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

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

Course material:

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

Slides by Ian Roberts and Robert Stojnić

Day 1 review

1. Understanding the R environment
   - Installing, option setting & customization
2. Creating and manipulating R objects
3. Essential built-in R functions for working with vectors and matrices
   - Reading in and writing data frames
4. Basic statistical analysis
   - Two sample tests & linear regression

Day 2 Schedule

1. Writing scripts
    - Building scripts for routine analysis
    - Script automation using R batch mode

Morning coffee

2. Extending R with customized functions
    - Learn how to write your own R functions

Lunch

3. Advanced data analysis and basic R graphics
    - Integrate data from different sources and analyze

Afternoon coffee

4. More on R graphics
    - Produce publication quality charts

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

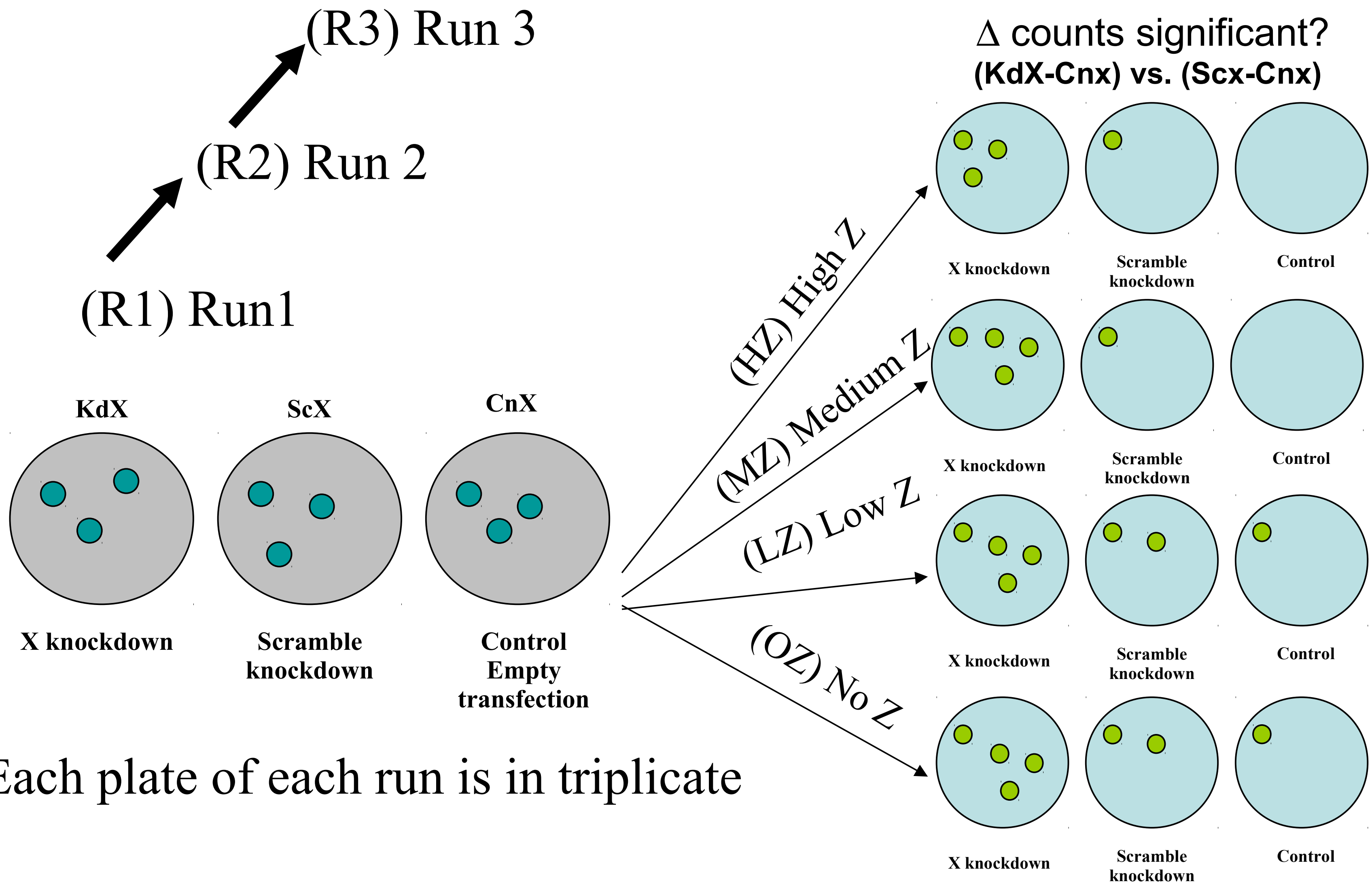
- Colony forming assays provide a measure of cellular proliferation. They are used as read outs for various biological systems
  - A well controlled study may involve multiple samples, treatments and controls (probably replicated).
  - This produces a lot of ‘count’ data, ideally suited to routine script 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			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	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	100
R1	KDX	OZ	2	104
R1	KDX	OZ	3	91
R1	KDX	LZ	1	85
R1	KDX	LZ	2	71
R1	KDX	LZ	3	69
R1	KDX	MZ	1	32
R1	KDX	MZ	2	12
R1	KDX	MZ	3	45
R1	KDX	HZ	1	5
R1	KDX	HZ	2	11
R1	KDX	HZ	3	3
R1	SCX	OZ	1	136
R1	SCX	OZ	2	99
R1	SCX	OZ	3	154
R1	SCX	LZ	1	79
R1	SCX	LZ	2	45
R1	SCX	LZ	3	35
R1	SCX	MZ	1	3
R1	SCX	MZ	2	7
R1	SCX	MZ	3	9
R1	SCX	HZ	1	1
R1	SCX	HZ	2	0
R1	SCX	HZ	3	3
R1	CNX	OZ	1	110
R1	CNX	OZ	2	136
R1	CNX	OZ	3	79
R1	CNX	LZ	1	12
R1	CNX	LZ	2	31
R1	CNX	LZ	3	22
R1	CNX	MZ	1	0
R1	CNX	MZ	2	1
R1	CNX	MZ	3	0
R1	CNX	HZ	1	0
R1	CNX	HZ	2	0
R1	CNX	HZ	3	1

