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)`

For other useful web pages, start an R session, click on the menu item Help, click on Html help, & look under Resources on the browser window that should then appear.

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
> 2 > 1
[1] TRUE
```

The [1] says "first (& here, only) element will follow"

Syntax (Section 2.1)

Command separator	End of line (providing command is complete) or `;` `print(2+2); print(2+3)`
Quitting	To quit from R type `q()  # NB q(), not q`
Case matters	`volume` is different from `Volume`

Assignment	The assignment symbol is `<-`, e.g.
	`volume <- c(351, 955, 662, 1203, 557)`
	`# Store the column of numbers in volume`

Comments	Introduce with `#`

Demonstrations, Help & Help Examples (Sec 2.5)

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 2.3, 3.2 & 3.3)

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, 2.3 & 5.1)

```
> c(351, 955, 662, 1203, 557, 460)
[1]  351  955  662 1203  557  460
```

Assignment to a vector object (Sec 2.3)

```
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 earlie
   )
```

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

Input to a Data Frame (ss 2.4.1)

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

Display contents of **travelbooks.txt**

```
> 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"
> ## 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 statments, use `with()` (ephemeral attachment):

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

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

The Working Environment (Notes, 3.1 – 3.3)

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 `ls()` to list contents of current workspace.
`read.table()`	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 **.RData** or **.rda**)
`save.image()`	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.
`library()`	Use to attach a package, e.g. `library(DAAG)` (Binary R distributions include recommended packages. Install other packages, as required, prior to their use.)
Search List	The search list specifies the working directory, followed by other "databases" that should be searched if the object sought is not in the working directory.
Databases	Other "databases" that can be added to the search list include image (**.RData**) files, and data frames.

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.

```
splom(~ 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.)
`NA`s	The handling of `NA`s (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 `print()`, `plot()` & `summary()`
Modeling functions	Use to fit statistical models. Thus note `lm()` for *linear* 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., `summary()`, `coef()`, `residuals()`
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	`length()`, `rev()`, `sort()`, `order()`, `unique()`, `is.factor()`, `is.na()`; also other analagous functions
numeric	`sum()`, `cumsum()`, `mean()`, `sd()`, `range()`, `diff()`
character	`paste()`, `nchar()`, `substring()`, `grep()`[a] and friends, `strsplit()`[b], `charmatch()`
logical	`any()`, `all()` e.g., *any(x>0)*

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")
```

[a] Note the parameter `fixed`.

[b] Again, 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)

`names()`	Names of columns
`row.names()`	Row names
`dim()`	Dimensions (as for a matrix argument)
`summary()`	Summary details, e.g., `summary(travelbooks)`
`str()`	A different summary, e.g., `str(travelbooks)`

`sapply()`	Apply function columnwise:
	`sapply(travelboooks, is.factor)`
	`sapply(travelboooks[, 1:4], mean)`

`plot()`	`plot()` 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 5.4.3)

```
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 (Section 6.2); `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 (Section 6.2): `xtabs()`

```
> ## 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 (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 (5.2.5 & 5.2.6)
- User functions (5.4.3)
- Common sources of difficulty (5.8).

Next:

Base & Trellis Graphics

Base or "Traditional" Graphics (Notes Sections 7.1 & 7.2)

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:

