

A List of Topics

1. Documentation & help (help pages, vignettes, manuals, ...)
2. R GUIs (mainly R Commander)
3. The OS interface (working directory, data I/O, system access)
4. The workspace, and what sits above it on the search list
5. Improving the R experience – tinn-R, ESS, other editors
6. R objects (including functions)
7. The packaging of R – *base* base packages, recommended packages, other user contributed packages
8. Classes & methods (S3, S4)
9. Graphics - base graphics (*graphics*), *lattice*, *ggplot2*, *rgl* 3-D, *rggobi* dynamic, plotting expressions
10. Models; univariate & multivariate exploration – see the next slide.

Topics that will not be Covered

- ▶ Package construction
- ▶ Environments, manipulating language constructs, ...
- ▶ Models
 - ▶ linear (NB, linear in the parameters)
 - ▶ GLM
 - ▶ multi-level
 - ▶ time series
 - ▶ classification
- ▶ Multivariate exploration
 - ▶ distance measures
 - ▶ ordination

The R System (Notes, chapter 1):

- ▶ R is currently the environment of choice for
 - ▶ specialists who are implementing new methodology
 - ▶ highly trained professional data analysts.
 - ▶ increasingly, statistically skilled scientists.
- ▶ It is designed for interactive use: the next step may depend on the previous result.
- ▶ Twice-yearly major releases bring improvements & new features.
- ▶ It can be remarkably efficient, even though:
 - ▶ data resides (mostly) in memory
 - ▶ it is an interpreted language (but one command may start a lengthy computation)

Web Sites (Ch 1)

CRAN (Comprehensive R Archive Network; use an Australian mirror):
<http://cran.r-project.org>
Australian CRAN mirror: <http://cran.ms.unimelb.edu.au/>
R homepage: <http://www.r-project.org/>
DAAGUR (Data Analysis & Graphics Using R):
www.maths.anu.edu.au/~johnm/r-book.html
R-downunder:
<http://www.stat.auckland.ac.nz/mailman/listinfo/r-downunder>
Wikipedia:
[http://en.wikipedia.org/wiki/R_\(programming_language\)](http://en.wikipedia.org/wiki/R_(programming_language))
Other useful web pages: Start an R session, click on Help, then on Html help, & then under Resources.

Packages (Chapter 1 & Appendix A)

Under Windows & the MacOS X, with an internet connection, use the relevant R menu item to install packages. (usually easier than downloading, then installing).

Note the CRAN task views, which may help in locating packages.

Packages do most of R's work. They make the system extendable without limit.

Command line calculations (Notes, Section 2.1)

The **>** at the start of the line is the command prompt. User commands are typed following this prompt:

```
> 2+2
[1] 4
> 555+83+427+254
[1] 1319
```

The value that is returned is 'printed' on the output line. The [1] says "first (& here, only) element will follow".

Element numbers are helpful when output spills over to a 2nd line

```
> 11:36
[1] 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
[18] 28 29 30 31 32 33 34 35 36
```

The 18th element is 28.

Syntax (Section 2.1)

Command separator	End of line (providing command is complete) or ; <code>print(2+2); print(2+3)</code>
Printing	Output that goes to the command line is 'printed' (explicit use of <code>print()</code> is then unnecessary)
Quitting	To quit from R type <code>q() # NB q(), not q</code>
Case matters	<code>volume</code> is different from <code>Volume</code>
Assignment	The assignment symbol is <code><-</code> , e.g. <code>volume <- c(351, 955, 662, 1203, 557)</code> <i># Store the column of numbers in volume</i>
Comments	Introduce with <code>#</code>

Demonstrations, Help & Help Examples (Sec 3.4)

Demonstrations

```
demo(graphics) # Start graphics demonstrations
demo() # List all available demonstrations
```

Examples

```
example(plot) # Examples from help page for plot()
par(ask=FALSE)
```

Help

```
help() # Describe the use of help()
help(plot) # help on the plot function
help.start() # Open a browser interface to R help
# resources
```

Note also `help.search()`, `apropos()` and `help.start()`

Utility Functions (Sections 2.4 & 3.1)

Functions that act on the contents of the workspace

```
ls() # List contents of workspace
ls(pattern="cr") # List objects whose names include the
# character string "cr"
rm(x, y, z) # Remove x, y, and z from workspace
rm(list=c("x", "y", "z")) # Alternative to rm(x, y, z)
rm(list=ls()) # Remove contents of workspace
```