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)
```

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

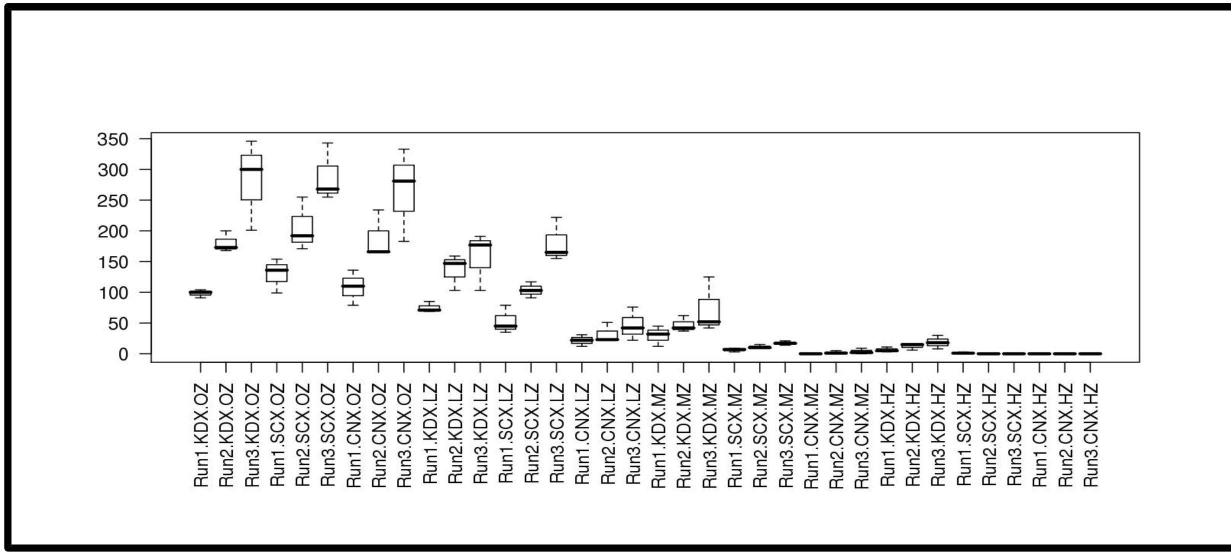
- we need a reshaped, background corrected, table of results on which to perform our tests
- for clarity where possible use dollar ($) notation (work only with data frames)

```
### 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 0Z 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")
```

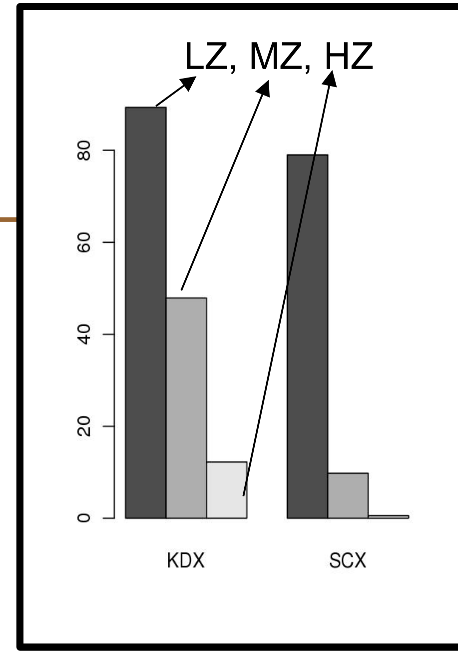
-ve subscripts mean 'delete'

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

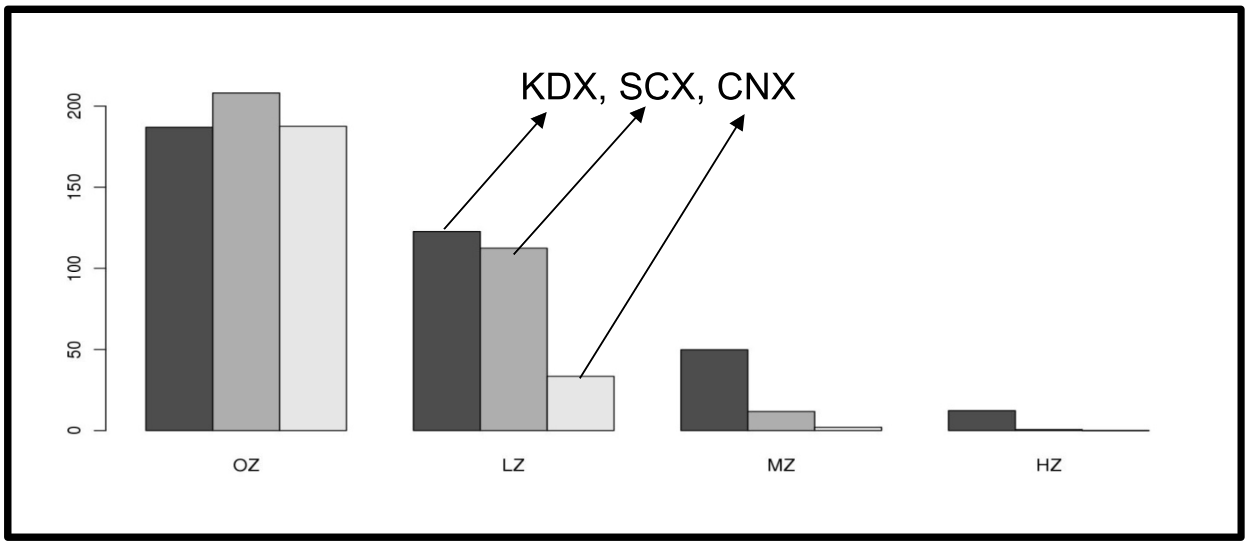
We find that the difference in means is not significant (would expect observation to occur 1:4 times), and that the scramble and knockdown counts have a 90% correlation



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
- Easy in this case, but what if there were many more than three files?
- Notice that each file contains the terms 'Counts.csv'

```
# 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)
}
```

Time for ...

MORNING COFFEE

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**