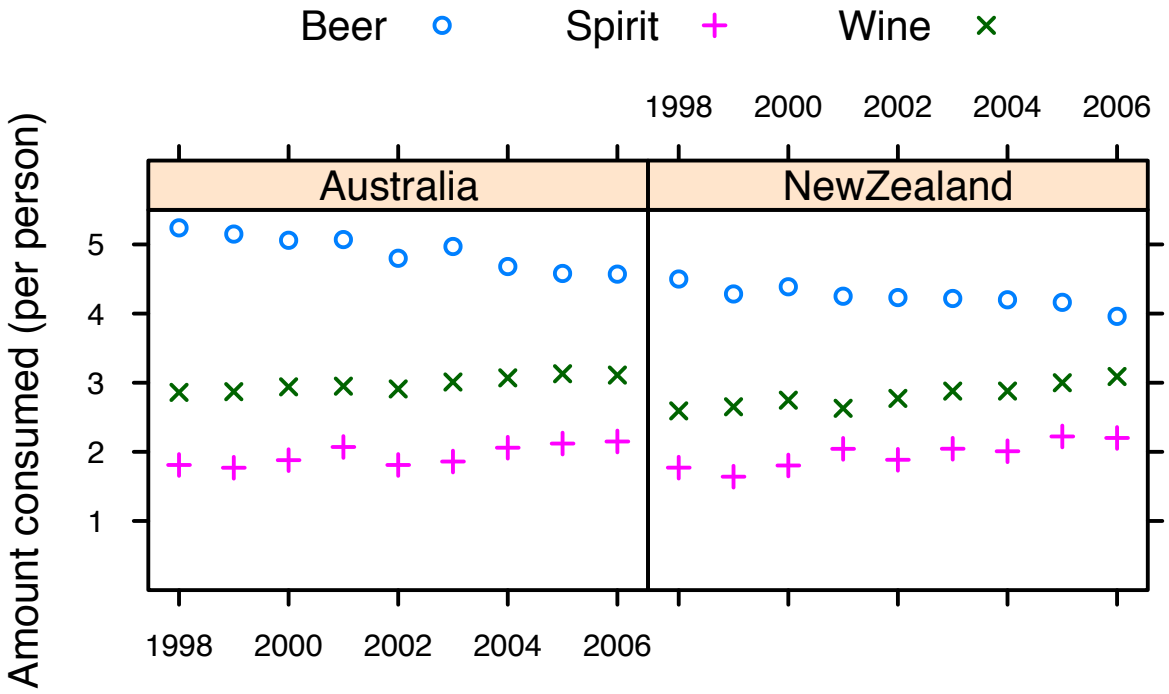
(i) lattice (trellis) graphics, using the *lattice* package,
and (ii) the low-level *grid* package on which *lattice* is built.

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

Lattice Graphics (Notes Section 7.3)

Lattice	Lattice is a flavour of trellis graphics (the S-PLUS flavour was the original implementation)
Grid	*grid* is a low-level graphics system. It was used to build *lattice*. For *grid*, see Part II of Paul Murrell's *R Graphics*
ggplot2	*ggplot2* is an R implementation of Wilkinson's *Grammar of Graphics*. 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 7.3.1)

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

Overlaid plots – use `groups` parameter (ss 7.3.1)

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

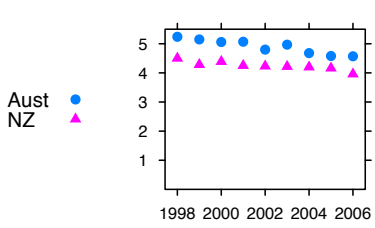
Use `auto.key` for a basic key to the labeling (`groups` parameter).

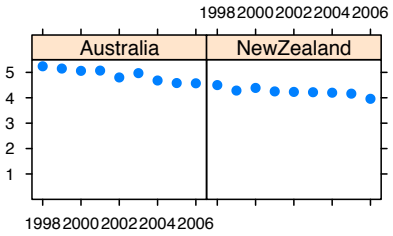
Parallel plots - separate panels (ss 7.3.1)

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

Grouping of Points, and Columns in Parallel

	Overplot (a single panel)	Separate panels (conditioning)
Levels of a factor	`Beer ~ Year, groups=Country`	`Beer ~ Year | Country`

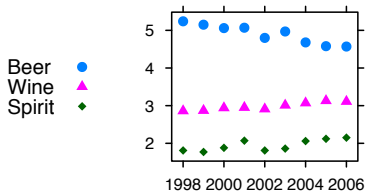




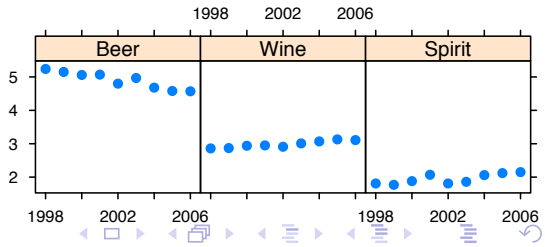
Columns in parallel

`Beer+Wine+Spirit ~ Year,`

`outer=FALSE`



`outer=TRUE`



Lattice parameter settings

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.]

