are tailored to your specific requirements.
- User functions are objects, just like vectors and data frames. This has a few useful implications.

Defining a new function

- A function has a name, arguments, procedural steps, and a 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. We can call **return** explicitly:

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

- After defining the function, we can use it:

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

Named and default arguments

- We can generalise our function by adding a second argument.

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

- The power argument has a default value of 2; if we don't supply a power when we call the function, x will be squared.
- 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**



Calculation with user functions

User functions can be used wherever a built in function can be used:

```
a <- matrix(1:100, ncol=10, byrow=T) # make some dummy data
sqXplusX(a)
```

functions are R objects, just like a vector or a data frame, and exist in our workspace:

```
> sqXplusX
function(x) x^2+x
> ls()
```

Just like we can create anonymous vectors, we can create anonymous functions:

```
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
> function(x) x^2+x
function(x) x^2+x
```

But why would we want to do this?

Assigned or anonymous ...

User functions

Anonymous functions are really useful for `apply()` functions. We'll use the **colony** data frame again. Let's say we wanted to know the percentage of plates in each condition with no growth at all. Recall how we were using **tapply**:

```
tapply(colony$Count, list(colony$Z, colony$X, colony$Run), mean)
```

We can write a function to calculate our percentage directly into the call to **tapply**:

```
tapply(colony$Count, list(colony$Z, colony$X, colony$Run),  
       function(x) length(x[x==0])/length(x)*100)  
, , 1
```

	Control	Knockdown
None	0.00000	0
Low	0.00000	0
Medium	66.66667	0
High	66.66667	0
...		

Variable scope

Objects created in functions are not available to the global environment unless returned. They are limited to the *scope* of the function.

```
> addone<-function(x) {x<-x+1; x}  
> x<-1  
> addone(x)  
[1] 2  
> x  
[1] 1
```

The **x** in the global environment has nothing to do with the **x** declared in the function, and is unchanged by the call to the function. To update the global **x**, we would need to assign the return value of the function:

```
> x<-addone(x)
```

A function can only return one object, but that object can be a list, so if you have many objects to return, package them up into a list first.

Script / function tips

User functions

- If your script repeats the same command with different values more than twice, you should consider writing a function to generalise that command.
- Writing functions makes your code more easily understandable because they encapsulate a procedure into a well-defined boundary with consistent input/output
- Functions should only do one thing. If a function is doing multiple tasks, try to split it up into multiple functions. This rule of thumb means functions tend to be short, not more than around one or two screens of code.
- Look at other functions to get ideas for how to write your own ...
 - Display function code by entering the function's name without brackets.

Checking input and reporting errors

- A function should fail gracefully if it does not receive valid input when it is called. We can use **if** statements to check for appropriate input.
- R has two useful commands to tell the user something is wrong. **warning** prints a message and continues to run the function. **stop** ends the function after printing the message.
- For example, we might rewrite our **powXplusX** function to check that the power argument is a whole number:

```
powXplusX<-function(x,power=2) {  
  if (power %% 1 != 0) stop("Power should be a whole number")  
  x^power+x  
}
```

```
> powXplusX(10,3)
```

```
[1] 1010
```

```
> powXplusX(10,3.5)
```

```
Error in powXplusX(10, 3.5) : Power should be a whole number
```

Checking input and reporting errors

R has a very useful set of functions called the **is** family, which check the type of input values. For example:

```
sqXplusX <- function(x) {  
  if (is.numeric(x)) {  
    x^2 + x  
  } else {  
    stop("Input should be numeric")  
  }  
}
```

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

```
> sqXplusX("ten")
```

```
Error in sqXplusX("ten") : Input should be numeric
```

Here's another, more concise way to do the same thing:

```
sqXplusX <- function(x) {  
  if (!is.numeric(x)) stop ("Input should be numeric")  
  x^2 + x  
}
```



is.array	is.language	is.primitive
is.atomic	is.leaf	is.qr
is.BaseNamespace	is.list	is.R
is.call	is.loaded	is.raw
is.character	is.logical	is.real
isClass	is.matrix	is.recursive
isClassDef	is.mts	is.relistable
isClassUnion	is.na	is.Restart
is.complex	is.na<-	isS4
is.data.frame	is.na.data.frame	isSealedClass
isdebugged	is.na<-.default	isSealedMethod
is.double	is.na<-.factor	isSeekable
is.element	is.name	is.single
is.empty.model	is.Namespace	is.stepfun
is.environment	is.nan	is.symbol
is.expression	is.na.POSIXlt	isSymmetric
is.factor	is.null	isSymmetric.matrix
is.finite	is.numeric	is.table
is.function	is.numeric.Date	isTRUE
isGeneric	is.numeric.POSIXt	is.ts
isGrammarSymbol	is.numeric_version	is.tskernel
isGroup	is.object	is.unsorted
isIncomplete	isOpen	is.vector
is.infinite	is.ordered	isVirtualClass
is.integer	isoreg	isXS3Class
is.limits	is.package_version	
	is.pairlist	

The **is.family**

Checking input and reporting errors

Here's another, more concise way to do the same thing:

```
sqXplusX <- function(x) {  
  if (!is.numeric(x)) stop ("Input should be numeric")  
  x^2 + x  
}
```

This is not only shorter, but it also gets all the error checking out of the way before the main processing steps.

You may also find the **%in%** command useful, which checks to see if the elements of one vector are present in another:

```
> levels(colony$Z)  
[1] "None"    "Low"      "Medium"   "High"  
> "Low" %in% colony$Z  
[1] TRUE  
> "Zero" %in% colony$Z  
[1] FALSE  
> c("None", "Low") %in% colony$Z  
[1] TRUE TRUE
```

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 should take two named arguments:

temperature (**t**) is numeric

units (**unit**) is character

Both arguments should have appropriate default values.

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 Fahrenheit if given in Centigrade, and vice versa.

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:
 - write a skeleton function definition (eg just a name and brackets)
 - add appropriate argument names and defaults
 - write code to convert Centigrade into Fahrenheit and check it works
 - write code to convert Fahrenheit to Centigrade and check it works
 - add error checking code, including the checks from the previous slide, and any others you can think of
 - write a set of test calls to confirm that your function handles correct *and incorrect* input
- If you get stuck, call us to help you!