```
.array              is.language            is.primitive
s.atomic            is.leaf                is.qr
isBaseNamespace     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                  isRestart
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      isNamespace            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
s.infinite          is.ordered             isVirtualClass
.integer            isoreg                 isXS3Class
nds                 is.package_version
                    is.pairlist
```

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

Time for

LUNCH (RESUME 1:30 PM)

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 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

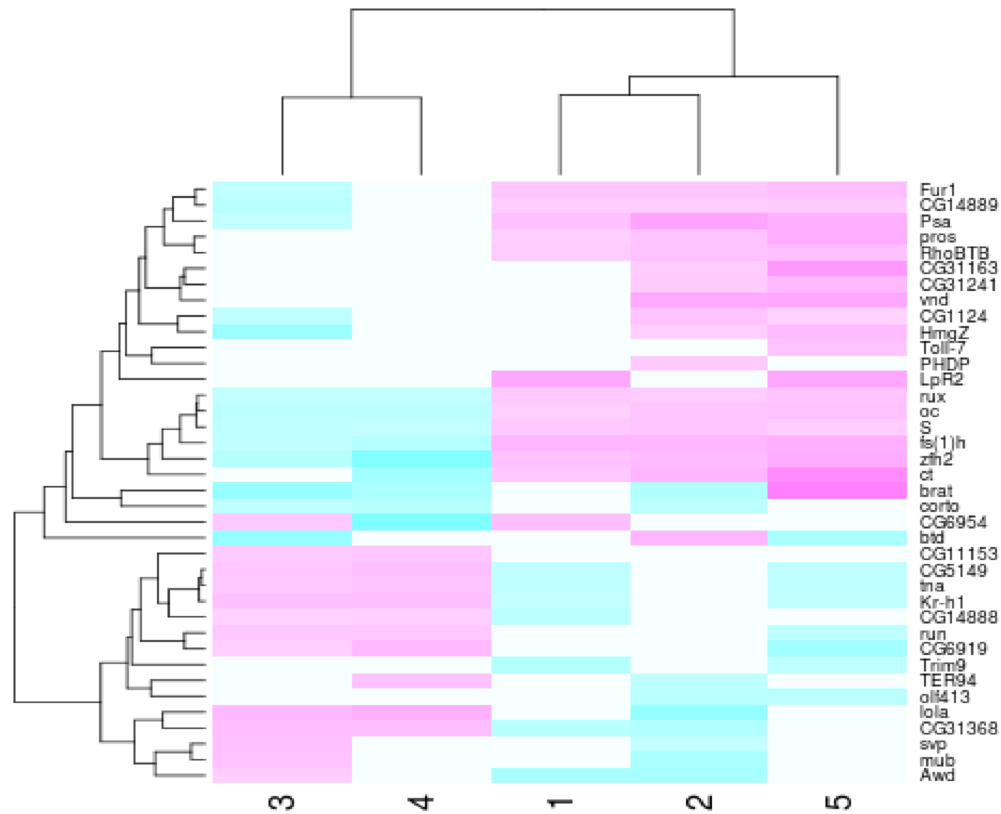
Gene clustering

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

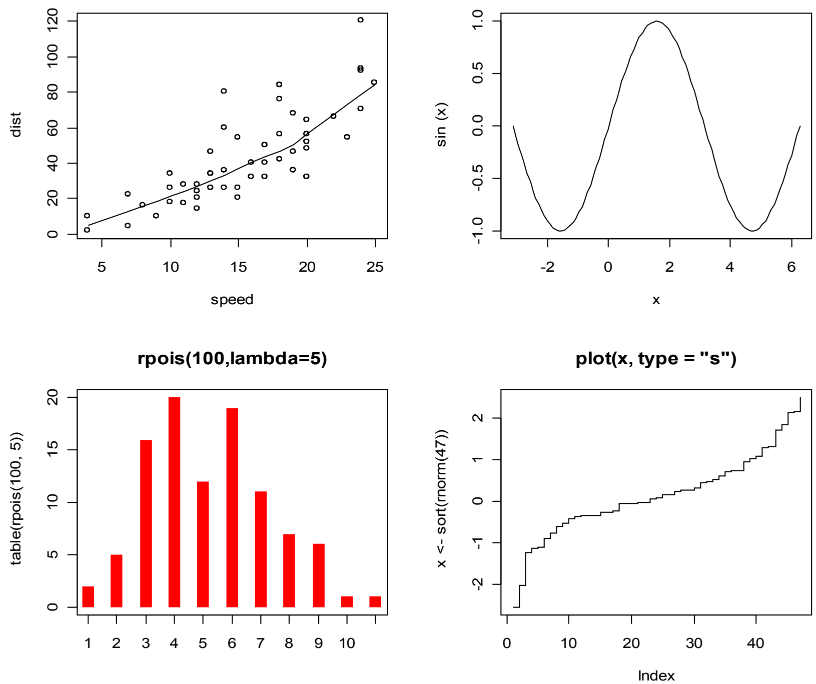
Graphics

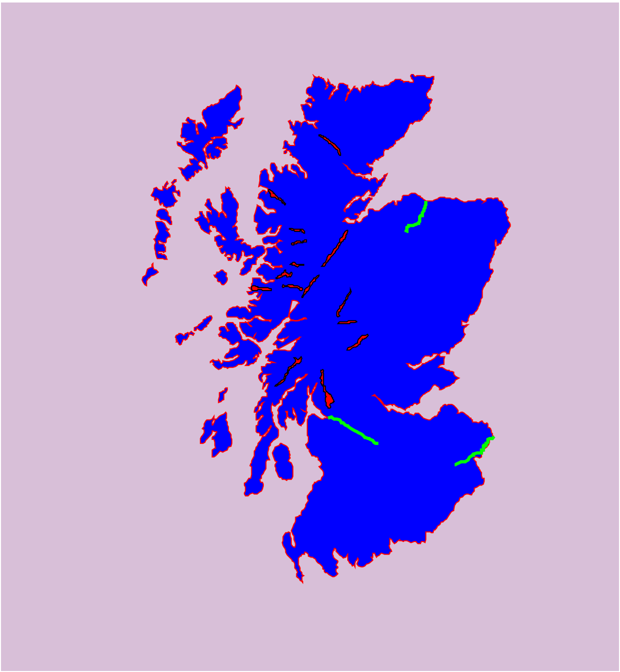
4

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





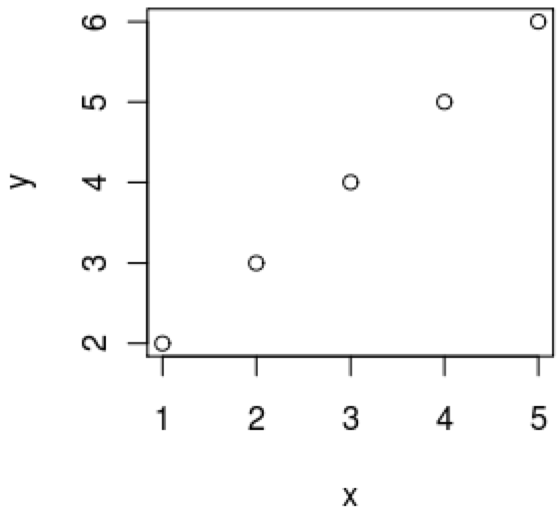
High level plotting
example(plot)

Low level plotting
(Scotland by blighty package)

Essential plotting - plot()

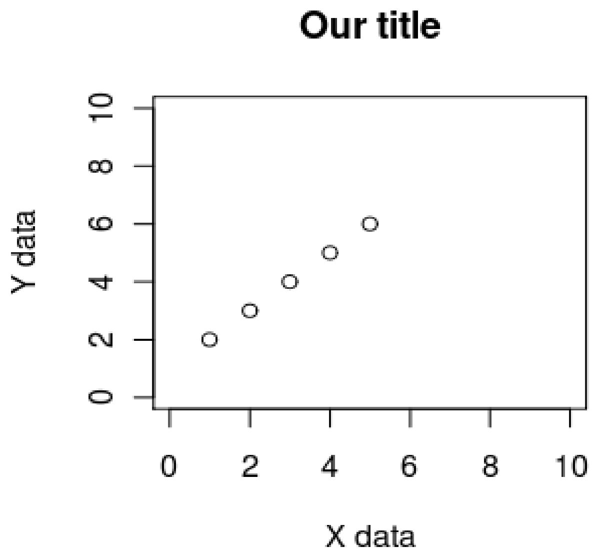
- plot() is the main function for plotting, it takes x,y values to plot and also lots of graphical parameters (see **?par** for all of them)

default plotting




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

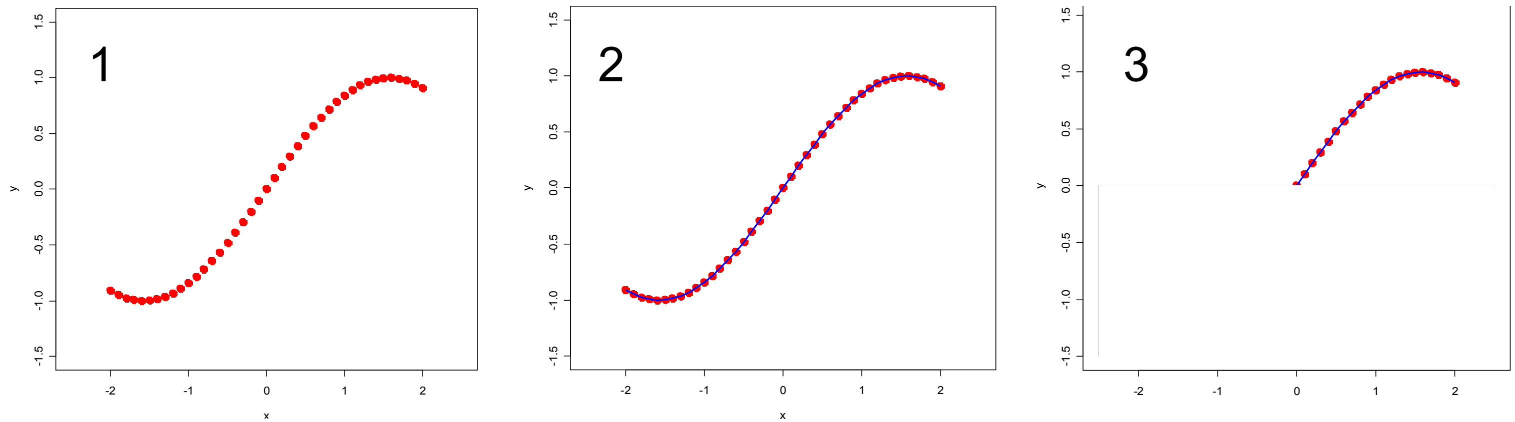
custom plotting