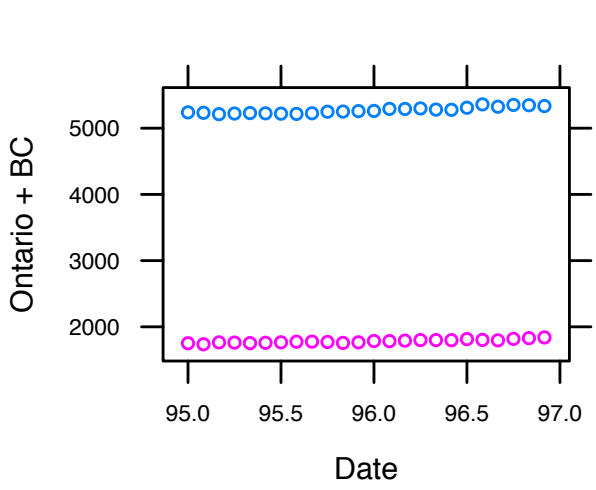
```
xyplot(Beer ~ Year | Country, data=grog,
       par.settings=miscSettings)
```

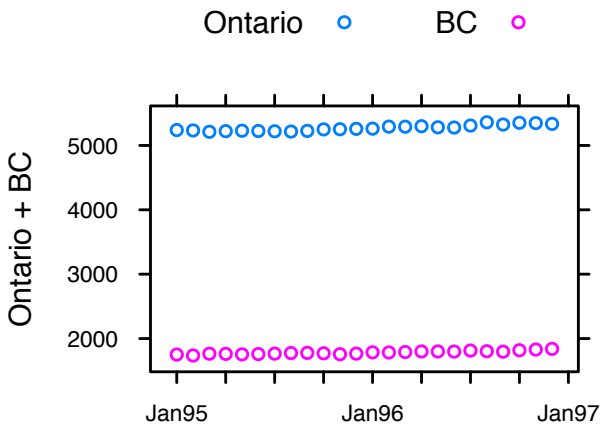
(ii) Supply the "theme" to `trellis.par.set()`, prior to plotting:
[Makes the change globally, until a new trellis device is opened]

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

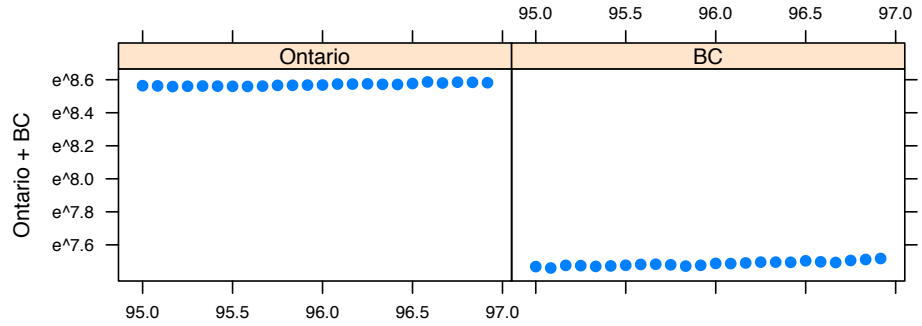
Axis, tick, tick label and axis label settings

```
jobplot <- xyplot(Ontario+BC ~ Date, data=jobs)
## Half-length ticks, each quarter, Label years, Add key
    tpos <- seq(from=95, by=0.25, to=97)
    tlabs <- rep(c("Jan95", "", "Jan96", "", "Jan97"),
                 c(1,3,1,3,1))
update(jobplot, auto.key=list(columns=2), xlab="",
       scales=list(tck=0.5, x=list(at=tpos, labels=tlabs)))
```