Temperature conversion exercise 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 temperature entered")
```

```
  if ( !(unit %in% c("c","f"))){  
    stop("Unrecognized temperature unit. Enter (c)entigrade or (f)ahrenheit.")  
  }
```

```
  converted<-0
```

```
  # Conversion for centigrade
```

```
  if ( unit=="c" ) {  
    converted <- 9/5 * t + 32  
  }
```

```
  # Conversion for Fahrenheit
```

```
  if(unit=="f"){  
    converted <- 5/9 * (t-32)  
  }
```

```
  converted
```

```
}
```

```
> convTemp(t=-273,unit="c")  
[1] -459.4
```

Example code:
2.2_convtemp.R

Temperature conversion application

An American colleague is moving to Cambridge and wants to know what the average minimum and maximum temperatures are for each season, in Fahrenheit.

The file **2.2_cambridge_temps.tsv** contains minimum and maximum temperatures per month.

1. Load this file into a data frame in R with **read.delim**.
2. Add a season variable to the data frame (define the seasons however you like).
3. Use **tapply**, **mean** and **convTemp** to calculate average minimum and maximum temperatures for each season.

Temperature conversion application

Here's one way of doing this:

```
camtemps<-read.delim("2.2_cambridge_temps.tsv")

camtemps<-cbind(camtemps, Season=c("Winter", "Winter", "Spring", "Spring",
"Spring", "Summer", "Summer", "Summer", "Autumn", "Autumn", "Autumn",
"Winter"))

tapply(camtemps$MinimumTemp, camtemps$Season, function(x) convTemp(mean(x)))
tapply(camtemps$MaximumTemp ,camtemps$Season, function(x) convTemp(mean(x)))
```

But because R functions are so flexible, we can do the conversion at any point in the calculation. These alternatives will also work:

```
> tapply(camtemps$MinimumTemp, camtemps$Season,
        function(x) convTemp(mean(x)))

> convTemp(tapply(camtemps$MinimumTemp, camtemps$Season, mean))