```
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")
```

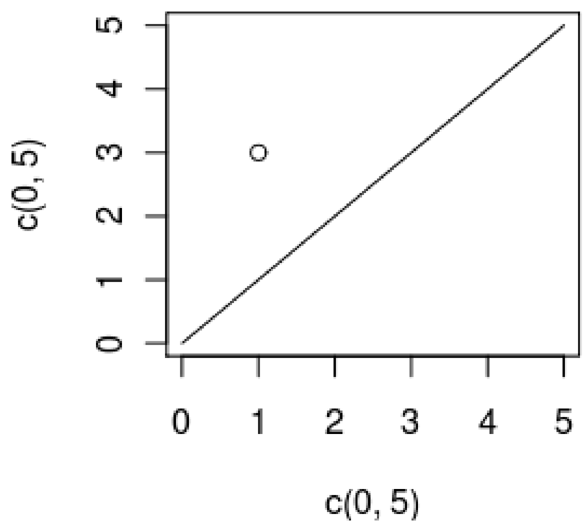
xlim, ylim = axis limits
col = line colour
pch = plotting character [`example(points)`]
cex = character expansion [scaling factor]

lty = line type
lwd = line width
rect = rectangle

Example code:
14_painterModel.R

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

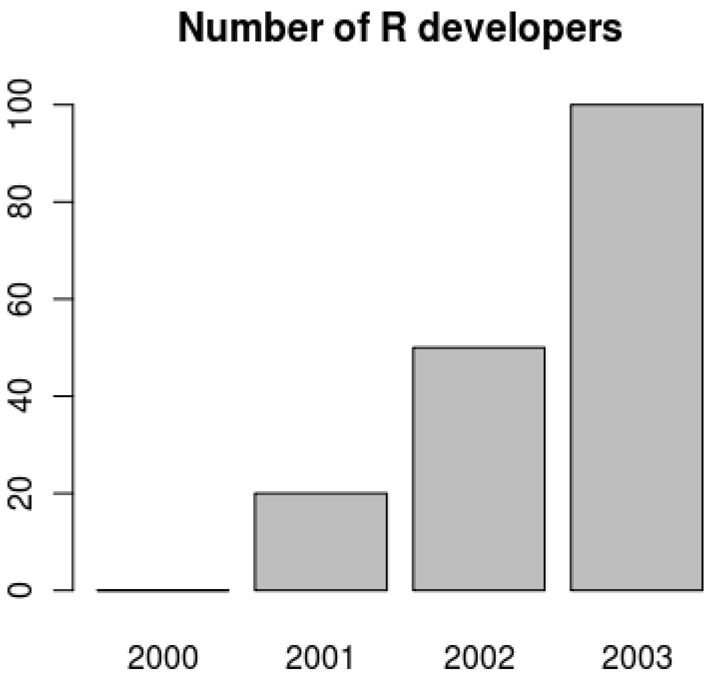
- **plot()** is used to start a new plot, accepts x,y data, but also data from some objects (like linear regression). Use the parameter **type** to draw points, lines, etc (see **?plot**)
- **points()** is used to add points to an existing plot
- **lines()** is used to add lines to an existing plot




```
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()

