

Lecture Notes for STAC32

Ken Butler

September 10, 2013

Contents

1	Course Outline	7
1.1	What the course will be about	7
1.2	The Instructor	7
1.3	Course texts	8
1.4	Structure of course	8
1.5	Course material	8
1.6	Assessment	9
1.7	Missed work and documentation	9
1.8	Academic integrity	10
1.9	Accessibility statement	11
2	Hello, world	13
3	Installation and connection	21
3.1	Some history	21
3.2	Installing R	22
3.3	Connecting to SAS	22
3.3.1	Introduction	22
3.3.2	Setup	23
4	Using R and SAS	27
4.1	Using SAS	27
4.1.1	Starting SAS	27
4.1.2	Using SAS	27
4.1.3	Saving and opening	36
4.1.4	Copying and pasting	40
4.1.5	Batch mode	41
4.1.6	Setting up SAS to get HTML output	42
4.2	Using R	43
4.2.1	Introduction	43
4.2.2	Projects and R scripts	45
4.2.3	Reading data from a file	48
4.2.4	Means by group	50
4.2.5	Boxplot	51

4.2.6	Reading data from a spreadsheet	54
5	A data analysis example	57
6	Statistical Inference	69
6.1	Introduction	69
6.2	Example: learning to read	69
6.3	Randomization test	76
6.4	Jargon for testing	80
6.5	Power	81
6.6	Confidence intervals	91
6.7	The duality between confidence intervals and hypothesis tests . .	94
6.8	Sign test	100
6.9	Matched pairs	106
6.10	Comparing more than two groups	121
6.10.1	Tukey's method in ANOVA	129
7	Writing reports	135
7.1	Introduction	135
7.2	Writing a report	136
7.3	Example report	139
7.4	This Thing Called Science	141
7.5	Reproducible Research	141
7.5.1	Introduction	141
7.5.2	R Markdown	142
7.5.3	Pandoc and making Word docs	144
7.5.4	Making a Slidy slideshow using <code>pandoc</code>	148
7.6	Sweave and Statweave	151
8	More examples	153
8.1	The windmill data	153
8.2	Another regression example in SAS	173
8.3	The asphalt data	182
9	Intricacies of R	211
9.1	Introduction	211
9.2	Help!	212
9.2.1	Making a scatterplot	215
9.2.2	Making a time (index) plot	216
9.2.3	Making a bar chart	217
9.2.4	Making side-by-side boxplots	219
9.2.5	Plotting two factors against each other	223
9.2.6	Making a pairs plot	225
9.2.7	Different plot methods	228
9.2.8	Finding function names	232
9.3	Making and annotating plots	234

9.3.1	Introduction	234
9.3.2	Adding a title to a plot	234
9.3.3	Axis labels	235
9.3.4	Adding points to a plot	237
9.3.5	Adding a regression line to a plot	240
9.3.6	Horizontal and vertical lines	241
9.3.7	Adding lines with given intercept and slope	243
9.3.8	Different line types	244
9.3.9	Plotting curves	246
9.3.10	Extending (or shrinking) axes	247
9.3.11	Labelling points	251
9.3.12	Colours	254
9.3.13	Legends	260
9.3.14	Plotting symbols	261
9.3.15	More about legends	264
9.3.16	Changing text size	267
9.3.17	Variations on colours	270
9.3.18	Appearance of numbers on axes	272
9.4	Other types of plots	274
9.4.1	Time series plots	274
9.4.2	Plotting multiple series on one plot	277
9.4.3	Multiple plots on one page	281
9.5	Histograms and normal quantile plots	284
9.5.1	Introduction	284
9.5.2	Changing bar width	285
9.5.3	Kernel density curves	287
9.5.4	Assessing normality	290
9.6	Selecting and combining data	297
9.6.1	Selecting rows and columns of a data frame	297
9.6.2	Selecting based on conditions	299
9.7	The apply family	303
9.7.1	Introduction	303
9.7.2	<code>apply</code>	304
9.7.3	<code>tapply</code> and <code>aggregate</code>	305
9.7.4	<code>sapply</code> and <code>split</code>	306
9.8	Vectors and matrices in R	308
9.8.1	Vector addition	308
9.8.2	Scalar multiplication	309
9.8.3	“Vector multiplication”	309
9.8.4	Combining different-length vectors	310
9.8.5	Making matrices	310
9.8.6	Adding matrices	311
9.8.7	Multiplying matrices	311
9.8.8	Linear algebra stuff in R	312
9.9	Writing functions in R	313
9.9.1	Introduction	313

9.9.2	The anatomy of a function	313
9.9.3	Writing a function	317
9.9.4	One-liners and curly brackets	318
9.9.5	Using a function you wrote	318
9.9.6	Optional arguments	319
9.9.7	Geometric distribution	321
9.9.8	Functions calling other functions	323
9.9.9	Passing on arguments	325
10	Intricacies of SAS	331
10.1	Introduction	331
10.2	Reading stuff from a file	331
10.2.1	The basics	331
10.2.2	More than one observation on each line	332
10.2.3	Skipping over header lines	333
10.2.4	Delimited data	334
10.2.5	Reading text	335
10.2.6	Windows and Linux/Unix file formats	337
10.2.7	Another way to read spreadsheet data	338
10.2.8	Yet another way to read in a spreadsheet	338
10.3	Dates	339
10.4	Creating permanent data sets and referring to data sets	341
10.5	How SAS knows which data set to use	343
10.6	Creating new data sets from old ones	344
10.6.1	Selecting individuals	344
10.6.2	Selecting variables	346
10.6.3	New data sets from old ones	347
10.7	Plots	351
10.7.1	Histograms, normal quantile plots and boxplots	351
10.7.2	Scatterplots	365
11	Data munging	389
11.1	Introduction	389
11.2	Handling data	389
11.3	Checking data	390
11.3.1	Introduction	390
11.3.2	Direct checking	390
11.3.3	Looking for outliers	392
11.3.4	Missing values	404
11.4	Re-arranging data	410
11.4.1	Keeping, dropping and creating variables	410
11.4.2	Dropping observations	414
11.4.3	Reshaping data	417
11.4.4	Matched pairs and reshape in reverse	423

Chapter 1

Course Outline

1.1 What the course will be about

In this course, we learn something about data analysis: the process of using statistical methods (most of which you will know) to gain insight about data. We will be learning two statistical packages to help us in this, SAS and R. I will try to use lots of examples to demonstrate what I mean, and I want you to “get your hands dirty” with some analyses of your own.

After a data analysis has been completed, the important thing is to convey the results to the world, by writing a report. We will look at how this ought to be done, and you will get some practice at doing this yourself.

Time permitting, we will look into how you do certain useful statistical tasks in R and SAS, so that you can take these forward into future courses.

1.2 The Instructor

- Ken Butler (butler@utsc.utoronto.ca), office IC 471.
- Lectures Tuesday 13:00-14:00 in IC200, Thursday 12:00-14:00 in IC 208 (first hour) and IC 306 (second hour). The second hour of the Thursday class is in a computer lab.
- Office hours: Tuesday 14:00-15:00, Thursday 14:00-16:00 or longer if needed. Or by appointment (e-mail me, address above, to set one up).
- Course website:

<http://www.utsc.utoronto.ca/%7ebutler/c32/>

- E-mail: Use only your UTSC/UToronto e-mail address. I will aim to respond within two working days. If you have something other than a simple question, you should bring it to office hours.

1.3 Course texts

Recommended, both:

- “SAS Essentials: Mastering SAS for research” by Alan C. Elliott and Wayne A. Woodward, publ. Jossey-Bass. ISBN 978-0-470-46129-7.
- “Using statistical methods in R: a how-to guide” by Ken Butler. Free for the download at <http://www.utsc.utoronto.ca/%7ebutler/r/book.pdf>. Chapters 1–7 and possibly 19.

1.4 Structure of course

We have three hours per week of class time. I plan to have some (an hour) of this time in a computer lab, so that you can practice what you have learned from the lectures and get help if needed. I will give you lab exercises to work on.

1.5 Course material

Some or all of this, possibly not in this order:

- Demo of what R and SAS can do
- Installation and connection
- Using SAS
- Using R
- Basics of data analysis, with examples
- Statistical inference
- Writing reports and reproducible research
- More data analysis (including regression and multiple regression)
- A look at the intricacies of R
- A look at the intricacies of SAS

We might not get to the last two, but we'll probably need to refer to some bits of them.

1.6 Assessment

Your grade in this course will be determined by four things:

- Assignments, to be given out every week or two. You may work on the assignments with your friends, but what you hand in must be **entirely your own work**.
- Midterm exam (2 hours), open book (see below).
- A data analysis report. This is to be a data analysis carried out on a data set of your choosing, using R or SAS or both, with a report written in the manner described in class.
- Final exam (3 hours), open book (see below).

For the exams, you may bring:

- the SAS text
- the R text
- my lecture notes
- any other notes that you have made in this course.

The final grade will be calculated as below:

Item	Percent
Assignments	25
Midterm exam	25
Data analysis report	15
Final exam	35

1.7 Missed work and documentation

There are **no** make-up assignments or exams in this course. Work handed in late may or may not be accepted, at the instructor's discretion. (You improve your chances of having late work accepted by telling the instructor ahead of time that the work will be late and why.)

If you miss assessed work due to illness or injury, you and your medical professional must complete the form at

<http://www.illnessverification.utoronto.ca/>

and submit it to the instructor as soon as possible. The weight of the missed work will be transferred to other assessments of the same type (assignments/exams).

Any other missed work counts *zero*.

1.8 Academic integrity

See <http://www.utoronto.ca/academicintegrity/>.

Academic integrity is essential to the pursuit of learning and scholarship in a university, and to ensuring that a degree from the University of Toronto is a strong signal of each student's individual academic achievement. As a result, the University treats cases of cheating and plagiarism very seriously. The University of Toronto's Code of Behaviour on Academic Matters:

<http://www.governingcouncil.utoronto.ca/policies/behaveac.htm>

outlines the behaviours that constitute academic dishonesty and the processes for addressing academic offences. Potential offences include, but are not limited to:

- In papers and assignments:
 - Using someone else's ideas or words without appropriate acknowledgement.
 - Submitting your own work in more than one course without the permission of the instructor.
 - Making up sources or facts.
 - Obtaining or providing unauthorized assistance on any assignment.
- On tests and exams:
 - Using or possessing unauthorized aids.
 - Looking at someone else's answers during an exam or test.
 - Misrepresenting your identity.
- In academic work:
 - Falsifying institutional documents or grades.
 - Falsifying or altering any documentation required by the University, including (but not limited to) doctor's notes.

All suspected cases of academic dishonesty will be investigated following procedures outlined in the Code of Behaviour on Academic Matters. If you have questions or concerns about what constitutes appropriate academic behaviour

or appropriate research and citation methods, you are expected to seek out additional information on academic integrity from your instructor or from other institutional resources.

1.9 Accessibility statement

Students with diverse learning styles and needs are welcome in this course. In particular, if you have a disability/health consideration that may require accommodations, please feel free to approach me and/or the AccessAbility Services Office as soon as possible. I will work with you and AccessAbility Services to ensure you can achieve your learning goals in this course. Enquiries are confidential. The UTSC AccessAbility Services staff (located in S302) are available by appointment to assess specific needs, provide referrals and arrange appropriate accommodations: (416) 287-7560 or ability@utsc.utoronto.ca.

Chapter 2

Hello, world

It's customary for a computing course to begin with a “hello, world” program: one that prints those words to the screen in the language of choice. This is about the simplest programming task imaginable.

To begin our course, I want to demonstrate the statistical version of this. I see this as “read in some data, calculate a few things, draw a picture”. I want you to focus on the output, and the strategy by which we get that output. The tactics, that is to say the details, can come later.

We have, in each case, one variable, whose values we read in. We calculate some summary statistics for that variable, and draw a histogram and boxplot.

Let's start with R:

```
x=c(10,11,13,14,14,15,15,17,20,25,29)
mean(x)

[1] 16.63636

sd(x)

[1] 5.852738

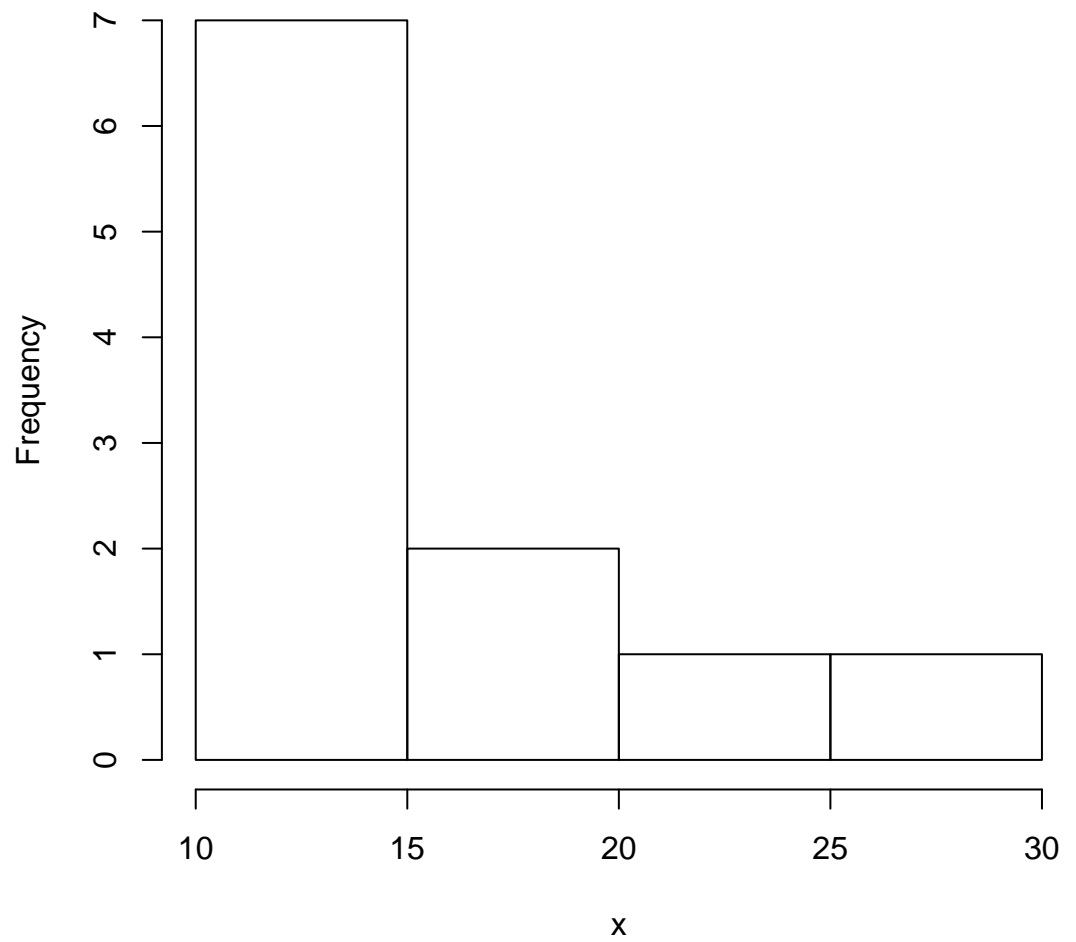
quantile(x)

 0%  25%  50%  75% 100%
10.0 13.5 15.0 18.5 29.0

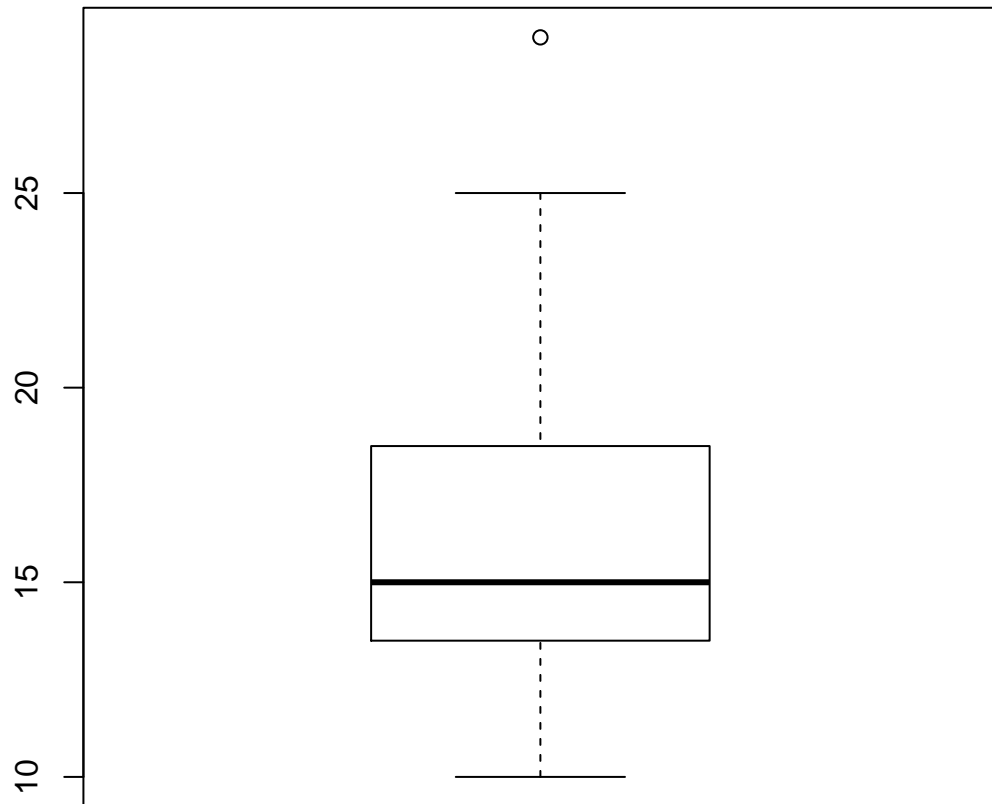
IQR(x)

[1] 5

hist(x)
```

Histogram of x

```
boxplot(x)
```

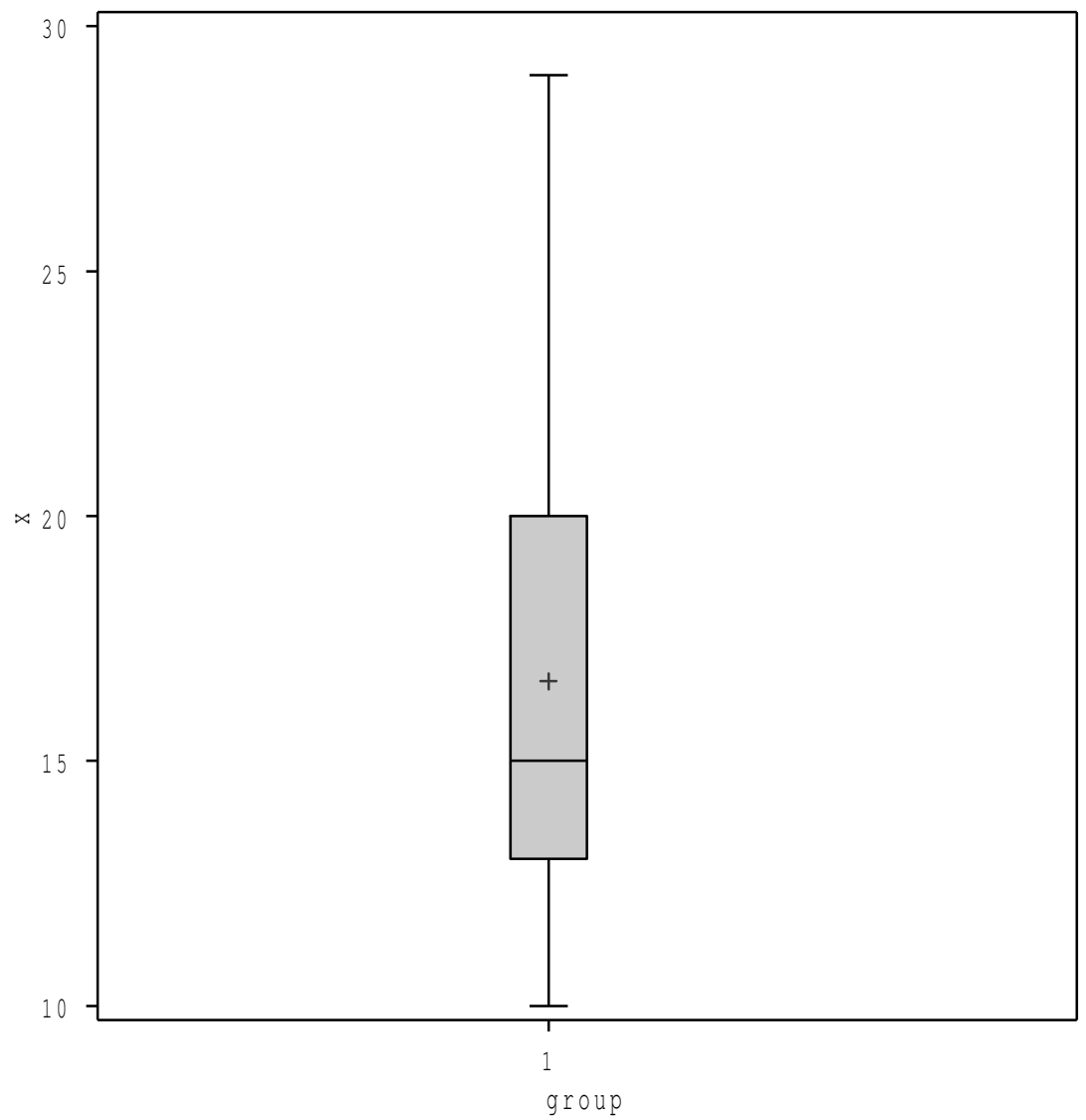


R operates by telling it what to do, in words, and gives you an answer each time. Thus the mean is 16.64, the median is 15, and the inter-quartile range is 5. The histogram suggests that the distribution is skewed to the right, and the boxplot confirms this, adding that there is a high outlier.

Now for the same thing in SAS, on the same data:

```
data xx;
  input x @@;
  group=1;
```

```
cards;  
  10 11 13 14 14 15 15 17 20 25 29  
;  
  
proc univariate;  
  var x;  
  histogram;  
  
proc boxplot;  
  plot x*group / boxtype=schematic;
```



and this code produces this output (including the boxplot, which seems to have come out above):

The UNIVARIATE Procedure

Variable: x

Moments			
N	11	Sum Weights	11
Mean	16.6363636	Sum Observations	183
Std Deviation	5.85273829	Variance	34.2545455
Skewness	1.1877081	Kurtosis	0.77406932
Uncorrected SS	3387	Corrected SS	342.545455
Coeff Variation	35.1803941	Std Error Mean	1.76466699

Basic Statistical Measures			
Location		Variability	
Mean	16.63636	Std Deviation	5.85274
Median	15.00000	Variance	34.25455
Mode	14.00000	Range	19.00000
		Interquartile Range	7.00000

Note: The mode displayed is the smallest of 2 modes with a count of 2.

Tests for Location: Mu0=0				
Test	-Statistic-	-----p Value-----		
Student's t	t 9.42748	Pr > t	<.0001	
Sign	M 5.5	Pr >= M	0.0010	
Signed Rank	S 33	Pr >= S	0.0010	

Quantiles (Definition 5)

Quantile	Estimate
100% Max	29
99%	29
95%	29
90%	25
75% Q3	20
50% Median	15
25% Q1	13
10%	11
5%	10
1%	10
0% Min	10

Extreme Observations			
----Lowest----		----Highest----	
Value	Obs	Value	Obs
10	1	15	7
11	2	17	8

13 3 20 9

The UNIVARIATE Procedure

Variable: x

Extreme Observations

----Lowest----		----Highest---	
Value	Obs	Value	Obs
14	5	25	10
14	4	29	11

For SAS, you amass a collection of commands and “submit” them all in one go. Then you get a (usually) large pile of output, and pick out what you’re looking for.

In this case, you can ignore **Moments** and look under **Basic Statistical Measures** to see that the mean, SD and median are the same as before. But the quartiles are not: R got 13.5 and 18.5, while SAS got 13 and 20. So the interquartile range is different (5 for R, 7 for SAS).

This means that SAS’s boxplot is different, because the largest value, 29, is not an outlier by SAS’s definition. Both programs use the same 1.5 times IQR above/below quartile rule, but because the quartiles are different, the judgement of outliers is different.

I intended this to be a nice easy example, but here we are, 10 minutes into the course, and we run into a problem. So why are those quartiles different?

Here were the data:

10, 11, 13, 14, 14, 15, 15, 17, 20, 25, 29

There are 11 values, so the median is the 6th one, which is the first 15.

If you come from STAB22, the instructions are that Q1 is the median of the lower half of the data. Let’s say we throw away the actual median for this. Then Q1 is the median of 10, 11, 13, 14, 14, which is 13, and Q3 is the median of 15, 17, 20, 25, 29 which is 20. This is the same answer that SAS got, both times.

Or, using the same idea, if you *keep* the median in both halves, Q1 is $(13 + 14)/2 = 13.5$ and Q3 is $(17 + 20)/2 = 18.5$. This is the same as R got, both times.

If you come from B57, you do something like this: here $n = 11$, and you note for Q1 that $2/11 = 0.18 < 0.25$ and $3/11 = 0.27 > 0.25$. So Q1 is between the 2nd and 3rd data values, but closer to the third. The formula in the B57 text gives 13.75 for Q1 and 17.75 for Q3. This is different from both SAS and R.

Now, if we were using Excel, the formula by which the quartiles were calculated would be an undocumented mystery. But we are not. We can find out what

both SAS and R do, and how to change it to our preferred definition.¹

SAS first. I found this:

Calculating Percentiles

The UNIVARIATE procedure automatically computes the 1st, 5th, 10th, 25th, 50th, 75th, 90th, 95th, and 99th percentiles (quantiles), as well as the minimum and maximum of each analysis variable. To compute percentiles other than these default percentiles, use the PCTLPTS= and PCTLPRE= options in the OUTPUT statement.

You can specify one of five definitions for computing the percentiles with the PCTLDEF= option. Let n be the number of nonmissing values for a variable, and let x_1, x_2, \dots, x_n represent the ordered values of the variable. Let the r th percentile be y , set $p = \frac{r}{100}$, and let

$$np = j + g \quad \text{when PCTLDEF=1, 2, 3, or 5}$$

$$(n+1)p = j + g \quad \text{when PCTLDEF=4}$$

where j is the integer part of np , and g is the fractional part of np . Then the PCTLDEF= option defines the r th percentile, y , as described in the following table.

PCTLDEF	Description	Formula
1	weighted average at x_{np}	$y = (1 - g)x_j + gx_{j+1}$ where x_0 is taken to be x_1
2	observation numbered closest to np	$y = x_j$ if $g < \frac{1}{2}$ $y = x_j$ if $g = \frac{1}{2}$ and j is even $y = x_{j+1}$ if $g = \frac{1}{2}$ and j is odd $y = x_{j+1}$ if $g > \frac{1}{2}$
3	empirical distribution function	$y = x_j$ if $g = 0$ $y = x_{j+1}$ if $g > 0$
4	weighted average aimed at $x_{(n+1)p}$	$y = (1 - g)x_j + gx_{j+1}$ where x_{n+1} is taken to be x_n
5	empirical distribution function with averaging	$y = \frac{1}{2}(x_j + x_{j+1})$ if $g = 0$ $y = x_{j+1}$ if $g > 0$

Weighted Percentiles

This is complicated, but the point is that SAS has five definitions of percentile, and can use any one of them. The B57 definition is #1.

R likewise has nine (!) definitions of percentile, any of which can be used. These are listed eg. at <http://stat.ethz.ch/R-manual/R-patched/library/stats/html/quantile.html>. The B57 definition is #4.

I guess the moral of this story is to make sure you know what your software has calculated for you, and to consult the documentation to be sure that it is what you wanted. So much for trying to kick you off with a simple example, eh?

¹This last bit is coming later, since it is rather beyond our scope now.

Chapter 3

Installation and connection

3.1 Some history

SAS is a venerable (read, “ancient”) statistical package. It began life in the late 1960s at North Carolina State University as the “Statistical Analysis System”, a collection of routines to analyze agricultural data. This was before there were such things as stand-alone microcomputers, or even terminals attached to a larger system; in those days you did programming on punched cards, “submitting a job” to a mainframe computer, waiting for it to be your job’s turn to run, and then retrieving printed output which (you hope) didn’t contain any errors.¹ This history is still evident in the way that SAS works now: you collect together a list of things that you want SAS to do, submit² them, and then, if there were no errors, you get some output.

The strength of SAS is that you know exactly what commands you submitted, so if you submit the same commands again you’ll get the same output. Also, SAS’s long history means that there are solid, well-tested implementations of all basic statistical methods. This reliability means that SAS has long been the software of choice in places like government, health and parts of the corporate world, where standard analyses are run over and over, and documentation is key. Its monolithic nature means that the interface with it is completely consistent: a **data** step to read in data, one or more **proc** steps to carry out an analysis.

SAS probably *is* your father’s Oldsmobile!

R came out of New Zealand. It is an open-source, free-software project that began in 1993. It has a core group of about 10 individuals who oversee its development, but anyone can contribute “packages” that extend its capabilities.

¹If it did, you had to correct them, re-submit your job and go through the whole process again.

²SAS still uses that word.

As the Wikipedia article says, “Since all work on R is strictly of a voluntary nature, the organisation is very loose, with members contributing when and as they can.”. This is in stark contrast to the highly commercialized, corporate environment of SAS. R grew out of commercial software called S, which began life in the mid-1970s, and emerged from Bell Labs into the larger world in 1980. At that time, terminals and graphic-display devices were just becoming available, and S was designed to take advantage of these. I used S on a network of Sun Workstations when I was in grad school in the early 1990s.

The R programming style is still to enter (text) commands one at a time, and see the results right away; S and thus R have always stressed the importance of graphics in statistical work, and have made those graphs easy to obtain. The one-command-at-a-time way of operating encourages the user to try something and see what happens, and the concept of the “function” allows users to add onto R in carrying out their own research, which doesn’t have to be using standard methods; R makes it easy to build your own analyses.

The downside of the user-contributed, voluntary nature of R is that different parts of R can behave in apparently inconsistent ways, not always according to what you would expect. But for many people, myself included, this is a price well worth paying for the freedom, essentially, to build your own analyses.

3.2 Installing R

R is free open-source software, so you can download it and run it on your own computer. You need two things, R itself and R Studio, which is a front end to R. My R book details the process of installing R and getting it running, so I direct you there, specifically to Chapter 1.

3.3 Connecting to SAS

3.3.1 Introduction

SAS is big, and expensive. U of T has a “site license” that allows those affiliated with it to buy SAS for their own computers for cheaper, but it has to be re-licensed each year, etc., etc., etc. If you want to go that way, you can. But the way in which SAS operates is basically the same whether you are accessing it on your own computer or not.

We are going to use SAS “remotely”. That is to say, SAS runs on a machine called `cms-chorus.utsc.utoronto.ca`, and we will connect to that machine, run SAS there, and then grab the results and do something with them on our own machine. The major issues involved in making that happen revolve around organizing the connection to the `cms-chorus` machine.

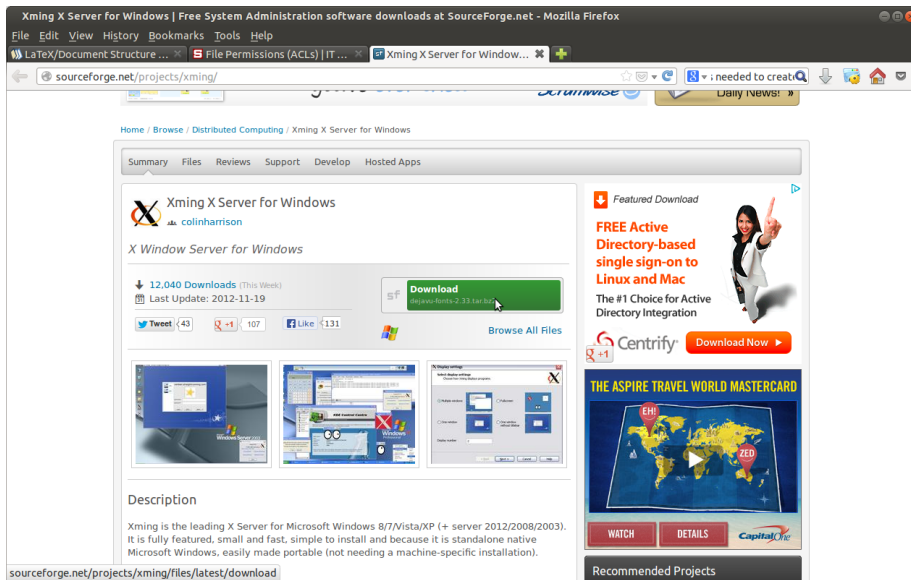


Figure 3.1: Downloading Xming

3.3.2 Setup

If you happen to be running Linux (or a Mac), there's no setup required, beyond being able to open a terminal (command-line) window. Go straight to the next section.

In the likelier event that you're running Windows, you have a couple of things to organize first.

First you need a program called **Xming** that allows SAS's windowing environment to happen in Windows (even though **cms-chorus** runs Linux). Download this from sourceforge.net/projects/xming (see Figure 3.1), clicking on the big green button, and install it in the usual way. Allow it to place a shortcut on your desktop for ease of use later. Also (optional) a thing called "Xming fonts", at <http://sourceforge.net/projects/xming/files/Xming-fonts/7.5.0.47/>, which, I am told, makes the connection run faster (it can be pretty slow if you are connecting from home). To my understanding, you just have to install this; you don't have to do anything with it.

Then you need a program called **putty**, which you can get from putty.org. See Figures 3.2 and 3.3, and note what I clicked on by looking for my cursor arrow in the screenshots. Putty actually enables you to connect to **cms-chorus**. Just download **putty.exe** and save it on your desktop - there's no install required. (Choose "save" rather than "run" to download it.)