Functions that access the working or other specified directory

```
## Functions that act on the working or other directory
dir() # List contents of working directory
file.show() # List file contents on screen.
```

Columns of data (Sec 2.2 & 4.1)

```
> c(351, 955, 662, 1203, 557, 460)
[1] 351 955 662 1203 557 460
volume <- c(351, 955, 662, 1203, 557, 460)
type <- c("Guide", "Guide", "Roadmaps", "Roadmaps",
"Roadmaps", "Guide"),
description <- c("Aird's Guide to Sydney",
"Moon's Australia handbook",
"Explore Australia Road Atlas",
"Australian Motoring Guide",
"Penguin Touring Atlas", "Canberra - The Guide")
```

Data Frames – Lists of Columns (Sec 2.3 & 5.2.2)

```
thickness <- c(1.3, 3.9, 1.2, 2, 0.6, 1.5)
width <- c(11.3, 13.1, 20, 21.1, 25.8, 13.1)
height <- c(23.9, 18.7, 27.6, 28.5, 36, 23.4)
weight <- c(250, 840, 550, 1360, 640, 420)
## volume, type & description were input earlier?
```

This can get unmanageable (many objects). We might prefer:

```
travelbooks <- data.frame(
  thickness = c(1.3, 3.9, 1.2, 2, 0.6, 1.5),
  width = c(11.3, 13.1, 20, 21.1, 25.8, 13.1),
  ...
  type = type # type was created earlier
  row.names = description # description was created earlier
)
```

Data frames offer a tidy way to supply data to modeling functions.

Put `travelbooks.txt` in Place, Ready for Input (Sec 2.4)

Place the file where it can readily be found

```
library(DAAGxtras) # DAAGxtras has the needed function
datafile("travelbooks") # Place file in directory
dir() # Check contents of working directo
```

NB: The function `datafile()` is merely a convenience, for use in an R tutorial, for making sure that the file is in a known location.

Other possible arguments include `"fuel"` (places `fuel.txt` in working directory), `"oneBadRow"`, `"scan-demo"`, `"molclock1"`, `"molclock2"`, `"bestTimes"` and `"bostonnc"`.

Input to a Data Frame (Sec 2.4)

Display contents of `travelbooks.txt`

NB: As we are assuming that file is in the working directory, it is not necessary to specify the path.

```
> file.show("travelbooks.txt")
"thickness" "width" "height" "weight" "volume" "type"
"Aird's Guide to Sydney" 1.3 11.3 23.9 250 351 "Guide"
"Moon's Australia handbook" 3.9 13.1 18.7 840 955 "Guide"
"Explore Australia Road Atlas" 1.2 20 27.6 550 662 "Roadmaps"
"Australian Motoring Guide" 2 21.1 28.5 1360 1203 "Roadmaps"
"Penguin Touring Atlas" 0.6 25.8 36 640 557 "Roadmaps"
"Canberra - The Guide" 1.5 13.1 23.4 420 460 "Guide"
```

Input file contents to `travelbooks`

```
> ## Now input the file
> travelbooks <- read.table("travelbooks.txt")
```

Accessing Data Frame Columns (Sec 2.3 & 5.2)

```
travelbooks[, 4]
travelbooks[, "weight"]
travelbooks$weight
travelbooks[["weight"]]
```

Repeated reference to `travelbooks` is unnecessary!

```
rm(weight, volume) # If present, remove from the workspace
attach(travelbooks)
plot(weight ~ volume)
cor(weight, volume)
detach(travelbooks)
```

For one or a few statements, use `with()` (ephemeral attachment):

```
with(travelbooks, cor(weight, volume))
```

To execute a block of code, enclose it within braces (`{}`)

The Working Environment (Notes, Ch 3)

Working directory	R will by default read files from this directory, or write files to it
Object	A data structure or function that R recognizes Functions, as well as data, exist as "objects" Note also, e.g., formula objects, expression objects, ...
Workspace	This is the user's "database", where the user can make additions or changes, or delete objects, as desired. Use <code>ls()</code> to list contents of current workspace.
<code>read.table()</code>	Use to read data, from a file, into the workspace
Image files	Use to store R objects, e.g., workspace contents. (The expected file extension is <code>.RData</code> or <code>.rda</code>)
<code>save.image()</code>	Use to store all or some workspace contents. For safety, use from time to time in a session Alternatively, use the relevant menu item.