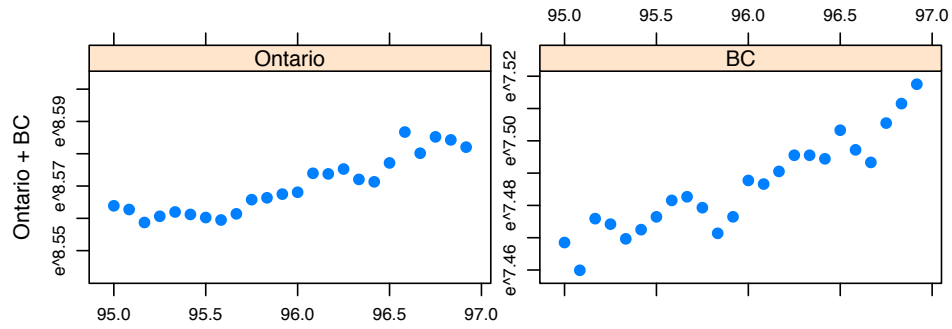


Now use logarithmic *y*-scale



Natural log scale

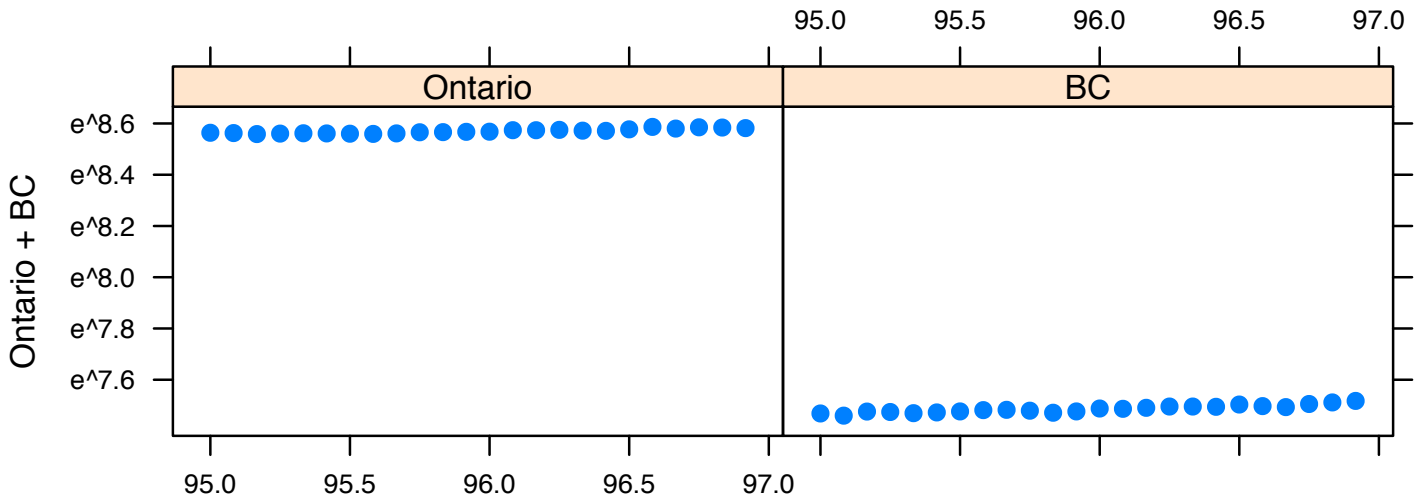
```
logplot <-
  xyplot(Ontario+BC ~ Date, data=jobs, outer=TRUE,
         xlab="", scales=list(y=list(log="e")))
```



Natural log scale, `"sliced"`

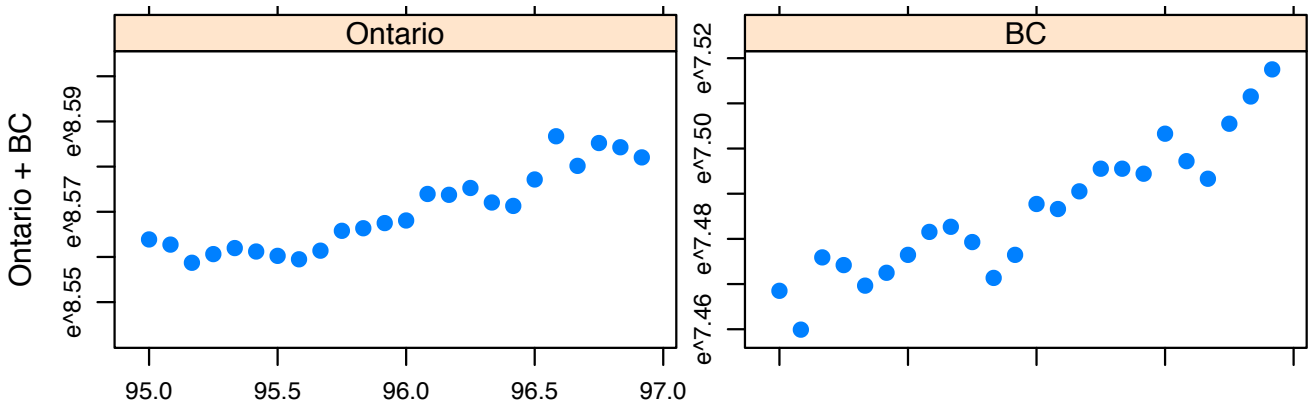
```
update(logplot, scales=list(y=list(relation="sliced")))
```