To test your installation, first run Xming by double-clicking on its icon. This

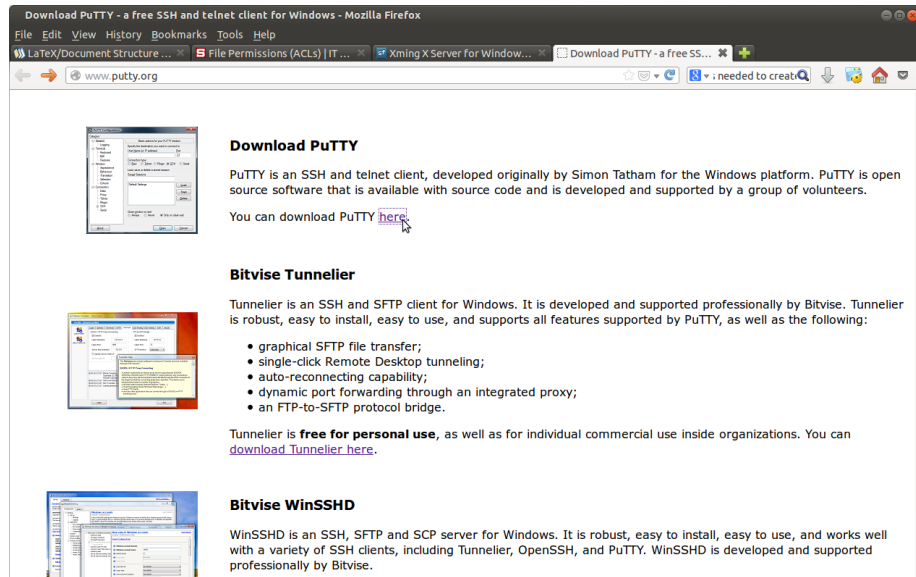


Figure 3.2: Downloading Putty (part 1)

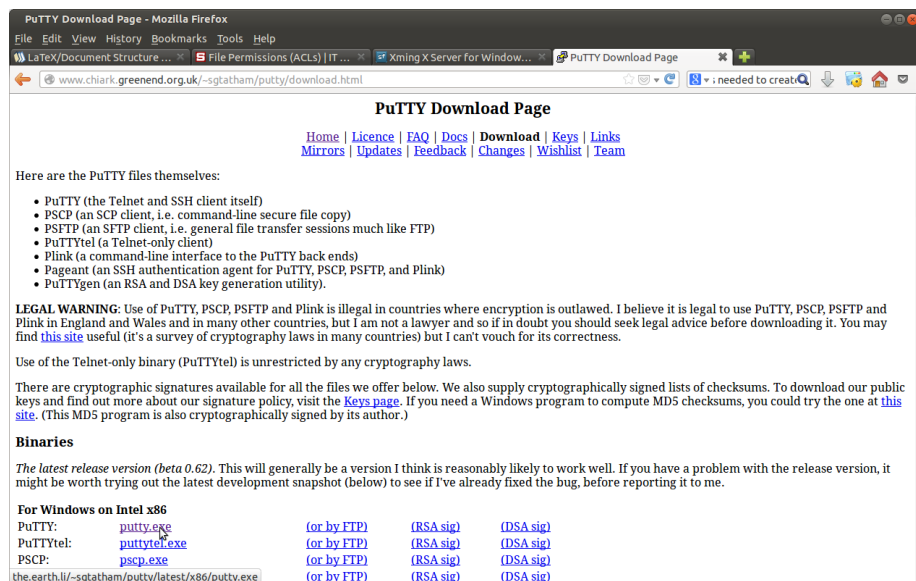


Figure 3.3: Downloading Putty (part 2)

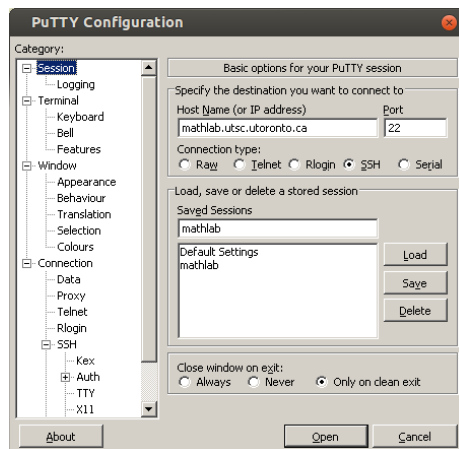


Figure 3.4: Putty setup part 1

will appear to do nothing except put a big X in your system tray. Now start Putty as well (ignoring any “unknown publisher” warnings). There are a few things to enter before you can connect to **cms-chorus**. See Figure 3.4. In the Host name box, enter **cms-chorus.utsc.utoronto.ca**. Leave Port 22 and Connection Type SSH as they are.

Then in the Category window on the left, look for SSH with a plus sign next to it (down near the bottom). Click on the plus sign. Look for X11 in the choices that pop out (the fourth one). Click on it. You should see something like Figure 3.5. On the right there is “Enable X11 forwarding” with a check box next to it. Click on the check box. Then look in the Category window on the left for Session (right at the top) and click on it. This takes you back to where you started. Save this (so you don’t have to do it every time) by first typing a name like **chorus** into the Saved Sessions box, and then clicking on the Save button on the right. The name you chose appears in the bottom box below Default Settings.

Now you can get connected. In Putty, click Open. This will bring up a black screen asking for your **cms-chorus** username and password (the same as your UTorID ones). If you can’t log in, check your password, and if you still can’t, let me know.³ If you can, you’ll see a line containing your UTorID and **@cms-chorus: \$**, and then it waits for you to type something. First start a web browser by typing **firefox&**. Then type **sas&**, and you should (eventually) see a splash screen and a whole bunch of windows appear, including SAS Explorer, SAS Log and SAS Program Editor, as in Figure 4.2.⁴ This is enough to show

³In theory, being enrolled in the course should ensure that you have an account on **cms-chorus** with the same username and password as your UTorID, but theory and practice do not always coincide!

⁴If you get some kind of error message, check that you do indeed have Xming running. Is

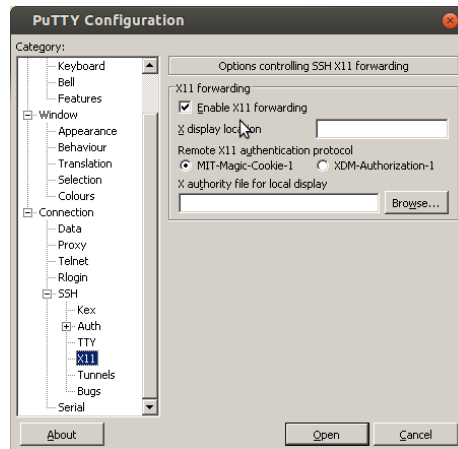


Figure 3.5: Putty setup part 2

that you have SAS running. To exit, go to SAS Explorer's File menu, select Exit, and then confirm this in the dialog box that follows.

All that setup only needs to be done once. Once you have done it and gotten everything working, starting SAS again later is the subject of the next chapter.

there a big X in your system tray?

Chapter 4

Using R and SAS

4.1 Using SAS

4.1.1 Starting SAS

On Linux or a Mac, open up a terminal window and type

```
ssh -X username@cms-chorus.utoronto.ca
```

where you replace **username** with your actual UTorID username. Jump over the next paragraph, and ignore anything about Xming.

On Windows, start up Xming, then run Putty, loading your saved **chorus** profile (click on **chorus**, then click Load, as in Figure 4.1). Click Open to connect. Enter your username (UTorID) when prompted.

Then enter your password (the one that goes with your UTorID). If it doesn't work, check it and try again; if it still doesn't work, ask me.

You'll see a whole bunch of things indicating that you are connected to **cms-chorus**. At the command prompt, first type **firefox**¹ and Enter, then type **sas** (and Enter), and in due course a whole bunch of windows will appear as in Figure 4.2. If typing **sas** gets you an error, the likely cause is that you forgot to start Xming first. The reason for opening a web browser first is that SAS produces HTML output by default.

4.1.2 Using SAS

From here on, it's the same whatever system you are running, or whatever machine you are running SAS on. (If you happen to have SAS installed on your

¹To open the web browser of that name, which is actually running *on cms-chorus*.

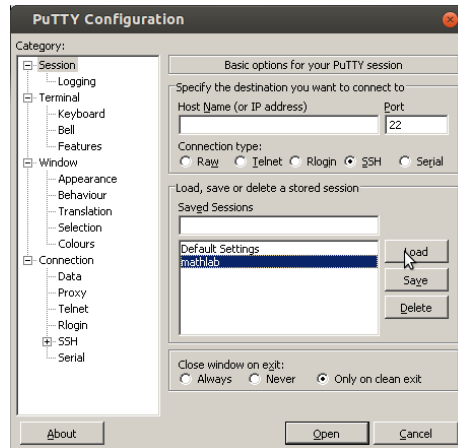


Figure 4.1: Starting up Putty again later

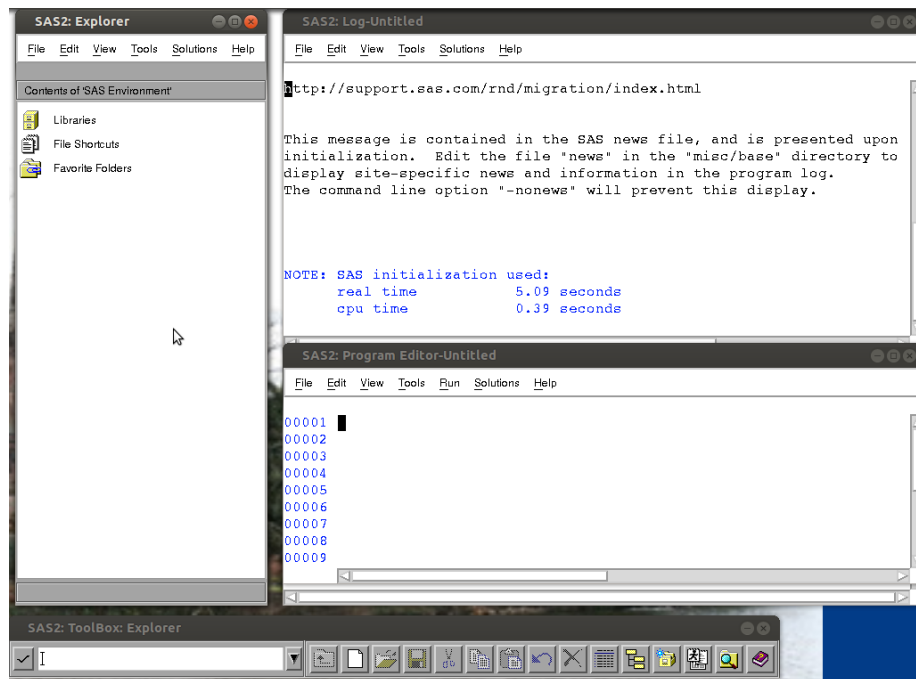


Figure 4.2: What SAS looks like when it opens

own machine, it looks exactly the same.)

SAS can produce HTML output, which looks nice. But sometimes it isn't the easiest thing to deal with. SAS also has an Output Window in which text-format output will appear, which is also just fine for us. To make this happen, we have a little work to do first. Go to the Tools menu in Program Editor. Click on Tools, Options and Preferences, and then click on the Results tab. This produces a dialog like Figure 4.3. The idea is that by default, SAS produces HTML output, which we don't want. So we have to go back and get the old-fashioned text style of output. To make that happen, click on the button next to Create Listing to check that option, and also click on the button next to Create HTML to *uncheck* that option. On Linux, which is the operating system that **cms-chorus** is running, a checked button looks pushed-down, and an unchecked button looks not-pushed-down. Figure 4.3 shows what it should look like when you're done. This only needs to be done once, because SAS will save your preferences for the future. And if you decide you *do* want HTML output again, you can go back and reverse this.

To give SAS something to work on, find the Program Editor window (with a list of line numbers down the left side). This works something like most editors you may have seen, with a couple of non-intuitive features. First off, the program editor's default is "overwrite mode", where anything you type overwrites anything that was there before. Most editors you know probably use "insert mode"; to switch to insert mode here, press the "insert" key, and now any text you already typed will get shunted to the right when you type in something new.²

The URL <http://tinyurl.com/sas-editor> redirects to the SAS support page on the text editor.

As an example, type the below (with the semicolons exactly as shown) into the Program Editor.³

²To use insert mode every time, go to the menus in Program Editor, select Tools, then Options, then Preferences. Then click the Editing tab, go down to the bottom where it says Cursor, and click on the button next to Insert.

³Confusingly, hitting Enter at the end of a line doesn't insert a new line, even in insert mode. To insert a new line, go back onto the line numbers, type "i" and Enter, and a blank line will be inserted below where you typed the "i". What if you want a blank line at the top? I haven't worked that out yet. Let me know if you discover the secret.

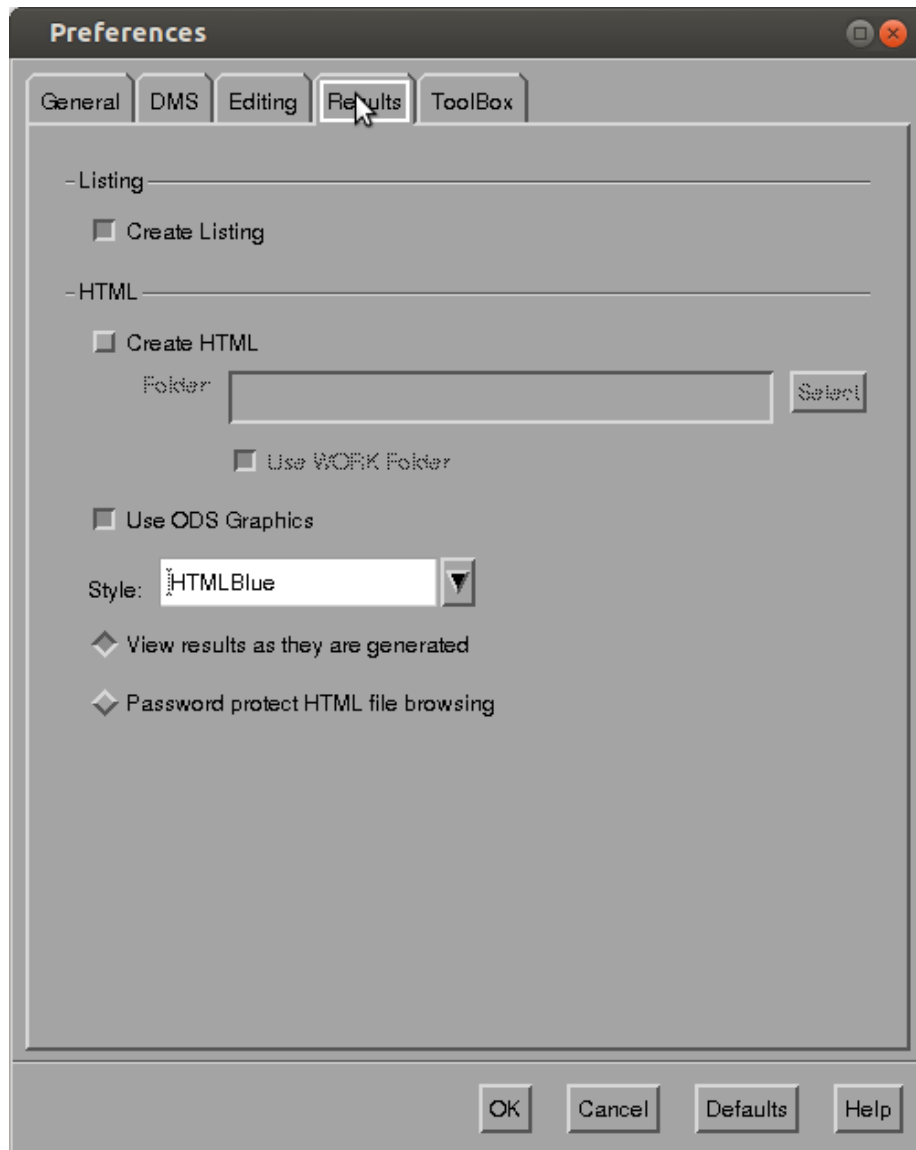


Figure 4.3: Getting text and not HTML output

```
data x;  
  input x;  
  cards;  
1  
2  
3  
5  
7  
;  
  
proc print;  
  
proc means;  
  
run;
```

This means the following:

- Here comes a data set called **x**, with one variable, also called **x**.
- Here come the data values. (You can use **datalines** instead of **cards**, but I like the throwback to the days of punched cards.)
- The five values for **x** are 1,2,3,5 and 7.
- **proc print** just lists the data, so you can check that the values were read in properly.
- **proc means** calculates means and SDs for all the variables (just **x**, here).

Once you have this right to your satisfaction, see whether it works by clicking on the Run Menu in the Program Editor and selecting Submit. This (rather worryingly) makes your painstakingly-typed SAS commands disappear⁴ and runs SAS on them. If everything (or even something) is OK, you should get some text output in the Output window. Use Page Up or Page Down to scroll.

If things didn't work (for example, you see some of what you were expecting but not all), find the SAS Log window, and look for any errors there. One time, I mistakenly typed **proc means** as **proc meanbubbles**, and got an output window with just the output from **proc print** in it, and this in the Log window:

⁴Don't worry, you can get them back.

```
1 data x;  
2 input x;  
3 cards;
```

NOTE: The data set WORK.X has 5 observations and 1 variables.

NOTE: DATA statement used (Total process time):

real time	0.46 seconds
cpu time	0.01 seconds

```
9 ;  
10  
11 proc print;  
12
```

NOTE: There were 5 observations read from the data set WORK.X.

NOTE: PROCEDURE PRINT used (Total process time):

real time	0.20 seconds
cpu time	0.05 seconds

```
13 proc meanbubbles;  
ERROR: Procedure MEANBUBBLES not found.  
14  
15 run;
```

NOTE: The SAS System stopped processing this step because of errors.

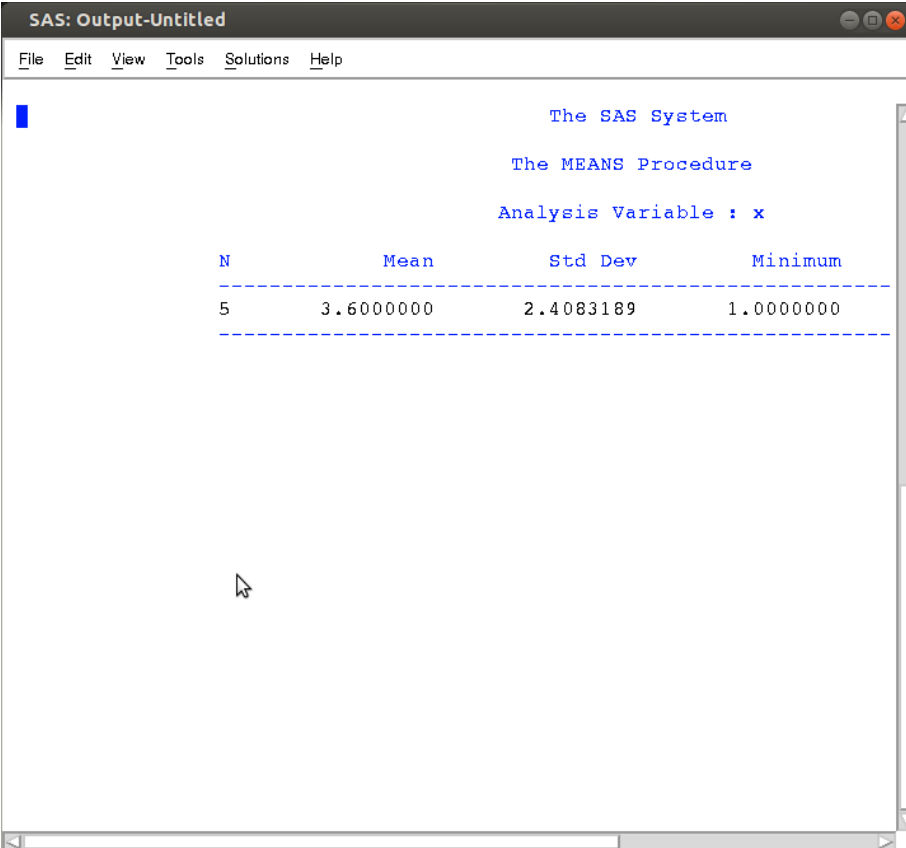
NOTE: PROCEDURE MEANBUBBLES used (Total process time):

real time	0.05 seconds
cpu time	0.00 seconds

This all means:

- The data were read in properly (there should be 5 observations on 1 variable).
- `proc print` worked just fine (no errors) and any output from it will appear.
- `proc meanbubbles` does not exist, so SAS can't run it. This is (predictably) an Error.

To get my commands back, I brought up the Program Editor window, clicked on Run, and selected Recall Last Submit. I went down to the line which mistakenly had `proc meanbubbles` on it (it was line 13), and changed it to the correct `proc means`.⁵



The SAS System

The MEANS Procedure

Analysis Variable : x

N	Mean	Std Dev	Minimum
5	3.6000000	2.4083189	1.0000000

Figure 4.4: Output window

My output window looked like Figure 4.4, with some of the results off the right-hand side. Expand the window to see it all. Also, this is the output from `proc means`; if you do expand the window, you'll see a page number 2 in the top corner. There is also a page 1, the output from `proc print`. Page Up will get you to it.⁶ SAS put the output window to the front since there were no errors.

The whole output looks like this:

```
Obs      x
1        1
2        2
3        3
4        5
5        7

The MEANS Procedure

                        Analysis Variable : x

N           Mean          Std Dev      Minimum      Maximum
-----
5           3.6000000      2.4083189      1.0000000      7.0000000
-----
```

⁵I needed to use Page Down rather than the cursor down key to get to this line. The enter key also works to move down; as previously discussed, it does *not* insert blank lines.

⁶Unless you have a Mac. In that case, do whatever you usually do to see the previous page.

`proc print` confirms that the data were read in correctly, while `proc means` actually tells us something interesting about the data.

Sometimes the output from the Output window is rather wide, and needs to be shrunk quite a bit to get it on your page.⁷ You can change this by going to the SAS Explorer window, selecting Tools, Options and System (in order), clicking the plus sign next to Log and Procedure Output Control, then clicking on SAS log and procedure output. Not exactly easy to find.⁸ In the right-hand part of the window, find Linesize (by default 100) and *right-click* on it, holding the button down and selecting Modify Value to change it to something like 80.

If you don't want to set this permanently, you can do it each time you need it⁹ by starting your code with the line `options linesize=80;`, above the `data` line.

Go back to the Program Editor, recall the commands you just submitted (Run and Recall Last Submit), and then submit them again. The output from now looks like this:

```
Obs      x
1         1
2         2
3         3
4         5
5         7
```

The MEANS Procedure

```

                        Analysis Variable : x

N               Mean           Std Dev       Minimum       Maximum
-----
5             3.6000000       2.4083189       1.0000000       7.0000000
-----
```

⁷The default width is 100 characters, while I think 80 or even less is better.

⁸ “ ‘You hadn’t exactly gone out of your way to call attention to them had you? I mean like actually telling anyone or anything’. ‘But the plans were on display...’ ‘On display? I eventually had to go down to the cellar to find them.’ ‘That’s the display department.’ ‘With a torch.’ ‘Ah, well the lights had probably gone.’ ‘So had the stairs.’ ‘But look you found the notice didn’t you?’ ‘Yes,’ said Arthur, ‘yes I did. It was on display in the bottom of a locked filing cabinet stuck in a disused lavatory with a sign on the door saying “Beware of The Leopard”’. ”

⁹I first wrote this as “on an as-needed basis”, but caught myself just in time!

which required a good bit less shrinking to get it onto the page.

4.1.3 Saving and opening

This is a good place to mention that you can save your commands (and embedded data) should you wish to use them again later. Recall the commands into the Program Editor, then select Save or Save As from the Program Editor's File Menu. This will enable you to put them in a file *on the cms-chorus machine*. By tradition, the file uses a `.sas` extension. I called mine `testing.sas`. Likewise, you can re-open a file of commands. If the Program Editor has anything in it, you'll probably want to clear it first (SAS by default adds any read-in lines to the end of the code that's already there) by selecting Edit and Clear All. Then you can open a file of commands in the way you'd guess: File and Open. You can open, edit and re-save something other than commands by looking at the bottom of the Open dialog box for File Type, and changing it to All Files.

Rather than embedding your data into your programming code, you can also save your data into a file. One way to do this is to type¹⁰ your data into the (empty) Program Editor and then save it into a file on `cms-chorus`, traditionally with the extension `.dat`, although you can use whatever you like.¹¹ The data layout (unless you are prepared to go through some contortions in SAS) is one observation per line, with values for all the variables separated by whitespace. The data below are values of a response variable y from three groups labelled a, b, and c:

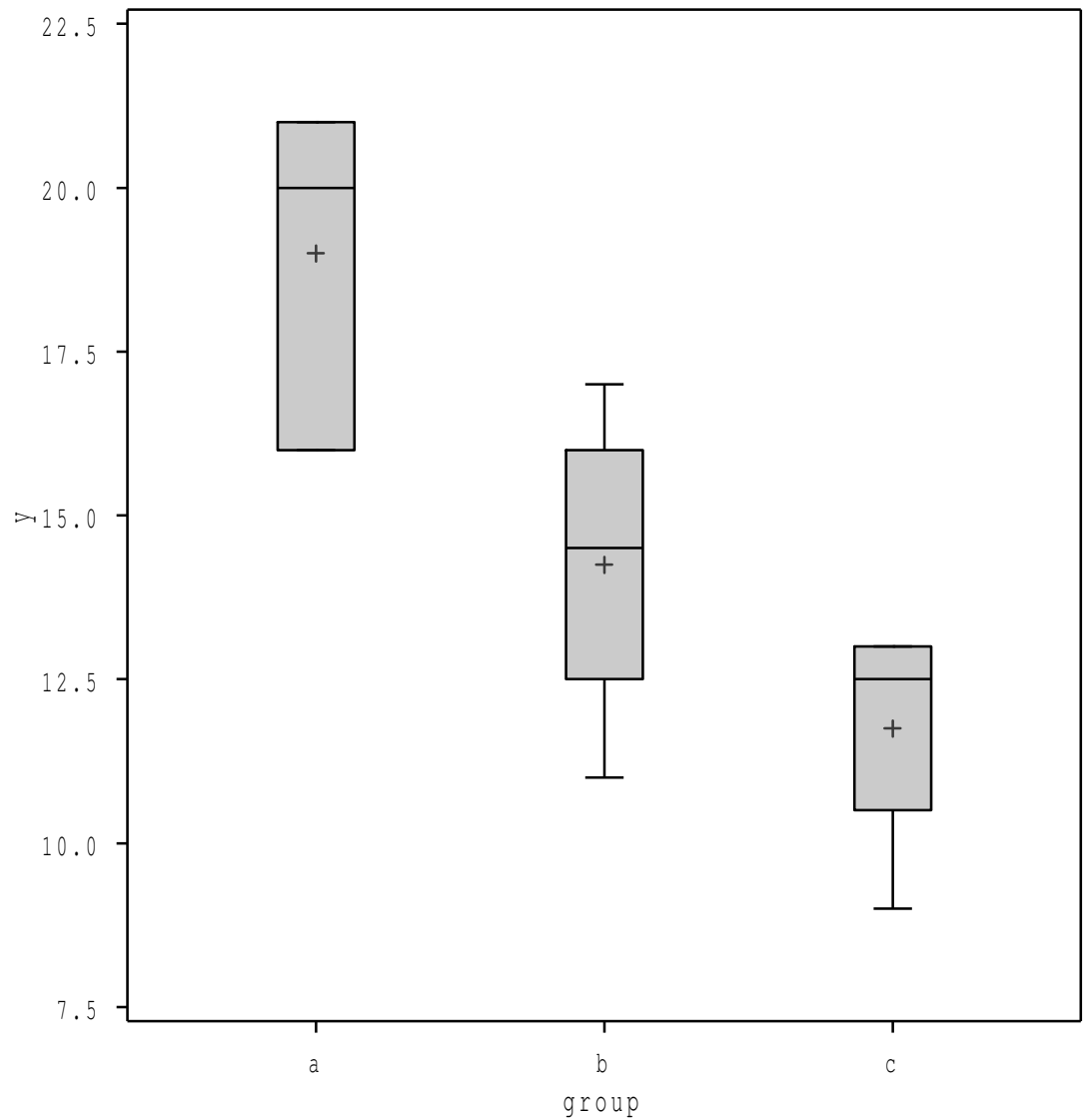
```
a 20
a 21
a 16
b 11
b 14
b 17
b 15
c 13
c 9
c 12
c 13
```

You can type these, one per line, into the Program Editor, and then save everything in a file called `threegroups.dat`. Then you can clear the Program Editor window (Edit, Clear All: don't try to submit these lines!) and type the following program:

¹⁰See the next section for some thoughts on copying and pasting.

¹¹Something like `.txt` is also fine

```
SAS> data groups;
SAS>   infile 'threegroups.dat';
SAS>   input group $ y;
SAS>
SAS> proc print;
SAS>
SAS> proc means;
SAS>   class group;
SAS>   var y;
SAS>
SAS> proc boxplot;
SAS>   plot y*group / boxtype=schematic;
SAS>
SAS> run;
```



Note that the filename has quotes around it. This is a bit cleaner than the code with `cards` (or `datalines`) and the actual data in it, because you can see rather more clearly what's going on. Running this produces no errors (check the Log window to be sure) and two pages of output plus a graph. The first page of output just lists the data, and the second shows the means for each group separately.¹² The `schematic` stuff with the boxplot gets the kind of boxplot we know, with outliers plotted separately.

¹²Your table from `proc means` might come out all on one line, depending on how wide your output text width, eg. as set by `linesize`, is.

Obs	group	y
1	a	20
2	a	21
3	a	16
4	b	11
5	b	14
6	b	17
7	b	15
8	c	13
9	c	9
10	c	12
11	c	13

The MEANS Procedure

Analysis Variable : y

group	N	Obs	Mean	Std Dev	Minimum
a	3	3	19.0000000	2.6457513	16.0000000
b	4	4	14.2500000	2.5000000	11.0000000
c	4	4	11.7500000	1.8929694	9.0000000

Analysis Variable : y

group	N	Obs	Maximum
a	3	3	21.0000000
b	4	4	17.0000000
c	4	4	13.0000000

As you would guess from looking at the data, group A has the largest mean and group C the smallest. The boxplots look whiskerless because there isn't very much data.

4.1.4 Copying and pasting

Copying into SAS is mainly straightforward: if your data have been typed into a text editor like Notepad, you can copy the values as normal and then select Edit and Paste within SAS.

One (solvable) problem is if your data are in a spreadsheet and you copy them from there. The values wind up in SAS separated by tabs rather than spaces (even though it doesn't look like it) and they have to be read into SAS carefully.

For example, suppose your spreadsheet contains this:

	A	B	C	D
1	1	4	7	
2	2	5	8	
3	3	6	9	
4	10	11	12	
5				
6				
7				

You can copy the values into the SAS Program Editor and save them as a file, say `x.dat`, but then you need to read them like this:

```
SAS> data x;
SAS>   infile 'x.dat' expandtabs;
SAS>   input a b c;
SAS>
SAS> proc print;
SAS>
SAS> run;
```

w

and you correctly get this output:

```
Obs      a      b      c
1         1      4      7
2         2      5      8
3         3      6      9
4        10     11     12
```


If you don't put in the `expandtabs` part, you will get a large number of incomprehensible error messages.

Now, the next problem is to arrange your output into a report, as for an assignment for example. If you managed to get HTML output in a web browser, there are no problems: just copy anything, including graphs, and paste into Word.¹³

Copying from SAS's output window might work all right for you. You can try it. But I seem to be unable to copy things into Word (well, LibreOffice in my case). I remember having this same problem when I used to use Windows, and I solved it by pasting into Wordpad (or some other text editor) as a staging point. *Then* I could copy from there into my document. If you're going to do that, you might as well select everything in SAS (Edit and Select All), paste it all into Wordpad, and then copy into Word just the bits that you need. Or you can use Wordpad for the entire report.

The same procedure can be used to copy from the Program Editor or even Log windows (making sure, of course, to Recall the Last Submit if you want to copy your code).

A final remark: when you paste into your Word doc (or whatever), note that SAS output comes in a monospaced (non-proportional) font, so it needs to look the same way when you hand it in. A font like Courier, or Lucida Console, is good. Also, you want to make the font small enough so that lines don't wrap (bearing in mind that SAS output lines are rather long). Otherwise it looks *really* ugly.¹⁴

4.1.5 Batch mode

In the old days of running SAS (or anything else for that matter: we are talking *decades* ago), you used a command line to run SAS, rather than having a windowing environment as we do now. This way of running SAS still exists, and is sometimes helpful to know.

You start by creating a file of SAS commands. This file is not on your computer, but on `cms-chorus`, so you need to use a text editor on `cms-chorus` to create it. There are several; I like `gedit`, which looks a bit like Notepad. To run `gedit`, type at the `cms-chorus` command line (where you typed `sas` before) `gedit myfile.sas&`, where you replace `myfile` by whatever name you want to use. What goes in this file is the same kind of thing as you would type into the Program Editor: a data step, and one or more proc steps.

¹³or OpenOffice or whatever you're using. Having said that this was easy, sometimes the copying process will mis-align columns from tables and such. Keep an eye on this.

¹⁴And, if you do it twice in my course, you'll lose marks the second time. There's no loss to you in making everything, including your comments, be in a fixed-width font. Or you can just use Wordpad for the whole thing.

When you have it as you want it, save the file and switch back to the command line. If you remembered to type the ampersand on the end of the `gedit` line, the command line will be ready for you to type something.¹⁵ What you do now is to run SAS on the file of commands you created. At the command line, type `sas myfile.sas`¹⁶ After a brief pause while SAS considers the matter, you'll get your command line back.

This, however, does not necessarily mean that everything worked. What happened is that SAS created two files, one called `myfile.log` and the other called `myfile.lst`.¹⁷ The first file, `myfile.log`, is a log of how everything went. It's the same as the log window in the windowing environment. Go back to `gedit` and open it up (you can open up several files at once in `gedit`). As with the log window, scan this file to make sure that you have the right number of observations on the right number of variables, that there are no Errors, and that you understand why any Warnings happened. If all is good, you should find a file `myfile.lst` with your output in it. Open that up in `gedit` too, and copy-paste your results to wherever you wish them to go.

What if you have graphs? These will come up in their own window, one at a time. You have to do something with the graph and close it (or click on it) before the next graph will appear, and you have to close the last graph before you get the command line back. There doesn't seem to be an obvious way to copy and paste these graphs, so you might have to take a screenshot (alt-PrintScreen) and paste *that* into your report.