Packages, and the Search List

Packages	Packages are collections of R functions and/or data. <code>library()</code> Use to attach a package, e.g. <code>library(DAAG)</code> (Binary R distributions include recommended packages. Install other packages, as required, prior to their use.)
Search List	This is a list of "databases", starting with the working directory, that are searched in turn for R objects.
Databases	Other "databases" that can be added to the search list include image (<code>.RData</code>) files and data frames (use <code>attach()</code>), and packages (use <code>library()</code>).

Worked Examples (Ch 4)

► World record times for track and field events:

```
library(DAAGxtras)
str(worldRecords) # Check columns in data
library(lattice) # Prepare to plot data
xyplot(Time ~ Distance, groups=roadORtrack,
       data=worldRecords, scales=list(log=10),
       auto.key=list(columns=2))
## Fit regression line
lm(log(Time) ~ log(Distance), data=worldRecords)
```

► Regression with two explanatory variables; the `nihills` data.

```
spolm(~ log(nihills),
      varnames=c("log(dist)", "log(climb)",
                "log(time)", "log(timef)"))
```

► Exploration of time series data – Australian rain records.

Different types of data objects:

Vectors	These collect together elements that are all of one mode. (Possible modes are "logical", "integer", "numeric", "complex", "character" and "raw")
Factors	Factors identify category levels in categorical data. Modeling functions know how to represent factors. (Factors do not quite manage to be vectors! Why?)
Data frame	A list of columns – same length; may have different modes. Data frames are commonly a huge help for organizing data.
Lists	Lists group together an arbitrary collection of objects. (These are recursive structures; elements of lists are lists.)
NAs	The handling of <code>NAs</code> (missing values) can be tricky.

All R objects have a length, which can be 0. (Why is this useful?)

Vectors (Notes, ss 5.1.1)

Subsets of Vectors

```
z <- c(2,3,5,2,7,1)
z[c(1:3, 5)]      # Elements 1 to 3 and 5
z[-c(3,5)]        # All except elements 3 & 5
subset(z, z>2)    # Extract elements that are > 2
```

Names for Vector Elements

```
> booksales <- c(Dec07=555, Jan07=83,
+                 Feb07=427, Mar07=254)
> booksales[c("Jan07", "Feb07")]
Jan07 Feb07
 83  427
```

Factors (ss 5.1.2)

Create a character vector

```
> field <- c("Med", "Lit", "Chem", "Med", "Med")
> field
[1] "Med" "Lit" "Chem" "Med" "Med"
# Nobel winners: Katz, White, Comforth, Doherty, Marshall
```

From `field`, create the factor `fieldfac`

```
> fieldfac <- factor(field)
> fieldfac
[1] Med Lit Chem Med Med
Levels: Chem Lit Med
> unclass(fieldfac)
[1] 3 2 1 3 3
attr("levels")
[1] "Chem" "Lit" "Med"
```

Notice that, by default, the levels are taken in alphanumeric order.

Different Kinds of Functions (Sec 5.4)

Generic functions	They examine the object given as argument, before deciding what action is needed. Examples include <code>print()</code> , <code>plot()</code> & <code>summary()</code>
Modeling functions	Use to fit statistical models. Thus note <code>lm()</code> for <i>linear</i> modeling. Output may be stored in a model object.
Extractor functions	Use extractor functions to obtain summaries, coefficients, residuals, etc., from model objects. e.g., <code>summary()</code> , <code>coef()</code> , <code>residuals()</code>
User	Create functions that automate & document computations
Anonymous	Functions that are defined in place do not need a name

Vectors – Useful Functions (Notes 5.4.1)

Any mode of vector	<code>length()</code> , <code>rev()</code> , <code>sort()</code> , <code>order()</code> , <code>unique()</code> , <code>is.factor()</code> , <code>is.na()</code> ; also other analogous functions
numeric	<code>sum()</code> , <code>cumsum()</code> , <code>mean()</code> , <code>sd()</code> , <code>range()</code> , <code>diff()</code>
character	<code>paste()</code> , <code>nchar()</code> , <code>substring()</code> , <code>grep()</code> ^a and friends, <code>strsplit()</code> ^b , <code>charmatch()</code>
logical	<code>any()</code> , <code>all()</code> e.g., <code>any(x>0)</code>

To search for a needed function, guess a character string that might appear in the name, e.g., `str` or `char` for character manipulations.
Then do, e.g.

```
help.search("str", package="base")
```

or

```
apropos("str")
```

^aNote the parameter `fixed`.

