



Introduction to Programming in R

Course Website: <http://bcb.dfci.harvard.edu/~aedin/courses/R/>

CENTERS FOR DISEASE CONTROL AND PREVENTION

26-28TH JUNE 2012

Aedin Culhane

Email: aedin@jimmy.harvard.edu

<http://bcb.dfci.harvard.edu/~aedin>

<http://www.hsph.harvard.edu/research/aedin-culhane/>

Contents

1	Introduction to R Software	2
1.1	Obtaining R and RStudio	2
1.2	The default R interface	2
1.2.1	Default R Console	3
1.2.2	Default R Editor	3
1.2.3	R Shortcuts	4
1.3	RStudio Interface	5
1.3.1	console	5
1.3.2	Editor	6
1.3.3	Workspace, history	6
1.3.4	Files, Plots, Packages, Help	7
1.3.5	Projects, SVN in RStudio	7
1.4	Starting out - setting a working directory	7
1.4.1	Managing and accessing folders in R	8
1.5	First R Encounter	11
1.6	Getting help with functions and features	11
1.7	R as a big calculator	12
1.8	A few important points on R	13
1.8.1	Use Arrows to key browse command history	15
1.9	Basic operators	16
1.9.1	Comparison operators	16
1.9.2	Logical operators	16
1.10	Data sets in R	16
1.11	R sessions (workspace) and saving session history	18
1.12	R Packages	19
1.13	Installing new R libraries	19
1.14	Customizing Startup	22
2	Objects in R	24
2.1	Using ls and rm to managing R Objects	24
2.2	Types of R objects	24
2.3	Attributes of R Objects	29
2.4	Creating and accessing objects	31
2.5	Modifying elements	33
2.5.1	Sorting and Ordering items	34
2.5.2	Missing Values	36
2.5.3	Creating empty vectors and matrices	36

2.6	Quick recap	38
2.7	Exercise 1	39
3	Reading and writing data to and from R	40
3.1	Importing and reading text files data into RStudio	40
3.2	Importing data using R command read.table()	41
3.2.1	Using read.table() and read.csv()	41
3.3	Exercise 2	44
3.3.1	Importing text files Using scan()	45
3.4	Reading data from Excel into R	46
3.5	Import/Export from other statistical software	47
3.5.1	Reading data from SAS	47
3.5.2	S	48
3.5.3	Stata or Systat	49
3.6	From a Database Connection	49
3.6.1	Parsing each line - Readlines	49
3.7	Writing Data table from R	50
3.7.1	Other considerations when reading or writing data	51
3.8	Exercise 3	53
3.9	Sampling and Creating simulated data	54
3.10	Exercise 4	55
4	Introduction to programming and writing Functions in R	56
4.1	Why do we want to write functions?	56
4.2	Conditional statements (if, ifelse, switch)	57
4.2.1	Repetitive execution: For and While loops	58
4.3	Viewing Code of functions from R packages	59
4.4	The Apply Functions	61
4.5	Exercise 5	64
4.5.1	Efficient For Loop in R (Use apply)	65
4.6	Functions for parsing text	67
4.7	missing values	68
4.8	Exercise 6: Parsing Real Data - World Population Data from Wikipedia	69
4.8.1	Writing functions: More on arguments	70
4.9	Writing functions: more technical discussion -Scoping	71
4.10	Options for Running memory or CPU intensive jobs in R	72
4.10.1	Distributed computing in R	72
4.10.2	Running R in the Cloud	72
4.11	Efficient R coding	73
4.11.1	What is an R script	73
4.11.2	What a script should look like ;-)	73
4.11.3	Coding Recommendations	74
4.11.4	Debugging R Code	75
4.11.5	End-User Messages	75
4.11.6	system.time	75
4.11.7	Etiquette when emailing the R mailing list	76

5	Introduction to graphics in R	79
5.1	The R function plot()	79
5.2	Exercise 7	83
5.2.1	Arguments to plot	84
5.2.2	Other useful basic graphics functions	85
5.3	Editing the default plot with low-level plotting commands	96
5.4	Default parameters - par	99
5.4.1	Interactive plots in R Studio - Effect of changing par	100
5.4.2	R Colors	102
5.5	Interacting with graphics	105
5.5.1	Exercise 8 - Plotting	107
5.6	Saving plots	108
5.6.1	Rstudio	108
5.6.2	Devices	108
5.6.3	Difference between vector and pixel images	108
5.7	Useful Graphics Resources	109
6	Advanced Graphics	110
6.1	Advanced plotting using Trellis; ggplots2, Lattice	110
6.1.1	ggplots2	110
6.1.2	lattice	119
6.1.3	Examples that Present Panels of Scatterplots using xyplot()	120
6.1.4	Simple use of xyplot	120
6.2	GoogleVis and GoogleMaps visualization	131
6.3	Graph theory and Network visualization using R packages network and igraph	133
6.4	Tag Clouds, Literature Mining	137
6.5	Other visualization resources	139
6.6	Summary on plotting	140
7	Statistical Analysis, linear models and survival analysis in R	141
7.1	This section	141
7.2	Basic Statistics	143
7.2.1	Continuous Data: t test	143
7.2.2	adjusting for multiple testing	144
7.2.3	Continuous Data: One- and two-way analysis of variance	145
7.2.4	Discrete Data: Contingency Table	148
7.2.5	Common statistical Tests in R	151
7.3	Model formulae and model options	153
7.3.1	Model formulae	154
7.3.2	Example of linear regression	155
7.3.3	Contrasts, model.matrix	156
7.4	Exercise 9	157
7.5	Output and extraction from fitted models	158
7.6	Exercise 10 :Multivariate linear regression	161
7.6.1	Residual plots, diagnostics	162
7.6.2	ANOVA and updating models	164
7.6.3	Model selection	166
7.7	Cross-validation	167

7.8	Statistical models	169
7.8.1	Linear Regression: Weighted Models, Missing Values	169
7.8.2	Generalized linear modeling	170
7.8.3	Other packages	174
7.9	Survival modeling	175
7.9.1	Censored Data	175
7.9.2	Kaplan-Meier curve estimation	176
7.9.3	Cox proportional hazards model	180
7.10	Exercise 11: Survival Anlaysis	182
8	Solutions to Exercises	183
8.1	Solution to Exercise 1	183
8.2	Solution to Exercise 2	185
8.3	Solution to Exercise 3	186
8.4	Solution to Exercise 4	187
8.5	Solution to Exercise 5	188
8.6	Solution to Exercise 6	188
8.7	Solution to Exercise 7	192
8.8	Solution to Exercise 8	193
8.9	Solution to Exercise 9	195
8.10	Solution to Exercise 10	195
8.11	Solution to Exercise 11	197
8.12	sessionInfo()	201

R set up script for this manual

We will run this course with R>2.15 and RStudio 0.96.

To ensure you have all of the packages needed to run this course, either

1. install course R package (available on the course website)
2. source the following commands into your R session

```
pkgs <- list("lme4", "xtable", "xlsx", "RODBC", "Hmisc", "SAScii",
            "R2HTML", "BiocInstaller", "RColorBrewer", "lattice", "ggplot2",
            "googleVis", "plyr", "network", "igraph", "vcd", "MASS",
            "survival")
Biocpkgs <- c("parallel")

checkPkg <- function(x) {
  print(x)
  if (!require(x, character.only = TRUE))
    install.packages(x)
}

checkBioCPkg <- function(x) {
  print(x)
  if (!require(x, character.only = TRUE))
    biocLite(x)
}

lapply(pkgs, checkPkg)
lapply(Biocpkgs, checkBioCPkg)

search()
```

Chapter 1

Introduction to R Software

1.1 Obtaining R and RStudio

R can be downloaded from the website: <http://cran.r-project.org/>. See additional online notes which give a very detailed description on downloading and installing R (and Bioconductor). R is available for all platforms: Unix/Linux, Windows and Mac.

When you download R, its default editor and console are relatively basic (we describe it below). In this course we will call R using RStudio which provides a richer interface and nicer editor of R code. RStudio can be obtained from www.rstudio.org). RStudio is available as a desktop program for all platforms Unix/Linux, Windows and Mac.

If you don't want to install R on your own machine, or tie up your own machine, with a memory or CPU intensive task. You can run R through web server or even in the Cloud. Its really simple to run R jobs in the Amazon cloud as described in Section 4.10 and these can be run through the RStudio interface, which looks exactly as it does on the desktop version.

In this course, we will concentrate on the Windows implementation of R. The differences between the platforms are minor, so most of the material (R, Rstudio) is applicable to the other platforms.

1.2 The default R interface

When you click on the R icon, you will open the basic R interface

- On the menu bar, there are the menus:
 - File - load script, load/save session (workspace) or command history. **Change Directory**
 - Edit - Cut/Paste. GUI preferences
 - View
 - Misc - stop computations, list/remove objects in session
 - Packages - allows one to install new, update packages
 - Windows
 - Help - **An essential resource!**
- The *icons* below will allow one to
 - open script (.R file),

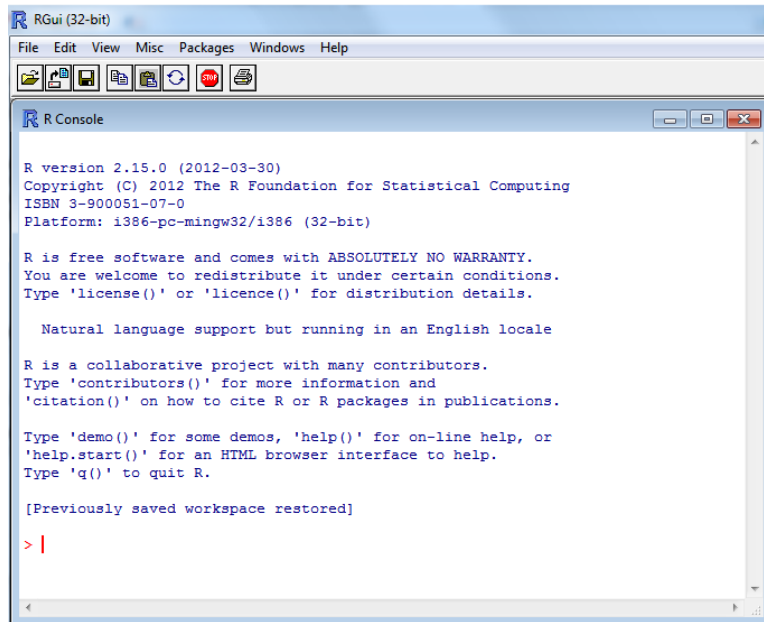


Figure 1.1: R interface with windows installation of R

- load image (previous R session, .RData file)
- save .R script
- copy and paste
- stop a computation (this can be an important button, but the ESC also works!)
- print.

1.2.1 Default R Console

The main window that you see in the R Console. This is where you type R commands. Note it says type `demo()` or `help()` or `q()` to view example R in code, to see R help or to quit R respectively.

Within the R console, you can

- Type at the prompt `>` symbol.
- If the prompt instead show the continuation prompt is `'+'`, it means the command you typed is incomplete and R is waiting for it to complete (usually its missing a closing quote or bracket)
- Use up and down arrow keys to scroll through previous commands. This is useful if you would like to repeat a previous command
- R also includes basic automatic completions for function names and filenames. Type the "tab" key to see a list of possible completions for a function or filenames.

1.2.2 Default R Editor

- Within R, one can open an editor using the menu command *File -> New script*
- Can type commands in the editor

- Highlight the commands and type `Ctrl^R` to submit the commands for R evaluation
- Evaluation of the commands can be stopped by pressing the `Esc` key or the `Stop` button
- Saving (`Ctrl^S`) and opening saved commands (`Ctrl^O`)
- Can open a new editor by typing `Ctrl^N`

1.2.3 R Shortcuts

Keyboard Shortcuts for traditional R GUI

- `Ctrl + A`: gets you to the beginning of the line
- `Ctrl + E`: gets you to the end
- `Ctrl + K`: wipes everything to the right of your cursor
- `Esc`: kills the current comment and takes you back to `;`
- `Up Arrow`: brings back last command you typed
- `Down Arrow`: brings back the next command (Up and down basically scrolls up and down through your history)

Keyboard shortcuts for RStudio are listed online at http://www.rstudio.org/docs/using/keyboard_shortcuts.

1.3 RStudio Interface

RStudio is a free and open source integrated development environment for R. Those familiar with matlab will recognize the layout as its pretty similar. RStudio have a brief 2 minute guide to the interface on their website <http://rstudio.org/> which I recommend watching.

On startup R Studio brings up a window with 3 or 4 panels. If you only see 3 panels, click on *File* -> *New* -> *New R Script*.

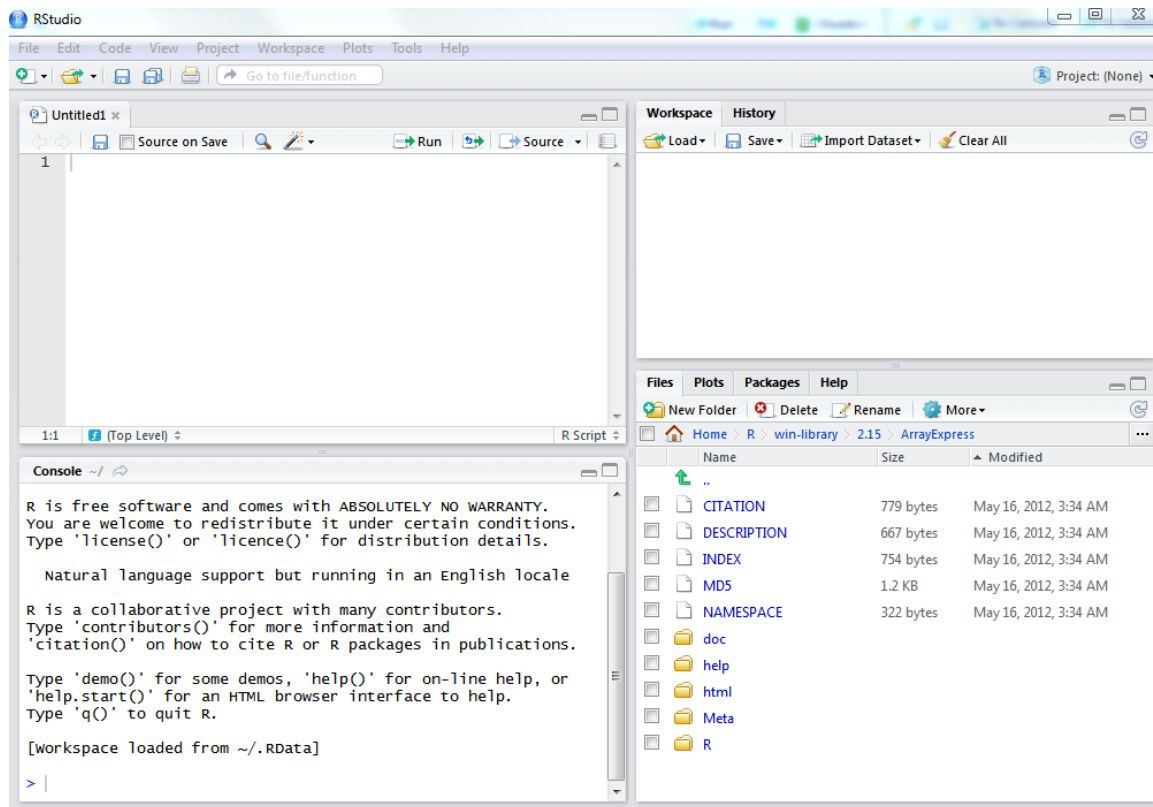


Figure 1.2: RStudio v0.96 interface with 4 panels

The first thing to notice, is that the bottom left panel "console" is the exact same as the standard R console. RStudio just loads your local version of R. You can specify a different version of R (if you have multiple versions of R running on your machine) by clicking on *Tools* -> *Options* and selecting R version.

1.3.1 console

RStudio has a nice console features

- start typing a command, for example `fi`, press the TAB key, it will suggest function that begin with `fi`
- Select `fis` for `fisher.test`. Type

```
fisher.test(
```

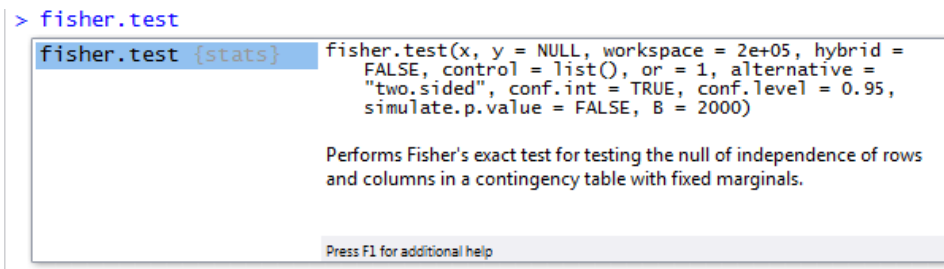


Figure 1.3: Tab not only auto-completes, but also suggests parameters and input to the function. Note it says press F1 for further help on this function

and then press the TAB key. You will notice it bring up help on each parameter, you can browse these and it will help you get familiar with R.

- Press **F1**, it will bring up a **help** document about the function in the help panel (right bottom)
- Press **F2**, it will show the **source code** for the function

There are many useful keyboard shortcuts in RStudio for a full list of these see http://rstudio.org/docs/using/keyboard_shortcuts

1.3.2 Editor

The top left panel is an editor which can be used to edit R scripts (.R), plain text (.txt), html web files or Sweave (rnw) or markdown (md), the latter two of these can be converted to pdf files. There are several nice features to this text editor which we will describe during the course. But for now note, that it highlights R code, and that the code is searchable (click Control-F to search)

In the menu **code**, you can set preferences to highlight, indent or edit code.

1.3.3 Workspace, history

On the top right there is a tab menu workspace and history. We will talk about these in detail.

- Workspace list the objects in the current R session. You can load, save or "Clear All" object for a workspace
- Note that under the workspace panel there is the option to **Import Dataset**
- The history panel lists all of the command that have been typed or input in the console. There are options to load, save, search or delete history
- One can easily repeat a command by highlighting one or more line(s) and sending these **To Console**
- One can easily copy a command to a new R script or text file, by highlighting one or more line(s) and sending these **To Source**

1.3.4 Files, Plots, Packages, Help

On the bottom right there is a tab menu Files, Plots, Packages and Help.

- **Files** is a file browser, which allows you to create a new folder, rename a folder or delete a folder. Click on the triple dot icon (...) on the far right on the menu to browse folders. Under the More menu you can set your current working directory (more about that below). If you double click on a text, .R, Sweave or html file it will automatically open in the Editor.
- The **Plots** window displays plots generated in R. Simply type the following command into the Console window

```
plot(1:10)
plot(rnorm(10), 1:10)
```

It will create 2 plots. Use the arrows keys to browse plots, click on zoom, export or delete to manage plots.

- **Packages** list all of the packages installed in your computer. The packages with tick marked are those loaded in your current R session. Click on a package name to view help on that package. Note you can **install packages or check for updates**. You can also search for a package or search package descriptions using the search window.
- The **Help** menu provides an extensive R help. The arrows button go forward or back through recent help pages you have viewed. You can go home (house icon), print or open help in new window. You can search help, use the search window. Help can also be browsed through main menu bar at the open of the page

1.3.5 Projects, SVN in RStudio

RStudio provides an easy approach to managing projects. In the main menu there is a **Projects** menu which will create folder for your project and retain all data files and command history for your project. It is also possible to set up a backup subversion management control system for your code as RStudio will directly communicate with your SVN or github account.

1.4 Starting out - setting a working directory

Setting up a RStudio project folder is a great way to start a new project. It aids with managing R code, data and results. However if you don't use the project option or are accessing R through the default or another interface, you will need to be able to set a current working directory (in which all your output files are saved).

The first thing to do when starting an R session, is to ensure that you will be able to find your data and also that your output will be saved to a useful location on your computer hard-drive. Therefore, set a "working directory"

This maybe set by default to the depths of the operating system (C:/program files/R), which is a poor "working" location.

There are numerous way to set the working directory. To change directory:

1. In the classic R interface. Use the file menu, to change directory File – > Change dir

2. If you start R by clicking on an R icon. You may wish to change the default start location by right mouse clicking on the R icon on the desktop/start menu and changing the "Start In" property. For example make a folder 'C:/work', and make this as a "Start in" folder
3. In RStudio Tools – > Set Working Directory
4. Or in RStudio click on the Files tab (on the bottom right panel). Use the File browser window to view the contents of a directory and navigate to the directory you wish to set as your home directory.
 - click on the triple dot icon on the top right. Navigate to the correct directory
 - Once you are in the correct directory and see your data files click on the More (blue cogwheel), and select "Set as Working Directory"

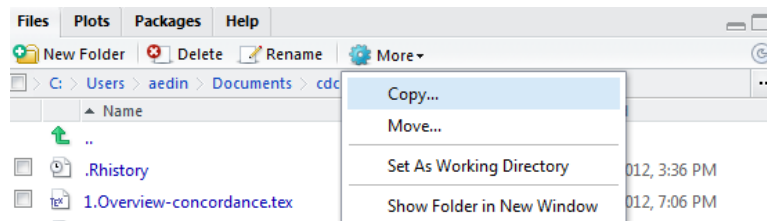


Figure 1.4: Note the triple dot icon on the far right and the blue cog wheel

5. The commands to set the working directory.

```
# What is my current directory
getwd()
```

To change the directory:

```
# Set working directory
setwd("C:/work")
```

To see folders or files in the working directory, use the command *dir()* (or browse the files using the Files Browser panel in RStudio)

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

1.4.1 Managing and accessing folders in R

I can create a new working directory for an R session for example, I try to include a project Name Date in my folder names eg colonJan13.

```
# check if the directory or file exists
file.exists("colonJan13")
# Create a directoryfolder
dir.create("colonJan13")
setwd("colonJan13")
```

To create a full directory or more complex directory path. A path can be relative to the current location, in this case two dots mean "the directory above"

```
setwd("../../RWork/colonJan13")
```

Or you can specify a full directory path. For cross-platform compatibility, it's best to use `file.path()` to create paths. for example

```
wkdir <- getwd()
newdir <- "MyNewDir"
subdir <- "Project1"
newdirPath <- file.path(wkdir, newdir, subdir)

if (file.exists(wkdir)) dir.create(newdirPath)
dir(pattern = "My")
```

Important side note: R doesn't like windows a back slash (\) that separate folders in a file path. Indeed it will return a rather cryptic error

```
> setwd("C:\\Users\\aedin\\Documents\\Rwork\\colonJan13")
Error: "\U" used without hex digits
in character string starting "C:\\U"
```

There are a couple of ways to prevent this, either replace backslash (\) with forward (/) slash or double back slash (\\). The first back slash tells R to treat the character literally, it is called an escape character which invokes an alternative interpretation on subsequent characters.

```
setwd("C:/Users/aedin/Documents/Rwork/colonJan13")
setwd("C:\\Users\\aedin\\Dropbox\\Talks\\CDC\\Notes")
```

This can be rather tedious. A nice way to make scripts work across platform is to use the command `path.expand`, `file.path`. The tilde symbol (~) is a shortcut to your home drive (on any operating system)

```
path.expand("~")
myhome <- path.expand("~/")
newdir <- file.path(path.expand("~/"), "Rwork", "colonJan13")
setwd(newdir)
```

In my case, I work on several windows boxes, each have fairly different directory structures, but by setting the home directory properly I can sync code between computers and have them run properly on each one since where I run my R projects have similar directory structures.

Advanced Side Note: What is your system home directory

Your system HOME (~) is set by your operating system. To view or change it, type

```
Sys.getenv("HOME")

## [1] "C:/Users/aedin/Documents"
```

```
Sys.setenv(HOME = "path") # 'path' is the directory you wish to set as  
your new home directory
```

1.5 First R Encounter

When you start R, and see the prompt `>`, it may appear as if nothing is happening. But this prompt is now awaiting commands from you. This can be daunting to the new user, however it is easy to learn a few commands and get started

```
demo()
```

Note we place **round brackets** after all functions ALWAYS. Note if the command is not complete on one line (missing close bracket or quote), R will use continuation prompt is `'+'`.

If the brackets are empty `()` it runs the function with default parameters. To specify a parameters these are inserted into the brackets. For example to run a demo on a specific topics we give that parameters to demo

```
demo(graphics)
demo(nlm)
```

Type `q()` to quit the program

To find out more about parameters and options for the function demo, lets look at help for demo.

1.6 Getting help with functions and features

There are many resources for help in R. Within R, you can find help on any command (or find commands) using the follow:

- If you know the command (or part of it)

```
## help or ? will open a help page
help(lm)

# ?matrix
`?`(matrix)

## The functions find() or apropos() will search for a command
apropos("mean")

## The function example will run example of a command
example(rep)
```

The last command will run all the examples included with the help for a particular function. If we want to run particular examples, we can highlight the commands in the help window and submit them by typing `Ctrl^V`

- If you don't know the command and want to do a keyword search for it.

```
help.search("combination")
help.start()
```


`help.search` will open a html web browser or a MSWindows help browser (depending on the your preferences) in which you can browse and search R documentation.

- Finally, there is a large R community who are incredibly helpful. There is a mailing list for R, Bioconductor and almost every R project. It is useful to search the archives of these mailing lists. Frequently you will find someone encountered the same problem as you, and previously asked the R mailing list for help (and got a solution!).
 - The R search engine <http://www.Rseek.org>
 - R bloggers website <http://www.r-bloggers.com/>
- There numerous useful resources for learning R on the web including the R project <http://www.r-project.org> and its mailing lists but also I recommend the following:
 - Emmanuel Paradis has an excellent beginners guide to R available from http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf
 - There is an introduction to R classes and objects on the R website <http://cran.r-project.org/doc/manuals/R-intro.html>
 - also see Tom Guirkes manual at http://manuals.bioinformatics.ucr.edu/home/R_BioCondManual
 - Tom Short's provides a useful short R reference card at <http://cran.r-project.org/doc/contrib/Short-refcard.pdf>
- Coming from SAS or SPSS background?
 - In the December 2009 issue of the R Journal. Transitioning to R: Replicating SAS, Stata, and SUDAAN Analysis Techniques in Health Policy Data. Anthony Damico http://journal.r-project.org/archive/2009-2/RJournal_2009-2_Damico.pdf
 - SAS and R. A blog devoted to examples of tasks (and their code) replicated in SAS and R <http://sas-and-r.blogspot.com/>
 - R for SAS and SPSS Users. Download a free 80 page document, <http://rforsasandspssusers.com/>
R for SAS and SPSS Users which contains over 30 programs written in all three languages.

1.7 R as a big calculator

Type the following into an R session (or copy and paste from this document).

```
2 + 2

## [1] 4

2 * 2

## [1] 4

2 * 100/4
```

```
## [1] 50

2 * 100/4 + 2

## [1] 52

2 * 100/(4 + 2)

## [1] 33.33

2^10

## [1] 1024

log(2)

## [1] 0.6931
```

Note even in the simple use of R as a calculator, it is useful to store intermediate results. For example lets store the value of `(tmpVal=log(2))`.

```
tmpVal <- log(2)
print(tmpVal)

## [1] 0.6931

tmpVal

## [1] 0.6931

exp(tmpVal)

## [1] 2
```

In this case, we assigned a *symbolic variable* `tmpVal`. Note when you assign a value to such a variable, there is no immediate visible result. We need to `print(tmpVal)` or just type `tmpVal` in order to see what value was assigned to `tmpVal`

1.8 A few important points on R

- Elementary commands: *expressions* are evaluated, printed and value lost; *assignments* evaluate expressions, passes value to a variable, but not automatically printed

```
2 * 5^2

## [1] 50

x <- 2 * 5^2
print(x)
```

```
## [1] 50
```

- Assignment operators are: `<-`, `=`, `->`

```
2 * 5^2
```

```
## [1] 50
```

```
y <- 2 * 5^2
```

```
z <- 2 * 5^2
```

```
z <- 2 * 5^2
```

```
print(y)
```

```
## [1] 50
```

```
x == y
```

```
## [1] TRUE
```

```
y == z
```

```
## [1] TRUE
```

- `<-` is the most popular assignment operator, and `=` is a recent addition.

There is no space between `<` and `-`

It is `<-` (less than and a minus symbol)

When assigning a value spaces are ignored so `z<-3` is equivalent to `z <- 3`

- R is case sensitive, i.e. `myData` and `Mydata` are different names

```
x
```

```
## [1] 50
```

```
Y
```

```
## [1] 50
```

```
Z <- 20
```

```
x == z
```

```
## [1] TRUE
```

```
x == Z
```

```
## [1] FALSE
```

- '==' and '=' have very different uses in R. == is a binary operator, which test for equality (A==B determines if A 'is equal to' B).
- Missing values called represented by NA
- Comments can be put anywhere. To comment text, insert a hashmark #. Everything following it to end of the line is commented out (ignored, not evaluated).

```
print(y) # Here is a comment
```

- Quotes, you can use both " double or ' single quotes, as long as they are matched.
- For names, normally all alphanumeric symbols are allowed plus . and _ Start names with a character [Aa-Zz] not a numeric character [0-9]. Avoid using single characters or function names t, c, q, diff, mean, plot etc.
- Arguments (parameters) to a function calls f(x), PROC are enclosed in round brackets. Even if no arguments are passed to a function, the round brackets are required.

```
print(x)
getwd()
```

- Commands can be grouped together with braces ({ and }).
- Note on brackets. It is very important to use the correct brackets.

Bracket	Use
()	To set priorities 3*(2+4). Function calls f(x)
[]	Indexing in vectors, matrices, data frames
{ }	Creating new functions. Grouping commands {mean(x); var(x)}
[[]]	Indexing of lists

1.8.1 Use Arrows to key browse command history

Note you can recover or browser previous commands using the up and down arrow keys.

For example `rnorm` generates 10 random numbers from a normal distribution. Type this a few times (hint: the up arrow key is useful).

```
rnorm(5)
```

If you wish to generate the same set of random numbers each time, you could `set.seed(10)`
You can view previous expressions entered into the R session (default 25) using the function

```
history()
```

(this is discussed in more detail later on) You can also view the history of R commands in the **history** tab on top right panel in RStudio

1.9 Basic operators

We already saw that `==` tests for equality or a match between 2 objects. Other operators are:

1.9.1 Comparison operators

- equal: `==`
- not equal: `!=`
- greater/less than: `>` `<`
- greater/less than or equal: `>=` `<=`

```
1 == 1

## [1] TRUE
```

1.9.2 Logical operators

- AND & Returns TRUE if both comparisons return TRUE.

```
x <- 1:10
y <- 10:1
x > y & x > 5

## [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

- OR — Returns TRUE where at least one comparison returns TRUE.

```
x == y | x != y

## [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

- NOT ! The '!' sign returns the negation (opposite) of a logical vector.

```
!x > y

## [1] TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE
```

These return a logical vector of TRUE or FALSE and can be useful for filtering (we will see this later)

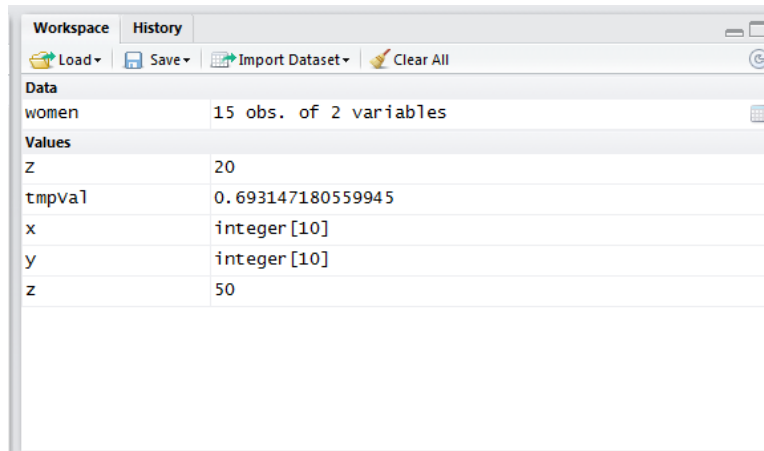
1.10 Data sets in R

Both the R core installation and contributed R package contain data sets, which are useful example data when learning R. To list all available data sets:

```
data()
```

To load a dataset, for example, the dataset `women` which gives the average heights and weights for 15 American women aged 30-39.

```
data(women)
ls()
ls(pattern = "w")
```



The screenshot shows the R Studio Workspace window. At the top, there are tabs for 'Workspace' and 'History'. Below the tabs is a toolbar with icons for 'Load', 'Save', 'Import Dataset', and 'Clear All'. The main area is divided into two sections: 'Data' and 'Values'. The 'Data' section lists the 'women' dataset with the description '15 obs. of 2 variables' and a table icon. The 'Values' section lists several objects: 'Z' with the value '20', 'tmpval' with the value '0.693147180559945', 'x' with the value 'integer [10]', 'y' with the value 'integer [10]', and 'z' with the value '50'.

Data	
women	15 obs. of 2 variables

Values	
Z	20
tmpval	0.693147180559945
x	integer [10]
y	integer [10]
z	50

Figure 1.5: The Workspace window lists the object currently in the R workspace. You can click on each item to view or edit it. Note `women` is a data table with dimensions 15 rows x 2 columns, you can click on the table icon to view it. `Z` and `y` are a single value (50). `x` and `y` are integer vectors of length 10.

1.11 R sessions (workspace) and saving session history

To finish up today, we will save our R session and history

1. R session One can either save one or more R object in a list to a file using `save()` or save the entire R session (workspace) using `save.image()`.

```
save(women, file = "women.RData")
save.image(file = "entireL2session.RData")
```

To load this into R, start a new R session and use the `load()`

```
rm(women)
ls(pattern = "women")
load("women.RData")
ls(pattern = "women")
```

2. R history R records the commands history in an R session. To view most recent R commands in a session

```
history()
help(history)
history(100)
```

To search for a particular command, for example "save"

```
history(pattern = "save")
```

To save the commands in an R session to a file, use `savehistory()`

```
savehistory(file = "L2.Rhistory")
```

3. Note you can also browse and search history in RStudio, really easily in the history window. A really nice feature of this window is the ease of sending command either to the console (to execute code again) or to Source (to a text file or script you are writing in the editor)

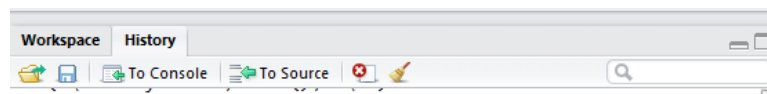


Figure 1.6: You can easily save or search command history, send commands to the R console or a source (script) file

4. Default saving of RData and Rhistory

By default, when you quit `q()` an R session, it will ask if you wish to save the R workspace image. If you select yes, it will create two file in the current working directory, there are `.RData` and `.Rhistory`. These are hidden system files, unless you choose to "Show Hidden Files" in the folder options. There are output files are the same as running `save.image(file=".RData")` and `savehistory(file=".Rhistory")` respectively.

1.12 R Packages

By default, R is packaged with a small number of essential packages, however as we saw there are many contributed R packages.

1. Some packages are loaded by default with every R session. The libraries included in the Table are loaded on the R startup.

Table 1.1: Preloaded packages

Package	Description
base	Base R functions
datasets	Base R datasets
grDevices	Graphics devices for base and grid graphics
graphics	R functions for base graphics
methods	Formally defined methods and classes for R objects, plus other programming tools
stats	R statistical functions.
utils	R utility functions

2. To see which packages are currently loaded, use

```
search()  
sessionInfo()
```

3. To see which packages are installed on your computer, issue the command

```
library()
```

Within **RStudio** installed packages can be view in the **Package Tab of the lower right panel**. You can tick or select a library to load it in R.

You will very likely want to install additional packages or libraries.

1.13 Installing new R libraries

There are several thousand R packages and >500 Bioconductor packages (also called libraries) available. These are not installed by default, so we have to select and install additional packages that will be of use to us. Not all of them, actually a small subset, will be useful to us. R users are free to selected which libraries to install.

On windows, sometimes I have encountered problems with program installation using RStudio. This is normally when R tries to write to a folder and doesn't have write permission to this folder. In this case, either

1. Run RStudio as administrator

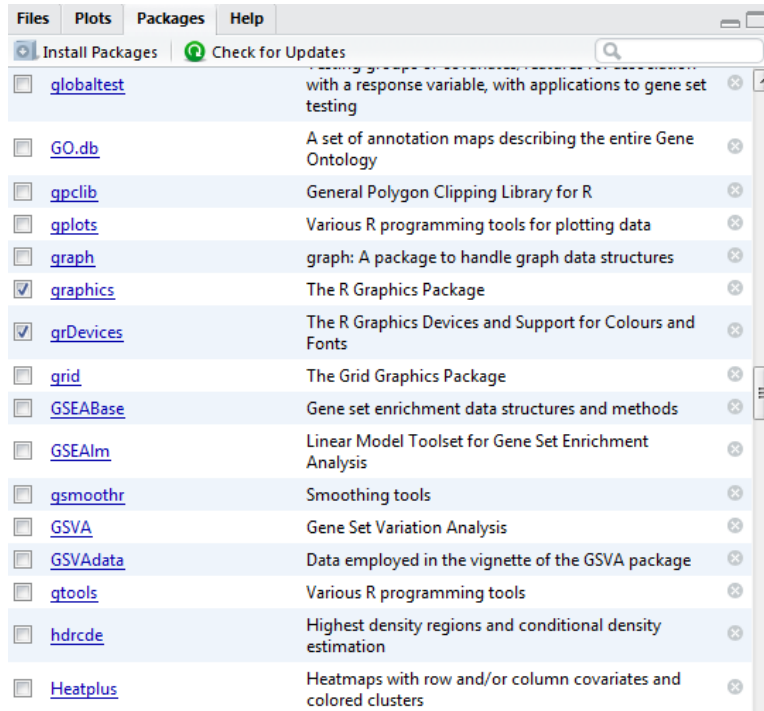


Figure 1.7: The list is all installed packages. A tick marked indicates its loaded into the current R session. Clicking on the package name will open help for that package

2. check you have write permission to the path where it will write the files (use the following R function to list your RStudio Paths) `;;rsPref, eval=FALSE;=.rs.defaultUserLibraryPath()` Path where RStudio installs packages `.rs.rpc.get_package_install_context()` Other useful RStudio configuration info
3. Install packages using the basic R GUI using the drop-down menu `Packages` or command line (`install.packages`). First Click on “Packages” and “Set CRAN mirror” and choose an available mirror (choose one close by, it’ll be faster hopefully). Then If you know the name of the package you want to install, or if you want to install all the available packages, click on “install Packages”
4. Open a CMD shell and type R CMD INSTALL packageName.tar.gz. You can open a CMD shell from RStudio `Tools -> Shell`

Installing packages using RStudio

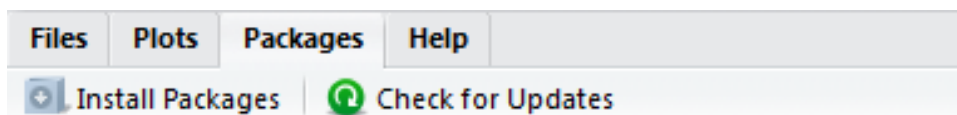


Figure 1.8: Click on Install Packages

This will open an install window:
Commands to install packages

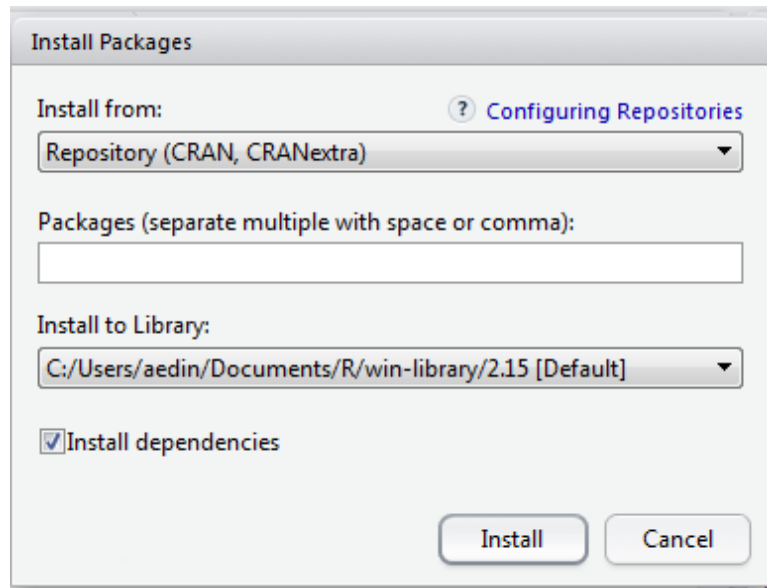


Figure 1.9: Type the name of the package. Unfortunately RStudio does not display a list of all available packages

```
# Installing and updating R libraries
install.packages("lme4")
update.packages("lme4")
```

Installation of all packages takes some time and space on your computer. If the name of the package is not known, you could use taskviews help or archives of the mailing list to pinpoint one. Also look on the R website Task views description of packages (see Additional Notes in Installation which I have provided).

To get an information on a package, type

```
library(help = lme4)
```

Once you have installed a package, you do NOT need to re-install it. But to load the library in your current R session use the commands

```
library(lme4) # Load a package
## Or the alternative
require(lme4)

sessionInfo() #List all packages loaded in the current R session
library() # List all installed packages
```

You can unload the loaded package pkg by

```
search()
detach(package:lme4)
search()
```

NOTE: Packages are often inter-dependent, and loading one may cause others to be automatically loaded.

1.14 Customizing Startup

To customize your R environment, you can change options through RStudio File -> Tools -> Options

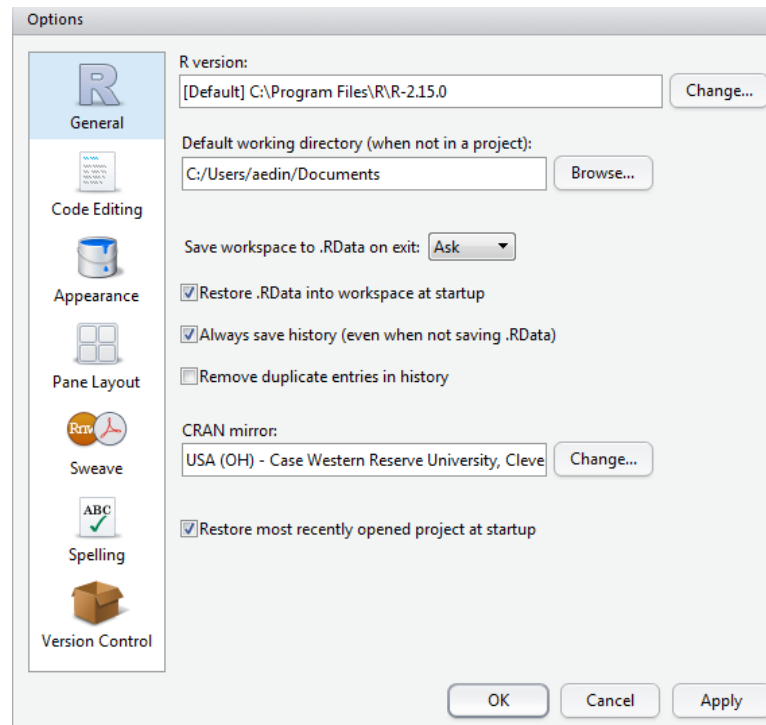


Figure 1.10: Within the options window you can change the version of R, your default directory and SVN preferences

In addition you can specify preferences for either a site or local installation in `Rprofile.site`. On Windows, this file is in the `C:\Program Files\R\R-x.y.z\etc` directory where `x.y.z` is the version of R. Or you can also place a `.Rprofile` file in the directory that you are going to run R from or in the user home directory.

At startup, R will source the `Rprofile.site` file. It will then look for a `.Rprofile` file in the current working directory. If it doesn't find it, it will look for one in the user's home directory.

There are two special functions you can place in these files. `.First()` will be run at the start of the R session and `.Last()` will be run at the end of the session. These can be used to load a set of libraries that you use most.

```
# Sample Rprofile.site file

# Things you might want to change options(papersize='a4')
# options(editor='notepad') options(pager='internal')

# R interactive prompt options(prompt='> ') options(continue='+ ')
```

```

# General options
options(tab.width = 2)
options(width = 100)
options(digits = 5)

.First <- function() {
  library(lme4)
  library(design)
  library(xtable)
  cat("\nWelcome at", date(), "\n")
}

.Last <- function() {
  cat("\nGoodbye at ", date(), "\n")
}

```

For more help on this see <http://www.statmethods.net/interface/customizing.html>

Chapter 2

Objects in R

- Everything (variable, functions etc) in R is an *object*
- Every object has a *class*

2.1 Using ls and rm to managing R Objects

R creates and manipulates *objects*: variables, matrices, strings, functions, etc. *objects* are stored by name during an R session.

During a R session, you may create many objects, if you wish to list the objects you have created in the current session use the command

```
objects()  
ls()
```

The collection of objects is called *workspace*.

If you wish to delete (remove) objects, issue the commands:

```
rm(x, y, z, junk)
```

where `x`, `y`, `z`, `junk` were the objects created during the session.

Note `rm(list=ls())` will remove everything. Use with caution

2.2 Types of R objects

Objects can be thought of as a container which holds data or a function. The most basic form of data is a single element, such as a single numeric or a character string. However one can't do statistics on single numbers! Therefore there are many other objects in R.

- A `vector` is an ordered collection of numerical, character, complex or logical objects. Vectors are collection of *atomic* (same data type) components or modes. For example

```
# Numeric  
vec1 <- 1:10  
vec1
```

```
## [1] 1 2 3 4 5 6 7 8 9 10

# Character
vec2 <- LETTERS[1:10]
vec2

## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"

# logical
vec3 <- vec2 == "D"
vec3

## [1] FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
```

In each case above, these vectors have 10 elements, and are of length=10.

- A *matrix* is a multidimensional collection of data entries of the same type. Matrices have two dimensions. It has rownames and colnames.

```
mat1 <- matrix(vec1, ncol = 2, nrow = 5)
print(mat1)

##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10

dim(mat1)

## [1] 5 2

colnames(mat1) <- c("A", "B")
rownames(mat1) <- paste("N", 1:5, sep = "")
print(mat1)

##      A  B
## N1 1  6
## N2 2  7
## N3 3  8
## N4 4  9
## N5 5 10
```

- A `list` is an ordered collection of objects that can be of different modes (e.g. numeric vector, array, etc.).

```
a <- 20
newList1 <- list(a, vec1, mat1)
print(newList1)

## [[1]]
## [1] 20
##
## [[2]]
## [1] 1 2 3 4 5 6 7 8 9 10
##
## [[3]]
##      A  B
## N1 1  6
## N2 2  7
## N3 3  8
## N4 4  9
## N5 5 10
##

newList1 <- list(a = a, myVec = vec1, mat = mat1)
print(newList1)

## $a
## [1] 20
##
## $myVec
## [1] 1 2 3 4 5 6 7 8 9 10
##
## $mat
##      A  B
## N1 1  6
## N2 2  7
## N3 3  8
## N4 4  9
## N5 5 10
##
```

- Though a `data.frame` is a restricted list with class `data.frame`, it may be regarded as a matrix with columns that can be of different modes. It is displayed in matrix form, rows by columns. (Its like an excel spreadsheet)

```
df1 <- as.data.frame(mat1)
df1
```

```
##      A  B
## N1 1  6
## N2 2  7
## N3 3  8
## N4 4  9
## N5 5 10
```

- A factor is a vector of categorical variables, it can be ordered or unordered.

```
charVec <- rep(LETTERS[1:3], 10)
print(charVec)

## [1] "A" "B" "C" "A" "B" "C" "A" "B" "C" "A" "B" "C" "A" "B" "C" "A"
## [17] "B" "C" "A" "B" "C" "A" "B" "C" "A" "B" "C" "A" "B" "C"

table(charVec) # Tabulate charVec

## charVec
##  A  B  C
## 10 10 10

fac1 <- factor(charVec)
print(fac1)

## [1] A B C A B C A B C A B C A B C A B C A B C A B C A B C
## Levels: A B C

attributes(fac1)

## $levels
## [1] "A" "B" "C"
##
## $class
## [1] "factor"
##

levels(fac1)

## [1] "A" "B" "C"
```

- array An array in R can have one, two or more dimensions. I find it useful to store multiple related data.frame (for example when I jack-knife or permute data). Note if there are insufficient objects to fill the array, R recycles (see below)


```
array(1:24, dim = c(2, 4, 3))
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    9   11   13   15
## [2,]   10   12   14   16
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
## [1,]   17   19   21   23
## [2,]   18   20   22   24
##
```

```
array(1:23, dim = c(2, 4, 3))
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    9   11   13   15
## [2,]   10   12   14   16
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
## [1,]   17   19   21   23
## [2,]   18   20   22    1
##
```

```
array(1:23, dim = c(2, 4, 3), dimnames = list(paste("Patient",
  1:2, sep = ""), LETTERS[1:4], c("X", "Y", "Z")))
```

```
## , , X
##
```

```
##           A B C D
## Patient1 1 3 5 7
## Patient2 2 4 6 8
##
## , , Y
##
##           A B C D
## Patient1 9 11 13 15
## Patient2 10 12 14 16
##
## , , Z
##
##           A B C D
## Patient1 17 19 21 23
## Patient2 18 20 22 1
##
```

2.3 Attributes of R Objects

1. Basic attributes

The most basic and fundamental properties of every object is its `mode` and `length`. These are intrinsic attributes of every object. Examples of `mode` are "logical", "numeric", "character", "list", "expression", "name/symbol" and "function".