Running SAS this way is equivalent to running it via the windowing environment and getting text output. You can run SAS this way on a `.sas` file that you created in the Program Editor, and you can take a `.sas` file that you created in `gedit`, open it in the Program Editor and submit from there. Either way around works.

I called this Batch mode because you are rather clearly amassing a batch of commands and submitting them all at once. (This is the way SAS works.) Then you get a pile of output to scrutinize. This is in contrast to R, where you enter one command, get some output from it, and then decide what to do next.

4.1.6 Setting up SAS to get HTML output

This may or may not work for you on `cms-chorus`. I think it is set up to do so, but you never know with things like this!

Before you type `sas &` on the command line, open a web browser *on cms-chorus*

¹⁵If not, hold down Control and press `z`. This will get you the command line back, but you'll see the word `stopped`. Type `bg` and press Enter, and everything will be good.

¹⁶With the obvious adjustments if you called your file something else.

¹⁷With the obvious adjustments, blah blah blah.

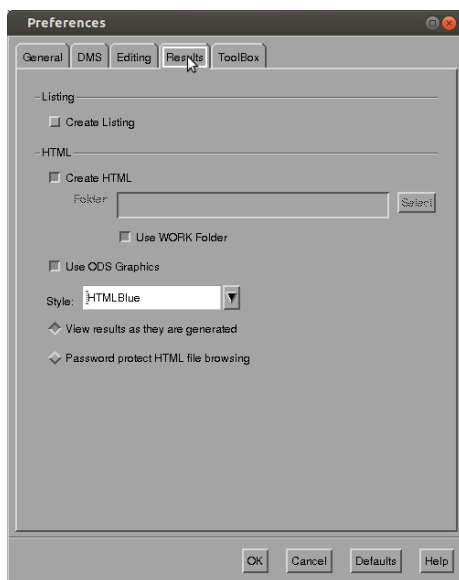


Figure 4.5: Ensuring that you will get HTML output

by typing `firefox &` and pressing Enter. This sets up a “remote browser” by which SAS on `cms-chorus` can communicate the web browser to show you your output. Then run SAS.

To make sure you get HTML output (this is the default, but you might have changed it) select Tools, Options and Preferences from any of the menus. Click on the Results tab. You’ll see a dialog like Figure 4.5. Make sure that the “Create HTML” button is checked. A checked button looks as if it has been pushed down. If you want text output (as well) the “Create Listing” button should be *down*.

Now, when you submit commands, you should see all the output, text and graphs glued together, in the web browser. It looks very nice, but if you copy it to Word, you will sometimes find that the columns in the tables have become misaligned, or have the wrong headers, or similar. So you have to be careful.

4.2 Using R

4.2.1 Introduction

Let’s try to mimic the above SAS “analysis” using R. The major difference in approach is that we run the corresponding commands one at a time and see the

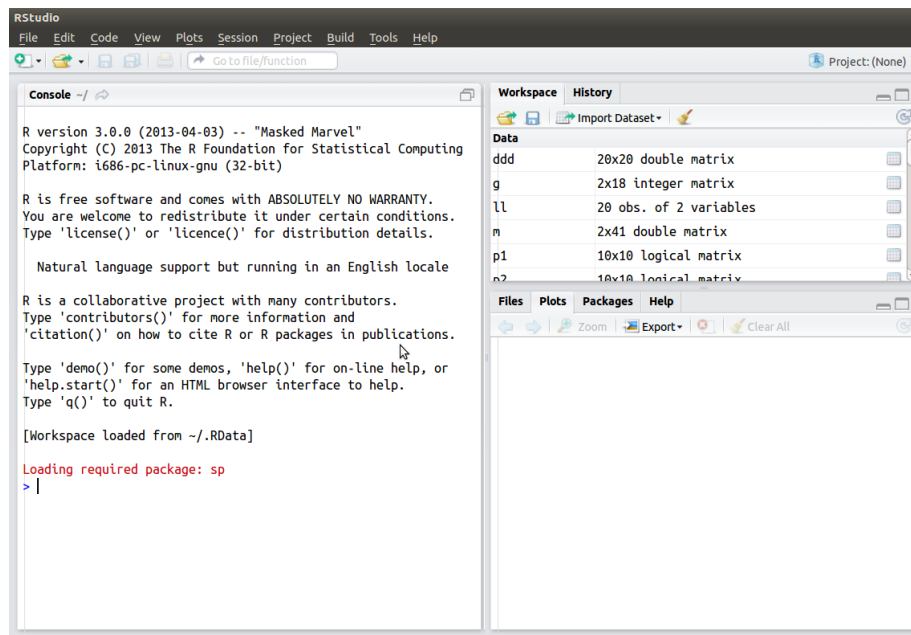


Figure 4.6: R Studio when it opens

results right away. As a result, if there are any errors, we find out about *those* right away too.

Fire up R Studio to begin. You should see something like Figure 4.6. You might have four windows instead of three. The one we’re concerned with now is the Console window, on the left (or bottom left). Click on that window. You’ll see a blue `>` with a flashing cursor next to it. R is waiting for you to ask it to do something. Try entering the line below.

```
R> x=c(1,2,3,5,7)
```

You’ll get the prompt back. No comment means no error. This means “glue the values 1, 2, 3, 5 and 7 together into a list and call it `x`”. To convince yourself that `x` really does contain those values, enter its name at the prompt:

```
R> x
[1] 1 2 3 5 7
```

This shows you that `x` does indeed contain the proper values. The `[1]` before the values says “these are the values, starting at the 1st one”.

To get the mean of the values in `x`, ask for it, like this:

```
R> mean(x)
[1] 3.6
```

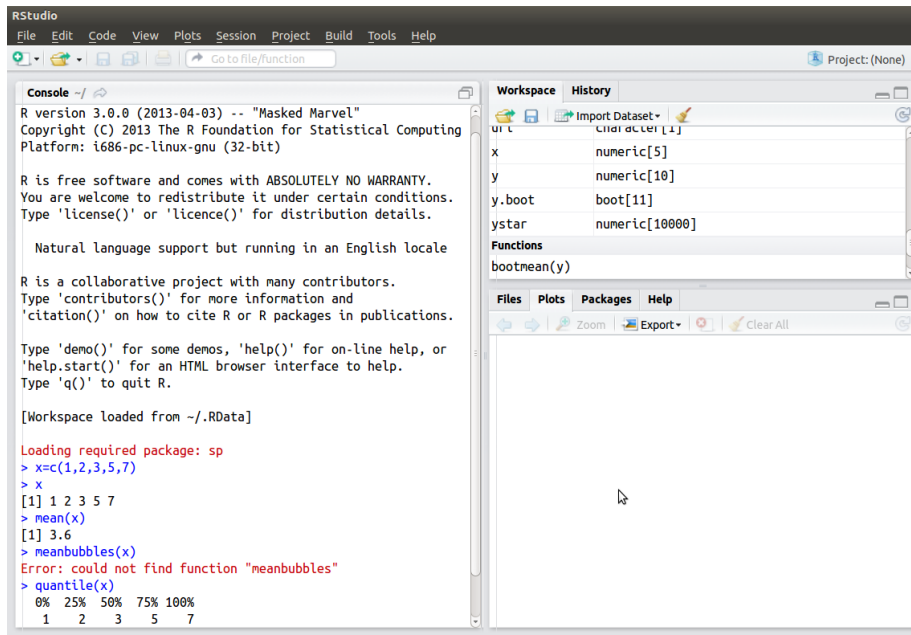


Figure 4.7: Screenshot of commands and output in R Studio

`sd(x)` works similarly. You can try it. One more useful function is `quantile`, with an “n”, which gets you (by default) the quartiles, with an “r”:

```
R> quantile(x)

 0%  25%  50%  75% 100%
 1   2   3   5   7
```

In Figure 4.7, you see how my R Studio looked after entering those commands, in blue. I also made an error, which you can see flagged in red. Since R works one line at a time, you know about an error right away; you don’t have to go back, fix up, and re-submit a whole pile of commands. The output — the results — are in black.

4.2.2 Projects and R scripts

I also want to show you how to read in data from a file, but before we get to that, I want to show you some housekeeping stuff that will help you organize your work in R. We’re going to create a Project, which is a kind of container for commands, data and stuff all connected with one piece of work, like an assignment. After that, we’re going to create and use an R script, which is like that string of commands we fed into SAS, only more flexible.

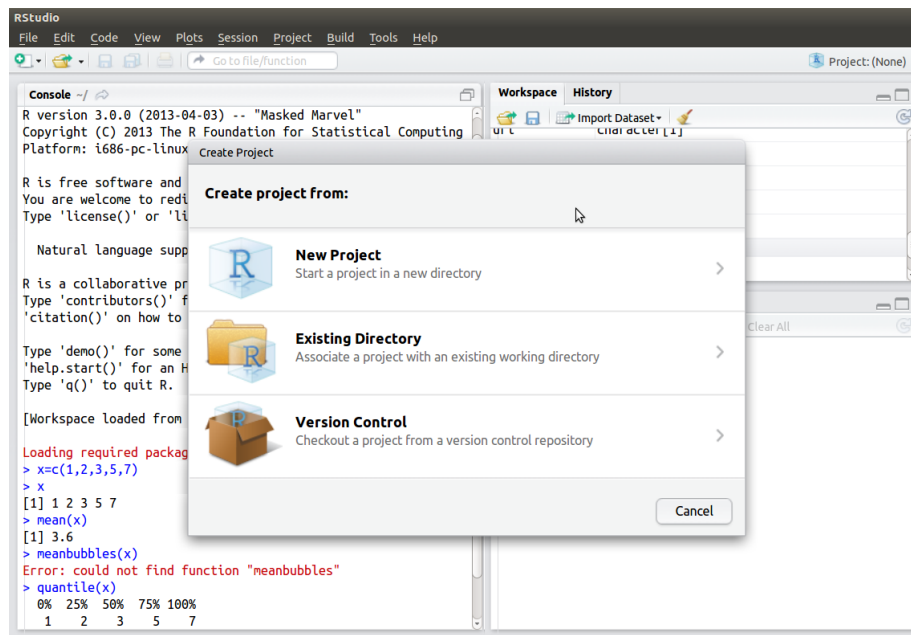


Figure 4.8: Create Project

All right. In R Studio, look for the Project menu, or click on where it says “Project: (none)” at the top right. Either way, select Create Project. You’ll see Figure 4.8. Now, you can choose to create a new directory (folder) to hold the stuff for this project, or you can use a folder that already exists. Click on the appropriate thing, and then use the Browse button to navigate to (as appropriate) where you want the new folder to be created, or where the already-existing folder lives. Figure 4.9 shows me creating a new project `r-work` in a new folder on my desktop. Click on Create Project. R Studio switches to the new project.

One more thing. Select File, New and R Script. R Studio now has *four* windows, as in Figure 4.10. The top left one is a place to type commands and, if you will, submit them. The advantage to that is you get to save and re-use your commands and you don’t have to type them again.

Figure 4.11 shows R Studio with the commands typed into the R Script box. I really ought to save them before I do anything else! To “submit” them, you can either click on the right-hand Source, which will run them all at once, SAS style, or you can put the cursor on the line you want to run and then click where my mouse pointer is,¹⁸ which will run just that line (and move your cursor down to the next line). A handy keyboard shortcut for this is Control-Enter. Another way to “submit” several lines is to select them first, and then click on Run, which

¹⁸You can see the tooltip on the screenshot.

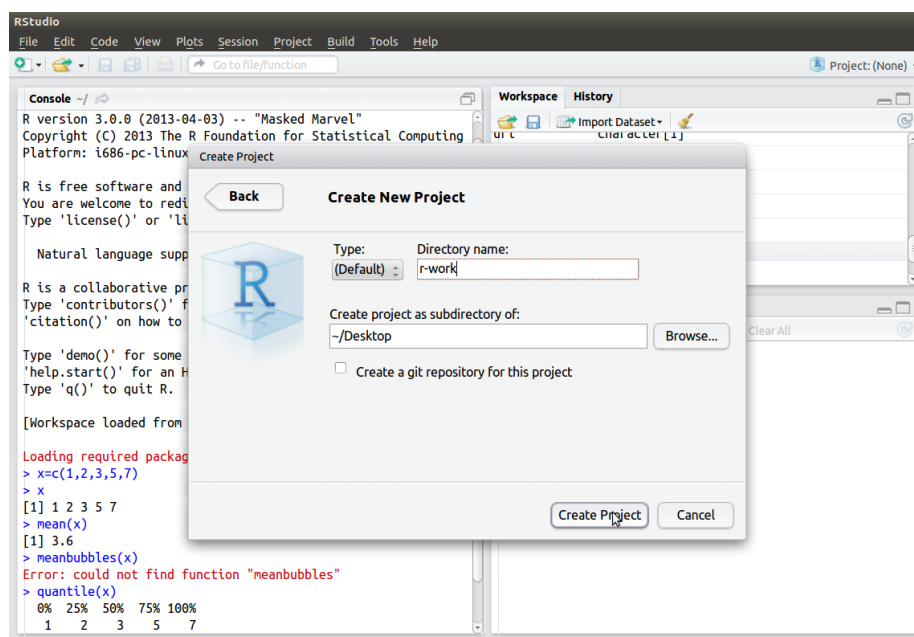


Figure 4.9: Creating a new project

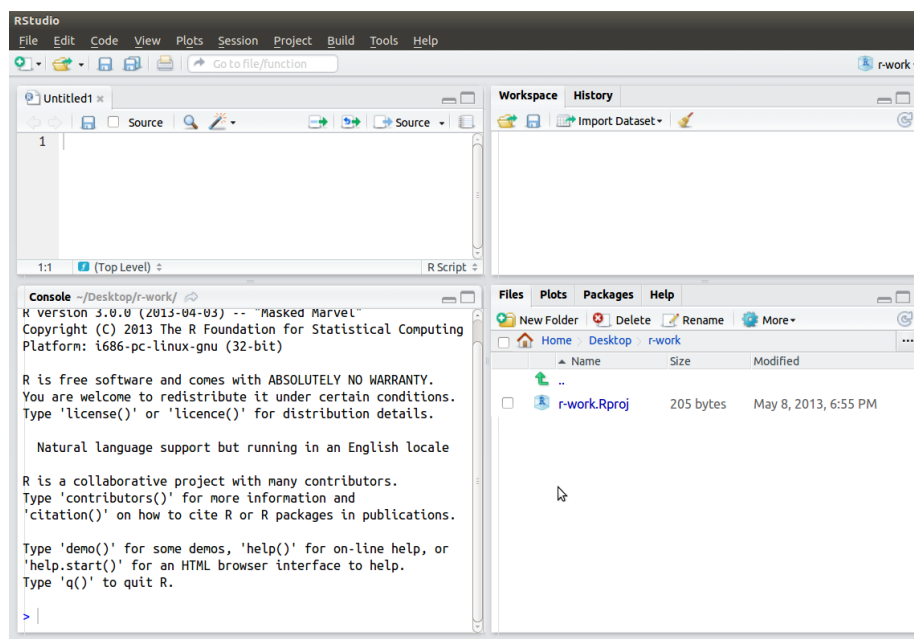


Figure 4.10: Project with R script ready to go

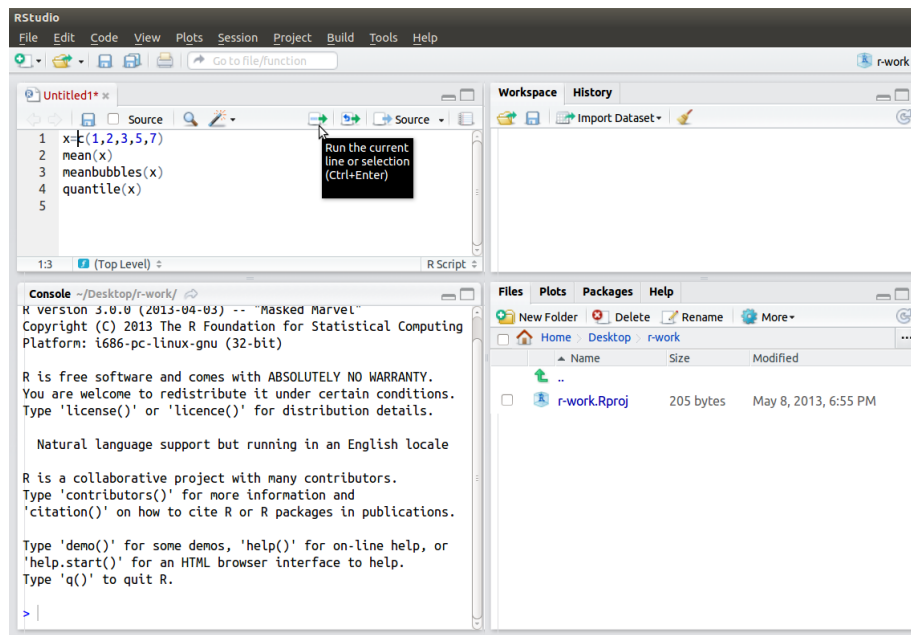


Figure 4.11: Ready to run some commands

will run all the lines you selected.

When you've done this, the commands and output will appear together in the bottom left Console window, and from there you can copy them to a report.

4.2.3 Reading data from a file

The “basic” way of reading data from a file is, as for SAS, to have one observation per line and the values for all the variables separated by whitespace.

The top-left corner of R-studio can also be used as a text editor. You can create a new data file with File, New, Text File, or you can open an existing data file, such as the file `threegroups.dat` that we used with SAS. For SAS, we just had `group` in one column and `x` in the other, because the `data` step gave names to the variables when we read them in. With R, however, it's easier to put the variable names on the first line of the data file, like this. I saved this file as `threegroups.txt`:

```
group y
a 20
a 21
a 16
```



```
b 11
...
```

and then when you read the values in, you tell R that the first row is headers, like this:¹⁹

```
R> mydata=read.table("threegroups.txt",header=T)
R> mydata
```

```
  group  y
1     a 20
2     a 21
3     a 16
4     b 11
5     b 14
6     b 17
7     b 15
8     c 13
9     c  9
10    c 12
11    c 13
```

The variable `mydata` that holds all the data values is called a **data frame**: it's a rectangular array with rows and columns, the rows being observations (individuals) and the columns variables. There are a couple of ways of getting at the variables in a data frame:

```
R> mydata$y

[1] 20 21 16 11 14 17 15 13  9 12 13
```

This means “the `y` that lives in `mydata`”. Alternatively, you can do this:

```
R> attach(mydata)
R> y

[1] 20 21 16 11 14 17 15 13  9 12 13
```

There is no variable called `y`, but when you `attach` a data frame, any variables that R cannot otherwise find are looked for in that data frame. Here, `y` must be `mydata$y`.

The problem with attaching data frames is that you get a lot of extra variables floating around, and you might end up wondering “where did that `y` and `group` come from anyway?”²⁰ Unless you’ve been keeping track. When you are finished with the `attached` variables, you can do this:

```
R> detach(mydata)
```

¹⁹If you *don't* have headers, it seems to pay to specify that with `header=F`. R doesn't always guess right.

²⁰Programmers call this “polluting the name space”.

and now there are no “extra” variables called `y` or `group` any more.

4.2.4 Means by group

This is a little trickier in R than it was in SAS (where we just called `proc means`). What we do looks like this:

```
R> attach(mydata)
R> tapply(y,group,mean)
```

```
      a      b      c
19.00 14.25 11.75
```

First of all, to save some typing, we **attach** the data frame. (No need to do this if the data frame is already attached.) Next, to calculate something “for each group”, `tapply` is often what we need. It needs three things: the variable we’re calculating for, then the variable that does the dividing into groups (a categorical variable, or in R terms, a **factor**), then finally the thing you want to calculate, in our case the mean.

The same logic applies to calculating anything else by group. This, for example, calculates the inter-quartile ranges for each group. I **detached** the data frame afterwards, since I don’t need it **attached** any more:

```
R> tapply(y,group,IQR)
R> detach(mydata)
```

```
      a      b      c
2.50 2.25 1.75
```

Another way of doing this is via the function `aggregate`, which works like this:

```
R> aggregate(y~group,data=mydata,mean)
```

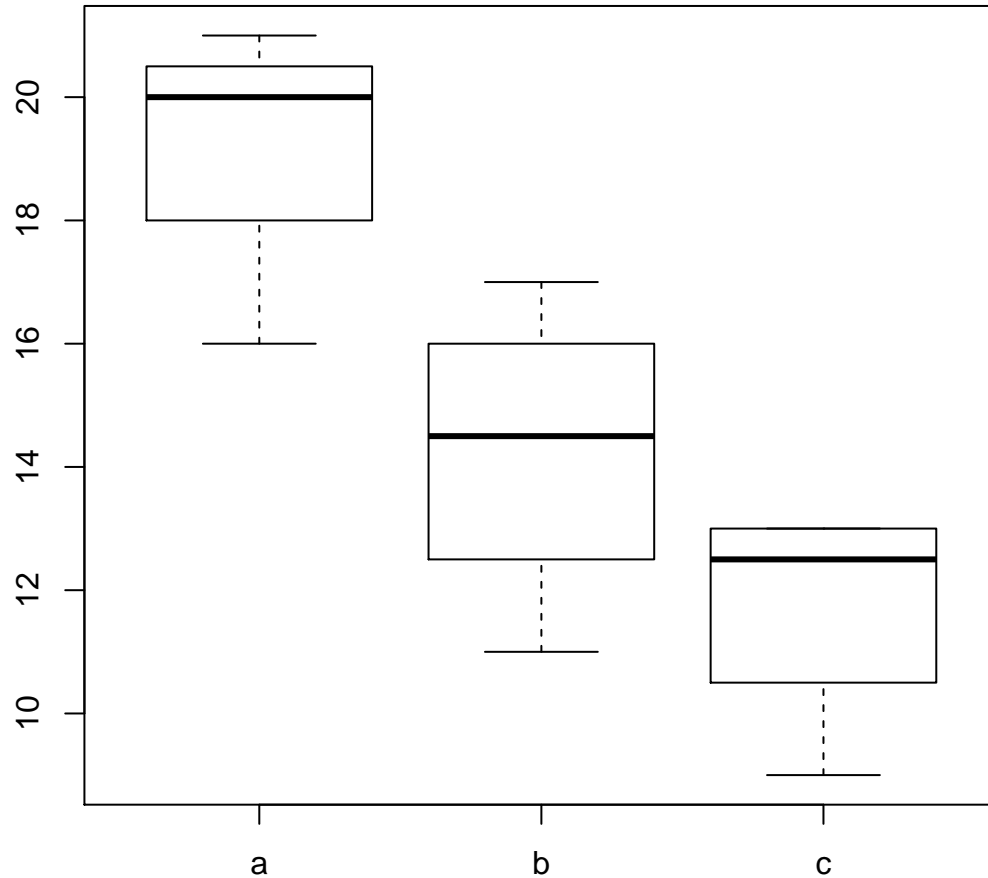
```
  group      y
1     a 19.00
2     b 14.25
3     c 11.75
```

An advantage of this is that you don’t need to **attach** the data frame first; you specify the variables (“`y` as it depends on `group`”), then the data frame where those variables live, and then what you want to calculate for each group. The thing with a squiggle in it, the first thing fed into `aggregate`, is called a **model formula**. We’re going to see a lot of those, indeed in a few seconds when we draw a boxplot.

4.2.5 Boxplot

The same “model formula” idea applies to how you define the groups for side-by-side boxplots. You can use an attached data frame, in which case you don’t need the `data=` below (or in `aggregate` either, come to that):

```
R> boxplot(y~group,data=mydata)
```

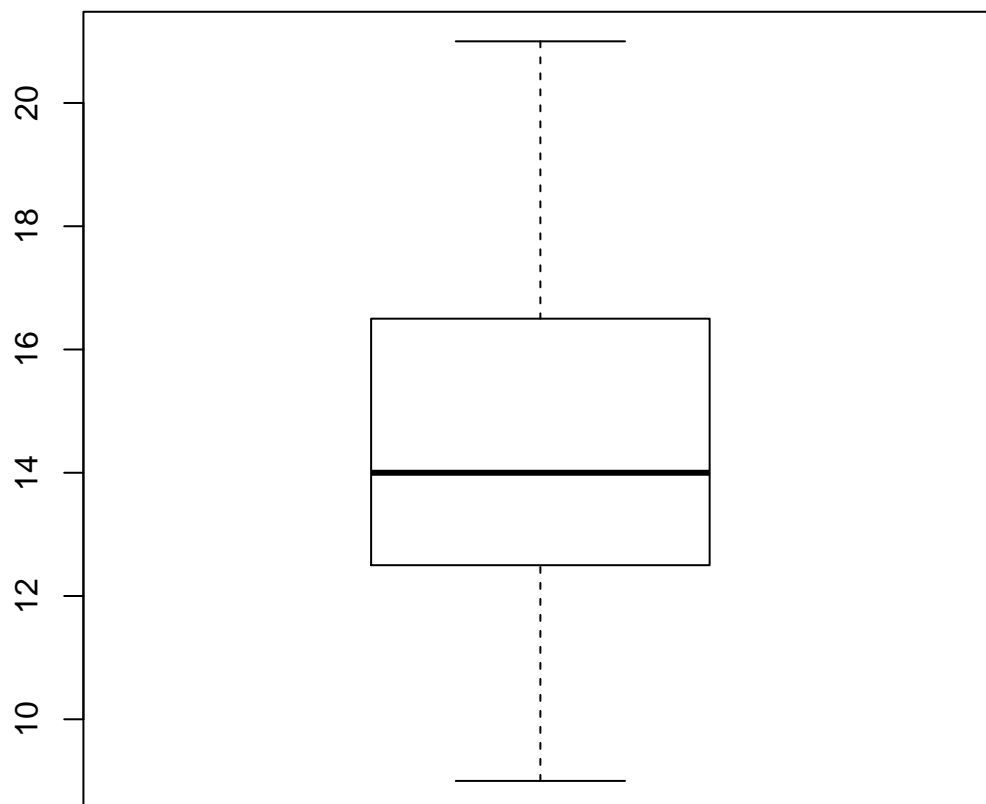


Again we get those rather silly boxplots, because we don't have much data.²¹ Notice that when you do this in R Studio, the plot appears in the bottom right window, on the Plots tab.

If you don't have a grouping variable, you omit the squiggle and the grouping variable. For example, this gets a boxplot of all the values in `y`, regardless of group. Note that with no model formula, we can't use `data=`, so we have to use the full name of `y` (or `attach` the data frame again):

```
R> boxplot(mydata$y)
```

²¹They are slightly different from SAS, though, because of the differing definitions of quartile.



This boxplot comes up on top of the first one in R Studio, but the other boxplot isn't lost: use the arrows just below the Plots tab to cycle back and forth among your graphs.²²

In SAS, if you don't have a grouping variable, you have to rather artificially create one. R doesn't make you go through those contortions.

²²There is apparently a limit of 30 graphs that R Studio will keep. I discovered this while working with someone who was producing 36 graphs at one time, and he wondered why the first few kept disappearing.

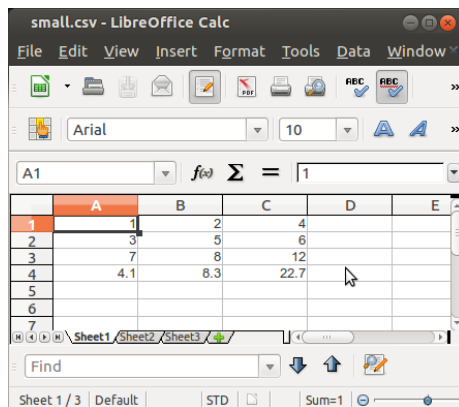


Figure 4.12: A small spreadsheet

4.2.6 Reading data from a spreadsheet

The best way to get data from a spreadsheet into R is via a `.csv` file. This stands for “comma-separated values”, and is a way of translating the values in a spreadsheet, but not the formulas, into plain text that programs like R can read.²³ So the first step in reading spreadsheet data into R is to save the spreadsheet as a `.csv` file, using File and Save As in your spreadsheet program.

Figure 4.12 shows a small spreadsheet. The columns don’t have names. The numbers are actually all values rather than formulas, but it wouldn’t make any difference if any of them were formulas.

I saved the spreadsheet in Figure 4.12 as `small.csv`, in my project folder that I created earlier. I read it into R like this, and you can check that the values are correct:

```
R> zz=read.csv("small.csv",header=F)
R> zz
```

	V1	V2	V3
1	1.0	2.0	4.0
2	3.0	5.0	6.0
3	7.0	8.0	12.0
4	4.1	8.3	22.7

Now I can do whatever I like with `zz`. Because the `.csv` file didn’t have variable names, I read it in with `header=F`, and R supplied variable names for me. If you don’t like the ones supplied, you can change them:

²³The formulas get translated into the values that the formulas evaluate to.

```
R> mynames=c("Foo","Blah","Ding")
R> names(zz)=mynames
R> zz
```

```
      Foo Blah Ding
1 1.0  2.0  4.0
2 3.0  5.0  6.0
3 7.0  8.0 12.0
4 4.1  8.3 22.7
```

and you see that the variable names have changed.

SAS can read `.csv` files too. There's one trick to make it skip over the commas separating the values (`dlim` being short for "delimiter"):

```
SAS> data stuff;
SAS>   infile 'small.csv' dlm=',';
SAS>   input foo blah ding;
SAS>
SAS> proc print;
```

```
Obs      foo      blah      ding

1         1.0        2.0        4.0
2         3.0        5.0        6.0
3         7.0        8.0       12.0
4         4.1        8.3       22.7
```

Going back to R, you can also copy and paste the values from a spreadsheet into R Studio. Open a new Text File first, and paste the values into the empty window. Save this into your project folder. I used the filename `small.txt`. Then use `read.table` as usual:

```
R> fred=read.table("small.txt",header=F)
R> fred
```

```
      V1 V2  V3
1 1.0 2.0 4.0
2 3.0 5.0 6.0
3 7.0 8.0 12.0
4 4.1 8.3 22.7
```

I don't know whether the values got copied from the spreadsheet with embedded tabs or not, but either way, `read.table` read them in all right.

In case you're wondering, `read.csv` is a "special case" of `read.table`. You could also use `read.table` to read in a `.csv` file, but it would be more work:

```
R> morefred=read.table("small.csv",header=F,sep=",")
R> morefred
```

	V1	V2	V3
1	1.0	2.0	4.0
2	3.0	5.0	6.0
3	7.0	8.0	12.0
4	4.1	8.3	22.7

Chapter 5

A data analysis example

Many studies have suggested that there is a link between exercise and healthy bones. Exercise stresses the bones and this causes them to get stronger. One study (done at Purdue University) examined the effect of jumping on the bone density of growing rats. The 30 rats were randomly assigned to one of three treatments: a control with no jumping, a low-jump condition (the jump height was 30 cm), and a high-jump condition (60 cm). After 8 weeks of 10 jumps per day, 5 days per week, the bone density of the rats (expressed in mg/cm³) was measured.

The data are saved in a file `jumping.txt`, in two columns, the first being the name of the group (as one word), and the second being the bone density, a number. We're going to do two parallel analyses, one in R and one in SAS. Let's start with SAS:

```
SAS> data rats;
SAS>   infile 'jumping.txt' firstobs=2;
SAS>   input group $ density;
```

Open up the Program Editor and type these lines in. There are two new things here: first, the top line of the data file contains the names of the variables, but we are about to tell SAS that anyway, so we want to start on line 2. We tell SAS this on the `infile` line by specifying `firstobs` and the line where we want it to start reading data. Second, the groups have *names* rather than numbers. When you read something into SAS that isn't a number, you need to put a `$` after the name of the variable, here `group`. You can submit just these lines. There won't be any output, but you can check the Log window to be sure everything looks good. Mine looked like this:

```
21 data rats;
22     infile 'jumping.txt' firstobs=2;
23     input group $ density;
24
25 run;
```

NOTE: The infile 'jumping.txt' is:
Filename=/home/ken/teaching/c32/notes/jumping.txt,
Owner Name=ken,Group Name=ken,
Access Permission=rw-rw-r--,
Last Modified=Fri May 10 15:33:22 2013,
File Size (bytes)=415

NOTE: 30 records were read from the infile 'jumping.txt'.
The minimum record length was 12.
The maximum record length was 13.

NOTE: The data set WORK.RATS has 30 observations and 2 variables.