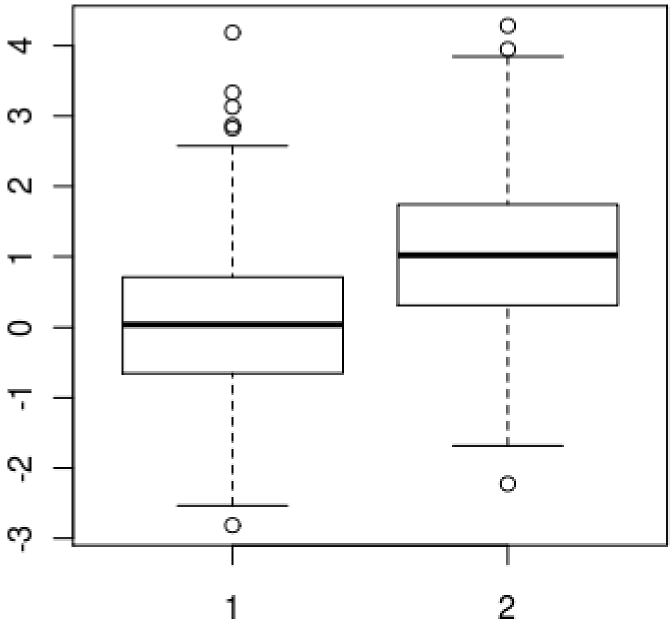
- visualizing a vector of data can be done with bar plots, using function **barplot()**




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

Making box plots - boxplot()

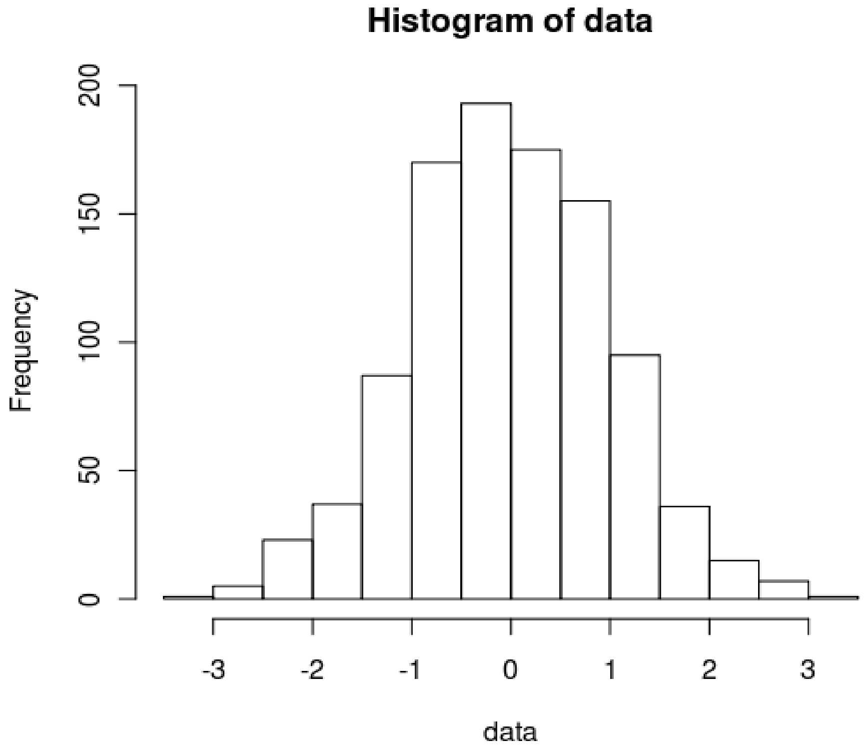
• when a spread of data needs to be visualised, we can use boxplots with function **boxplot()**




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

Making histograms - hist()