Of which the most basic of these are:

- `character`: a character string
- `numeric`: a real number, which can be an integer or a double
- `integer`: an integer
- `logical`: a logical (true/false) value

```
# Numeric
x <- 3
mode(x)

## [1] "numeric"

# Character
x <- "apple"
mode(x)

## [1] "character"

x <- 3.145
x + 2 # 5.145
```

```
## [1] 5.145

# FALSE, logical
x == 2

## [1] FALSE

x <- x == 2
x

## [1] FALSE

mode(x)

## [1] "logical"
```

All R objects have mode

```
# vectors of different modes Numeric
x <- 1:10
mode(x)

## [1] "numeric"

x <- matrix(rnorm(50), nrow = 5, ncol = 10)
mode(x)

## [1] "numeric"

# Character
x <- LETTERS[1:5]
mode(x)

## [1] "character"

# logical
x <- x == "D"
mode(x)

## [1] "logical"
```

Quick Exercise

Repeat above, and find the `length` and `class` of `x` in each case.

2. Other attributes, dimension

```
x <- matrix(5:14, nrow = 2, ncol = 5)
x

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    5    7    9   11   13
## [2,]    6    8   10   12   14

attributes(x)

## $dim
## [1] 2 5
##
```

In summary

Object	Modes	Allow >1 Modes*
vector	numeric, character, complex or logical	No
matrix	numeric, character, complex or logical	No
list	numeric, character, complex, logical, function, expression, ...	Yes
data frame	numeric, character, complex or logical	Yes
factor	numeric or character	No
array	numeric, character, complex or logical	No

*Whether object allows elements of different modes. For example all elements in a vector or array have to be of the same mode. Whereas a list can contain any type of object including a list.

2.4 Creating and accessing objects

We have already created a few objects: `x`, `y`, `junk`. Will create a few more and will select, access and modify subsets of them.

- Create vectors, matrices and data frames using `seq`, `rep`, `rbind` and `cbind`

```
# Vector
x.vec <- seq(1, 7, by = 2)

# The function seq is very useful, have a look at the help on seq
# (hint ?seq)

names(x.vec) <- letters[1:4]
# Matrices
```

```
xMat <- cbind(x.vec, rnorm(4), rep(5, 4))
yMat <- rbind(1:3, rep(1, 3))
z.mat <- rbind(xMat, yMat)
# Data frame
x.df <- as.data.frame(xMat)
names(x.df) <- c("ind", "random", "score")
```

- Accessing elements

NOTE Use square brackets to access elements. The number of elements within the square bracket must equal the dimension of the object.

1. vector [1]
2. matrix [1,1]
3. array with 3 dimensions [1,1,1]

```
# Access first element of 'x.vec'
x.vec[1]

## a
## 1

# or if you know the name
x.vec["a"]

## a
## 1

# Access an element of 'xMat' in the second row, third column
xMat[2, 3]

## [1] 5

# Display the second and third columns of matrix 'xMat'
xMat[, c(2:3)]

##
## a -0.1267 5
## b -1.2919 5
## c  0.8820 5
## d -0.6514 5

# or
xMat[, -c(1)]

##
## a -0.1267 5
## b -1.2919 5
## c  0.8820 5
## d -0.6514 5
```

Here -1 means everything except for the first column.

Quick Exercise What does this command do?

```
xMat[xMat[, 1] > 3, ]

##      x.vec
## c      5  0.8820 5
## d      7 -0.6514 5
```

If the object has class `data.frame` or `list`, you can use the dollar symbol `$` to access elements. The `$` can only access columns of `data.frame`

```
# Get the vector of 'ind' from 'x.df'
colnames(x.df)

## [1] "ind"      "random" "score"

x.df$ind

## [1] 1 3 5 7

x.df[, 1]

## [1] 1 3 5 7

names(newList1)

## [1] "a"          "myVec" "mat"

newList1$a

## [1] 20
```

2.5 Modifying elements

```
# Change the element of 'xMat' in the third row and first column to
# '6'
xMat[3, 1] <- 6
# Replace the second column of 'z.mat' by 0's
z.mat[, 2] <- 0
```

2.5.1 Sorting and Ordering items

Frequently we need to re-order the rows/columns of a matrix or see the rank order or a sorted set elements of a vector

The functions *sort* and *order* are designed to be applied on vectors. Sort returns a sorted vector. Order returns an index which can be used to sort a vector or matrix.

```
# Simplest 'sort'
z.vec <- c(5, 3, 8, 2, 3.2)
sort(z.vec)

## [1] 2.0 3.0 3.2 5.0 8.0

order(z.vec)

## [1] 4 2 5 1 3
```

Sorting the rows of a matrix. We will use an example dataset in R called ChickWeight. First have a look at the ChickWeight documentation (help)

Lets take a subset of the matrix, say the first 36 rows.

```
# ?ChickWeight
ChickWeight[1:2, ]

##   weight Time Chick Diet
## 1     42    0     1    1
## 2     51    2     1    1

chick.short <- ChickWeight[1:36, ]
```

Now order this matrix by time and weight

```
## by just weight
chickOrd <- chick.short[order(chick.short$weight), ]
chickOrd[1:5, ]

##   weight Time Chick Diet
## 26     39    2     3    1
## 13     40    0     2    1
## 1      42    0     1    1
## 25     43    0     3    1
## 14     49    2     2    1

## By both time and weight
chick.srt <- chick.short[order(chick.short$Time, chick.short$weight),
  ]
chick.srt[1:5, ]
```

##		weight	Time	Chick	Diet
##	13	40	0	2	1
##	1	42	0	1	1
##	25	43	0	3	1
##	26	39	2	3	1
##	14	49	2	2	1

2.5.2 Missing Values

Missing values are assigned special value of 'NA'

```
z <- c(1:3, NA)
z

## [1] 1 2 3 NA

ind <- is.na(z)
ind

## [1] FALSE FALSE FALSE TRUE
```

To remove missing values from a vector

```
print(z)

## [1] 1 2 3 NA

x <- z[!is.na(z)]
print(x)

## [1] 1 2 3
```

Check to see if a vector has all, any or a certain number of missing values. These create logical vectors which can be used to filter a matrix or data.frame

```
all(is.na(z))

## [1] FALSE

any(is.na(z))

## [1] TRUE

sum(is.na(z))

## [1] 1

sum(is.na(z)) > 1

## [1] FALSE
```

2.5.3 Creating empty vectors and matrices

To create a empty vector, matrix or data.frame

```
x1 <- numeric()  
x2 <- numeric(5)  
x1.mat <- matrix(0, nrow = 10, ncol = 3)
```

2.6 Quick recap

- R Environment, interface, R help and R-project.org and Bioconductor.org website
- installing R and R packages.
- assignment `<-`, `=`, `->`
- operators `==`, `!=`, `<`, `>`, Boolean operators `&`, `|`
- Management of R session, starting session, `getwd()`, `setwd()`, `dir()`
- Listing and deleting objects in memory, `ls()`, `rm()`
- R Objects

Object	Modes	Allow >1 Modes*
vector	numeric, character, complex or logical	No
matrix	numeric, character, complex or logical	No
list	numeric, character, complex, logical, function, expression, ...	Yes
data frame	numeric, character, complex or logical	Yes
factor	numeric or character	No
array	numeric, character, complex or logical	No

*Whether object allows elements of different modes. For example all elements in a vector or array have to be of the same mode. Whereas a list can contain any type of object including a list.

There are other objects type include *ts* (time series) data time etc. See the R manual for more information. All R Objects have the attributes mode and length.

- Creating objects; `c()`, `matrix()`, `data.frame()`, `seq()`, `rep()`, etc
- Adding rows/columns to a matrix using `rbind()` or `cbind()`
- Subsetting/Accessing elements in a `vector()`, `matrix()`, `data.frame()`, `list()` by element name or index.

2.7 Exercise 1

For this exercise we will work on data from a study which examined the weight, height and age of women. Data from the women Study is available as an R dataset and information about this study can be found by using R help (hint ?women) which we will read directly from the website URL <http://bcb.dfci.harvard.edu/~aedin/courses/Bioconductor/Women.txt> into the object women

Basic tools for reading and writing data are respectively: *read.table* and *write.table*. We will go into further detail about each later today, but first lets read in this file by typing these commands:

```
myURL <-  
"http://bcb.dfci.harvard.edu/~aedin/courses/Bioconductor/Women.txt"  
women <- read.table(myURL, sep = "\t", header = TRUE)
```

(Or in RStudio Click on Workspace -> Import Dataset)

Tasks

1. Get help on the command *colnames*
2. what is the class of this dataset?
3. How many rows and columns are in the data? (hint try using the functions *str*, *dim*, *nrow* and *ncol*)
4. Use the *summary()*, to view the mean height and weight of women
5. Compare the result to using the function *colMeans*
6. How many women have a weight under 120?
7. What is the average height of women who weigh between 124 and 150 pounds (hint: need to select the data, and find the mean).
8. Sort the matrix *women* by 'weight' hint use *order*
9. Give the 5th row the rowname "Lucy"

Chapter 3

Reading and writing data to and from R

So far, we have only analyzed data that were already stored in R. Basic tools for reading and writing data are respectively: *read.table* and *write.table*. We will go into further detail about each. First we will talk about reading in simple text documents; comma and tab-delimited format text files, then Excel and import/export from other statistical software.

3.1 Importing and reading text files data into RStudio

RStudio has a nice user interface to reading in file. Click on Workspace → Import Dataset.

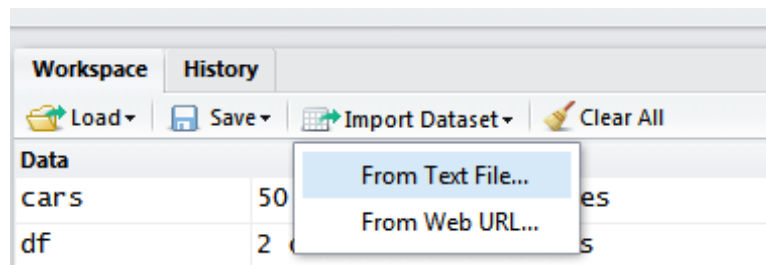


Figure 3.1: which provides an easy approach to read a text from a local directory or directly from a web URL

Enter a file location (either local or on the web), and RStudio will make a "best guess" at the file format. There are a limited number of options (heading yes or no), separators (comma, space or tab) etc but these should cover the most common data exchange formats (The R interfaces RCommander and RExcel also provide rich support for data import of many different file formats into R)

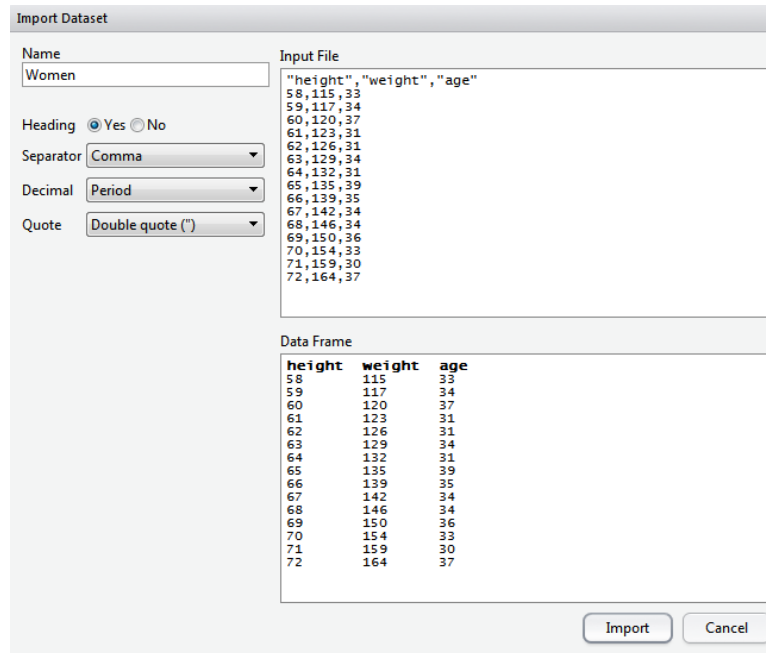


Figure 3.2: The top panel shows the plain text of the file, and the lower panels displays how R is interpreting the data. Black rows are the column headings

3.2 Importing data using R command `read.table()`

If you are calling R from a script, or are using R on a machine in which RStudio is not available, knowledge of commands to read and write files are vital

3.2.1 Using `read.table()` and `read.csv()`

1. The most commonly used function for reading data is `read.table()`. It will read the data into R as a *data.frame*.

By Default `read.table()` assumes a file is space delimited and it will fail if the file is in a different format with the error below.

```
Women<-read.table("Women.txt")
```

In order to read files that are tab or comma delimited, the defaults must be changed. We also need to specify that the table has a header row

```
# Tab Delimited
Women <- read.table("Women.txt", sep = "\t", header = TRUE)
Women[1:2, ]

##   height weight age
## 1     58    115  33
## 2     59    117  34
```

```
summary(Women)
```

```
##      height      weight      age
##  Min.   :58.0   Min.   :115   Min.   :30.0
## 1st Qu.:61.5   1st Qu.:124   1st Qu.:32.0
## Median :65.0   Median :135   Median :34.0
## Mean   :65.0   Mean   :137   Mean   :33.9
## 3rd Qu.:68.5   3rd Qu.:148   3rd Qu.:35.5
## Max.   :72.0   Max.   :164   Max.   :39.0
```

```
class(Women$age)
```

```
## [1] "integer"
```

Note by default, character vector (strings) are read in as factors. To turn this off, use the parameter `as.is=TRUE`

2. Important options:

<code>header==TRUE</code>	should be set to 'TRUE', if your file contains the column names
<code>as.is==TRUE</code>	otherwise the character columns will be read as factors
<code>sep=""</code>	field separator character (often comma ',' or tab '\t' eg: <code>sep=","</code>)
<code>na.strings</code>	a vector of strings which are to be interpreted as 'NA' values.
<code>row.names</code>	The column which contains the row names
<code>comment.char</code>	by default, this is the pound # symbol, use "" to turn off interpretation of commented text.

```
# Read the help file
help(read.table)
```

Note the defaults for `read.table()`, `read.csv()`, `read.delim()` are different. For example, in `read.table()` function, we specify `header=TRUE`, as the first line is a line of headings among other parameters.

3. `read.csv()` is a derivative of `read.table()` which calls `read.table()` function with the following options so it reads a comma separated file:

```
read.csv(file, header = TRUE, sep = ",", quote="\\"", dec=".",
         fill = TRUE, comment.char="", ...)
```

Read in a comma separated file:

```
# Comma Delimited
Women2 <- read.csv("Women.csv")
Women2[1:2, ]

##      height weight age
## 1       58     115   33
## 2       59     117   34
```

4. Reading directly from Website You can read a file directly from the web

```
myURL <-  
"http://bcb.dfci.harvard.edu/~aedin/courses/Bioconductor/Women.txt"  
read.table(myURL, header = TRUE)[1:2, ]  
  
##      height weight age  
## 1         58     115  33  
## 2         59     117  34
```


3.3 Exercise 2

The ToothGrowth data are from a study which examined the growth of teeth in guinea pigs ($n=10$) in response to three dose levels of Vitamin C (0.5, 1, and 2 mg), which was administered using two delivery methods (orange juice or ascorbic acid). Data from the Tooth Growth Study is available as an R dataset and information about this study can be found by using R help (hint ?ToothGrowth)

1. Download the data set "ToothGrowth.xls" which is available on the course website. Save it in your local directory. Open this file "ToothGrowth.xls" in Excel.
 2. Export the data as both a comma or tab delimited text files. In Excel select File -> Save as and
Tab: select the format Text (Tab delimited) (*.txt).
CSV: select the format CSV (Comma delimited) (*.csv).
-
1. Load each data file (.txt and .csv) into R
 2. How many rows are there is ToothGrowth?
 3. what is the mean and sd of Tooth length
 4. Does treatment have a significant effect?

3.3.1 Importing text files Using scan()

NOTE: `read.table()` is not the right tool for reading large matrices, especially those with many columns. It is designed to read 'data frames' which may have columns of very different classes. Use `scan()` instead.

`scan()` is an older version of data reading facility. Not as flexible, and not as user-friendly as `read.table()`, but useful for Monte Carlo simulations for instance. `scan()` reads data into a *vector* or a *list* from a file.

```
myFile <- "outfile.txt"
# Create a file
cat("Some data", "1 5 3.4 8", "9 11 23", file = myFile, sep = "\n")
exampleScan <- scan(myFile, skip = 1)
print(exampleScan)

## [1] 1.0 5.0 3.4 8.0 9.0 11.0 23.0
```

Note by default `scan()` expects numeric data, if the data contains text, either specify `what="text"` or give an example `what="some text"`.

Other useful parameters in `scan()` are `nmax` (number of lines to be read) or `n` (number of items to be read).

```
scan(myFile, what = "some text", n = 3)

## [1] "Some" "data" "1"
```

3.4 Reading data from Excel into R

There are several packages and functions for reading Excel data into R, however I normally export data as a .csv file and use *read.table()*.

However if you wish to directly load Excel data, here are many the options available to you. See the section on "Importing-from-other-statistical-systems" in the webpage <http://cran.r-project.org/doc/manuals/R-data.html> for more information

1. xlsx seems to be the simplest option at the moment

```
library(xlsx)
ww <- read.xlsx(file = "Women.xlsx", sheetIndex = 1)
```

read.xlsx accepts .xls and .xlsx format. You must include a worksheet name or number. It is optional to specify a row or column index to indicate a section of a Worksheet

2. There is also a packages call XLConnect which is similar

```
require(XLConnect)

## Loading required package: XLConnect
## Loading required package: XLConnectJars
## Loading required package: rJava
## XLConnect 0.1-9 by Mirai Solutions GmbH
## <xlconnect@mirai-solutions.com>
## http://www.mirai-solutions.com ,
## http://miraisolutions.wordpress.com

wb <- loadWorkbook("Women.xlsx", create = TRUE)
WW <- readWorksheet(wb, sheet = 1)
```

Or you can read direct from a connection, calling the file directly.

```
Ww <- readWorksheetFromFile("Women.xlsx", name = "sheet1")
```

3. RExcel R can be ran from within Excel on Windows using RExcel (<http://rcom.univie.ac.at/>). This add a menu to Excel that allows you to call R functions from within Excel. RExcel is part of the much large Statconn project.
4. RODBC library. We are not sure it will not work with .xlsx files. See the vignette for more information

```
library(RODBC)
RShowDoc("RODBC", package = "RODBC")
```

The following RODBC function works under windows, but may have issues under MacOS or Linux as may need to install ODBC drivers.

```
channel<-odbcConnectExcel("ToothGrowth.xls")
#list the spreadsheets
sqlTables(channel)

#retrieve the contents of the Excel Sheet ToothGrowth using either of the foll
ToothGrowth<-sqlFetch(channel, "ToothGrowth")
ToothGrowth<-sqlQuery(channel, "select * from [ToothGrowth$]")
ToothGrowth[1:2,]
```

5. The gdata library function *read.xls()*

Perl must be installed on your computer in order for these to work, as it uses the Perl functions 'xls2csv' or 'xls2tab'.

3.5 Import/Export from other statistical software

Most binary data files written by statistical software other than R such as EpiInfo, Minitab, S-PLUS, SAS, SPSS, Stata and Systat, can be loaded into R using the R package *foreign* or *Hmisc*. Details can be found in the R manual: R data Import/Export.

3.5.1 Reading data from SAS

If you have SAS, you can use Frank Harrell's 'Hmisc' package which has functions *sas.get* and *sasxport.get*, and other utility functions such as *label.sas.get*, *contents*, *describe*. For those without a SAS license, package 'foreign' has *read.ssd*, *lookup.xport*, and *read.xport*.

- From SAS, save SAS dataset in transport format

```
libname out xport c:/mydata.xpt;
data out.mydata;
set sasuser.mydata;
run;
```

In R

```
library(Hmisc)
mydata <- sasxport.get("c:/mydata.xpt")
```

- SAScii. Anthony Joseph Damico recently announced SAScii is a new packages to parse SAS input code to read.fwf However although they stated the code below should work, I have been not so successful with it in my hands.

```
require("SAScii")

## Loading required package: SAScii
## Loading required package: stringr
```

```

# Load the 2010 National Health Interview Survey Persons file as an
# R data frame from CDC
NHIS10_personsx_SASInst <-
"ftp://ftp.cdc.gov/pub/Health_Statistics/NCHS/Program_Code/NHIS/2010/PERSONSX.sas"

NHIS10_personsx_SASInst <-
"ftp://ftp.cdc.gov/pub/Health_Statistics/NCHS/Datasets/NHIS/2010/personsx.zip"

# store the NHIS file as an R data frame!
NHIS10_personsx_df <- read.SAScii(NHIS10_personsx_SASInst,
  NHIS10_personsx_SASInst, zipped = TRUE)

# or store the NHIS SAS import instructions for use in a read.fwf
# function call outside of the read.SAScii function
NHIS10_personsx_sas <- parse.SAScii(NHIS10_personsx_SASInst)

# save the data frame now for instantaneous loading later
save(NHIS10_personsx_df, file = "NHIS10_personsx_data.RData")

```

- Recently Xin Wei of Roche Pharmaceuticals published a SAS macro called Proc_R that may potentially ease integrating R and SAS (reference Xin Wei PROC_R: A SAS Macro that Enables Native R Programming in the Base SAS Environment J. Stat Software. Vol. 46, Code Snippet 2, Jan 2012) which allows you to put R code within a SAS macro.

```

%include "C:\aedin\sasmacros\Proc_R.sas";
%Proc_R (SAS2R =, R2SAS =);
Cards4;

*****
***Please Enter R Code Here***
*****

;;;
%Quit;

```

3.5.2 S

PSS

From SPSS, save SPSS dataset in trasport format

```

get file=c:\mydata.sav .
export outfile=c:\mydata.por .

```

In R

```

library(Hmisc)
mydata <- spss.get("c:/mydata.por", use.value.labels = TRUE)

```

3.5.3 Stata or Systat

```
library(foreign)
mydata <- read.dta("c:/mydata.dta")
```

3.6 From a Database Connection

There is also support for database connectivity including for mySQL, Oracle and specialized file formats including network Common Data Form (netCDF) etc. See <http://cran.r-project.org/doc/manuals/R-data.html> for more details.

Note installation of RMySQL or ROracle is simple on Linux or Mac, but maybe complex on MSWindows, as there is no binary file. See the README associated with the package on the R website

```
# ----- mySQL
# -----
library(RMySQL)
drv <- dbDriver("MySQL")

con <- dbConnect(drv, dbname = "myDB", user = "aedin", password =
"xxx",
  host = "myServer.harvard.edu")
dbListTables(con)
dbWriteTable(con, "annot_30", samp)

# ----- Oracle
# -----

library(ROracle)
# create an Oracle instance and create one connection.
drv <- dbDriver("Oracle")
con <- dbConnect(drv, "username/password")
rs <- dbSendQuery(con, "select * from USER_TABLES")
rel <- fetch(rs)
dbGetInfo(con)
```

3.6.1 Parsing each line - Readlines

There are several function in R for parsing large files. You can use the command *readLine* or *readLines* to parse a file line by line.

3.7 Writing Data table from R

1. Function *sink()* diverts the output from the console to an external file

```
myPath <- getwd()
sink(file.path(myPath, "sinkTest.txt"))
print("This is a test of sink")
ls()
sin(1.5 * pi)
print(1:10)
sink()
```

2. Writing a data matrix or data.frame using the *write.table()* function *write.table()* has similar arguments to *read.table()*

```
myResults <- matrix(rnorm(100, mean = 2), nrow = 20)
write.table(myResults, file = "results.txt")
```

This will write out a space separated file.

```
df1 <- data.frame(myResults)
colnames(df1) <- paste("MyVar", 1:5, sep = "")
write.table(df1, file = "results2.txt", row.names = FALSE,
            col.names = TRUE)
read.table(file = "results2.txt", head = TRUE)[1:2, ]
```

```
##   MyVar1 MyVar2 MyVar3 MyVar4 MyVar5
## 1  1.940  2.569  3.199  3.502  3.259
## 2  4.146  2.752  3.839  1.351  2.194
```

3. Important options

append = FALSE	create new file
sep = " "	separator (other useful possibility sep=";")
row.names = TRUE	may need to change to row.names=FALSE
col.names = TRUE	column header

4. Output to a webpage

The package R2HTML will output R objects to a webpage

```
# Write data directly to a new webpage
library(R2HTML)
HTML(df1, outdir = myPath, file = "results.html")
# Capture output to a webpage
HTMLStart(outdir = myPath, filename = "Web_Results", echo = TRUE)
```

```
##
## *** Output redirected to directory: C:/Users/aedin/Dropbox/Talks/CDC/Notes
## *** Use HTMLStop() to end redirection.
## [1] TRUE

print("Capturing Output")

## [1] "Capturing Output"

df1[1:2, ]

##      MyVar1 MyVar2 MyVar3 MyVar4 MyVar5
## 1   1.940   2.569   3.199   3.502   3.259
## 2   4.146   2.752   3.839   1.351   2.194

summary(df1)

##      MyVar1      MyVar2      MyVar3      MyVar4
## Min.      :0.546   Min.      :1.03   Min.      : -0.133   Min.      :0.856
## 1st Qu.:1.237   1st Qu.:1.77   1st Qu.: 1.501   1st Qu.:1.696
## Median :1.914   Median :2.19   Median : 1.838   Median :2.127
## Mean    :1.827   Mean    :2.25   Mean    : 1.929   Mean    :2.176
## 3rd Qu.:2.359   3rd Qu.:2.54   3rd Qu.: 2.377   3rd Qu.:2.474
## Max.    :4.146   Max.    :4.24   Max.    : 3.839   Max.    :3.631
##      MyVar5
## Min.      :0.369
## 1st Qu.:1.684
## Median :2.106
## Mean    :2.256
## 3rd Qu.:3.075
## Max.    :4.029

print("hello and Goodbye")

## [1] "hello and Goodbye"

HTMLStop()

## [1] "C:/Users/aedin/Dropbox/Talks/CDC/Notes/Web_Results_main.html"
```

3.7.1 Other considerations when reading or writing data

It is often useful to create a variable with the path to the data directory, particular if we need to read and/or write more than one dataset. NOTE: use double backslashes to specify the path names, or the forward slash

can be used.

```
myPath <- file.path("C://Project1")
file.exists(myPath)

## [1] FALSE

# Set myPath to be current directory
myPath <- file.path(getwd())
```

It is better to expand a path using *file.path()* rather than *paste()* as *file.path()* will expand the path with delimiting characters appropriate to the operating system in use (eg / unix, \, windows etc)

```
myfile <- file.path(myPath, "Women.txt")
```

Use *file.exists()* to test if a file can be found. This is very useful. For example, use this to test if a file exists, and if TRUE read the file or you could ask the R to warn or stop a script if the file does not exist

```
if (!file.exists(myfile)) {
  print(paste(myfile, "cannot be found"))
} else {
  Women <- read.table(myfile, sep = "\t", header = TRUE)
  Women[1:2, ]
}
```

```
##   height weight age
## 1     58    115  33
## 2     59    117  34
```

3.8 Exercise 3

1. Use *read.table()* to read the space separated text file WomenStats.txt directly from the website "<http://bcb.dfc.harvard.edu/~aedin/courses/R/WomenStats.txt>", Call this *data.frame* women.
2. Change the rownames to be the letters of the alphabet eg "A", "B" "C" "D" etc
3. Write out this file as a tab delimited file using *write.table()*
4. Read this into R using *read.table()*. What parameters need modifying to read the data as a tab-delimited file?

3.9 Sampling and Creating simulated data

1. *seq* and *rep*. we have already seen the function *seq* and *rep* which generate a sequence or repeat elements.
2. Create data from a specific distribution

Often we want to sample data from a specific distribution, also sometimes called simulating data. This data is usually used to test some algorithm or function that someone has written. Since the data is simulated, you know where it came from and so what the answer should be from your algorithm or function. Simulated data lets you double-check your work.

Each distribution has 4 functions associated with it:

For example, *rnorm()*, *dnorm()*, *pnorm()*, and *qnorm()* give random normals, the normal density (sometimes called the differential distribution function), the normal cumulative distribution function (CDF), and the inverse of the normal CDF (also called the quantile function), respectively.

Almost all of the other distributions have similar sets of four functions. The 'r' versions are *rbeta*, *rbinom*, *rcauchy*, *rchisq*, *rexp*, *rf*, *rgamma*, *rgeom*, *rhyper*, *rlogis*, *rlnorm*, *rmultinom*, *rnbinom*, *rnorm*, *rpois*, *rsignrank*, *rt*, *runif*, *rweibull*, and *rwilcox* (there is no *rtukey* because generally only *ptukey* and *qtukey* are needed).

For example, generate 5 observations from a normal distribution with mean 0 and stdev 1, or 10 observation with a mean of 20 and a stdev of 2

```
rnorm(5, 0, 1)

## [1] -0.2986 -0.3762  0.7070  0.1605  0.7018

rnorm(10, 6, 2)

## [1]  7.530  7.741  5.947  4.622  8.804  8.149 10.949  3.339  6.213
## [10]  6.554
```

For most of the classical distributions, these simple function provide probability distribution functions (p), density functions (d), quantile functions (q), and random number generation (r). Beyond this basic functionality, many CRAN packages provide additional useful distributions. In particular, multivariate distributions as well as copulas are available in contributed packages. See <http://cran.r-project.org/web/views/Distributions.html> and <http://cran.r-project.org/doc/manuals/R-intro.html#Probability-distributions> for more information.

3. Sample from existing data

The second type of simulation you may wish to perform is to bootstrap or permute existing data. In bootstrapping one generally follows the same basic steps

- (a) Resample a given data set a specified number of times
- (b) Calculate a specific statistic from each sample
- (c) Find the standard deviation of the distribution of that statistic

The function *sample()* will resample a given data set with or without replacement

```
sample(1:10)

## [1] 3 8 9 7 2 6 4 10 1 5

sample(1:10, replace = TRUE)

## [1] 1 1 8 6 4 8 1 6 10 10
```

You can also add weights to bias selection or probability of selecting of a certain subset. For example bootstrap sample from the same sequence (1:10) with probabilities that favor the numbers 1-5

```
weights <- c(rep(0.25, 5), rep(0.05, 5))
print(weights)

## [1] 0.25 0.25 0.25 0.25 0.25 0.05 0.05 0.05 0.05 0.05

sample(10, replace = T, prob = weights)

## [1] 5 2 10 5 1 3 5 4 9 2
```

3.10 Exercise 4

1. Create the vector which contains the first 20 letters of the alphabet and the sequence of number 0:200 in increments of 10 (hint use *seq()*).
2. Use *sample()* to randomize the order of the vector.
3. Use the function *cat()* to write this vector to a file called "myVec.txt".
4. Use *scan()* to read the first 10 items in the file, what value do you give to the parameter 'what'. Compare running *scan()* with different data types; eg: what="text", what=123 and what=TRUE

Chapter 4

Introduction to programming and writing Functions in R

4.1 Why do we want to write functions?

We have a general knowledge of function use, so how do we write our own functions. In other words, we want to create objects of mode *function*.

1. Why do we want to write functions?
2. Definition of a function: assignment of the form

```
myFunction <- function(arg1, arg2, ...) expression
```

expression is an R expression using arguments *arg1*, *arg2* to calculate a value. Function returns the value of the expression

3. Call to a function within R

```
myFunction(expr1, expr2, ...)
```

4. Lets write a short function, a function to calculate the means of a vector.

```
myMean <- function(y1) {  
  mean <- sum(y1)/length(y1)  
  return(mean)  
}
```

Now Lets test out function

```
testVec <- rnorm(50, 20, 4)  
mean(testVec)  
  
## [1] 19.72  
  
myMean(testVec)  
  
## [1] 19.72
```

5. A more complex example

Example of a function 'twosam': takes as arguments two vectors 'y1' and 'y2', calculates the 2-sample t-test statistic (assuming equal variance), and returns the t-statistic

```
twosam <- function(y1, y2) {  
  n1 <- length(y1)  
  n2 <- length(y2)  
  yb1 <- mean(y1)  
  yb2 <- mean(y2)  
  s1 <- var(y1)  
  s2 <- var(y2)  
  s <- ((n1 - 1) * s1 + (n2 - 1) * s2) / (n1 + n2 - 2)  
  tst <- (yb1 - yb2) / sqrt(s * (1/n1 + 1/n2))  
  return(tst)  
}
```

4.2 Conditional statements (if, ifelse, switch)

1. if statement

```
if (condition) expr1 else expr2
```

condition must evaluate to a single logical value, ie either TRUE or FALSE.

```
x <- 9  
if (x > 0) sqrt(x) else sqrt(-x)  
  
## [1] 3
```

Vectorized version of the if/else construct: *ifelse(condition, expr1, expr2)* function which returns a vector with elements expr1 if condition is true, otherwise it returns expr2.

```
ifelse(x >= 0, sqrt(x), sqrt(-x))  
  
## [1] 3
```

The *switch* function, a generalization of the *if* statement

```
spread <- function(x, type) {  
  switch(type, sd = sd(x), mad = mad(x), IQR = IQR(x) / 1.349)  
}  
samp <- rnorm(50)  
spread(samp, 2)  
  
## [1] 1.043
```

```
spread(samp, "IQR")
```

```
## [1] 0.9385
```

Why $IQR(x)/1.349$? In a normal distribution 50% of the data (between 0.25 and 0.75 quartiles). So the distance between the two quartiles $IQR(x) = \text{quantile}(x, 0.75) - \text{quantile}(x, 0.25)$. For a normal distribution IQR is $\text{qnorm}(0.75) - \text{qnorm}(0.25) \approx 1.349$. Therefore $IQR/1.349$ is an estimator of the standard deviation of a normal distribution.

4.2.1 Repetitive execution: For and While loops

(a) for loops

```
for (i in expr1) expr2
```

where i is the loop variable, expr1 is usually a sequence of numbers, and expr2 is an expression.

```
for (i in 1:5) print(i^2)
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

(b) while loops

while (condition) expr continues till the *condition* becomes false. Used often in iterative calculations

```
x <- 1
y <- 16
while (x^2 < y) {
  cat(x, "squared is ", x^2, "\n") # print x and sq(x)
  x <- x + 1
}
```

```
## 1 squared is 1
## 2 squared is 4
## 3 squared is 9
```

A word of caution, it is easy to write a *while*() loop that doesn't terminate, in which case your script may go into a never-ending cycle. Therefore if possible, write a *for*() loop in preference to a *while*() loop.

4.3 Viewing Code of functions from R packages

Its often useful to view the code of R functions. To see the code, type the name of that code without parenthesis. Take a look closer at a built-in function *IQR*. We see it is simply taking the difference (*diff()*) of the 25

```
help(IQR)

## starting httpd help server ...
## done

IQR

## function (x, na.rm = FALSE, type = 7)
## diff(quantile(as.numeric(x), c(0.25, 0.75), na.rm = na.rm, names = FALSE,
##      type = type))
## <bytecode: 0x0000000002ee6778>
## <environment: namespace:stats>

args(IQR)

## function (x, na.rm = FALSE, type = 7)
## NULL

body(IQR)

## diff(quantile(as.numeric(x), c(0.25, 0.75), na.rm = na.rm, names = FALSE,
##      type = type))

xx <- sample(1:30, 10)
quantile(xx)

##      0%   25%   50%   75%  100%
##    3.0   9.5  13.5  17.5  25.0

IQR(xx)

## [1] 8
```

(Sometimes, functions don't appear to be "visible". In this case, use *methods* or *getAnywhere* to view the function. This is a bit more complex for S4 functions).

```
mean

## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x000000000064e3038>
## <environment: namespace:base>
```



```

methods(mean)

## [1] mean.data.frame mean.Date          mean.default      mean.difftime
## [5] mean.POSIXct      mean.POSIXlt

mean.default

## function (x, trim = 0, na.rm = FALSE, ...)
## {
##     if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
##         warning("argument is not numeric or logical: returning NA")
##         return(NA_real_)
##     }
##     if (na.rm)
##         x <- x[!is.na(x)]
##     if (!is.numeric(trim) || length(trim) != 1L)
##         stop("trim must be numeric of length one")
##     n <- length(x)
##     if (trim > 0 && n) {
##         if (is.complex(x))
##             stop("trimmed means are not defined for complex data")
##         if (any(is.na(x)))
##             return(NA_real_)
##         if (trim >= 0.5)
##             return(stats::median(x, na.rm = FALSE))
##         lo <- floor(n * trim) + 1
##         hi <- n + 1 - lo
##         x <- sort.int(x, partial = unique(c(lo, hi)))[lo:hi]
##     }
##     .Internal(mean(x))
## }
## <bytecode: 0x000000000064e3828>
## <environment: namespace:base>

```

Some function are "Non visible" that means you can see the code. When you These are hidden in the package namespace. use the function methods, non visible functions are marked by an asterisk.

```

`?`(t.test)
t.test

## function (x, ...)
## UseMethod("t.test")
## <bytecode: 0x00000000004e640d8>
## <environment: namespace:stats>

methods(t.test)

## [1] t.test.default* t.test.formula*
##
## Non-visible functions are asterisked

```

To view a hidden or non-visible function use "PackageName:::function"

```
stats:::t.test.default
```

To reduce the output and save paper in the manual, we will just view the first 5 and last 10 lines of the function.

```
head((stats:::t.test.default), 5)

##
## 1 function (x, y = NULL, alternative = c("two.sided", "less", "greater"),
## 2     mu = 0, paired = FALSE, var.equal = FALSE, conf.level = 0.95,
## 3     ...)
## 4 {
## 5     alternative <- match.arg(alternative)

print("truncated...")

## [1] "truncated..."

tail((stats:::t.test.default), 10)

##
## 102     names(mu) <- if (paired || !is.null(y))
## 103         "difference in means"
## 104     else "mean"
## 105     attr(cint, "conf.level") <- conf.level
## 106     rval <- list(statistic = tstat, parameter = df, p.value = pval,
## 107         conf.int = cint, estimate = estimate, null.value = mu,
## 108         alternative = alternative, method = method, data.name = dname)
## 109     class(rval) <- "htest"
## 110     return(rval)
## 111 }
```

There are some functions that you will not be able to see using these commands. These are most likely written in object orientated R (called S4). Much of Bioconductor's functions are written in S4. However a full discussion of S4 functions is beyond the scope of this course.

4.4 The Apply Functions

Iterative "For loops" in R may sometimes be memory intensive, and functions such as `apply`, `sweep` or `aggregate` should be used instead.

(a) `apply`

apply() applies a function over the rows or columns of a matrix. The syntax is

```
apply(X, MARGIN, FUN, ARGS)
```

where X: array, matrix or data.frame; MARGIN: 1 for rows, 2 for columns, c(1,2) for both;
FUN: one or more functions; ARGS: possible arguments for function

For example, lets go back to the example dataset women which we loaded from the web.

```
summary(women)

##      height      weight      age
##  Min.   :58.0   Min.   :115   Min.   :30.0
##  1st Qu.:61.5   1st Qu.:124   1st Qu.:32.0
##  Median :65.0   Median :135   Median :34.0
##  Mean   :65.0   Mean   :137   Mean   :33.9
##  3rd Qu.:68.5   3rd Qu.:148   3rd Qu.:35.5
##  Max.   :72.0   Max.   :164   Max.   :39.0

colMeans(women)

## height weight      age
##  65.00 136.73  33.93

apply(women, 2, mean)

## height weight      age
##  65.00 136.73  33.93

testEq <- all(rowMeans(women) == apply(women, 1, mean))
print(testEq)

## [1] TRUE

if (testEq) print("rowMeans is equivalent to apply(df, 1,
mean) ")

## [1] "rowMeans is equivalent to apply(df, 1, mean) "
```

Create a function using apply

```
colSd <- function(df) apply(df, 2, sd)
colSd(women)

## height weight      age
##  4.472 15.499  2.576
```

(b) `tapply`

`tapply()` is a member of the very important `apply()` functions. It is applied to "ragged" arrays, that is array categories of variable lengths. Grouping is defined by vector.

The syntax is:

```
tapply(vector, factor, FUN)
```

Example:

```
ageSplit <- ifelse(women$age < 35, "under35", "over35")
print(ageSplit)

## [1] "under35" "under35" "over35" "under35" "under35" "under35"
## [7] "under35" "over35" "over35" "under35" "under35" "over35"
## [13] "under35" "under35" "over35"

tapply(women$weigh, ageSplit, length)

## over35 under35
##      5      10

tapply(women$weigh, ageSplit, summary)

## $over35
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      120    135    139    142    150    164
##
## $under35
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      115    124    130    134    145    159
##
```

(c) `lapply` and `sapply`

`lapply()` and `sapply()` are applied to lists. `lapply()` returns a *list* (of the same length as input). `sapply()` is a user-friendly version of `lapply` by default returning a *vector* or *matrix* if appropriate.

`tapply` will

```
myList <- list(ToothGrowth = TG, WomenAge = women$age, beta =
exp(-3:3),
  logicalVec = c(TRUE, FALSE, FALSE, TRUE))
# compute the list mean for each list element
res1 <- lapply(myList, length)
```

```
print(res1)
print(paste("Class of res1:", class(res1)))

res2 <- sapply(myList, length)
print(res2)
print(paste("Class of res2:", class(res2)))
```

(d) for more apply functions see the library `plyr`

4.5 Exercise 5

- (a) Write a *for* loop printing the consecutive powers of 2, from 0 to 10
- (b) Write a *while* loop printing the consecutive powers of 2, less than 1000

4.5.1 Efficient For Loop in R (Use apply)

sapply and lapply will repeat a function over a list

```
myList <- list(WomenMat = women, WomenAge = women$age, beta =
exp(-3:3),
  logicalVec = c(TRUE, FALSE, FALSE, TRUE))
# compute the list mean for each list element
res1 <- lapply(myList, length)
print(res1)
print(paste("Class of res1:", class(res1)))

res2 <- sapply(myList, length)
print(res2)
print(paste("Class of res2:", class(res2)))
```

Note apply is much more computational efficient than a for loop. But if you can use built in functions like rowMeans or colMeans these are quicker still

```
myMA <- matrix(rnorm(1e+06), 1e+05, 10, dimnames = list(1:1e+05,
  paste("C", 1:10, sep = " ")))

results <- NULL
system.time(for (i in seq(along = myMA[, 1])) results <- c(results,
  mean(myMA[i, ])))

##      user  system elapsed
##    27.55    14.55    42.46

results <- numeric(length(myMA[, 1]))
system.time(for (i in seq(along = myMA[, 1])) results[i] <-
  mean(myMA[i,
    ]))

##      user  system elapsed
##     2.33     0.00     2.33

system.time(myMAmean <- apply(myMA, 1, mean))

##      user  system elapsed
##     2.04     0.00     2.05

system.time(myMAmean <- rowMeans(myMA))
```

```
##      user  system elapsed
##      0      0      0

system.time(myMA$d <- apply(myMA, 1, sd))

##      user  system elapsed
##      3.94    0.01    3.96

system.time(myMA$d <- sqrt((rowSums((myMA -
rowMeans(myMA))^2))/(length(myMA[1,
]) - 1)))

##      user  system elapsed
##      0.03    0.00    0.03
```

4.6 Functions for parsing text

There are many functions with R for parsing text. We will cover a few here.

- To search for text within an R vector, use *grep*. It uses the same regular expression patterns as perl is you set `perl=TRUE`

```
grep("A", LETTERS)

## [1] 1

grep("A", LETTERS, value = TRUE)

## [1] "A"
```

- To substitute characters within a string use *sub*

```
sub("B", "A", LETTERS)

## [1] "A" "A" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P"
## [17] "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

- To split a character string use *strsplit*

```
a <- date()
strsplit(a, " ")

## [[1]]
## [1] "Tue" "Jun" "19" "15:01:08" "2012"
##

strsplit(a, "J")

## [[1]]
## [1] "Tue " "un 19 15:01:08 2012"
##

b <- strsplit(a, "11")
class(b)

## [1] "list"

b <- unlist(b)
class(b)

## [1] "character"
```


- For special characters you need to precede them with a double back slash

```
a <- "aedin@jimmy.harvard.edu"
strsplit(a, "\\.")

## [[1]]
## [1] "aedin@jimmy" "harvard"      "edu"
##
```

4.7 missing values

```
a <- c(1:10, NA, NA)
a <- c(1:10, NA, NA)
summary(a)

##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.     NAs
##      1.00   3.25   5.50   5.50   7.75  10.00         2

mean(a)

## [1] NA

mean(a, na.rm = TRUE)

## [1] 5.5
```

4.8 Exercise 6: Parsing Real Data - World Population Data from Wikipedia

We demonstrated how to get data tables a URL. This means we can retrieve data from most any table on the web. The function `readHTMLTable` is very flexible for this. Please retrieve the table entitled "Estimated world population at various dates (in millions)" (Table 13) from

http://en.wikipedia.org/wiki/World_population.

```
require(XML)

## Loading required package: XML

worldPop <- readHTMLTable("http://en.wikipedia.org/wiki/World_population")
names(worldPop)

## [1] "NULL"
## [2] "toc"
## [3] "NULL"
## [4] "World population milestones (USCB estimates)"
## [5] "The 10 countries with the largest total population:"
## [6] "10 most densely populated countries (with population above 1 million)"
## [7] "Countries ranking highly in terms of both total population (more than 15 million p"
## [8] "NULL"
## [9] "UN (medium variant 2010 revision) and US Census Bureau (December 2010) estimates"
## [10] "UN 2008 estimates and medium variant projections (in millions)[98]"
## [11] "World historical and predicted populations (in millions)[102][103]"
## [12] "World historical and predicted populations by percentage distribution [102][103]"
## [13] "Estimated world population at various dates (in millions)"
## [14] "Starting at 500 million"
## [15] "Starting at 375 million"
## [16] "NULL"
## [17] "NULL"
## [18] "NULL"
## [19] "NULL"

worldPop <- worldPop[[13]] # Just look at Table 13
```

1. First tidy the data. The data are factors, its easier to edit data that are characater. Apply `as.character` to each column
2. Remove rows with dates before 1750. Remove the additional header in row 32.
3. Remove colum 9 notes
4. Use the `sub` to remove the comma in the data values
5. convert the data to numeric
6. In what year did the population of Europe, Africa and Asia exceed 500 million?
7. Bonus: Plot the population growthof the World, Africa or Europe since 1750. Given this plot, would you guess that the the population of the World, Africa or Europe would be more likely to double again before the end of 21st century?

4.8.1 Writing functions: More on arguments

We are equipped now with all basic tools we need for writing functions. We include a few tips on arguments to functions.

1. Function arguments: Default values In many cases arguments have default values. For example the `qnorm(x, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)` has default values for the mean, standard deviation, cdf calculation and probabilities on the original scale.

```
prob <- c(0.5, 0.9, 0.95, 0.975, 0.99)
args(qnorm)
qnorm(prob)
qnorm(prob, 2)
qnorm(prob, mean = 2, sd = 1)
```

2. Function arguments: order is important

- The argument sequence may begin in the unnamed, positional form, and specify named arguments after the positional arguments
- If arguments to functions are given in the form *name=object* form, they may be given in any order
- The argument sequence may be given in the unnamed, positional form
- For example the following statements are equivalent

```
prob <- c(0.5, 0.9, 0.95, 0.975, 0.99)
args(qnorm)
qnorm(p = prob, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(prob, 0, 1, log = FALSE, low = TRUE)
qnorm(prob, 0, 1, TRUE, FALSE)
```

3. Functions: A few points

- Sometimes you may see the parameter "...", this is normally when functions call other functions and arguments are passed from one function to another.
- If commands are stored in an external R script file, say `L2.R` they can be executed at any time in R

```
source(paste(myPath, "L2.R", sep=))
```

- Once a function is defined, can call it from other functions
- The built-in functions supplied with R are a valuable resource for learning about R programming

4.9 Writing functions: more technical discussion -Scoping

1. Scoping

Symbols in the body of a function can be divided into three classes:

- Formal parameters (appear in the argument list of the function)
- Local variables (values are determined by the evaluation of expressions in the body of the functions)
- Free variables (neither of the above)

In this example: x - formal parameter, y - local variable, z - free variable.