Now use logarithmic *y*-scale



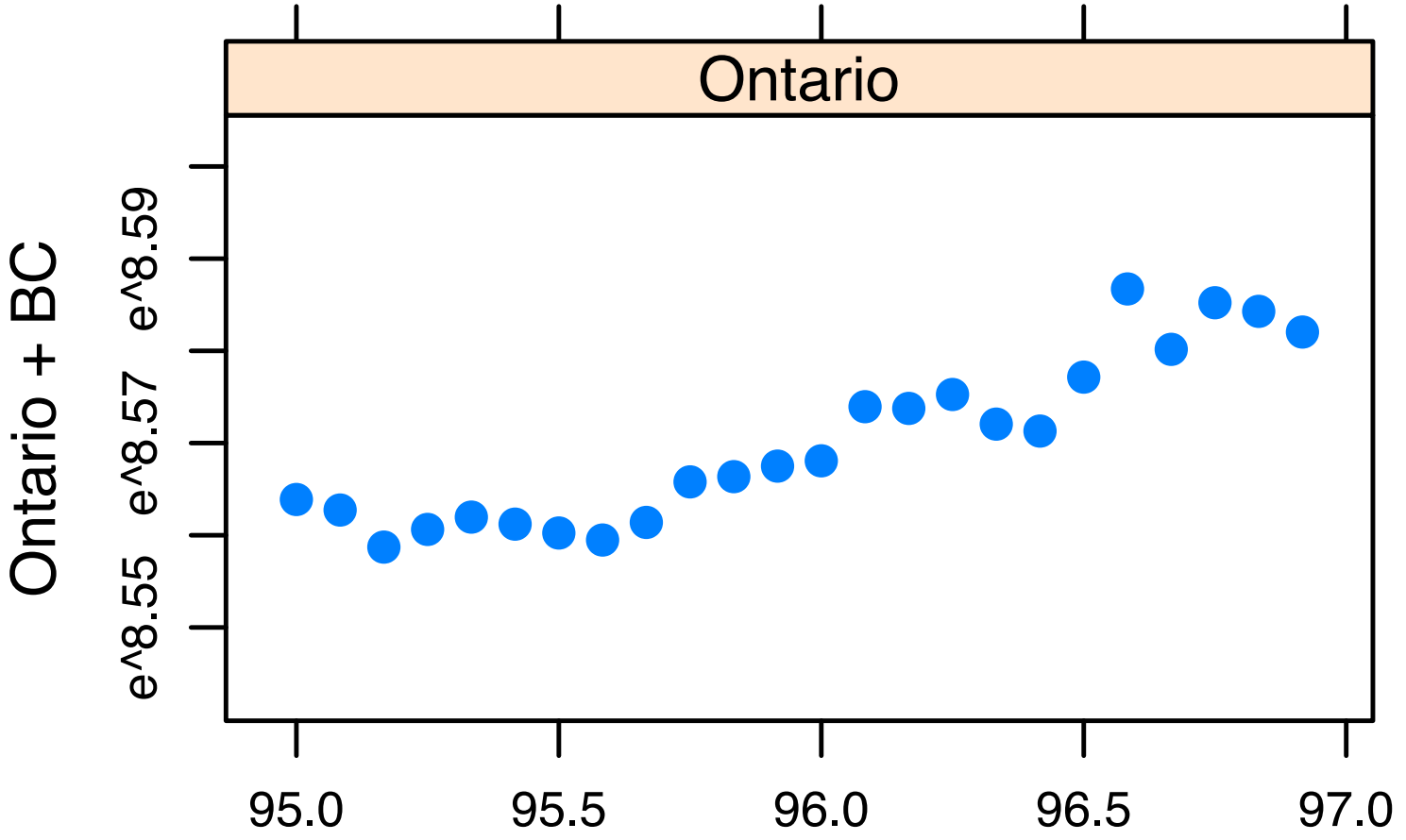

```
logplot <-
  xyplot(Ontario+BC ~ Date, data=jobs, outer=TRUE,
         xlab="", scales=list(y=list(log="e")))
```