> tapply(convTemp(camtemps$MinimumTemp), camtemps$Season, mean)
```

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
rux	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	-3.0595

+

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	2.2802
rux	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
rux	-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
rux	-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
rux	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 an empty collection of genes
```

```
genes <- c()
```

```
for( fileNum in 1:5 ){
```

```
  # load in files 2.3_DiffGenes1.tsv ...
```

```
  t <- read.delim(paste("2.3_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:
2.3_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("2.3_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

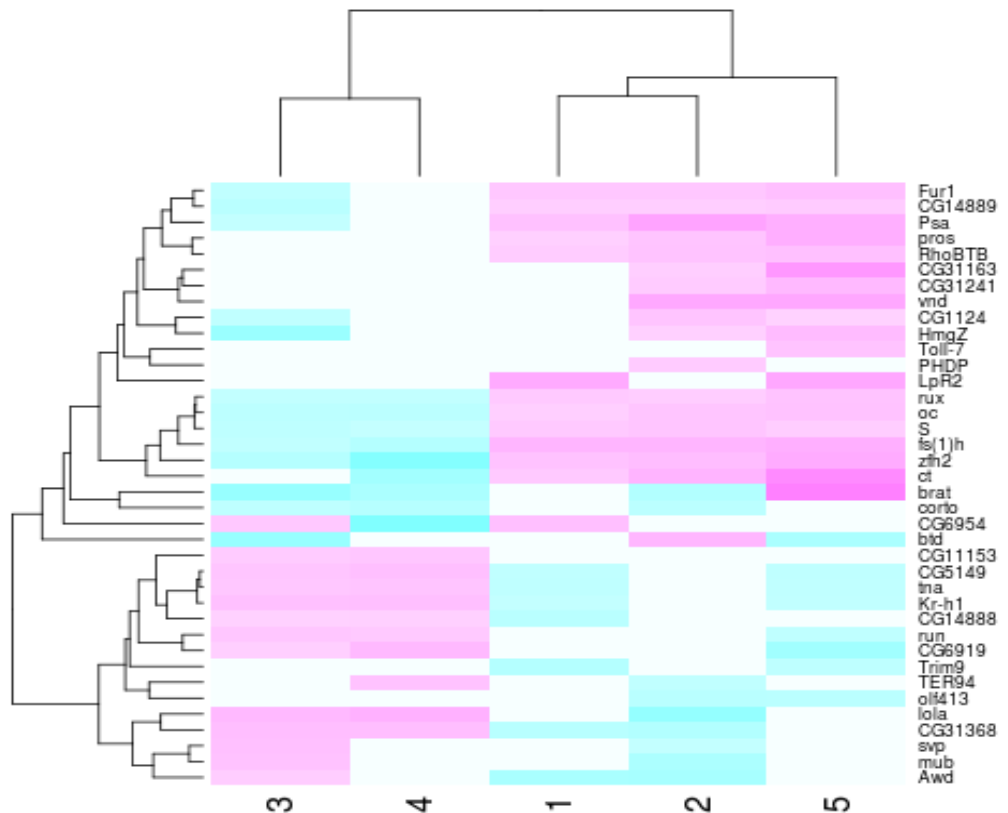
Gene clustering

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("2.3_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 result is a list of split values for each element of the 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 the difference between **rownames(values) %in% experim.genes** and **experim.genes %in% rownames(values)**

Example code:
2.3_geneClustering.R

Graphics

4

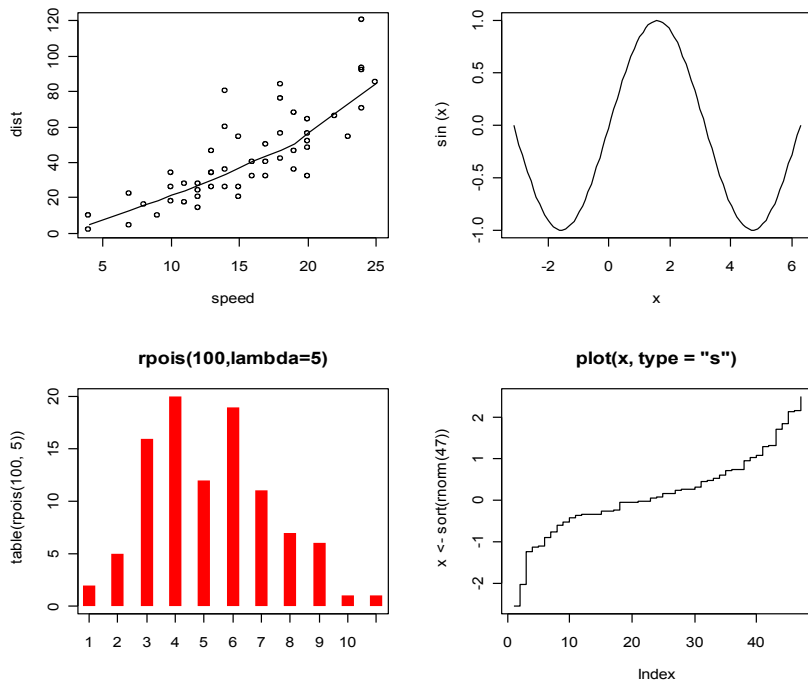
Starting out with R graphics

Graphics

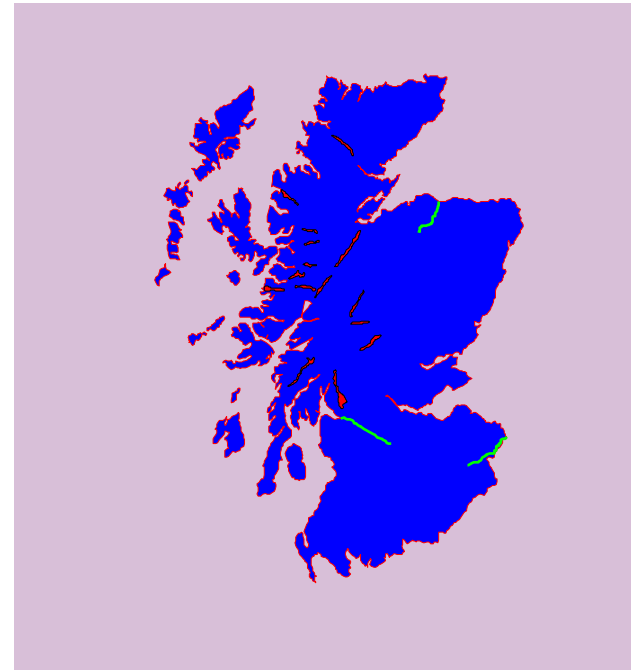
- R provides several mechanisms for producing graphical output
 - Functionality depends on the level at which the user seeks interaction with R
 - graphics systems, packages, devices & engines
- High level graphics
 - Functions compute an appropriate chart based up on the information provided. Optional 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) ↱

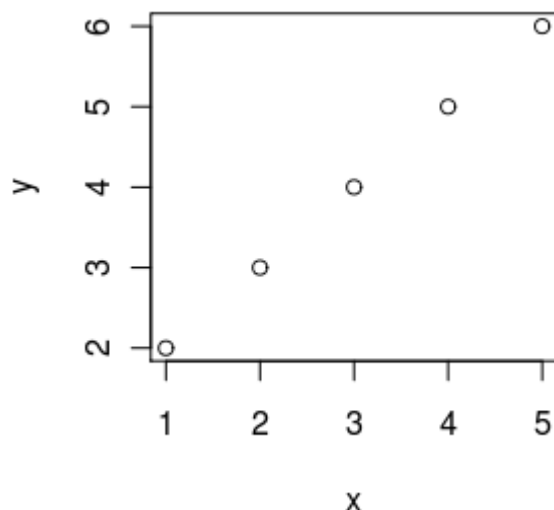


Low level plotting
(Scotland by blighty package) ↲

Essential plotting - plot()

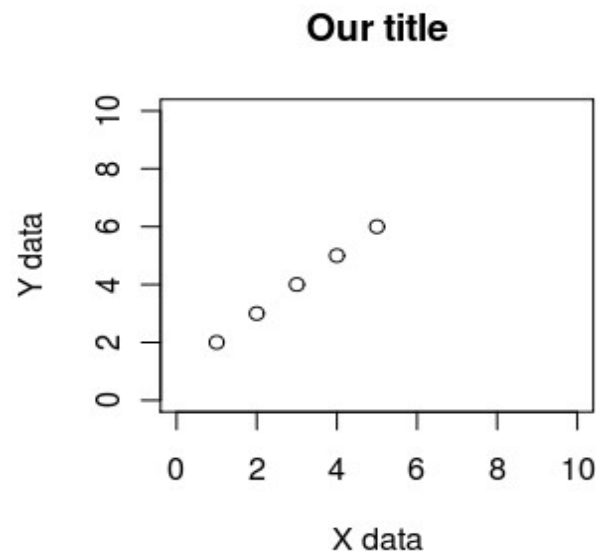
- plot() is the main function for plotting, it takes x,y values to plot and also lots of graphical parameters (see **?par** for all of them)

default plotting



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

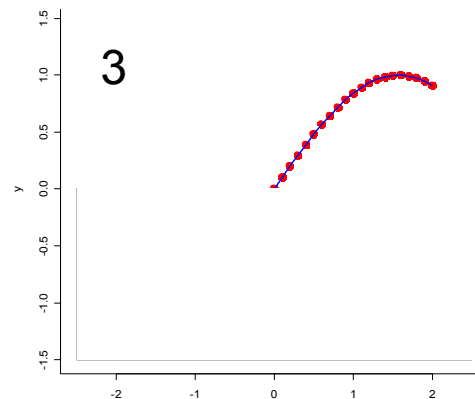
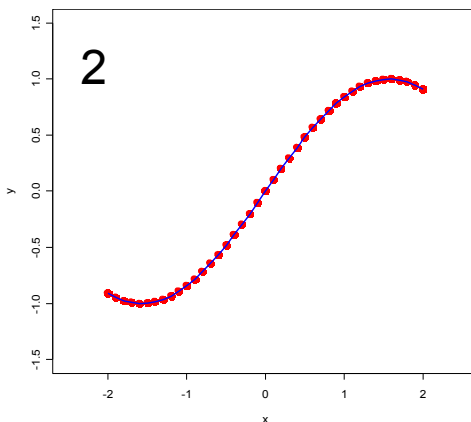
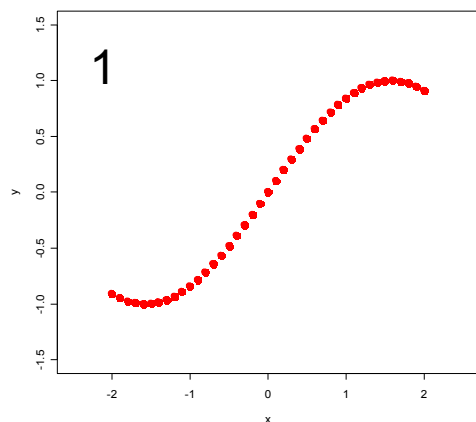
custom plotting