Example:

```
fn <- function(x) {  
  y <- 2 * x  
  print(x)  
  print(y)  
  print(z)  
}  
  
z <- 2  
x <- 4  
fn(x = 2)  
  
## [1] 2  
## [1] 4  
## [1] 2
```

2. Lexical scope.

Example: function called *cube*.

```
cube <- function(n) {  
  sq <- function() n * n  
  n * sq()  
}  
cube(2)  
  
## [1] 8  
  
n <- 4  
cube(2)  
  
## [1] 8
```

4.10 Options for Running memory or CPU intensive jobs in R

4.10.1 Distributed computing in R

There are two ways to split or distribute a big job. The simplest is to send jobs to different processors on the same machine (assuming it has multiple cores, which most new machines do). The second option is to split or parallelize a job across many machines or a cluster of machines. For both of these, see the Bioconductor package *parallel* which builds upon the older R packages *snow* and *multicore*

To install *parallel* use the Bioconductor package installer, *BiocInstaller*

```
library(BiocInstaller)
biocLite("parallel")
```

The package *parallel* has many functions which work like *apply* to distribute a computation. For example use *mclapply* just like *lapply* to split a job over 4 cores.

```
library(parallel)
system.time(mclapply(1:4, function(i) mc.cores <- 4))
```

mclapply is a parallelized version of *lapply*, and will not work on windows (as far as I know) but on Windows you can use functions *parLapply*, *clusterApply* and *clusterApplyLB* all in the *parallel* package.

The package has several functions for different types of *apply* loops including *parLapply*, *parSapply*, and *parApply* which are parallel versions of *lapply*, *sapply* and *apply* respectively.

```
library(parallel)
cl <- makeCluster(3)
parLapply(cl, 1:3, sqrt)
stopCluster(cl)
```

For more help on this package, see the vignette

```
vignette("parallel", package = "parallel")
```

There are several other packages for distributed computing see the reviews of R packages on CRAN task views <http://cran.r-project.org/web/views/HighPerformanceComputing.html>. I have received recommendations on R packages *biglm*, *ff* and *bigmemory*.

4.10.2 Running R in the Cloud

One quick-start approach to running R in the Cloud is to register for an Amazon cloud account and then simply direct your web browser at the Bioconductor RCloud instance

<http://www.bioconductor.org/help/bioconductor-cloud-ami/>.

It will open a RStudio interface and it has the same look and feel as the desktop version, making the transition pretty seamless.

If you wish to set up your own instance, Louis Alsett at Trinity College Dublin provides an RStudio Server Amazon Machine Image (AMI) which can install to your Cloud account see http://www.louisaslett.com/RStudio_AMI/.

For more information about distributed computing on the Cloud based including using Hadoop (which I think is used by Revolution Analytics) see the recent book "parallel R" by Q Ethan McCallum <http://www.amazon.com/Parallel-R-Q-Ethan-McCallum/dp/1449309925>

4.11 Efficient R coding

4.11.1 What is an R script

A R script is simply a text file, with R commands. There are two ways to call these R commands, start R and using the R function *source*, or at the command line using R CMD BATCH

4.11.2 What a script should look like ;-)

```
#####
### Author: Mr Bob Parr
### Date: 2011-01-20
### Version: 1.0
### License: GPL (>= 3)
###
### Description: survival analysis function
#####

## This function censors the survival data at a specific point in
## time. This is useful if you used datasets having different
## follow-up periods.
##
## Arguments:
## -----
## surv.time: vector of times to event occurrence [numeric]
## surv.event: vector of indicators for event occurrence [0/1]
## time.cens: point in time at which the survival data must
##             be censored [integer]. Defaults value is '0'
##
## Output: [list of two items]
## -----
## surv.time.cens: vector of censored times to event occurrence [numeric]
## surv.event.cens: vector of censored indicators for event occurrence [0/1]

censor.time <- function(surv.time, surv.event, time.cens=0) {
  stc <- surv.time
  sec <- surv.event
  cc.ix <- complete.cases(stc, sec)
  if(time.cens != 0) {
    stc[cc.ix][surv.time[cc.ix] > time.cens] <- time.cens
    sec[cc.ix][surv.time[cc.ix] > time.cens] <- 0
  }
  return(list("surv.time.cens"=stc, "surv.event.cens"=sec))
}
```

You can save this script in a file named `censortime.R` in your working directory. If you want to define this function in your workspace, just type `source("censortime.R")`.

Of course, an R script may contain more than functions, it may also contain any analytical pipeline. Here is another example:

```
#####
### Author: Mr Bob Parr
```

```

### Date: 2011-01-20
### Version: 1.0
### License: GPL (>= 3)
###
### Description: Script fitting a Cox model on the colon data
### and writing the coefficients in a txt file
#####

## load library
library(survival)

## load colon dataset
data(colon)

## Fit the cox model
coxmodel <- coxph(Surv(time, status) ~ rx, data=colon)

## save summary in a txt file in the working directory
write.table(t(coxmodel$coefficients), sep="\t", file="cox_coefficients_colon.txt",
            row.names=FALSE)

```

Save this script in a file named `coxColon.R` in your working directory. you can run it from your R session using the command `source("coxColon.R")` or you can run it in batch mode from a command line (e.g., shell console) using the command **R CMD BATCH coxColon**.

4.11.3 Coding Recommendations

These are the coding recommendations from the Bioconductor project, and whilst you do not have to do these, it is handy to adopt good working practice when you learn a new language.

1. Indentation

- Use 4 spaces for indenting. No tabs.
- No lines longer than 80 characters. No linking long lines of code using “;”

2. Variable Names

- Use camelCaps: initial lowercase, then alternate case between words.

3. Function Names

- Use camelCaps: initial lower case, then alternate case between words.
- In general avoid ‘.’, as in `some.func`

Whilst beyond the scope of this class, R packages are written to either S3 or S4 standards. In the S3 class system, `some(x)` where `x` is class func will dispatch to this function. Use a ‘.’ if the intention is to dispatch using S3 semantics.

4. Use of space

- Always use space after a comma. This: a, b, c. Not: a,b,c.
- No space around "=" when using named arguments to functions. This: somefunc(a=1, b=2), not: somefunc(a = 1, b = 2).
- Space around all binary operators: a == b.

5. Comments

- Use "##" to start comments.
- Comments should be indented along with the code they comment.

6. Misc

- Use "<-" not "=" for assignment.

7. For Efficient R Programming, see slides and exercises from Martin

<http://www.bioconductor.org/help/course-materials/2010/BioC2010/>

8. R packages which tidy your code There is a package called formatR <https://github.com/yihui/formatR/wiki/> which will format all R script in a folder, indenting loops, convert the = to -> etc. See its wiki pages above if you are interesting in testing it.

4.11.4 Debugging R Code

Use the cat() and print() functions to print values in scripts as you go. I also use the *traceback* to find out what went wrong when a function doesn't work A full list of functions for debugging R code is beyond the scope of this lecture, but see the following useful tips from Duncan Murdoch <http://www.stats.uwo.ca/faculty/murdoch/software/debuggingR/>

4.11.5 End-User Messages

- message() communicates diagnostic messages (e.g., progress during lengthy computations) during code evaluation.
- warning() communicates unusual situations handled by your code.
- stop() indicates an error condition.
- cat() or print() are used only when displaying an object to the user, e.g., in a show method.

4.11.6 system.time

If you wish to check the efficient of your code to see how long it is taking to run, use the function *system.time* which gives the compute time for a function

```
df <- matrix(rnorm(5e+06), ncol = 20000)
system.time(apply(df, 1, mean))
```



```
##      user  system elapsed
##      0.18    0.03    0.22
```

```
system.time(rowMeans(df))
```

```
##      user  system elapsed
##      0.01    0.00    0.01
```

```
system.time()
```

4.11.7 Etiquette when emailing the R mailing list

When all else failed, ask an expert. The R mailing list is a wonderful resource with a very help bunch of experts who will be more than willing to help. But before you email, please do check if someone has asked the same question before or if there is a simple answer to your problem in the R manual or frequently asked questions (FAQ) documentation. The easiest way to do this is to search on <http://www.rseek.org>

If you still need to ask an expert on the mailing list

- Do Send in example code
- Include information about your operating system and version of R. The easiest way to do this is using `sessionInfo()` for example, see this recent post on the mailing list <https://stat.ethz.ch/pipermail/r-sig-mixed-models/2010q3/004467.html>

Writing R packages

Once you have written all your functions in one or several R files, you can use the function `package.skeleton` to generate the necessary directories and empty help pages for your package.

```
package.skeleton(name = "myFirstRPackage")
```

For coding recommendations see <http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html> or http://wiki.fhcrc.org/bioc/Coding_Standards

Hint: all the packages on CRAN and BioConductor are open source, so you can easily download the source of any package to take a closer look at it. It may be extremely insightful to see how experienced R developers implemented their own packages.

Using SVN

RStudio v0.96 contains an easy interface to subversion (either GIT or SVN), but here is a detailed guide to using svn

http://tortoisesvn.net/docs/release/TortoiseSVN_en/tsvn-repository.html#tsvn-repository-create-tortoisesvn

Step 1. Create local SVN repository

1. Open the windows explorer
2. Create a new folder and name it e.g. SVNRepository
3. Right-click on the newly created folder and select TortoiseSVN Create Repository here....
4. A repository is then created inside the new folder. Don't edit those files yourself!!!. If you get any errors make sure that the folder is empty and not write protected.
5. For Local Access to the Repository you now just need the path to that folder. Just remember that Subversion expects all repository paths in the form file:///C:/SVNRepository/. Note the use of forward slashes throughout.
6. So far this is an empty repository, even though Subversion has created several directories and files! We need to fill it with our project files and connect it with our working project directory

Step 2: Initial import.

1. Somewhere in your hard drive create a directory (e.g. tmp) with the following three subdirectories:

```
C:\tmp\new\branches
C:\tmp\new\tags
C:\tmp\new\trunk
```

2. Backup and Tidy your existing scripts and project files (C:\Projects\MyProject). (ie delete unnecessary files)
3. Copy the contents of \MyProject into the trunk subdirectory (C:\tmp\new\trunk).
4. Import the 'new' directory into the repository (Right-click/TortoiseSVN/Import). Select URL as file:///C:/SVNRepository/Myproject (forward slashes!)
5. To see it works, right mouse click start TortoiseSVN/Repo-browser... see your Imported files.. Happy days. Now you have an SVN with all your files

Step 3. Using SVN

1. Now we have created the SVN, the trick is to use it!!! Start by checking out your data. Create a new scripts directory (or go back to your old one and delete its contents). And right mouse click and select "SVN Checkout"
2. To use the SVN Sending (checking in) your changes to the repository: Right-click on selected files then "SVN Commit"
3. To add new files to the repository. This is a two step process: , first Right-click on selected files then "TortoiseSVN/Add" Then Right-click on selected files then "SVN Commit"

4. If you wish to delete files (remember the SVN will always have a history of them) use "TortoiseSVN/Delete"
5. Happy Subversioning!

Chapter 5

Introduction to graphics in R

To start let's look at the basic plots that can be produced in R using the `demo()` function

```
demo(graphics)
```

On start up, R initiates a graphics device driver which opens a special graphics window for the display of interactive graphics. If a new graphics window needs to be opened either `win.graph()` or `windows()` command can be issued.

Once the device driver is running, R plotting commands can be used to produce a variety of graphical displays and to create entirely new kinds of display.

5.1 The R function `plot()`

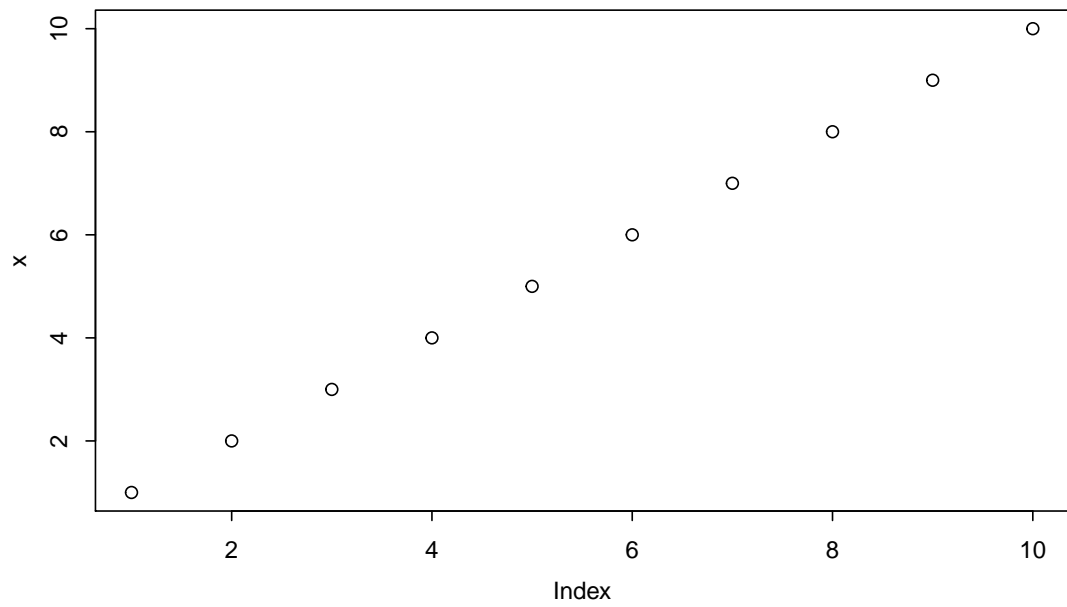
The `plot()` function is one of the most frequently used plotting functions in R.

IMPORTANT: This is a generic function, that is the type of *plot* produced is dependent on the *class* of the first argument.

- Plot of Vector(s)

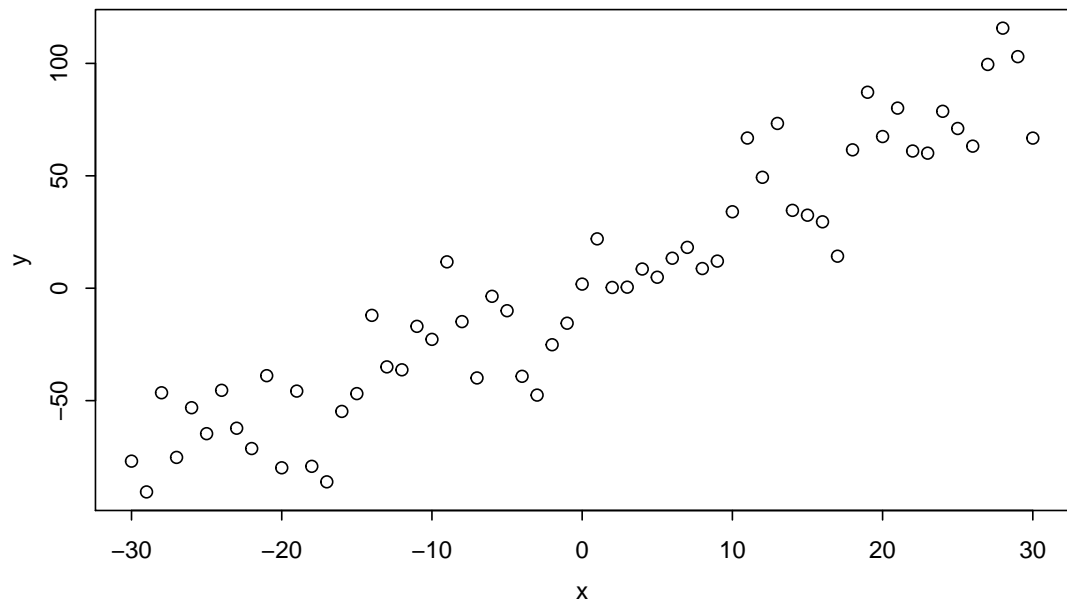
1. One vector `x` (plots the vector against the index vector)

```
x <- 1:10  
plot(x)
```



2. Scatter plot of two vectors x and y

```
set.seed(13)
x <- -30:30
y <- 3 * x + 2 + rnorm(length(x), sd = 20)
plot(x, y)
```



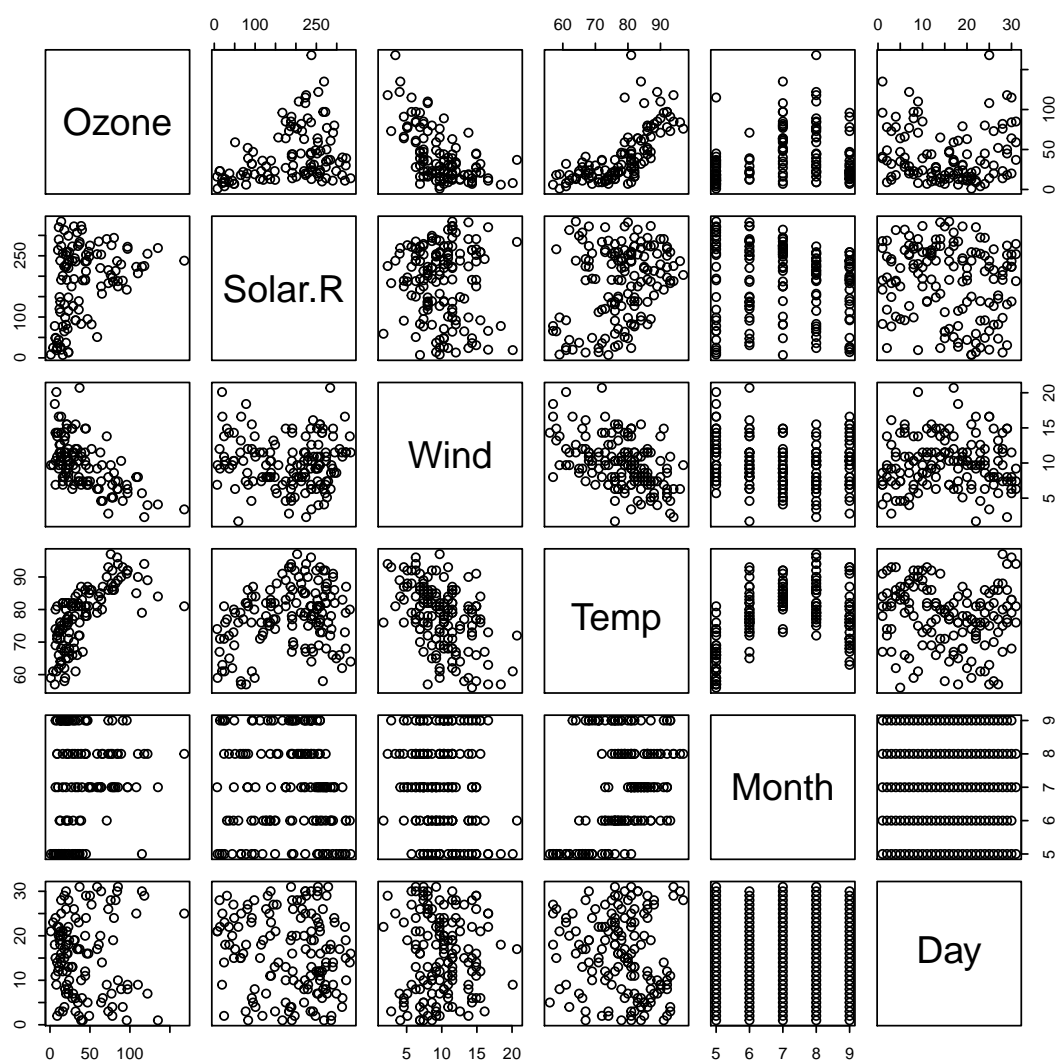
- Plot of *data.frame* elements If the first argument to *plot()* is a *data.frame*, this can be as simply as *plot(x,y)* providing 2 columns (variables in the *data.frame*).

Lets look at the data in the *data.frame* *airquality* which measured the 6 air quality in New York, on a daily basis between May to September 1973. In total there are 154 observation (days).

```
airquality[1:2, ]
```

```
##      Ozone Solar.R Wind Temp Month Day
## 1      41     190  7.4   67     5    1
## 2      36     118  8.0   72     5    2
```

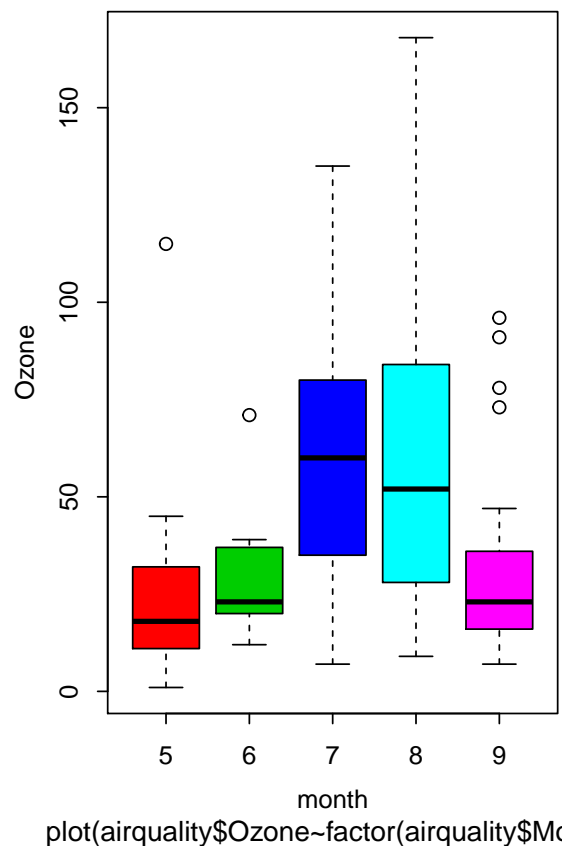
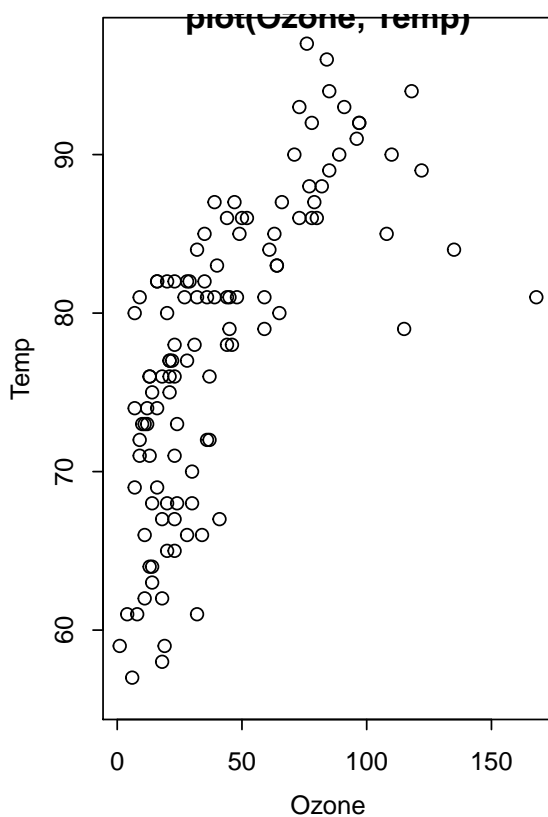
```
plot(airquality) # all variables plotted against each other
pairs()
```



Note most plotting commands always start a new plot, erasing the current plot if necessary. We'll

discuss how to change the layout of plots so you can put multiple plots on the same page a bit later. But a simple way to put multiple plots in the same window is by splitting the display using *mfrow*. Note if you give plot a vector and factor plot(factor, vector) or plot(vector factor) it will produce a boxplot.

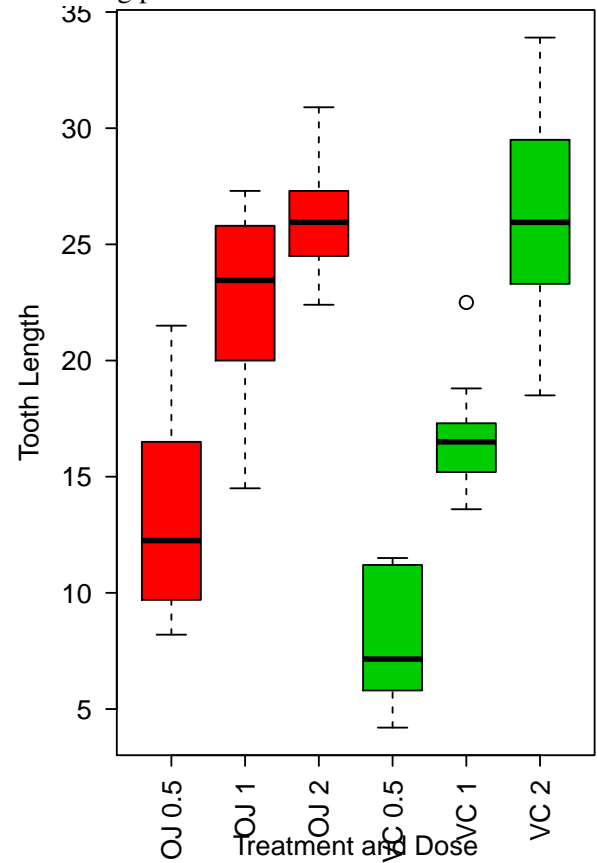
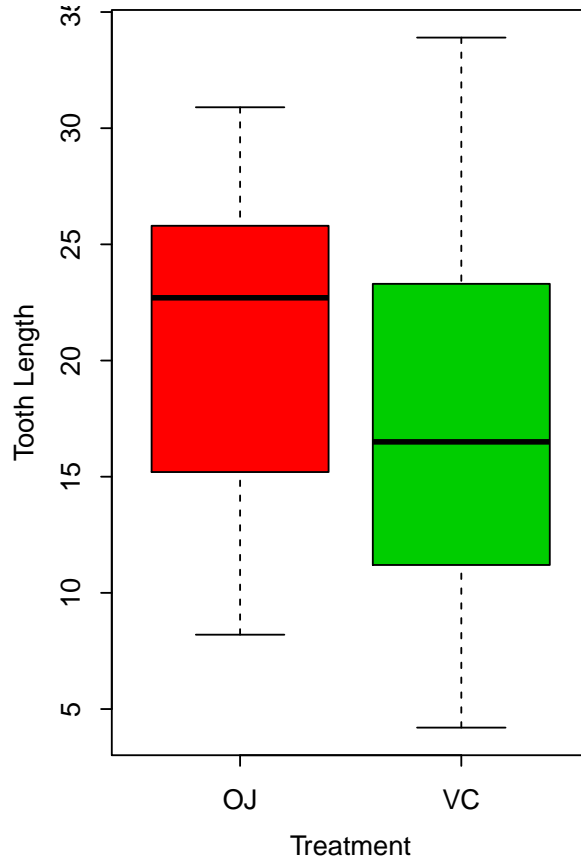
```
par(mfrow = c(1, 2))
attach(airquality)
plot(Ozone, Temp, main = "plot(Ozone, Temp)")
plot(airquality$Ozone ~ factor(airquality$Month), col = 2:6,
      sub = "plot(airquality$Ozone~factor(airquality$Month))", ylab =
"Ozone",
      xlab = "month")
```



```
detach(airquality)
```

5.2 Exercise 7

Using the ToothGrowth data we read earlier. Please draw the following plot



5.2.1 Arguments to plot

axes=FALSE Suppresses generation of axes-useful for adding your own custom axes with the `axis()` function. The default, `axes=TRUE`, means include axes.

type= The `type=` argument controls the type of plot produced, as follows:

`type="p"` Plot individual points (the default)

`type="l"` Plot lines

`type="b"` Plot points connected by lines (both)

`type="o"` Plot points overlaid by lines

`type="h"` Plot vertical lines from points to the zero axis (high-density)

`type="n"` No plotting at all. However axes are still drawn (by default) and the coordinate system is set up according to the data. Ideal for creating plots with subsequent low-level graphics functions.

xlab=string

ylab=string Axis labels for the x and y axes. Use these arguments to change the default labels, usually the names of the objects used in the call to the high-level plotting function.

main=string Figure title, placed at the top of the plot in a large font.

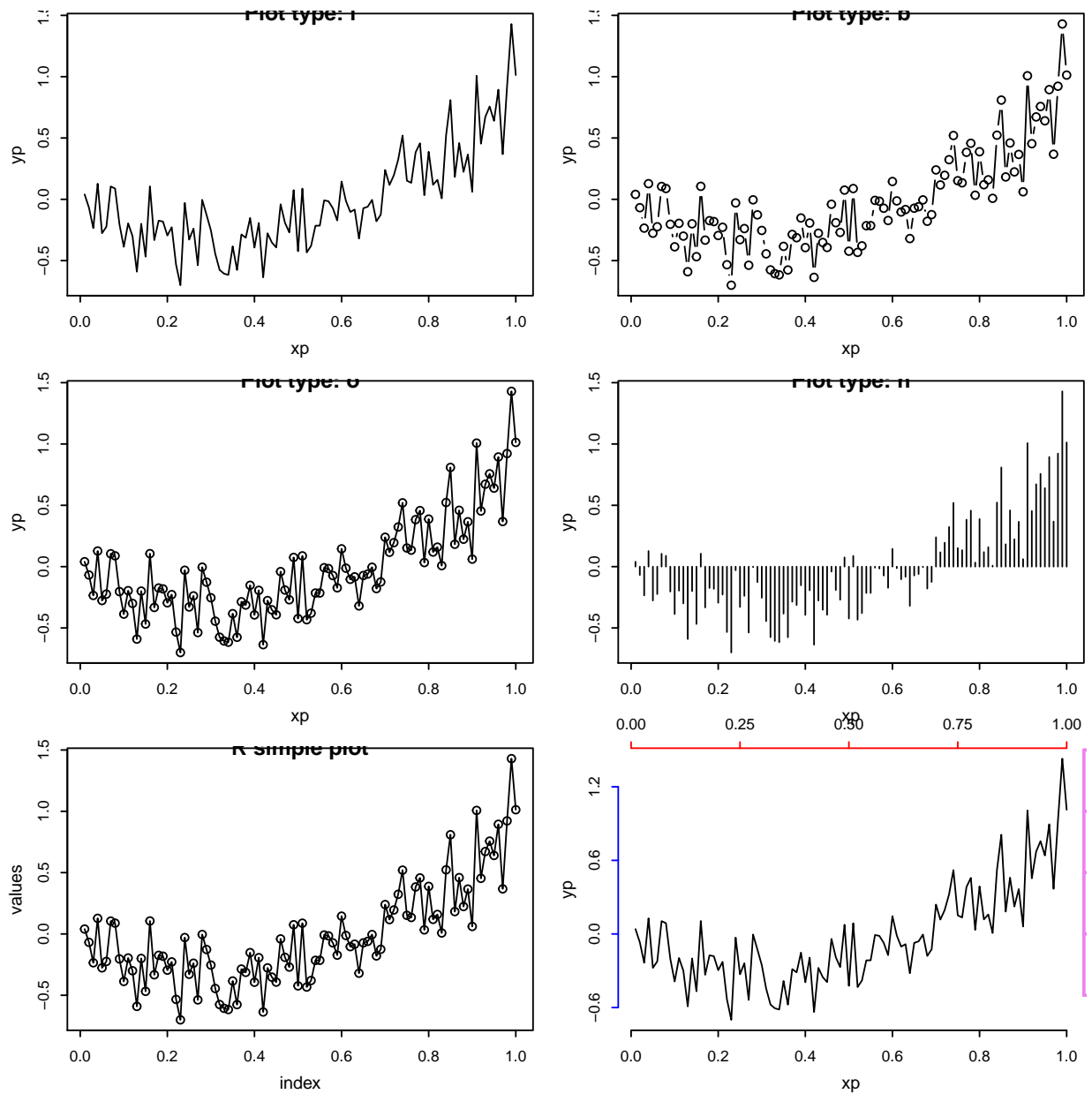
sub=string Sub-title, placed just below the x-axis in a smaller font.

Some Examples of Plotting using different plot types and axes

```
xp <- 1:100/100
yp <- 3 * xp^2 - 2 * xp + rnorm(100, sd = 0.2)

par(mfrow = c(3, 2))
for (i in c("l", "b", "o", "h")) plot(xp, yp, type = i, main =
  paste("Plot type:",
    i))

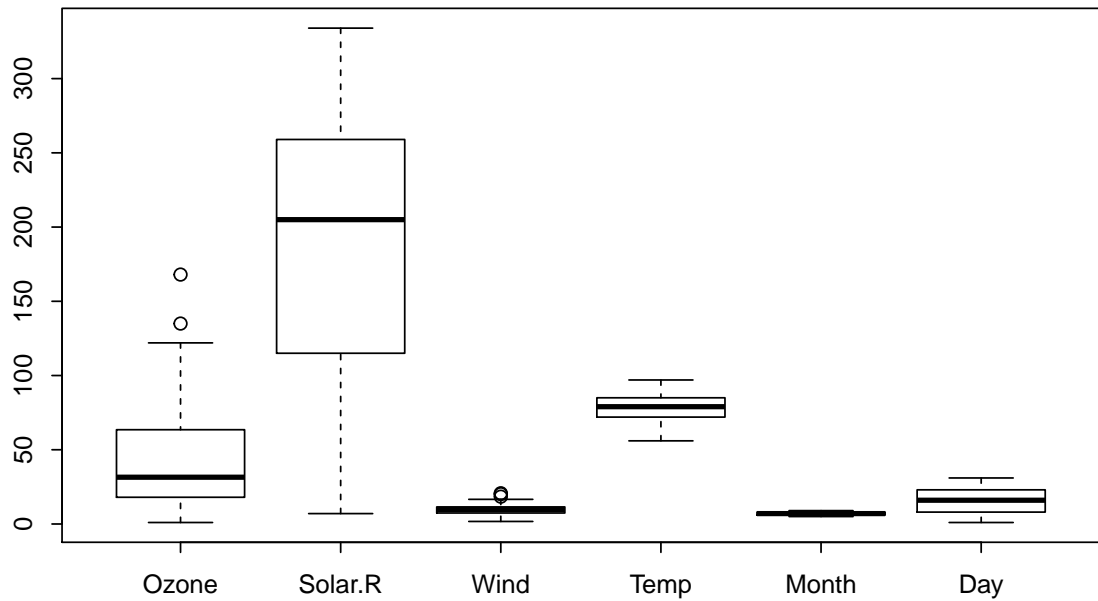
plot(xp, yp, type = "o", xlab = "index", ylab = "values",
  main = "R simple plot")
plot(xp, yp, type = "l", axes = FALSE)
axis(1)
axis(2, at = c(-0.6, 0, 0.6, 1.2), col = "blue")
axis(3, at = c(0, 0.25, 0.5, 0.75, 1), col = "red")
axis(4, col = "violet", col.axis = "dark violet", lwd = 2)
```



5.2.2 Other useful basic graphics functions

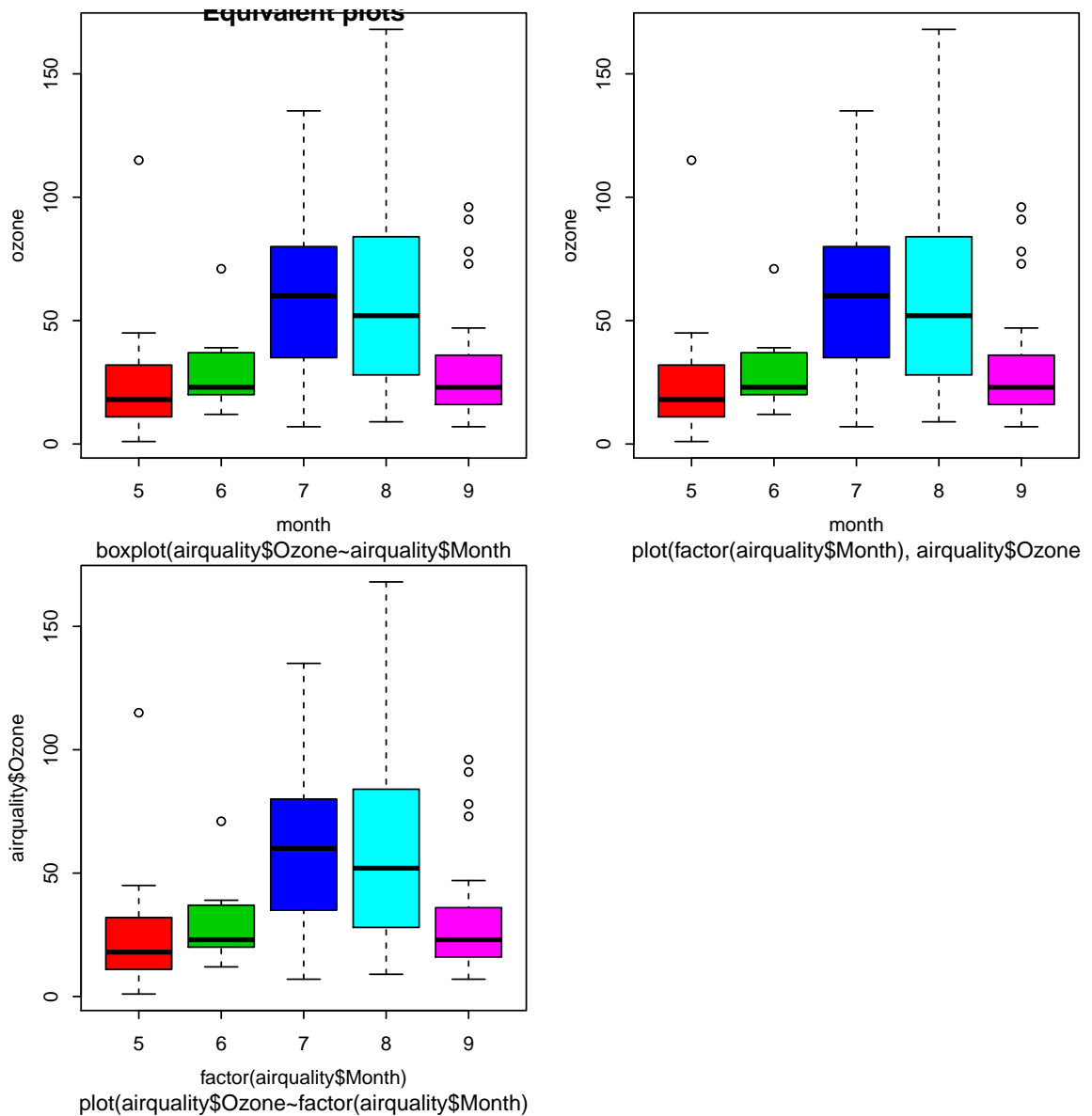
- *boxplot(x)* a boxplot show the distribution of a vector. It is very useful to example the distribution of different variables.

```
boxplot(airquality)
```



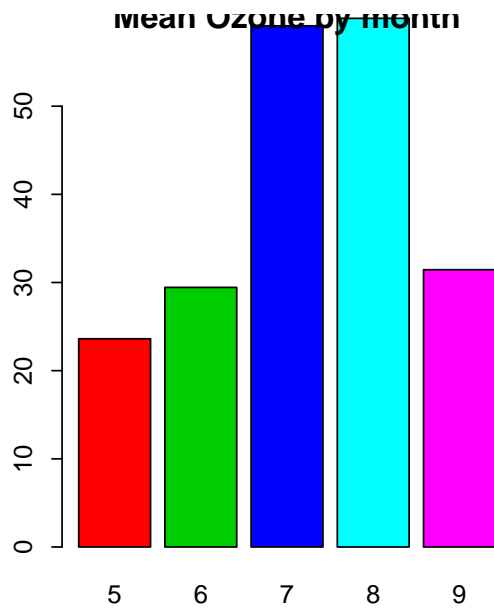
Note if you give plot a vector and factor `plot(factor, vector)` or `plot(vector factor)` it will produce a boxplot.

```
par(mfrow = c(2, 2))
boxplot(airquality$Ozone ~ airquality$Month, col = 2:6, xlab =
"month",
       ylab = "ozone", sub =
"boxplot(airquality$Ozone~airquality$Month)")
title("Equivalent plots")
plot(factor(airquality$Month), airquality$Ozone, col = 2:6,
     xlab = "month", ylab = "ozone", sub =
"plot(factor(airquality$Month), airquality$Ozone)")
plot(airquality$Ozone ~ factor(airquality$Month), col = 2:6,
     sub = "plot(airquality$Ozone~factor(airquality$Month) ")
```



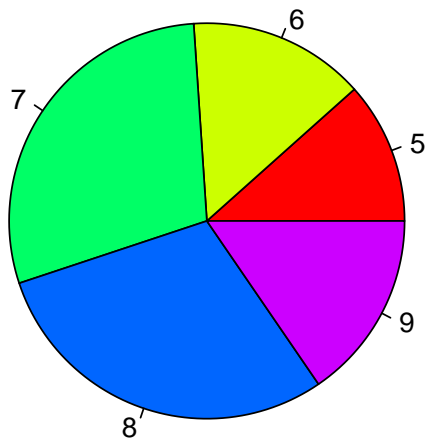
- **barplot** Plot a bar plot of the mean ozone quality by month. First use `tapply` to calculate the mean of ozone by month

```
OzMonthMean <- tapply(airquality$Ozone, factor(airquality$Month),
  mean, na.rm = TRUE)
par(mfrow = c(1, 2))
barplot(OzMonthMean, col = 2:6, main = "Mean Ozone by month")
```



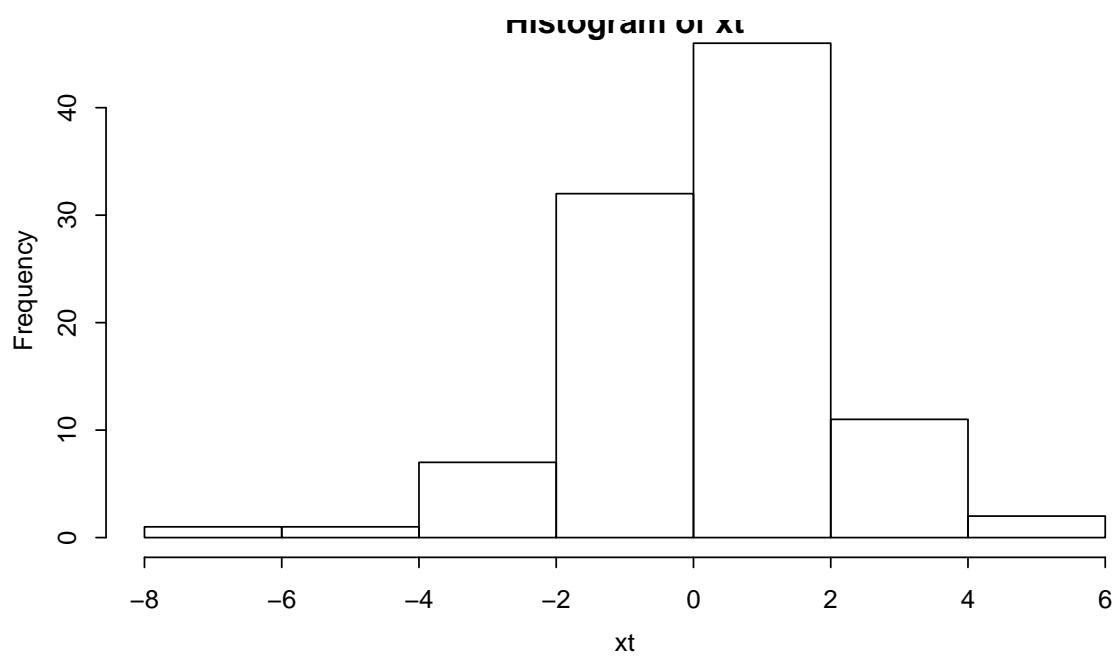
- pie chart

```
pie(OzMonthMean, col = rainbow(5))
```

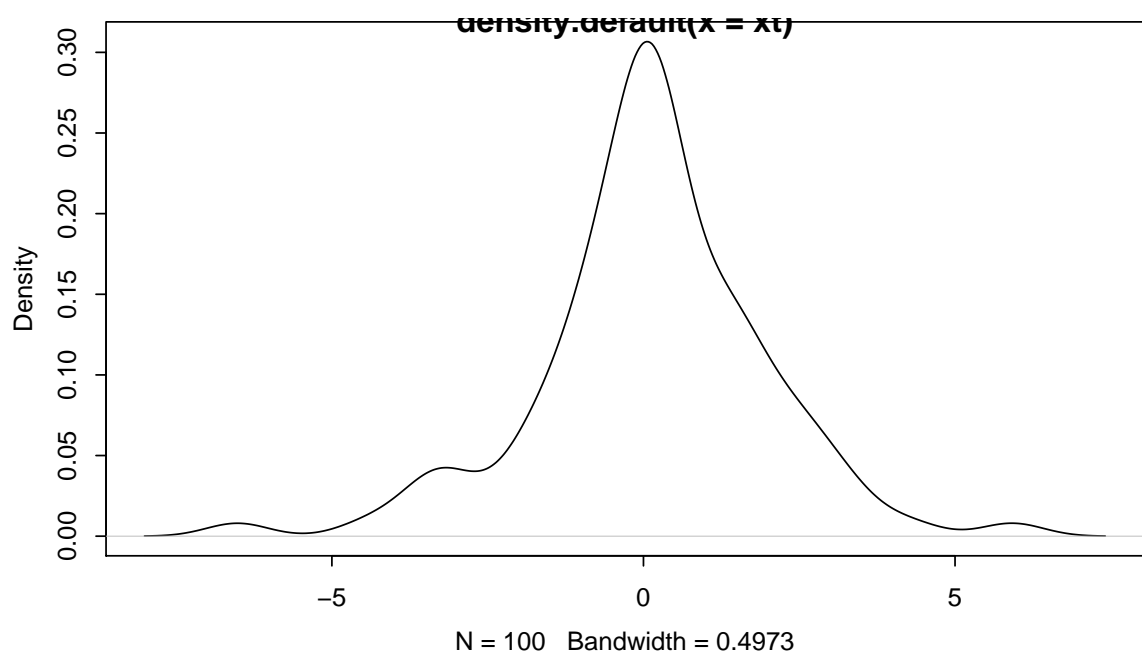


- *hist(x)*- histogram of a numeric vector *x* with a few important optional arguments: *nclass*= for the number of classes, and *breaks*= for the breakpoints

```
xt <- rt(100, 3)  
hist(xt)
```



```
plot(density(xt))
```



- 3D scatterplot

```

require(scatterplot3d)

## Loading required package: scatterplot3d

data(trees)
trees[1:2, ]

##      Girth Height Volume
## 1    8.3      70    10.3
## 2    8.6      65    10.3

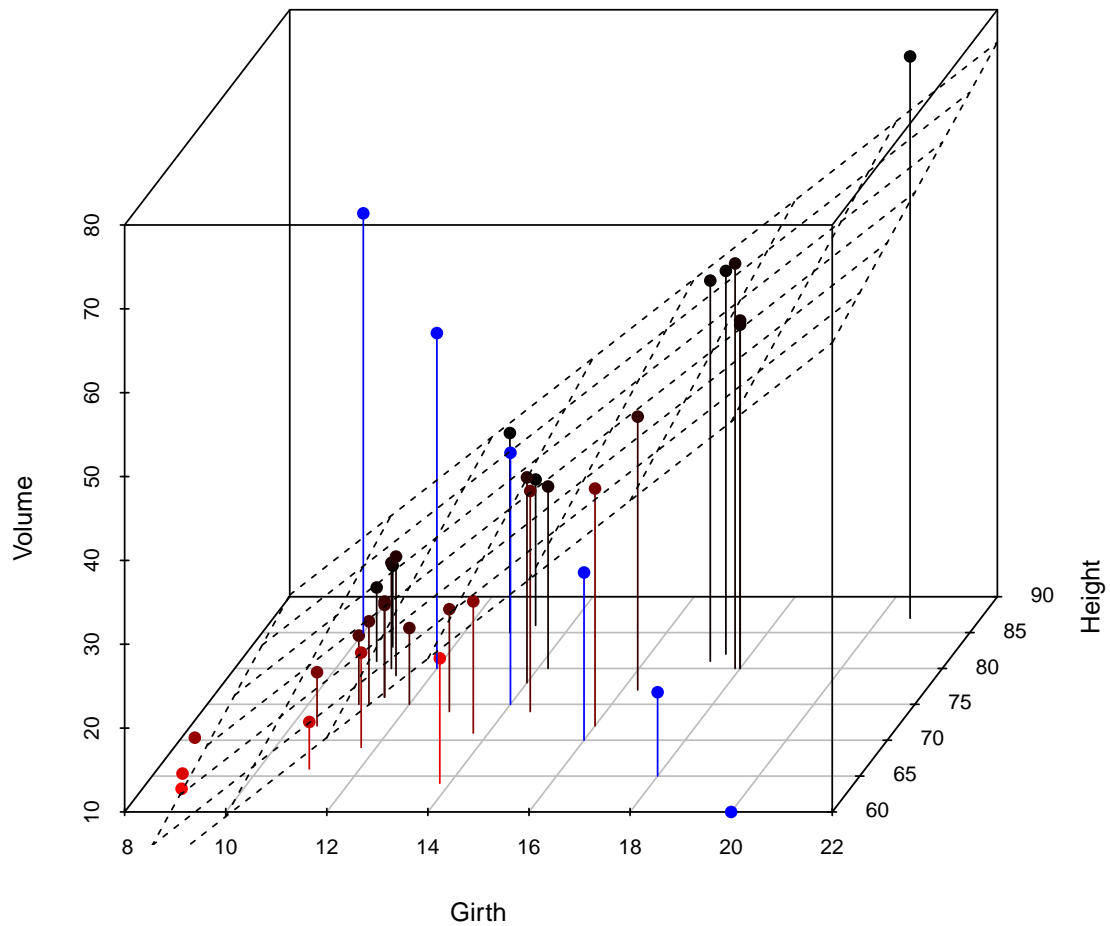
s3d <- scatterplot3d(trees, type = "h", highlight.3d = TRUE,
  angle = 55, scale.y = 0.7, pch = 16, main = "Example of
scatterplot3d plot: Tree Data")

# Now adding some points to the 'scatterplot3d'
s3d$points3d(seq(10, 20, 2), seq(85, 60, -5), seq(60, 10,
  -10), col = "blue", type = "h", pch = 16)

# Now adding a regression plane to the 'scatterplot3d'
attach(trees)
my.lm <- lm(Volume ~ Girth + Height)
s3d$plane3d(my.lm)