^bAgain, note the parameter `fixed`.

Functions that create vectors (Notes 5.4.1)

```
numeric(5)      Creates a numeric vector of length 5, all elements 0.
numeric(0)      Numeric vector of length 0.
logical(5)      Logical vector of length 5, all elements FALSE.
character(5)    Character vector of length 5, all elements "".
```

Check or change (coerce) class

```
> as("1.23", "numeric")  # Equivalently, as.numeric("1.23")
[1] 1.23
> as(TRUE, "numeric")
[1] 1
> as(1.23, "character") # Equivalently paste(1.23)
[1] "1.23"
```

Functions that are useful with data frames (ss 5.4.1)

<code>names()</code>	Names of columns
<code>row.names()</code>	Row names
<code>dim()</code>	Dimensions (as for a matrix argument)
<code>summary()</code>	Summary details, e.g., <code>summary(travelbooks)</code>
<code>str()</code>	A different summary, e.g., <code>str(travelbooks)</code>
<code>sapply()</code>	Apply function columnwise: <code>sapply(travelbooks, is.factor)</code> <code>sapply(travelbooks[, 1:4], mean)</code>
<code>plot()</code>	<code>plot()</code> does indeed accept a data frame as argument.

Note the possibility of using anonymous functions with `sapply()`
`sapply(travelbooks, function(x)if(is.factor(x))levels(x))`

User-defined Functions (ss 5.4.2)

```
mean.and.sd <- function(x){  
  av <- mean(x)  
  sdev <- sd(x)  
  c(mean=av, sd = sdev) # Return value (vector)  
}
```

The usage is:

```
function( arglist ) { expr  
  return(value)  
}
```

Default Arguments are a Good Idea

```
mean.and.sd <- function(x=rnorm(20)){  
  av <- mean(x)  
  sdev <- sd(x)  
  c(mean=av, sd = sdev) # Return value (vector)  
}
```

Tables and Cross-tabulation (ss 5.4.4); `table()`

```
> library(DAAGxtras) # Get data frame nassCDS from here  
> ## First count numbers of records. (Misleading?)  
> ## I: Use table()  
> with(nassCDS, table(sex, dead)) # NB: unweighted  
  dead  
sex alive dead  
f 11784 464  
m 13253 716
```

Tables and Cross-tabulation (ss 5.5): `xtabs()`

The function `xtabs()` has provision to give each observation a frequency (or weight).

```
> ## II: Use xtabs()  
> xtabs(~ sex + dead, data=nassCDS) # NB: unweighted  
+  
> ## Now weight records a/c 1/(sampling fraction)  
> xtabs(weight ~ sex + dead, data=nassCDS)  
  dead  
sex      alive      dead  
f 5899999.64  25677.26  
m 6167937.23 39917.87
```

Review, & Additional Points

- ▶ Vignettes (Sec 2.5) are pdf files that accompany packages
- ▶ Saving into and retrieving objects from image files (3.2)
- ▶ Attaching image files (3.3.2)
- ▶ Matrices (5.3)
- ▶ Lists (5.2.6)
- ▶ User functions (5.4.2)
- ▶ Common sources of difficulty (5.7).

Next:

Base & Trellis Graphics

Base or "Traditional" Graphics (Sec 6.1)

Base graphics comprises `plot()` and allied functions

Functions `plot()`, `points()`, `lines()`, `text()`, `mtext()`, `axis()`, `identify()` etc. form a suite

Choice: old `plot(x, y)` syntax vs newer `plot(y ~ x)` formula syntax:
`plot(x, y)` with(women, `plot(height, weight)`)
`plot(y ~ x)` `plot(weight ~ height, data=women)`

`par()`, etc Use to set parameters in base graphics (Sec 6.2)
Some parameters must be set using `par()`
Specify others in function call. Or there may be a choice.