```
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")
```

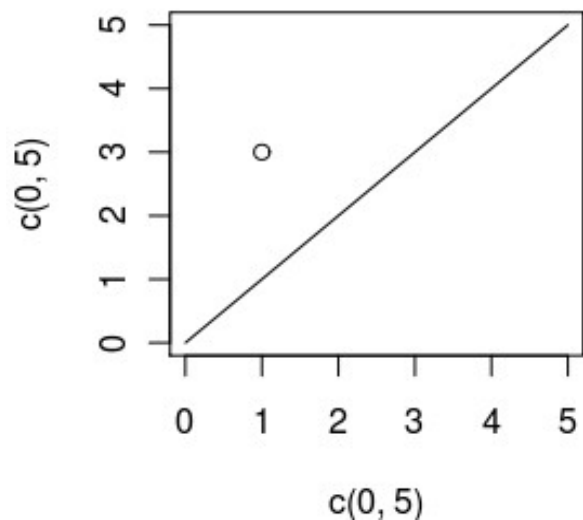
xlim, ylim = axis limits
col = line colour
pch = plotting character [**example (points)**]
cex = character expansion [scaling factor]

lty = line type
lwd = line width
rect = rectangle

Example code:
14_painterModel.R

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

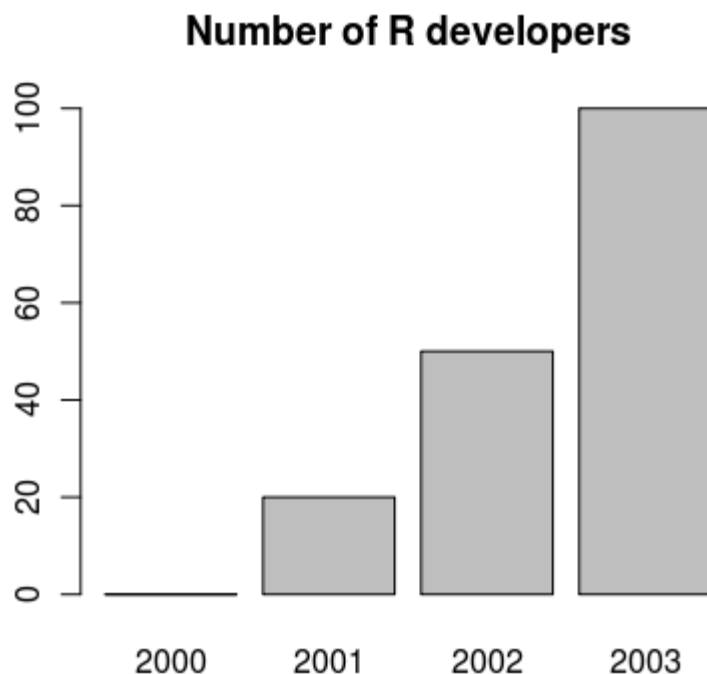
- **plot()** is used to start a new plot, accepts x,y data, but also data from some objects (like linear regression). Use the parameter **type** to draw points, lines, etc (see **?plot**)
- **points()** is used to add points to an existing plot
- **lines()** is used to add lines to an existing plot



```
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()

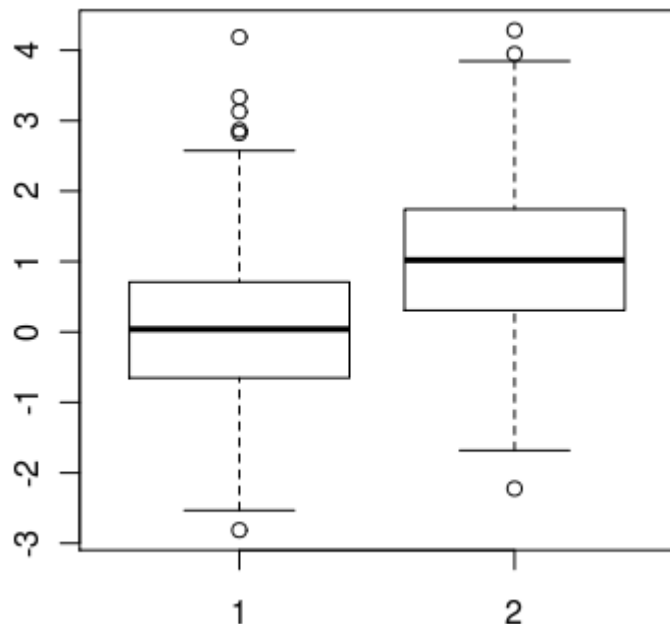
- visualizing a vector of data can be done with bar plots, using function **barplot()**



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

Making box plots - boxplot()