```

Example of scatterplot3d plot: Tree Data



```
detach(trees)
```

- **Rvenn** - draw a venn diagram. Input is a list. It will draw a venn diagram showing the intersect between 2-6 vectors in a list.

```
require(gplots)
sample(LETTERS, 10)

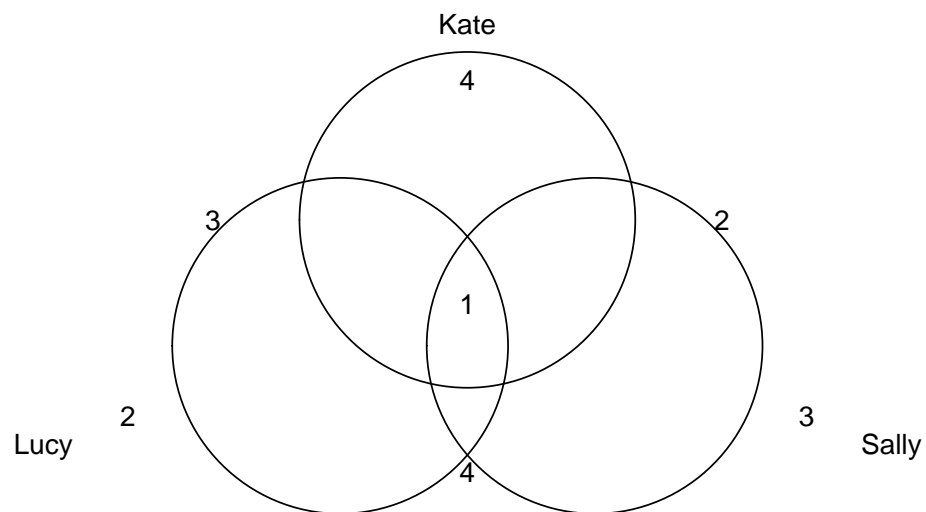
## [1] "Z" "R" "M" "U" "Q" "E" "L" "S" "J" "D"

tt <- lapply(1:3, function(x) sample(LETTERS, 10))
names(tt) <- c("Lucy", "Sally", "Kate")
tt
```



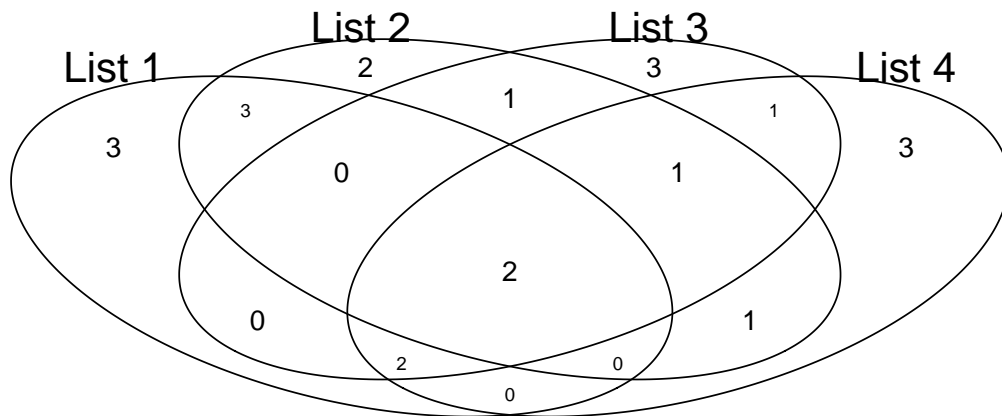
```
## $Lucy
## [1] "Q" "P" "Z" "M" "J" "T" "F" "O" "A" "D"
##
## $Sally
## [1] "B" "Z" "V" "K" "D" "F" "Y" "Q" "M" "S"
##
## $Kate
## [1] "Y" "P" "H" "N" "Q" "J" "V" "I" "O" "G"
##
```

```
venn(tt)
```



Plot 4 intersections

```
tt <- lapply(1:4, function(x) sample(LETTERS, 10))
names(tt) <- paste("List", 1:4)
venn(tt)
```



Color plots

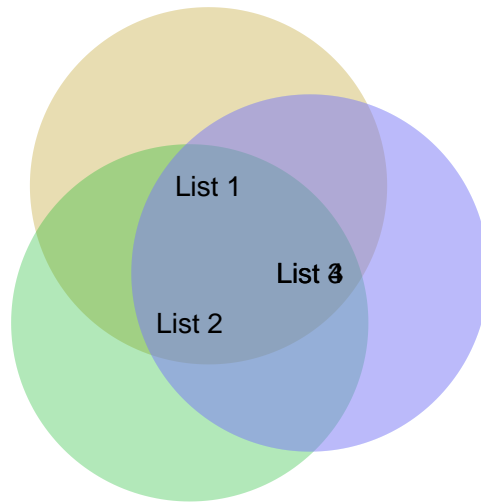
```
require(venneuler)

## Loading required package: venneuler

IntersectMatrix <- function(tt) {
  allElements <- unique(unlist(tt))
  outMat <- sapply(1:length(tt), function(i) allElements %in%
tt[[i]])
  rownames(outMat) <- allElements
  colnames(outMat) <- names(tt)
  return(outMat)
}
xx <- IntersectMatrix(tt)
print(xx[1:4, ])

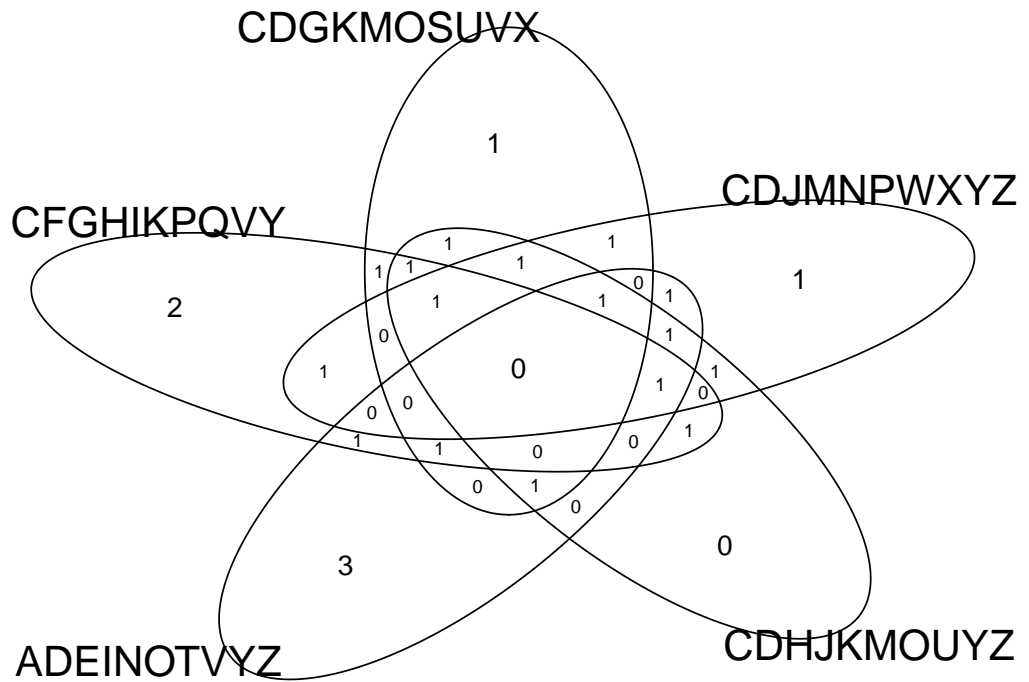
##   List 1 List 2 List 3 List 4
## H   TRUE  FALSE FALSE  FALSE
## G   TRUE  TRUE  TRUE   TRUE
## J   TRUE  FALSE TRUE   TRUE
## B   TRUE  FALSE FALSE  FALSE

plot(venneuler(xx))
```



It will even plot 5 intersections

```
tt <- lapply(1:5, function(x) sample(LETTERS, 10))  
names(tt) <- sapply(tt, function(x) paste(sort(x), collapse = ""))  
venn(tt)
```



5.3 Editing the default plot with low-level plotting commands

Sometimes the standard plot functions don't produce exactly the kind of plot you desire. In this case, low-level plotting commands can be used to add edit or extra information (such as points, lines or text) to the current plot. Some of the more useful low-level plotting functions are:

points(x, y)

lines(x, y) Adds points or connected lines to the current plot.

text(x, y, labels, ...) Add text to a plot at points given by x, y. Normally labels is an integer or character vector in which case labels[i] is plotted at point (x[i], y[i]). The default is 1:length(x). Note: This function is often used in the sequence

The graphics parameter type="n" suppresses the points but sets up the axes, and the `text()` function supplies special characters, as specified by the character vector names for the points.

abline(a, b) Adds a line of slope b and intercept a to the current plot.

abline(h=y) Adds a horizontal line

abline(v=x) Adds a vertical line

polygon(x, y, ...) Draws a polygon defined by the ordered vertices in (x, y) and (optionally) shade it in with hatch lines, or fill it if the graphics device allows the filling of figures.

legend(x, y, legend, ...) Adds a legend to the current plot at the specified position. Plotting characters, line styles, colors etc., are identified with the labels in the character vector legend. At least one other argument v (a vector the same length as legend) with the corresponding values of the plotting unit must also be given, as follows:

legend(, fill=v) Colors for filled boxes

legend(, col=v) Colors in which points or lines will be drawn

legend(, lty=v) Line styles

legend(, lwd=v) Line widths

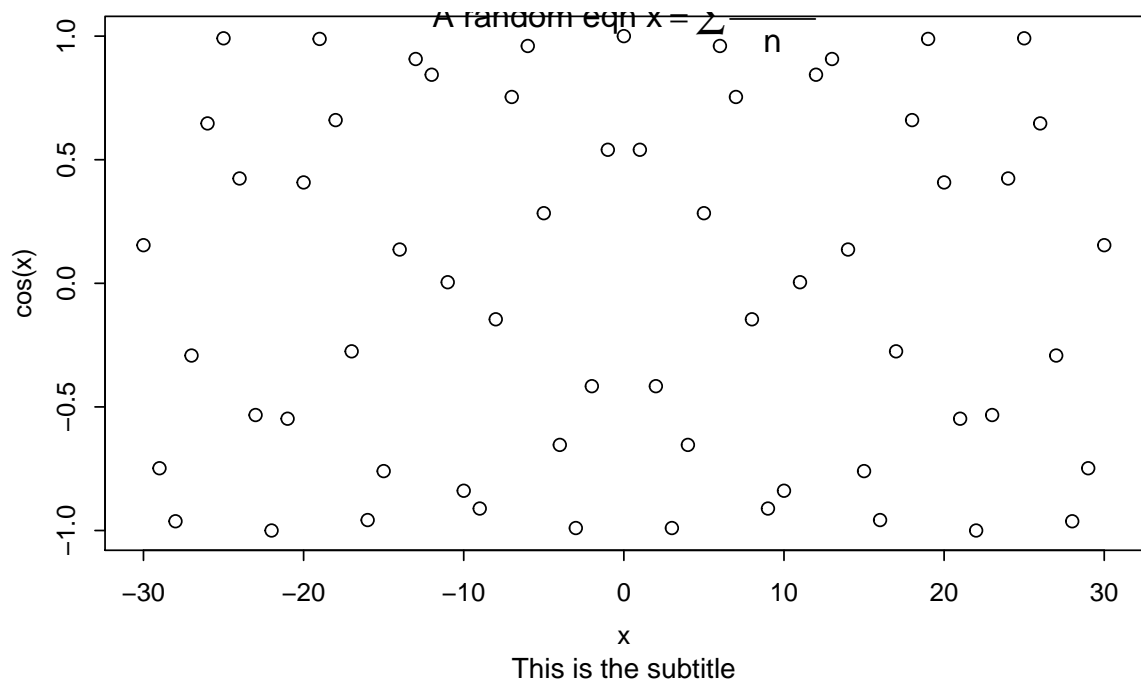
legend(, pch=v) Plotting characters

title(main, sub) Adds a title main to the top of the current plot in a large font and (optionally) a sub-title sub at the bottom in a smaller font.

axis(side, ...) Adds an axis to the current plot on the side given by the first argument (1 to 4, counting clockwise from the bottom.) Other arguments control the positioning of the axis within or beside the plot, and tick positions and labels. Useful for adding custom axes after calling plot() with the axes=FALSE argument.

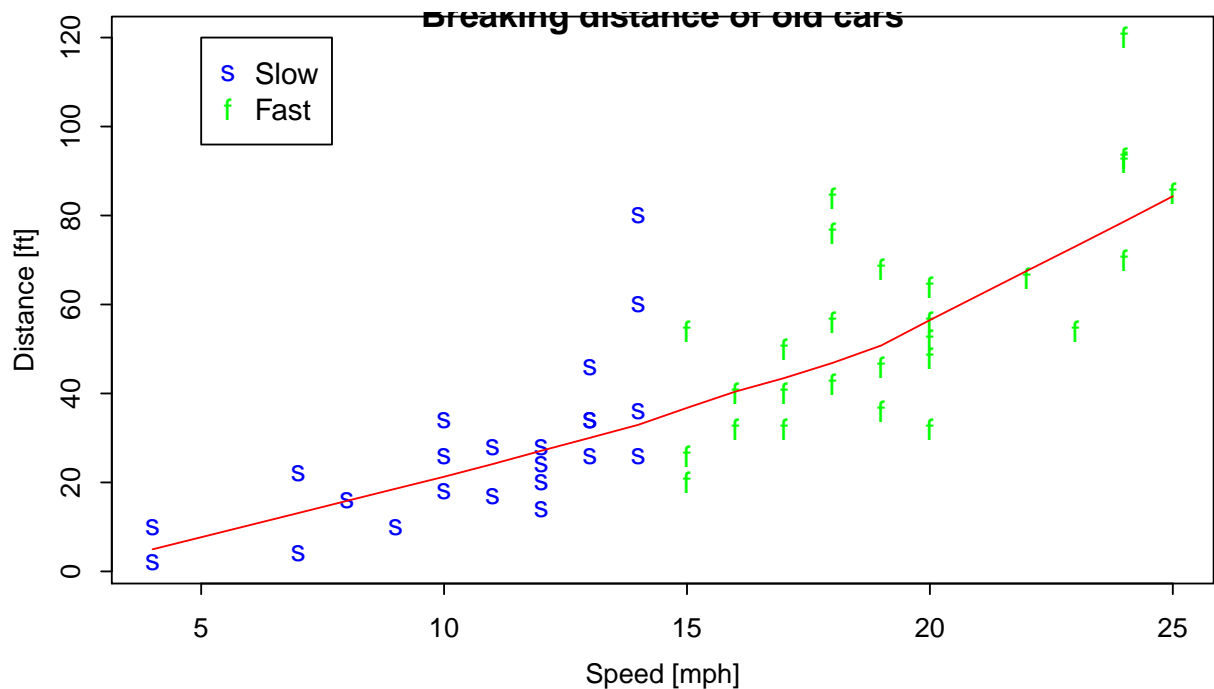
To add Greek characters, either specify font type 5 (see below) or use the function *expression*

```
plot(x, cos(x), main = expression(paste("A random eqn ", bar(x)) ==  
    sum(frac(alpha[i] + beta[z], n))), sub = "This is the  
    subtitle")
```



Example using points lines and legend

```
attach(cars)
plot(cars, type = "n", xlab = "Speed [mph]", ylab = "Distance [ft]")
points(speed[speed < 15], dist[speed < 15], pch = "s", col = "blue")
points(speed[speed >= 15], dist[speed >= 15], pch = "f", col = "green")
lines(lowess(cars), col = "red")
legend(5, 120, pch = c("s", "f"), col = c("blue", "green"),
      legend = c("Slow", "Fast"))
title("Breaking distance of old cars")
```

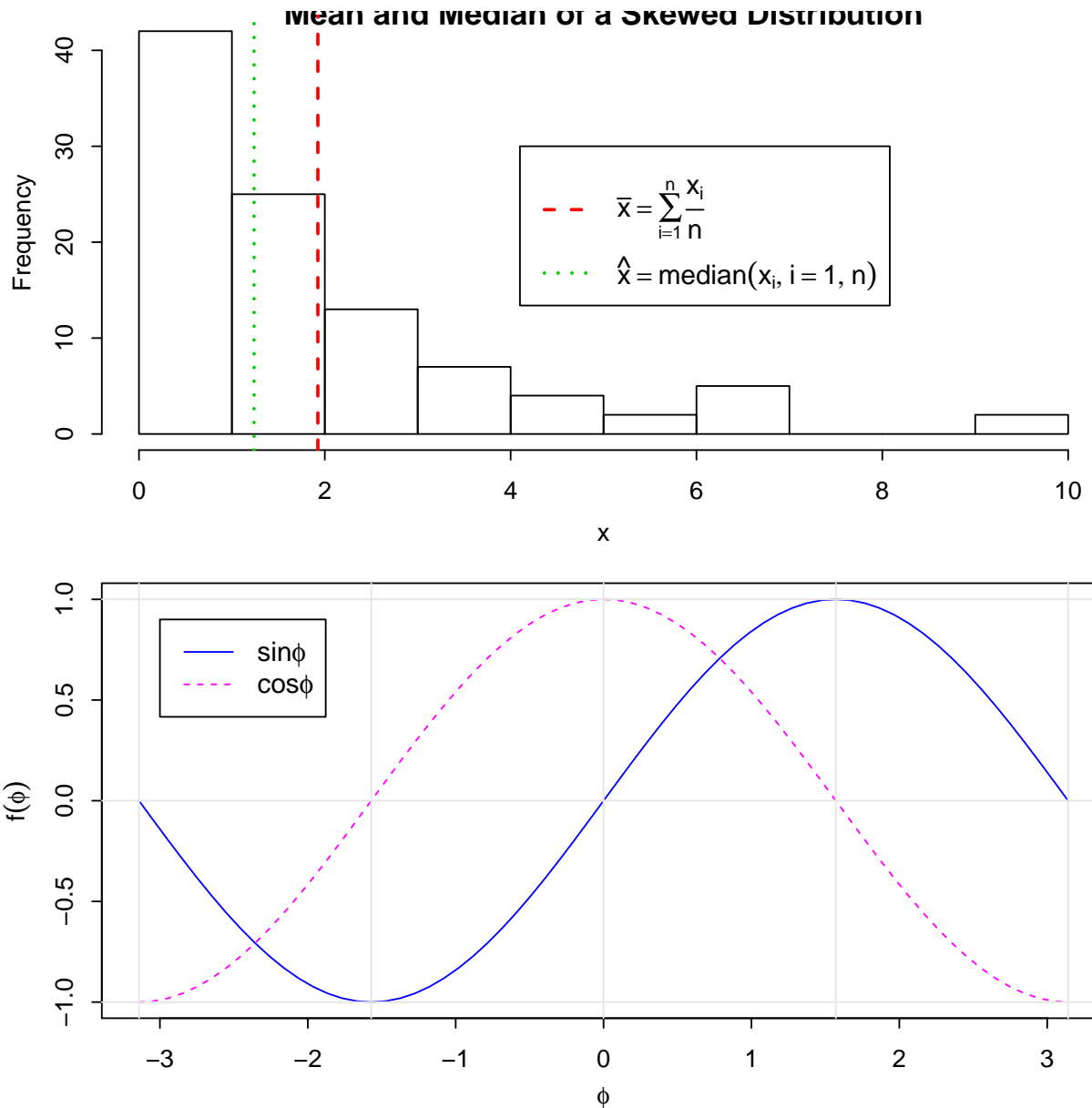


```
detach(2)
```

To add formulae or Greek characters to a plot

```
par(mfrow = c(2, 1))
# Mean and Median Plot
x <- rexp(100, rate = 0.5)
hist(x, main = "Mean and Median of a Skewed Distribution")
abline(v = mean(x), col = 2, lty = 2, lwd = 2)
abline(v = median(x), col = 3, lty = 3, lwd = 2)
ex1 <- expression(bar(x) == sum(over(x[i], n), i == 1, n),
  hat(x) == median(x[i], i == 1, n))
legend(4.1, 30, ex1, col = 2:3, lty = 2:3, lwd = 2)

x <- seq(-pi, pi, len = 65)
plot(x, sin(x), type = "l", col = "blue", xlab = expression(phi),
  ylab = expression(f(phi)))
lines(x, cos(x), col = "magenta", lty = 2)
abline(h = -1:1, v = pi/2 * (-6:6), col = "gray90")
ex2 <- expression(plain(sin) * phi, paste("cos", phi))
legend(-3, 0.9, ex2, lty = 1:2, col = c("blue", "magenta"),
  adj = c(0, 0.6))
```



5.4 Default parameters - par

When creating graphics, particularly for presentation or publication purposes, R's defaults do not always produce exactly that which is required. You can, however, customize almost every aspect of the display using graphics parameters. R maintains a list of a large number of graphics parameters which control things such as line style, colors, figure arrangement and text justification among many others. Every graphics parameter has a name (such as 'col', which controls colors,) and a value (a color number, for example.) Graphics parameters can be set in two ways: either permanently, affecting all graphics functions which access the current device; or temporarily, affecting only a single graphics function call.

The `par()` function is used to access and modify the list of graphics parameters for the current graphics

device. See help on *par()* for more details.

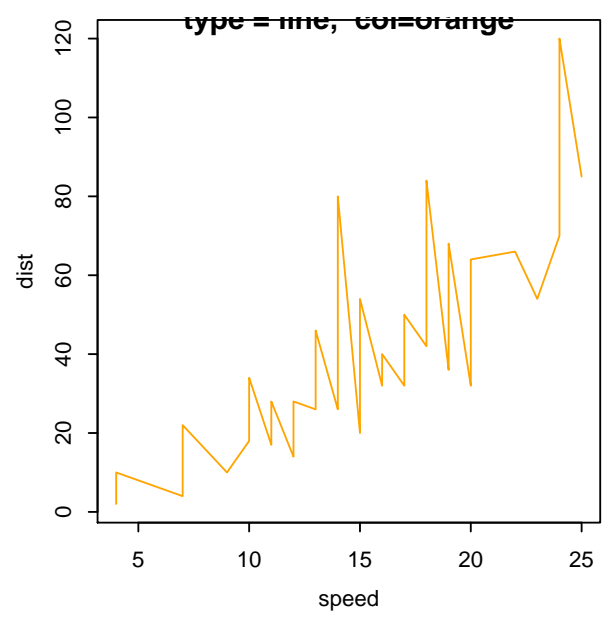
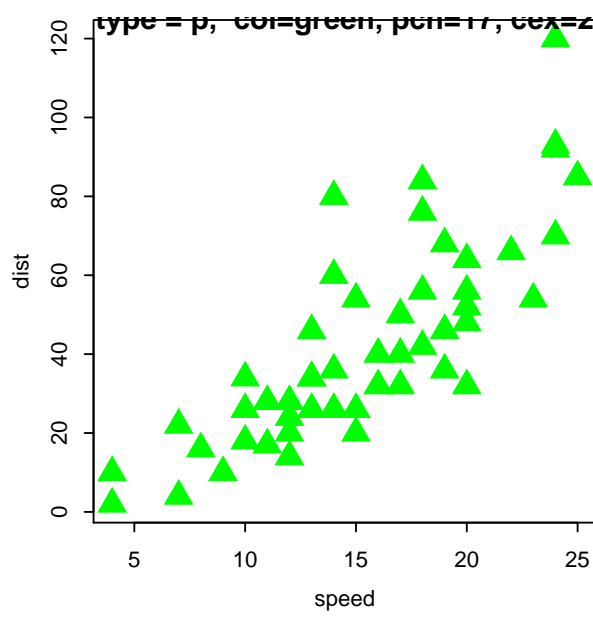
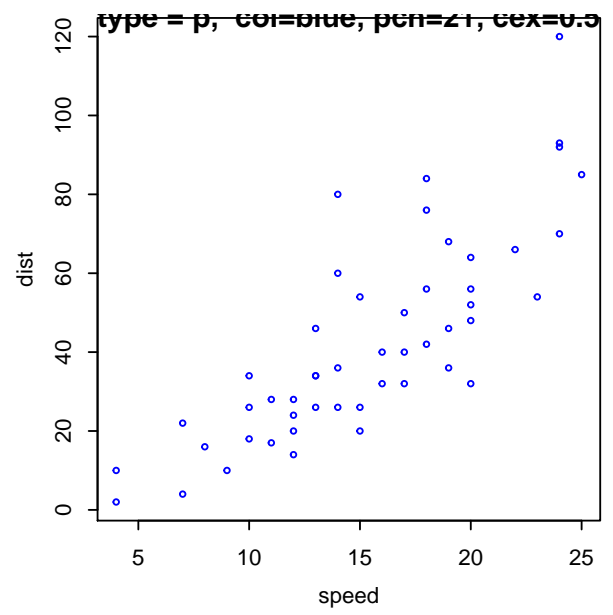
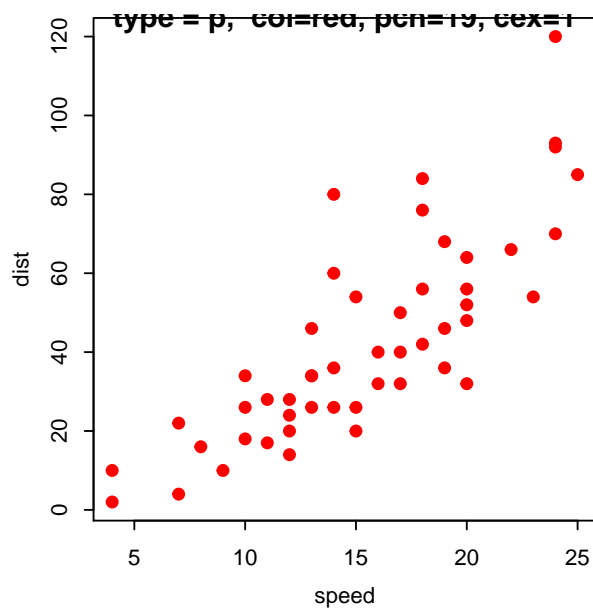
To see a sample of point type available in R, type

```
example(pch)
```

5.4.1 Interactive plots in R Studio - Effect of changing par

In RStudio the `manipulate` function accepts a plotting expression and a set of controls (e.g. slider, picker, or checkbox) which are used to dynamically change values within the expression. When a value is changed using its corresponding control the expression is automatically re-executed and the plot is redrawn.

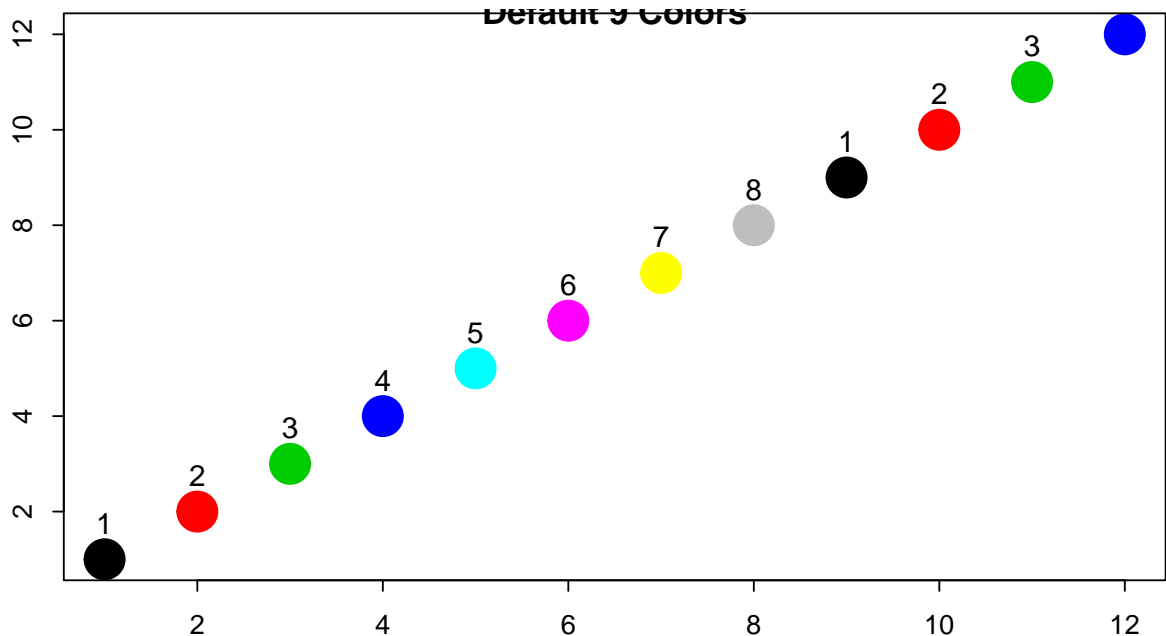
```
library(manipulate)
manipulate(plot(1:x), x = slider(1, 100))
manipulate(plot(cars, xlim = c(0, x.max), type = type, ann = label,
  col = col, pch = pch, cex = cex), x.max = slider(10, 25, step = 5,
  initial = 25), type = picker(Points = "p", Line = "l", Step = "s"),
  label = checkbox(TRUE, "Draw Labels"), col = picker(red = "red",
    green = "green", yellow = "yellow"), pch = picker(`1` = 1, `2` = 2,
    `3` = 3, `4` = 4, `5` = 5, `6` = 6, `7` = 7, `8` = 8, `9` = 9,
    `10` = 10, `11` = 11, `12` = 12, `13` = 13, `14` = 14, `15` = 15,
    `16` = 16, `17` = 17, `18` = 18, `19` = 19, `20` = 20, `21` = 21,
    `22` = 22, `23` = 23, `24` = 24), cex = picker(`1` = 1, `2` = 2,
    `3` = 3, `4` = 4, `5` = 5, `6` = 6, `7` = 7, `8` = 8, `9` = 9,
    `10` = 10))
```



5.4.2 R Colors

Thus far, we have frequently used numbers in plot to refer to a simple set of colors. There are 8 colors where 0:8 are white, black, red, green, blue, cyan, magenta, yellow and grey. If you provide a number greater than 8, the colors are recycled. Therefore for plots where other or greater numbers of colors are required, we need to access a larger palette of colors.

```
plot(1:12, col = 1:12, main = "Default 9 Colors", ylab = "",
     xlab = "", pch = 19, cex = 3)
text(1:12, c(1:12) + 0.75, c(1:8, 1:4))
```



R has a large list of over 650 colors that R knows about. This list is held in the vector `colors()`. Have a look at this list, and maybe search for a set you are interested in.

```
colors()[1:10]
```

```
## [1] "white"          "aliceblue"      "antiquewhite"   "antiquewhite1"
## [5] "antiquewhite2"  "antiquewhite3"  "antiquewhite4"  "aquamarine"
## [9] "aquamarine1"    "aquamarine2"
```

```
length(colors())
```

```
## [1] 657
```

```
grep("yellow", colors(), value = TRUE)
```

```
## [1] "greenyellow"          "lightgoldenrodyellow"
## [3] "lightyellow"          "lightyellow1"
```

```
## [5] "lightyellow2"      "lightyellow3"
## [7] "lightyellow4"      "yellow"
## [9] "yellow1"           "yellow2"
## [11] "yellow3"           "yellow4"
## [13] "yellowgreen"
```

R has defined palettes of colors, which provide complementing or contrasting color sets. For example look at the color palette rainbow.

```
example(rainbow)
```

For a more complete listing of colors, along with the RGB numbers for each color, the following script generates a several page pdf document which maybe a useful reference document for you.

```
source("http://research.stowers-institute.org/efg/R/Color/Chart/ColorChart.R")
```

A very useful **RColorBrewer** <http://colorbrewer.org>. This package will generate a ramp color to provide color palettes that are sequential, diverging, and qualitative ramped, for example:

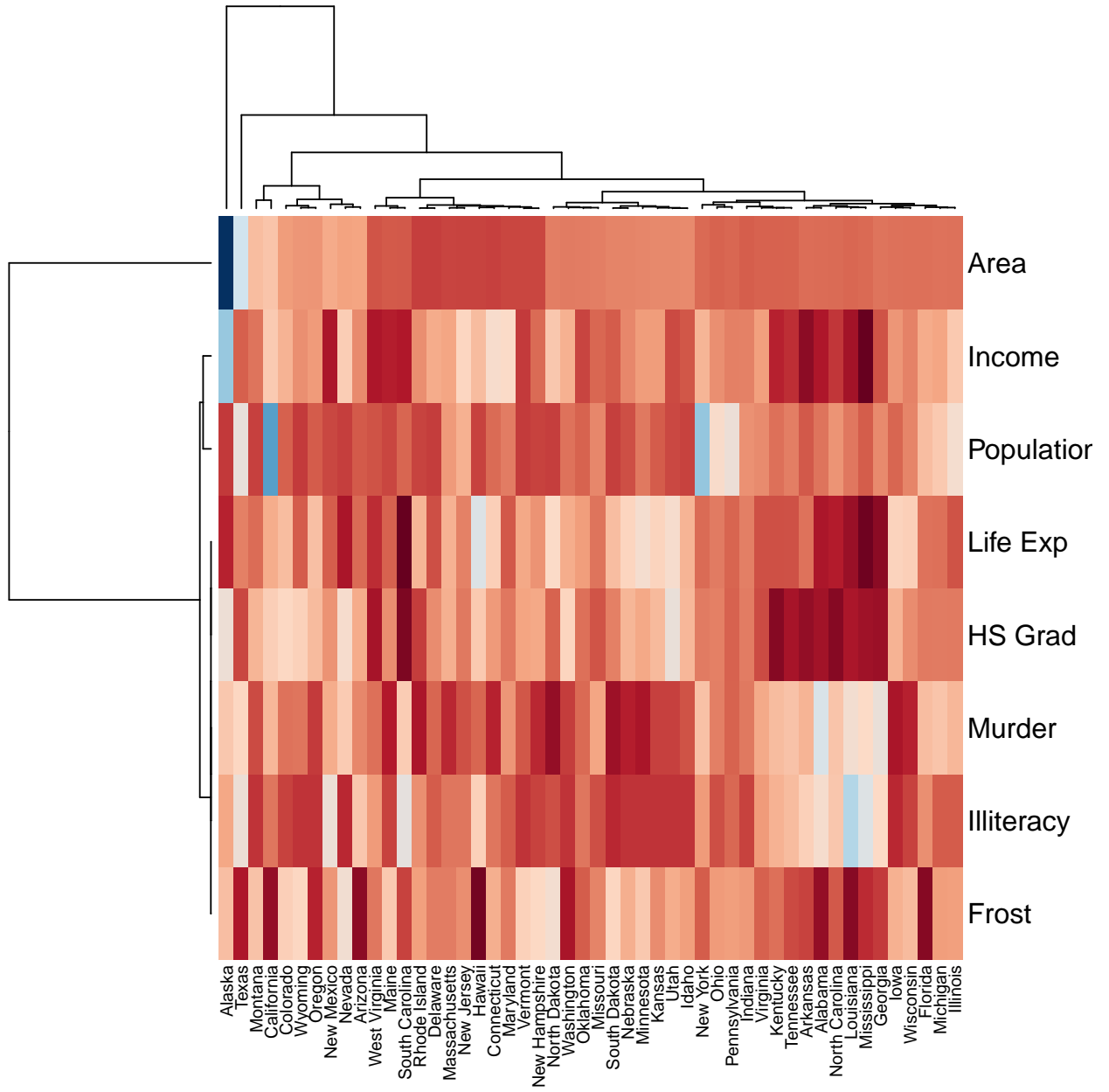
- Sequential palettes are suited to ordered data that progress from low to high. Lightness steps dominate the look of these schemes, with light colors for low data values to dark colors for high data values.
- Diverging palettes put equal emphasis on mid-range critical values and extremes at both ends of the data range. The critical class or break in the middle of the legend is emphasized with light colors and low and high extremes are emphasized with dark colors that have contrasting hues.
- Qualitative palettes do not imply magnitude differences between legend classes, and hues are used to create the primary visual differences between classes. Qualitative schemes are best suited to representing nominal or categorical data.

To see more about RColorBrewer run the example

```
library(RColorBrewer)
example(brewer.pal)
```

I use RColorBrewer to produce nicer colors in clustering heatmaps. For example if we look at the US state fact and figure information in the package **state**, which contains a matrix called **state.x77** containing information on 50 US states (50 rows) on population, income, Illiteracy, life expectancy, murder, high school graduation, number of days with frost, and area (8 columns). The default clustering of this uses a rather ugly red-yellow color scheme which I changed to a red/brown-blue.

```
library(RColorBrewer)
hmccl <- colorRampPalette(brewer.pal(10, "RdBu"))(500)
heatmap(t(state.x77), col = hmccl, scale = "row")
```

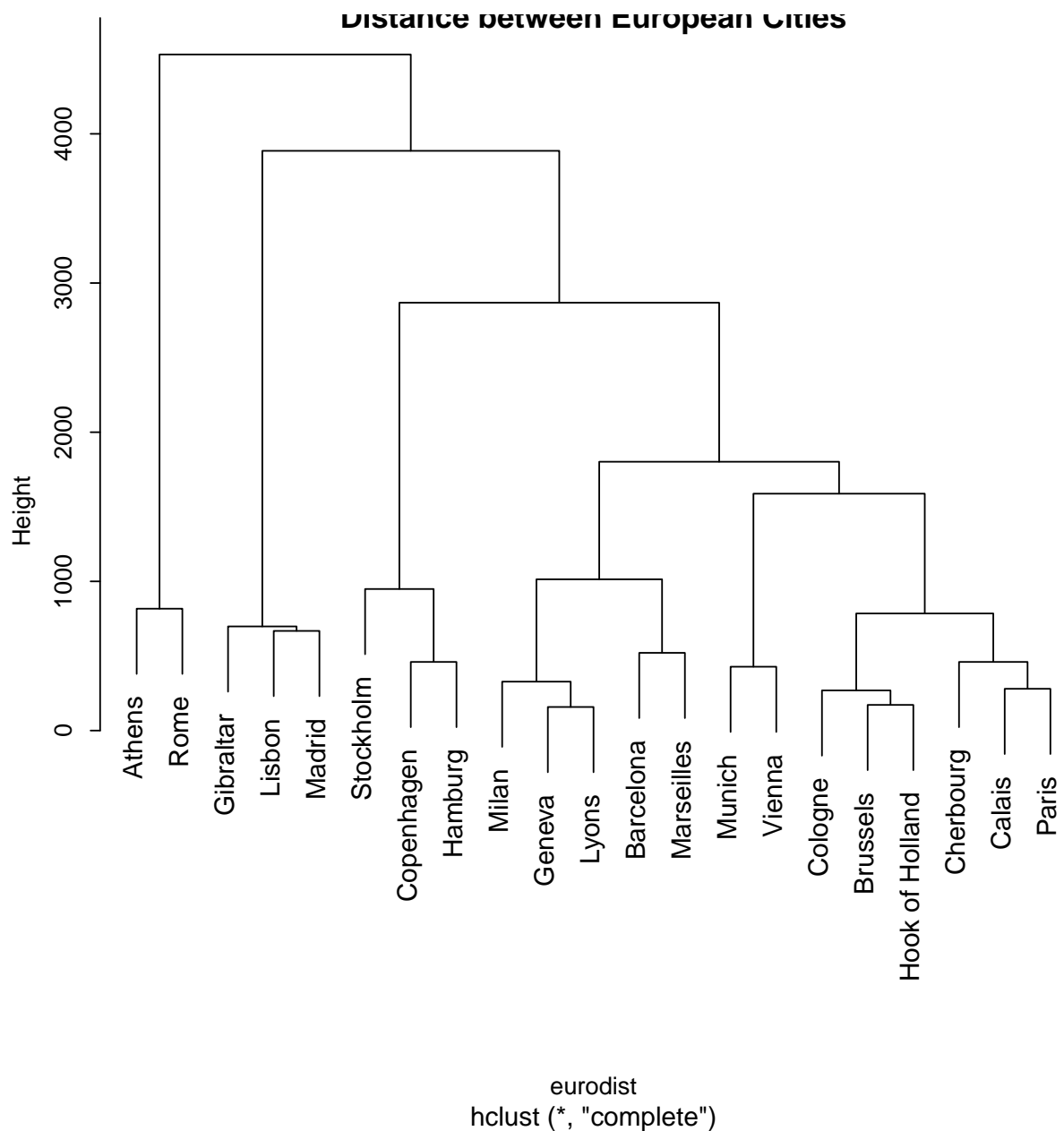


5.5 Interacting with graphics

R also provides functions which allow users to extract or add information to a plot using a mouse via *locator()* and *verb+identify()* functions respectively.

Identify members in a hierarchical cluster analysis of distances between European cities

```
hca <- hclust(eurodist)
plot(hca, main = "Distance between European Cities")
```



```
(x <- identify(hca))
x
```

```
plot(1:20, rt(20, 1))  
text(locator(1), "outlier", adj = 0)
```

Waits for the user to select locations on the current plot using the left mouse button.

```
attach(women)  
plot(height, weight)  
identify(height, weight, women)  
detach(2)
```

Allow the user to highlight any of the points (`identify(x, y, label)`) defined by `x` and `y` (using the left mouse button) by plotting the corresponding component of labels nearby (or the index number of the point if labels is absent).

Right mouse click, to "stop".

5.5.1 Exercise 8 - Plotting

Using the women dataset

1. Set the plot layout to be a 2 x 2 grid (ie 2 rows, 2 columns)
2. Draw weight on the Y axis and height on the X axis.
3. Switch the orientation, Draw weight on the X axis and height on the Y axis.
4. Drawing a new plot, set the pch (point type) to be a solid circle, and color them red. Add a title "study of Women" to the plot
5. Drawing another plot, set the pch (point type) to be a solid square, Change the X axis label to be "Weight of Women" and make the point size (using the parameter cex) larger to 1.5

5.6 Saving plots

5.6.1 Rstudio

In RStudio, there is a simple interface to export plots. Click on the "Export" button in the plot window.

5.6.2 Devices

R can generate graphics (of varying levels of quality) on almost any type of display or printing device. Before this can begin, however, R needs to be informed what type of device it is dealing with. This is done by starting a device driver. The purpose of a device driver is to convert graphical instructions from R ("draw a line," for example) into a form that the particular device can understand. Device drivers are started by calling a device driver function. There is one such function for every device driver: type *help(Devices)* for a list of them all.

The most useful formats for saving R graphics:

postscript() For printing on PostScript printers, or creating PostScript graphics files.

pdf() Produces a PDF file, which can also be included into PDF files.

jpeg() Produces a bitmap JPEG file, best used for image plots.

5.6.3 Difference between vector and pixel images

Note there is a big difference between saving files in jpeg or postscript files. Image files save in jpeg, bmp, gif etc are pixel image files, these are like photographs, where you can just select a line and change its color. By contrast vector graphic, such as postscript, or windows meta files can be imported into drawing packages such as Adobe Illustrator (or some even into PowerPoint), you can double click on an axes, and since it's a vector graphic you can change the color of the line easily.

Format *	Type	Description (name)	Designed for
TIFF, TIF	image	Tagged Image File Format	High resolution printing of images, even to postscript printers
PNG	image	Portable network graphic	High resolution bitmap image, Screen display, printing
BMP	image	bitmap image	Screen display under Windows
GIF	image	Graphic Interchange Format	Screen display especially online images/Web
JPEG, JPG	image	Joint Photographic Experts Group	Screen display especially online images/Web
EPS, PS	vector	(Encapsulated) postscript	High resolution printing of illustrations, Printing to PostScript printers/Imagesetters
PDF	vector	Portable Document File	High resolution printing of illustrations, Printing to PostScript/PDF printers/Imagesetters
EMF, WMF **	vector	(Enhanced) Windows Metafile	Screen display under Windows printing to non-PostScript printer

For more information on image file format see http://en.wikipedia.org/wiki/Image_file_formats

** EMF files are a vector like files that can be inserted into PowerPoint. To insert an EMF image in a PowerPoint slide, click on Insert-Picture-From File and locate the file. Click OK. This will add the EMF file to your page. Right mouse click on the image to "ungroup", now you can select lines/points to change colors/widths etc.

When in doubt, I save files in postscript format (eps), as several journals request this format. EPS files can be open directly in Adobe Illustrator or other vector editing graphics packages.

In R, to save the current image to file. Either use the file menu File -> Save as. Or use the functions *dev2bitmap*, *dev.copy2eps* or *dev.copy(device, file)*, where device can be one of *png*, *jpeg* or *pdf* and file is your filename. For example:

```
plot(1:10, col = "red", pch = 19)
dev.copy(png, file = "test.png")
dev.off()
```

```
plot(1:10, col = "red", pch = 19)
dev.copy(pdf, file = "test.pdf")
dev.off()
```

To find out more about the image formats that can be saved in R, see the help on *?Devices*.

If you wish to write an image directly to a file, without "seeing" the plot screen (called X11 or Quartz depending on the operating system). Use the functions *pdf()*, *postscript()*, *jpeg()* with the syntax:

```
pdf(file = "myplot.pdf")
plot(1:10, col = "blue", xlab = "X axis", ylab = "Y axis")
dev.off()
```

Remember it is very important to type *dev.off* in order to properly save the file

To list the current graphics devices that are open use *dev.cur*. When you have finished with a device, be sure to terminate the device driver by issuing the command *dev.off()*.

If you have open a device to write to for example *pdf* or *png*, *dev.off* will ensure that the device finishes cleanly; for example in the case of hardcopy devices this ensures that every page is completed and has been sent to the printer or file.

Example:

```
myPath <- file.path("P:/Bio503/Plots")
pdf(file = paste(myPath, "nicePlot.pdf", sep = ""))
x <- seq(0, 2 * pi, length = 100)
y <- sin(3 * x) + cos(x) + rnorm(100, sd = 0.2)
plot(x, y)
dev.off()
```

5.7 Useful Graphics Resources

If you have plots saved in a non-vector format, we have found the web-site VectorMagic from Stanford <http://vectormagic.stanford.edu/> to be very useful. It will convert bmp or jpeg files to vector format.

The free software ImageMagick <http://www.imagemagick.org> can be downloaded and is also useful for converting between image format.

Chapter 6

Advanced Graphics

6.1 Advanced plotting using Trellis; ggplots2, Lattice

One of the strengths of R is the variety and quality of its graphics capabilities. Both Lattice and ggplots2 offer trellis (layered graphics) which are both prettier and much more flexible than basic R plotting. Between these two packages, there is no clear winner, but like Lattice other say ggplots2 is more flexible. But it is worth investigating these packages if you wish to generate nice R graphics.

6.1.1 ggplots2

qplot is the basic plotting function in the ggplot2 package and is a convenient wrapper for creating a number of different types of plots using a consistent calling scheme. See <http://had.co.nz/ggplot2/book/qplot.pdf> for the chapter in the ggplot2 book which describes the usage of qplot in detail.

A nice introduction to ggplots is written by its author Hadley Wickham and is available from http://www.ceb-institute.org/bbs/wp-content/uploads/2011/09/handout_ggplot2.pdf.