NB: Some base graphics functions do not take a `data` parameter

Other (mostly 2-D) graphics:

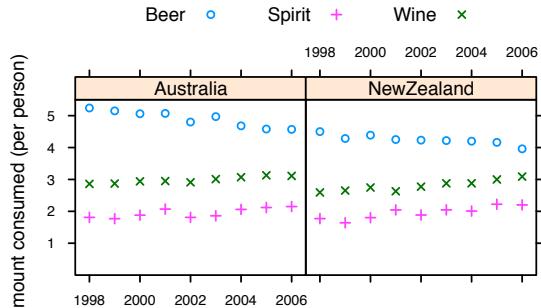
(i) lattice (trellis) graphics, using the `lattice` package,
and (ii) the low-level `grid` package on which `lattice` is built.

3-D Graphics: Note `rgl`, `misc3d` and `tkrplot`.

Lattice Graphics (Notes Sec 6.2–6.5)

Lattice	Lattice is a flavour of trellis graphics (the S-PLUS flavour was the original implementation)
Grid	<code>grid</code> is a low-level graphics system. It was used to build <code>lattice</code> . For <code>grid</code> , see Part II of Paul Murrell's <i>R Graphics</i>
ggplot2	<code>ggplot2</code> is an R implementation of Wilkinson's <i>Grammar of Graphics</i> . It has some nice features.
Lattice vs base	Lattice is more structured, automated and stylized. Much is done automatically, without user intervention. Changes to the default style are harder than for base.
Lattice syntax	Lattice syntax is consistent and tightly regulated For use of lattice, graphics formulae are mandatory.

Beer+Wine+Spirit, conditioning on Country



```
xyplot(Beer+Spirit+Wine ~ Year | Country, data=grog,
       outer=FALSE, auto.key=list(columns=3))
```

NB: Code has been simplified; next slide has full details.

Beer+Wine+Spirit, conditioning on Country, with frills

```
grogplot <-
  xyplot(Beer+Spirit+Wine ~ Year | Country, data=grog,
         outer=FALSE, auto.key=list(columns=3))
```

Send output from `update()` to command line, causing ‘printing’

```
update(grogplot, ylim=c(0,5.5),
       xlab="", ylab="Amount consumed (per person)",
       par.settings=simpleTheme(pch=c(1,3,4)))
```

Alternatively, spell out the details – ‘print’ explicitly

```
frillyplot <-
  update(grogplot, ylim=c(0,5.5),
         xlab="", ylab="Amount consumed (per person)",
         par.settings=simpleTheme(pch=c(1,3,4)))
print(frillyplot)
```

Simple Lattice Examples

Conditioning (|) – separate panels (ss 6.2.2)

```
xyplot(Beer ~ Year | Country, data=grog)
```

Overlaid plots – use `groups` parameter (ss 7.1.1)

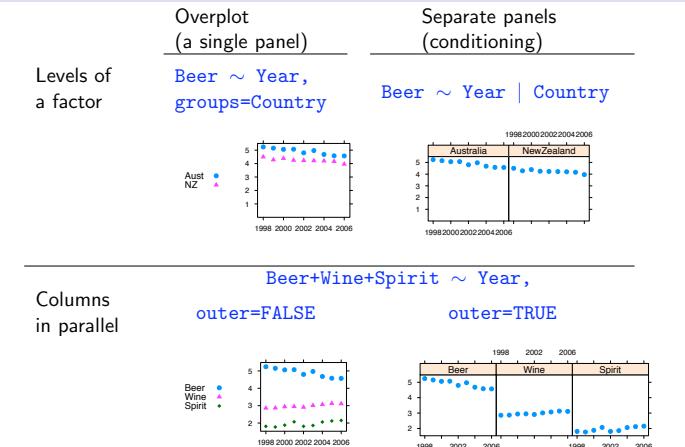
```
xyplot(Beer ~ Year, groups=Country, data=grog)
```

Use `auto.key` for a basic key to the labeling (`groups` parameter).

Parallel plots - separate panels (ss 6.2.2)

```
xyplot(Beer+Wine+Spirit ~ Year, data=grog)
```

Grouping of Points, and Columns in Parallel (Sec 6.2.2)



Lattice parameter settings (Sec 6.2.3)

1. The ‘theme’ determines point and line settings. Changes are readily made using `simpleTheme()` (recent version of *lattice*).
2. For axis, axis tick, tick label and axis label settings use the argument `scales` in the function call.
3. Lattice objects can be created, then updated – use `update()`.
4. Note also the arguments `layout` (# rows \times # columns \times # pages) and `aspect` (aspect ratio).
5. The `type` argument can specify any combination of `p` (points), `l` (lines), `b` (points & lines), `r` (regression lines) and `smooth` (a smooth curve). Set `span` to control the smoothness of any curve.