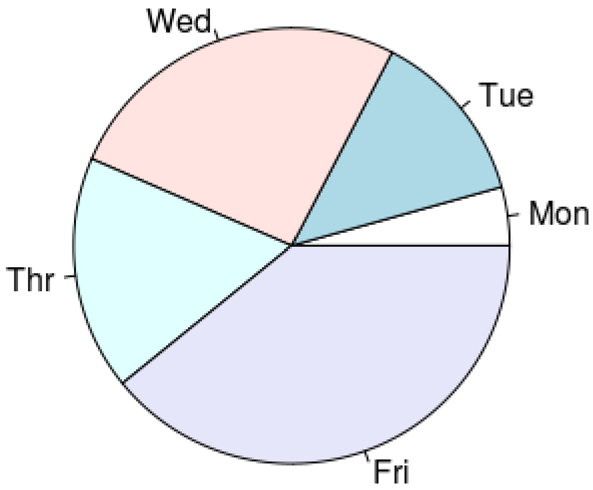
• when we need to look at the distribution of data, we can visualize it using histograms with function hist()




```
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, …)`

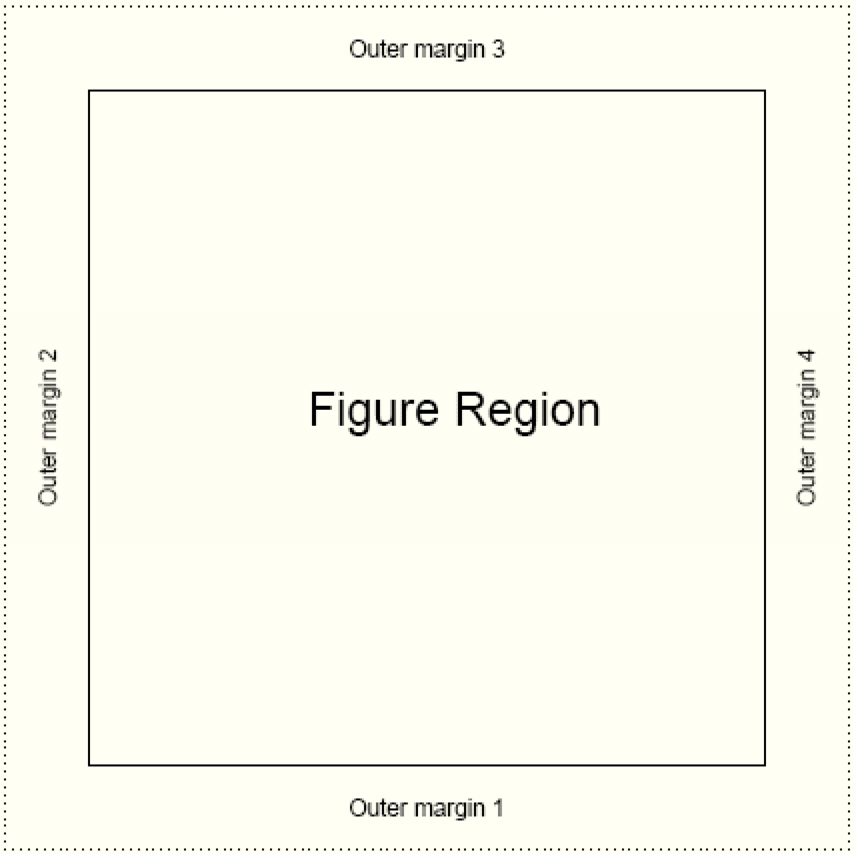
Setting graphics layout and style - par()

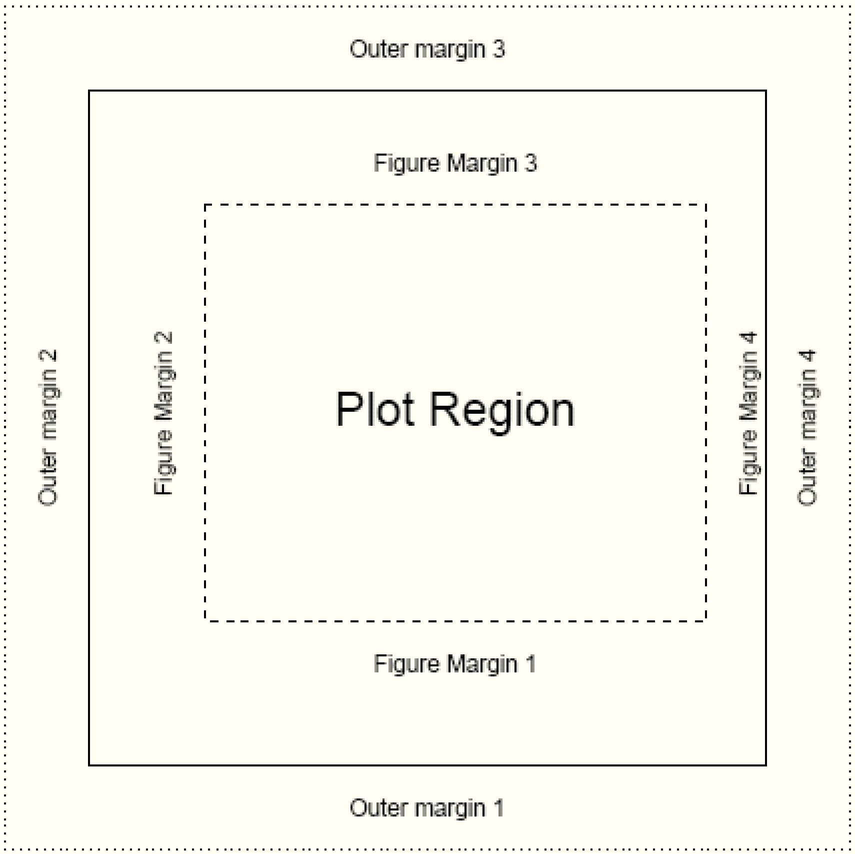
`par()` Top level graphics function

- 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

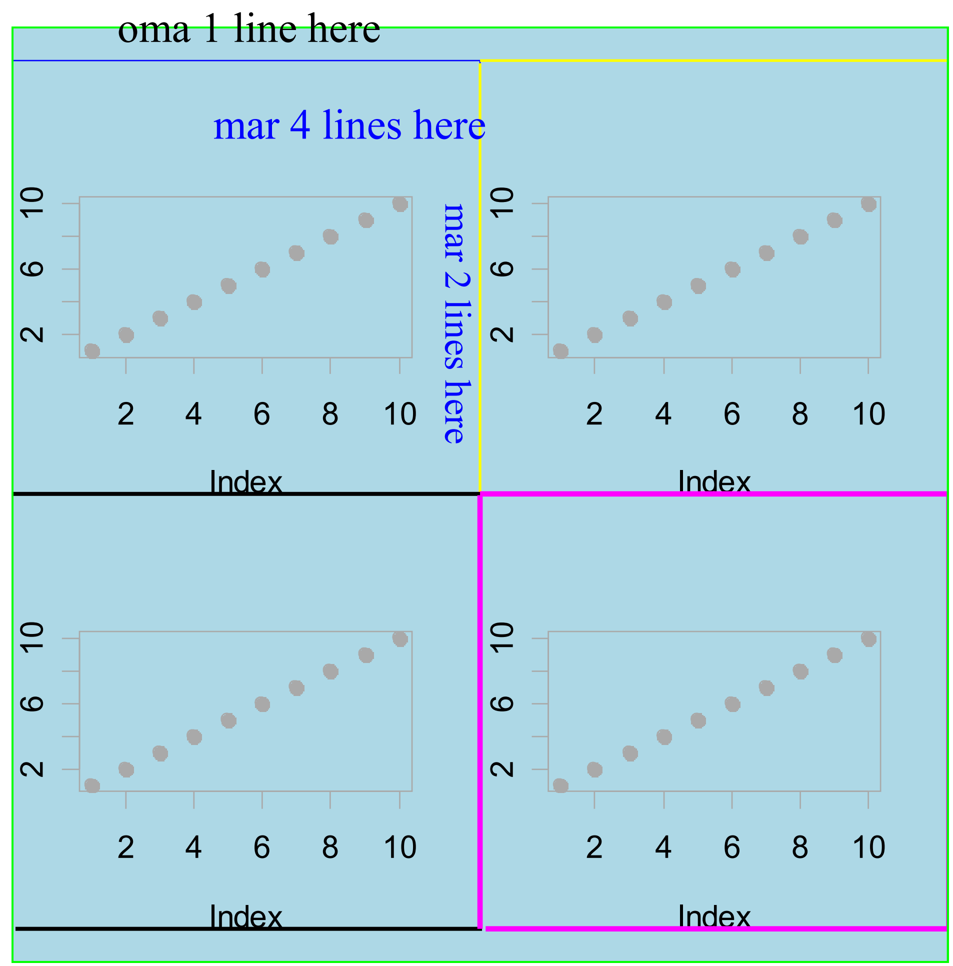
```
par(mar=c(4,2,4,2))
```
- 4 lines at bottom & top
- 2 lines left & right

```
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

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= ...