The following examples are taken from that tutorial

Basic "Quick Plot" aka qplot in ggplots2

```
require("ggplot2")

## Loading required package: ggplot2

data(mtcars)
head(mtcars)
```

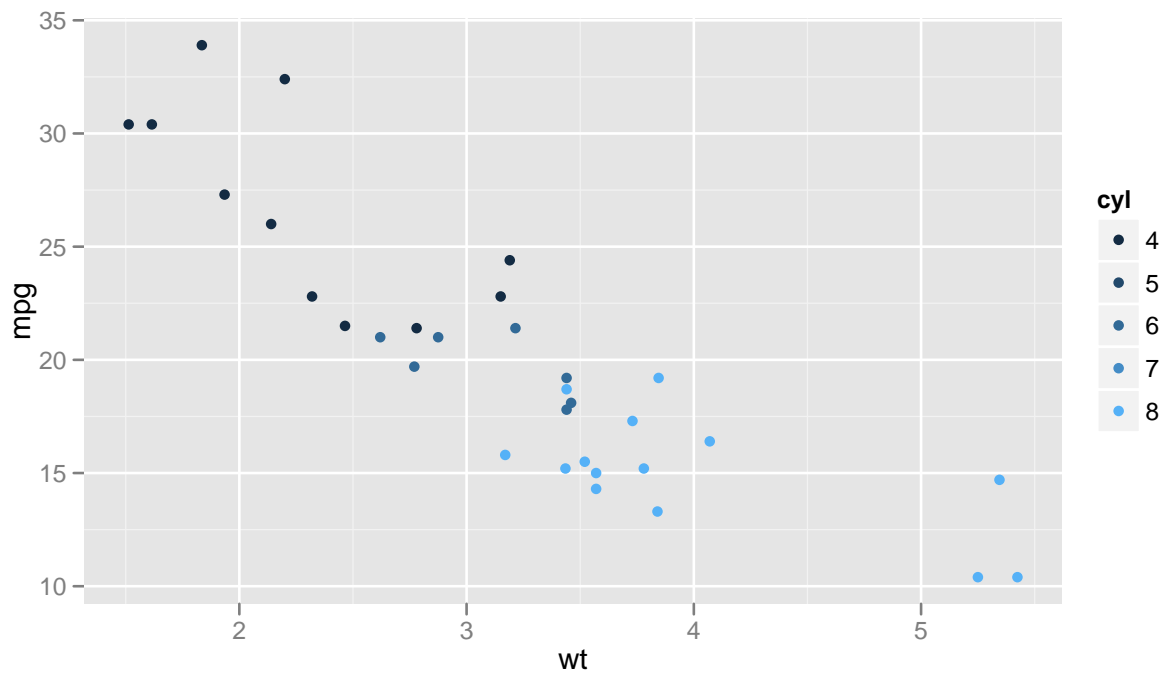
	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear
## Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4
## Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4
## Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4
## Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3
## Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3
## Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3
## carb										
## Mazda RX4		4								
## Mazda RX4 Wag		4								
## Datsun 710		1								

```
## Hornet 4 Drive      1
## Hornet Sportabout  2
## Valiant             1

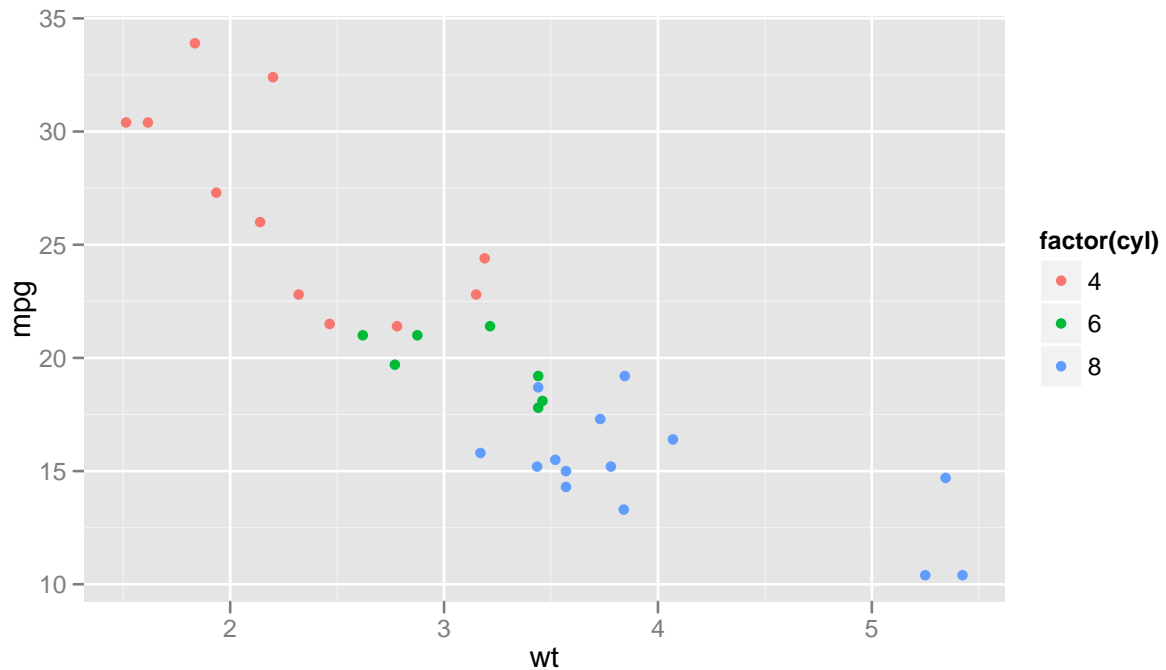
levels(mtcars$cyl)

## NULL

qplot(wt, mpg, data = mtcars, colour = cyl)
```



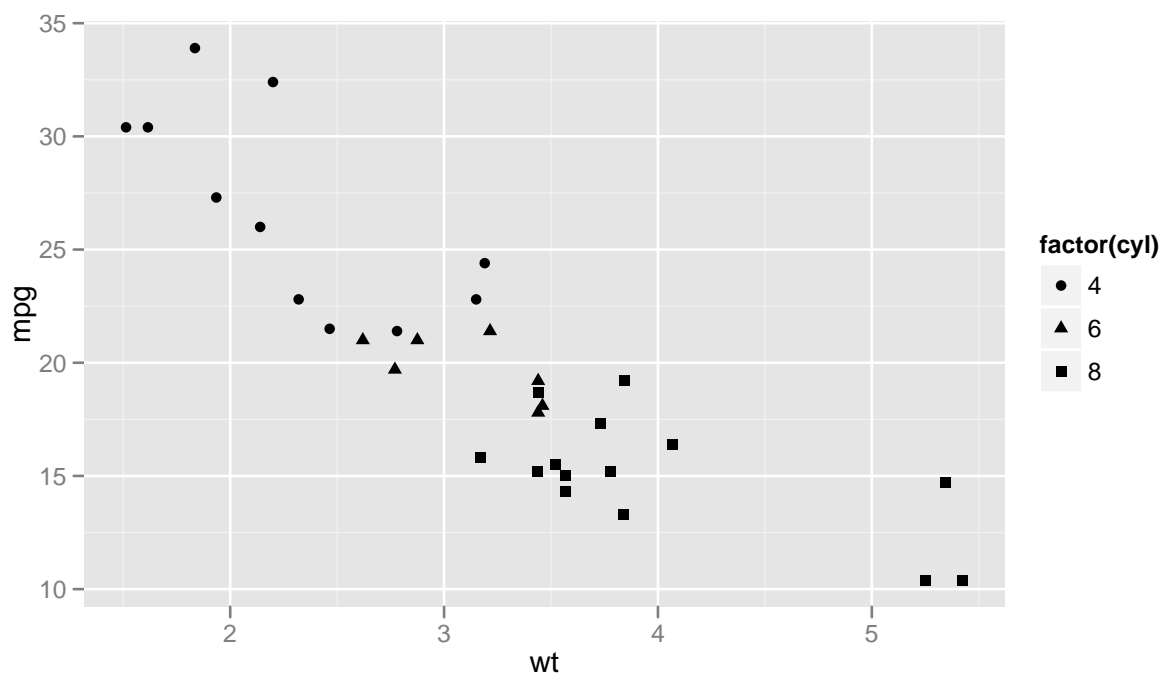
```
qplot(wt, mpg, data = mtcars, colour = factor(cyl))
```



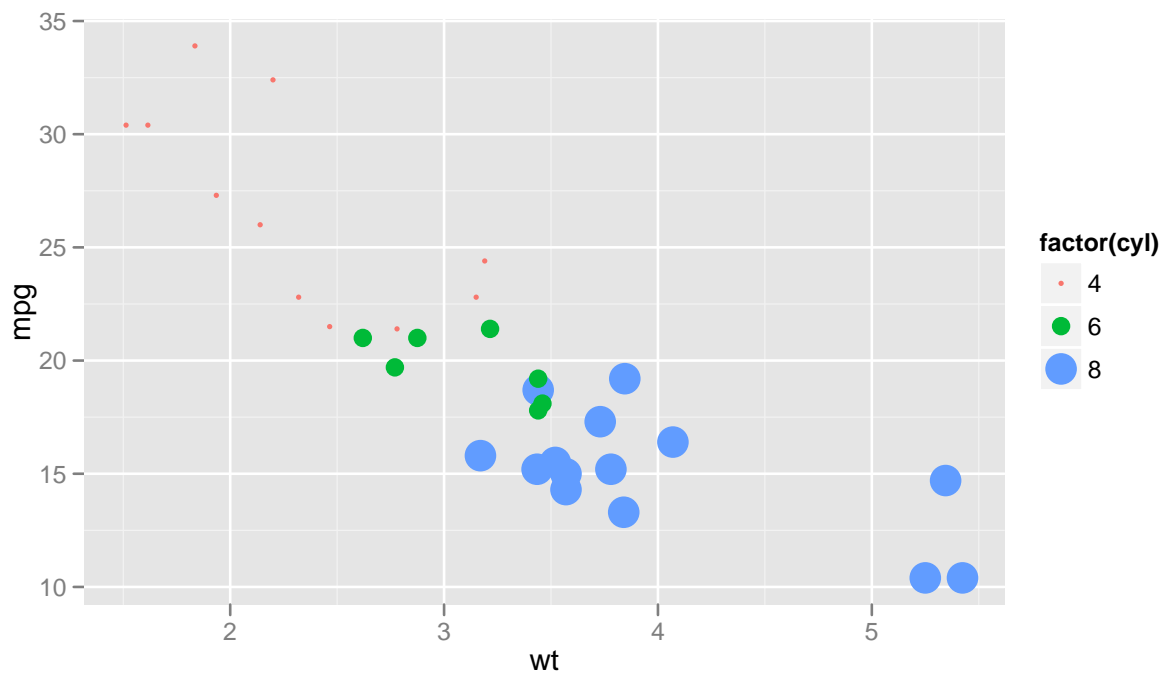
mtcars is a dataset from 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance (eg miles/gallon, number of cylinders, displacement, gross horsepower, weight, seconds to complete a quarter mile, etc) for 32 automobiles. In the above plot we see a plot of weight x miles per gallon given the number of cylinders in the car.

In the above plot, we view cylinder by color, but it could also be by shape or size

```
ggplot(wt, mpg, data = mtcars, shape = factor(cyl))
```



```
qplot(wt, mpg, data = mtcars, size = factor(cyl), colour = factor(cyl))
```



The function `mflow` and `layout` don't work with `ggplots2`, so here is a little script to make layout of multiple plots using `ggplots2` (acknowledgement to Stephen Turner). First assign each `ggplot2` plots to an object, and then use the `arrange` function to display two or more.

```
arrange <- function(..., nrow = NULL, ncol = NULL, as.table = FALSE) {
  vp.layout <- function(x, y) viewport(layout.pos.row = x,
layout.pos.col = y)
  dots <- list(...)
  n <- length(dots)
  if (is.null(nrow) & is.null(ncol)) {
    nrow <- floor(n/2)
    ncol <- ceiling(n/nrow)
  }
  if (is.null(nrow)) {
    nrow <- ceiling(n/ncol)
  }
  if (is.null(ncol)) {
    ncol <- ceiling(n/nrow)
  }
  ## NOTE see n2mfrow in grDevices for possible alternative

  grid.newpage()
  pushViewport(viewport(layout = grid.layout(nrow, ncol)))
```

```

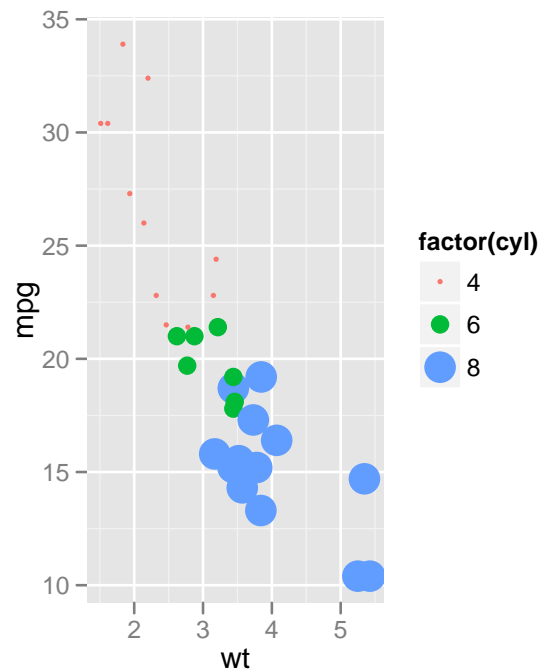
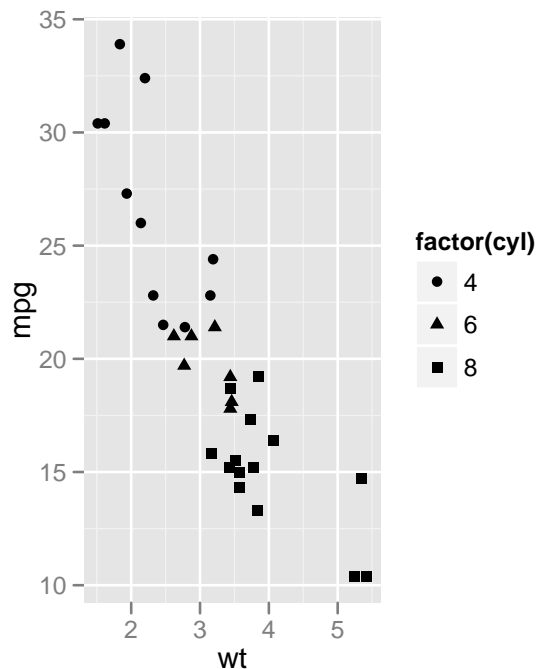
ii.p <- 1
for (ii.row in seq(1, nrow)) {
  ii.table.row <- ii.row
  if (as.table) {
    ii.table.row <- nrow - ii.table.row + 1
  }
  for (ii.col in seq(1, ncol)) {
    ii.table <- ii.p
    if (ii.p > n)
      break
    print(dots[[ii.table]], vp = vp.layout(ii.table.row,
ii.col))
    ii.p <- ii.p + 1
  }
}

```

```

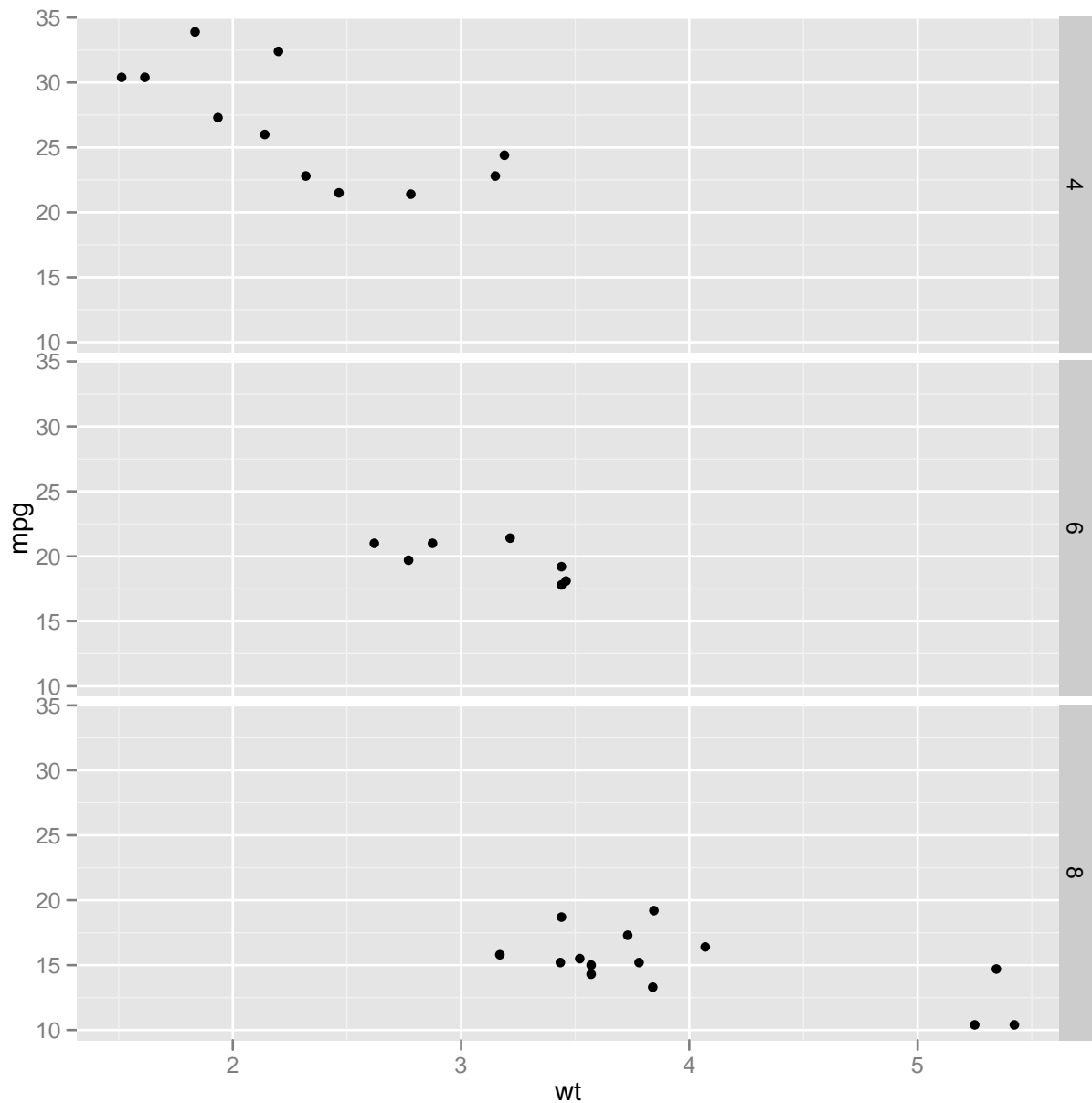
p1 <- qplot(wt, mpg, data = mtcars, shape = factor(cyl))
p2 <- qplot(wt, mpg, data = mtcars, size = factor(cyl), colour =
factor(cyl))
# Arrange and display the plots into a 2x1 grid
arrange(p1, p2, nrow = 1)

```



Using Facets to plot several plots

```
ggplot(wt, mpg, data = mtcars, facets = cyl ~ .)
```



More complex Facets to view cross-tabulated categories. For example you would expect a strong interaction between cylinder and horsepower.

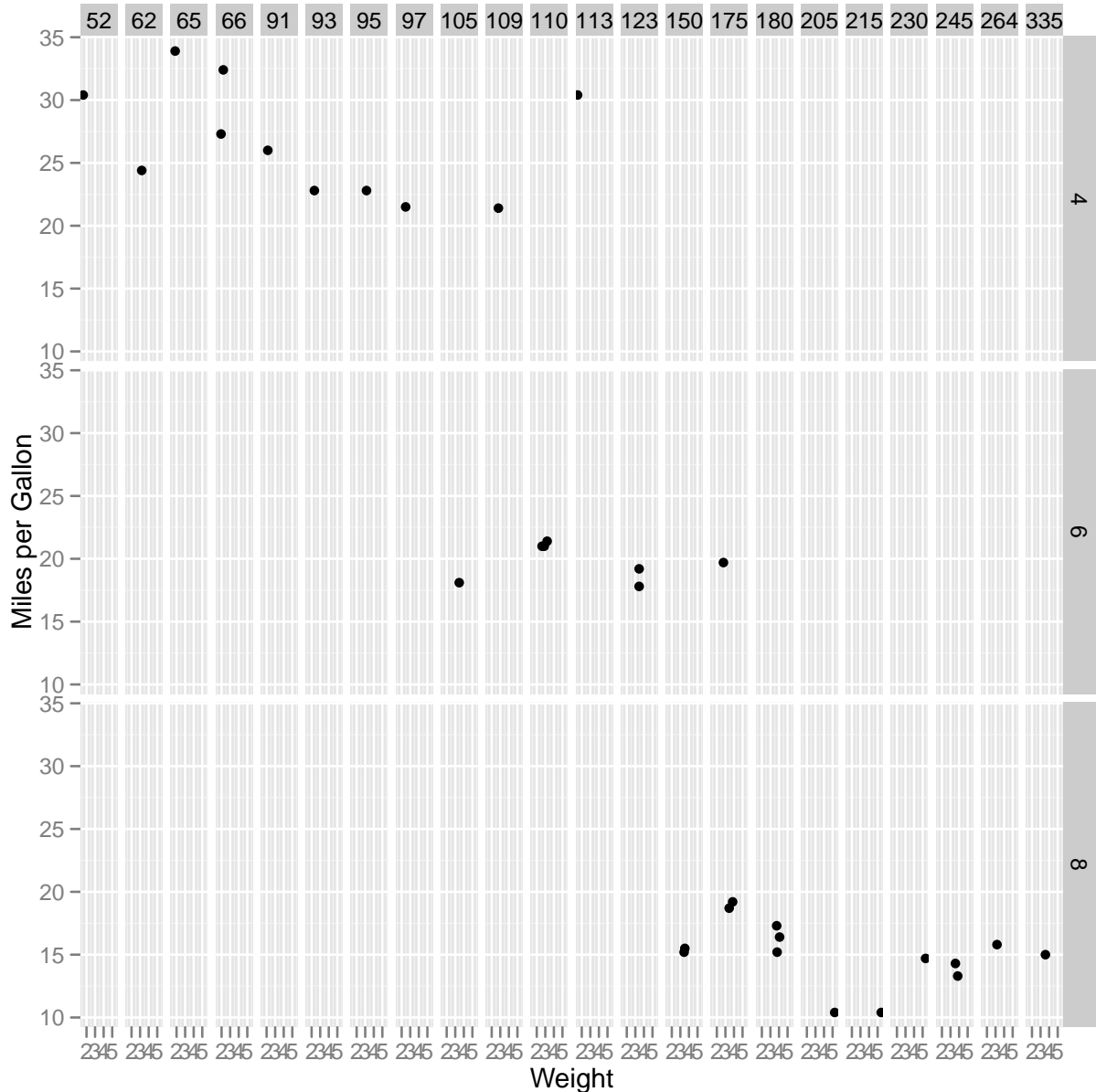
```
table(mtcars$cyl, mtcars$hp)
```

```
##
##      52 62 65 66 91 93 95 97 105 109 110 113 123 150 175 180 205 215
##  4    1  1  1  2  1  1  1  1  0  1  0  1  0  0  0  0  0  0
##  6    0  0  0  0  0  0  0  0  1  0  3  0  2  0  1  0  0  0
##  8    0  0  0  0  0  0  0  0  0  0  0  0  0  2  2  3  1  1
```



```
##
##      230 245 264 335
##    4    0    0    0    0
##    6    0    0    0    0
##    8    1    2    1    1

qplot(wt, mpg, data = mtcars, facets = cyl ~ hp, xlab = "Weight",
      ylab = "Miles per Gallon")
```



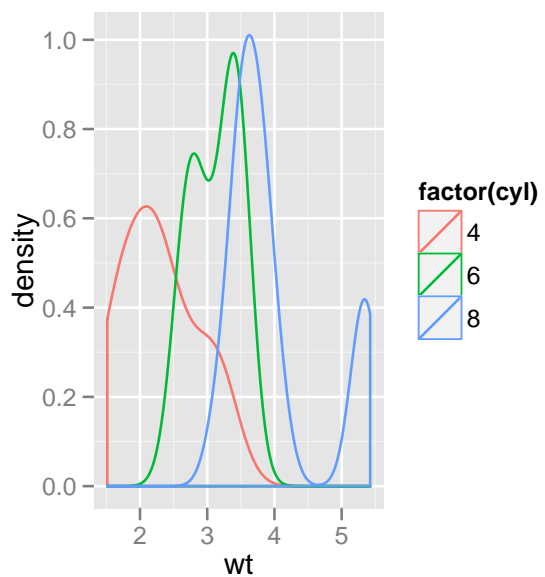
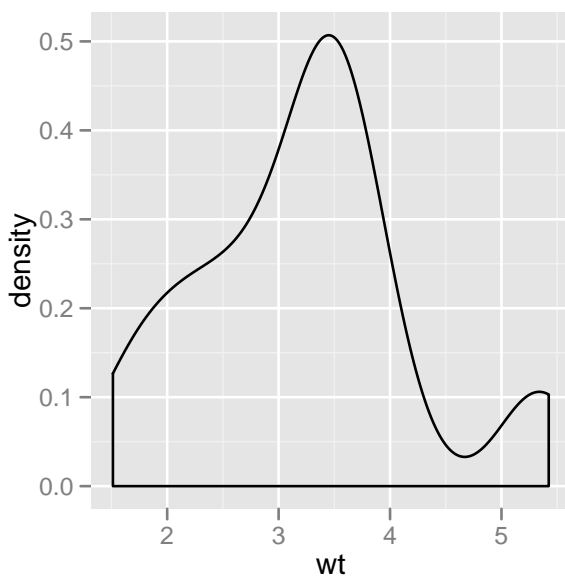
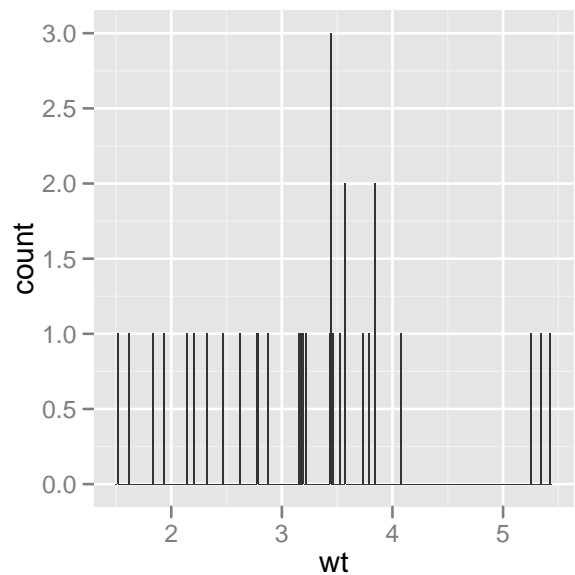
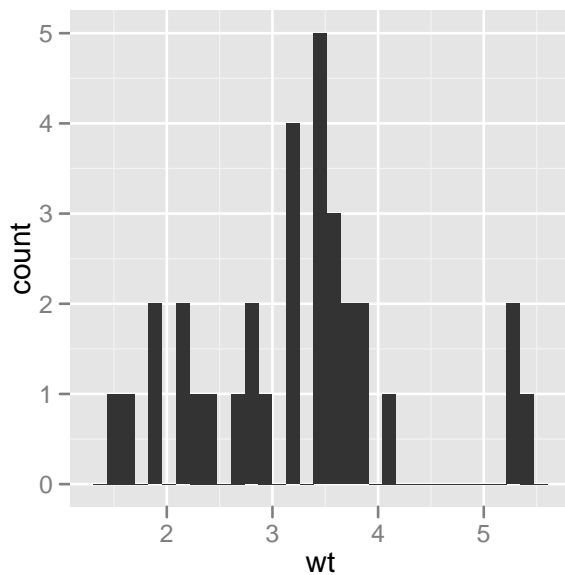
In the scatterplot examples above, we implicitly used a *point geom*, the default when you supply two arguments to `qplot()`. `qplots` can produce several other plots, if a different geom is defined (modified from *ggplots2: Elegant Graphics for Data Analysis*, Chapter 3)

	Plot	Geom	Other features
scatterplot	point		
bubblechart	point	size defined by variable	
barchart	bar		
box-whisper	boxplot		
line	line		

When given a single vector, the default geom is Histogram. Defining geom as density will instead draw a density (smoothed histogram)

```
p1 <- qplot(wt, data = mtcars)
p2 <- qplot(wt, data = mtcars, binwidth = 0.01)
p3 <- qplot(wt, data = mtcars, geom = "density")
p4 <- qplot(wt, data = mtcars, geom = "density", colour = factor(cyl))
# Arrange and display the plots into a 2x1 grid
arrange(p1, p2, p3, p4, nrow = 2, ncol = 2)

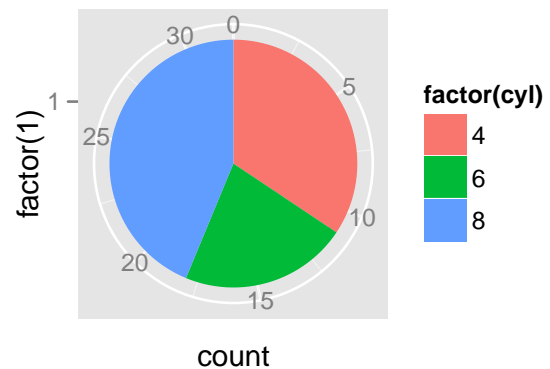
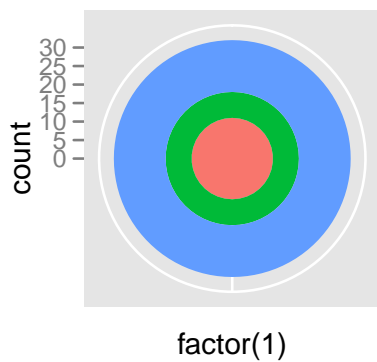
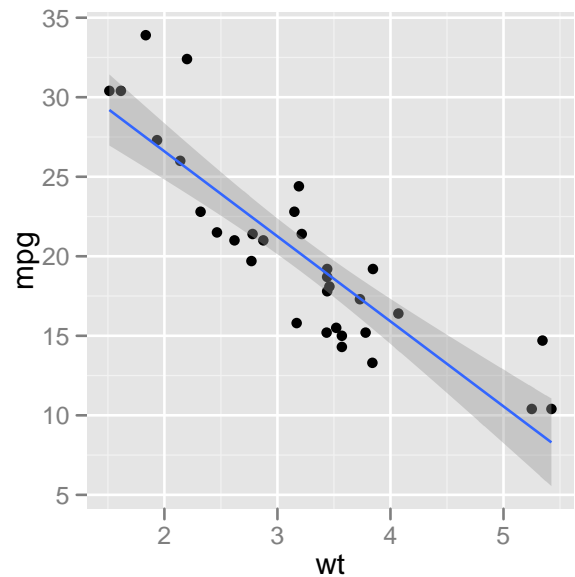
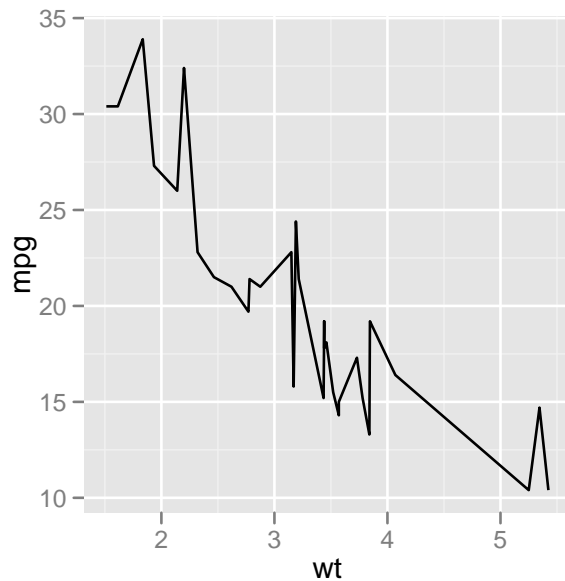
## stat_bin: binwidth defaulted to range/30. Use binwidth = x to
## adjust this.
```



Geoms - point and smooth and coord_polar (Pie Charts)

```
p1 <- ggplot(wt, mpg, data = mtcars, geom = "line")
p2 <- ggplot(wt, mpg, data = mtcars, geom = c("point", "smooth"),
  method = "lm")

px <- ggplot(mtcars, aes(x = factor(1), fill = factor(cyl))) +
  geom_bar(width = 1)
# map a barchart to a polar coordinate system
p3 <- px + coord_polar()
p4 <- px + coord_polar(theta = "y")
arrange(p1, p2, p3, p4, nrow = 2, ncol = 2)
```



6.1.2 lattice

Lattice plots allow the use of the layout on the page to reflect meaningful aspects of data structure. They offer abilities similar to those in the S-PLUS trellis library.

The lattice package sits on top of the grid package. To use lattice graphics, both these packages must be installed. Providing it is installed, the grid package will be loaded automatically when lattice is loaded.

Resources for lattice

- Book on Lattice <http://lmdvr.r-forge.r-project.org/figures/figures.html>
- See examples at <http://cm.bell-labs.com/cm/ms/departments/sia/project/trellis/display.examples.html>

- To get on help on **lattice** functions, use `help` just like you would do for any package `help(package = lattice)`

6.1.3 Examples that Present Panels of Scatterplots using `xyplot()`

The basic function for drawing panels of scatterplots is `xyplot()`. We will use the data frame 'ChickWeight' to demonstrate the use of `xyplot()`. The 'ChickWeight' data frame has 578 rows and 4 columns from an experiment on the effect of diet on early growth of chicks. This data frame contains the following columns:

weight a numeric vector giving the body weight of the chick (gm).

Time a numeric vector giving the number of days since birth when the measurement was made.

Chick an ordered factor with levels '18' ; ... ; '48' giving a unique identifier for the chick. The ordering of the levels groups chicks on the same diet together and orders them according to their final weight (lightest to heaviest) within diet.

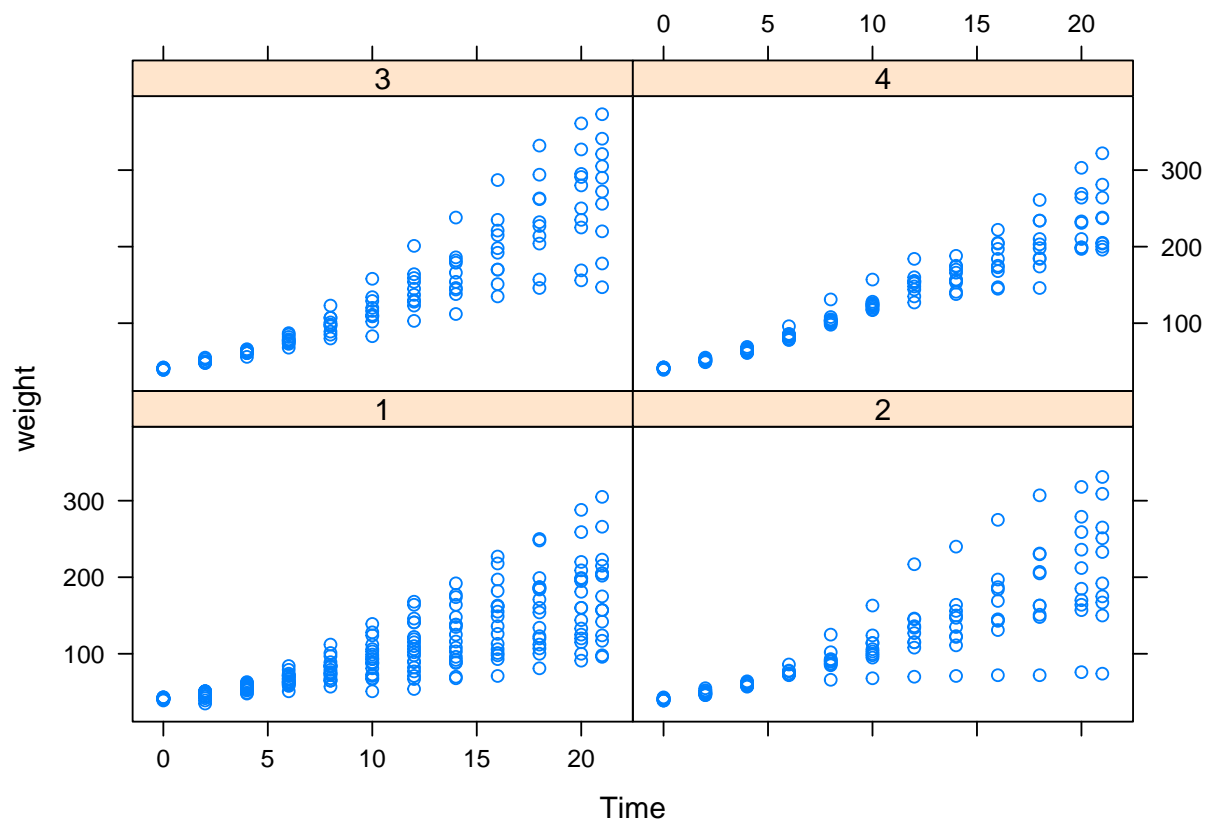
Diet a factor with levels 1,...,4 indicating which experimental diet the chick received.

Figure below shows the style of graph that one can get from `xyplot()`.

6.1.4 Simple use of `xyplot`

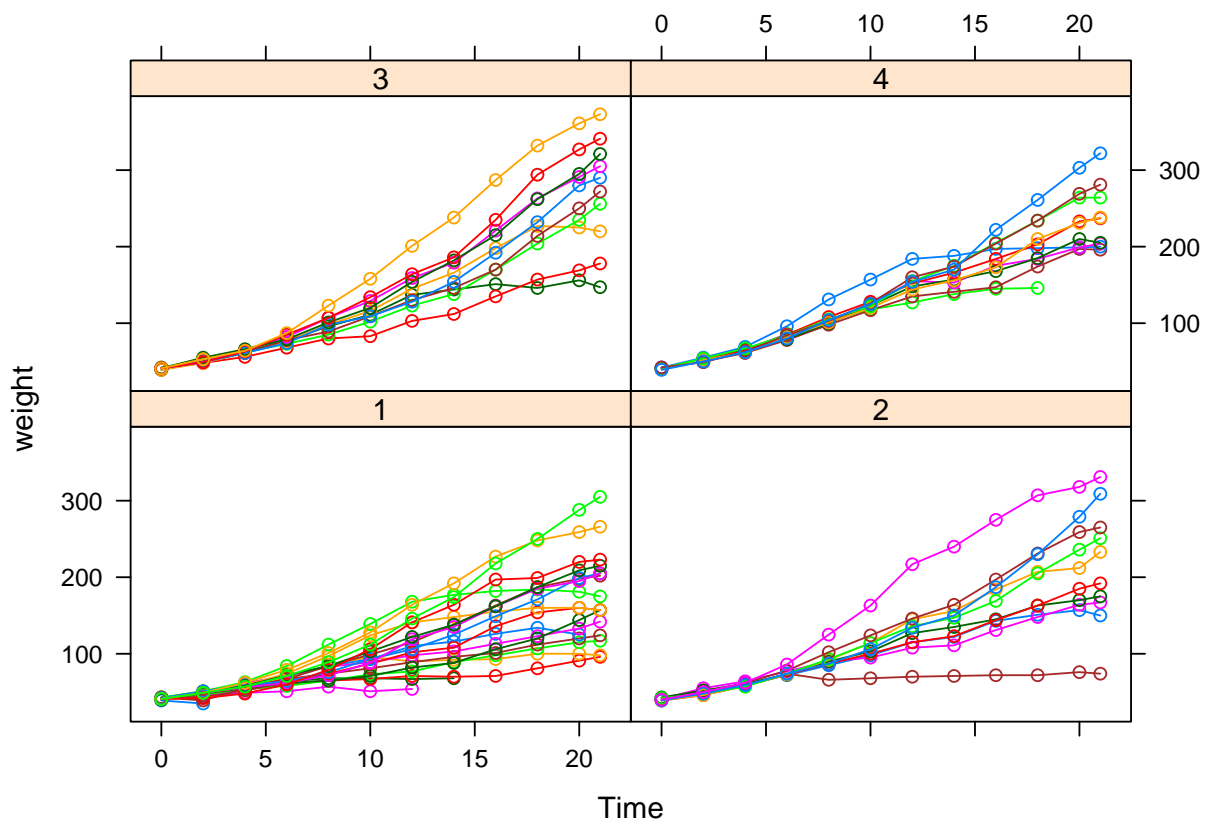
The lattice function `xyplot()` is the most commonly used **lattice** function, and plots pairs of variables. Whilst designed mainly for two continuous variates, factors can be supplied as well, in which case they will simply be coerced to numeric.

```
library(lattice)
xyplot(weight ~ Time | Diet, data = ChickWeight) # Simple use of
xyplot
```



Here is the statement used to get a figure with the observations for the same Chick connected via lines.

```
xyplot(weight ~ Time | Diet, data = ChickWeight, panel =
panel.superpose,
groups = Chick, type = "b")
```



This function shows the defaults for the graphical display of Trellis displays

```
show.settings()
```

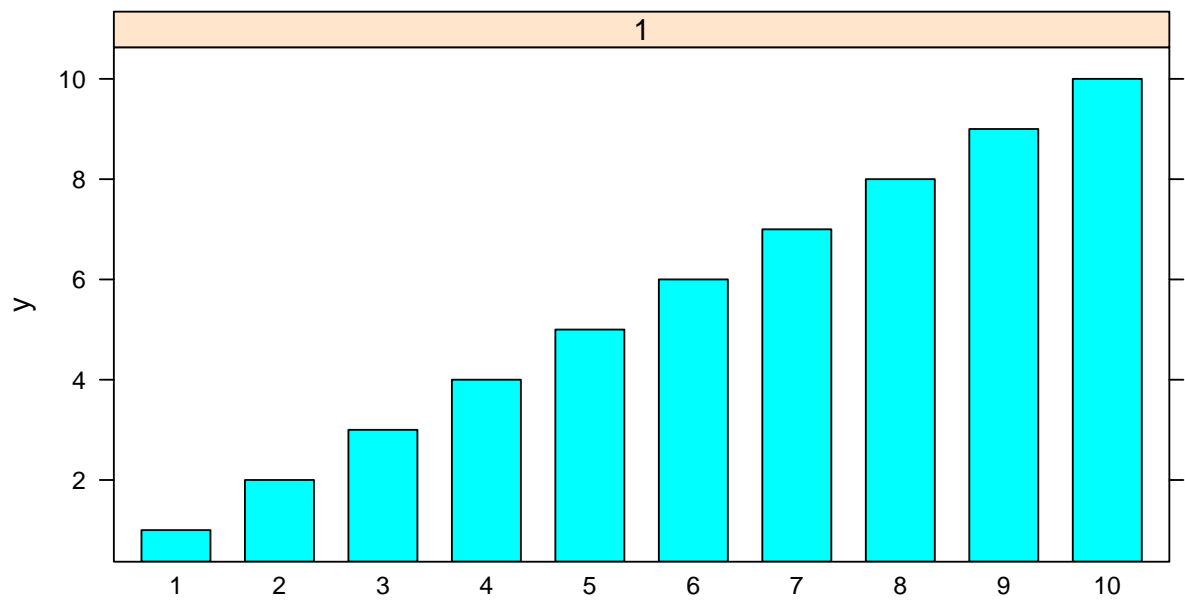
An incomplete list of lattice Functions

```
splom( ~ data.frame)          # Scatterplot matrix
bwplot(factor ~ numeric , . .) # Box and whisker plot
dotplot(factor ~ numeric , . .) # 1-dim. Display
stripplot(factor ~ numeric , . .) # 1-dim. Display
barchart(character ~ numeric,...)
histogram( ~ numeric, ...)    # Histogram
densityplot( ~ numeric, ...)  # Smoothed version of histogram
qqmath(numeric ~ numeric, ...) # QQ plot
splom( ~ dataframe, ...)      # Scatterplot matrix
parallelplot( ~ dataframe, ...) # Parallel coordinate plots
```

In each instance, conditioning variables can be added.