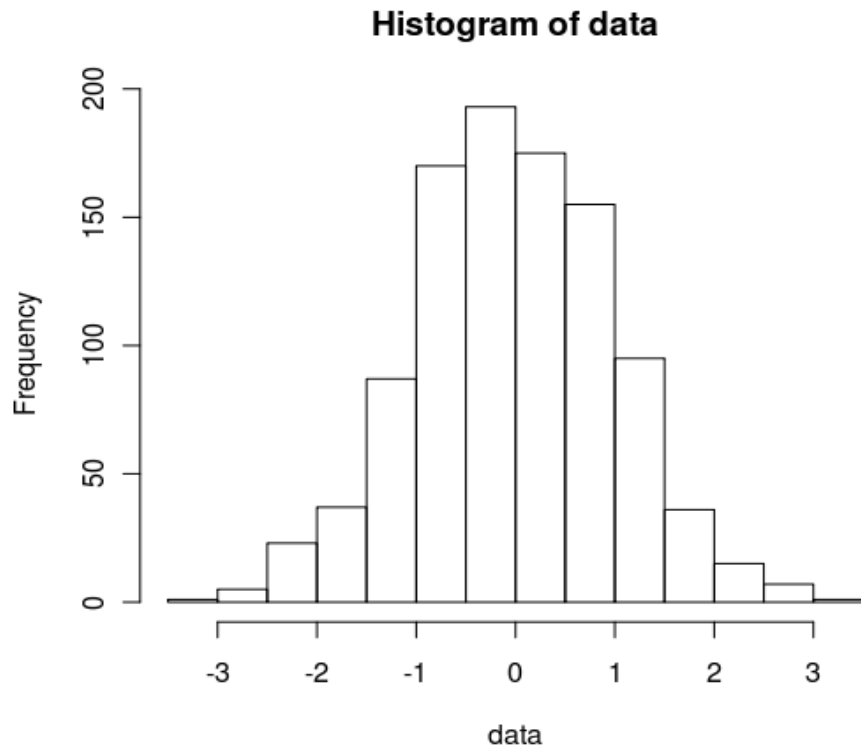
- when a spread of data needs to be visualised, we can use boxplots with function **boxplot()**



```
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()

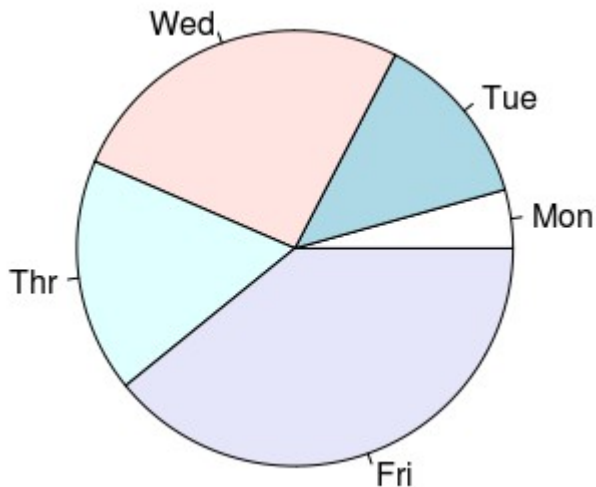


```
data <- rnorm(1000)
```

```
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, ...)`

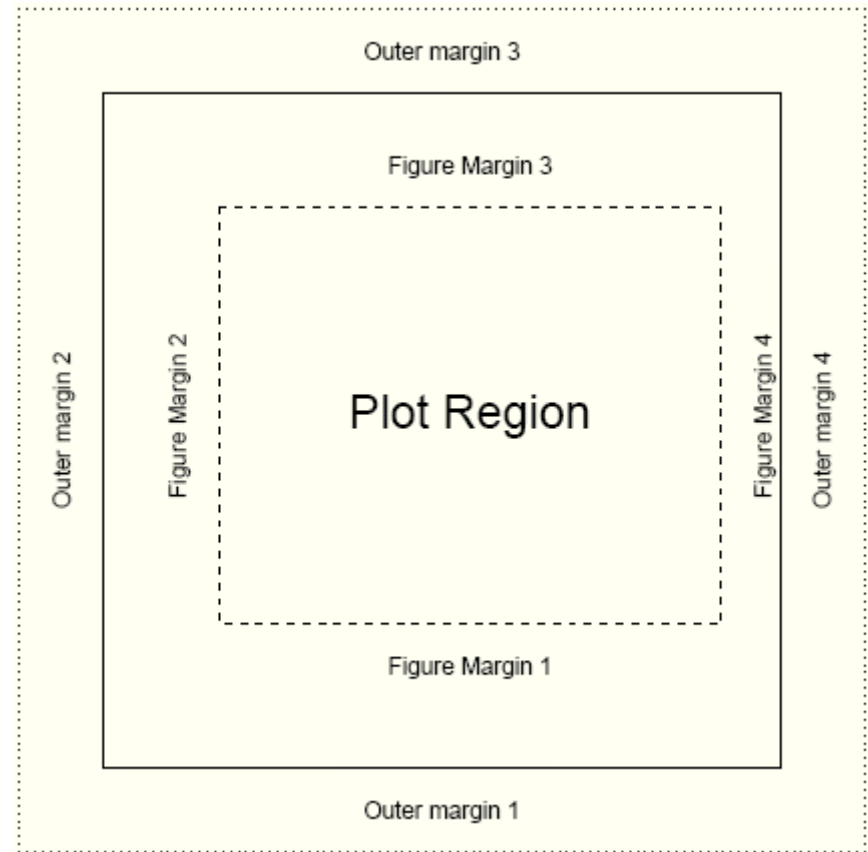
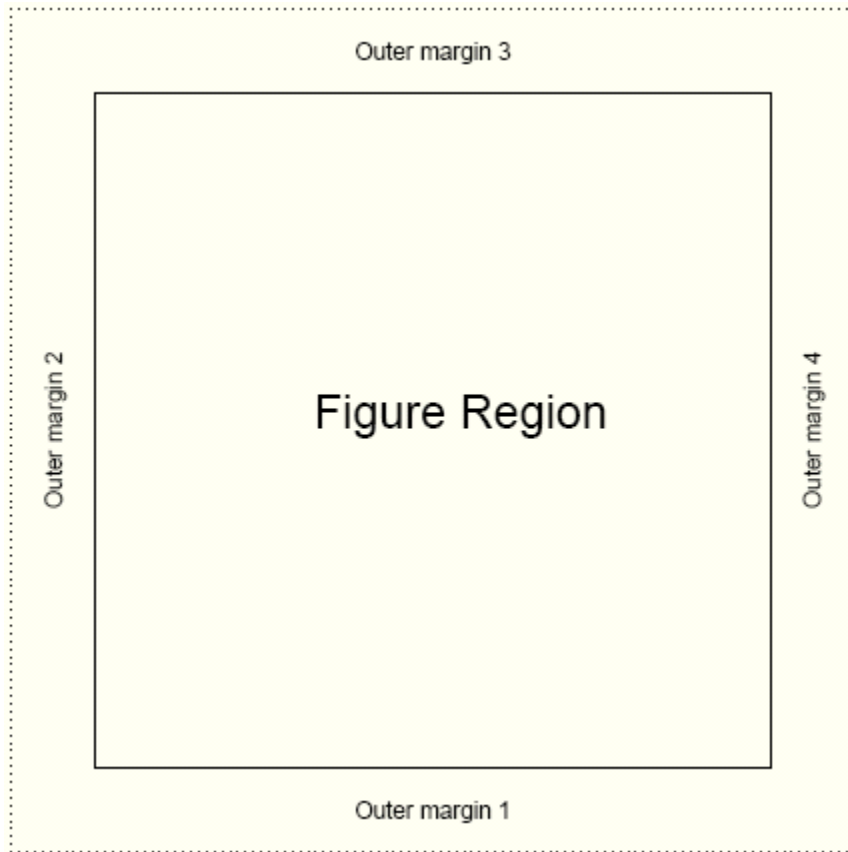
Setting graphics layout and style - par()