NOTE: DATA statement used (Total process time):
real time 0.03 seconds
cpu time 0.00 seconds

At the top, we have a repeat of our code. The first NOTE is all about the file that we're reading from. At the end, SAS tells us that 30 lines were read with 2 variables on each line. We know this is correct, so, so far so good. You can confirm this by running `proc print`:

```
SAS> proc print;
```

Obs	group	density
1	Control	611
2	Control	621
3	Control	614
4	Control	593
5	Control	593
6	Control	653
7	Control	600
8	Control	554
9	Control	603
10	Control	569
11	Lowjump	635
12	Lowjump	605
13	Lowjump	638
14	Lowjump	594
15	Lowjump	599
16	Lowjump	632
17	Lowjump	631
18	Lowjump	588
19	Lowjump	607
20	Lowjump	596
21	Highjump	650
22	Highjump	622
23	Highjump	626
24	Highjump	626
25	Highjump	631
26	Highjump	622
27	Highjump	643
28	Highjump	674
29	Highjump	643
30	Highjump	650

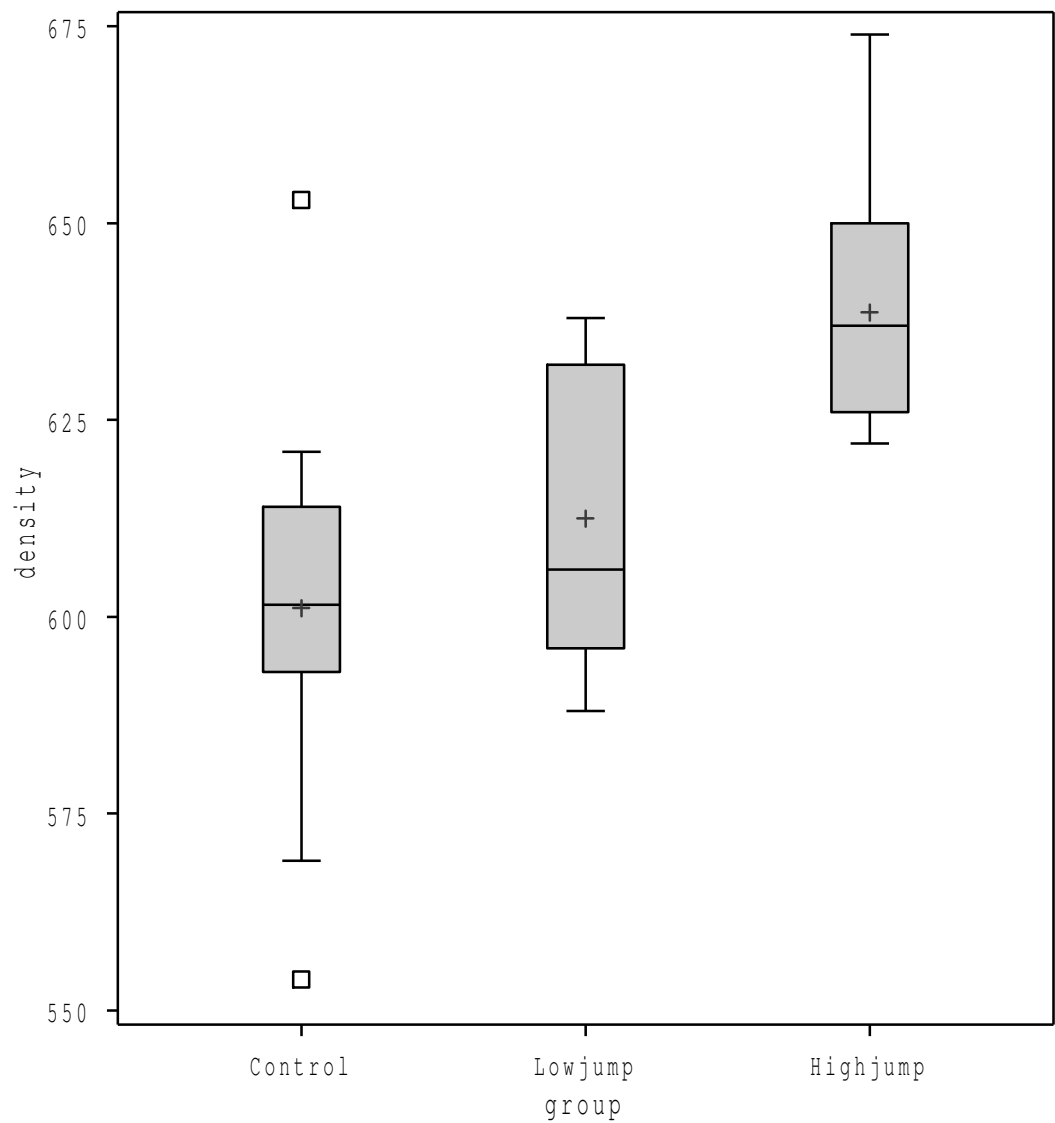
In case you're worried about where those submitted lines are going, SAS is keeping track of them for you. When you Recall Last Submit, you get the lines you submitted last, and if you select it *again*, you get the lines you submitted before *that* (placed at the top of the Program Editor), and so on. But it's probably sensible to recall the lines and save them from time to time.

That data set `rats` that we created will stick around until we close down SAS. Later on, we'll see how to save it as a "permanent" SAS data set, so that we don't have to go through the `data` step again to read it in.

Anyway, our data look good. You can check a few lines of the data set with the data file and be sure that they correspond.

A good first step is to draw a graph. This helps to show up any erroneous values, and gives us a sense of what's going on. I like a boxplot in this kind of situation, one for each group. We already know how to do that. I'll use `boxstyle=schematic` to show up any outliers. A histogram (or stemplot) for each group would also work.

```
SAS> proc boxplot;  
SAS>   plot density*group / boxstyle=schematic;
```



What do we learn from the boxplots? The lines across the middle of the boxplots mark the median, so the medians for the `control` and `lowjump` groups are similar. The median for the `highjump` group is higher, though. Something similar is true for the means (marked by + on the SAS boxplots), though the `lowjump` mean is higher than the `control` mean.

What about variability? The top and bottom of the boxes are the quartiles, so the height of the box is the inter-quartile range. From that point of view, the `lowjump` numbers have the biggest spread, but once you get outside the quartiles for that group, the values don't spread much further (the whiskers are very short). The other two groups, though, have one long whisker, and, in the case of the `control` group, some big-time outliers, so you'd expect the standard deviation to be telling a different story from the inter-quartile range.¹

The boxplots also tell us about shape. The `highjump` numbers are skewed to the right, since the upper whisker is longer, and the mean is a little bigger than the median. The `control` group *looks* skewed if you just look at the whiskers, but there are two outliers, one each end, that confuse the issue. The `lowjump` group is also kind of confusing: the whiskers don't indicate any kind of skewness, but the mean is noticeably bigger than the median. This must be because most of the values inside the box are down at the bottom, with a few values being right up near the top. That's about the only configuration of the data that would make that boxplot possible.

Let's do the same thing with R now. Open up R Studio, open up a project (or create a new one if you like), and grab an R Command window. In that window, type the following:

```
R> rats=read.table("jumping.txt",header=T)
R> rats
```

When you run this (select both lines and click on Run, or place your cursor on the first line and press Control-Enter twice), you'll see all 30 lines of data. If not, well, you *did* put the data file in the project folder, didn't you?

	group	density
1	Control	611
2	Control	621
3	Control	614
4	Control	593
5	Control	593
6	Control	653
7	Control	600
8	Control	554
9	Control	603
10	Control	569
11	Lowjump	635

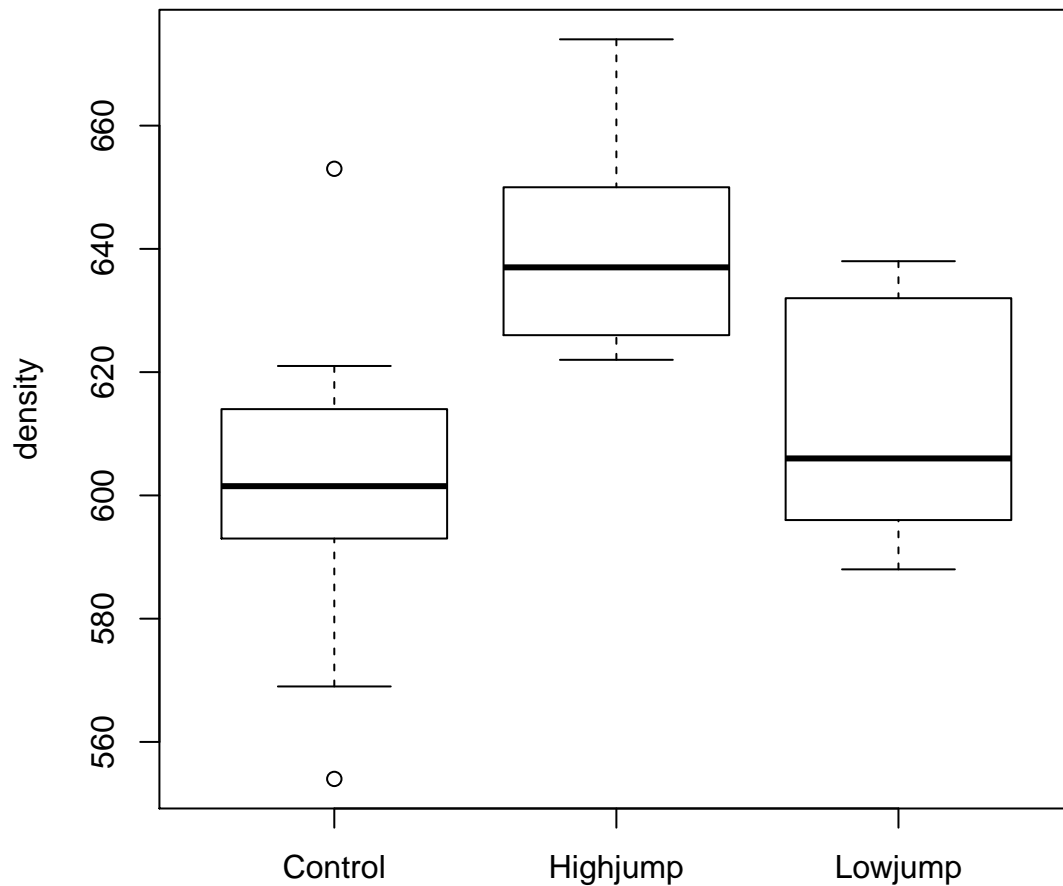
¹We'll look at that in a minute.

12	Lowjump	605
13	Lowjump	638
14	Lowjump	594
15	Lowjump	599
16	Lowjump	632
17	Lowjump	631
18	Lowjump	588
19	Lowjump	607
20	Lowjump	596
21	Highjump	650
22	Highjump	622
23	Highjump	626
24	Highjump	626
25	Highjump	631
26	Highjump	622
27	Highjump	643
28	Highjump	674
29	Highjump	643
30	Highjump	650

I find R's code for getting the boxplots more meaningful than SAS's. This uses a model formula and a `data=` statement to tell R which data frame to get the variables from.²

```
R> boxplot(density~group,data=rats,ylab="density")
```

²Or you could say `attach(rats)` and omit the `data=`, but then you have variables named `group` and `density` lying around.



R's boxplots look a bit shorter and fatter than SAS's. It's rather easy to get confused, since R put the group names in *alphabetical order*, while SAS left them in the same order that it found them in the data file. Apart from that, though, the boxplots from R look similar to those from SAS: the **control** group has two big outliers, **lowjump** and **control** have similar medians, while the median for **highjump** is definitely bigger.

All right, how about means, standard deviations and such for each group? This is one case where SAS makes it easier than R does. Here's SAS:

```

SAS>  proc means;
SAS>      var density;
SAS>      class group;

```

The MEANS Procedure

			Analysis Variable : density			
group	N					
	Obs	N	Mean	Std Dev	Minimum	Maximum
Control	10	10	601.1000000	27.3636011	554.0000000	653.0000000
Highjump	10	10	638.7000000	16.5935061	622.0000000	674.0000000
Lowjump	10	10	612.5000000	19.3290225	588.0000000	638.0000000

and here's what I think is the best way in R:

```
R> aggregate(density~group,data=rats,mean)
R> aggregate(density~group,data=rats,sd)
```

```
      group density
1 Control    601.1
2 Highjump   638.7
3 Lowjump    612.5
      group density
1 Control  27.36360
2 Highjump 16.59351
3 Lowjump  19.32902
```

Either way, the mean bone density for the `highjump` group is higher than for the other groups, while the `control` and `lowjump` groups have similar bone density on average. The standard deviations tell us that the bone density is most variable for the control group. If we you back and look at the boxplots, you might guess that this is because of the outliers; the inter-quartile ranges tell a quite different story. I want to delve into the R code a bit. You might think we can base our calculations on

```
R> mean(rats$density)
```

```
[1] 617.4333
```

but that just gives the mean bone density for *all* the rats, never mind about which group they're in. How do we pick out the rats that were, say, in the `control` group?

We will discover that R is *very* flexible: you can do almost anything with R; you just have to figure out how. The key to selecting stuff is to do something like this:

```
R> attach(rats)
R> iscontrol=(group=="Control")
R> iscontrol
```

```
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE FALSE FALSE FALSE FALSE FALSE
```

The piece `group=="Control"` is a *logical* statement: it is asking “is *group* equal to `Control`?” and is expecting an answer of `TRUE` or `FALSE`. The question gets asked one rat at a time, so we get 30 answers, which we save in the variable `iscontrol`. You can see that the first 10 rats are `TRUE`, that is, they *are* in the control group, and the rest are not.³

³The `[13]` on the start of the second line indicates that the first `FALSE` on that line is the 13th one, so that the two `FALSE`s on the previous line are the 11th and 12th, and there are ten `TRUE`s altogether.

Now, the next stage of this is to get the bone densities that belong with the `control` group. That's done like this, with the second line showing you how to get the mean of these values:

```
R> density[iscontrol]
R> mean(density[iscontrol])

[1] 611 621 614 593 593 653 600 554 603 569
[1] 601.1
```

Or you can select the rows of the data frame that go with being in the control group. This requires extra thought, because a data frame has rows *and* columns, and we need to say which of both of those we want. For rows, we want the rows for which `iscontrol` is `TRUE`, and we want all the columns. Below, the rows we want go before the comma, and the columns we want go after. Since we want all the columns, we leave the “columns we want” part blank. So we end up with this:

```
R> rats[iscontrol,]

      group density
1 Control      611
2 Control      621
3 Control      614
4 Control      593
5 Control      593
6 Control      653
7 Control      600
8 Control      554
9 Control      603
10 Control     569
```

A shorthand (though more confusing) way is to do it all in one go:

```
R> density[group=="Control"]
R> mean(density[group=="Control"])

[1] 611 621 614 593 593 653 600 554 603 569
[1] 601.1
```

where you would read the second line as “for those rats in the control group, take their bone densities, and then find the mean of those.” I personally hate the complicatedness of nested brackets, and prefer to do things one step at a time. So my habit is to define my logical vector of `TRUE`s and `FALSE`s according to what I want to select, and then use that to select what I want.

Something that might have confused you is the presence of `=` and `==`, sometimes on the same line. The `==` is a “logical equals”, that is, it produces something that is either true or false. The single `=` says “work out what is on the right-hand side of the `=`, and store it in the variable whose name is on the left side of the `=`”. So, in

```
R> iscontrol=(group=="Control")
```

the `==` says “decide, for each rat, whether its group is equal to `Control`”, thus producing a string of `TRUE` and `FALSE`, one for each rat. This is a “logical vector” in the R jargon. Then the `=` says to take that logical vector and call it `iscontrol`, so that we can use it again later.

You might like to think about how you would find the mean of those rats in the `highjump` group. One way is to find the means of all the groups via `aggregate` as I did it above, but you can do it separately, either via the overwhelmingly scary

```
R> mean(density[group=="Highjump"])
[1] 638.7
```

or in steps via

```
R> ishigh=(group=="Highjump")
R> highdens=density[ishigh]
R> mean(highdens)
[1] 638.7
```

The one-liner says, starting from the inside:⁴ “take the rats in the `highjump` group, pull out their densities, and find the mean of those”. The three-liner almost requires you to think backwards: “I want the mean of something. The something is some bone densities. Which ones? The ones that are in the high jump group.” Or, train yourself to think about the selection first: “Which rats am I interested in? The ones in the high-jump group. Let’s figure out which ones those are first. What do I want to know about them? Their bone densities. What do I want to know about *those*? The mean.”

Finally, we should tidy up after ourselves:

```
R> detach(rats)
```

The logical next stage is to ask ourselves is the research question (that the people who collected these data undoubtedly asked themselves): “is it *really* true that the amount of jumping is associated with bone density?” The flip side of that is “or are the differences we observed just chance?”. This takes us into the murky waters of statistical inference, coming up next.

⁴The best way to read these, if you want to read them at all.

Chapter 6

Statistical Inference

6.1 Introduction

In the previous example, we did *descriptive* statistics. Our attitude was “here are the data. What do they have to say?” That is often all we need. But we may need to take some action based on what the data tell us, or to *generalize* beyond the data we have to a larger world. This is the way science proceeds: First you make a guess about how the world works. Then you collect some data from a small subset of the world. Then you decide whether, based on the part of the world you have seen, whether your guess is true for the whole world, or not.

This is what statistical inference is about.

There are a few more things that need to be said, but we’ll follow up on them as we work through an example.

6.2 Example: learning to read

Let’s suppose you have devised a new method for teaching children to read. Your guess is that it will be more effective than the current way of doing it, but you don’t know that. So what you do is to collect some data. Not just any data, though. You want to generalize to “all children in the world”, or something like that, so the children in your study really ought to be a simple random sample of all the children in the world. This is hard to arrange, though, so what is usually done is to argue that the children that are actually in your study are “typical” of all the children you care about, even though they all come from some small number of schools in Ontario, say.

Note the role of randomization here: whether a child is in the study or not has nothing to do with anything else about the child (like, whether they are good at reading already, or they need some help).

As an aside, you might have developed a program that is good for teaching *struggling* children to read, in which case the “all kids” you care about is “all kids who are having trouble learning to read”, and you take a sample of *those*. But in our case, we’re assuming that our method of teaching reading is good for everybody.

The next thing we need is a response variable: something that measures how well each child has learned to read. Let’s imagine we have some standard reading test in mind, and we use the score from that.

Now, we could just give all the children in our study our new reading program, and the end of the study say “look how well they can read!”. But you might imagine that the kids in this study might get a bit more attention than they normally would in school, and that might encourage them to do better. So we should compare them against a “placebo”: the usual reading program, but under otherwise the same conditions as the new one. That way, we are comparing apples with apples.

So now we need to decide which of the children in our study should get the new reading program, and which ones should get the standard one. We need to be careful about this too. For example, we might give the new program to the children who appear to need it most. But that would bias the results: the new program probably wouldn’t look so good, not because the program itself is bad, but because the children doing it have more trouble reading, compared to the other group.

The right way to do this is to assign children to the new or standard reading program in a way that has nothing to do with anything else. That means randomization again. R makes that easy. Let’s suppose we have 44 children available (as in the data set we are going to use). We’ll split them in half¹ and choose 22 at random to use the new program to learn reading. We need to label the 44 children somehow, either by their names or by numbering them 1 through 44 (as here):

```
R> set.seed(457299)
R> s=sample(1:44,22)
R> sort(s)
```

```
[1] 2 5 6 8 9 10 13 14 15 16 21 22 23 24 26 28 30 38 39 41 42 44
```

This shows which children end up getting the new reading program. The rest get the standard program. I sorted the sampled values to make it easier to see what they were.

¹Not literally!

Another way of doing it is to “sample” all 44 children, which has the effect of shuffling them around:

```
R> sample(1:44,44)

[1]  6 28 38 25 32 40  9 30 19  7 39 14 15 26 36 29  2 24 23 27 17 34 37 44  5
[26] 35 11 43 13 18 31  1 22  3 20  4 16 33 42 12 41 10 21  8
```

Then we take the first 22 to get the new reading program, and the rest to get the standard one.

This second approach works well if you are selecting more than two groups. For our jumping rats, we had 30 rats altogether that were randomly assigned to 3 groups. We might have done the randomization by numbering the rats from 1 to 30 and selecting the groups like this:

```
R> sample(1:30,30)

[1]  6 29 26 19 25  3 22 16 21 28 30 20 12 23  7 17  4 11  1  9  2  8 27 10 18
[26] 14 15 13  5 24
```

then putting the rats with the first 10 of these shuffled numbers into the control group, the next 10 into `lowjump` and the last 10 into `highjump`.

You might have wondered why we chose our groups to be the same size. The idea is that we want to have the best chance of finding any differences between the groups that might exist.² This is done by making sure that the overall group means³ are all as well-estimated as possible, given that we only have 30 rats (or 44 children) to work with. This is done by ensuring that none of the means are too badly estimated, that is, that the smallest group is as big as possible. And **this** is done by making all the groups the same size.

All right, let’s go back to our children learning to read. There’s no way of pairing up the children in the two groups, and we haven’t done any matching or anything like that, so you might think of doing a two-sample *t*-test. Before we do that, though, we should do some descriptive stuff to make sure that everything is OK for that. I’m going to use SAS for this. In the data file, the groups are labelled `t` for “treatment” (the new program) and `c` for “control” (the standard program):

```
SAS> data kids;
SAS>   infile 'drp.txt' firstobs=2;
SAS>   input group $ score;
SAS>

SAS> proc means;
SAS>   class group;
SAS>   var score;
```

²You might recall that this is the *power* of a test.

³Ie. the means of “all rats that undergo the different types of jumping”, not just the ones we happened to observe.

```

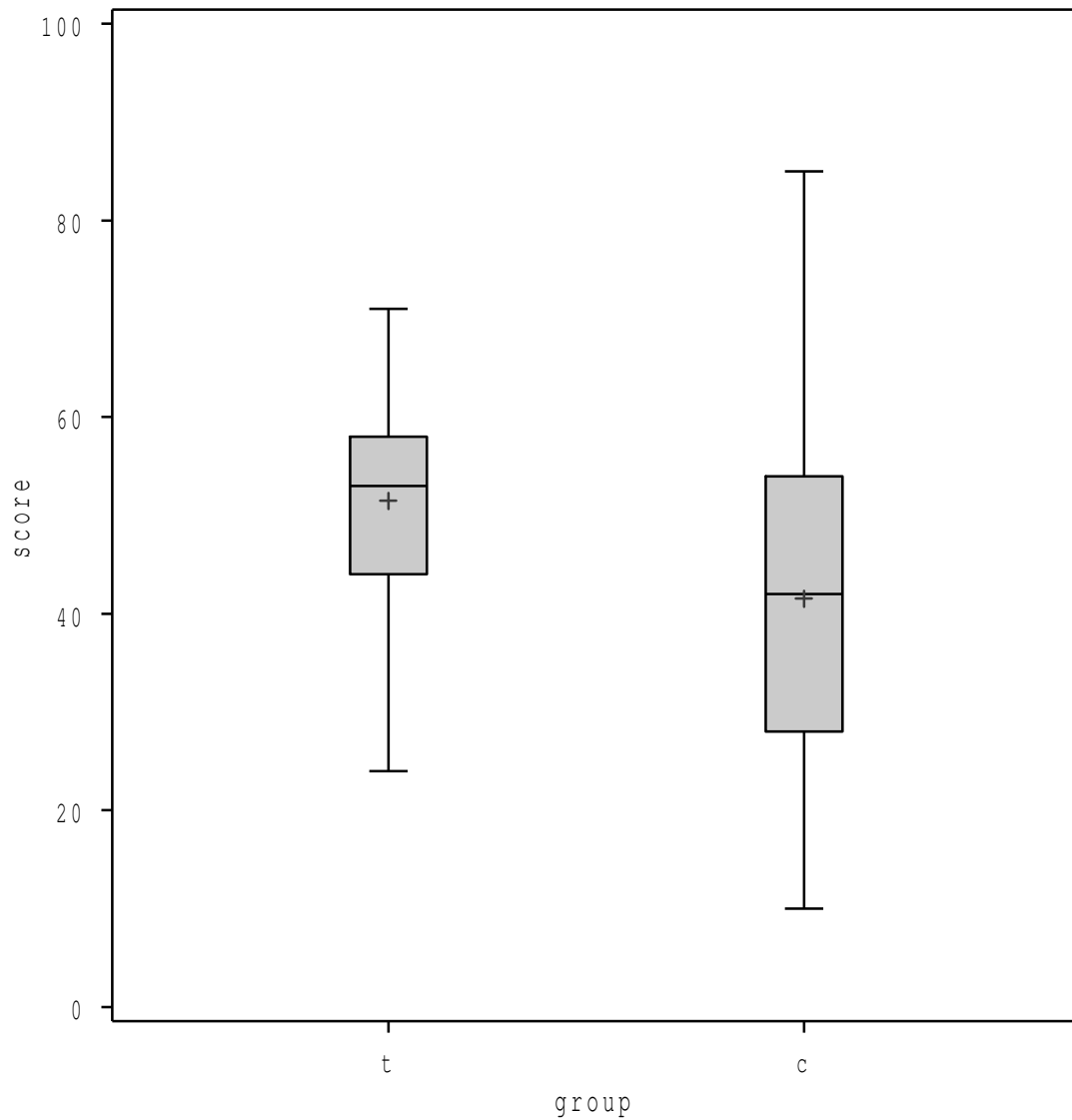
SAS>
SAS> proc boxplot;
SAS>     plot score*group / boxstyle=schematic;
SAS>

```

The MEANS Procedure

Analysis Variable : score

group	Obs	N	Mean	Std Dev	Minimum	Maximum
c	23	23	41.5217391	17.1487332	10.0000000	85.0000000
t	21	21	51.4761905	11.0073568	24.0000000	71.0000000



First, we read in the data, not forgetting the \$ after `group` because it is letters and not numbers. We could also use `proc print` to check that the data were read in properly, but I skipped that here.

The output from `proc means` shows that the 44 children weren't actually equally split between the reading programs, with 23 in the new-program group. The usual reason for this is that the study *starts* with equal group sizes, but some of the experimental subjects drop out for some reason.⁴

⁴In this case, maybe the parents of a couple of the kids moved to California, so the kids concerned couldn't stay in the study.

The means do differ by what looks like quite a bit (10 points), but there is a fair bit of variability, so we might imagine this 10-point difference could have happened by chance.

The standard way of doing this is a two-sample t -test. This is based on the data coming from normal distributions with possibly different standard deviations.⁵ If the new reading program is really better, the mean score for children in that group will be higher, but if the new program has no effect whatever, the population means for the two groups will differ only by chance.

For our data, the normality part looks reasonable, since both distributions are more or less symmetric with no outliers.⁶ So I would trust this test here.

SAS eats this kind of thing for breakfast. There is a `proc ttest` that does all the variations of t -tests (one or two samples, matched pairs or independent samples). The only problem is organizing which way the comparison is. If you look back at the output from `proc means`, you'll see that the control group `c` is listed first. So the comparison is control vs. treatment, and therefore if the new reading program is effective, the control should be *less* than the treatment in terms of mean reading scores. So we put a `side=1` ("l" for "lower") on our `proc ttest` line, and specify our response variable `score` and our grouping variable `group` as in `proc means`.

```
SAS> proc ttest side=l;
SAS>     var score;
SAS>     class group;
```

The TTEST Procedure

Variable: score

group	N	Mean	Std Dev	Std Err	Minimum	Maximum
c	23	41.5217	17.1487	3.5758	10.0000	85.0000
t	21	51.4762	11.0074	2.4020	24.0000	71.0000
Diff (1-2)		-9.9545	14.5512	4.3919		

group	Method	Mean	95% CL Mean	Std Dev	95% CL S
c		41.5217	34.1061 48.9374	17.1487	13.2627 2
t		51.4762	46.4657 56.4867	11.0074	8.4213 3
Diff (1-2)	Pooled	-9.9545	-Infity -2.5675	14.5512	11.9981 4
Diff (1-2)	Satterthwaite	-9.9545	-Infity -2.6913		

Method	Variances	DF	t Value	Pr < t
Pooled	Equal	42	-2.27	0.0143
Satterthwaite	Unequal	37.855	-2.31	0.0132

Equality of Variances

Method	Num DF	Den DF	F Value	Pr > F
Folded F	22	20	2.43	0.0507

⁵This is the Welch-Satterthwaite version of the two-sample t test, rather than the pooled one that assumes that the two populations have equal standard deviations.

⁶In fact, the normality part doesn't matter too much.

SAS being SAS, you get more output than you could possibly use. The important part is the line with **Satterthwaite** on it. This is the proper t -test here. The P-value is 0.0132, which is smaller than the usual cutoff of 0.05, so we can conclude that reading scores on average *are* higher for the new program.

We already said that the two groups don't appear to have the same (population) SD, so we are not entitled to use the pooled t -test. Having said that, you can compare the P-values from the pooled and Satterthwaite tests, and they are almost identical (0.0143 vs. 0.0132). This often seems to be the case. The bottom test on the SAS output is a test of equality of standard deviations for the two groups. This P-value of 0.0507 is very nearly significant, so we should have doubts about the scores in the two groups having equal standard deviations, and therefore we should have doubts about using the pooled test. As we concluded.⁷

The analysis in R proceeds much the same way. I'll skip the drawing of the boxplots, since you already know how that goes.

As with SAS, the only real difficulty is with saying what happens if the new reading program really does work. It turns out that we need to say "less" again, this time because *c* is before *t* alphabetically. You can rely on R treating categorical variables, or "factors", this way.

```
R> kids=read.table("drp.txt",header=T)
R> t.test(score~group,data=kids,alternative="less")

Welch Two Sample t-test

data:  score by group
t = -2.3109, df = 37.855, p-value = 0.01319
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
 -Inf -2.691293
sample estimates:
mean in group c mean in group t
 41.52174      51.47619
```

R does only the Welch-Satterthwaite test by default. R is more concise this way. But it provides you ways to get the other stuff by asking for it.

If you happen to want the pooled test, you ask for it via `var.equal=T`:

```
R> t.test(score~group,data=kids,alternative="less",var.equal=T)
```

⁷A famous statistician compared the use of a test on standard deviations before deciding whether to use the pooled or Satterthwaite t -test as "closing the stable door after the horse has bolted". His reasoning is that having unequal SDs matters most when the groups are small, but in this case the test for SDs is *less* likely to reject. Once you have moderate or large sized groups, as we do here, having unequal SDs doesn't matter so much.

Two Sample t-test

```
data:  score by group
t = -2.2666, df = 42, p-value = 0.01431
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
      -Inf -2.567497
sample estimates:
mean in group c mean in group t
      41.52174      51.47619
```

If you happen to want a two-sided test, you leave out the `alternative=` part:

```
R> t.test(score~group,data=kids)
```

Welch Two Sample t-test

```
data:  score by group
t = -2.3109, df = 37.855, p-value = 0.02638
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
      -18.67588      -1.23302
sample estimates:
mean in group c mean in group t
      41.52174      51.47619
```

If you want to test the standard deviations for equality,⁸ there is also `var.test`. This one is two-sided:

```
R> var.test(score~group,data=kids)
```

F test to compare two variances

```
data:  score by group
F = 2.4272, num df = 22, denom df = 20, p-value = 0.05067
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
      0.9973228  5.7984382
sample estimates:
ratio of variances
      2.427158
```

6.3 Randomization test

Another way of testing whether the new reading program has a positive effect is via a *randomization test*. The idea here is that if the new reading program

⁸Though see the other footnote for why you probably don't want to do that.

has no effect, then each observation might equally well have come from either group. We observed a difference in sample means of about 10. How likely is it, if we assign `c` and `t` at random to the observed scores, that would observe a treatment mean minus control mean of 10 or bigger?

This is something we'll need to build ourselves, which means that SAS is *not* the tool for the job. We can build anything in R⁹, if we just can work out how.¹⁰

All right, step 1: randomly shuffle the treatment and control labels. Let's `attach` the `kids` data frame first. We already know how to shuffle things randomly: use `sample`:

```
R> attach(kids)
R> myshuffle=sample(group,length(group))
R> myshuffle

[1] t t c c c t t t t c c c c c c c t c t t t c t c c t t c t t c c t c c c t
[39] c t t t t c
Levels: c t
```

Now we need to calculate the means for each shuffled group:

```
R> themeans=aggregate(score~mysuffle,data=kids,mean)
R> themeans

  mysuffle  score
1         c 47.13043
2         t 45.33333
```

and then we need the difference between them. This means taking the second thing in the second column of `themeans` and taking away the first thing in the second column:

```
R> meandiff=themeans[2,2]-themeans[1,2]
R> meandiff

[1] -1.797101
```

This time, the mean score for our simulated treatment group is a little smaller than for the simulated control group. This is the kind of thing we'd expect since in our simulated world there is no difference between treatment and control.

All right, let's do it again:

```
R> myshuffle=sample(group,length(group))
R> themeans=aggregate(score~mysuffle,data=kids,mean)
```

⁹Rather like Bob the Builder.

¹⁰The same is no doubt true for SAS as well, but the kinds of solutions I've seen have always looked terribly complicated.

```
R> meandiff=themeans[2,2]-themeans[1,2]
R> meandiff

[1] -1.797101
```

The difference in means is exactly the same as before! That's very odd, but it seems to be correct. One more time:

```
R> myshuffle=sample(group,length(group))
R> themeans=aggregate(score~mysuffle,data=kids,mean)
R> meandiff=themeans[2,2]-themeans[1,2]
R> meandiff

[1] -0.7950311
```

This time the difference in means is still negative, but closer to zero. From this rather meagre information, it looks as if a difference in means as big as 10 is unlikely to happen by chance. Can we do this, say, 1000 times? Sure. But it requires a little organization.