Examples:

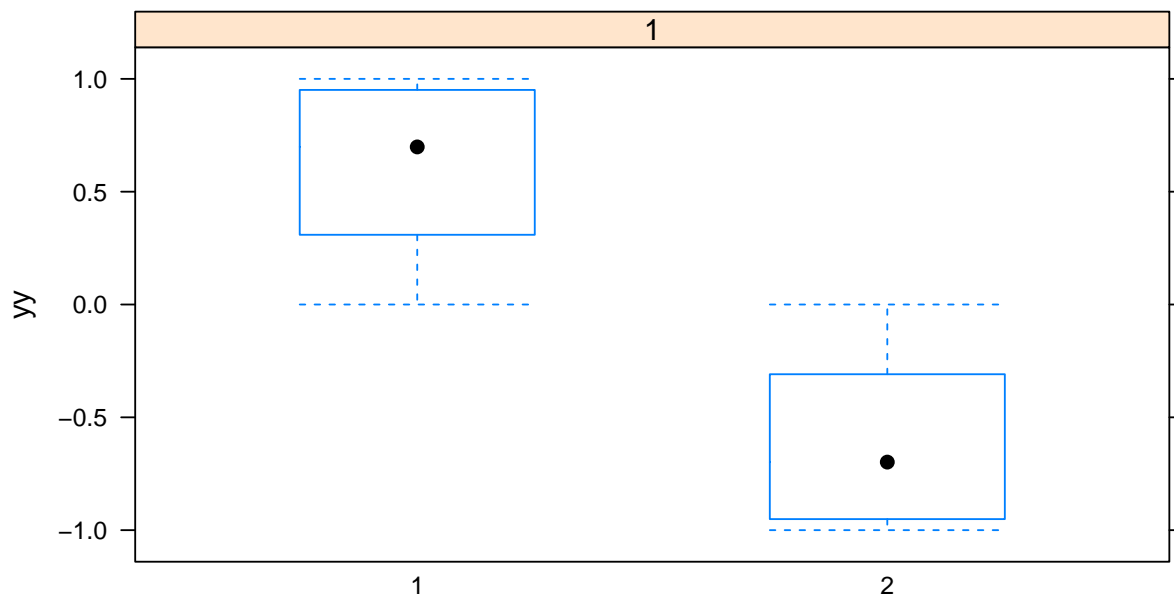
```
x <- 1:10
y <- 1:10
g <- factor(1:10)
barchart(y ~ g | 1)
```



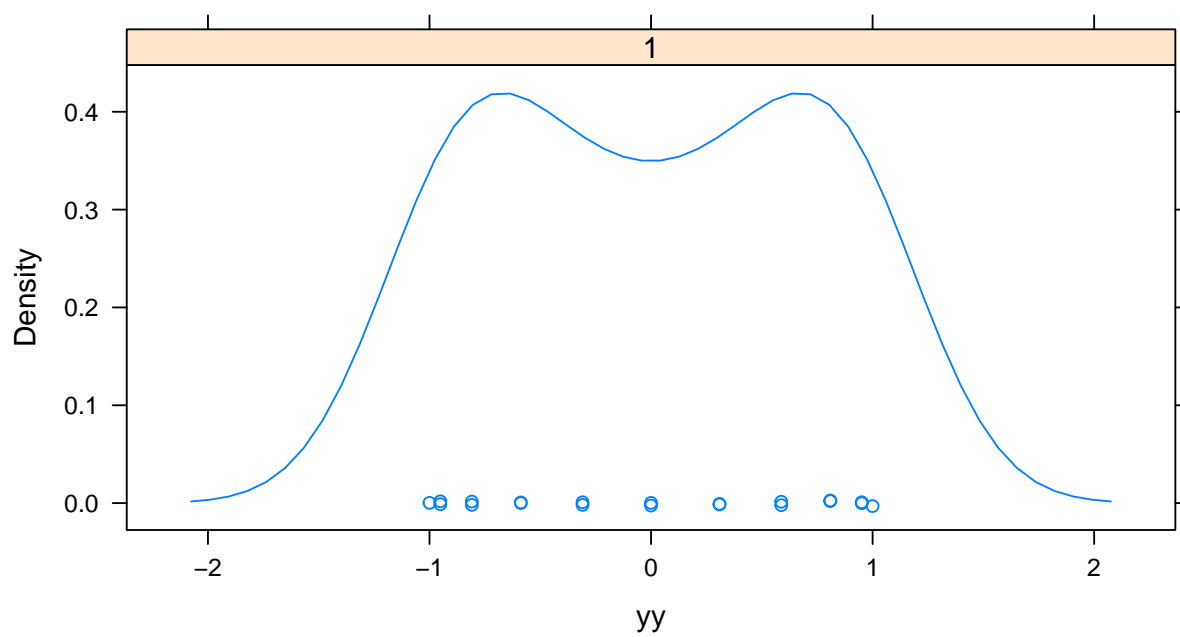
More examples:Lattice density histogram splom and more

```
angle <- seq(0, 2 * pi, length = 21)[-21]
xx <- cos(angle)
yy <- sin(angle)
gg <- factor(rep(1:2, each = 10))

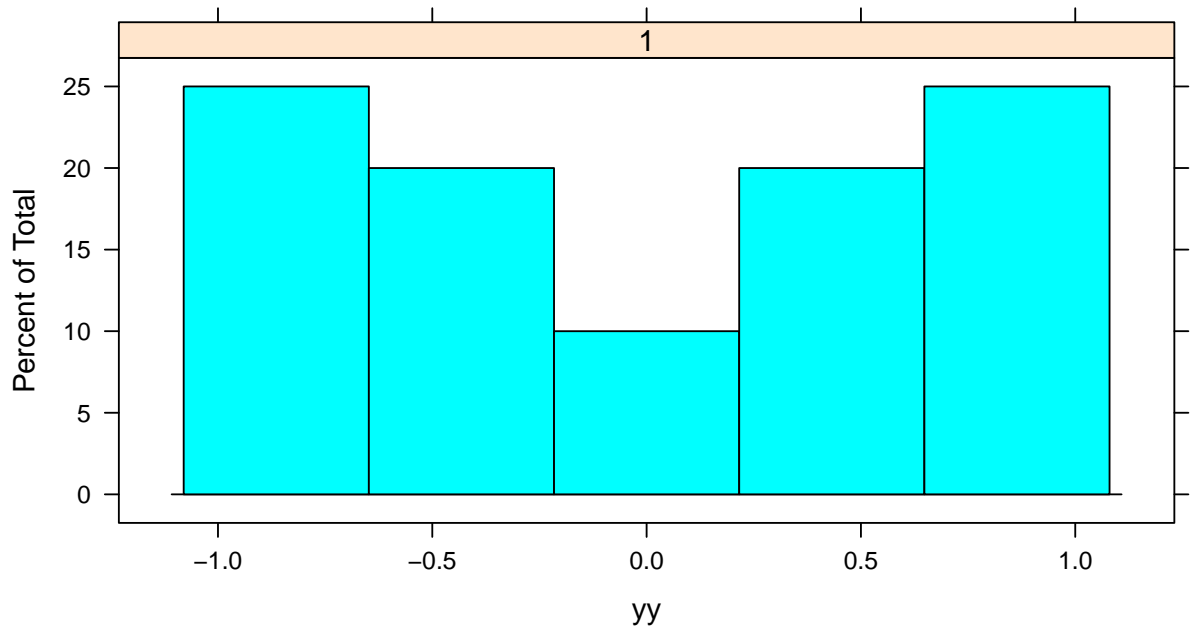
bwplot(yy ~ gg | 1)
```

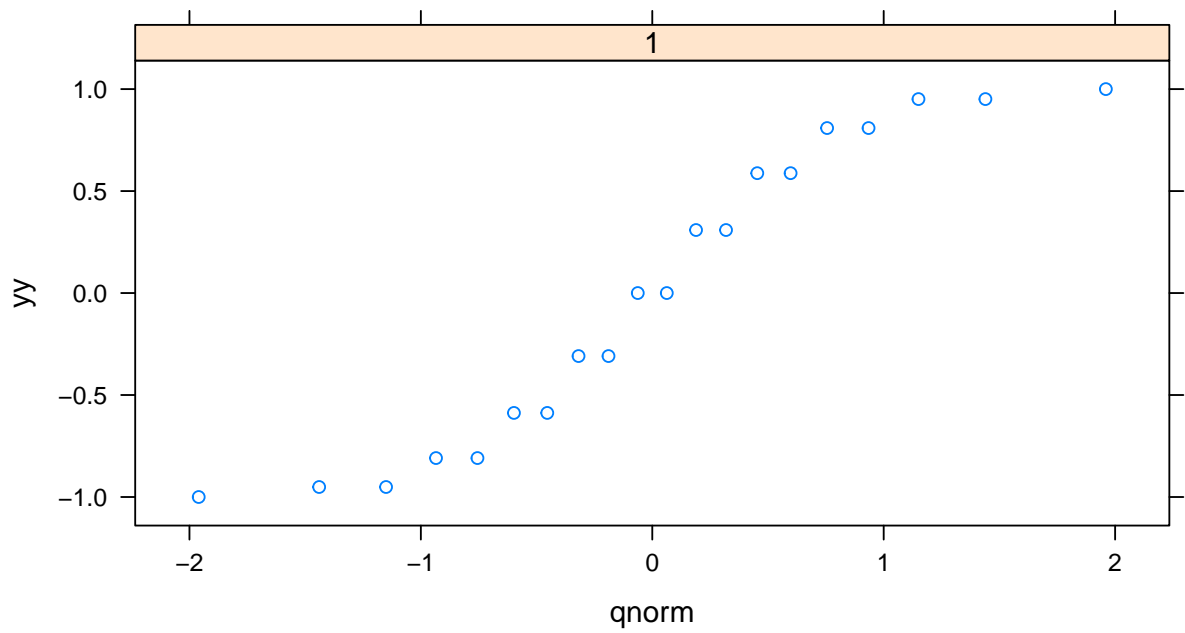
```
densityplot (~yy | 1)
```



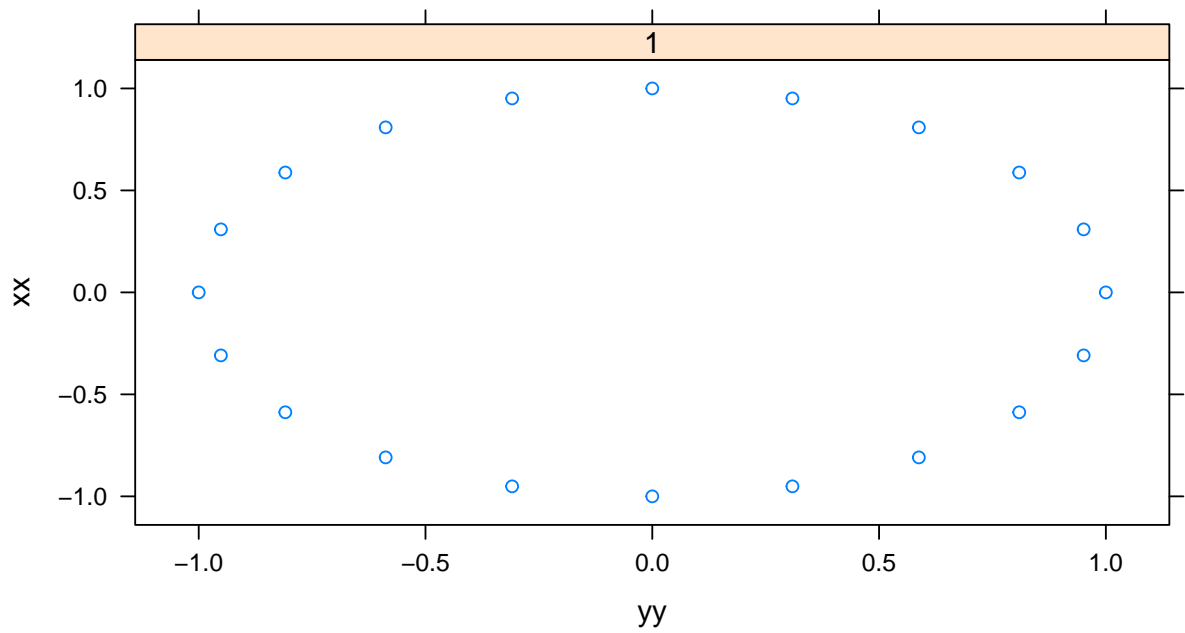
```
histogram(~yy | 1)
```



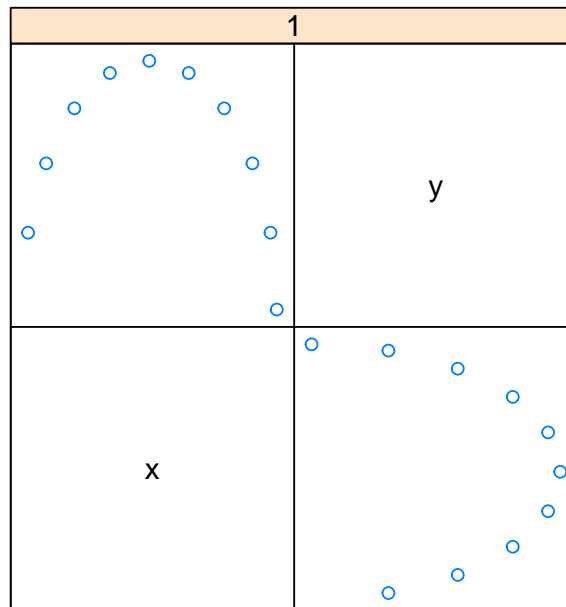
```
qqmath(~yy | 1)
```



```
xyplot(xx ~ yy | 1)
```

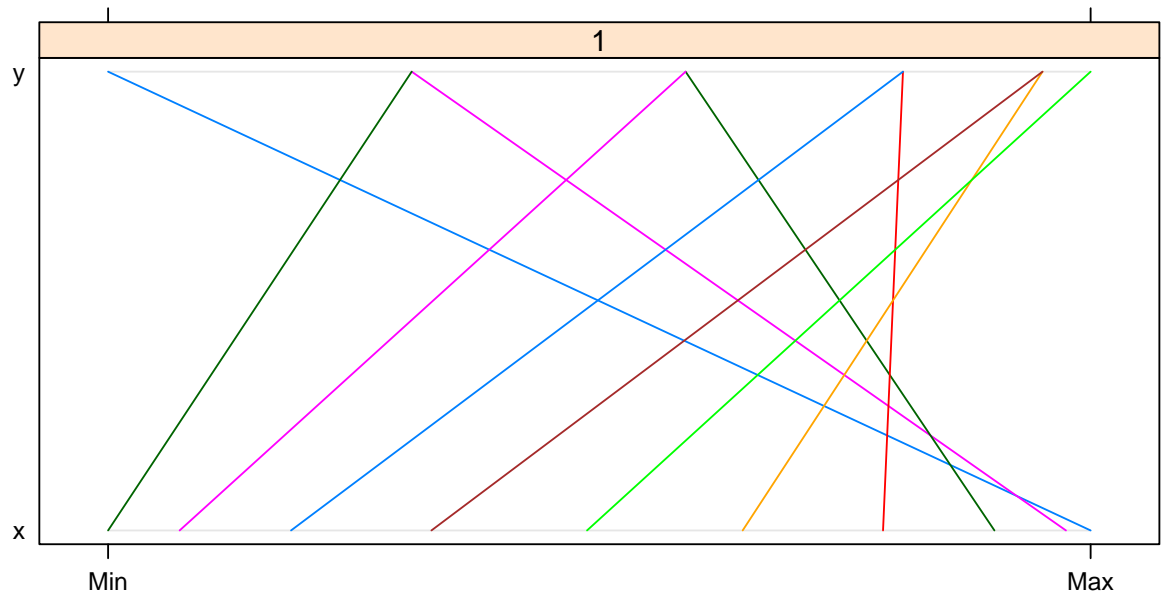


```
splom(~data.frame(x = xx[1:10], y = yy[1:10]) | 1, pscales = 0)
```



Scatter Plot Matrix

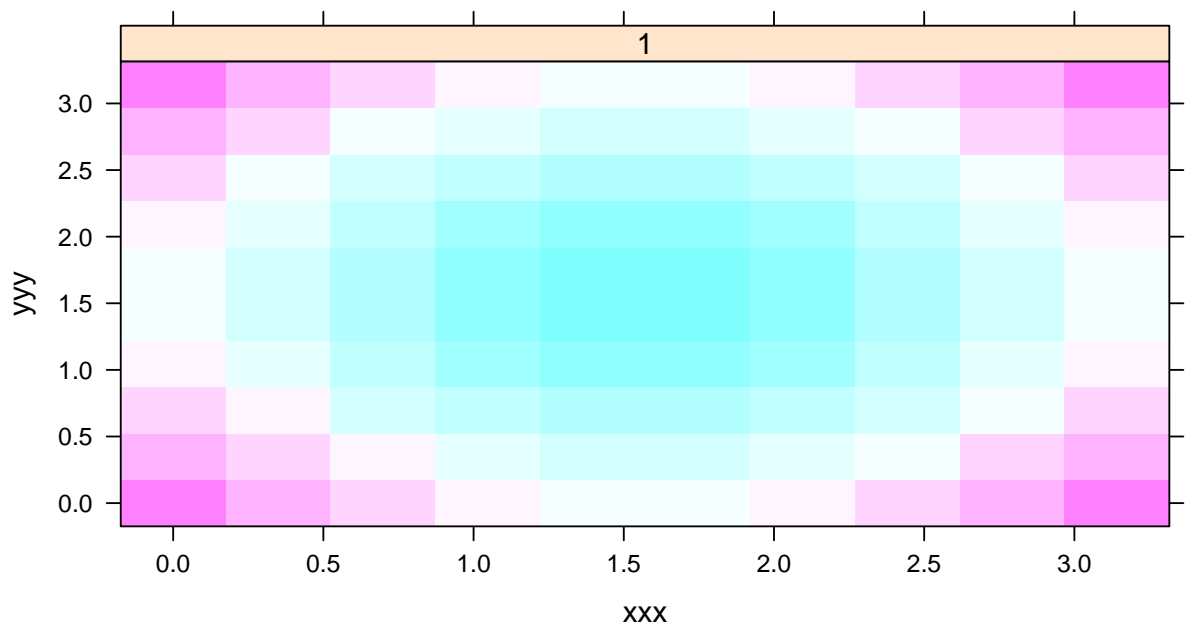
```
parallelplot(~data.frame(x = xx[1:10], y = yy[1:10]) | 1)
```



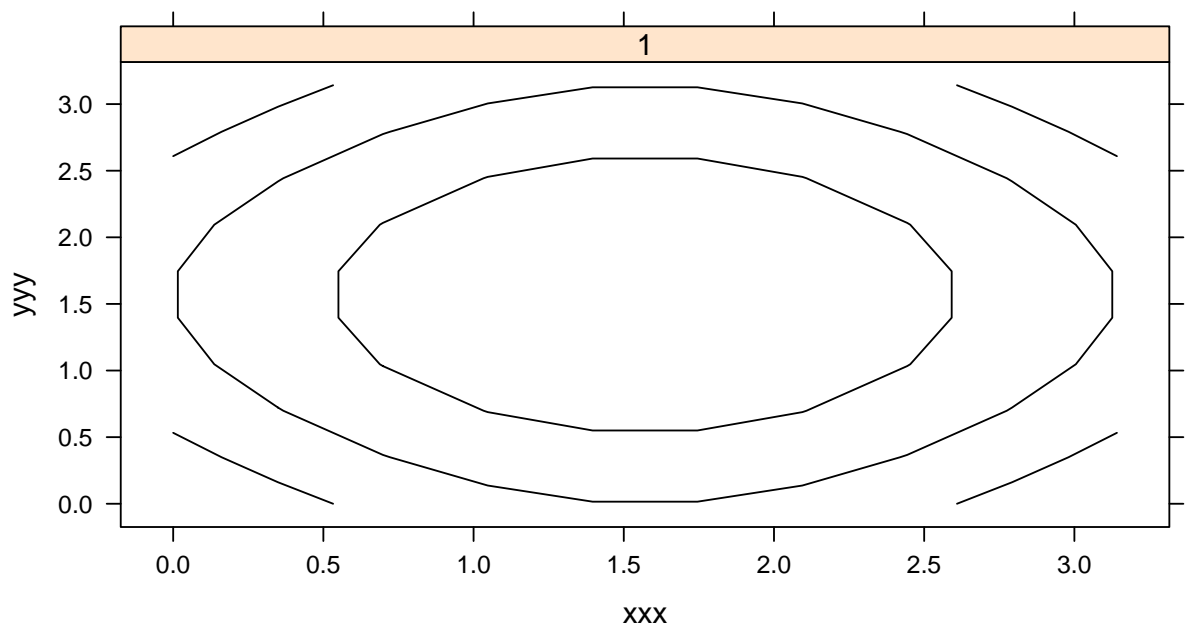
More than two variables

```
aaa <- seq(0, pi, length = 10)
xxx <- rep(aaa, 10)
yyy <- rep(aaa, each = 10)
zzz <- sin(xxx) + sin(yyy)

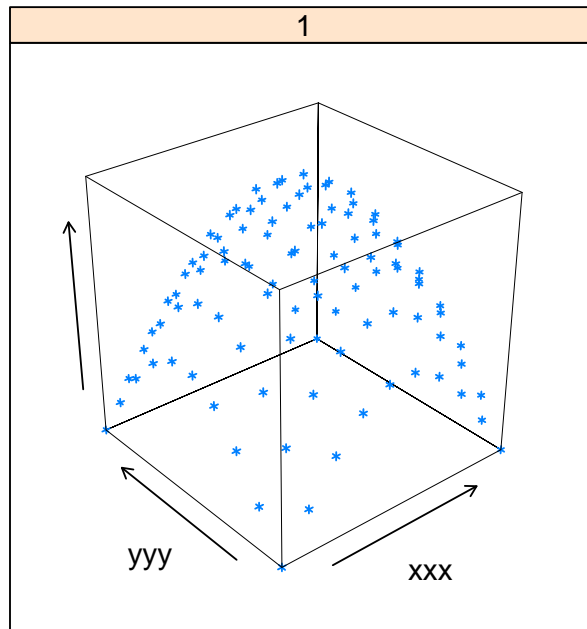
levelplot(zzz ~ xxx + yyy | 1, colorkey = FALSE)
```



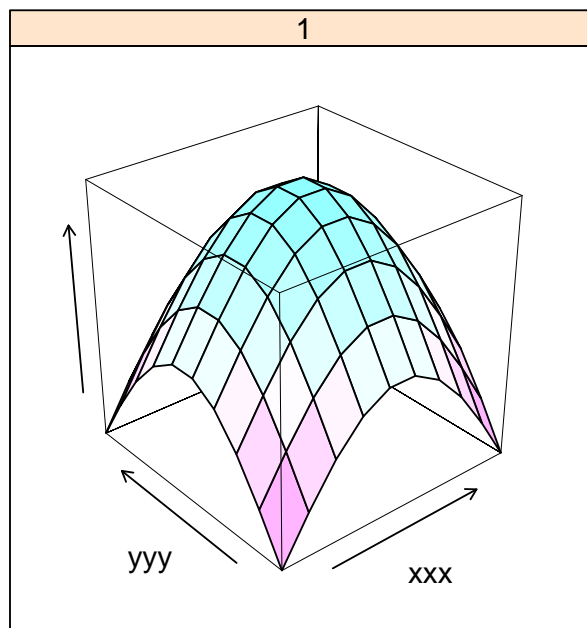
```
contourplot (zzz ~ xxx + yyy | 1, labels = FALSE, cuts = 8)
```



```
cloud(zzz ~ xxx + yyy | 1, zlab = NULL, zoom = 0.9, par.settings =
list(box.3d = list(lwd = 0.01)))
```



```
wireframe(zzz ~ xxx + yyy | 1, zlab = NULL, zoom = 0.9, drape = TRUE,
par.settings = list(box.3d = list(lwd = 0.01)), colorkey = FALSE)
```



Note whilst, lattice plots are highly customizable. **Note:** the base graphics settings; in particular, *par()* settings usually have no effect on lattice plots. Use *trellis.par.get()* and *trellis.par.set()* to change default plot parameters.

6.2 GoogleVis and GoogleMaps visualization

There are multiple visualization tools available within the googleVis library. These include the Hans Rosling type bubble plots. See <http://code.google.com/apis/visualization/documentation/gallery/motionchart.html>

See <http://blog.revolutionanalytics.com/graphics/> for some examples of R code

```
# install.packages('googleVis')
library(googleVis)

## Loading required package: RJSONIO
## Welcome to googleVis version 0.2.16
##
## Please read the Google API Terms of Use before you use the package:
## http://code.google.com/apis/terms/index.html
##
## Type ?googleVis to access the overall documentation and
## vignette(googleVis) for the package vignette. You can execute a
## demo of the package via: demo(googleVis)
##
## More information is available on the googleVis project web-site:
## http://code.google.com/p/google-motion-charts-with-r/
##
## Contact: <rvisualisation@gmail.com>
##
## To suppress the this message use:
## suppressPackageStartupMessages(library(googleVis))

M <- gvisMotionChart(Fruits, "Fruit", "Year")
plot(M)
cat(M$html$chart, file = "tmp.html")
```

GoogleVis also provide nice support for Maps and spatial visualization of trends.

```
# Pretty plots competition
library(googleVis)
help(package = "googleVis")

# looking at all loaded datasets
data(state)
states <- as.data.frame(state.x77)
states$location <- rownames(states)
attach(states)

## The following object(s) are masked from package:googleVis:
##
##      Population

states.Inc <- gvisGeoMap(states, locationvar = "location",
```

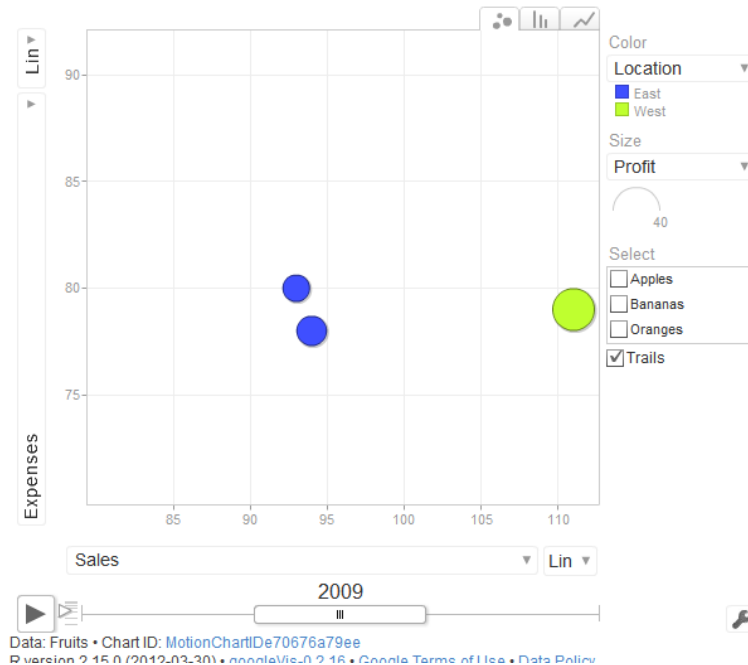



Figure 6.1: This is actually an interactive animated html file

```
numvar = "Income", options = list(region = "US", dataMode =
"regions",
  colors = "[0xBEBEBE, 0x00008B]"))
# note this is per capita income by state from 1974

plot(states.Inc)

states.Illit <- gvisGeoMap(states, locationvar = "location",
  numvar = "Illiteracy", options = list(region = "US", dataMode =
"regions",
  colors = "[0xBEBEBE, 0x8B0000]"))
plot(states.Illit)

merge <- gvisMerge(states.Inc, states.Illit, horizontal = FALSE)
merge.plot <- plot(merge)
detach(states)
```

For alternative approaches to spatial or info Maps see <http://ryouready.wordpress.com/2009/11/16/infomaps-using-r-visualizing-german-unemployment-rates-by-color-on-a-map/>

Also see the packages RgoogleMaps which provide a comfortable R interface to query the Google server for static maps, and to also use the map as a background image to overlay plots within R.

Another packages is osmar (OpenStreetMap and R) which package provides infrastructure to access OpenStreetMap data.

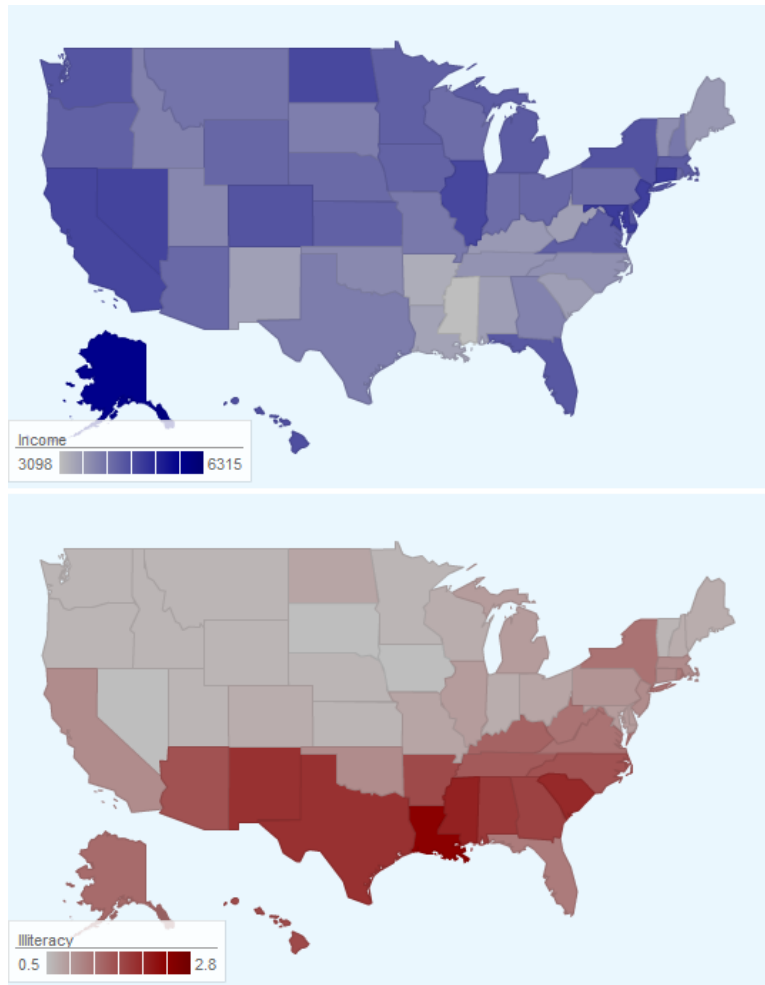


Figure 6.2: This is actually an interactive html file, if you hover over a state you see its information

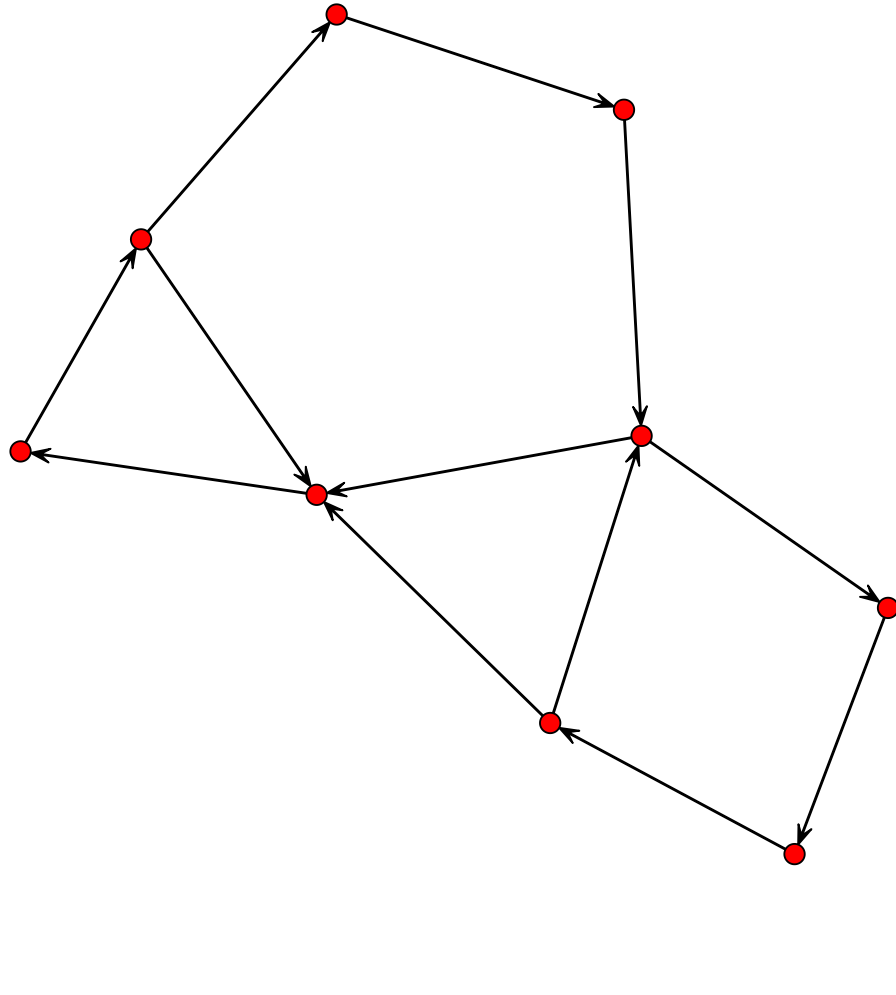
6.3 Graph theory and Network visualization using R packages network and igraph

```
# install.packages(network)
library(network)

## network: Classes for Relational Data Version 1.7-1 created on March
## 1, 2012. copyright (c) 2005, Carter T. Butts, University of
## California-Irvine Mark S. Handcock, University of Washington David
## R. Hunter, Penn State University Martina Morris, University of
## Washington For citation information, type citation("network").
## Type help("network-package") to get started.

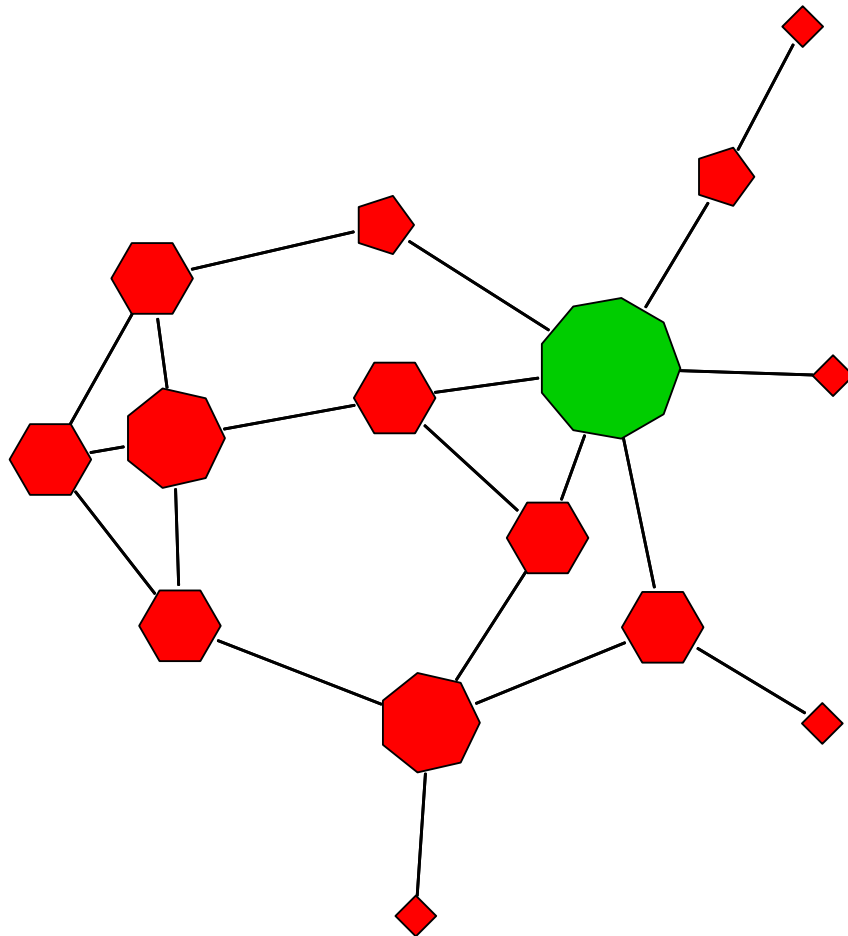
m <- matrix(rbinom(100, 1, 1.5/9), 10)
diag(m) <- 0
```

```
g <- network(m)
# Plot the graph
plot(g)
```



```
# Load Padgett's marriage data
data(flo)
nflo <- network(flo)
# Display the network, indicating degree and flagging the Medicis
plot(nflo, vertex.cex = apply(flo, 2, sum) + 1, usearrows = FALSE,
      vertex.sizes = 3 + apply(flo, 2, sum), vertex.col = 2 +
(network.vertex.names(nflo) ==
```

```
"Medici"))
```

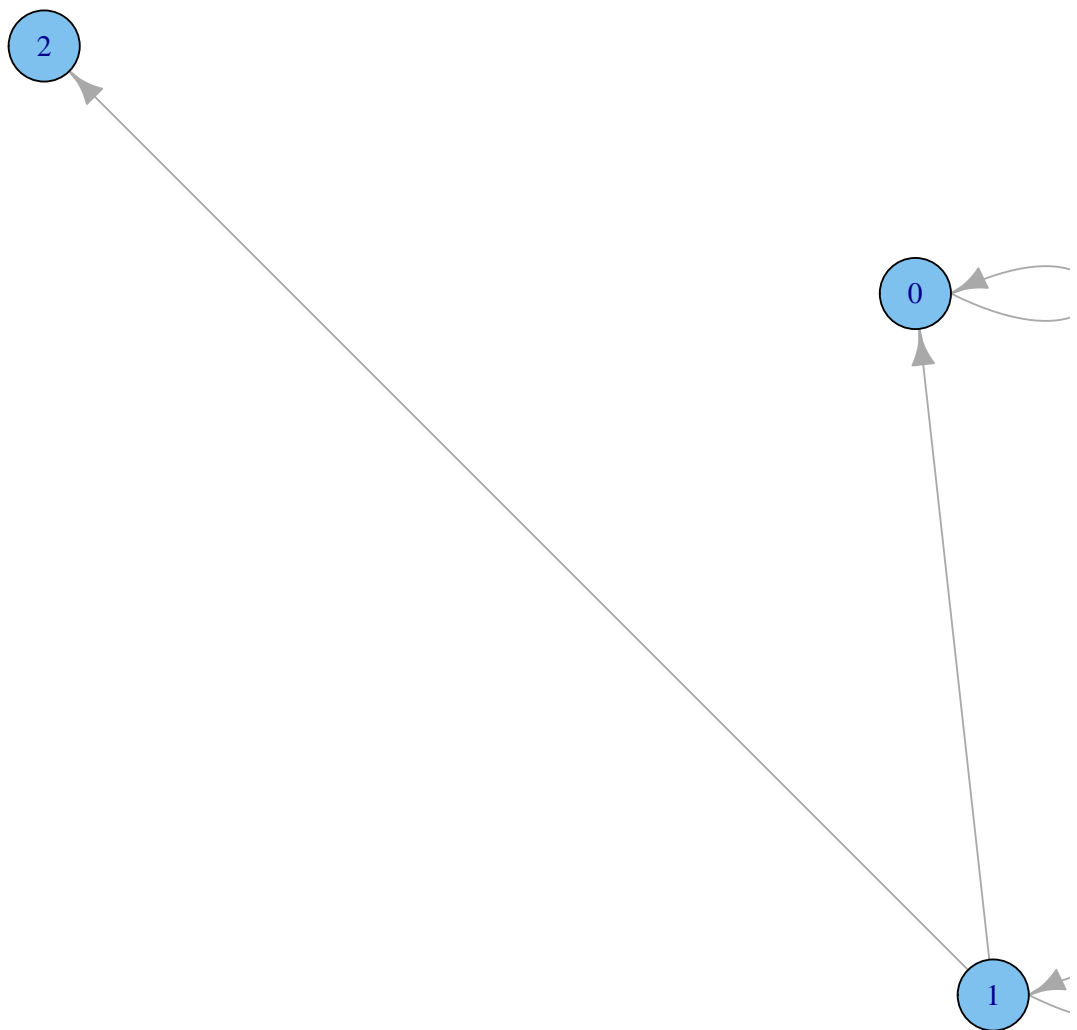


using the package igraph

```
# install.packages('igraph')
library(igraph)

## Attaching package: igraph
## The following object(s) are masked from package:network:
##
## %c%, %s%, add.edges, add.vertices, delete.edges, delete.vertices,
## get.edge.attribute, get.edges, get.vertex.attribute, is.bipartite,
```

```
## is.directed, list.edge.attributes, list.vertex.attributes,  
## set.edge.attribute, set.vertex.attribute  
  
adj.mat <- matrix(sample(c(0, 1), 9, replace = TRUE), nr = 3)  
g <- graph.adjacency(adj.mat)  
plot(g)
```



6.4 Tag Clouds, Literature Mining

The following script will create tag cloud given a list of PubMed abstract identifiers.

```
## ----- Given a
## list of PMIDs get their annotation Aedin, Dec 2011 To Run given
## pmids2tagcloud a list of pmids eg pmids=c(10521349, 10582678,
## 11004666, 11108479, 11108479, 11114790, 11156382, 11156382,
## 11156382, 11165872) pmids2tagcloud(pmids)
## -----
getPMIDAnnot <- function(pmidlist) {
  require(annotate)
  require(XML)
  print("Using annotate and XML to get info on each PMID")
  pubmedRes <- xmlRoot(pubmed(pmidlist))
  numAbst <- length(xmlChildren(pubmedRes))
  absts <- list()
  for (i in 1:numAbst) {
    absts[[i]] <- buildPubMedAbst(pubmedRes[[i]])
  }

  # unlist(lapply(absts, function(x) authors(x)[1]))

  ## Write Output to PMIDInfo
  PMIDInfo <- data.frame(matrix(NA, nrow = length(pmidlist)))
  PMIDInfo$FirstAuthor <- unlist(lapply(absts, function(x)
authors(x)[1]))
  PMIDInfo$Journal <- unlist(lapply(absts, function(x)
journal(x)[1]))
  PMIDInfo$pubDate <- unlist(lapply(absts, function(x)
pubDate(x)[1]))
  PMIDInfo$articleTitle <- unlist(lapply(absts, function(x)
articleTitle(x)[1]))
  PMIDInfo$abstText <- unlist(lapply(absts, function(x)
abstText(x)[1]))
  PMIDInfo$PubMedID <- unlist(lapply(absts, function(x) pmid(x)[1]))
  rownames(PMIDInfo) <- PMIDInfo$PubMedID
  PMIDInfo <- PMIDInfo[, -1]

  # Res<-cbind(outMat, Total= apply(outMat, 1, sum), PMIDInfo[,c(5,
# 1,3,4,2)]) Res$pubDate<-unlist(strsplit(Res$pubDate, ' '))[seq(2,
# length(Res$pubDate)*2, 2)] names(Res)[10] ='Year' print(Res)

  # print(PMIDInfo[1:2,])
  return(PMIDInfo)
}
```

```

pmids2tagcloud <- function(pmids, addTitle = TRUE, colorPalette =
c("orange",
  "cyan", "green4", "maroon", "slateblue")) {
  require(tm)
  require(wordcloud)
  require(RColorBrewer)
  print(paste("Using tm and wordcloud to create tag cloud from",
length(pmids),
  "abstracts"))
  pubmedAbsts <- getPMIDAnnot(as.character(unique(pmids)))
  words <-
tolower(unlist(strsplit(as.character(pubmedAbsts$abstText),
  " ")))
  # remove parentheses, comma, [semi-]colon, period, quotation marks
  words <- words[-grep("[\\]\\(,;:\\.\\'\\\\\"", words)]
  words <- words[-grep("^\\d+$", words)]
  words <- words[!words %in% stopwords()]
  wt <- table(words)

  ## Use R Color Brewer Colors
  pal <- brewer.pal(9, "BuGn")
  pal <- pal[-(1:4)]

  wordcloud(names(wt), as.vector(wt), colors = colorPalette)
  if (addTitle)
    title(main = paste("TagCloud generated from", length(pmids),
      "PubMed Abstracts"), sub = paste("PMIDS:", paste(pmids,
collapse = " "),
      sep = ""), cex.sub = 0.5, col.main = "green4", col.sub =
"gray")
}

pmids <- c(10521349, 10582678, 11004666, 11108479, 11108479,
  11114790, 11156382, 11156382, 11156382, 11165872)
pmids2tagcloud(pmids)

## [1] "Using tm and wordcloud to create tag cloud from 10 abstracts"
## [1] "Using annotate and XML to get info on each PMID"

```

Other visualization resources that you may like to explore

using Rgraphviz http://www2.warwick.ac.uk/fac/sci/moac/students/peter_cock/r/rgraphviz/ or see the many examples on the bioconductor website

a recent discussion online about the topic: <http://stats.stackexchange.com/questions/6155/graph-theory-analysis-and-visualization>

R cytoscape <http://db.systemsbiology.net:8080/cytoscape/RCytoscape/vignette/RCytoscape.html>

3. Additional demos available in the graphics package: *demo(image)*, *demo(persp)* and *example(symbol)*.

The following web pages have many examples (and code) to produce different R plots. Browse through the plots, see what you like and try some.

- Basic plotting examples from Paul Murrell book, R Graphics <http://www.stat.auckland.ac.nz/~paul/RGraphics/rgraphics.html>
- The homepage of the R package ggplot2 <http://had.co.nz/ggplot2/>. This package produces nice plots and can easily add color scale legend bars to plots
- rggobi and ggplot2 run workshops in R graphics. Their course website provides examples of basic and advanced plots, animated movies, lectures and R code to reproduce the plots at <http://lookingatdata.com/>
- The R Gallery wiki provides examples of R plots and code to reproduce these <http://addictedtor.free.fr/graphiques/>. Here is one random sample from this website:

6.6 Summary on plotting

- Basic plotting:
 - *plot()*, *pairs()*, *histogram*, *pie* etc
 - Low-level plotting: *points()*, *lines()*, *abline()*
low-level other: *text()*, *legend()*, *title()*
- Manipulating the plotting window
 - Temporary changes to just one command: “...” argument to *plot()* function
 - To view or change default plot settings: *par()*. This will change the settings for all subsequent plot commands.
- Advanced plotting using ggplots2 library.

Chapter 7

Statistical Analysis, linear models and survival analysis in R

7.1 This section

1. Basic statistics such as t-test, χ^2 ,...
2. Intro to linear models in R
3. Model formulae and model options
4. Output and extraction from fitted models
5. `model.matrix`, contrasts
6. Models considered:
 - Linear regression: `lm()`
 - Logistic regression: `glm()`, Poisson regression: `glm()`
 - Survival analysis: `Surv()`, `coxph()` in the `survival` and functions in the package `survcomp`
7. Advanced model options are covered in detail in the recommended text of Venables and Ripley.
 - Generalized Linear Mixed-Effects Models `lmer`, `lme`
 - Generalized additive models `gam()`
 - Non-Linear models `nls`, `nlme`
 - Other useful packages `lme4`, `gmodels`
 - The `arm` package contains R functions for Bayesian inference using `lm`, `glm`, `mer` and `polr` objects. The `bayesm` is aims at marketing and micro economics fields but includes functions for Bayes Regression and Hierarchical Linear Models.
 - The R package `doBy` is useful for groupwise computations of summary statistics. Facilities for groupwise computations of summary statistics and other facilities for working with grouped data (similar to what can be achieved by `proc means` or `proc summary` of the sas system).

- See <http://cran.r-project.org/src/contrib/Views/> for lists of more R packages.

7.2 Basic Statistics

7.2.1 Continuous Data: t test

The *t.test* performs a one or two sample t test. To see the arguments of *t.test*, look at the help documentation *?t.test*

Arguments include *alternative* which is one of "two.sided", "less" or "greater", and *var.equal* which is a logical (FALSE or TRUE) to indicate unequal or equal variance (default is unequal). The input to *t.test* is one vector (one sample t test), two vectors or a formula (two sample t test). A formula is given by $y \sim x$, where the tilde '~' operator specifies "described by"

One sample t test:

```
data(ChickWeight)
ChickWeight[1:2, ]

##      weight Time Chick Diet
## 1       42    0     1     1
## 2       51    2     1     1

t.test(ChickWeight[, 1], mu = 100)

##
## One Sample t-test
##
## data: ChickWeight[, 1]
## t = 7.38, df = 577, p-value = 5.529e-13
## alternative hypothesis: true mean is not equal to 100
## 95 percent confidence interval:
##  116.0 127.6
## sample estimates:
## mean of x
##      121.8
##
```

Two sample t test. Note these are equivalent

```
t.test(ChickWeight$weight[ChickWeight$Diet == "1"],
ChickWeight$weight[ChickWeight$Diet ==
  "2"])

##
## Welch Two Sample t-test
##
## data: ChickWeight$weight[ChickWeight$Diet == "1"] and ChickWeight$weight[ChickW
## t = -2.638, df = 201.4, p-value = 0.008995
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -34.900 -5.042
## sample estimates:
```

```
## mean of x mean of y
##      102.6      122.6
##

t.test(weight ~ Diet, data = ChickWeight, subset = Diet %in%
        c("1", "2"))

##
##  Welch Two Sample t-test
##
## data:  weight by Diet
## t = -2.638, df = 201.4, p-value = 0.008995
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -34.900  -5.042
## sample estimates:
## mean in group 1 mean in group 2
##           102.6           122.6
##
```

For a pairwise comparisons or multiple testing use:

```
pairwise.t.test(ChickWeight$weight, ChickWeight$Diet, p.adjust.method =
"bonferroni")

##
##  Pairwise comparisons using t tests with pooled SD
##
## data:  ChickWeight$weight and ChickWeight$Diet
##
##      1      2      3
## 2 0.06838 -      -
## 3 2.5e-06 0.14077 -
## 4 0.00026 0.95977 1.00000
##
## P value adjustment method: bonferroni
```

7.2.2 adjusting for multiple testing

The *p.adjust* can be used to correct p-values for multiple testing. Adjustment methods include the Bonferroni correction in which the p-values are multiplied by the number of comparisons. Less conservative corrections are also included by Holm (1979) ("holm"), Hochberg (1988) ("hochberg"), Hommel (1988) ("hommel"), Benjamini and Hochberg (1995) ("BH" or its alias "fdr"), and Benjamini and Yekutieli (2001) ("BY")

```
x <- rnorm(50, mean = c(rep(0, 25), rep(3, 25)))
p <- 2 * pnorm(sort(-abs(x)))
```

```

pVal <- round(p, 3)
Bonferroni <- round(p.adjust(p, "bonferroni"), 3)

## FDR and BH are equivalent

FDR <- round(p.adjust(p, "fdr"), 3)
BH <- round(p.adjust(p, "BH"), 3)

res <- cbind(none = pVal, Bonferroni = Bonferroni, FDR = FDR,
             BH = BH)
res <- res[order(res[, "Bonferroni"]), ]
print(res[1:20, ])

##           none Bonferroni    FDR    BH
## [1,] 0.000      0.001 0.001 0.001
## [2,] 0.000      0.003 0.001 0.001
## [3,] 0.000      0.008 0.002 0.002
## [4,] 0.000      0.010 0.002 0.002
## [5,] 0.000      0.010 0.002 0.002
## [6,] 0.000      0.021 0.003 0.003
## [7,] 0.000      0.023 0.003 0.003
## [8,] 0.001      0.040 0.005 0.005
## [9,] 0.001      0.042 0.005 0.005
## [10,] 0.001      0.056 0.006 0.006
## [11,] 0.001      0.065 0.006 0.006
## [12,] 0.002      0.096 0.008 0.008
## [13,] 0.003      0.143 0.011 0.011
## [14,] 0.003      0.163 0.011 0.011
## [15,] 0.003      0.165 0.011 0.011
## [16,] 0.004      0.187 0.011 0.011
## [17,] 0.004      0.197 0.011 0.011
## [18,] 0.004      0.204 0.011 0.011
## [19,] 0.010      0.505 0.027 0.027
## [20,] 0.016      0.784 0.039 0.039

```

@

7.2.3 Continuous Data: One- and two-way analysis of variance

To run a one-way analysis of variance, use *lm*. To use *lm*, the input is a *vector* and a *factor*. Note here that Diet is a factor. The function *lm* provides limited information. Use *summary* to provide a short summary of the distribution of each of the variables. Extract the analysis of variance with *anova*.

```

lmDiet <- lm(weight ~ Diet, data = ChickWeight)
lmDiet

##
## Call:

```

```
## lm(formula = weight ~ Diet, data = ChickWeight)
##
## Coefficients:
## (Intercept)      Diet2      Diet3      Diet4
##      102.6       20.0       40.3       32.6
##

summary(lmDiet)

##
## Call:
## lm(formula = weight ~ Diet, data = ChickWeight)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -104.0   -53.6   -13.6    40.4   230.1
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   102.65      4.67    21.96 < 2e-16 ***
## Diet2         19.97      7.87     2.54  0.011 *
## Diet3         40.30      7.87     5.12  4.1e-07 ***
## Diet4         32.62      7.91     4.12  4.3e-05 ***
## ---
## Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1 1
##
## Residual standard error: 69.3 on 574 degrees of freedom
## Multiple R-squared:  0.0535, Adjusted R-squared:  0.0485
## F-statistic: 10.8 on 3 and 574 DF,  p-value: 6.43e-07
##

anova(lmDiet)

## Analysis of Variance Table
##
## Response: weight
##              Df Sum Sq Mean Sq F value Pr(>F)
## Diet           3  155863   51954    10.8 6.4e-07 ***
## Residuals    574 2758693    4806
## ---
## Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1 1
```

In some statistical packages, the sum of the squares are labeled "between groups" and "within groups". Since `lm` and `anova` tables are used for a wide range of statistical models, the output from R is different. The Between groups sum of the squares is labeled by the name of the factor groupings (Diet). The within sum of the squares is labeled Residuals. The `aov` function is a wrapper which calls `lm`, but express the results these in the traditional language of the analysis of variance rather than that of linear models.