Use of `simpleTheme()` for Point & Line Settings

First use `simpleTheme()` to create a “theme” with the new settings:

```
miscSettings <- simpleTheme(pch = 16, cex=1.25)
```

Alternatives are then:

(i) Supply the “theme” to `par.settings` in the function call.
[This stores the settings with the object. These stored settings over-ride the global settings at the time of printing.]

```
xyplot(Beer ~ Year | Country, data=grog,
       par.settings=miscSettings)
```

(ii) Supply the “theme” to `trellis.par.set()`, prior to plotting:
[Makes the change globally, until a new trellis device is opened]

```
trellis.par.set(miscSettings)
xyplot(Beer ~ Year | Country, data=grog)
```

Linear Models, in the style of `lm()` (Notes 7.1 – 7.2)

Linear model	Any model that <code>lm()</code> will fit is a “linear” model. <code>lm()</code> can fit highly non-linear forms of response!
Diagnostic plots	Use <code>plot()</code> with the model object as argument, to get a basic set of diagnostic plots.
<code>termplot()</code>	If there are no interaction terms, use <code>termplot()</code> to visualize the contributions of the different terms. (Why are interactions a problem for <code>lm()</code> ?)
Factors	In model terms, use factors to model qualitative effects.
Model matrices	How should coefficients be interpreted? Examine the model matrix. (This is an especial issue for factors.)
GLMs	Generalized Linear Models are an extension of linear models, commonly used for analyzing counts.

[NB: `lm()` assumes independently & identically distributed (iid) errors, perhaps after applying a weighting function.]



Models with Non-iid Errors (Notes 7.3)

Error Term	Errors do not have to be (and often are not) iid
Multi-level models	Multi-level models are a (relatively) simple type of non-iid model. Fit using <code>lme()</code> (<code>nlme</code>) or <code>lmer()</code> (<code>lme4</code> package). Such models allow different errors of prediction, depending on the intended prediction. (The error term does matter!)
<code>aov</code> models	For suitably balanced designs, these give the information needed for a multi-level type of analysis. [See Chapters 4 & 7 of DAAGUR]
Time series	Points that are close together in time are likely to show a (usually, positive) correlation. R's <code>acf()</code> and <code>arima()</code> functions are highly useful tools for time series.



Multivariate Models and Methods (Notes Sec 7.3)

Ordination	Principal components, multi-dimensional scaling [D-Ch 12] Multivariate distances – do variables have equal weight? Phylogenetics – distances are from evolutionary model.
2D or 3D views	Ordination may allow a low-dimensional view. Which view is best, or which is the “right” view NB: The “view” can be rotated arbitrarily.
Classification methods	Linear Discriminant Analysis [D-Ch 12]: simple. Trees [D-Ch 11]: simple to fit; may be hard to interpret. Random forests [Ch 11]: easy to fit, superior to trees? Neural nets, SVMs: Watch for exaggerated claims!
Classify, then ordinate	A clear criterion determines the distance measure. Different classifications will give different axes (views).



Key Language Ideas (Notes Ch 11)

Classes	Classes make generic functions (methods) possible. Examples are <code>print()</code> , <code>plot()</code> , <code>summary()</code> , etc.
S4 vs S3	S3 is the original implementation of classes & methods S4, which uses the <code>methods</code> package, is more recent.
Formulae	As of now, there are model, graphics and table formulae. Formulae can be manipulated, just as with other objects.
Expressions	They can be evaluated (of course!). They can also be printed (on a graph)
Argument lists	Argument lists can be constructed in advance, as a list of named values, with <code>do.call()</code> then used to pass the argument list to the function
Environments	Environments hold various subtleties. There are basic matters that it helps to know.



THE END

You may think that this is the end,
Well it is, but to prove we're all liars,
We're going to sing it again,
Only this time we'll sing a little higher.

Actually, this is not the end, for there are many other analysis methods and R packages to explore, even if not in this workshop!