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) ) ↑
```

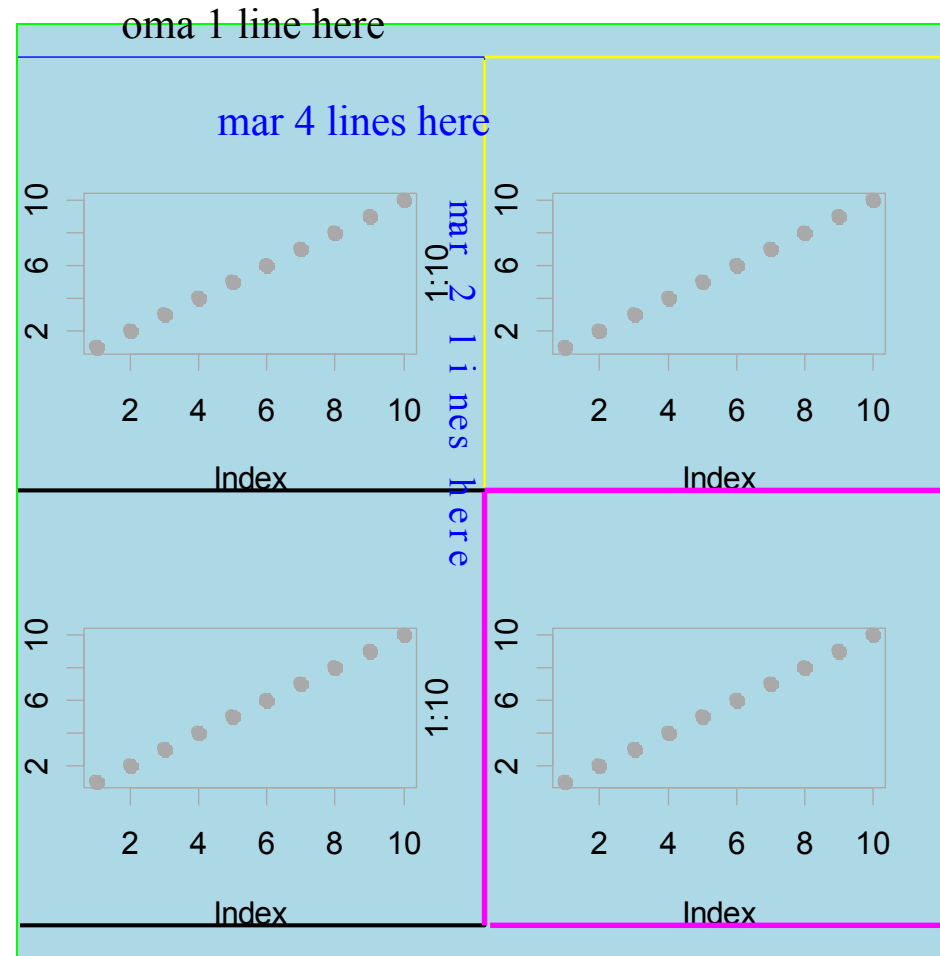
Sets space for x & y labels, a main title, and a thin margin on the right

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

Time for

COFFEE, RESUME 3:30 PM

Plotting characters for plot() size and orientation

pch= ...

Sets one of the 26 standard plotting character used.

Can also use characters, such as "."

cex= ...

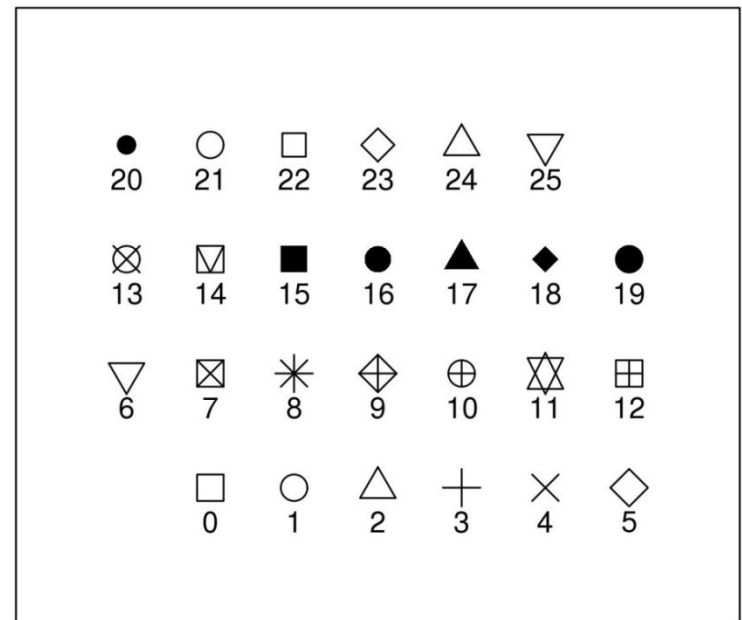
Character expansion. Sets the scaling factor of the printing character

las= ...

Axes label style. 1 normal, 2 rotated 90°

4 styles (0-3)

26 standard plotting characters



Plotting characters exercise

Graphics

16_plottingChars.R

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

X-Y coordinates,
Plotting character index counter

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

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

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

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

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

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

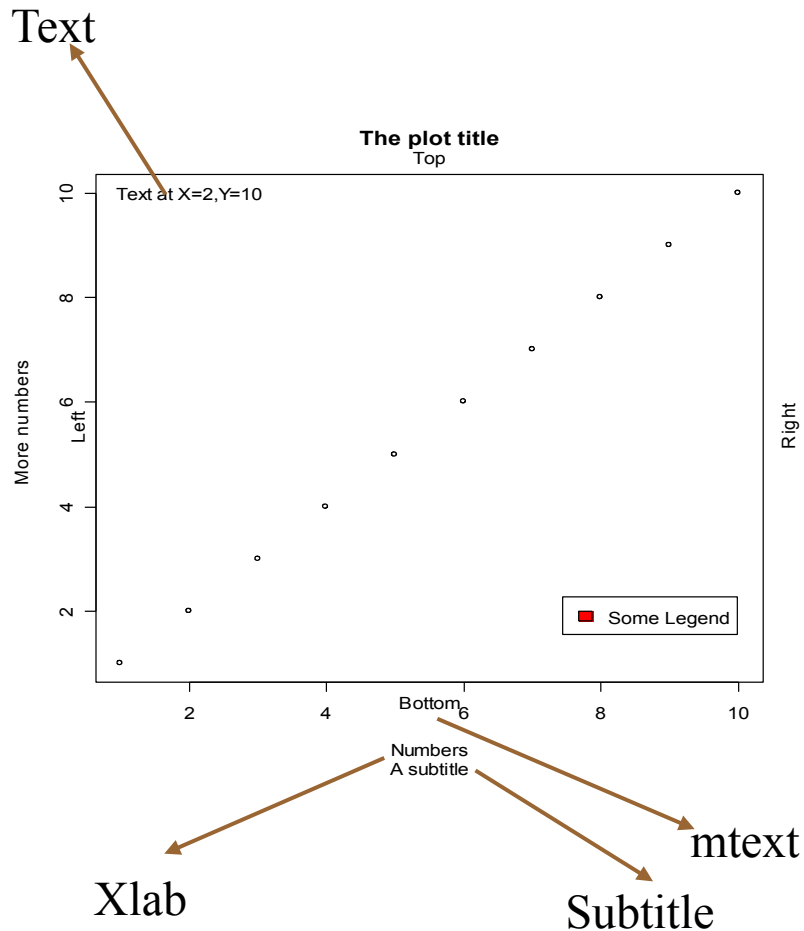
```
legend(x,y, legend=...) ↑
```

- 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



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) ↑
```