For examples of different analysis of variance (using *aov*) at <http://personality-project.org/r/r.anova.html>

```
aov(weight ~ Diet, data = ChickWeight)

## Call:
## aov(formula = weight ~ Diet, data = ChickWeight)
##
## Terms:
##              Diet Residuals
## Sum of Squares 155863    2758693
## Deg. of Freedom      3         574
##
## Residual standard error: 69.33
## Estimated effects may be unbalanced
```

For a two-way analysis of variance, provide a second factor to *lm*

```
lmDiet <- lm(weight ~ Diet + Time, data = ChickWeight)
summary(lmDiet)

##
## Call:
## lm(formula = weight ~ Diet + Time, data = ChickWeight)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -136.8   -17.1    -2.6    15.0   141.8
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   10.924      3.361    3.25   0.0012 **
## Diet2         16.166      4.086    3.96   8.6e-05 ***
## Diet3         36.499      4.086    8.93   < 2e-16 ***
## Diet4         30.233      4.107    7.36   6.4e-13 ***
## Time           8.750      0.222   39.45   < 2e-16 ***
## ---
## Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1 1
##
## Residual standard error: 36 on 573 degrees of freedom
## Multiple R-squared:  0.745, Adjusted R-squared:  0.744
## F-statistic: 419 on 4 and 573 DF, p-value: <2e-16
##

anova(lmDiet)

## Analysis of Variance Table
##
```



```
## Response: weight
##           Df Sum Sq Mean Sq F value Pr(>F)
## Diet       3  155863    51954    40.1 <2e-16 ***
## Time       1 2016357 2016357  1556.4 <2e-16 ***
## Residuals 573   742336     1296
## ---
## Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1 1
```

7.2.4 Discrete Data: Contingency Table

A contingency table (also referred to as cross tabulation or cross tab) is often used to record and analyze the relation between two or more discrete/categorical variables. It displays the (multivariate) frequency distribution of the variables in a matrix format. Test for association between such categorical variables are very common in research. Given two *factors* of at least 2 (usually unordered) levels, you can use the package `vcd` to compute several association statistics for a contingency table.

The simplest measure of association between two categorical variables is the ϕ coefficient defined by

$$\phi = \sqrt{\frac{\chi^2}{N}}$$

with N is the total number of observation and $\chi^2 = \sum_{i=1}^n \frac{(O_i - E_i)^2}{E_i}$

where

χ^2 = Pearson's cumulative test statistic

O_i = an observed frequency;

E_i = an expected (theoretical) frequency, asserted by the null hypothesis;

n = the number of cells in the table.

Another measure of association is the Cramer's V statistic that is generalizable to rectangular contingency table

$$V = \sqrt{\frac{\chi^2}{N(k-1)}},$$

k being the number of rows or the number of columns, whichever is less.

See http://en.wikipedia.org/wiki/Contingency_table for more details about measure of association in a contingency table.

In R, first create a contingency table and then use the function `assocstats` to computes the Pearson chi-Squared test, the Likelihood Ratio, chi-Squared test, the phi coefficient, the contingency coefficient, and Cramer's V statistics.

```
## load library
library(vcd)

## Loading required package: colorspace
```

```

## load data
attach(Arthritis)

## check the variables of interest
is.factor(Arthritis$Treatment)

## [1] TRUE

print(levels(Arthritis$Treatment))

## [1] "Placebo" "Treated"

is.factor(Arthritis$Improved)

## [1] TRUE

print(levels(Arthritis$Improved))

## [1] "None" "Some" "Marked"

## build the contingency table
tab <- table(Arthritis$Treatment, Arthritis$Improved)
print(tab)

##
##           None Some Marked
## Placebo    29   7     7
## Treated    13   7    21

## compute statistics
res <- assocstats(tab)
print(res)

##
##           X^2 df  P(> X^2)
## Likelihood Ratio 13.530  2 0.0011536
## Pearson          13.055  2 0.0014626
##
## Phi-Coefficient   : 0.394
## Contingency Coeff.: 0.367
## Cramers V         : 0.394

detach(Arthritis)

```

The structure of the `res` object can be printed using the `str` function

```
str(res)

## List of 5
## $ table      : table int [1:2, 1:3] 29 13 7 7 7 21
## ..- attr(*, "dimnames")=List of 2
## .. ..$ : chr [1:2] "Placebo" "Treated"
## .. ..$ : chr [1:3] "None" "Some" "Marked"
## $ chisq_tests: num [1:2, 1:3] 13.52981 13.05502 2 2 0.00115 ...
## ..- attr(*, "dimnames")=List of 2
## .. ..$ : chr [1:2] "Likelihood Ratio" "Pearson"
## .. ..$ : chr [1:3] "X^2" "df" "P(> X^2)"
## $ phi        : num 0.394
## $ contingency: num 0.367
## $ cramer      : num 0.394
## - attr(*, "class")= chr "assocstats"
```

You can easily access the various statistics from the `res` object

```
## Pearson chi squared test
print(res$chisq_tests[2, 1])

##      X^2      df  P(> X^2)
## 13.055020 2.000000 0.001463

## Cramer's V statistic
print(res$cramer)

## [1] 0.3942
```

If you want to compute the agreement between two classifications or raters, you can estimate the κ coefficient which can have the following typical values

Kappa value	magnitude of agreement
< 0	no
0 - 0.2	small
0.2 - 0.4	fair
0.4 - 0.6	moderate
0.6 - 0.8	substantial
0.8 - 1	almost perfect

```
## two random classification
set.seed(12345)
c1 <- sample(0:1, 100, replace = TRUE)
c2 <- sample(0:1, 100, replace = TRUE)
tab <- table(C1 = c1, C2 = c2)
Kappa(x = tab, weights = matrix(rep(1, 4), ncol = 2))

##      value      ASE
```

```
## Unweighted -0.04839 0.10073
## Weighted      NaN 0.08598
```

In a practical situation, your Kappa coefficient needs to be over 0.6 to claim that your categorization is valid. You may also want to report both the agreement (%)

```
agr <- sum(diag(tab)) / sum(tab)
cat(sprintf("Agreement: %.2g%%\n", agr))

## Agreement: 0.48%
```

7.2.5 Common statistical Tests in R

Here is a quick (incomplete) list of useful R functions for basic statistical comparisons:

- Continuous Data
 - *t.test*
 - *pairwise.t.test*: pairwise comparisons
 - *var.test*: comparison of two variances.
 - *lm(y~x)*: linear regression analysis
 - *lm(y~f1)*: one-way analysis of variance
 - *lm(y~f1+f2)*: two-way analysis of variance (ANOVA), f1 and f2 are factors.
 - *lm(y~f1+x)*: analysis of co-variance
 - *lm(y~x1+x2+x3)*: multiple regression analysis
 - *bartlett.test*: Bartlett’s test of the null that the variances in each of the groups (samples) are the same
- Non-Parametric
 - *wilcox.test*: one- and two-sample Wilcoxon tests on vectors of data; the latter is also known as Mann-Whitney test.
 - *kruskal.test*: non-parametric one-way analysis of variance.
 - *friedman.test*: non-parametric two-way analysis of variance.
- Correlation
 - cor*, *cor.test*: correlation and Correlation tests. *Cor.test* methods include "kendall" "spearman" or "pearson".
- Discrete response data

- *chisq.test*: chi-squared contingency table tests, *fisher.test* exact test for small tables.
- *binom.test*: binomial test
- *prop.test*: *prop.trend.test* comparison of proportions.
- *glm(y ~ x1+x2+x3, binomial)*: logistic regression

* As a complete aside and to continue stories of Ireland's mathematicians and statisticians, which I started with the story of George Boole, the first professor of mathematics of University College Cork. The t statistic was introduced by William Sealy Gosset to monitoring the quality of brewing in the Guinness brewery in Dublin, Ireland. Guinness's has an innovative policy of recruiting the best graduates from Oxford and Cambridge to apply biochemistry and statistics to Guinness' industrial processes. Gosset published the t test in Biometrika in 1908, but published using the pen name Student.

7.3 Model formulae and model options

- Most modeling done in a standard way
- Data set is usually a single *data frame* object
- Model is fitted using *model fitting function*
- Form of the model specified by a *formula*
- Resulting *fitted model* object can be interrogated, analyzed and modified

Basic output of the model fitting process is minimal. Details obtained via extractor functions.

7.3.1 Model formulae

We have already seen in several functions (boxplot, t.test, lm) that a simply function is defined by $y \sim x$. We will now discuss formulae in much more detail.

Define a template for statistical models

$$y_i = \sum_{j=0}^p \beta_j x_{ij} + \epsilon_i, \quad \epsilon_i \sim \text{NID}(0, \sigma^2), \quad i = 1, \dots, n$$

In the matrix form this model is

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

where \mathbf{y} is the response vector, \mathbf{X} is the *model matrix* or *design matrix* with columns $\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_p$.
NOTATION:

$y, x, x_0, x_1, x_2, \dots$ - numeric variables
 A, B, C, \dots - factors

Examples of model *formula* with numeric variables:

```
y ~ x
y ~ 1 + x          - simple regression (first - implicit intercept,
                    second - explicit intercept)
-----
y ~ 0 + x
y ~ x - 1
y ~ -1 + x         - regression through the origin
-----
y ~ x1 + x2 + x3   - multiple regression
-----
y ~ x + I(x^2)     - quadratic regression
-----
log(y) ~ x1 + x2   - multiple regression of transformed variable
```

Examples of model *formula* with factors and numeric variables:

```
y ~ A              - single analysis of variance model
y ~ A + x          - single analysis of covariance model with
                    covariate x
-----
y ~ A*B
y ~ A + B + A:B    - two-factor model with interaction
-----
y ~ (A + B + C)^2  - all two-factor interactions
-----
y ~ A*B + Error(C) - two factor model with interaction and error
                    strata determined by C
```

General form:

```
response ~ op_1 term_1 + op_2 term_2 + ...
```

where

`response` - a vector or expression evaluating to a vector defining the response variable
`op_i` - an operator, either + (inclusion of a term) or - (exclusion of a term) in the model
`term_i` - either a vector or matrix expression, or 1, a factor, or a formula expression consisting of factors, vectors or matrices connected by formula operators.

7.3.2 Example of linear regression

Will use data 'cats' from the MASS library.

```
library(MASS)
help("cats")
str(cats)

## data.frame: 144 obs. of 3 variables:
## $ Sex: Factor w/ 2 levels "F","M": 1 1 1 1 1 1 1 1 1 1 ...
## $ Bwt: num 2 2 2 2.1 2.1 2.1 2.1 2.1 2.1 2.1 ...
## $ Hwt: num 7 7.4 9.5 7.2 7.3 7.6 8.1 8.2 8.3 8.5 ...

cats.lmB <- lm(Hwt ~ Bwt, data = cats)
cats.lmS <- lm(Hwt ~ Sex, data = cats)
cats.lmBS <- lm(Hwt ~ Bwt + Sex, data = cats)
cats.lmBxS <- lm(Hwt ~ Bwt * Sex, data = cats)
cats.lmB2 <- lm(Hwt ~ Bwt + I(Bwt^2), data = cats)
```


7.3.3 Contrasts, model.matrix

We need to understand how model formulae specify the columns of the model matrix.

1. Continuous variables (simplest): each variable provides a column of the model matrix (and the intercept will provide a column of ones if included in the model).

2. k-level factor A

The answer differs for unordered and ordered factors.

- Unordered factors

$k - 1$ columns are generated for the indicators of the second, third, . . . , up to k^{th} levels of the factor. (Implicit parameterization is to contrast the response at each level with that at the first.)

- Ordered factors

$k - 1$ columns are the orthogonal polynomials on $1, \dots, k$, omitting the constant term.

If the intercept is omitted in a model that contains a factor term, the first such term is encoded into k columns giving the indicators for all the levels.

R default setting is:

```
options(contrasts = c("contr.treatment", "contr.poly"))
```

Contrasts can be defined using the *contrasts* or *C* function **Example**

```
contr.treatment(n = 3, base = 2)
```

```
##      1 3
## 1 1 0
## 2 0 0
## 3 0 1
```

```
contr.sum(n = 3)
```

```
##      [,1] [,2]
## 1      1      0
## 2      0      1
## 3     -1     -1
```

7.4 Exercise 9

```
# What is the difference in output?
lm(Hwt ~ Sex, data = cats)

##
## Call:
## lm(formula = Hwt ~ Sex, data = cats)
##
## Coefficients:
## (Intercept)      SexM
##          9.20         2.12
##

lm(Hwt ~ Sex - 1, data = cats)

##
## Call:
## lm(formula = Hwt ~ Sex - 1, data = cats)
##
## Coefficients:
## SexF  SexM
##   9.2  11.3
##
```

model.matrix is useful to view the terms in the fitted model.

7.5 Output and extraction from fitted models

As mentioned earlier, the printed output of the model fit is minimal. However, the value of a fitted model object is stored in an object. Information about the fitted model can be displayed, extracted and plotted.

Extractor functions:

```
coef(obj)           - regression coefficients
resid(obj)          - residuals
fitted(obj)         - fitted values
summary(obj)        - analysis summary
predict(obj, newdata = ndat) - predict for new data
deviance(obj)       - residual sum of squares
print(obj)          - Print concise summary
plot(obj)           - produce diagnostic plots
formula(obj)        - extract the model formula
anova(obj1, obj2)   - compare 2 models (one is a submodel with the outer model)
step(obj)           - add, drop terms
update(obj, formula) - update a model with a new formula
```

Let's go back to the `cats` example. Now, we can extract more information about the fits.

```
attach(cats)
cats.lmBS <- lm(Hwt ~ Bwt + Sex, data = cats)
coef(cats.lmBS)

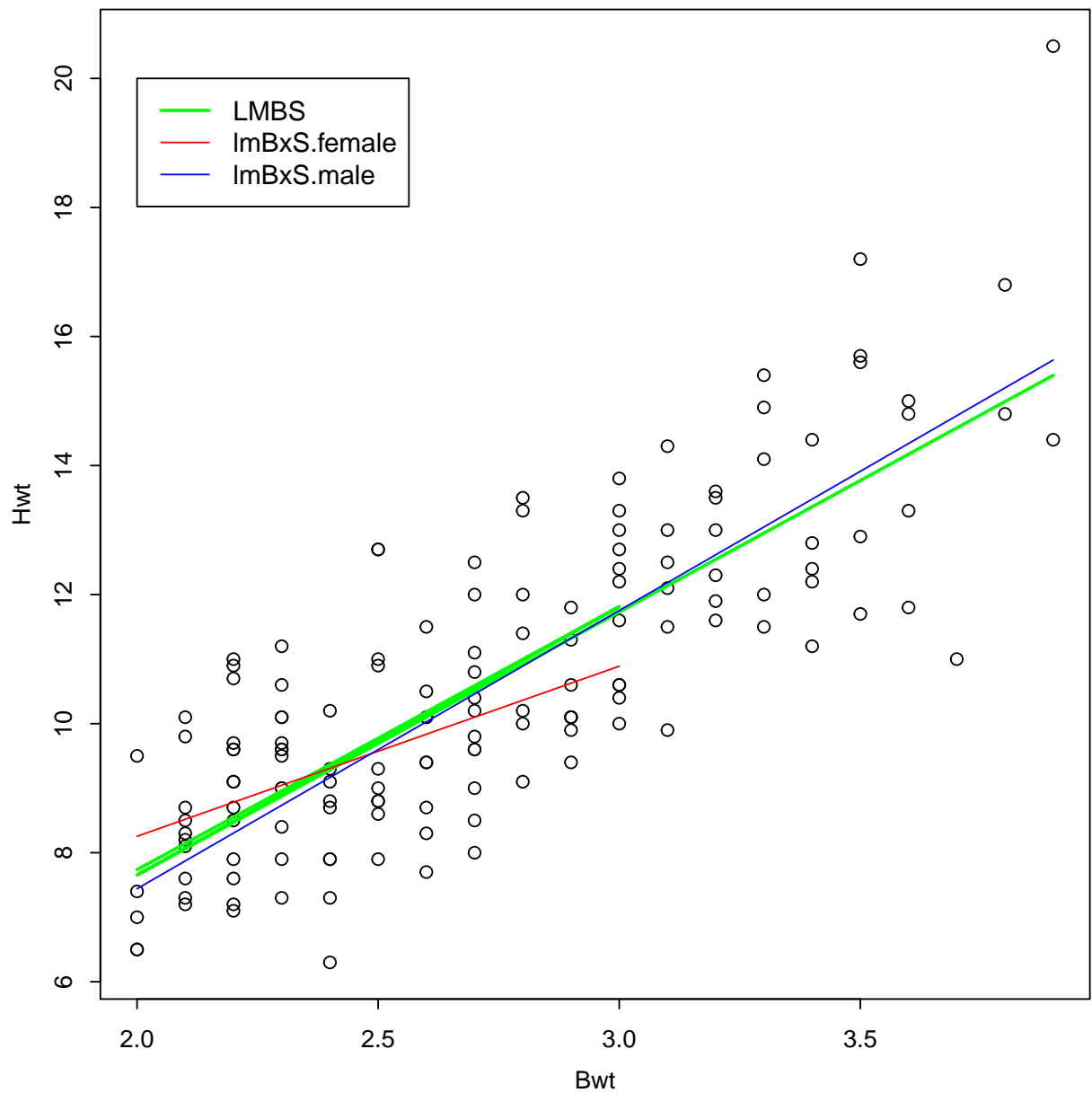
## (Intercept)      Bwt      SexM
##      -0.4150      4.0758     -0.0821

fit.catsBS <- fitted(cats.lmBS)
```

```
cats.lmBxS <- lm(Hwt ~ Bwt * Sex, data = cats)
fit.catsBxS <- fitted(cats.lmBxS)
```

Plot these 2 models.

```
plot(Bwt, Hwt)
lines(Bwt, fit.catsBS, col = "green", lwd = 2) # OR
# abline(cats.lmBS, col='green', lwd=2)
lines(Bwt[Sex == "F"], fit.catsBxS[Sex == "F"], col = "red")
lines(Bwt[Sex == "M"], fit.catsBxS[Sex == "M"], col = "blue")
legend(x = 2, y = 20, legend = c("LMBS", "lmBxS.female", "lmBxS.male"),
      col = c("green", "red", "blue"), lwd = c(2, 1, 1))
```



Prediction of Hwt values

```
predict(cats.lmBxS, data.frame(Bwt = seq(2, 5, 1), Sex = "M"))

##      1      2      3      4
## 7.441 11.754 16.067 20.379

detach(cats)
```

Some more useful, but non-standard, ways of extracting information from a model.

```
df.residual(obj) - residual degrees of freedom
names(obj)       - gives names of the components in obj
```

`names(summary(obj))` - gives names of the components in `summary(obj)`

How to get the residual variance of the fit? There are at least 2 ways. The first is the direct calculation

```
var.catsB <- deviance(cats.lmB)/df.residual(cats.lmB) # direct  
calculation  
var.catsB <- summary(cats.lmB)$sigma^2
```

7.6 Exercise 10 :Multivariate linear regression

- Read the data contained in the file `lungs.csv` from the course website into R. Fit a multivariate regression model (function *lm*) of `pemax` using all variables. Call the result `lungFit`.
- Which terms appear to be significant (summary)?
- What is the residual error of this model?
- Which are the most and least significant variables in this model?

7.6.1 Residual plots, diagnostics

Important part of modeling - checking the model assumptions. Some easy to check and interpret model diagnostics

1. Fit to the data (predictions) vs. raw data (observations)
2. Histogram of the residuals
3. Scatterplot of the residuals vs. fitted values
4. QQ-plot of the residuals

```
attach(cats)
plot(Bwt, Hwt, main = "Model fit")
abline(cats.lmB, col = "green", lwd = 2)
```

Fit the model

```
hist(resid(cats.lmB), main = "Residual histogram")
```

Residual histogram

```
plot(fitted(cats.lmB), resid(cats.lmB), main = "Residuals vs. fitted
values")
lines(lowess(fitted(cats.lmB), resid(cats.lmB)), col = "red")
abline(h = 0)
```

Residuals vs. fitted values

```
qqnorm(resid(cats.lmB))
qqline(resid(cats.lmB))
detach()
```

QQ-plot of residuals

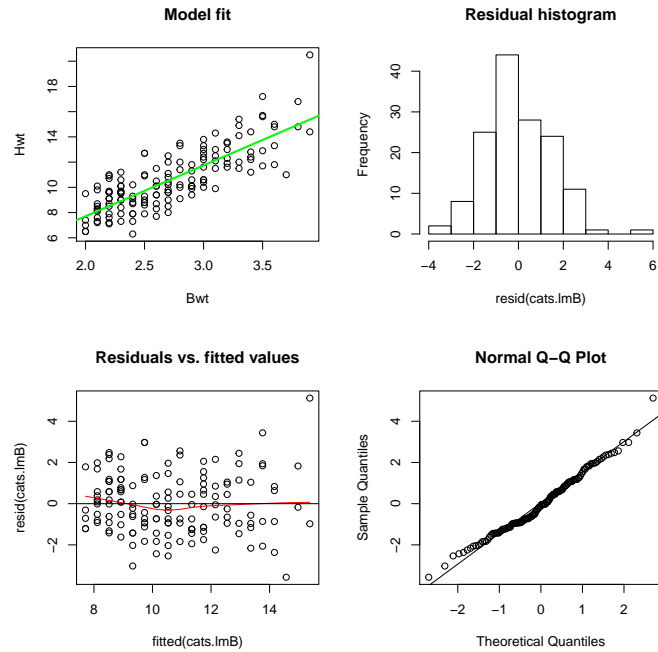
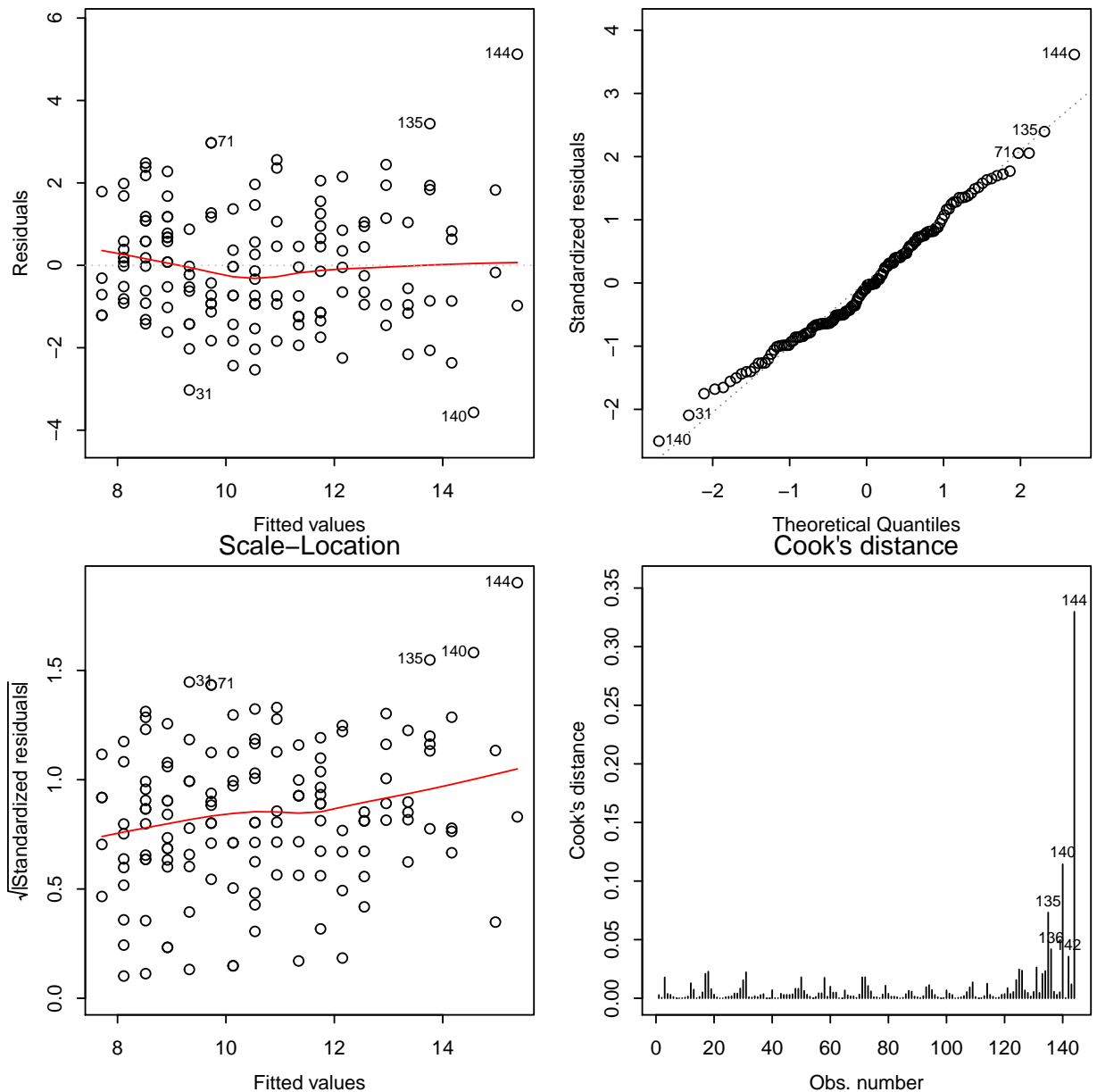


Figure 7.1: Plots of the fitted Model, Residuals vs. fitted values and QQ-Plots of the residuals

Default plots available in *plot* of class *lm*

```
par(mfrow = c(2, 2))
plot(cats.lmB, which = 1:4, id.n = 5)
```

7.6.2 ANOVA and updating models

Anova tables for a sequence of fitted models

`anova(obj_1, obj_2)` - compare two models where `obj_1` and `obj_2` are two regression

The sums of squares shown are the decrease in the residual sums of squares resulting from an inclusion of that term in the model at that place in the sequence. Only for orthogonal experiments will the order of inclusion be inconsequential.

```
anova(cats.lmB, cats.lmBS)

## Analysis of Variance Table
##
## Model 1: Hwt ~ Bwt
## Model 2: Hwt ~ Bwt + Sex
##   Res.Df RSS Df Sum of Sq    F Pr(>F)
## 1      142 300
## 2      141 299  1      0.155 0.07  0.79
```

```
anova(cats.lmB, cats.lmBxS)

## Analysis of Variance Table
##
## Model 1: Hwt ~ Bwt
## Model 2: Hwt ~ Bwt * Sex
##   Res.Df RSS Df Sum of Sq    F Pr(>F)
## 1      142 300
## 2      140 291  2      8.49 2.04  0.13
```

```
anova(cats.lmB, cats.lmBS, cats.lmBxS)

## Analysis of Variance Table
##
## Model 1: Hwt ~ Bwt
## Model 2: Hwt ~ Bwt + Sex
## Model 3: Hwt ~ Bwt * Sex
##   Res.Df RSS Df Sum of Sq    F Pr(>F)
## 1      142 300
## 2      141 299  1      0.15 0.07  0.785
## 3      140 291  1      8.33 4.01  0.047 *
## ---
## Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1 1
```

The *update()* function allows a model to be fitted that differs from one previously fitted usually by just a few additional or removed terms.

Syntax:

```
new.model <- update(old.model, new.formula)
```

Special name in the *new formula* - a period '.' - can be used to stand for “corresponding part of the old model formula”.

Example:

Data set *mtcars*, fuel consumption and 10 aspects of automobile design and performance for 32 automobiles.

```
help("mtcars")
cars.lm <- lm(mpg ~ hp + wt, data = mtcars)
cars.lm2 <- update(cars.lm, . ~ . + disp)
```

```
# cars.lms <- update(cars.lm2, sqrt(.) ~ .)
```

what does the following do?

```
# anova(cars.lm, cars.lm2, cars.lms)
anova(cars.lm, cars.lm2)

## Analysis of Variance Table
##
## Model 1: mpg ~ hp + wt
## Model 2: mpg ~ hp + wt + disp
##   Res.Df RSS Df Sum of Sq    F Pr(>F)
## 1      29 195
## 2      28 195  1    0.0571 0.01  0.93
```

7.6.3 Model selection

There are functions in R automating the choice of terms in the statistical models. Adding, dropping and performing stepwise selection in a sequence of models.

- Dropping terms: `drop1()` or `dropterm()` from the MASS library
- Adding terms: `add1()` or `addterm()` from the MASS library
- Stepwise selection: `step()` or `stepAIC()` from the MASS library

More details:

dropterm Fits all models that differ from the current model by dropping a single term, maintaining marginality.

```
dropterm(model.big, test=F)      # for linear models
dropterm(model.big, test=Chisq)  # for generalized linear models
```

addterm Fits all models that differ from the current model by adding a single term from those supplied, maintaining marginality

```
addterm(model.small, scope=model.big, test=F) # for linear models
```

stepAIC Performs stepwise model selection by exact AIC

```
stepAIC(model.small,
        scope=list(upper=model.big, lower=~1),
        test=F)      # for linear models
```

Example

Data set `mtcars`

```
# Fit regression model with all other covariates as predictors
cars.all <- lm(mpg ~ ., data=mtcars)
# Drop terms
dropterm(cars.all, test = "F")

# Probably not appropriate analysis. Try
cars.some <- lm(mpg ~ factor(cyl) + hp + wt + disp + qsec + factor(gear),
               data=mtcars)
dropterm(cars.some, test = "F")

# Start with one variable
cars.sm <- update(cars.some, .~ wt)
addterm(cars.sm, cars.some, test=F)

# Stepwise selection
cars.step <- stepAIC(cars.some, scope=list(lower = ~ wt))
```

7.7 Cross-validation

Another approach, developed by the machine Learning community, is *cross-validation*. The idea is to sequentially divide the dataset in training and test sets to fit and assess the performance of the model, respectively. This approach enables to use all the observations both for training and testing the prediction model.

Here is an example of a 10-fold cross-validation on the `mtcars` dataset where we compare the model with one variable (`wt`) and all the variables to predict `mpg`. Once the root mean squared error (RMSE) is computed for each fold, a paired Wilcoxon Rank Sum test is used to compare the performance of the small and big models.

```
nfold <- 10
## nr is the number of observations
nr <- nrow(mtcars)
## nfold is the number of folds in the cross-validation
if (nfold > 1) k <- floor(nr/nfold) else {
  k <- 1
  nfold <- nr
}
smpl <- sample(nr)
mse.big <- mse.small <- NULL

for (i in 1:nfold) {
  if (i == nfold)
    s.ix <- smpl[c(((i - 1) * k + 1):nr)] else s.ix <- smpl[c(((i -
1) * k + 1):(i * k))]
  ## fit the model
```

```

mm.big <- lm(mpg ~ ., data = mtcars[-s.ix, , drop = FALSE])
mm.small <- lm(mpg ~ wt, data = mtcars[-s.ix, , drop = FALSE])
## assess the performance of the model
pp.big <- predict(object = mm.big, newdata = mtcars[s.ix,
!is.element(colnames(mtcars),
  "mpg")])
pp.small <- predict(object = mm.small, newdata = mtcars[s.ix,
!is.element(colnames(mtcars),
  "mpg")])
## compute mean squared error (MSE)
mse.big <- c(mse.big, sqrt(mean((mtcars[s.ix, "mpg"] - pp.big)^2)))
mse.small <- c(mse.small, sqrt(mean((mtcars[s.ix, "mpg"] -
pp.small)^2)))
}
names(mse.big) <- names(mse.small) <- paste("fold", 1:nfold,
  sep = ".")

## compare the performance of the big and small models using a
## Wilcoxon Rank Sum test
wilcox.test(mse.big, mse.small, paired = TRUE, alternative = "less")

##
## Wilcoxon signed rank test
##
## data: mse.big and mse.small
## V = 35, p-value = 0.7842
## alternative hypothesis: true location shift is less than 0
##

```

As can be seen, there is not enough evidence in the dataset to claim that the big prediction model outperforms the small one ($p\text{-value} > 0.05$). You can easily change the number of folds in the cross-validation by setting the variable *nfold* to another value, *nfold* = 1 for leave-one-out cross-validation.

7.8 Statistical models

Will talk today about 3 classes of statistical models: linear regression, generalized linear models (e.g. logistic and Poisson regression), and survival models.

7.8.1 Linear Regression: Weighted Models, Missing Values

We have talked and went through examples of linear regression using the function `lm()`. Will expand here on the options for the function `lm()`

```
lm(formula, data, subset, weights, na.action, ...)
```

subset - (optional) a subset of observations to be used in the fitting process

weights - (optional) weights to fit the model using weighted list squares method

na.action - what happens to data containing missing values 'NA's;

na.omit - is the default; another option *na.fail*

```
attach(ChickWeight)
time.wgt <- tapply(weight, Time, var)
time.wgt.rep <- as.numeric(time.wgt[match(Time,
as.numeric(names(time.wgt)))])
detach(2)
Chick.anl <- data.frame(ChickWeight, time.wgt.rep = time.wgt.rep)
chick.lm.wgt <- lm(weight ~ Time, data = Chick.anl, weight =
1/time.wgt.rep)
chick.lm.T0 <- lm(weight ~ Time, data = Chick.anl, subset = (Time ==
0))
```

7.8.2 Generalized linear modeling

- One generalization of multiple linear regression.
- Response, y , predictor variables x_1, x_2, \dots, x_p
- The distribution of Y depends on the x 's through a single linear function, the 'linear predictor'

$$\nu = \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_p x_p \quad (7.1)$$

with x_i having no influence on y if and only if $\beta_i = 0$

- There may be an unknown 'scale' (or 'variance') parameter ϕ to estimate as well
- The mean, μ , is a smooth invertible function of the linear predictor

$$\mu = m(\nu), \quad \nu = m^{-1}(\mu) = l(\mu) \quad (7.2)$$

and this inverse function, $l()$, is called the *link function*

- The deviance is a generalization of the residual sum of squares.
- The protocols are very similar to linear regression and the inferential logic is virtually identical.

The class of generalized linear models handled by facilities supplied in R includes gaussian, binomial, poisson, inverse gaussian and gamma response distributions.

Families of distributions and links

Distribution	Link
-----	-----
binomial	logit, probit, log, cloglog
gaussian	identity, log, inverse
Gamma	identity, inverse, log
inverse.gaussian	1/mu^2, identity, inverse, log
poisson	identity, log, sqrt

The R function to fit a generalized linear model is `glm()` which uses the form

```
fitted.model <- glm(formula, family=family.generator, data=data.frame)
```

The only difference from `lm()` is the `family.generator`, which is the instrument by which the family is described. It is the name of a function that generates a list of functions and expressions that together define and control the model and estimation process.

We will concentrate on the *binomial* family with the *logit* link or as you probably know it 'logistic regression',.

Logistic regression

To fit a binomial model using `glm()` there are three possibilities for the response:

1. If the response is a vector it is assumed to hold binary data, and so must be a 0/1 vector.
2. If the response is a two-column matrix it is assumed that the first column holds the number of successes for the trial and the second holds the number of failures.

3. If the response is a factor, its first level is taken as failure (0) and all other levels as 'success' (1).

Syntax:

```
glm(y ~ x, family=binomial(link=logit), data = data.frame)
```

Link is optional, since the default link is *logit*. Necessary, if another link is desired, e.g. *probit*.

Example of logistic regression using data set *esophagus*

Data from a case-control study of (o)esophageal cancer in Ile-et-Vilaine, France containing records for 88 age/alcohol/tobacco combinations with the 3 covariates grouped into 6, 4 and 4 groups respectively.

```
summary(esoph)
```

```
##      agegp          alcgp          tobgp          ncases
## 25-34:15  0-39g/day:23  0-9g/day:24  Min.      : 0.00
## 35-44:15  40-79      :23  10-19      :24  1st Qu.: 0.00
## 45-54:16  80-119     :21  20-29      :20  Median : 1.00
## 55-64:16  120+       :21  30+        :20  Mean    : 2.27
## 65-74:15                                     3rd Qu.: 4.00
## 75+       :11                                     Max.    :17.00
##      ncontrols
## Min.      : 1.0
## 1st Qu.: 3.0
## Median   : 6.0
## Mean     :11.1
## 3rd Qu.:14.0
## Max.     :60.0
```

effects of alcohol and tobacco, age-adjusted

```
eso.age <- glm(cbind(ncases, ncontrols) ~ agegp, data = esoph,
              family = binomial())
```

```
eso.base <- glm(cbind(ncases, ncontrols) ~ agegp + tobgp +
               alcgp, data = esoph, family = binomial())
```

```
eso.base <- update(eso.age, . ~ . + tobgp + alcgp)
```

```
eso.TA <- glm(cbind(ncases, ncontrols) ~ agegp + tobgp * alcgp,
              data = esoph, family = binomial())
```

```
eso.2way <- glm(cbind(ncases, ncontrols) ~ (agegp + tobgp +
               alcgp)^2, data = esoph, family = binomial())
```

Stepwise model selection

```
eso.base <- glm(cbind(ncases, ncontrols) ~ agegp + tobgp +
               alcgp, data = esoph, family = binomial())
```

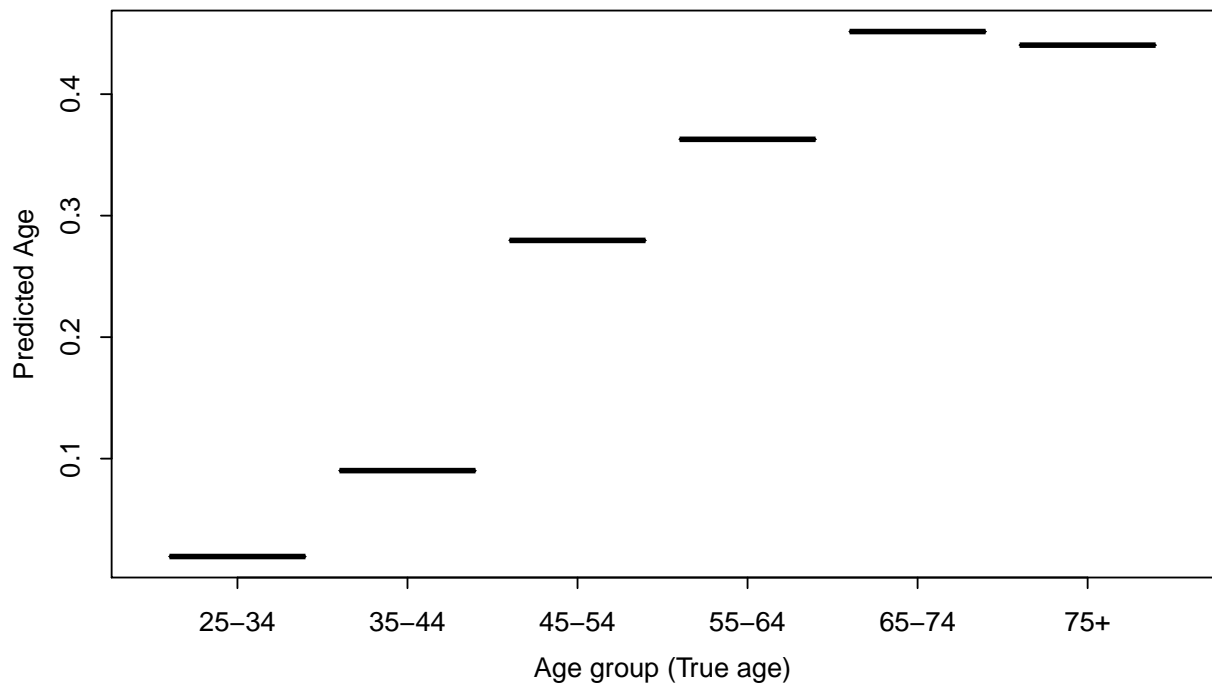
```
eso.stp <- stepAIC(eso.age, scope = list(upper = ~agegp +
               tobgp + alcgp, lower = ~1), test = "Chisq")
```



```
## Start:  AIC=298.6
## cbind(ncases, ncontrols) ~ agegp
##
##           Df Deviance AIC   LRT Pr(Chi)
## + alcgp   3      64.6 230 74.5 4.5e-16 ***
## + tobgp   3     120.0 286 19.1 0.00026 ***
## <none>          139.1 299
## - agegp   5     227.2 377 88.1 < 2e-16 ***
## ---
## Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1 1
##
## Step:  AIC=230.1
## cbind(ncases, ncontrols) ~ agegp + alcgp
##
##           Df Deviance AIC   LRT Pr(Chi)
## + tobgp   3      54.0 226 10.6  0.014 *
## <none>          64.6 230
## - agegp   5     138.8 294 74.2 1.4e-14 ***
## - alcgp   3     139.1 299 74.5 4.5e-16 ***
## ---
## Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1 1
##
## Step:  AIC=225.4
## cbind(ncases, ncontrols) ~ agegp + alcgp + tobgp
##
##           Df Deviance AIC   LRT Pr(Chi)
## <none>          54.0 226
## - tobgp   3      64.6 230 10.6  0.014 *
## - alcgp   3     120.0 286 66.1 3.0e-14 ***
## - agegp   5     131.5 293 77.5 2.8e-15 ***
## ---
## Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1 1
```

Prediction and residuals Plot the fitted values for 'age' effect

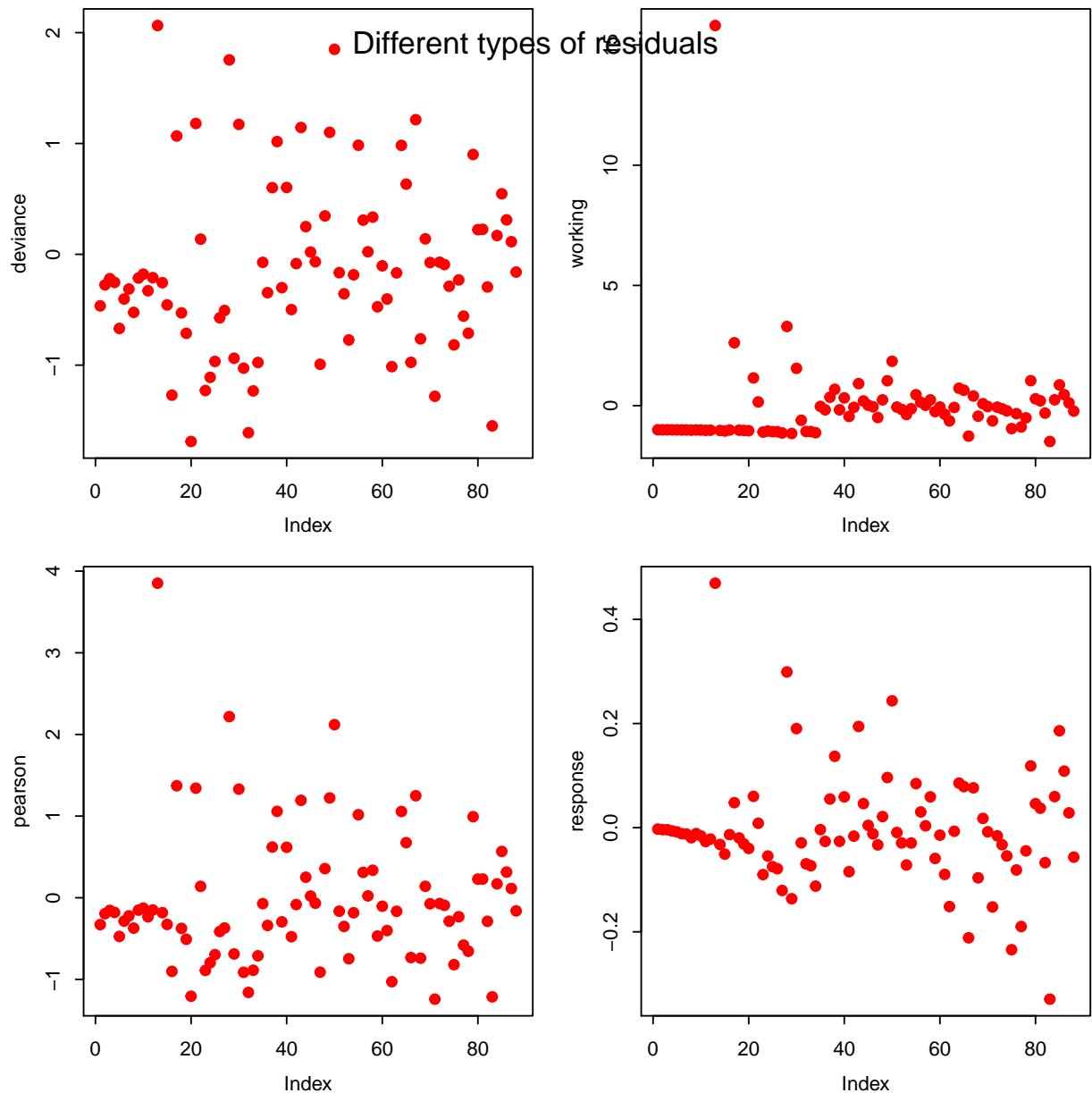
```
attach(esoph)
eso.pred.age <- predict.glm(eso.base, data.frame(agegp = agegp,
  tobgp = rep("30+", 88), alcgp = rep("40-79", 88)), type =
"response")
plot(eso.pred.age ~ agegp, ylab = "Predicted Age", xlab = "Age group
(True age)")
```



```
detach(esoph)
```

4 types of residuals can be requested for the *glm()* models: *deviance*, *working*, *Pearson*, *response*

```
opar <- par(mfrow = c(2, 2))
for (i in c("deviance", "working", "pearson", "response"))
plot(resid(eso.base,
  type = i), ylab = i, pch = 19, col = "red")
mtext("Different types of residuals", line = -2, outer = T,
  cex = 1.2)
```



```
par(opar)
```

7.8.3 Other packages

- Tibshirani and Hastie's provide elastic net, lasso, ridge regression, adaptive lasso and the adaptive elastic net regularized generalized linear models available in the package `glmnet`
- Other packages for machine learning are listed on <http://cran.r-project.org/web/views/MachineLearning.html>

7.9 Survival modeling

Survival Analysis is a class of statistical methods for studying the occurrence and timing of events. These methods are most often applied to the study of deaths but can also handle different kinds of events, including the onset of disease and equipment failure for instance. For instance a disease consists of a transition from an healthy state to a diseased state. Moreover, the timing of the event is also considered for analysis.

Survival data have a common feature, namely *censoring*, that is difficult to handle with conventional statistical methods. Consider the following example, which illustrates the problem of censoring. A sample of breast cancer patients were followed during 10 years after diagnosis. The event of interest was the appearance of a distant metastasis (a tumor initiated from the primary breast tumor cells and that is located in another organ). The aim was to determine how the occurrence and timing of distant metastasis appearance depended on several variables.

7.9.1 Censored Data

An observation on a random variable t is right-censored if all you know about t is that it is greater than some value c . In survival analysis, t is typically the time of occurrence for some event, and cases are right-censored because observation is terminated before the event occurs.

Random censoring occurs when observations are terminated for reasons that are not under the control of the investigator. This situation can be illustrated in our example. Patients who are still free of distant metastasis after 10 years are censored by a mechanism identical to that applied to the singly right-censored data. But some patients may move away, and it may be impossible to contact them. Some patients may die from another cause. Still other patients may refuse to participate after, say, 5 years. These kinds of censoring are depicted in Figure 7.2, where the symbol "+" for the patients A and C indicates that observation is censored at that point in time.

Figure 7.2: Randomly censored data.