There are two concepts that we need to use here. One is that we need what programmers call a **loop**: we need to repeat something (here the randomization) over and over. Also, we need to save the mean difference from each randomization, so that we can do something with them at the end. The code below does all that. Let me show it to you first, and then explain what it does.

```
R> nsim=1000
R> ans=numeric(nsim)
R> for (i in 1:nsim)
R> {
R>   myshuffle=sample(group,length(group))
R>   themeans=aggregate(score~mysuffle,data=kids,mean)
R>   ans[i]=themeans[2,2]-themeans[1,2]
R> }
```

First, I decided to do 1000 randomizations. I'm saving that in a variable `nsim` so that if I change my mind about it later, I only have to change one thing. Next, I set up a place to hold my randomized mean-differences. This code says "make a vector of 1000 items, empty, ready to be filled with numbers."¹¹

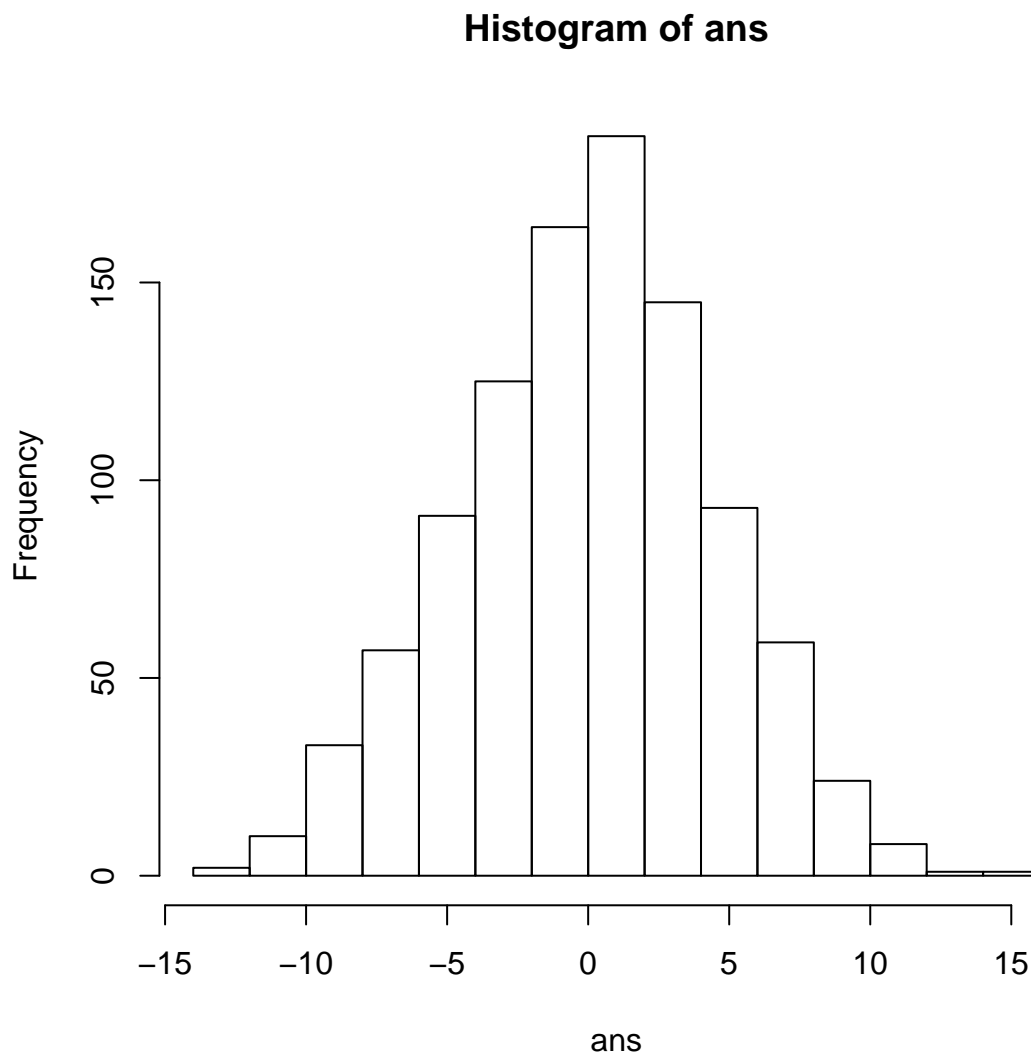
Next comes the actual loop, which is really what we had before: first, randomly shuffle the `c` and `t` labels, then calculate the means of the shuffled groups, and then calculate the difference in means between the two groups, treatment minus control. The difference this time is that instead of printing out the mean, I save it in a variable. Doing a loop like this means that the first time, $i = 1$; the second time, $i = 2, \dots$, the last time, $i = 1000$. Sometimes, you're just using the variable `i` to count the number of times you're repeating something; sometimes, as here, you actually use `i` for something. In this case, `ans[i]` means "the i th

¹¹When you write a loop in R, you will often initialize a variable to "empty" first

entry in the vector `ans`", so that the results of each randomization are stored in their own place.

This means that if you take a look at the vector `ans`, you can see what kind of differences in means are likely by chance. If we look at a histogram of them:

```
R> hist(ans)
```



we see that a difference in means bigger than 10 is possible by chance, but not very likely. How many times did our randomizations produce a difference of 10

or bigger? We are selecting the values in `ans` that are 10 or bigger, and then counting them, so something like this will work:

```
R> isbig=(ans>=10);
R> table(isbig)
```

```
isbig
FALSE  TRUE
  990    10
```

The first line we know about. Either each value of `ans` is 10 or bigger or it is not, so we have a logical vector full of `TRUE` (it is 10 or bigger) and `FALSE` (it is not). The `table` command takes a logical or categorical variable and counts up how many of each value you have.¹² In our case, only 10 out of 1000, a fraction 0.010, of randomized mean-differences came out 10 or bigger. This is our estimated P-value, and it is totally consistent with the one from the two-sample *t*-test, which was 0.013. So the randomization test and two-sample *t*-test are telling us pretty much the same thing: if the new reading program were no help, we would be pretty unlikely to see a difference in sample means as big as 10, and so we conclude that there *is* a positive effect of the new reading program.

Why did I choose to do 1000 randomizations? There's no especially good answer. I wanted to do enough so that we could get a decent picture of the kinds of sample mean-differences that we'd get by chance, but I didn't want to wait for ever for R to finish. If you think 10,000 (or a million) would be better, you just need to go back to my code, change `nsim` to the appropriate value, and run it again.

6.4 Jargon for testing

I've avoided most of the usual words that go with hypothesis testing so far.¹³ I like to think of the **alternative hypothesis** as **the thing we are trying to prove**, or the thing we are seeking evidence for. In the above example, we are trying to demonstrate that the new reading program is *better*, so we have a *one-sided* alternative hypothesis. My habit is to think about the alternative hypothesis first, and then fit the null hypothesis to that. In our example, the logical null hypothesis is one that takes up all the other possibilities in comparing the means. Here that would be that the new reading program is *equally good or worse*. This is probably not what you've seen before; you'd be expecting the null hypothesis to be *the two programs are equally good*. In operation, though, it doesn't matter, because the null hypothesis needs an equals in it to give you

¹²For a categorical variable, you could use `table` as the first step in drawing a bar chart: first you have to count how many of each category you have, and then you draw a chart with the bars that high.

¹³I wanted to see how I could get along without them.

a value to work from.¹⁴

The logic of the hypothesis testing procedure is this (which always seems upside-down):

- Work out what would happen *if the null hypothesis were true*.
- Compare what actually did happen to what would happen if the null hypothesis were true.
- If these are too far apart, conclude that the null hypothesis wasn't true after all.

To aid in that decision, we calculate the P-value as *the probability of a result as extreme or more extreme than the one observed, if the null hypothesis is true*. If this is *small*, we don't want to believe the null hypothesis (and this is where $\alpha = 0.05$ comes in).

There is nothing here about what happens if the null hypothesis *is not* true. That's all tied up with power and type II errors. I like to summarize those in a table:

Truth	Decision	
	Do not reject	Reject null
Null true	Correct	Type I error
Null false	Type II error	Correct

The tension here is between the truth, which you don't know, and the imperfect decision you have to make about the truth, which might be wrong. We normally set up a test to control the probability of making a type I error: this is the α which is commonly 0.05. Unfortunately, then we have no idea about what the probability of a type II error, denoted β , is. In fact, we need some more information about our unknown parameter (mean, say) before we can say anything about that.

6.5 Power

Suppose that our null hypothesis is $H_0 : \theta = 10$,¹⁵ and that it is wrong. Let's say our alternative hypothesis is $H_a : \theta \neq 10$. What does that tell about θ ? Not much. We could have $\theta = 11$ or $\theta = 8$ or $\theta = 496$, and in all of those cases the null would be wrong.

So in order to calculate the probability β of a type II error, or, more optimistically, $1 - \beta$, which is the probability of *not* making a type II error, called the **power**, we need to know what the unknown parameter actually is. Why does this matter? Well, it should be easier to declare the null hypothesis wrong if

¹⁴Or, in the case of the randomization test, to give you a basis for doing the randomizations from.

¹⁵ θ is generic notation for "some parameter, I don't care what it is".

it's *very wrong*, even if you don't have much data. For example, if $\theta = 496$, you shouldn't need much data to reject $H_0 : \theta = 10$. But if $\theta = 11$ in actual fact, you might have a hard time rejecting $H_0 : \theta = 10$, even though that's wrong, because it's "not very wrong". This might still be true even with a big sample.

All of this is saying that the power, the chance of *not* making a type II error, depends on two things: the actual value of the parameter, and how big a sample size you have.

The time to figure out power is before you collect any data. You might be thinking "well, I don't know *anything* about my parameter before I collect any data. How am I supposed to supply a value for it?" Usually, you have some sense of what kind of difference from the null hypothesis is important to you. For the reading programs example, you might say that a reading program that can increase the mean test score by 5 points is worth knowing about. (This is a subject-matter decision, not really a statistical one.) Then you can say to yourself "I have a certain sample size in mind; how much power do I have?", or "I want to get this much power; how big a sample size do I need?"

Both R and SAS have built-in stuff for calculating power, at least for the various flavours of t -test. Let's revisit our reading-programs example and think about how we might have planned things. We'll also need to specify which kind of t -test we are using (a two-sample one, not one-sample or matched pairs), what kind of alternative hypothesis we have (a one-sided one), and what the population standard deviation is. This last is usually unknown, so you'll have to make a guess. R first.

The code below says: for 22 children in each group (which is very close to what we had), a population standard deviation of 15 (our sample SDs were 11 and 17, so this seems a fair guess), test as we had, and wanting to detect a difference of 5 points.

```
R> power.t.test(n=22,delta=5,sd=15,type="two.sample",alternative="one.sided")

Two-sample t test power calculation

      n = 22
  delta = 5
     sd = 15
sig.level = 0.05
   power = 0.2887273
alternative = one.sided
```

NOTE: n is number in **each** group

The power is a rather disappointing 29%. This says that in only 29% of hypothetical repetitions of this experiment would we correctly reject the null hypothesis of the two programs being equally good, in favour of the alternative

that the new one is better.

If the new program is really an average of 10 points better than the old one, we should have an easier time concluding that it *is* better, larger differences being easier to detect:

```
R> power.t.test(n=22,delta=10,sd=15,type="two.sample",alternative="one.sided")
```

Two-sample t test power calculation

```
      n = 22
    delta = 10
      sd = 15
sig.level = 0.05
  power = 0.7020779
alternative = one.sided
```

NOTE: n is number in *each* group

Now, the power has gone all the way up to 70%, which is much better. Likewise, if we increase the sample size to, say, 40 children in each group, we should have a better chance of detecting a 5-point difference:

```
R> power.t.test(n=40,delta=5,sd=15,type="two.sample",alternative="one.sided")
```

Two-sample t test power calculation

```
      n = 40
    delta = 5
      sd = 15
sig.level = 0.05
  power = 0.4336523
alternative = one.sided
```

NOTE: n is number in *each* group

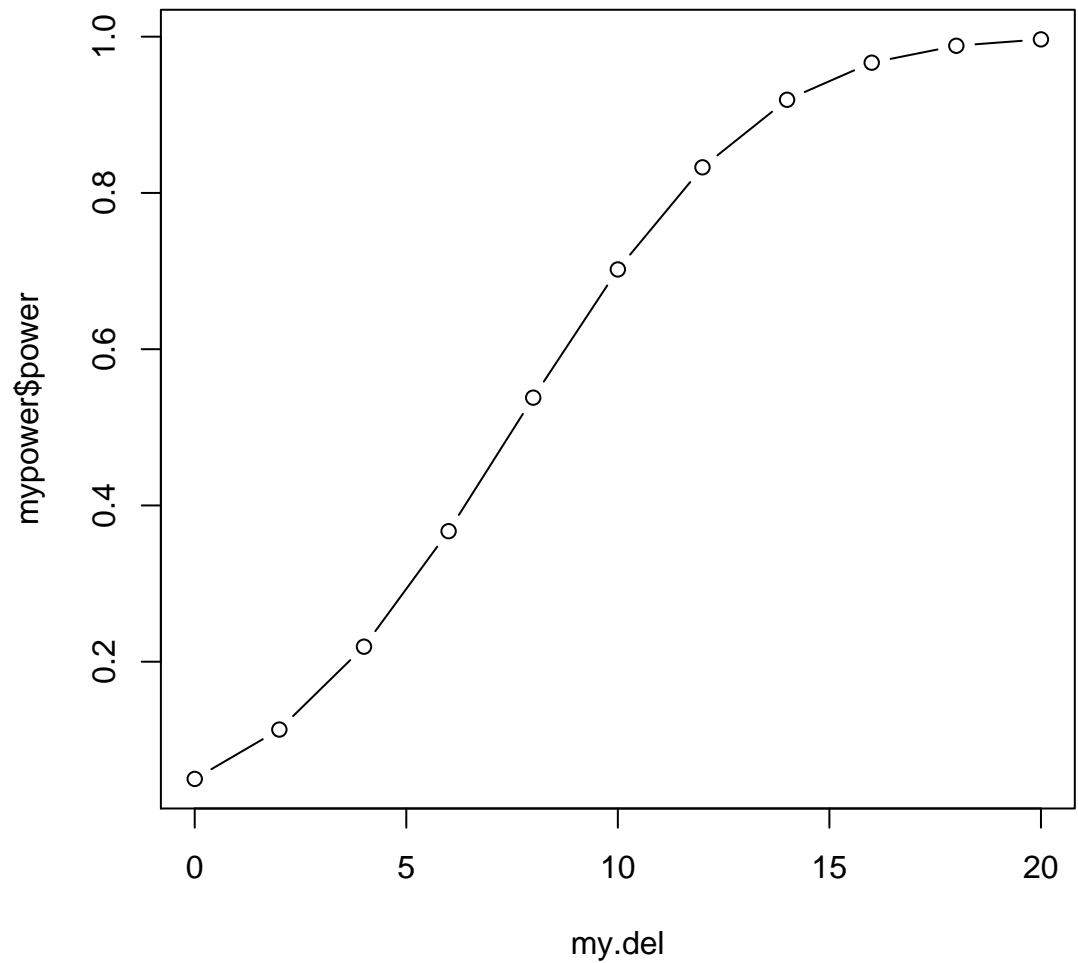
The power has gone up from 29% to 43%. As is often the case, increasing the sample size by a little has a discouragingly small effect on the results.

A nice way of graphing the results of a power analysis is to make a “power curve”. There are a couple of ways to do this: you can keep the sample size fixed and use a variety of values for the true difference between means, or you can keep the difference in means fixed and try a variety of sample sizes. R makes this easy by allowing you to specify a vector rather than a single number for the mean difference `delta`, and then you get a vector of power values back, which you can plot:

```
R> my.del=seq(0,20,2)
```

```
R> mypower=power.t.test(n=22,delta=my.del,sd=15,type="two.sample",alternative="one.sided")
```

```
R> plot(my.del,mypower$power,type="b")
```



Let's take you through the code. First we create a vector of possible mean differences, from 0 to 20 in steps of 2. Then we feed that into `power.t.test` in place of the single number for `delta` that we had before, saving the results in a variable `mypower`. Then we plot the power values calculated by R against our mean differences, plotting both points and lines.

The plot shows how the power goes up as the difference between the means goes up. This plot is for two samples of size 22. The plot shows only about a 30% chance of detecting a difference of size 5, about a 70% chance of detecting a

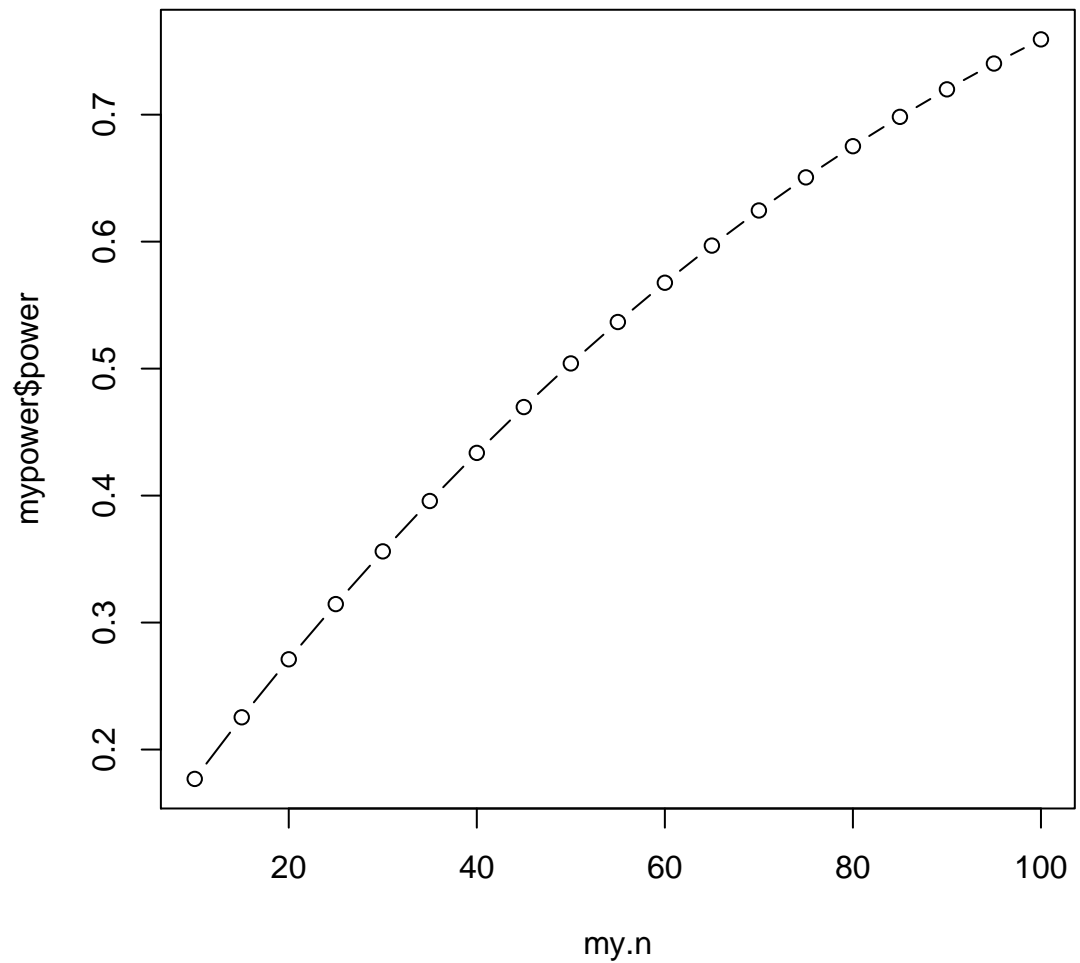
difference of size 10, and a near-certain chance of detecting a difference of size 20.

Now, let's suppose that we are only interested in a difference of 5 between the means, but we want to know about the dependence on sample sizes. This time, we make a vector of sample sizes, but we leave **delta** as 5:

```
R> my.n=seq(10,100,5)
R> my.n
```

```
[1] 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
```

```
R> mypower=power.t.test(n=my.n,delta=5,sd=15,type="two.sample",alternative="one.sided")
R> plot(my.n,mypower$power,type="b")
```



This time, the power goes more or less steadily up as the sample size increases, but even with 100 children in each group, the power is still only a bit over 70%. So you might ask, “how big a sample size do I need to get 80% power?” You can answer this by leaving out the `n=` in `power.t.test` and putting `power=` instead:

```
R> power.t.test(power=0.80,delta=5,sd=15,type="two.sample",alternative="one.sided")
Two-sample t test power calculation
```

```

        n = 111.9686
    delta = 5
        sd = 15
    sig.level = 0.05
        power = 0.8
    alternative = one.sided

```

NOTE: n is number in **each** group

We need 112 children in each group. Sample size calculations are apt to give you depressingly large answers.

Let's see whether we can reproduce this in SAS. SAS has `proc power` which does this kind of stuff. We're doing a two-sample *t*-test for means, so we need `twosamplemeans`. The default is a one-sample test, but we have a two-sample test, so we need to say that. The other default is a pooled test, but we're using the Satterthwaite test, so we have to say that. Then, as with R, we have to specify the difference in means that we're looking for, the group standard deviations (both 15) and the *total* sample size (SAS wants the number of children *altogether*). Finally, we want SAS to find the power in this situation, so we put `power=.`. The dot is SAS's code for "missing", so the implication is "I've given you everything else, find this." Note the presence and absence of semicolons; in SAS terms, the whole thing from `twosamplemeans` right down to `power=.` is *one* statement, so there are no intervening semicolons. It makes sense for ease of reading to put it on several lines, though.

```

SAS>   proc power;
SAS>   twosamplemeans
SAS>       test=diff_satt
SAS>       sides=1
SAS>       meandiff=5
SAS>       stddev=15
SAS>       ntotal=44
SAS>       power=.;
SAS>
SAS>   run;

```

The POWER Procedure

Two-Sample t Test for Mean Difference with Unequal Variances

Fixed Scenario Elements

Distribution	Normal
Method	Exact
Number of Sides	1
Mean Difference	5
Standard Deviation	15
Total Sample Size	44
Null Difference	0
Nominal Alpha	0.05

```

Group 1 Weight          1
Group 2 Weight          1

```

```

Computed Power
Actual
  Alpha    Power
0.0498    0.288

```

The power is again 29%.

SAS allows you to put several values on one line, and it does all possible combinations of the values you supply. Let's look for the effects of desired mean difference being 10 rather than 5, and also at the sample size being 40 in each group (total 80) as well as 22 in each group (total 44):

```

SAS> proc power;
SAS> twosamplemeans
SAS> test=diff_satt
SAS> sides=1
SAS> meandiff=5 10
SAS> stddev=15
SAS> ntotal=44 80
SAS> power=.;
SAS>
SAS> run;

```

The POWER Procedure

Two-Sample t Test for Mean Difference with Unequal Variances

Fixed Scenario Elements

```

Distribution          Normal
Method                Exact
Number of Sides       1
Standard Deviation    15
Null Difference        0
Nominal Alpha         0.05
Group 1 Weight         1
Group 2 Weight         1

```

Computed Power				
Index	Mean Diff	N Total	Actual Alpha	Power
1	5	44	0.0498	0.288
2	5	80	0.0499	0.433
3	10	44	0.0498	0.701
4	10	80	0.0499	0.905

This time, we get four lines of output, since there are four combinations of mean difference and sample size. Upping the difference in means ups the power quite

a bit, but upping the sample sizes doesn't help quite so much. These results are identical with R's.

Similarly, if you leave `ntotal` blank and fill in the power, you get the sample size required to achieve a given power:

```
SAS> proc power;
SAS>   twosamplemeans
SAS>     test=diff_satt
SAS>     sides=1
SAS>     meandiff=5
SAS>     stddev=15
SAS>     ntotal=.
SAS>     power=0.80;
SAS>
SAS> run;
```

The POWER Procedure

Two-Sample t Test for Mean Difference with Unequal Variances

Fixed Scenario Elements

Distribution	Normal
Method	Exact
Number of Sides	1
Mean Difference	5
Standard Deviation	15
Nominal Power	0.8
Null Difference	0
Nominal Alpha	0.05
Group 1 Weight	1
Group 2 Weight	1

Computed N Total

Actual	Actual	N
Alpha	Power	Total
0.05	0.800	224

This, because of the way SAS does these things, is 224 children *altogether*, or 112 in each group, which is the same answer that R got.

SAS is actually a little more flexible than R about the whole power and sample size thing. If we go back to our original data, we didn't actually have 22 observations in each group, we had 21 treatment and 23 control. Also, the group SDs weren't both 15, they were 11 for the treatment group and 17 for the control group.¹⁶ Under those conditions, the power for detecting an increase of 5 in mean score might be slightly different. SAS lets us calculate what it is, like this:

¹⁶At least, those were the *sample* SDs.

```

SAS> proc power;
SAS> twosamplemeans
SAS> test=diff_satt
SAS> sides=1
SAS> meandiff=5
SAS> groupstddevs=11|17
SAS> groupns=21|23
SAS> power=.;
SAS>
SAS> run;

```

The changes are that we have to use `groupstddevs` to specify the two group SDs, and `groupns` to specify the two group sample sizes. After a little experimentation, I found that I needed the vertical bars between the numbers.¹⁷

The POWER Procedure

Two-Sample t Test for Mean Difference with Unequal Variances

Fixed Scenario Elements

Distribution	Normal
Method	Exact
Number of Sides	1
Mean Difference	5
Group 1 Standard Deviation	11
Group 2 Standard Deviation	17
Group 1 Sample Size	21
Group 2 Sample Size	23
Null Difference	0
Nominal Alpha	0.05

Computed Power

Actual

Alpha	Power
0.0499	0.309

This time, the power is calculated to be 31%, a tiny increase over the 29% we got first. Having unequal sample sizes will *decrease* the power in general, which is why it's a good thing to have equal sample sizes, but this has been more than counterbalanced by one group having a smaller SD. If the groups had had equal SDs (let's go back to our value 15), having unequal sample sizes should decrease the power. I'm using very unequal sample sizes here to exaggerate the effect:

```

SAS> proc power;
SAS> twosamplemeans
SAS> test=diff_satt

```

¹⁷The logic seems to be that if you put vertical bars between, SAS applies the values one at a time to groups, whereas if you don't, SAS does all possible combinations of the values.

```

SAS>    sides=1
SAS>    meandiff=5
SAS>    stddev=15
SAS>    groupns=10/34
SAS>    power=.;
SAS>
SAS>    run;

```

The POWER Procedure

Two-Sample t Test for Mean Difference with Unequal Variances

Fixed Scenario Elements

Distribution	Normal
Method	Exact
Number of Sides	1
Mean Difference	5
Standard Deviation	15
Group 1 Sample Size	10
Group 2 Sample Size	34
Null Difference	0
Nominal Alpha	0.05

Computed Power

Actual

Alpha	Power
0.0505	0.225

The power is 27% instead of 29%. Actually, not much of a drop, considering how unbalanced I made the sample sizes.

6.6 Confidence intervals

A hypothesis test answers the question “could my parameter be *this*”, where you supply the value for “this”. In the case of our two-sample *t*-test, “this” was the null-hypothesis value of zero, and from our data we concluded that “this” was not zero, and that the new reading program *was* effective.

The flip side of that question is to ask “how much difference in mean score could there be in the reading programs?”. In general, we are asking “what could my parameter be?”. There is going to be uncertainty involved in any answer to this question, because our samples are going to give an imperfect picture of the populations from which they come. The right way to handle this is to create an interval, a **confidence interval**, that says “I think the parameter is between this and that”.

As with a test, there is no way to do things perfectly; we can never be *certain* that the true population value of the parameter lies within the interval that we

give.¹⁸ So we have to make a compromise. We can do that by using, say, a 95% confidence interval, where in 95% of all possible samples, the procedure of making the confidence interval will make a true statement about the thing we're estimating. We exchange the 5% chance of being wrong for getting an interval that is actually helpful.

Confidence intervals are inherently two-sided things, so when we're getting them, we pretend we are doing a two-sided test (the default in both SAS and R) and look carefully at our output. For the reading program data, we therefore feed this into SAS, which is exactly what we had before with the exception of `side=1`, which I removed:

```
SAS> proc ttest;
SAS>   var score;
SAS>   class group;
```

The TTEST Procedure

Variable: score

group	N	Mean	Std Dev	Std Err	Minimum	Maximum
c	23	41.5217	17.1487	3.5758	10.0000	85.0000
t	21	51.4762	11.0074	2.4020	24.0000	71.0000
Diff (1-2)		-9.9545	14.5512	4.3919		

group	Method	Mean	95% CL Mean	Std Dev	95% CL S
c		41.5217	34.1061 48.9374	17.1487	13.2627 2
t		51.4762	46.4657 56.4867	11.0074	8.4213 3
Diff (1-2)	Pooled	-9.9545	-18.8176 -1.0913	14.5512	11.9981 4
Diff (1-2)	Satterthwaite	-9.9545	-18.6759 -1.2330		
Method	Variances	DF	t Value	Pr > t	
Pooled	Equal	42	-2.27	0.0286	
Satterthwaite	Unequal	37.855	-2.31	0.0264	

Equality of Variances

Method	Num DF	Den DF	F Value	Pr > F
Folded F	22	20	2.43	0.0507

Some of the output disappeared off to the right, but the important stuff is here. Look for the line with **Satterthwaite** under **Method**. The difference in sample means is -9.95 , and a 95% confidence interval for the difference in population means is -18.7 to -1.2 . Because this is control (standard program) vs. treatment (new program), we therefore think that the new program is between 1.2 and 18.7 points better on average than the standard program. We may not know very precisely *how much* better the new program is, but we are pretty sure that it *is* better, which is what the test told us.

To get, say, a 90% confidence interval instead, you specify **alpha** on the **proc ttest** line, as one minus the confidence level, like this:

¹⁸Unless it goes down to minus infinity and up to plus infinity, which wouldn't be very helpful.

```
SAS> proc ttest alpha=0.10;
SAS>   var score;
SAS>   class group;
```

The TTEST Procedure

Variable: score

group	N	Mean	Std Dev	Std Err	Minimum	Maximum
c	23	41.5217	17.1487	3.5758	10.0000	85.0000
t	21	51.4762	11.0074	2.4020	24.0000	71.0000
Diff (1-2)		-9.9545	14.5512	4.3919		
group	Method	Mean	90% CL Mean	Std Dev	90% CL Std Dev	
c		41.5217	35.3816 47.6618	17.1487	13.8098 22.8992	
t		51.4762	47.3334 55.6190	11.0074	8.7834 14.9440	
Diff (1-2)	Pooled	-9.9545	-17.3414 -2.5675	14.5512	12.3693 17.7758	
Diff (1-2)	Satterthwaite	-9.9545	-17.2176 -2.6913			
Method	Variances	DF	t Value	Pr > t		
Pooled	Equal	42	-2.27	0.0286		
Satterthwaite	Unequal	37.855	-2.31	0.0264		
Equality of Variances						
Method	Num DF	Den DF	F Value	Pr > F		
Folded F	22	20	2.43	0.0507		

The 90% confidence interval is shorter than the 95% one.¹⁹

Likewise with R, if you ask for a two-sided test,²⁰ you'll get a (95%, by default) confidence interval:

```
R> t.test(score~group,data=kids)

Welch Two Sample t-test
```

```
data: score by group
t = -2.3109, df = 37.855, p-value = 0.02638
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -18.67588 -1.23302
sample estimates:
mean in group c mean in group t
 41.52174      51.47619
```

which is the same as SAS got, and if you want something other than 95%, you ask for it via `conf.level`, which actually *is* the confidence level you want:²¹

```
R> t.test(score~group,data=kids,conf.level=0.90)

Welch Two Sample t-test
```

¹⁹You know why this is, don't you?

²⁰Which is done by *not* asking for a one-sided test.

²¹Unlike SAS.

```
data:  score by group
t = -2.3109, df = 37.855, p-value = 0.02638
alternative hypothesis: true difference in means is not equal to 0
90 percent confidence interval:
  -17.217609  -2.691293
sample estimates:
mean in group c mean in group t
      41.52174      51.47619
```

There's not a great deal else to say about confidence intervals as such. The other thing that gets shown in previous courses is that when you have a larger sample size, the confidence interval should get shorter,²² since your sample will be a better representation of your population.

6.7 The duality between confidence intervals and hypothesis tests

“Duality” is a fancy mathematical way of saying that two different things are really the same thing, if you look at them the right way. Confidence intervals and hypothesis tests are like that. At least if you do two-sided tests.²³

Let's illustrate with some data. I'll pursue the two-sample thing. I'll arrange the data differently, just for fun:²⁴

```
R> group1=c(10,11,11,13,13,14,14,15,16)
R> group2=c(13,13,14,17,18,19)
R> t.test(group1,group2)
```

Welch Two Sample t-test

```
data:  group1 and group2
t = -2.0937, df = 8.71, p-value = 0.0668
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  -5.5625675  0.2292342
sample estimates:
mean of x mean of y
  13.00000  15.66667
```

If you have two separate columns of data, as we do here, you just feed them one after the other into `t.test`, and you don't use a model formula.

²²With the *t* test, this is not absolutely true, since a larger sample could come with a sufficiently larger sample SD to make the confidence interval *longer*, but this will be rare.

²³Since confidence intervals are two-sided things.

²⁴Note that the two groups don't have to be the same size.

6.7. THE DUALITY BETWEEN CONFIDENCE INTERVALS AND HYPOTHESIS TESTS 95

The P-value of 0.0668 is not quite smaller than 0.05, so we cannot quite reject a null that there is no difference in means. This is echoed by the (95%, by default) confidence interval, which contains some positive and some negative values for the difference in means. These are both saying “no evidence of a difference”.

Now, suppose we had used $\alpha = 0.10$. Then we *would* have concluded that there was a difference. $\alpha = 0.10$ goes with a 90% confidence interval. What does that look like?

```
R> t.test(group1,group2,conf.level=0.90)

Welch Two Sample t-test

data: group1 and group2
t = -2.0937, df = 8.71, p-value = 0.0668
alternative hypothesis: true difference in means is not equal to 0
90 percent confidence interval:
 -5.010308 -0.323025
sample estimates:
mean of x mean of y
 13.00000  15.66667
```

This time, the confidence interval for the difference in means only includes negative values. This is consistent with the $\alpha = 0.10$ conclusion from the test: there is a difference between the groups because we think the group 1 mean is less than the group 2 mean.

Now, let's see if we can do the same thing in SAS, which involves a bit of cleverness at the `data` step. Here's now I laid out the data:

```
10 13
11 13
11 14
13 17
13 18
14 19
14 .
15 .
16 .
```

The first sample is in the first column, and the second sample in the second. The samples in a two-sample *t*-test don't have to be the same size, and here they're not, so I have to supply some missing values to make the columns the same length. In SAS, a dot is a missing value.

Now, we need to think about strategy. In SAS, the *only* way to do a two-sample *t*-test is with one column of values and one column of groups. So we have to make that happen here.

There are some SAS tricks I'm going to use here:

1. The @@ thing continues to read values on the same line.
2. The special variable `_N_` says how many "lines" SAS has read so far.
3. The function `mod(x,y)` calculates the remainder when `x` is divided by `y`.

Let me illustrate `_N_` first, by reading in these data the way you'd expect to read them.²⁵ Recall that you define new variables in terms of other things in the `data` step:

```
SAS> data xy;
SAS>   infile 'twocols.txt';
SAS>   input x y;
SAS>   line=_n_;
SAS>
SAS> proc print;
```

Obs	x	y	line
1	10	13	1
2	11	13	2
3	11	14	3
4	13	17	4
5	13	18	5
6	14	19	6
7	14	.	7
8	15	.	8
9	16	.	9

See how `line` goes from 1 to 9?

SAS's `proc ttest` requires one variable of data and one variable of groups. We have our data, but it's in *two* columns, and we don't have groups at all. I had an idea how to do this.²⁶ Read in the data values *one* at a time, and then find a way to associate them with the right groups: the first one with `x`, the second one with `y`, and so on. Let me show you something. The `mod` function gives you the *remainder* when the first number is divided by the second:

```
SAS> data xonly;
SAS>   infile 'twocols.txt';
SAS>   input x @@;
SAS>   line=_n_;
SAS>   g=mod(line,2);
SAS>
SAS> proc print;
```

²⁵SAS is not case sensitive, so `_N_` and `_n_` are the same.

²⁶On the #38 bus, somewhere around Neilson.