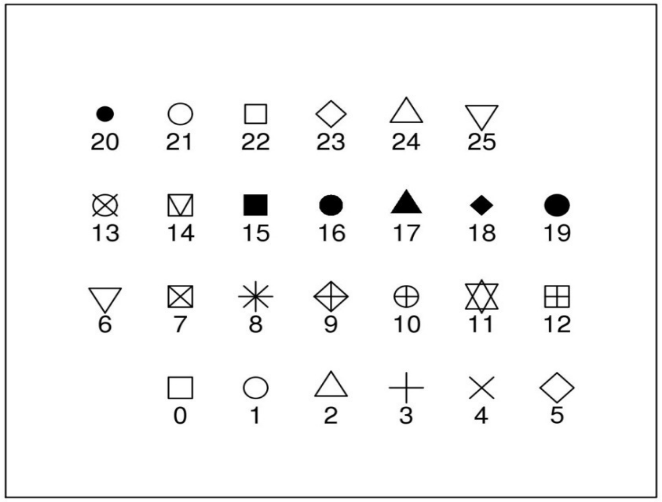
Character expansion. Sets the scaling factor of the printing character

las= ...

Axes label style. 1 normal, 2 rotated 90º

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
  }

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

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

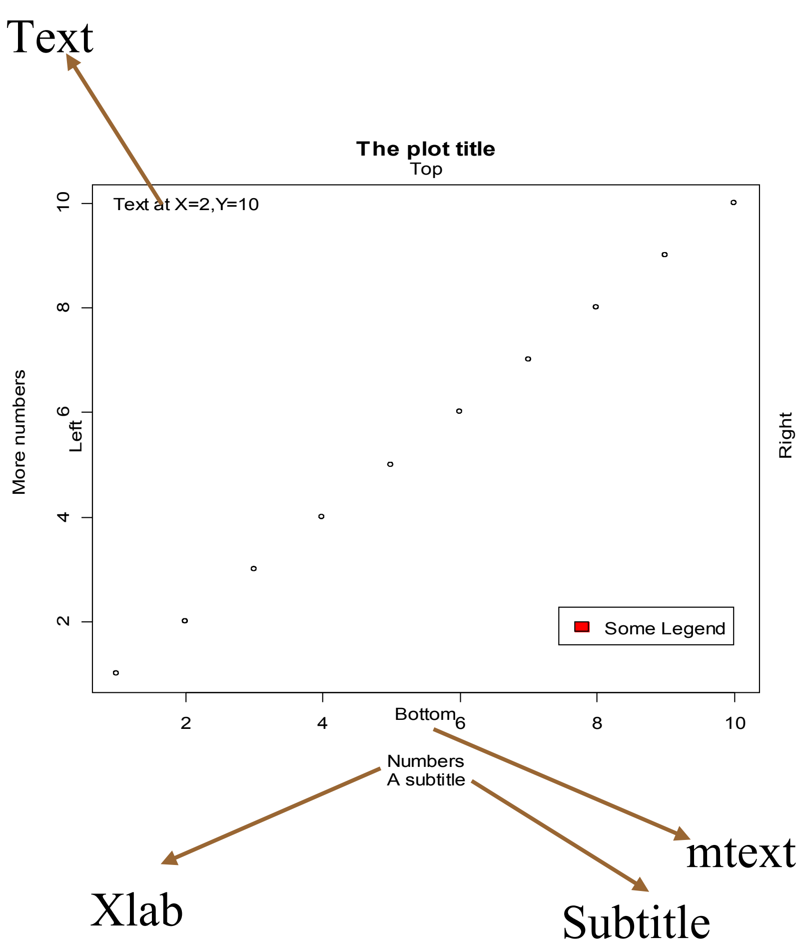
```
legend(x,y, legend=…)
```

- 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

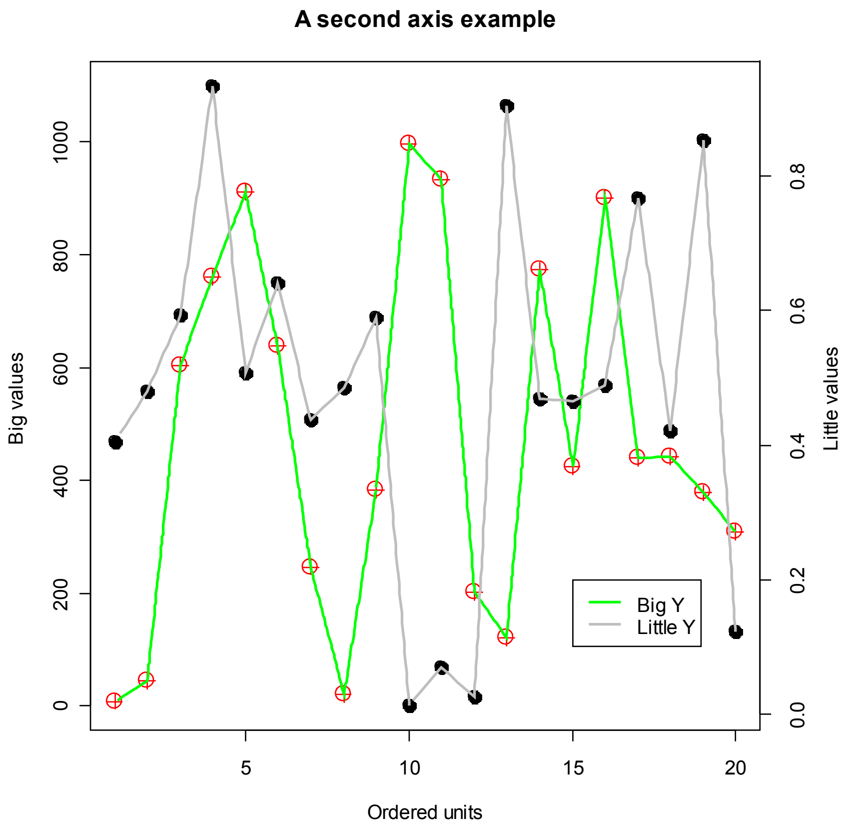
- R `plot` doesn't support multiple Y axis by default
  - You have to make additional axes yourself!
- Adding custom axis

```
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")
points(x1,y1,type="l",lty=3,lwd=2,col="green")
```

Plot and label first Y series

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

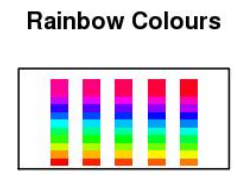
Use of colour in R
Graphics

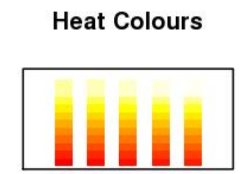
- Colour is usually expressed as a hexadecimal code of Red, Green, and Blue counterparts
  - No good for humans.
- R supports numerous colour palettes which are available through several "colour" functions.
  - `colours()` # get inbuilt names of known colours
    - RGB primaries may take on a decimal 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)