Adding legend ...

Don't forget to mouse click!

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

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

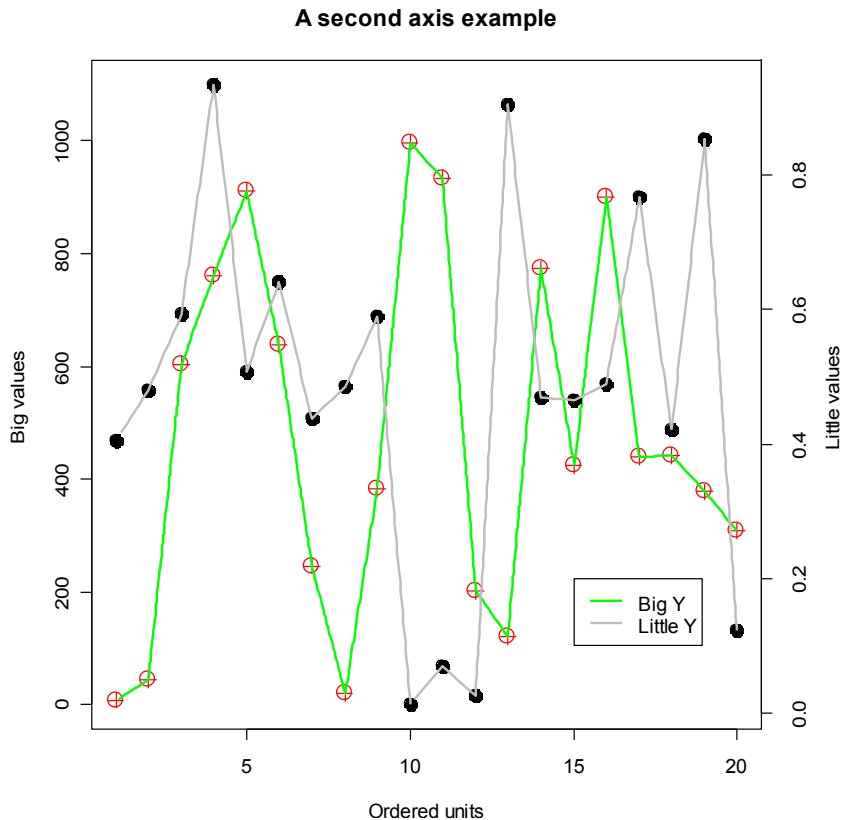
`plot(..., axes=F)` ↴

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 hexadecimal
 - 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)`