The vast majority of the he functions we need to do survival analysis are in the package `survival`. Check if the package `survival` is already loaded into your work space, if it isn't load the library `survival`

```
search()  
library(survival)  
  
## Loading required package: splines
```

We will work with the data set 'leukemia' containing times of death or censoring in patients with Acute Myelogenous Leukemia. The survival data are usually stored in a `Surv` object that is a one-column matrix containing the survival times and events/censoring.

```
data(leukemia)  
head(leukemia)
```

```
##      time status      x
## 1      9      1 Maintained
## 2     13      1 Maintained
## 3     13      0 Maintained
## 4     18      1 Maintained
## 5     23      1 Maintained
## 6     28      0 Maintained

`?`(Surv)
mysurv <- Surv(leukemia$time, leukemia$status)
head(mysurv)

## [1]  9  13  13+ 18  23  28+
```

Several methods for survival analysis are implemented in R, mainly in the **survival** package:

Surv - creates a survival object used as a response variable in a model formula,
e.g. *Surv(time, status)*

survfit - computes an estimate of a survival curve for censored data using the Kaplan-Meier method,
e.g. *survfit(Surv(time, status) ~ group)*

survdiff - Tests if there is a difference between two or more survival curves,
e.g. *survdiff(Surv(time, status) ~ group)*

survreg - regression for a parametric survival model with special case, the accelerated failure models that use a log transformation of the response.

coxph - fits a Cox proportional hazards regression model

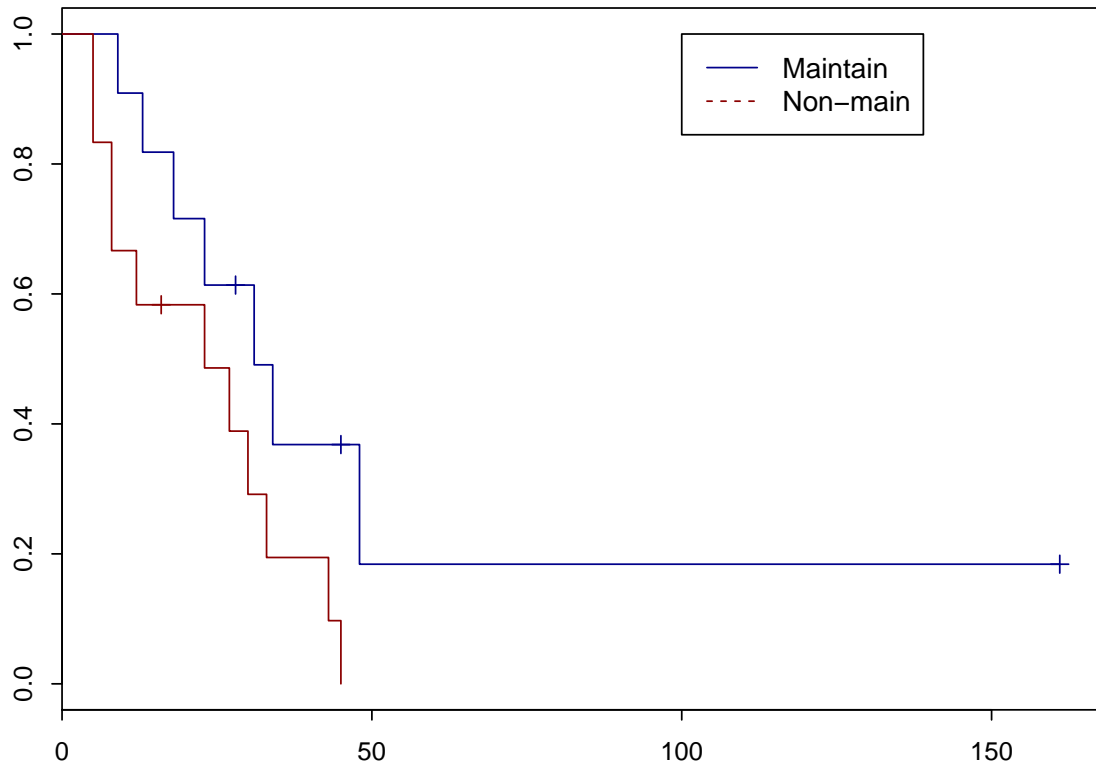
7.9.2 Kaplan-Meier curve estimation

We can easily draw the survival curve of patients representing the proportion of patients who survived over time. We provide the function *survfit* with the following set of arguments:

```
survfit(formula, data, weights, subset, na.action,
        newdata, individual=F, conf.int=.95, se.fit=T,
        type=c("kaplan-meier", "fleming-harrington", "fh2"),
        error=c("greenwood", "tsiatis"),
        conf.type=c("log", "log-log", "plain", "none"),
        conf.lower=c("usual", "peto", "modified"))
```

```
# To see help on this function or a description of these arguments
?survfit
```

```
leuk.km <- survfit(Surv(time, status) ~ x, data = leukemia)
plot(leuk.km, lty = 1, col = c("darkblue", "darkred"))
legend(100, 1, legend = c("Maintain", "Non-main"), lty = 1:2,
      col = c("darkblue", "darkred"))
```



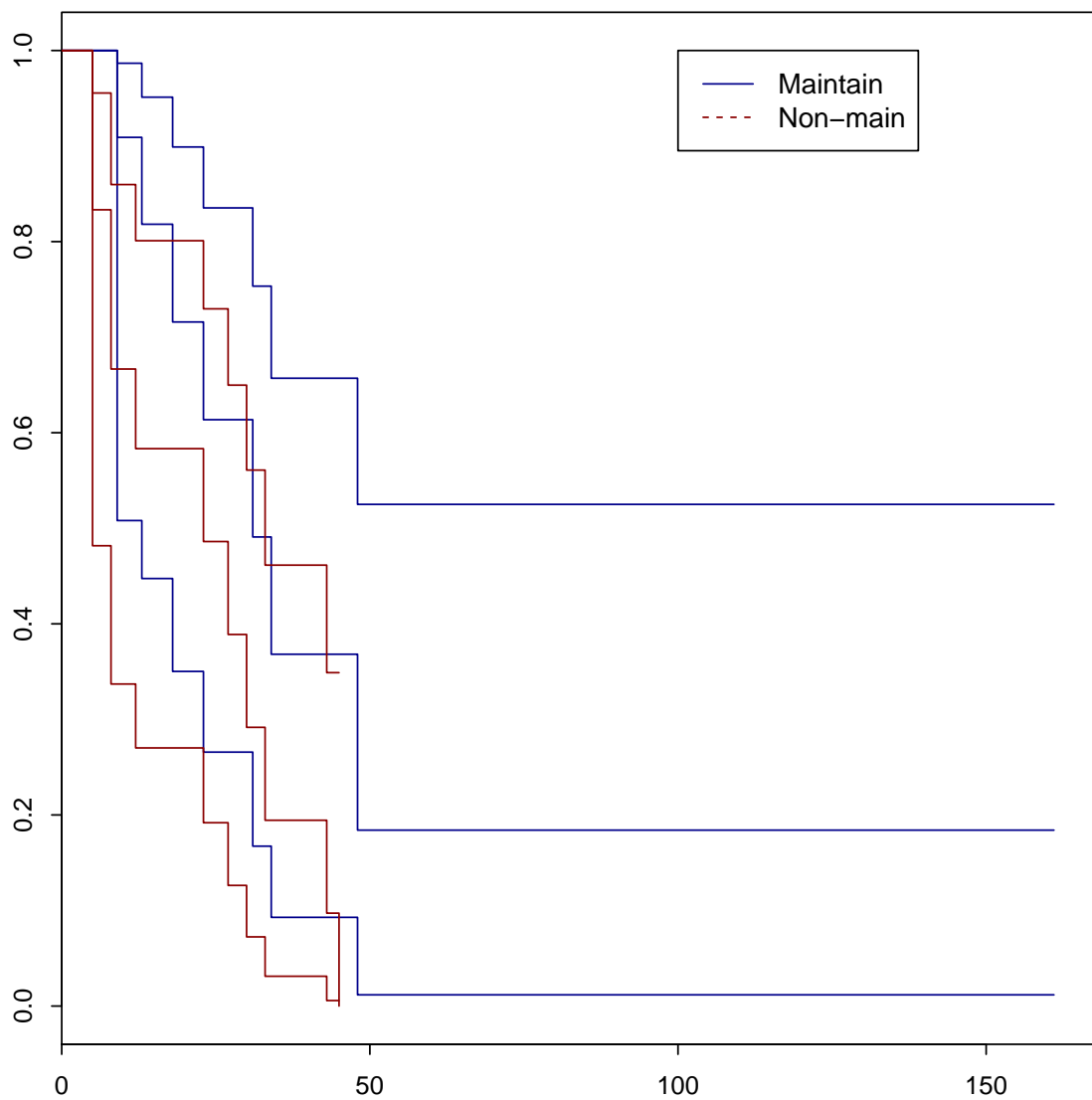
Compute confidence intervals and plot them

```
leuk.km2 <- survfit(Surv(time, status) ~ x, data = leukemia,
  conf.type = "log-log")
summary(leuk.km2)
```

```
## Call: survfit(formula = Surv(time, status) ~ x, data = leukemia, conf.type = 
## 
##               x=Maintained
##   time n.risk n.event survival std.err lower 95% CI upper 95% CI
##    9      11       1   0.909  0.0867   0.5081    0.987
##   13      10       1   0.818  0.1163   0.4474    0.951
##   18       8       1   0.716  0.1397   0.3502    0.899
##   23       7       1   0.614  0.1526   0.2658    0.835
##   31       5       1   0.491  0.1642   0.1673    0.753
##   34       4       1   0.368  0.1627   0.0928    0.657
##   48       2       1   0.184  0.1535   0.0117    0.525
```

```
##
##               x=Nonmaintained
##  time n.risk n.event survival std.err lower 95% CI upper 95% CI
##    5     12      2   0.8333  0.1076    0.48171    0.956
##    8     10      2   0.6667  0.1361    0.33702    0.860
##   12      8      1   0.5833  0.1423    0.27014    0.801
##   23      6      1   0.4861  0.1481    0.19188    0.730
##   27      5      1   0.3889  0.1470    0.12627    0.650
##   30      4      1   0.2917  0.1387    0.07240    0.561
##   33      3      1   0.1944  0.1219    0.03120    0.461
##   43      2      1   0.0972  0.0919    0.00575    0.349
##   45      1      1   0.0000      NaN           NA           NA
##
```

```
plot(leuk.km2, mark.time = FALSE, conf.int = TRUE, lty = 1,
     col = c("darkblue", "darkred"))
legend(100, 1, legend = c("Maintain", "Non-main"), lty = 1:2,
     col = c("darkblue", "darkred"))
```



Test for difference (log-rank test)

```
survdif(Surv(time, status) ~ x, data = leukemia)

## Call:
## survdif(formula = Surv(time, status) ~ x, data = leukemia)
##
##              N Observed Expected (O-E)^2/E (O-E)^2/V
## x=Maintained   11         7    10.69      1.27      3.4
## x=Nonmaintained 12        11     7.31      1.86      3.4
##
## Chisq= 3.4  on 1 degrees of freedom, p= 0.0653
```


7.9.3 Cox proportional hazards model

The (semi-parametric) Cox regression model refers to the method first proposed in 1972 by the British statistician Cox in his seminal paper “Regression Models and Life Tables”. It is difficult to exaggerate the impact of this paper. In the 1992 *Science Citation Index*, it was cited over 800 times, making it the most highly cited journal article in the entire literature of statistics. In fact, Garfield reported that its cumulative citation count placed it among the top 100 papers in all branches of science.

This enormous popularity can be explained by the fact that, unlike the parametric methods, Cox’s method does not require the selection of some particular probability distribution to represent survival times. For this reason, the method is called *semi-parametric*. Cox made two significant innovations. First, he proposed a model that is often referred to as the *proportional hazards model*. Second, he proposed a new estimation method that was later named *maximum partial likelihood*. The term *Cox regression* refers to the combination of the model and the estimation method

Here is an example of Cox regression estimating the benefit of maintaining chemotherapy of with respect to the survival of the patients.

```
leuk.ph <- coxph(Surv(time, status) ~ x, data = leukemia)
summary(leuk.ph)

## Call:
## coxph(formula = Surv(time, status) ~ x, data = leukemia)
##
##      n= 23, number of events= 18
##
##              coef exp(coef) se(coef)      z Pr(>|z|)
## xNonmaintained 0.916      2.498    0.512 1.79   0.074 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##              exp(coef) exp(-coef) lower .95 upper .95
## xNonmaintained      2.5         0.4    0.916    6.81
##
## Concordance= 0.619  (se = 0.073 )
## Rsquare= 0.137    (max possible= 0.976 )
## Likelihood ratio test= 3.38  on 1 df,   p=0.0658
## Wald test            = 3.2   on 1 df,   p=0.0737
## Score (logrank) test = 3.42  on 1 df,   p=0.0645
##
##
## plot(leuk.km2, mark.time=F, lty=1:2) lines(survfit(leuk.ph),
## lty=1:2, lwd=2)
```

It is not trivial to estimate the relevance of a variable with survival. If this variable is categorical, you can draw the survival curves and statistically compare them. If the variable under interest is continuous you can arbitrarily discretize it (not advisable) or use many existing performance criteria published so far for survival analysis: hazard ration (see *coxph*), D.index, concordance.index, time-

dependent ROC curve, Brier score,... The `survcomp` package contains functions to estimate these criteria.

7.10 Exercise 11: Survival Analysis

- Use the *colon* dataset from the library *survival*
- draw the Kaplan-Meier survival curves for the three group of patients encode by 'rx'.
- Use different colors for the curves and plot the lines twice as thick as the default size (parameter *lwd*).
- Which color encodes which group? Add a legend to the plot to make this clear.
- Generate a PDF output of the plot and put it in the website dropbox along with your code.
- Test if the different patient group have significantly different outcome?

Chapter 8

Solutions to Exercises

8.1 Solution to Exercise 1

Women Data

```
myURL <-  
"http://bcb.dfci.harvard.edu/~aedin/courses/Bioconductor/Women.txt"  
women <- read.table(myURL, sep = "\t", header = TRUE)  
`?`(colnames)  
women  
  
##      height weight age  
## 1      58    115  33  
## 2      59    117  34  
## 3      60    120  37  
## 4      61    123  31  
## 5      62    126  31  
## 6      63    129  34  
## 7      64    132  31  
## 8      65    135  39  
## 9      66    139  35  
## 10     67    142  34  
## 11     68    146  34  
## 12     69    150  36  
## 13     70    154  33  
## 14     71    159  30  
## 15     72    164  37  
  
class(women)  
  
## [1] "data.frame"  
  
str(women)
```

```
## data.frame: 15 obs. of 3 variables:
## $ height: int 58 59 60 61 62 63 64 65 66 67 ...
## $ weight: int 115 117 120 123 126 129 132 135 139 142 ...
## $ age : int 33 34 37 31 31 34 31 39 35 34 ...
```

```
nrow(women)
```

```
## [1] 15
```

```
ncol(women)
```

```
## [1] 3
```

```
dim(women)
```

```
## [1] 15 3
```

```
summary(women)
```

```
##      height      weight      age
##  Min.   :58.0   Min.   :115   Min.   :30.0
## 1st Qu.:61.5   1st Qu.:124   1st Qu.:32.0
##  Median :65.0   Median :135   Median :34.0
##  Mean   :65.0   Mean   :137   Mean   :33.9
## 3rd Qu.:68.5   3rd Qu.:148   3rd Qu.:35.5
##  Max.   :72.0   Max.   :164   Max.   :39.0
```

```
colMeans(women)
```

```
## height weight  age
## 65.00 136.73 33.93
```

```
colnames(women)
```

```
## [1] "height" "weight" "age"
```

```
`?`(colnames)
```

```
sum(women$weight < 120)
```

```
## [1] 2
```

```
women[order(women$weight), ]
```

```
##      height weight age
## 1      58     115  33
## 2      59     117  34
## 3      60     120  37
## 4      61     123  31
## 5      62     126  31
## 6      63     129  34
## 7      64     132  31
## 8      65     135  39
## 9      66     139  35
## 10     67     142  34
## 11     68     146  34
## 12     69     150  36
## 13     70     154  33
## 14     71     159  30
## 15     72     164  37

mean(women$height[women$weight > 124 & women$weight < 150])

## [1] 65

rownames(women)[5] <- "Lucy"
```

8.2 Solution to Exercise 2

ToothGrowth data

```
TG <- read.table("ToothGrowth.txt", sep = "\t", header = TRUE)
TG2 <- read.csv("ToothGrowth.csv")
nrow(TG)

## [1] 60

nrow(TG2)

## [1] 60

mean(TG$len)

## [1] 18.81

sd(TG$len)

## [1] 7.649
```

```

mean(TG2$len)

## [1] 18.81

sd(TG2$len)

## [1] 7.649

anova(lm(len ~ supp + dose, data = TG))

## Analysis of Variance Table
##
## Response: len
##          Df Sum Sq Mean Sq F value    Pr(>F)
## supp      1    205      205    11.4 0.0013 **
## dose      1   2224     2224   124.0 6.3e-16 ***
## Residuals 57   1023        18
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

8.3 Solution to Exercise 3

```

women <-
read.table("http://bcb.dfci.harvard.edu/~aedin/courses/R/WomenStats.txt",
  sep = "\t", header = TRUE)
nrow(women)

## [1] 16

ncol(women)

## [1] 1

colnames(women)

## [1] "X.html."

summary(women)

##          X.html.
## </div>          :2
## </body>         :1

```

```
## </head> :1
## </html> :1
## </td></tr></table> :1
## <body style=background-color:#fff>:1
## (Other) :9

rownames(women) <- LETTERS[1:nrow(women)]
write.table(women, "modifiedWomen.txt", sep = "\t")
women2 <- read.table("modifiedWomen.txt", sep = "\t", as.is = TRUE,
  header = TRUE)
```

8.4 Solution to Exercise 4

```
myVec <- c(LETTERS[1:20], seq(0, 200, 10))
myVec

## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
## [11] "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T"
## [21] "0" "10" "20" "30" "40" "50" "60" "70" "80" "90"
## [31] "100" "110" "120" "130" "140" "150" "160" "170" "180" "190"
## [41] "200"

myVec <- sample(myVec)
myVec

## [1] "T" "70" "80" "200" "160" "130" "Q" "M" "100" "120"
## [11] "180" "30" "S" "110" "50" "N" "I" "190" "C" "10"
## [21] "A" "F" "140" "P" "0" "20" "B" "G" "150" "J"
## [31] "40" "L" "60" "90" "H" "E" "170" "R" "K" "D"
## [41] "O"

cat(myVec, file = "myVec.txt")
scan("myVec.txt", n = 10, what = "text")

## [1] "T" "70" "80" "200" "160" "130" "Q" "M" "100" "120"

scan("myVec.txt", n = 10, what = 123)

## Error: scan() expected a real, got T

scan("myVec.txt", n = 10, what = TRUE)

## Error: scan() expected a logical, got 70
```


8.5 Solution to Exercise 5

```
for (i in 1:10) print(2^i)
```

```
## [1] 2
## [1] 4
## [1] 8
## [1] 16
## [1] 32
## [1] 64
## [1] 128
## [1] 256
## [1] 512
## [1] 1024
```

```
x <- 1
while (2^x < 1000) {
  print(2^x)
  x <- x + 1
}
```

```
## [1] 2
## [1] 4
## [1] 8
## [1] 16
## [1] 32
## [1] 64
## [1] 128
## [1] 256
## [1] 512
```

8.6 Solution to Exercise 6

```
require(XML)
## Reads all the tables in the webpage into a list
worldPop <-
readHTMLTable("http://en.wikipedia.org/wiki/World_population")

# There are 19 tables
class(worldPop)

## [1] "list"
```

```
length(worldPop)

## [1] 19

# With the following names, so of the names are very long so we
# have truncated them using substr
substr(names(worldPop), 1, 60)

## [1] "NULL"
## [2] "toc"
## [3] "NULL"
## [4] "World population milestones (USCB estimates)"
## [5] "The 10 countries with the largest total population:"
## [6] "10 most densely populated countries (with population above 1"
## [7] "Countries ranking highly in terms of both total population ("
## [8] "NULL"
## [9] "UN (medium variant 2010 revision) and US Census Bureau (De"
## [10] "UN 2008 estimates and medium variant projections (in million"
## [11] "World historical and predicted populations (in millions)[102"
## [12] "World historical and predicted populations by percentage dis"
## [13] "Estimated world population at various dates (in millions)"
## [14] "Starting at 500 million"
## [15] "Starting at 375 million"
## [16] "NULL"
## [17] "NULL"
## [18] "NULL"
## [19] "NULL"

worldPop <- worldPop[[13]] # Just look at Table 13
```

To tidy up this tables, lets look at dates after 1750AD, so lets remove rows 1 to 14 as these have only world population information. Also we will remove row 32 which is just column names

```
## Now lets check the structure of the table The data are factors,
## lets convert to characters as these are easier to edit
str(worldPop)

## data.frame: 32 obs. of 9 variables:
## $ Year : Factor w/ 32 levels "10,000 BC","1000",...: 27 1 30
## $ World : Factor w/ 32 levels "< 0.015","1",...: 1 2 14 22 29
## $ Africa : Factor w/ 19 levels "", "1,022", "106",...: 1 1 1 1 1
## $ Asia : Factor w/ 19 levels "", "1,398", "1,542",...: 1 1 1 1 1
## $ Europe : Factor w/ 19 levels "", "163", "203",...: 1 1 1 1 1
## $ Latin America[Note 1] : Factor w/ 19 levels "", "16", "167",...: 1 1 1 1 1
## $ Northern America : Factor w/ 19 levels "", "172", "187",...: 1 1 1 1 1
```

```

## $ Oceania          : Factor w/ 17 levels "", "12.8", "14.3", ...: 1 1 1 1 1
## $ Notes            : Factor w/ 6 levels "", "[104]", "[105]", ...: 2 1 1 3

## We need to convert them to numeric. But first lets get rid of
## the BC and $<$ characters

worldPop <- apply(worldPop, 2, as.character)
str(worldPop)

## chr [1:32, 1:9] "70,000 BC" "10,000 BC" "9000 BC" ...
## - attr(*, "dimnames")=List of 2
## ..$ : NULL
## ..$ : chr [1:9] "Year" "World" "Africa" "Asia" ...

## Remove rows 1-14, 32
worldPop <- worldPop[-c(1:14, 32), ]

## Remove 'Notes' Column
worldPop <- worldPop[, -9]

## Lets convert to numeric and remove the period
worldPop <- apply(worldPop, 2, function(x) as.numeric(sub("",
  "", x)))
worldPop <- as.data.frame(worldPop)

## Lets get rid of 'Note 1' To get rid of the square brakets we
## have to put \\ before them
colnames(worldPop) <- sub("\\[Note 1\\]", "", colnames(worldPop))
str(worldPop)

## data.frame: 17 obs. of 8 variables:
## $ Year          : num 1750 1800 1850 1900 1950 ...
## $ World         : num 791 978 1262 1650 2519 ...
## $ Africa        : num 106 107 111 133 221 247 277 314 357 408 ...
## $ Asia          : num 502 635 809 947 1398 ...
## $ Europe        : num 163 203 276 408 547 575 601 634 656 675 ...
## $ Latin America : num 16 24 38 74 167 191 209 250 285 322 ...
## $ Northern America: num 2 7 26 82 172 187 204 219 232 243 ...
## $ Oceania       : num 2 2 2 6 12.8 14.3 15.9 17.6 19.4 21.5 ...

# In what year did the population of Europe, Africa and Asia exceed
# 500 million?
for (i in c("Europe", "Asia", "Africa")) {

```

```

    print(paste(i, min(worldPop$Year[worldPop[, i] > 500]), sep = "
"))
}

## [1] "Europe 1950"
## [1] "Asia 1750"
## [1] "Africa 1985"

```

Now that the data are tidy, lets create the bonus plot

```

## Make a vector regions (exclude World and Year)
regions <- colnames(worldPop)[-c(1:2)]
regions

## [1] "Africa"          "Asia"            "Europe"
## [4] "Latin America"   "Northern America" "Oceania"

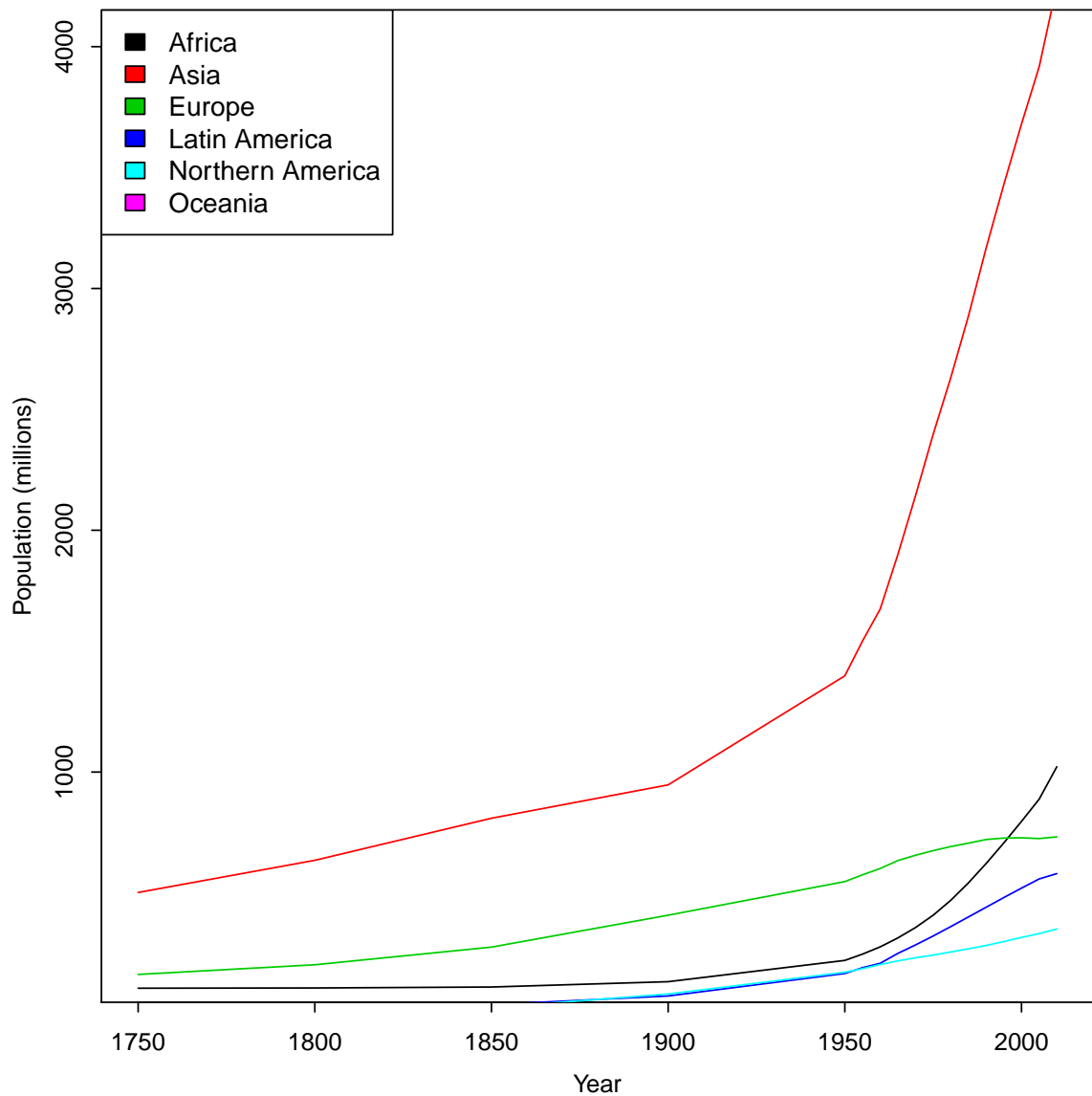
## Create a empty Plot
plot(worldPop$Year, worldPop$Asia, xlab = "Year", ylab =
"Population (millions)",
      col = "red", type = "n", ylim = c(200, 4000))

for (i in 1:length(regions)) {
  region <- regions[i]
  print(region)
  lines(worldPop$Year, worldPop[, region], col = i, type = "l")
}

## [1] "Africa"
## [1] "Asia"
## [1] "Europe"
## [1] "Latin America"
## [1] "Northern America"
## [1] "Oceania"

legend("topleft", regions, fill = 1:length(regions))

```



8.7 Solution to Exercise 7

```
TG <- read.table("ToothGrowth.txt", sep = "\t", header = TRUE)
summary(TG)
```

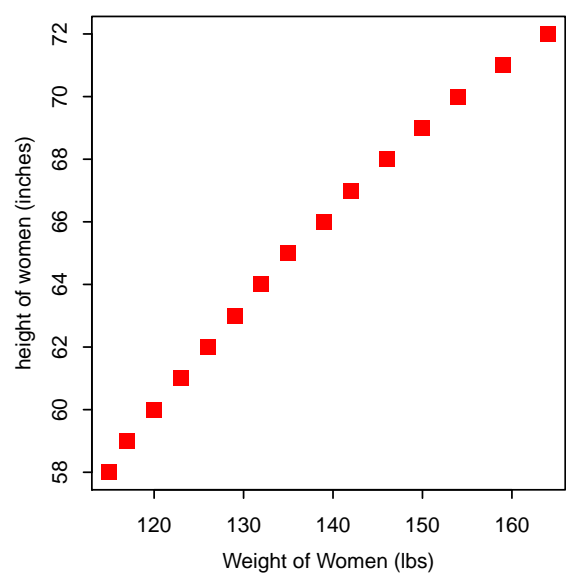
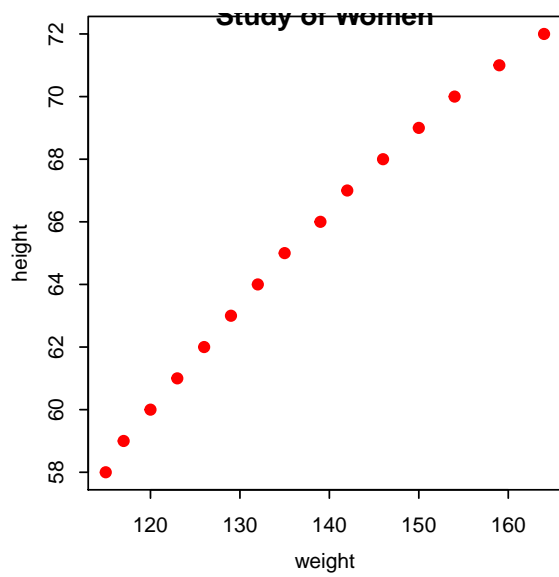
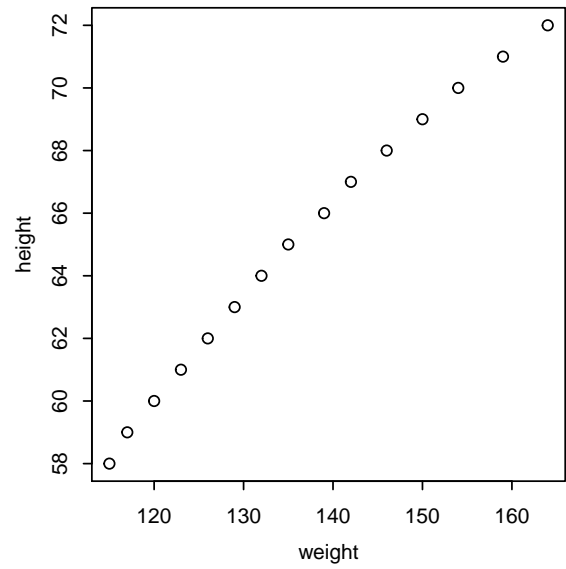
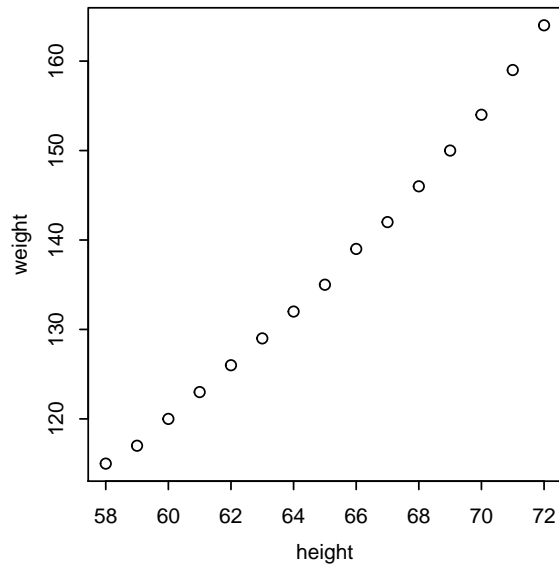
```
##      len      supp      dose
##  Min.   : 4.2    OJ:30    Min.    :0.50
##  1st Qu.:13.1    VC:30    1st Qu.:0.50
##  Median :19.2                Median :1.00
##  Mean   :18.8                Mean   :1.17
##  3rd Qu.:25.3                3rd Qu.:2.00
```

```
## Max.      :33.9          Max.      :2.00

par(mfrow = c(1, 2))
boxplot(TG$len ~ TG$supp, col = 2:4, las = 2, xlab = "Treatment",
        ylab = "Tooth Length")
boxplot(TG$len ~ paste(TG$supp, TG$dose), col = rep(2:3, each = 3),
        las = 2, xlab = "Treatment and Dose", ylab = "Tooth Length")
```

8.8 Solution to Exercise 8

```
par(mfrow = c(2, 2))
data(women)
attach(women)
plot(height, weight)
plot(weight, height)
plot(weight, height, pch = 19, col = "red", main = "Study of
Women")
plot(weight, height, xlab = "Weight of Women (lbs)", ylab = "height
of women (inches)",
     pch = 15, col = "red", cex = 1.5)
```



8.9 Solution to Exercise 9

```
mod1 <- lm(Hwt ~ Sex, data = cats)
model.matrix(mod1)[1:10, ]

##      (Intercept) SexM
## 1             1     0
## 2             1     0
## 3             1     0
## 4             1     0
## 5             1     0
## 6             1     0
## 7             1     0
## 8             1     0
## 9             1     0
## 10            1     0

mod1 <- lm(Hwt ~ Sex - 1, data = cats)
model.matrix(mod1)[1:10, ]

##      SexF SexM
## 1         1     0
## 2         1     0
## 3         1     0
## 4         1     0
## 5         1     0
## 6         1     0
## 7         1     0
## 8         1     0
## 9         1     0
## 10        1     0
```

8.10 Solution to Exercise 10

```
data.lungs <- read.csv("lungs.csv", stringsAsFactors = FALSE)
head(data.lungs)

##   age sex height weight bmp fev1  rv frc tlc pemax
## 1   7   0   109   13.1  68   32 258 183 137    95
## 2   7   1   112   12.9  65   19 449 245 134    85
## 3   8   0   124   14.1  64   22 441 268 147   100
## 4   8   1   125   16.2  67   41 234 146 124    85
## 5   8   0   127   21.5  93   52 202 131 104    95
## 6   9   0   130   17.5  68   44 308 155 118    80
```



```
lungFit <- lm(pemax ~ ., data = data.lungs)
summary(lungFit)

##
## Call:
## lm(formula = pemax ~ ., data = data.lungs)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -37.34 -11.53   1.08  13.39  33.41
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  176.058    225.891   0.78    0.45
## age          -2.542     4.802   -0.53    0.60
## sex          -3.737    15.460   -0.24    0.81
## height      -0.446     0.903   -0.49    0.63
## weight       2.993     2.008    1.49    0.16
## bmp         -1.745     1.155   -1.51    0.15
## fev1         1.081     1.081    1.00    0.33
## rv           0.197     0.196    1.00    0.33
## frc         -0.308     0.492   -0.63    0.54
## tlc          0.189     0.500    0.38    0.71
##
## Residual standard error: 25.5 on 15 degrees of freedom
## Multiple R-squared:  0.637, Adjusted R-squared:  0.42
## F-statistic: 2.93 on 9 and 15 DF,  p-value: 0.032
##

resid(lungFit)

##      1      2      3      4      5      6      7      8
## 10.031 -3.414 13.386 -11.532 18.691 -31.552 -11.480 20.034
##      9     10     11     12     13     14     15     16
## -20.307 -13.182 15.646 10.748 -3.664 -33.118 10.460 33.405
##     17     18     19     20     21     22     23     24
## 21.034 -3.002 12.096  1.081 -37.338 11.864 -4.332 -34.233
##     25
## 28.677

## most significant
sort(summary(lungFit)$coefficients[, 4], decreasing = FALSE)[1]

##      bmp
## 0.1517
```

```
## least significant
sort(summary(lungFit)$coefficients[, 4], decreasing = TRUE)[1]

##      sex
## 0.8123
```

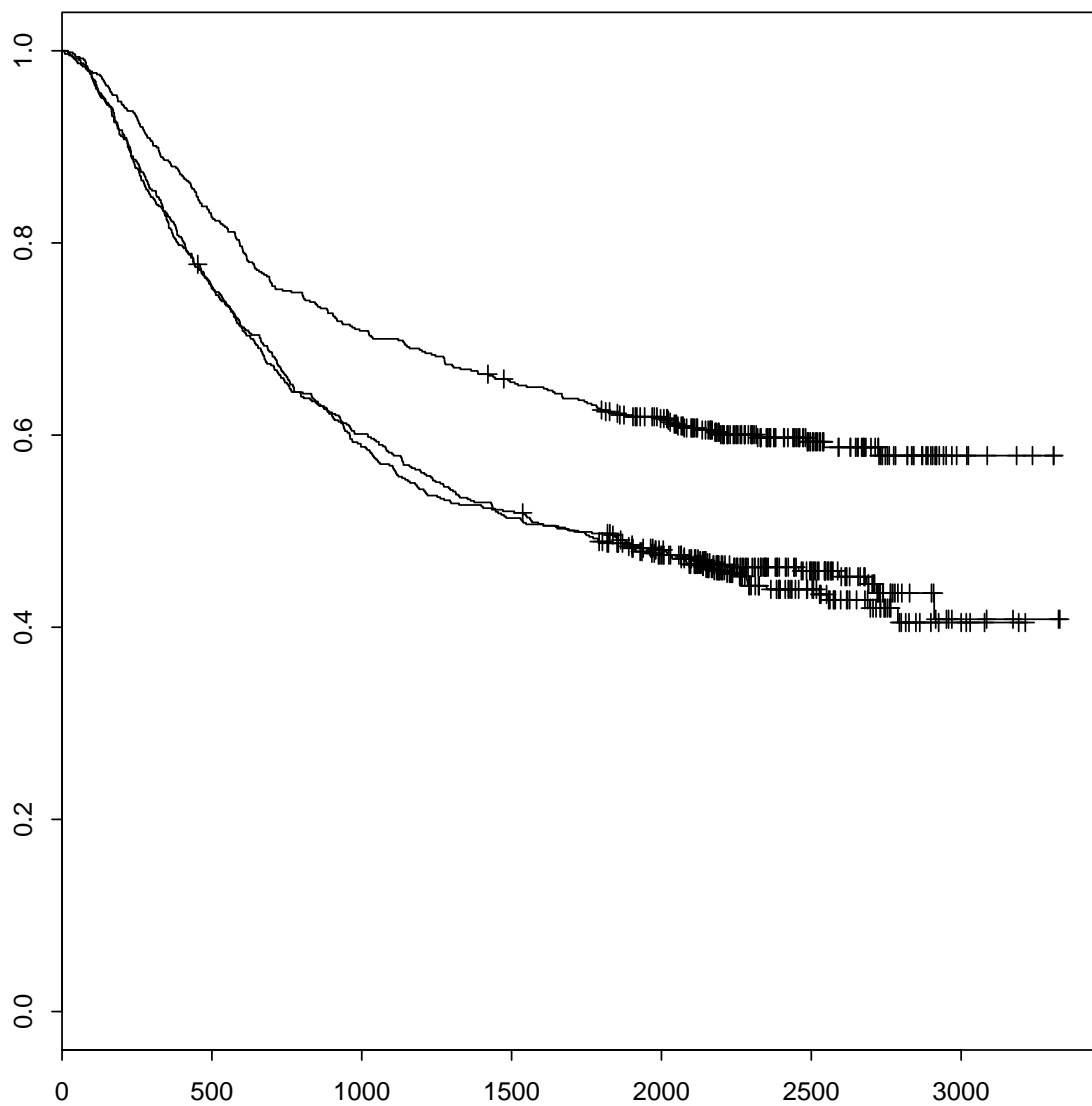
8.11 Solution to Exercise 11

```
library(survival)
head(colon)
```

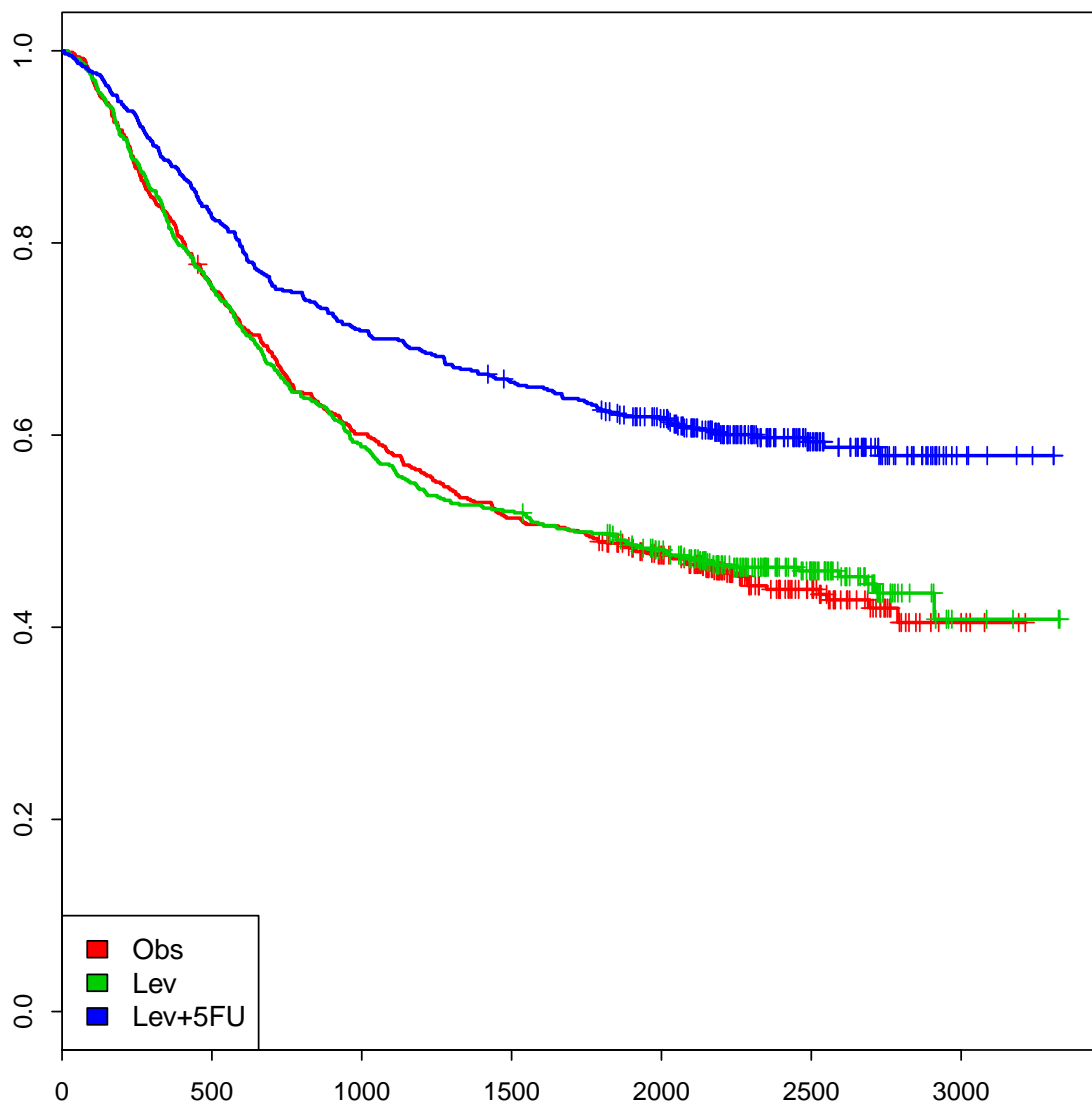
##	id	study	rx	sex	age	obstruct	perfor	adhere	nodes	status
## 1	1	1	Lev+5FU	1	43	0	0	0	5	1
## 2	1	1	Lev+5FU	1	43	0	0	0	5	1
## 3	2	1	Lev+5FU	1	63	0	0	0	1	0
## 4	2	1	Lev+5FU	1	63	0	0	0	1	0
## 5	3	1	Obs	0	71	0	0	1	7	1
## 6	3	1	Obs	0	71	0	0	1	7	1

##	differ	extent	surg	node4	time	etype
## 1	2	3	0	1	1521	2
## 2	2	3	0	1	968	1
## 3	2	3	0	0	3087	2
## 4	2	3	0	0	3087	1
## 5	2	2	0	1	963	2
## 6	2	2	0	1	542	1


```
colonFit <- survfit(Surv(time, status) ~ rx, data = colon)
plot(colonFit)
```



```
plot(colonFit, col = 2:4, lwd = 2)
plot(colonFit, col = 2:4, lwd = 2)
legend("bottomleft", legend = levels(colon$rx), fill = 2:4)
```



```
pdf(file = "Surv.pdf")
plot(colonFit, col = 2:4, lwd = 2)
legend("bottomleft", legend = levels(colon$rx), fill = 2:4)
dev.off()
```

```
## pdf
## 2
```

```
survdif(Surv(time, status) ~ rx, data = colon)
```

```
## Call:
```

```
## survdiff(formula = Surv(time, status) ~ rx, data = colon)
##
##              N Observed Expected (O-E)^2/E (O-E)^2/V
## rx=Obs       630      345      299      7.01      10.40
## rx=Lev       620      333      295      4.93       7.26
## rx=Lev+5FU  608      242      326     21.61     33.54
##
##  Chisq= 33.6  on 2 degrees of freedom, p= 4.99e-08

cp <- coxph(Surv(time, status) ~ rx, data = colon)
summary(cp)

## Call:
## coxph(formula = Surv(time, status) ~ rx, data = colon)
##
##    n= 1858, number of events= 920
##
##              coef exp(coef) se(coef)      z Pr(>|z|)
## rxLev       -0.0209   0.9793  0.0768 -0.27   0.79
## rxLev+5FU -0.4410   0.6434  0.0839 -5.26 1.5e-07 ***
## ---
## Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1 1
##
##              exp(coef) exp(-coef) lower .95 upper .95
## rxLev           0.979      1.02    0.842    1.138
## rxLev+5FU       0.643      1.55    0.546    0.758
##
## Concordance= 0.545 (se = 0.009 )
## Rsquare= 0.019 (max possible= 0.999 )
## Likelihood ratio test= 35.2 on 2 df, p=2.23e-08
## Wald test = 33.1 on 2 df, p=6.45e-08
## Score (logrank) test = 33.6 on 2 df, p=4.99e-08
##
```

8.12 sessionInfo()

sessionInfo()

```
## R version 2.15.0 (2012-03-30)
## Platform: x86_64-pc-mingw32/x64 (64-bit)
##
## locale:
## [1] LC_COLLATE=English_United States.1252
## [2] LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
## [1] splines      grid        stats      graphics  grDevices  utils
## [7] datasets    methods    base
##
## other attached packages:
## [1] survival_2.36-14      vcd_1.2-13           colorspace_1.1-1
## [4] annotate_1.34.0       AnnotationDbi_1.18.1 Biobase_2.16.0
## [7] BiocGenerics_0.2.0    wordcloud_2.0         Rcpp_0.9.10
## [10] tm_0.5-7.1            igraph_0.5.5-4       network_1.7-1
## [13] googleVis_0.2.16     RJSONIO_0.98-1       lattice_0.20-6
## [16] ggplot2_0.9.1         RColorBrewer_1.0-5   venneuler_1.1-0
## [19] gplots_2.11.0         MASS_7.3-17          KernSmooth_2.23-7
## [22] caTools_1.12          bitops_1.0-4.1       gdata_2.10.6
## [25] gtools_2.6.2          scatterplot3d_0.3-33 XML_3.9-4.1
## [28] R2HTML_2.2            SAScii_0.2           stringr_0.6
## [31] XLConnect_0.1-9       XLConnectJars_0.1-4  rJava_0.9-3
## [34] knitr_0.6
##
## loaded via a namespace (and not attached):
## [1] codetools_0.2-8 DBI_0.2-5           dichromat_1.2-4 digest_0.5.2
## [5] evaluate_0.4.2  formatR_0.4         highlight_0.3.1 IRanges_1.14.3
## [9] labeling_0.1    memoise_0.1         munsell_0.3      parser_0.0-14
## [13] plyr_1.7.1      proto_0.3-9.2       reshape2_1.2.1  RSQLite_0.11.1
## [17] scales_0.2.1    slam_0.1-24         stats4_2.15.0   tools_2.15.0
## [21] xtable_1.7-0
```