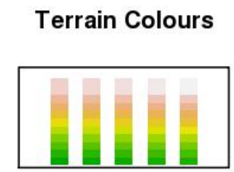
```
> rgb(1,1,1)
[1] "#FFFFFF"

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

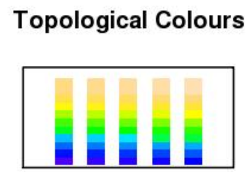
Colour Ramps & Palettes
Graphics

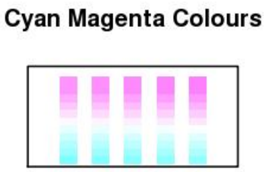
- Heatmaps use colour depth to convey data values. Cold colours are typically low values, and light colours are high state values. This is a colour ramp.
- R supports numerous graded colour charts. Specify *n*, to set the number of gradations required in the palette












```
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")
```

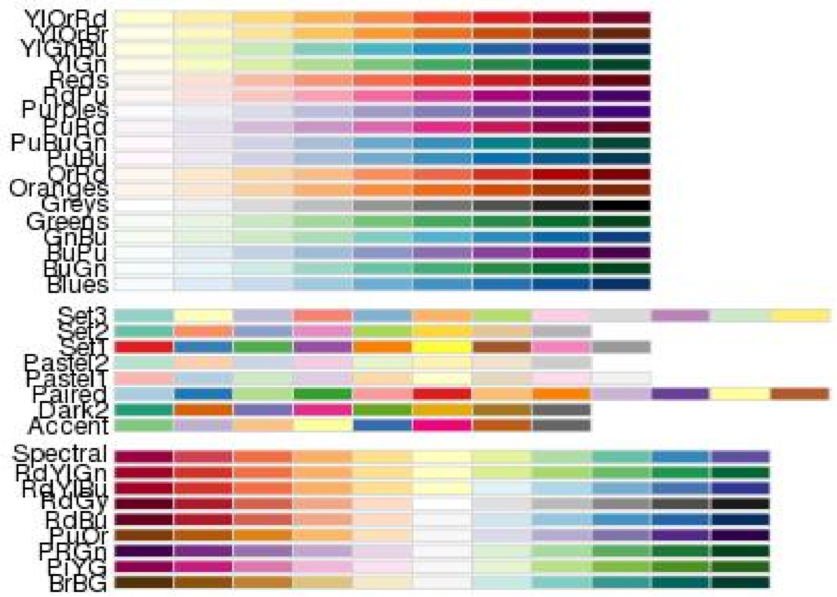
- Check out the available palettes

```
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

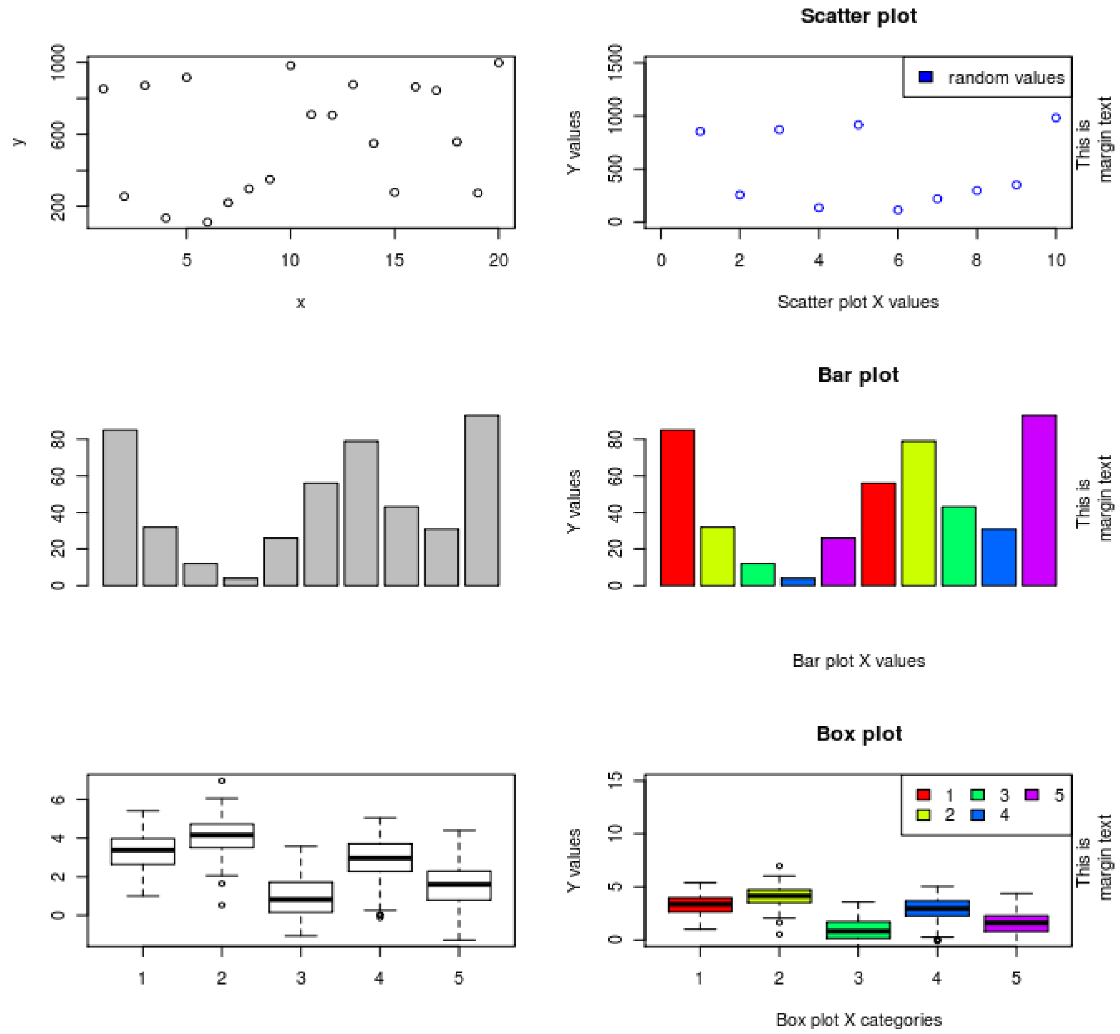
- Exercise:
  - Make a full A4 page figure comprising of 6 plots: 2 each of XY plot (`plot()`), barchart (`barplot()`) and box plots (`boxplot()`)
  - The two version of each plots should consistent of: the default plot and a customised plot (change for instance colours, range, captions...)
  - Output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

Thanks for your attention!

END OF COURSE