Now use logarithmic *y*-scale




```
update(logplot, scales=list(y=list(relation="sliced")))
```

Now use logarithmic y-scale



Grammar of Graphics (Notes Section 7.4)

ggplot2 Implements Wilkinson's *Grammar of Graphics*
At its best, it combines simplicity & power.
Use *grid* functions to extend *ggplot2* abilities.

```
## Plot annual rainfall vs Year; add smooth
qplot(Year, seRain, data=bomsoi, geom=c("point","smooth"))

## Scatterplot, add 2-d density contours, by sex
qplot(wt, ht, data=ais, geom=c("point", "density2d"),
      facets = sex ~.)
  # In lattice terminology; condition on sex.

## Boxplots, by sport, with panel split by sex
qplot(sport, ht, data=ais, geom="boxplot",
      facets = sex ~.)
```

The geometry tells all!

Linear Models, in the style of `lm()` (Notes 8.1 – 8.2)

Linear model	Any model that `lm()` will fit is a "linear" model. `lm()` can fit highly non-linear forms of response!
Diagnostic plots	Use `plot()` with the model object as argument, to get a basic set of diagnostic plots.
`termplot()`	If there are no interaction terms, use `termplot()` to visualize the contributions of the different terms. (Why are interactions a problem for `lm()`?)
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 8.3)

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 `lme()` (*nlme*) or `lmer()` (*lme4* package). Such models allow different errors of prediction, depending on the intended prediction. (The error term does matter!)
`aov` 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 `acf()` and `arima()` functions are highly useful tools for time series.

Multivariate Models and Methods (Notes Ch 10)

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).

Ordination (Sec 10.1)

Road Distances example

Can we recover the geographical configuration?

Calculate distances from points in *n*-space

Is a "good" representation possible in dimension 2 or 3?

NB: How should variables be weighted? Equally?

Phylogenetics

Distances should reflect time from Last Common Ancestor!

C.f. the `dist.dna()` function in *ape*. Choose between:
raw, JC69, K80 (default), F81, K81, F84, BH87, T92, TN93, GG95, logdet, & paralin.

Different models for evolutionary imply different distances.

There may not be a unique distance between two organisms.

Classification (Sec 10.2)

Linear Methods (ss 10.2.1)

The notes have examples of the use of `lda()` and `qda()`, both from the *MASS* package.

Tree-based Methods (ss 10.2.2)

These are about as non-parametric as is possible – a strong contrast with `lda()` and `qda()`. The notes demonstrate the use of `rpart()` which generates trees, and of `randomForest()` which generates forests of trees. The functions take their names from the packages in which they are the main workhorses.

Key Language Ideas (Notes Ch 11)

Classes	Classes make generic functions (methods) possible.
Methods	Examples are `print()`, `plot()`, `summary()`, etc.
S4 vs S3	S3 is the original implementation of classes & methods S4, which uses the *methods* 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 constucted in advance, as a list of named values, with `do.call()` then used to pass the argument list to the function
Environments	Environments hold various subtleties. There are basic matters that it helps to know.

Additional Notes (Notes Ch 12)

Errors in data input	My attempt to input data has generated an error. How can I locate it?
`scan()`	`scan()` is a more flexible alternative to `read.table()`
`sapply()` & friends	`sapply()`, `lapply()` and `apply()` apply functions in parallel across all columns of a data frame or (`apply()`) across all rows or columns of a matrix. Apply any function that will not generate an error. [e.g., `log("Hobart")` is not allowed.]
`Inf` & friends	The logarithm of zero returns `-Inf`. Take care!
Large datasets	A little knowhow can save a load of time.
Workspaces	Manage them carefully!

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!