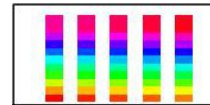
`heat.colors(n)`

`terrain.colors(n)`

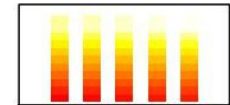
`topo.colors(n)`

`cm.colors(n)`

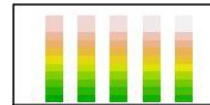
Rainbow Colours



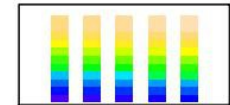
Heat Colours



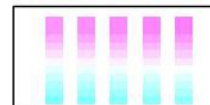
Terrain Colours



Topological Colours



Cyan Magenta Colours



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

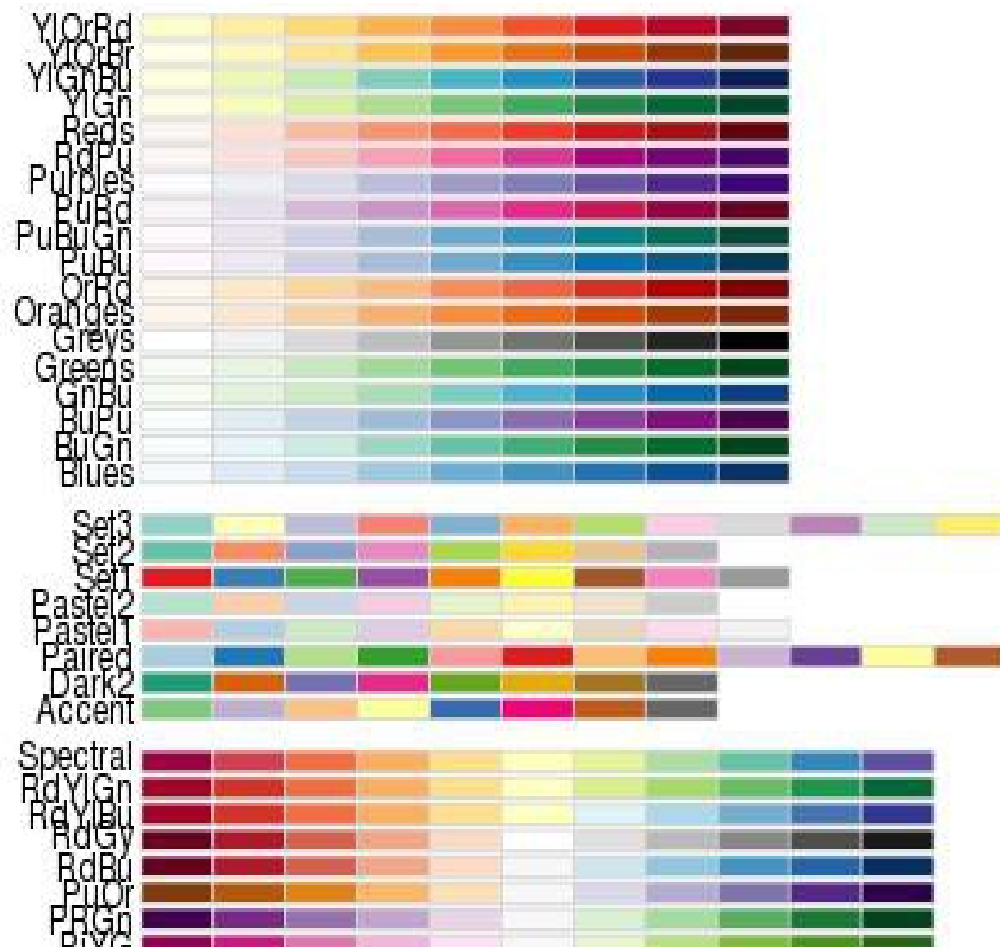
```
display.brewer.all(n=NULL, type="all", select=NULL, exact.n=TRUE)
```

- 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: jpeg, A4 @ 300 dpi, portrait, size in pixels
 - `jpeg(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

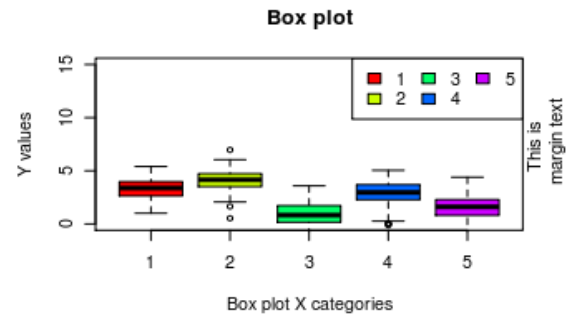
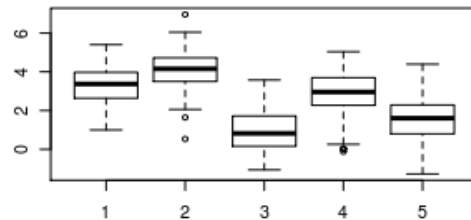
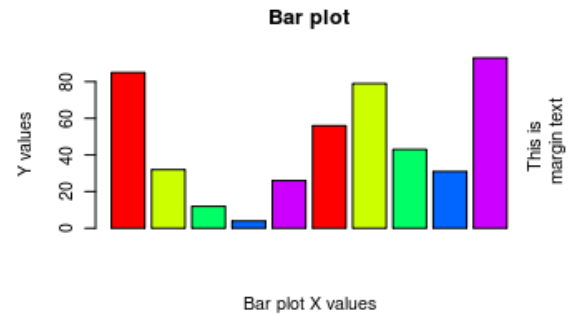
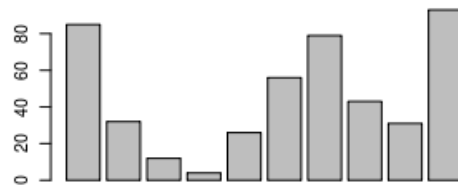
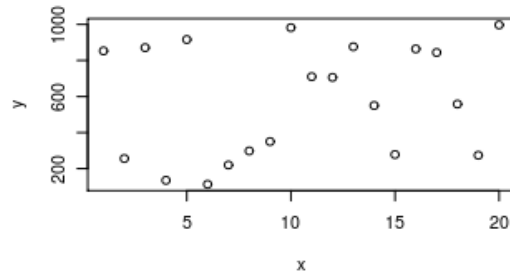
---

- 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

# 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>

---

Thanks for your attention!

**END OF COURSE**