6.7. THE DUALITY BETWEEN CONFIDENCE INTERVALS AND HYPOTHESIS TESTS97

```
Obs      x      line      g
  1      10         1      1
  2      13         2      0
  3      11         3      1
  4      13         4      0
  5      11         5      1
  6      14         6      0
  7      13         7      1
  8      17         8      0
  9      13         9      1
 10      18        10      0
 11      14        11      1
 12      19        12      0
 13      14        13      1
 14      .         14      0
 15      15        15      1
 16      .         16      0
 17      16        17      1
 18      .         18      0
```

This time `line` gets upped by one each time a *single* value is read, since we are only reading one value at a time. So `g` alternates between 1 and 0. You can check that `g` is 1 when an observation from the first column is read, and 0 when an observation from the second column is read. That is, `g` successfully distinguishes the groups, and we can use this as our grouping variable for `proc ttest`.

Let's check that the group means are correct:

```
SAS> proc means;
SAS>   var x;
SAS>   class g;
```

The MEANS Procedure

Analysis Variable : x							
	g	Obs	N	Mean	Std Dev	Minimum	Maximum
	0	9	6	15.6666667	2.6583203	13.0000000	19.0000000
	1	9	9	13.0000000	2.0000000	10.0000000	16.0000000

According to R, the first group had mean 13 and the second had mean 15.667. Apart from the fact that the groups have gotten flipped around, this is what we have here.

So now we can compare with R by running `proc ttest` in the normal way. We want a test with P-value, a 90% and a 95% confidence interval for the difference in means. This seems to mean running `proc ttest` twice.

```
SAS> proc ttest;
SAS>   var x;
SAS>   class g;
```

The TTEST Procedure

Variable: x

g	N	Mean	Std Dev	Std Err	Minimum	Maximum
0	6	15.6667	2.6583	1.0853	13.0000	19.0000
1	9	13.0000	2.0000	0.6667	10.0000	16.0000

Diff (1-2)	2.6667	2.2758	1.1995
------------	--------	--------	--------

g	Method	Mean	95% CL Mean	Std Dev	95% CL S
0		15.6667	12.8769 18.4564	2.6583	1.6593
1		13.0000	11.4627 14.5373	2.0000	1.3509
Diff (1-2)	Pooled	2.6667	0.0754 5.2580	2.2758	1.6499
Diff (1-2)	Satterthwaite	2.6667	-0.2292 5.5626		

Method	Variances	DF	t Value	Pr > t
Pooled	Equal	13	2.22	0.0446
Satterthwaite	Unequal	8.7104	2.09	0.0668

Equality of Variances

Method	Num DF	Den DF	F Value	Pr > F
Folded F	5	8	1.77	0.4518

We are doing the unequal-variances Satterthwaite test, so the P-value we want is 0.0668 (same as R). Look on the Satterthwaite line for the confidence interval, -0.2292 to 5.5626 . Apart from bring the other way around, this is the same as R gave us (because we now have the groups the other way around). The point is that 0 is *inside* this interval, so the groups are *not* significantly different at the α that goes with a 95% confidence interval, ie. $\alpha = 0.05$.

Let's get a 90% confidence interval. This time, the corresponding α for statistical significance is 0.10, and our P-value is less than that, so the means are significantly different at that level. Therefore, we would guess that 0 should be *outside* this interval. Are we right?

```
SAS> proc ttest alpha=0.10;
SAS>   var x;
SAS>   class g;
```

The TTEST Procedure

Variable: x

g	N	Mean	Std Dev	Std Err	Minimum	Maximum
0	6	15.6667	2.6583	1.0853	13.0000	19.0000
1	9	13.0000	2.0000	0.6667	10.0000	16.0000

Diff (1-2)	2.6667	2.2758	1.1995
------------	--------	--------	--------

g	Method	Mean	90% CL Mean	Std Dev	90% CL S
0		15.6667	13.4798 17.8535	2.6583	1.7865
1		13.0000	11.7603 14.2397	2.0000	1.4365
Diff (1-2)	Pooled	2.6667	0.5425 4.7909	2.2758	1.7352

Diff (1-2)	Satterthwaite	2.6667	0.3230	5.0103
Method	Variances	DF	t Value	Pr > t
Pooled	Equal	13	2.22	0.0446
Satterthwaite	Unequal	8.7104	2.09	0.0668
Equality of Variances				
Method	Num DF	Den DF	F Value	Pr > F
Folded F	5	8	1.77	0.4518

This time, the confidence interval goes from 0.3230 to 5.0103. Zero is indeed *not* in this interval.

This is the same interval that R got (with the groups switched around). This is comforting news.

The correspondence between test and confidence interval works for *any*²⁷ test. For example, the SAS output also gives the P-value for the pooled *t*-test²⁸, which is 0.0446. This is less than 0.05, so the 95% *pooled* confidence interval should *not* include zero. It goes from 0.0754 to 5.2580, which does not include zero as we guessed.

Here's a summary of the logic we just exemplified, which works both ways (it is an "if and only if"):

- If the P-value is less than 0.05, then the 95% confidence interval will *not* contain the null-hypothesis parameter value.
- If the 95% confidence interval does not contain the null-hypothesis parameter value, then the P-value for the test of that null hypothesis is less than 0.05.
- If the P-value is greater than 0.05, the 95% confidence interval *will* contain the null-hypothesis parameter value.
item If the 95% confidence interval does contain the null-hypothesis parameter value, then the P-value for the test of that null hypothesis is greater than 0.05.

All of this is still true if you change α to something else, and change the confidence level accordingly, as long as you are doing a two-sided test.

²⁷Well, with the exception of the usual one-sample test and confidence interval for the proportion. The test statistic is

$$z = \frac{\hat{p} - p_0}{\sqrt{p_0(1 - p_0)/n}},$$

which has the null hypothesis proportion p_0 in all over the place, while the confidence interval cannot use p_0 for the standard error, since there *is* no p_0 , so it is

$$\hat{p} \pm z^* \sqrt{\hat{p}(1 - \hat{p})/n}.$$

This makes a difference, but not much, in practice.

²⁸Don't worry if you don't know what that is.

This kind of thinking allows you to get a confidence interval from any test. The thought process is you figure out what values of your parameter you would *not* reject for, at the appropriate α , and those make your confidence interval.

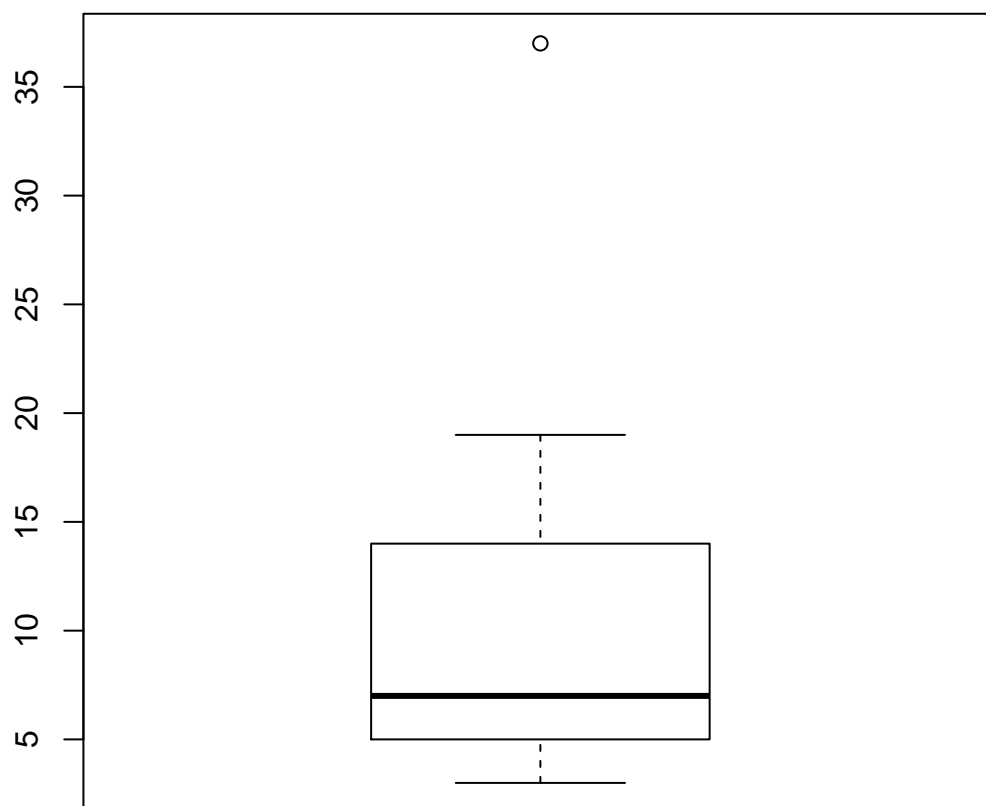
6.8 Sign test

I want to illustrate the idea of getting a confidence interval by “inverting” a test, but before I do that, I want to introduce you to a test for the population *median*, called the **sign test**.

Imagine you have some data like this, 10 values:

```
R> x=c(3, 4, 5, 5, 6, 8, 11, 14, 19, 37)
R> x
R> boxplot(x)
```

```
[1] 3 4 5 5 6 8 11 14 19 37
```



What can we say about the centre of these data? Probably the first thing we should do is to look at the picture; these data are seriously skewed to the right. So the mean is not going to be the measure of centre that we want. We should use the *median*. But how do we do inference for the median? For example, can we test that the median is equal to 12, against the alternative that it is not?

If the population median really is 12, then each value in the sample will be either

above or below 12, with probability 0.5 of either.²⁹ So our test statistic can be “the number of sample values above 12.”³⁰ Now, how do we get a P-value from that?

The definition of a P-value is **the probability of getting a value as extreme or more extreme than the one observed, if the null hypothesis is true.** If the median really is 12, then each data value either is or is not above 12, independently of all the others, with the same 0.5 probability. So the number of values above 12 in a sample of size 10 (like ours) has a *binomial distribution* with $n = 10$ and $p = 0.5$.

R has a binomial distribution calculator. Actually, it has three: `dbinom` calculates the probability of getting an exact number of successes³¹, `pbinom` for getting the probability of “less or equal”, and `qbinom` for the inverse of that. Let’s try them out on a binomial with $n = 4$ and $p = 0.3$:

```
R> dbinom(1,4,0.3)
```

```
[1] 0.4116
```

is the probability of getting *exactly* one success in four trials with success probability 0.2.

```
R> pbinom(1,4,0.3)
```

```
[1] 0.6517
```

is the probability of getting 1 success *or fewer*, and

```
R> qbinom(0.9,4,0.3)
```

```
[1] 2
```

says that the probability is at least 0.9 of getting 2 successes or fewer. We can confirm this:

```
R> pbinom(2,4,0.3)
```

```
[1] 0.9163
```

Confirmed.

All the distributions that R knows about³² have this name `p`, `d`, `q` structure. Thus

```
R> pnorm(-1.96)
```

```
[1] 0.0249979
```

²⁹We ignore any data values that are exactly equal to 12.

³⁰Or, “below 12” if you like that better.

³¹`d` for “density”, by analogy with continuous random variables.

³²All the ones you’ve heard about and more besides.

says that about 2.5% of the standard normal distribution lies below -1.96 , as you might remember.³³

Now, let's use what we just discovered about `pbinom` to get a P-value for the sign test that we just did. We were testing that the median was 12, and our data were in a variable imaginatively called `x`:

```
R> x
[1] 3 4 5 5 6 8 11 14 19 37
R> table(x>12)
FALSE TRUE
    7     3
```

There are 3 values above 12 and 7 below. Taking the number for “above 12”, more extreme would be 3 or *less*, so we fire up `pbinom` with 3 successes, $n = 10$ and $p = 0.5$ to get a P-value of:³⁴

```
R> 2*pbinom(3,10,0.5)
[1] 0.34375
```

There is no evidence that the median differs from 12.

The test told us whether the median was 12 or not, but it didn't say much about what the population median actually *was*. To do that, we need a confidence interval, and we can get that here by “inverting” the test: asking ourselves “what values of the population median would we not reject for”. That might seem like a lot of work, and it is, but it's work for R, not for us, once we get things set up right.

I want to introduce you to the concept of an **R function**. The idea behind a function is that it packages up the details of some more or less complicated calculation, so that we can use it over and over. I'm going to write a function that takes the data and the null-hypothesis median, and gives us back the P-value for the two-sided test. There are a couple of annoying details, which I'll explain below.

```
R> sign.test=function(mydata,med)
R> {
R>   tbl=table(mydata<med)
R>   stat=min(tbl)
R>   n=length(mydata)
R>   pval=2*pbinom(stat,n,0.5)
R>   return(pval)
R> }
```

³³If you have a normal distribution with a different mean and SD, you supply them to `pnorm` next; thus `pnorm(12,10,3)` will give you the probability that a normal random variable with mean 10 and SD 3 is less than 12.

³⁴Our test was two-sided, hence the multiplication by two.

First, we gave our function a name. I used `sign.test`. The stuff in brackets on the first line says that we are going to feed in first the data, and second the null-hypothesis median. Then the actual function itself is enclosed within curly brackets.

Within the definition of the function: first, we count up how many data values are below and above the median.³⁵ Then, our life is made easier if we can make our P-value be the probability of being less than or equal to something, so we take the smaller of those two values as our test statistic. Next, we need to know how many data values we have, for the `n` of our binomial distribution. We could either feed that in as well, or work it out. The `length` of a vector tells you how many values that vector contains, which is just what we want.

The fourth line calculates the P-value just as we did above, doubling the probability because we have a two-sided test. Finally, the last line takes that P-value and sends it back to the outside world.

Let's test this on our example to see whether it works:

```
R> sign.test(x,12)
[1] 0.34375
```

Success!³⁶

All right, confidence interval for median. The idea is that we use the same data, but try different values for the null median, until we find ones that we don't reject for. We'll go upwards first, remembering not to use any values that are in the data:

```
R> sign.test(x,13)
[1] 0.34375
R> sign.test(x,15)
[1] 0.109375
R> sign.test(x,20)
[1] 0.02148437
```

For a 95% CI, 13 and 15 should be inside and 20 outside. How about the lower end?

```
R> sign.test(x,5.5)
[1] 0.7539063
R> sign.test(x,4.5)
```

³⁵I am not being careful about what to do with values exactly equal to the median, so we should only call this function with a null median different from all the data values.

³⁶If you were a proper programmer, you'd test the function on some more values, to convince yourself that you hadn't just been lucky.


```
[1] 0.109375
```

```
R> sign.test(x,3.5)
```

```
[1] 0.02148437
```

4.5 and 5.5 should be inside, and 3.5 outside, the 95% confidence interval.³⁷ So the 95% confidence interval for the population median goes from 4.5 to 15.

This is not hugely impressive, partly because we have a small sample, and partly because the sign test is not very powerful. But a confidence interval it is.

I did all the above in R, but SAS also handles the sign test. It's one of the many things hiding in `proc univariate`. Note that the way I specify the null median is `location=` on the `proc univariate` line.

```
SAS> data x;
SAS>   input x@@;
SAS>   cards;
SAS>      3 4 5 5 6 8 11 14 19 37
SAS>   ;
SAS>
SAS> proc univariate location=12;
```

The UNIVARIATE Procedure

Variable: x

		Moments	
N	10	Sum Weights	10
Mean	11.2	Sum Observations	112
Std Deviation	10.3687565	Variance	107.51111
Skewness	2.00534072	Kurtosis	4.30578759
Uncorrected SS	2222	Corrected SS	967.6
Coeff Variation	92.5781829	Std Error Mean	3.2788887

Basic Statistical Measures			
Location		Variability	
Mean	11.20000	Std Deviation	10.36876
Median	7.00000	Variance	107.51111
Mode	5.00000	Range	34.00000
		Interquartile Range	9.00000

Tests for Location: Mu0=12			
Test	-Statistic-	-----p Value-----	
Student's t	t -0.24399	Pr > t	0.8127
Sign	M -2	Pr >= M	0.3438
Signed Rank	S -9.5	Pr >= S	0.3730

³⁷Note that I used halves to sidestep the issue of what happens when the null hypothesis median is equal to one of the data values.

Quantiles (Definition 5)

Quantile	Estimate
100% Max	37.0
99%	37.0
95%	37.0
90%	28.0
75% Q3	14.0
50% Median	7.0
25% Q1	5.0
10%	3.5
5%	3.0
1%	3.0
0% Min	3.0

Extreme Observations

----Lowest----		----Highest----	
Value	Obs	Value	Obs
3	1	8	6
4	2	11	7
5	4	14	8
5	3	19	9
6	5	37	10

If you cast your eye down to Tests for Location, you'll see the sign test as the second alternative, below the one-sample t -test.³⁸ The P-value is the same as for R. There is probably a way of persuading SAS to spit out a confidence interval for the median, but I couldn't find it.

6.9 Matched pairs

Take a look at these data:

Case	Drug A	Drug B	Case	Drug A	Drug B
1	2.0	3.5	7	14.9	16.7
2	3.6	5.7	8	6.6	6.0
3	2.6	2.9	9	2.3	3.8
4	2.6	2.4	10	2.0	4.0
5	7.3	9.9	11	6.8	9.1
6	3.4	3.3	12	8.5	20.9

³⁸Which we decided we wouldn't believe.

We are comparing two drugs for effectiveness at reducing pain. The 12 subjects were arthritis sufferers, and the response variable is the number of hours of pain relief gained from each drug.

What is different about this example, as compared to the children learning to read, is that each subject tries out *both* drugs and gives us two measurements. This is possible because an arthritis sufferer can try out one drug, and, provided you wait long enough, this has no influence on how the other drug might perform.³⁹ The advantage to this is that you get a focused comparison of the drugs: you compare how well the drugs performed *on the same person*. This is called a **matched-pairs experiment**. The same idea will work if you get measurements before and after some intervention, measurements of different treatments on a person's two eyes, using different people that have been paired up to be similar on important characteristics, and so on.

There is a point of technique here: how do you decide which member of the pair gets which treatment? In our case, this would be a matter of deciding who gets drug A first. The answer is the usual one: randomize. We can toss a coin for each subject to decide A is first or B is first, or we can randomly choose 6 of them to get drug A first.

We don't have any preconceived notion of which drug might be better, so a two-sided test is in order.

First, the "standard" analysis, using the *t*-test. SAS first:

```
SAS> data analgesic;
SAS>   infile "analgesic.txt";
SAS>   input subject druga drugb;
SAS>   diff=druga-drugb;
SAS>
SAS> proc print;
SAS>
SAS> proc ttest;
SAS>   paired druga*drugb;
SAS>
SAS> proc ttest h0=0;
SAS>   var diff;
SAS>
SAS> run;
```

Obs	subject	druga	drugb	diff
1	1	2.0	3.5	-1.5
2	2	3.6	5.7	-2.1
3	3	2.6	2.9	-0.3
4	4	2.6	2.4	0.2

³⁹Compare this to the children learning to read, where, once you've learned to read, you can't learn to read again!

5	5	7.3	9.9	-2.6
6	6	3.4	3.3	0.1
7	7	14.9	16.7	-1.8
8	8	6.6	6.0	0.6
9	9	2.3	3.8	-1.5
10	10	2.0	4.0	-2.0
11	11	6.8	9.1	-2.3
12	12	8.5	20.9	-12.4

The TTEST Procedure

Difference: `druga - drugb`

N	Mean	Std Dev	Std Err	Minimum	Maximum
12	-2.1333	3.4092	0.9841	-12.4000	0.6000
Mean	95% CL Mean	Std Dev	95% CL Std Dev		
-2.1333	-4.2994 0.0327	3.4092	2.4150 5.7884		
DF	t Value	Pr > t			
11	-2.17	0.0530			

The TTEST Procedure

Variable: `diff`

N	Mean	Std Dev	Std Err	Minimum	Maximum
12	-2.1333	3.4092	0.9841	-12.4000	0.6000
Mean	95% CL Mean	Std Dev	95% CL Std Dev		
-2.1333	-4.2994 0.0327	3.4092	2.4150 5.7884		
DF	t Value	Pr > t			
11	-2.17	0.0530			

I'll explain the `diff` business and the second *t*-test in a minute.

The analysis uses `proc ttest` again, but we have to tell SAS that we have paired data. This is done by the `paired druga*drugb` line.⁴⁰ This is by default a two-sided test, which is what we want.⁴¹ The P-value, on the end of the first `proc ttest` output, is 0.0530, so we cannot quite, at $\alpha = 0.05$, conclude that there is a difference in pain relief between the two drugs.

All right, what was that business with `diff`? You might recall that the way you do this by hand is to calculate the 12 `druga` minus `drugb` differences, and then test those differences to see if the mean is zero. I'm doing that here too, to show you that I get the same results.

In SAS, the place to calculate new variables is in the `data` step. Here I have calculated `diff` as the A-minus-B difference. Then I call `proc ttest` a second time, as a one-sample test, testing those differences to see whether they have a mean of zero. The test statistic and P-value are identical. Just to convince you that the results really are the same.

⁴⁰The `*` is the same idea as if you were making a scatterplot of `druga` vs. `drugb`.

⁴¹Had you wanted a one-sided test, you would have put `side=1` or `side=u` on the `proc ttest` line as before.

Now, let's try it in R. I forgot to give the variables names in the data file, so we'll use what we have.

```
R> pain=read.table("analgesic.txt",header=F)
R> pain
```

	V1	V2	V3
1	1	2.0	3.5
2	2	3.6	5.7
3	3	2.6	2.9
4	4	2.6	2.4
5	5	7.3	9.9
6	6	3.4	3.3
7	7	14.9	16.7
8	8	6.6	6.0
9	9	2.3	3.8
10	10	2.0	4.0
11	11	6.8	9.1
12	12	8.5	20.9

```
R> attach(pain)
R> t.test(V2,V3,paired=T)
```

Paired t-test

data: V2 and V3
t = -2.1677, df = 11, p-value = 0.05299
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-4.29941513 0.03274847
sample estimates:
mean of the differences
-2.133333

V2 contained the Drug A results and V3 the Drug B. The P-value is again 0.0503.

As previously, another way to do this test is to find the differences and test whether they have mean 0:

```
R> diff=V2-V3
R> diff
```

```
[1] -1.5 -2.1 -0.3 0.2 -2.6 0.1 -1.8 0.6 -1.5 -2.0 -2.3 -12.4
```

```
R> t.test(diff,mu=0)
```

One Sample t-test

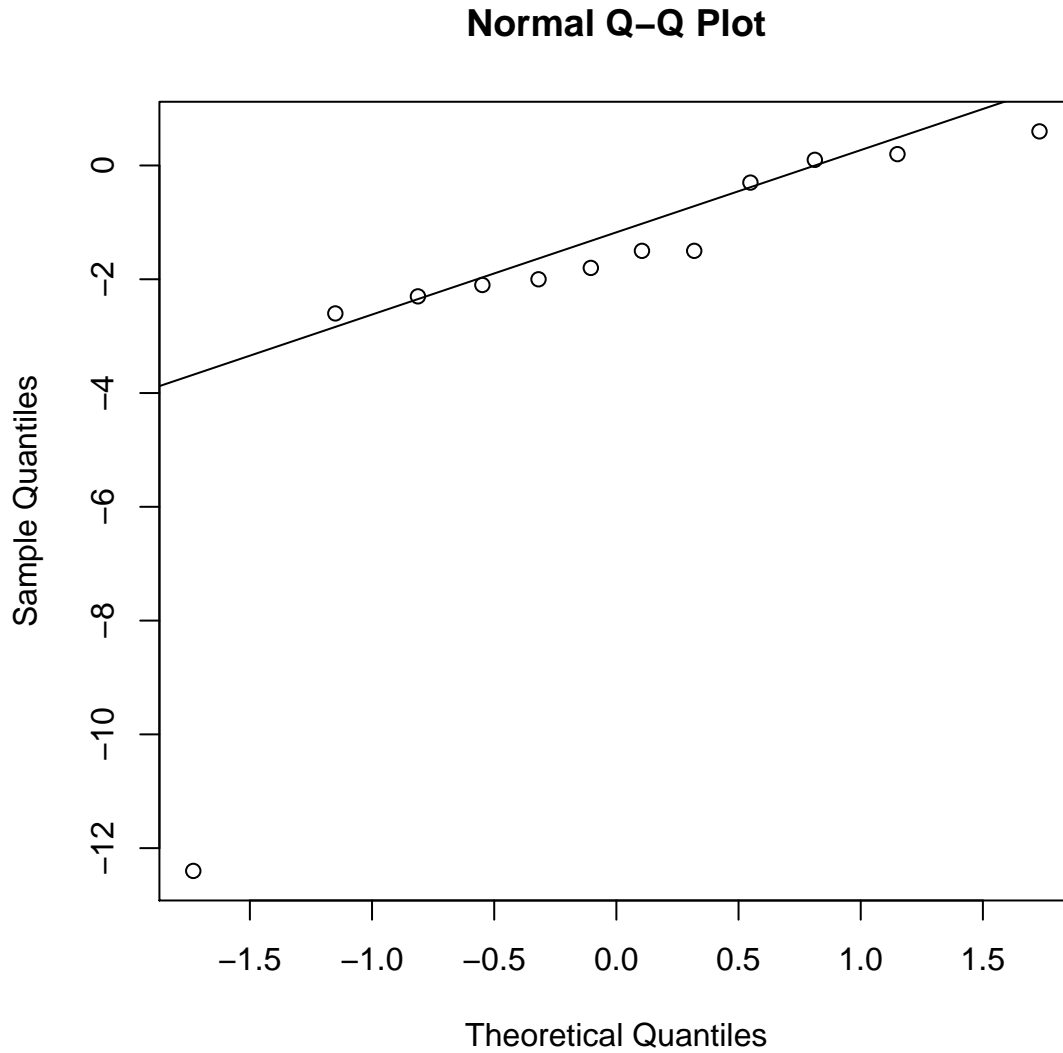
data: diff
t = -2.1677, df = 11, p-value = 0.05299

```
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 -4.29941513  0.03274847
sample estimates:
mean of x
-2.133333
```

Once again, the P-value is the same.

The analyses based on the *t*-test assume (theoretically) that the differences are normally distributed. The plot below will help us decide whether we believe that:

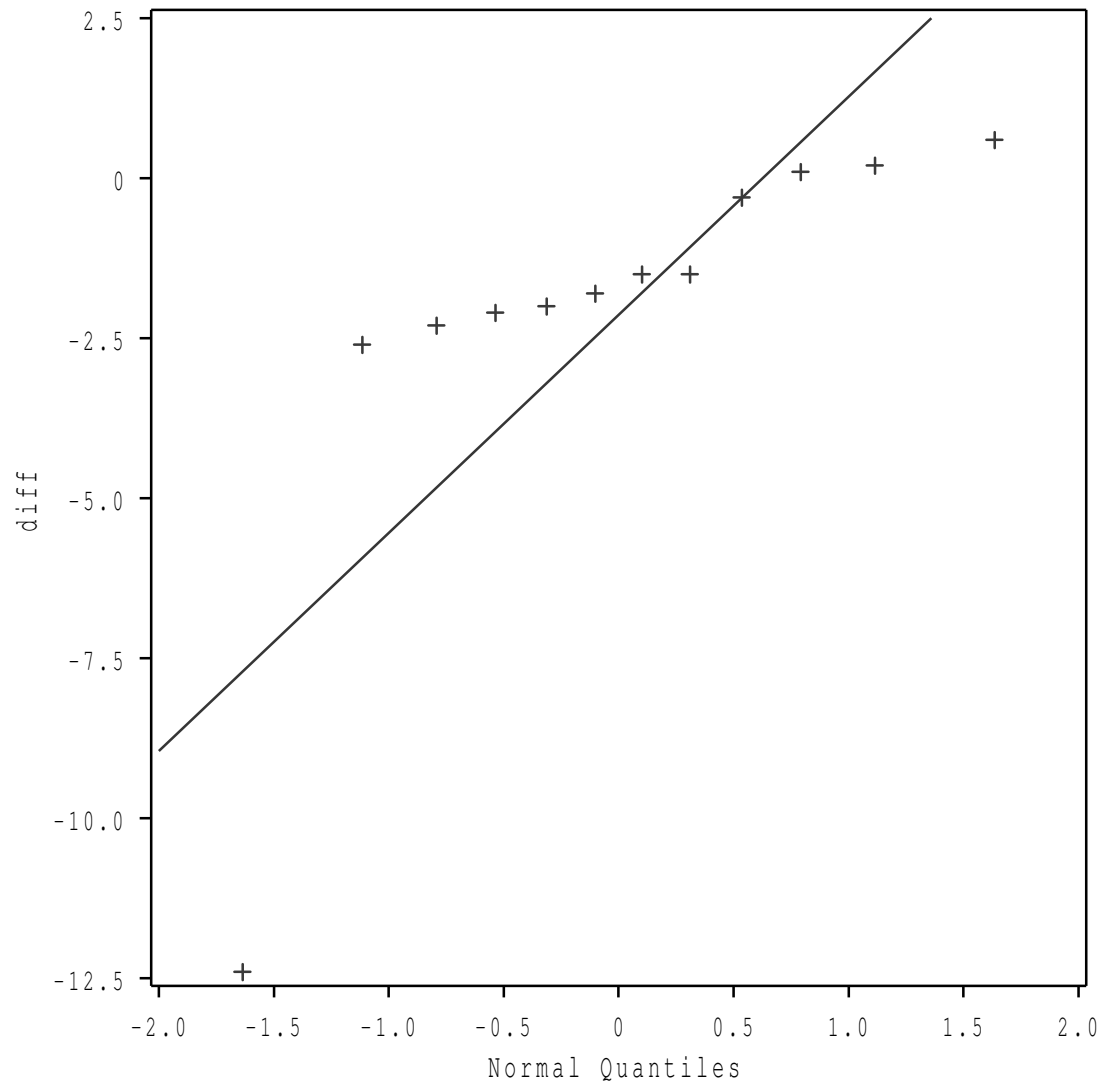
```
R> qqnorm(diff)
R> qqline(diff)
```



This is a normal quantile plot, sometimes called a normal QQ plot. The idea is to plot the observed values (y axis) against what you would expect to see if the normal distribution were correct (x axis). If the points are all close to the line, then we believe a normal distribution. Here, though, the point at the bottom left is way off the line, even though the others are all pretty close. The smallest observation is way too small to make a normal distribution believable here.

SAS also makes normal quantile plots, via `proc univariate`:

```
SAS> proc univariate noprint;
SAS> qqplot diff / normal(mu=est sigma=est);
```



Just the same idea as R.⁴² The line is different, since SAS gets it from the sample mean and SD (which are affected by the outlier), while R gets it from the quartiles (which are not).^{43 44}

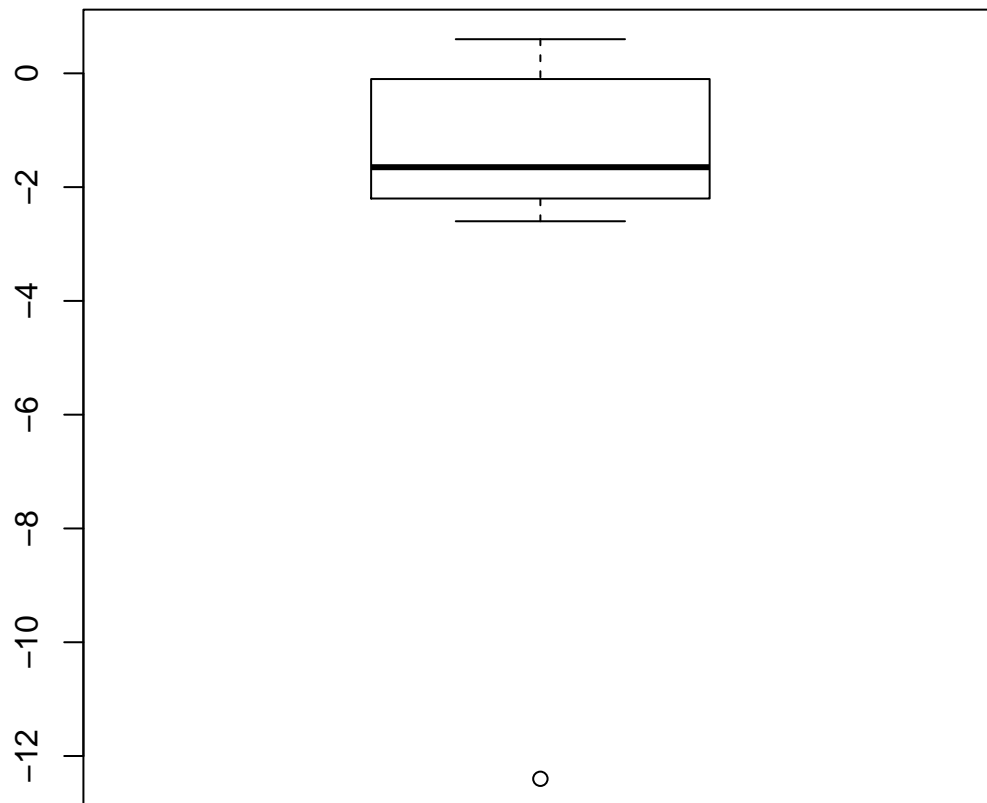
⁴²The `noprint` thing is to stop SAS printing all the other output that normally comes with `proc univariate`.

⁴³To ask SAS not to put the line on there, leave off the `/` and everything after it on the `qqplot` line.

⁴⁴Well, everything apart from the semicolon, anyway.

Another way to see the outlier is to make a boxplot:

```
R> boxplot(diff)
```



and you see a giant outlier at the lower end.

What to do about this? We have two possible approaches, and we will try both. One is a sign test, and the other is a randomization test.

How might the sign test go? Well, the 12 *differences* might be positive or

negative.⁴⁵ If the null hypothesis of no drug difference is true, those differences might be positive or negative, equally likely. So we can just count up how many positive differences (or negative ones) we have, and base a test on that.

In SAS, this is a question of asking for it: use `proc univariate` on the differences `diff` that we calculated. The null hypothesis of zero median is the default, so we don't have to give that, but we do want to concentrate only on `diff` and not the other variables:

```
SAS> proc univariate;
SAS>     var diff;
```

The UNIVARIATE Procedure
Variable: diff

Moments			
N	12	Sum Weights	12
Mean	-2.1333333	Sum Observations	-25.6
Std Deviation	3.40916768	Variance	11.6224242
Skewness	-2.8393709	Kurtosis	9.06926595
Uncorrected SS	182.46	Corrected SS	127.846667
Coeff Variation	-159.80473	Std Error Mean	0.98414194

Basic Statistical Measures			
Location		Variability	
Mean	-2.13333	Std Deviation	3.40917
Median	-1.65000	Variance	11.62242
Mode	-1.50000	Range	13.00000
		Interquartile Range	2.10000

Tests for Location: Mu0=0			
Test	-Statistic-	-----p Value-----	
Student's t	t -2.16771	Pr > t	0.0530
Sign	M -3	Pr >= M	0.1460
Signed Rank	S -32	Pr >= S	0.0088

Quantiles (Definition 5)

Quantile	Estimate
100% Max	0.60
99%	0.60
95%	0.60
90%	0.20
75% Q3	-0.10
50% Median	-1.65
25% Q1	-2.20
10%	-2.60
5%	-12.40

⁴⁵Or zero, but we don't have any of those here.

```

1%          -12.40
0% Min      -12.40

```

Extreme Observations			
----Lowest----		----Highest---	
Value	Obs	Value	Obs
-12.4	12	-1.5	9
-2.6	5	-0.3	3
-2.3	11	0.1	6
-2.1	2	0.2	4
-2.0	10	0.6	8

The sign test has P-value 0.1460, so we cannot reject the hypothesis that the median difference is zero. That is, there is no evidence of a difference between drugs.

R doesn't have a sign test built in, so we have to make one ourselves. That's not so hard, though.

The differences could be positive or negative, and if the null hypothesis of no difference is true, they are independently equally likely to be positive or negative. So the number of differences that actually come out (let's say) positive has a binomial distribution with $n = 12$ and $p = 0.5$. How many positive differences did we observe?

```

R> diff
[1] -1.5 -2.1 -0.3  0.2 -2.6  0.1 -1.8  0.6 -1.5 -2.0 -2.3 -12.4

R> table(diff>0)
FALSE  TRUE
   9     3

```

Three positive differences and nine negative ones.

The P-value for the sign test is the probability of observing 3 or fewer⁴⁶ positive differences, doubled, since the test is two-sided:

```

R> 2*pbinom(3,12,0.5)
[1] 0.1459961

```

This is, as with SAS, not anywhere near significant, and so there's no evidence that the median difference is anything other than zero. There are a couple of reasons for this: one is that the very large negative difference was having a disproportionate effect on the mean, and made the t -test come out more nearly significant than it ought to have done. Another is that the sign test is showing its lack of power.

⁴⁶“More extreme” is lower in that case. “9 or bigger” would be just as good, but that's more awkward to work out.

We can also get a confidence interval for the population median difference. This is done the same way as before, by inverting the test: asking which values of median difference would be rejected, with the data we had. Since we went to the trouble of writing a function to carry out the sign test, we can use that and a bit of trial and error. The results I got are:

```
R> sign.test(diff,-2.3)
[1] 0.03857422
R> sign.test(diff,-2.2)
[1] 0.1459961
R> sign.test(diff,0.1)
[1] 0.1459961
R> sign.test(diff,0.2)
[1] 0.03857422
```

which means that the confidence interval for the population median difference is from -2.2 to 0.1 , inclusive (-2.2 and 0.1 are definitely *inside* the interval.)

Another way to go is to do a randomization test, as we did above for the two separate samples (the reading programs). Now, our first thought needs to be “how to do the randomization”: what can be switched around if the null hypothesis is true? Well, if there is no difference between the drugs, the differences in pain relief could equally well have come out positive or negative. So we can make a randomization distribution by randomly allocating signs to our data and figuring out what kind of mean difference we get then.

First, we need the sample mean difference that we observed:

```
R> mean(diff)
[1] -2.133333
```

Then we need to generate a randomization sample, with random pluses and minuses, and find the mean of that:

```
R> set.seed(457299)
R> pm=c(1,-1)
R> random.pm=sample(pm,12,replace=T)
R> random.pm

[1] -1  1  1 -1  1 -1  1  1 -1 -1 -1  1

R> random.diff=diff*random.pm
R> random.diff

[1]  1.5 -2.1 -0.3 -0.2 -2.6 -0.1 -1.8  0.6  1.5  2.0  2.3 -12.4
```

```
R> mean(random.diff)
```

```
[1] -0.9666667
```

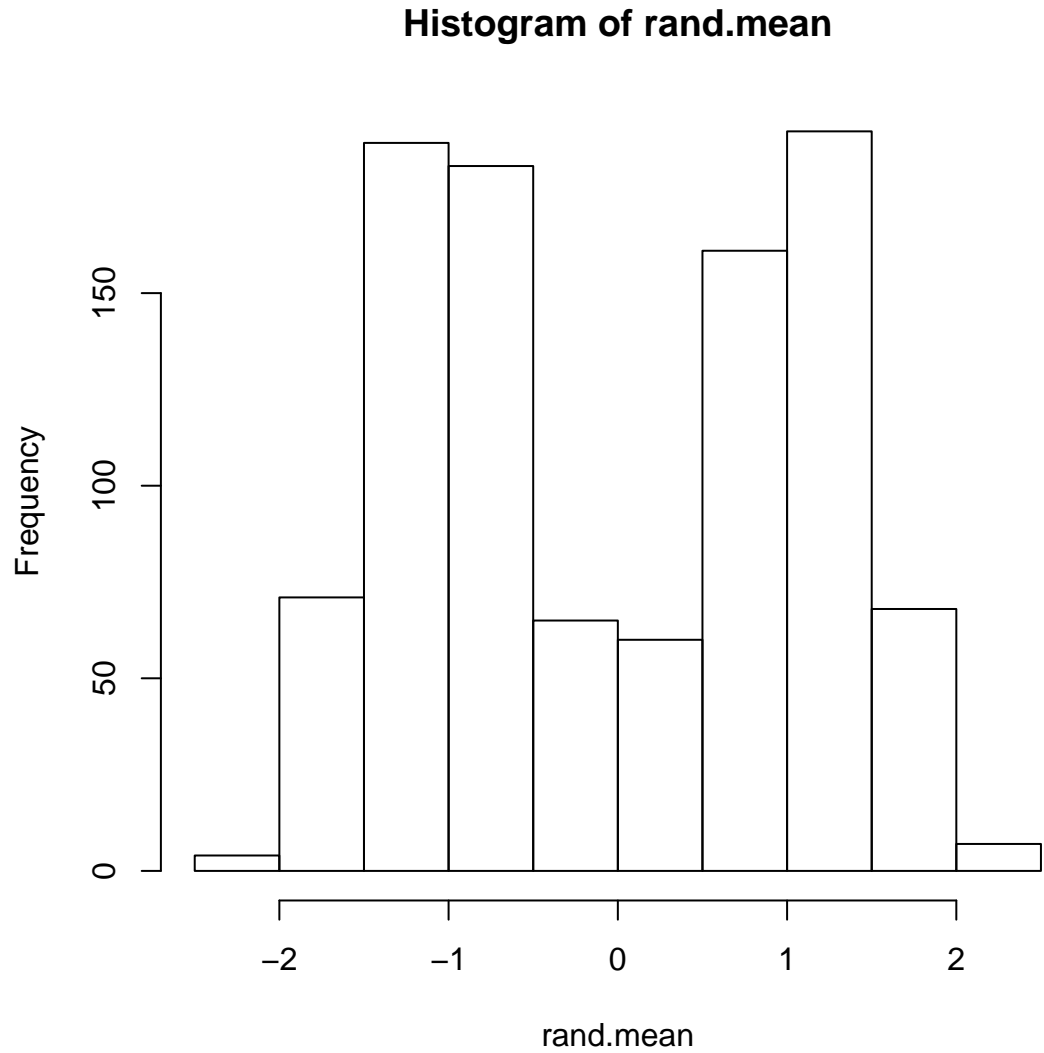
First, I set `pm` to be `+1` and `-1`. Then I generate a random string of 12 pluses and minuses by sampling from that with replacement. After that, I generate a randomization sample by taking the values in `diff` and multiplying them by the random signs. Finally, I take the mean of the result. Our first randomization sample has mean `-0.97`.

Now we need to do lots of these, and see where our observed `-2.13` falls on the distribution of values we get. The strategy is just like before. We create an empty vector to hold the results, and use a loop, filling in the `i`th value as we go. I'm using the plus/minus `pm` that I defined above.

```
R> nsim=1000;
R> rand.mean=numeric(nsim)
R> for (i in 1:nsim)
R> {
R>   random.pm=sample(pm,12,replace=T)
R>   random.diff=diff*random.pm
R>   rand.mean[i]=mean(random.diff)
R> }
```

Let's take a look at a histogram of those means:

```
R> hist(rand.mean)
```



This is interesting, because, if the paired t -test were plausible, this would look normal, which it doesn't. If you look back at our original sample of differences:

```
R> diff
[1] -1.5 -2.1 -0.3  0.2 -2.6  0.1 -1.8  0.6 -1.5 -2.0 -2.3 -12.4
R> mean(diff)
[1] -2.133333
```

the mean came out so negative largely because of the presence of that value -12.4 . This shows up again in the randomization distribution: if the 12.4 comes out with a plus sign, the sample mean will be about 1 ; if it comes out with a minus sign, the sample mean will be about -1 . There's a "hole" in the middle, because you can only get a mean near zero if most of the other values in the randomization sample have the *opposite* sign to the one attached to 12.4 , and the pluses and minuses should typically get about equally shared out.

This indicates a problem with a t -test in this situation, but the randomization test is just fine. We can get a P-value by counting how many of the randomized means came out less than -2.13 ("more extreme"), and then doubling it (two-sided test):

```
R> tab=table(rand.mean<=mean(diff))
R> tab
```

```
FALSE  TRUE
  998     2
```

```
R> 2*tab[2]/nsim
```

```
TRUE
0.004
```

Our randomization P-value is tiny, and now we *would* without doubt reject the hypothesis that mean pain relief for the two drugs is equal.

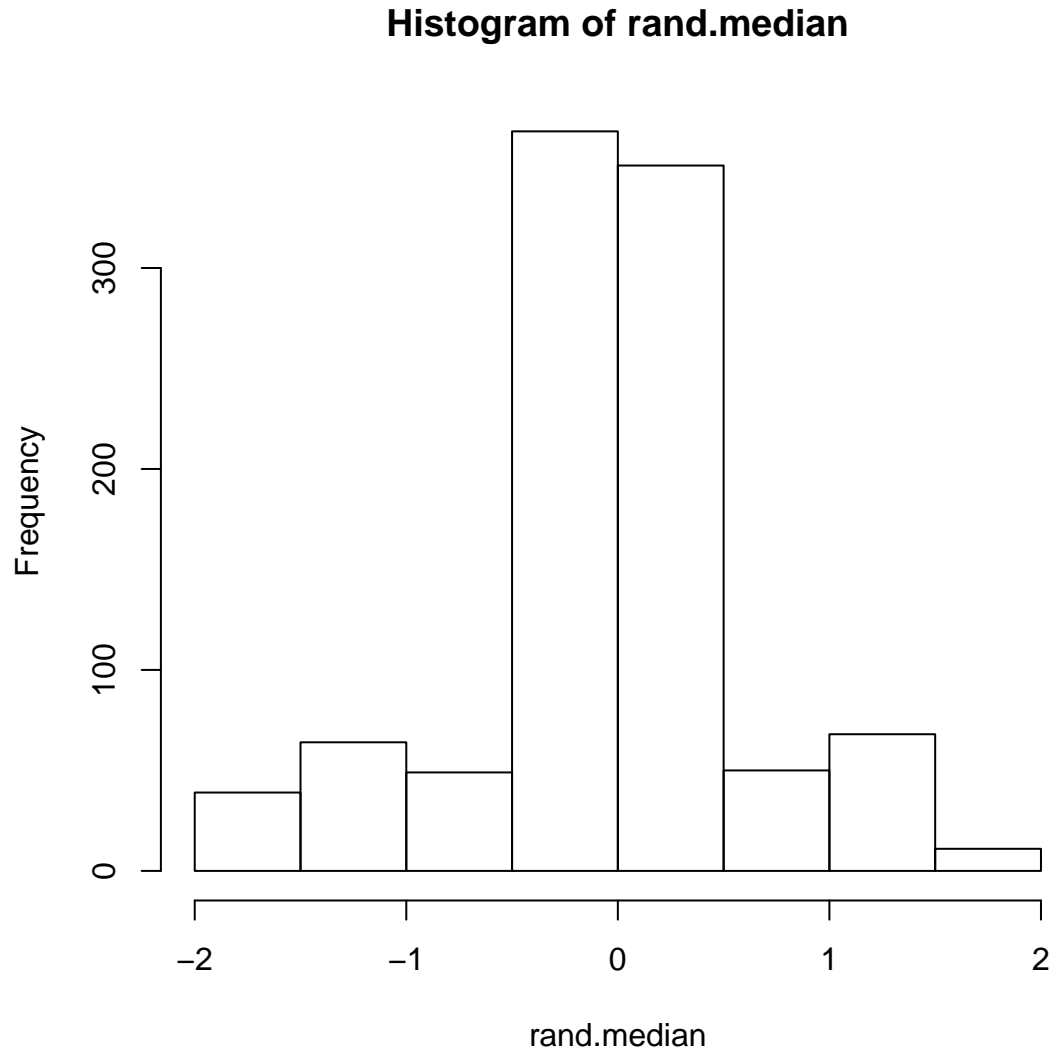
There was nothing wrong with the randomization test as a piece of inference, but maybe we should now be asking ourselves whether the mean is the right statistic to use. A randomization test is very flexible, though. Let's re-jig it to use the *median* difference:

```
R> nsim=1000;
R> rand.median=numeric(nsim)
R> for (i in 1:nsim)
R> {
R>   random.pm=sample(pm,12,replace=T)
R>   random.diff=diff*random.pm
R>   rand.median[i]=median(random.diff)
R> }
```

All I had to do was to change `mean` to `median` everywhere I saw it.

Histogram of the randomization distribution:

```
R> hist(rand.median)
```



This has one peak, at least, but it's more "peaky" than the normal distribution.

Let's get the P-value for the test that the *median* difference is zero:

```
R> tab=table(rand.median<=median(diff))
R> tab

FALSE  TRUE
  980   20
```


The sample median difference was -1.65 . There were 20 simulated median differences at least that small (actually, they were *all* equal to -1.65), so the P-value is

```
R> 2*tab[2]/nsim
TRUE
0.04
```

and we would (just) reject the hypothesis that the median pain relief times for the two drugs are equal at $\alpha = 0.05$.

It is curious that the sign test and the randomization test give such different results, since they were both based on the median. I suspect this is the result of the sign test not having much power.

6.10 Comparing more than two groups

Remember back in Chapter 5, when we were looking at jumping rats? We were trying to see whether a larger amount of exercise (represented in this experiment by jumping) went with a higher bone density. 30 rats were randomly assigned to one of three treatment groups (control, or no jumping; low jumping; high jumping). The random assignment means that the rats in each group should be similar in all important ways, including the ones we didn't think of, so that we are entitled to draw conclusions about cause and effect.

So now, we've decided to do a test to see whether jumping *does* have anything to do with bone density. What test?

You might remember that the usual test for comparing means of more than two groups is an analysis of variance (specifically a one-way analysis of variance). This has a complicated formula involving sums of squares and F -tests, but we don't need to remember any of that to do the test. Either R or SAS will do this. The usual way in R is via `aov`, but first we need to read the data in again:

```
R> rats=read.table("jumping.txt",header=T)
R> rats.aov=aov(density~group,data=rats)
R> summary(rats.aov)
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
group	2	7434	3717	7.978	0.0019 **
Residuals	27	12579	466		

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

We get the normal ANOVA table, which contains a small P-value. We can reject a null hypothesis, but what null, in favour of what alternative? We are trying to prove that the groups are somehow different in terms of mean bone density, so

the null hypothesis is that all the means are the same, and the alternative is the rather cagey “not all the means are the same”. When we had only two groups, there were several possibilities for the alternative, and we had to say which one we were looking for.⁴⁷ With three groups, there are a lot more, and it is hard to imagine knowing ahead of time which one we might be looking for.⁴⁸ So we pick the most general alternative, “the null hypothesis is wrong”.

OK, the P-value is much less than 0.05, so now we can conclude “the group means are not all equal”. This is as far as the ANOVA *F*-test will take us: “there is something going on”. But to find out what, we need some other test, coming up later.

SAS does the analysis of variance in a similar way. Again, we need to read the data first.⁴⁹

```
SAS> options linesize=70;
SAS>
SAS> data rats;
SAS>   infile 'jumping.txt' firstobs=2;
SAS>   input group $ density;
SAS>
SAS> proc anova;
SAS>   class group;
SAS>   model density=group;
SAS>
SAS> run;
```

The ANOVA Procedure

Class Level Information			
Class	Levels	Values	
group	3	Control	Highjump Lowjump

Number of Observations Read	30
Number of Observations Used	30

The ANOVA Procedure

Dependent Variable: density

Source	DF	Sum of Squares	Mean Square	F Value
Model	2	7433.86667	3716.93333	7.98
Error	27	12579.50000	465.90741	
Corrected Total	29	20013.36667		

⁴⁷One-sided or two sided; if one-sided, greater or less.

⁴⁸One possibility is that the group means are, if not all equal, in a specific order. This is the domain of the Terpstra-Jonckheere test, which we might look at later.

⁴⁹My reason for that is that I am in a different chapter of the notes from when I read the values in the first time. If you already read in these data *in this run of SAS*, you won't need to read in the values again.

Source		Pr > F			
Model		0.0019			
Error					
Corrected Total					
R-Square	Coeff Var	Root MSE	density	Mean	
0.371445	3.495906	21.58489		617.4333	
Source	DF	Anova SS	Mean Square	F Value	
group	2	7433.866667	3716.933333	7.98	
Source	Pr > F				
group	0.0019				

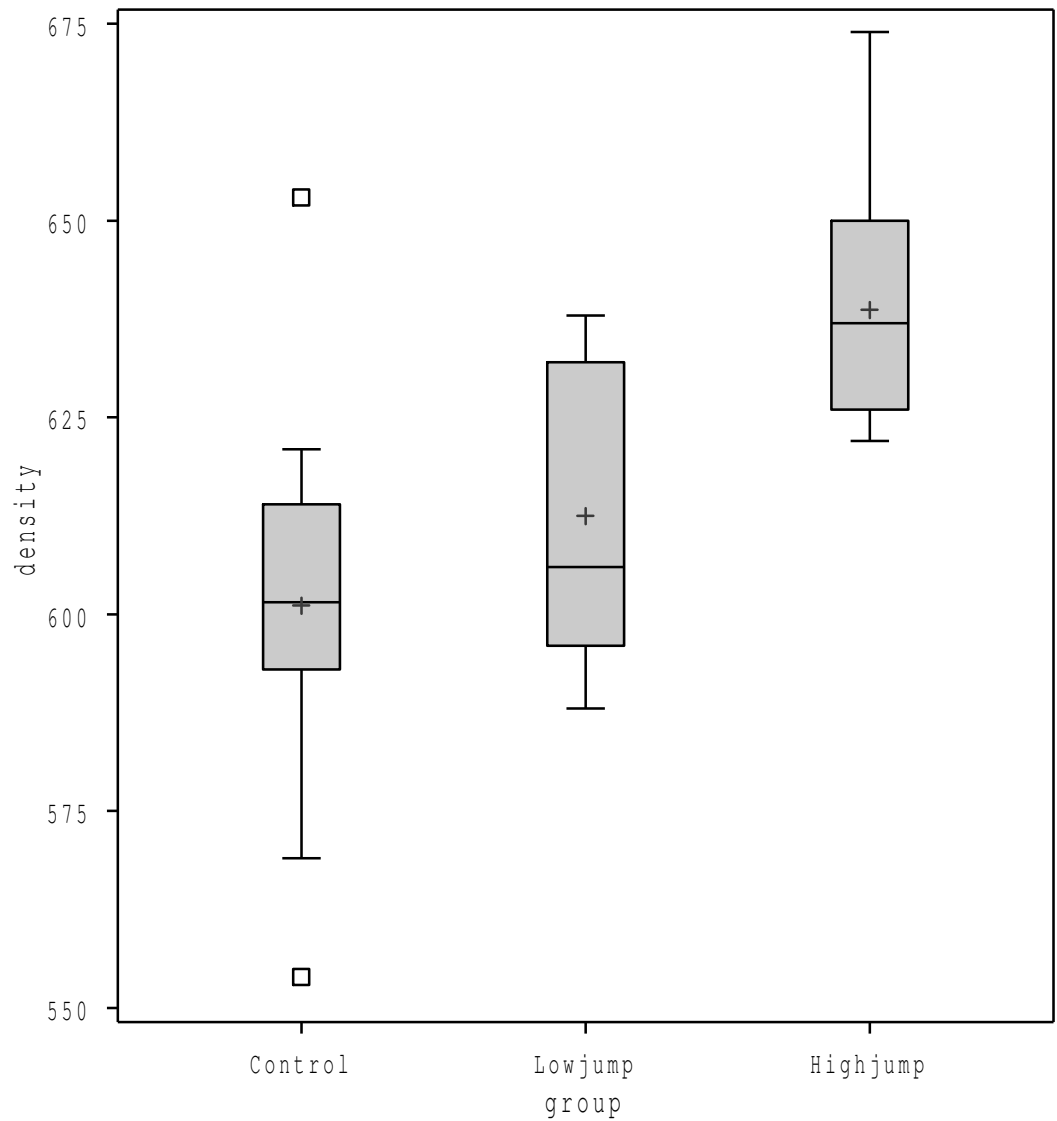
The P-value spills onto the next line,⁵⁰ but it's the same value as R produced.

We'll leave the issue of which groups are different from which until later, because I want to take up something else now.

If you obtain SAS HTML output, you'll also get the boxplots we had before. Here's a reminder of what they were:

```
SAS> proc boxplot;
SAS>   plot density*group / boxstyle=schematic;
```

⁵⁰It was that or have it disappear off the right side of the page.



One of the assumptions behind ANOVA is of equal variability within each group. According to the boxplots, the `control` group has outliers, while the `highjump` values are mostly consistent with each other. There are a couple of reasons why this might be. One is that the rats that got assigned to the control group just happen to have a large amount of variability in bone density. But the rats were assigned to groups at random, so this is hard to believe. Another possibility is that doing a lot of jumping makes the bone density values more consistent than they were before. In other words, it might be that doing a lot of jumping has the dual effect of increasing the mean bone density *and* decreasing the variability. This is hard to test directly.

ANOVA has a (theoretical) assumption of normality. Looking at the outliers on the control group values might make you doubt this. The question is not so much whether we should not do ANOVA (the Central Limit Theorem guarantees some robustness, even for samples of size 10 in each group), but whether the conclusions could be seriously in error. One way of handling that is via some kind of randomization test.

How can we do a randomization test here? There are two issues. One is the randomization part. There is no problem with this: the rats were randomly assigned to groups, so we randomly re-allocate the bone density values to the different groups, which we can do under the assumption that the null hypothesis of equal group means is true.⁵¹ The other question is what test statistic we use. We have a free choice on this: anything that helps us decide whether the groups are different or not. The F statistic from the ANOVA is one choice. I'm going to try something different: the largest group mean minus the smallest, whichever groups they come from. The idea here is that the largest group mean *has* to be bigger than the smallest, but if the group (population) means are all the same, then largest minus smallest won't be that big. If there really *is* a difference, the largest will be more bigger⁵² than the smallest.

All right, we'll do this in R, since that is easier for randomization tests. First, we need to find out largest mean minus smallest for our data. This is done via `aggregate`:

```
R> group.means=aggregate(density~group,data=rats,mean)
R> group.means

  group density
1 Control   601.1
2 Highjump   638.7
3 Lowjump    612.5
```

The actual means are in the second column of `group.means`. Let's first pull out the second column, and then take largest value minus smallest:

```
R> z=group.means[,2]
R> obs=max(z)-min(z)
R> obs

[1] 37.6
```

Now, we ask ourselves: “under the randomization, how often do we see a difference in largest minus smallest sample mean that is 37.6 or bigger?”.

To do that, we start by doing one randomization. We shuffle the group labels:

⁵¹You might be feeling a little uneasy about this, given that the high-jumping bone density values are less variable. You *should* feel a little uneasy. I'm going to press ahead anyway.

⁵²Yes, that really is grammatically correct.

```

R> set.seed(457299)
R> n.total=length(rats$group)
R> shuffled.groups=sample(rats$group,n.total)
R> shuffled.groups

[1] Highjump Control Control Lowjump Lowjump Lowjump Control Control
[9] Lowjump Lowjump Highjump Control Control Highjump Highjump Highjump
[17] Lowjump Highjump Lowjump Highjump Control Highjump Control Lowjump
[25] Highjump Highjump Lowjump Control Lowjump Control
Levels: Control Highjump Lowjump

```

You can argue about whether I should have attached `rats` first, to save those references to `rats$group`.

I also computed the total number of observations, which I called `n.total`, in case we need it again later. The groups have indeed been shuffled, since they were all in order before.

Now we need to calculate the group means, and find largest minus smallest. This is done as we did it for our data, except that we have to remember to use the *shuffled* means:

```

R> group.means=aggregate(density~shuffled.groups,data=rats,mean)
R> z=group.means[,2]
R> max(z)-min(z)

[1] 10.2

```

This time our largest minus smallest was 10.2.

That seems to work. Now we do it a bunch of times, saving the results from each time. You know the drill by now.

```

R> nsim=1000
R> test.stat=numeric(nsim)
R> for (i in 1:nsim)
R> {
R>   shuffled.groups=sample(rats$group,n.total)
R>   group.means=aggregate(density~shuffled.groups,data=rats,mean)
R>   z=group.means[,2]
R>   test.stat[i]=max(z)-min(z)
R> }

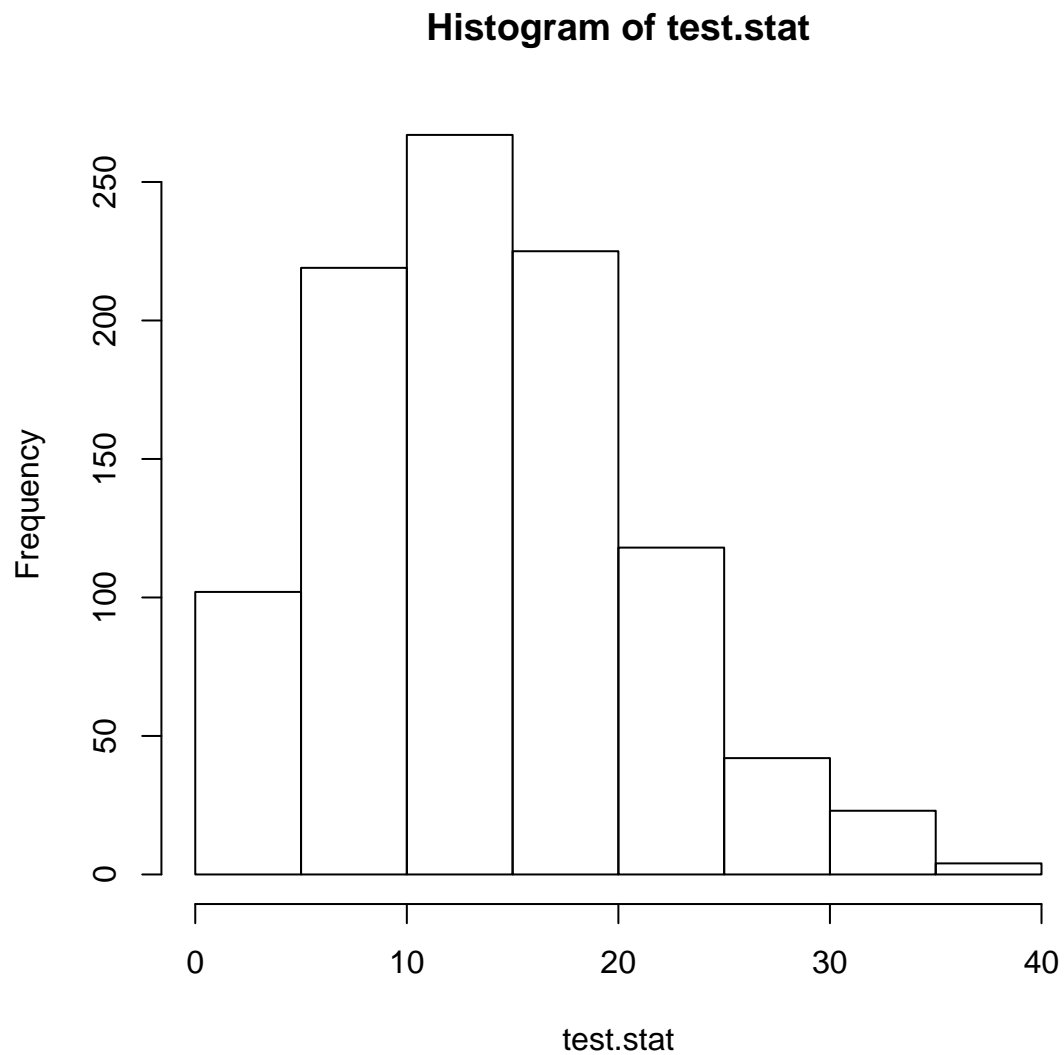
```

As a point of strategy, I got this wrong the first time. To debug the problem, I went back and set `nsim=10`, added `test.stat` as the last line below the loop (to print out the test statistic values), and set about finding my error.⁵³

⁵³I had `group` instead of `shuffled.groups`, so I was finding the means based on the *original* group allocation, and getting the same value every time! I come from the old school of debugging: print out all the variables I can think of until I find what went wrong. If you know something more sophisticated, go for it.

Let's take a look at our (correct) randomization distribution:

```
R> hist(test.stat)
```



This is a little bit skewed to the right. We wouldn't expect it to be normal, because the largest mean minus the smallest mean *has* to be bigger than zero, and distributions with bounds are often skewed. Anyway, we observed a largest minus smallest of

```
R> obs
```

[1] 37.6

which looks pretty unusually high. How many values are that high or higher? Count them. Or have R count them:

```
R> table(test.stat>=obs)
```

```
FALSE  TRUE
  998     2
```

We did 1000 randomizations, so the P-value is 0.002. This is an even smaller P-value than from the ANOVA. There is definitely a difference in mean bone density!

I wanted to mention an apparent contradiction here. We are doing a two-sided (in fact, multi-sided) test here, in that the alternative hypothesis is that *any* mean is different from the others. But our P-value is calculated in a one-sided way, by asking how often the values in the randomization distribution come out *bigger* than the 37.6 we observed. How come?

The idea is that *however* our null hypothesis of three equal means might be false, the effect on the test statistic is always the same: either the smallest mean will be smaller, or the largest mean will be larger. The effect on our test statistic largest-minus-smallest is the same: it will become bigger. So *only* large values of our test statistic should make us reject the null hypothesis that all the means are equal.

The same idea, though a bit less transparently, happens with the usual *F*-statistic for ANOVA: if the group means are in any way different, the group sum of squares will be larger than it would have been otherwise, and so the *F* statistic will be larger. So we should reject equal group means when the *F* statistic is large. But I think this is less obvious here than it is for my randomization test.

I wanted to finish off this section by returning to our randomization distribution and taking a look at how much larger the largest mean needs to be than the smallest mean before we declare the groups significantly different. This means finding the upper 5% and 1% points of the randomization distribution. R has a function `quantile` that can be used for this:

```
R> quantile(test.stat,probs=c(0.90,0.95,0.99))
```

```
  90%   95%   99%
23.600 27.010 32.702
```

Respectively. 90, 95 and 99% of the randomization distribution is less than (or equal to) the values shown, so 10, 5 and 1% of it is greater. At $\alpha = 0.05$, we would reject the null hypothesis that all the means are equal if we observe largest minus smallest of 27.01 or bigger.

6.10.1 Tukey's method in ANOVA

ANOVA (or my randomization test) only answers half the question we really have. It does tell us whether any of the group means are different, but it does *not* tell us *which ones* are different. For that, we need to do something else.

Let's start with the obvious. We could compare each *pair* of groups. Here, we have 3 groups, so we'd compare `control` vs. `lowjump`, `control` vs. `highjump` and then `lowjump` vs. `highjump`. We can do that for our data, with a little fiddling:

```
R> attach(rats)
R> controls=density[group=="Control"]
R> lows=density[group=="Lowjump"]
R> highs=density[group=="Highjump"]
```

The following object is masked from `kids`:

```
group
```

So far, we're picking out the `density` values for each group separately. So there should be ten of each, for example:

```
R> controls
[1] 611 621 614 593 593 653 600 554 603 569
```

Now we run `ttest`, feeding the appropriate two things in (and otherwise using the defaults). Here's the first one:

```
R> t.test(controls,lows)

Welch Two Sample t-test

data: controls and lows
t = -1.0761, df = 16.191, p-value = 0.2977
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -33.83725  11.03725
sample estimates:
mean of x mean of y
   601.1    612.5
```

There's no significant difference between these. What about the comparisons with `highjump`?

```
R> t.test(controls,highs)

Welch Two Sample t-test

data: controls and highs
```

```

t = -3.7155, df = 14.831, p-value = 0.002109
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  -59.19139 -16.00861
sample estimates:
mean of x mean of y
   601.1    638.7

R> t.test(lows, highs)

Welch Two Sample t-test

data: lows and highs
t = -3.2523, df = 17.597, p-value = 0.004525
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  -43.15242  -9.24758
sample estimates:
mean of x mean of y
   612.5    638.7

```

These two are both significant. So, high jumping produces a significant gain in bone density over low jumping or no jumping. Right?

I'm hoping you feel a certain amount of doubt right now. There is a problem, which can loosely be stated as “giving ourselves three chances to reject the null”. We are doing three tests, any of which might end with the conclusion that some groups are different. We can actually quantify this. Suppose that we have three groups whose population means are all the same (so the the ANOVA null hypothesis is true). Let's do the three pairwise comparisons using $\alpha = 0.05$. If we pretend that the three t -tests are independent⁵⁴, the probability of failing to reject any of the null hypotheses is $0.95^3 = 0.857$, so the probability of (incorrectly) rejecting at least one of the null hypotheses is $1 - 0.857 = 0.143$, which is bigger than 0.05. By doing three tests at once, we give ourselves too big a chance to falsely reject the overall null that all the means are equal.

John W. Tukey was an American statistician who was good at developing new ideas and, often, giving them cool names. He was one of the people behind the Fast Fourier Transform, and, more relevant to us, invented the boxplot, and also a way of avoiding the multiple-testing issue in ANOVA which he called “Honestly Significant Differences”.

The idea he had was like my randomization test example just above (only, he did the theory, and about 60 years before I thought of it). His idea was to start from the assumption that we have normally-distributed data with the same mean and SD from some number of groups. He also assumed that the groups were the

⁵⁴Which they are not, but the argument can be made rigorous without. See Boole's inequality, for example in the Wikipedia article on multiple comparisons.

same size, though this got generalized later. Under these conditions, he worked out how much bigger the largest sample mean would be than the smallest sample mean. His idea was that any groups whose means differed by the upper 5% point of this distribution could be called significantly different. The value of this is that only one test is done, to compare any number of groups: the comparison of the largest vs. the smallest. He saw, as I did,⁵⁵ that any differences between the population means would tend to make the largest sample mean bigger and/or the smallest one smaller, so that the difference would be bigger.

Honestly Significant Differences, or Tukey's method for short, can be done in both R and SAS. The R method illustrates a useful general technique in R: fit a model, and save the results in a "fitted model object". Then feed the fitted model object into something else that tells you about the model:

```
R> rats.aov=aov(density~group,data=rats)
R> TukeyHSD(rats.aov)
```

```
Tukey multiple comparisons of means
 95% family-wise confidence level
```

```
Fit: aov(formula = density ~ group, data = rats)
```

```
$group
```

	diff	lwr	upr	p adj
Highjump-Control	37.6	13.66604	61.533957	0.0016388
Lowjump-Control	11.4	-12.53396	35.333957	0.4744032
Lowjump-Highjump	-26.2	-50.13396	-2.266043	0.0297843

As before, there is no significant difference between `lowjump` and `control`, but `highjump` has a higher mean than the others. This is the same *conclusion* as we had with the pairwise *t*-tests, but let's compare the P-values:

Comparison	Tukey	t-tests
Highjump-Control	0.0016	0.0021
Lowjump-Control	0.4744	0.2977
Lowjump-Highjump	0.0298	0.0045

The Tukey P-values are higher.⁵⁶ This is the proper adjustment for the fact that we aren't looking at one pair of groups in isolation, but at all possible pairs of groups. Note that the `lowjump-highjump` comparison would no longer be significant at $\alpha = 0.01$.

The same idea in SAS:

```
SAS> proc anova;
SAS> class group;
```

⁵⁵only, 60 years earlier.

⁵⁶Mostly.

```
SAS> model density=group;
SAS> means group / tukey;
```

The ANOVA Procedure

Class Level Information			
Class	Levels	Values	
group	3	Control Highjump Lowjump	

Number of Observations Read	30
Number of Observations Used	30

The ANOVA Procedure

Dependent Variable: density

Source	DF	Sum of Squares	Mean Square	F Value
Model	2	7433.86667	3716.93333	7.98
Error	27	12579.50000	465.90741	
Corrected Total	29	20013.36667		

Source	Pr > F
Model	0.0019
Error	
Corrected Total	

R-Square	Coeff Var	Root MSE	density Mean
0.371445	3.495906	21.58489	617.4333

Source	DF	Anova SS	Mean Square	F Value
group	2	7433.86667	3716.93333	7.98

Source	Pr > F
group	0.0019

The ANOVA Procedure

Tukey's Studentized Range (HSD) Test for density

NOTE: This test controls the Type I experimentwise error rate, but it generally has a higher Type II error rate than REGWQ.

Alpha	0.05
Error Degrees of Freedom	27
Error Mean Square	465.9074
Critical Value of Studentized Range	3.50633
Minimum Significant Difference	23.933

Means with the same letter are not significantly different.

T
u

```

k
e
y
G
r
o
u
p
i
n
g
      Mean      N      group
A      638.700    10    Highjump
B      612.500    10    Lowjump
B
B      601.100    10    Control

```

Look at the end of the output, under **Tukey Grouping** (printed vertically⁵⁷). The **highjump** group has a letter **A** next to it, which is different from the **B** next to both **lowjump** and **control**. This means that there is no significant difference between **lowjump** and **control** (same letter), but **highjump** is significantly different from both (different letter).

One of the things that SAS gives you here and R doesn't is under the ANOVA table. There is a table of five values, with the bottom one being **Minimum Significant Difference**. Any means differing by more than this (the value here being 23.933) are significantly different at $\alpha = 0.05$. This is by Tukey's calculation. It is interesting⁵⁸ to compare that value with the corresponding one from my randomization distribution. I made a table like this before:

```

R> quantile(test.stat,probs=c(0.90,0.95,0.99))
      90%      95%      99%
23.600 27.010 32.702

```

and the appropriate one for comparison is the middle 0.95 one, or 27.010. This is a bit bigger than the 23.933 that comes out of the Tukey calculation, but they are in the same ballpark. According to this, doing a “randomization Tukey” for comparing the groups, **highjump** is no longer significantly higher than **lowjump**, because the group means differ by 26.2, which is less than 27.01 (though it is close to significance at $\alpha = 0.05$), but the other comparisons come out with the same conclusions.

⁵⁷Don't ask me.

⁵⁸To me. I don't know about you.

Chapter 7

Writing reports

7.1 Introduction

You might be the world's greatest researcher, but that is all worthless if nobody knows about what you've done. You have to be able to *communicate* your findings to the world at large, and that means writing reports and papers to describe what you've done.

One of the key principles about research is that it should be *reproducible*: that is, anyone else should be able to reproduce your research and verify that it is correct. That means that it is up to you to describe what you did clearly enough for someone else to do it themselves, and it is also up to you to support the conclusions you draw so that your readers will believe you. This is the way science, or human knowledge in general, works.

In science, if you're going to say something about how the world works, you need to say something *falsifiable*: that is, something that is open to being proved wrong. A statement that is falsifiable but not actually falsified is one that science considers **true**: something like "life arose by evolution", for example. Falsifiability is often what distinguishes science from pseudo-science. We know (or at least have great confidence) that drugs work because they have undergone randomized controlled trials: the developers have to show that a drug works significantly better than placebo (or the drug that it is designed to replace). The new drug *could* be worthless¹, but the drug trials show that it is actually effective. Drug trials typically use *huge* numbers of patients, so there are *many* chances for the drug not to work. And yet, on average, it does, more than chance would indicate.² The process is rigorous, and deliberately so. A new drug, or a new piece of science, has to prove its worth before it becomes accepted.

¹This is the "falsifiable" part.

²This is where statistical significance comes in.

***** example of pseudo-science here: eg. anti-vaccination

7.2 Writing a report

So how do you write a report? Think of it as telling the story of what you did. A story has characters, a plot, interaction between characters, conflict, resolution, stuff like that. You might not have all of that in a statistical report, but the idea is the same: the characters are the people or animals involved, the plot is what you are going to do with them and what you hope to show, the conflict is whether your hoped-for effect actually shows up, and the resolution is your description of how it did (or your reasons for why it didn't).

Different sciences have different preferred ways of writing a report. Some like to formalize the process, with specified sections. What I'll describe is probably best when the emphasis is on statistical methods. You can adapt as necessary.

The first thing to think about, as with any communication, is your intended audience. The report is not for you, it is for your audience, and should be written with them in mind. You are entitled to expect that they will know some statistics (of the kind covered in courses or textbooks), but you should be prepared to explain anything out of the ordinary. I would say that a two-sample *t*-test or ANOVA is something to expect your audience to know, but a randomization test is something you'd need to explain.

The report should be divided up into sections, typically numbered so that you can refer back to them.

Start with an Introduction. This needs to describe what your study is all about and why you are pursuing it. This is the place to describe what you are expecting to see, what research hypotheses you have, and so on. A British statistical journal advises that writing should be "as concise as possible without detracting from clarity", which is especially good advice to follow for an Introduction.

Depending on the purpose of the report, you might also want a literature review. This is a study of what other people have written on the subject matter of your report, or related matters. This will include lots of references, and statements like "It is known (Anderson and Ho, 2005) that..." and "Smith and Van Beerlenhausen (1994) assert that...". Save up the articles and books you refer to for your References list for later. You're trying to place your work in context, to convince your audience that you know what other people have done and that you have considered their opinions. Science really *does* proceed by seeing further because you are standing on the shoulders of giants!

There are different ways of citing things; use the one that is appropriate for your field of study. Sometimes references are indicated as I did above, with author and year. In that case, your References list should have the references sorted by alphabetical order of author. Sometimes, the reference is just indicated by

a number, like this: “In [1] it is shown that..., but the conclusions of [2] are that...”. In that case, you sort the references by *number* at the end. The first way is the norm in Statistics journals, while the second is used in mathematics journals.

It is a matter of taste whether the literature review is in a separate section titled as such, or whether you combine it with the Introduction. Well, a matter of taste and the norms of the subject area.

Now you need to describe what *you did*. This section is often called “Methods”, but sometimes you will see more descriptive section titles that summarize what you did. What experiment did you conduct? How did you design it? What results did you collect? How did you collect them? This is not the place for justifying what you did. That can go in the Conclusions, later. Here you need a matter-of-fact and above all *clear* story of what you did. Remember, you are aiming for reproducibility: you want someone else to be able to do exactly what you did. Sometimes the science requires you to use some kind of instrumentation to collect your data. If that’s the case, you need to say exactly what instrumentation was used, along with any settings etc. that are needed to allow somebody else to follow along. In some fields, this is a separate section called Methods.

Next come your Results. You may have to make a judgement about how you handle this. If you have one block of results, you can just state them (maybe in a Results section) and move on to the Conclusions. But you might have a more complicated statistical analysis to report on, along the lines of “first I did this and got these results, so then I did *that* and got *those* results, and that suggested *the other*, with some other results.” If that’s the case, you might need several sections to lay things out clearly. Take the time to tell your story clearly. There might be some overlap with the “what you did”, in which case you can combine Methods and Results into one. Or the various parts of your analysis might require a mini-Methods before each one. Organize things the way that tells your story best.

How should you present your results? There is an old saying that “a picture is worth a thousand words”. If you can display your results in a graph, or several graphs, that’s often best. You want to avoid large tables of numbers, which will send your audience to sleep! The point is to display your results in a way that will make your Conclusions (coming up) clearest.

Everyone asks “how many significant digits should I quote in my results?” Here’s advice I like:

Only quote figures that you are pretty sure of — perhaps plus one more.

You should know how accurate your data values are (generally, the accuracy that is given in the data set; if the values were more accurate, they would have been given to an extra decimal place). Quantities derived directly from the data,

like the mean or SD, deserve the same number of decimal places, or maybe one more. Things like regression slopes are a bit trickier, because they are change in y divided by change in x . A good rule here is to ask yourself how many significant digits there are in the y measurements, how many there are in x , and take the smaller of those as the number of significant digits in the result. Perhaps plus one more.

As with the Methods, “matter-of-fact” is key here. The purpose of this section is to *report* on the results that you observed, not to *comment* on them. You want your audience to be clear about *what* happened. *Why* it happened, and what it means, does not belong here. There is, it has to be said, a tension between the way in which you choose to show your results, and the “simply reporting” part of presenting them.

When displaying graphs (and tables of results, if those are necessary), put them in separate numbered Figures and Tables. This enables you easily to direct the reader to “Figure 4” or whatever, and it’s entirely clear which of your graphs you mean. Take advantage of your software’s ability to number things automatically and to create cross-references, so that if you decide to add another graph or change around the order of things, the numbering and cross-referencing will change accordingly.

Figures and Tables need to have captions (underneath) in a report. These serve the purpose of a title, so that a graph doesn’t need to have a separate title. The axes on a graph should be labelled so that it is clear what they refer to.

What you put in your Results section should not be so bulky that it gets in the way of the story you’re telling. If you have a large table of numbers, for example, that is important to your report, but would require a large amount of effort for your audience to digest, by which time they would have forgotten the story you’re telling, then create an Appendix at the end of the report and put it there. If you want to include other things like a listing of data or the code that you used, the Appendix is where this belongs too.³

Now you get to talk about what your results mean. After the Results come your Conclusions. This is where you get to sell your results and the consequences of them, so that the world will know about them. This is the place where you definitely *do* want to add your opinions: *why* these results are important, *what* they mean, *why* we should care about them. Back in the Introduction, you laid out what you were investigating, what you hoped to find out and how you planned to find it out. This is the place to tie up those loose threads: *did* you find out what you were hoping to? How do your results lead to that conclusion? If you didn’t find out what you were hoping to, why do you think you didn’t?⁴

³In general, your code is of secondary interest, and worth including only if it has something particular to say, for example if you’ve done something new. In this course, however, I *want* to see exactly how you did an analysis. Therefore, in this course, your code should be in the main body of the report.

⁴There are often issues of power or sample size here.

Do your results suggest future lines of research?⁵

Sometimes people call the Conclusions section Discussion, or have separate small sections of both titles. In that case, Conclusions contains what follows directly from your results⁶, and Discussion is where you place your conclusions in a broader context.⁷

At the end come acknowledgments to people and organizations that helped you, and a list of the references you cited.

And that's it.

I wanted to mention one more thing, which is that articles in a journal usually open with an Abstract, which is a summary of what the article is about. My preference is that the Abstract should *not* have references and ought not to give away the results, though you might have to adapt this for the norms of the journal that you're trying to get your paper published in. The point of an Abstract is that anyone should be able to skim it quickly enough to get a sense of whether the whole article is worth their time to read.

This brings up another point: your work is competing with a lot of other reports for people's attention, so you want to write your report in a way that makes it *as easy as possible* to read and understand. Writing in a deliberately confusing or long-winded way does not advance the cause of science one bit!

7.3 Example report

The following journal article illustrates the points I made above about structuring a report:

<http://0-jap.physiology.org.library.pcc.edu/content/100/3/839.full#abstract-1>

I don't know anything about physiology, but that's not the point here: the point is to see the structure of the report. I chose this report because I was thinking about the jumping rats, and because it is basically a statistical report.

My comments:

- This paper begins with an Abstract. To my mind, this abstract is really an Introduction (and Literature Review), because it contains quite a lot of detail and (in the last two or three paragraphs, references.) It also gives away some of the results (which, to my mind, spoils the telling of the story). Maybe that's what the journal editor wanted. I think an Abstract should stand on its own; by reading it you ought to be able to learn what

⁵They almost always do.

⁶"What do my results mean to me?"

⁷"What do my results mean to the world?"

the paper is trying to find out and thus whether it is worthwhile reading it.

- Some journals like you to choose some Key Words for your article, here “jump exercise”, “peak bone mass”, “high-impact training”. This helps the reader find more articles similar to yours, and thus to place your work in a broader context.
- This article uses numbered references. In an online report, the citations (numbered references) often link to the corresponding entry in the List of References at the end.
- This article’s “what you did” section is called Materials and Methods. This is a common title in scientific work; often the things that need to be explained are not only what you did, but also what technology you used to measure the results. In this article, a lot of different technology needed to be used, from X-rays to a diet database to a device for measuring the height of jumps.
- Note how the Results section only goes through the results obtained, but does not comment on them. Here, the non-significant results are reported as well, which is good science. Otherwise, the researchers could be accused of “cherry-picking” the best results and ignoring the others. There’s a nice quote by physicist Richard Feynman about this:

I would like to add something that’s not essential to the science, but something I kind of believe, which is that you should not fool the laymen when you’re talking as a scientist.... I’m talking about a specific, extra type of integrity that is not lying, but bending over backwards to show how you’re maybe wrong, [an integrity] that you ought to have when acting as a scientist. And this is our responsibility as scientists, certainly to other scientists, and I think to laymen.

<http://www.ar-tiste.com/feynman-on-honesty.html>

- Ideally, you should give enough detail in your results to allow others to reproduce your analysis. I think that happens here — the notation 40.88 ± 6.12 gives the mean and standard deviation for each group.⁸ Having the group means and SDs ought to be enough to reproduce an analysis of variance or *t*-test, though of course having the actual data would be better, since then you’d be able to do boxplots and other things to assess whether an ANOVA is reasonable.
- My last point clashes with the limited space available in most journals, and some kind of compromise has to be made. This is less of an issue than it used to be, now that a lot of research is written up online, with data available on websites and so on.

⁸ *Not*, I think, a confidence interval, even though it looks like one.

- This article’s Conclusions section is called Discussion. Notice the large number of references to the literature, to place the results in context. Note phrases like “our results are consistent with...”, and the recognition that “there are some limitations of the present study” that allow the authors to generalize the results less widely than would be possible otherwise.
- Note the clear plain-English last sentence: “low-repetition high-impact jumps are suggested to be one of the ideal training methods for enhancing and maintaining peak bone mass in young adult women”. I know nothing about physiology, but I can understand *that*!
- The Acknowledgements thank a bunch of people by name.
- This List of References is in both numerical and alphabetical order, which means that when the references are cited in the text, they are not in numerical order.⁹
- Medline is a subscription service that allows the reader to access abstracts or (sometimes) full text of medical journal articles. I don’t know why these links take you to Portland Community College. Perhaps one of the authors has an account there.

7.4 This Thing Called Science

What is science? A series of videos, about 2 mins each:

<https://www.youtube.com/watch?v=W9IoN8Tb1wg>

Or, going back a few more years:

<http://www.youtube.com/watch?v=EYPapE-3FRw>

7.5 Reproducible Research

7.5.1 Introduction

“Reproducible research” is kind of a buzzword now.

The old way of publishing your research was to submit your paper to a journal, have it edited down to a length that just barely describes your work, and be prepared to field requests for more information from people who read it.¹⁰

The modern way of publishing your research is to put your report *and* data on a website or forum somewhere, along with all your code, so that anyone with

⁹You can check this.

¹⁰Some researchers honoured those requests, and some didn’t.

R or SAS (as appropriate) can literally reproduce your analysis. Anyone who disagrees with your analysis can then contact you and you can then have a civil¹¹ discussion. This is peer review at its most immediate.

One potential problem (for you) is that the code you put up might not actually produce the results you claim. How is this possible? Well, think about how you put together your code to post it. You probably copied and pasted it from R Studio or the SAS Program Editor. But what happens if you change your mind and re-do part of your analysis? Your code has changed, and your output has changed, so you have to remember to copy and paste *both* of those, *and* if necessary re-write the bit of the report that includes changed code and results. It is terribly easy to forget one of those, and then your code, output and report are out of sync. Is there a way of guaranteeing that the code in the report actually did produce the results you say it did?

7.5.2 R Markdown

If you are running R, the answer is yes. What you do is construct a specially formatted document that includes code to run. When you process that document, the code gets replaced by its output *by actually running the code*, so that the out-of-sync thing cannot happen. Then you scan through the latest version of the report to make sure that the conclusions you drew still apply, and you are good.

R Studio makes it relatively easy to apply this. The key thing is the “special formatting”. This is done using a markup language¹² called R Markdown.¹³

So, for this you need to learn R Markdown. This is not so hard, and the way you express things is very mnemonic.

In R Studio, select File, New and R Markdown (from the dropdown). You’ll see an example document, which you can study to see how it works. Before we get to that, though, you’ll probably see a yellow bar at the top of the window saying “R Markdown requires the knitr package”. That means you have some work to do first.

If you’ve never used R Markdown before, you’ll need to go down to the Console window and type this:

```
R> install.packages("knitr")
```

You might get asked to “select a mirror”. Choose one geographically close from the list. Then you’ll see a lot of red text speed by, ending with DONE (**knitr**).

¹¹In theory.

¹²Like HTML. If you “view source” on a web page, you’ll see a list of codes enclosed between < and > that tell your browser how to display the page. This is HTML. The advantage is that the actual HTML code itself is just plain text.

¹³Markdown, markup. Get it?

That’s the first part. You won’t need to do that again.¹⁴

Next, you have to make the package available for R to use. This is done this way:

```
R> library(knitr)
```

Note the absence of quotes this time. This will need to be done each time you want to use R Markdown (strictly, once per run of R Studio).

Now you can process the trial document. Find “Knit HTML” next to the picture of the ball of wool. Click it. You’ll be prompted for a name for your R Markdown file. I called mine `test`, without a file extension. You’ll see that the tab has changed to something like `test.Rmd`. Then there is a little processing, and another window, marked Preview, will pop up with the formatted document. This is actually an HTML document, like a web page. You can save it to an HTML file if you wish (and then you can view it with a regular web browser).

If you put the R Markdown window and the preview window side by side, you can see what the formatting instructions in the R Markdown file do:

- The row of =====, with a blank line below, formats the text above it as a title.¹⁵
- Putting ****** around some text makes it bold.¹⁶
- The line ````{r}` starts an **R code chunk**. All the lines until a closing ````` are interpreted as R code. In this case, we are making a summary of one of R’s standard data sets called `cars`¹⁷ which has measurements of speed and distance. R Studio helpfully colours a code chunk grey so that you can see how long it is. This one is only one line, but you can have as many lines as you like. Now, turn to the preview window to see what the R code chunk has turned into. It has been replaced by the code itself (in a grey box) *and* the output from the code, in a white box.
- Including a plot in your report is just the same, as the next code chunk shows.

I invite you to play with the R Markdown file. For example, you can change the title, add some new text, or add some new analyses or plots. Try adding `quantile(cars[,2])` to the first code chunk, and compare to the output from `summary`. Add a new code chunk by placing the cursor where you want the new code chunk to go, and selecting Chunks (top right of the window) and Insert Chunk. Or control-shift-I if you like the keyboard. Try making a histogram of the distances (`cars$dist`). Use your imagination! When you want to see the effect of your changes, click Knit HTML again.

¹⁴This is called “installing a package”.

¹⁵An `<h1>` header, in HTML terms.

¹⁶A single `*` around text makes *italics*.

¹⁷Not, confusingly, *my* data set of the same name.

You might be able to remember how to do something in HTML but not in R Markdown; in that case, you can use HTML code as well.

If you know \LaTeX , you can add things like equations in \LaTeX format, such as this: `$y=x^2 + \frac{7}{x+2}$` . Try it and see what it does.

To write your own report, just delete the text that is there and save the file under a new name.

If you want to explore the Markdown language further, click the MD button to the left of the ball of knitting wool. This will pop up (in the Help window) a mini-manual that tells you how to do all kinds of things in Markdown.

Why is this helpful for reproducible research? Well, no results are produced until you click Knit HTML, so the results that appear in the preview window are guaranteed to be from the code chunks in the R Markdown document. That is, the code and results you see in the Preview window are *guaranteed* to be in sync.

If you look at the files in the R Project folder that you're currently working in, you'll see files like `test.Rmd`, which is the R Markdown file that you created, `test.html`, which is the HTML file created by Knit HTML, and also `test.md`. This last is a Markdown file.¹⁸ If you take a look at its contents, you'll see that it looks a lot like the R Markdown file that you created, except that instead of R code chunks it has the R code between ````r` and `````, and the results surrounded by `````. The way Knit HTML works is first to run R on the R Markdown file to produce the Markdown file, which has all the results in it, and then it formats the Markdown file to produce the HTML file that you see in the Preview window.

7.5.3 Pandoc and making Word docs

I mention the Markdown file, because the Markdown file can be processed in different ways to produce different output formats. The tool that does this is called **pandoc**.¹⁹ Possible output formats include:

- plain text
- Word
- Libre Office document
- Rich Text Format
- Slidy and slidify web-based slide show
- PDF via LaTeX (if latter installed)

¹⁸Not to be confused with the R Markdown file.

¹⁹“Pan” being a Greek prefix meaning “all”.

A better way to get PDF-via-LaTeX documents out of R is to use a different mechanism called Sweave. I'll talk about that later.

A good introduction to `pandoc` is at

<http://johnmacfarlane.net/pandoc/getting-started.html>

You'll need to install it on your computer, and run it, either from a command line or from within R Studio.²⁰ The website above leads you through the installation.

First, I need to say a word or two about using R as a command-line, since that's how `pandoc` works. On Windows, the DOS command `dir` produces a list of all the files in a folder. To get that list of files for the folder that R is currently in, go to an R Script window²¹, and type these two lines:

```
R> mycommand="dir"
R> system(mycommand)
```

There are two parts to this: first you construct the command you're going to send to the command line, and then you actually send it via `system`. The value of this is that `system` knows whether you're on Windows, Mac, Linux or whatever and sends the command to the appropriate command line.

Now, I'm on Linux, so the corresponding command for listing all the files is `ls`. I have rather a lot of files, so I'm just going to list the ones that start with `test`. On DOS, you'd type `dir test.*`; on Linux the appropriate thing is `ls test*`.²² That, with the output it produces, looks like this for me:

```
R> mycommand="ls test*"
R> system(mycommand)
```

```
testing.log
testing.lst
testing.odt
testing.sas
test.log
test.lst
test.md
test_pres.html
test.Rmd
test.sas
test_s.html
```

There you see the R Markdown file `test.Rmd`, the Markdown file `test.md` that Knitr made from it and the HTML file `test.html` that we saw in R Studio, among other things.²³

²⁰The latter is easier if you're in Windows.

²¹Or the Console window, if you must.

²²Which will also catch files whose name isn't exactly `test`, but which starts with `test`.

²³There's also a Word file `test.doc`. I'll show you how to get that in a minute.

A couple of other things about `system`, while we're here. The first is that you can construct the command you want to run using variables you already have. The `paste` command is handy for this: it glues together text and variables you give it, with spaces in between, like this:

```
R> pattern="test*"
R> mycommand=paste("ls",pattern)
R> mycommand
R> system(mycommand)
```

```
[1] "ls test*"
testing.log
testing.lst
testing.odt
testing.sas
test.log
test.lst
test.md
test_pres.html
test.Rmd
test.sas
test_s.html
```

The second other thing is that you can save the output from `system`, and access the bits (if it has more than one bit). This requires an option on `system` as shown:

```
R> files=system(mycommand,intern=T)
R> files

[1] "testing.log"      "testing.lst"      "testing.odt"      "testing.sas"
[5] "test.log"         "test.lst"         "test.md"          "test_pres.html"
[9] "test.Rmd"         "test.sas"         "test_s.html"
```

`output` is an R vector (of character strings), so you can find the third file name (say) like this:

```
R> files[3]

[1] "testing.odt"
```

Anyway, that's more about `system` than you probably want to know.

So, how do you use `pandoc` to make a Word document from a Markdown file? The basic syntax of `pandoc` is that you have to give it the name of a Markdown file and the name of the file where you want the output to go. The file extension says what kind of output you want to produce. Thus, a Word document is made like this (on the command line):

```
pandoc test.md -o test.doc
```

and thus you would run this from R Studio like this:

```
R> mycommand="pandoc test.md -o test.doc"
R> system(mycommand)
```

There's no output, but if you look in the folder you're working in, you'll a file `test.doc` has appeared, and you can open it in Word and look at it.

If you prefer to get a `.docx` file, replace `test.doc` by `test.docx` above.

There's a strategy issue here. You might look over the Word file and be tempted to make some changes to it. For example, you might see a caption under the plot "Plot of chunk unnamed-chunk-2" and feel the need to edit it. **Don't**. The reason is that any changes you make here are *not reproducible*, in that if you go back to R Studio and change any of your analysis, how are you going to change the corresponding Word document? The correct procedure²⁴ is this:

1. Do your analysis in R Studio, using an R Script window to store the commands you use. There will be false starts and things that didn't work. That's OK here.
2. Decide on how you want to actually present the analysis. Copy the commands that make this happen to the bottom of your R Script window. Or create a new one and copy them there, if you prefer. This is not the place to decide on extra things you want to add. Step 1 should contain everything you need here, plus some stuff you're deciding not to include in the final report.
3. Run the report-ready code from Step 2 again, to make sure it works. If it doesn't, go back to Step 1 and fix it up.
4. Create an R Markdown file out of your code from Step 2, once you've checked that it works. Put the code into small code chunks, and surround the code chunks with text to make up the rest of your report. I like the code chunks to be small enough so that I can describe what I'm going to do next, and afterwards talk about the results from that chunk. For example, I might make a scatter plot with a lowess curve on it (one chunk, two lines of code). Before the chunk, I might say why I am going to draw this scatter plot; afterwards, I can say something like "the scatter plot shows that the relationship between `foo` and `blah` is approximately linear", and that might lead on to doing a regression.
5. Knit HTML on your R Markdown file.
6. Take a look at your output. This is²⁵ what you'll be showing to the world. If you don't like it, go back to Step 4 and change the R Markdown to fix it. If you feel you need to add something, go right back to Step 1, test

²⁴The business-world term for this now appears to be "workflow".

²⁵More or less. The final version might look different, since it'll be a Word document or a presentation or something like that. But what it *contains* is what you'll be showing to the world.

what you need to do until it's working, put it in with your working code, and then add it to the R Markdown file. Go back to Step 5.

7. The point of this is that your R Markdown file is the core of your report. *This* is the thing you change, because the whole rest of your report is reproduced from that.
8. When you're happy with how the HTML in the Preview window looks, follow the steps above to turn it into a Word document, or the steps below to turn it into a presentation. This step comes right at the end of the report-writing process.
9. If there's something in your final document that you don't like, go all the way back to Step 4 and fix it in the R Markdown file. This could be R-related²⁶, or something with the text.
10. The only changes you make to your Word document are cosmetic ones that can't be fixed in R Markdown, things like numbering your Figures and cross-referencing them, or compiling the list of references and citations of them.

7.5.4 Making a Slidy slideshow using pandoc

All right, let's see how we turn an R Markdown file into a presentation. The format I'll use here is called `slidy`: this makes a presentation that can be viewed in a web browser, so that you don't even have to worry about the computer in the presentation room having PowerPoint.

The basic steps are the same as above: your R Markdown file contains the base document that you edit, and the final stage²⁷ is to produce the actual presentation. So start by generating an R Markdown document as before. I'll use the same one as before, `test.Rmd`.

To make a Slidy presentation, you'll need to run a command like this:

```
pandoc -t slidy -s test.md -o test_pres.html
```

The input file is the Markdown file as before, and the output is going to be an HTML file. I'm giving it a different name so as not to get confused with the HTML file that Knit HTML produces. The extra stuff on the `pandoc` line this time is `-t slidy`, to tell `pandoc` that we want not just any old HTML file, but one in `slidy` format that will make a slideshow.²⁸ Also the `-s` means "make a stand-alone file", that is,²⁹ your presentation computer doesn't even need an Internet connection because everything needed for your presentation is in the file `test_pres.html`.

²⁶But it shouldn't be by now, since that belongs in Steps 1 and 2.

²⁷Well, something like next-to-final, because you're bound to want to reorganize things.

²⁸The behind-the-scenes stuff is done with Javascript and CSS.

²⁹If I understand this correctly.

I would have said that you produce your presentation after you've got an R Markdown file in the right format, but it doesn't really work like that. In particular, the usual biggest issue with making a presentation is not putting too much stuff on one slide, which you can't easily judge beforehand, so you have to try it and see, and fix it if you don't like it. So let's plan for an iterative approach.

The first steps are the same as before: sort out your R code and plan how you want to present it. Then you arrange an R Markdown file that contains your analysis and accompanying text, with *short* code chunks.³⁰

Now you can try out the slide show to see how it looks. In an R script window (or the Console), construct the command to run, and run it:

```
R> mycommand="pandoc -t slidy -s test.md -o test_pres.html"
R> system(mycommand)
```

Now open up `test_pres.html` in your web browser. You'll probably find that everything has come out on one page that goes off the bottom of the screen. No good for a presentation.

The way to make a new page in your presentation is to put in a top-level heading, which, in the R Markdown file, is one with a row of `=` underneath it. This will also serve as the title of the slide that it will begin. In my case, putting in a heading like `A plot` with a row of `=` beneath it, in the R Markdown file, will make this happen.

To see the effect of this change on the presentation, you have three things to do:

1. Click on Knit HTML (which, for you, has the effect of creating the Markdown file). You don't need to look at the Preview window, since this is not previewing your presentation. Just close it up when it appears.
2. Run `pandoc` as above to generate the `slidy` presentation.
3. Open the presentation in a web browser. Or refresh it if it already is open.³¹

If your presentation has more than one page, you'll see a little help on the screen. There will also be two links at the bottom, one to a Help screen, which shows you how to move from page to page,³² and a link to a Table of Contents for your presentation, which allows you to jump to any other page.

When you actually give your presentation, you will probably want to display it full-screen. F11 will accomplish that, in Firefox at least. When you're making

³⁰This is especially important here because there's only so much room on one slide for output.

³¹In Firefox, control-R will refresh the page.

³²Pretty much as you'd guess, and also as PowerPoint does it

your presentation, displaying things full-screen will show you how much will fit on one slide.

If you want other things in your presentation, put them in your R Markdown file (using the Markdown help as necessary). For example, bullet points:

```
Some bullet points
=====
```

```
* Here is the first point
* and this is the second
* Here, finally, is the third point.
```

Inserting an image (in the current folder):

```
![caption text](image.png)
```

You can also insert an image directly from the web like this:

```
![text](http://example.com/logo.png)
```

This at least requires you to be online while you’re running `pandoc`.³³

Oh, and that text “plot of chunk unnamed-chunk-2” under plots. You can improve that a bit by giving the code chunk producing the plot a label, like this:

```
```{r, "Scatterplot"}
plot(cars)
lines(lowess(cars))
```
```

Then, under the plot, you get a caption “Plot of chunk Scatterplot”, which is a bit nicer. I’d prefer to do away with “Plot of chunk” entirely, which you do by setting the figure caption directly:

```
```{r, fig.cap="Scatterplot of distance against speed"}
plot(cars)
lines(lowess(cars))
```
```

You do either of these by inserting a code chunk in the usual way, and then putting the cursor on the spot next to `r` and inserting the text yourself.

You might like to extract a number from the R output and incorporate it into your text. You do that like this:

```
We had `r nrow(cars)` cars altogether.
```

³³I don’t think you need to be online while showing the presentation, since `pandoc` will insert a copy of the image when run. I think. But don’t hold me to that.

The format is a backtick³⁴ followed by an `r` to start, and just a backtick to finish. In between goes any R code, though usually you'll have something that produces a number. If it's more than that, it belongs in a code chunk.

Sometimes you also want a typewriter-type font for names of things, like “this presentation was made using `pandoc`”, which you do in a similar way to the above:

```
This presentation was made using `pandoc`.
```

If you happen to want some code to be suitably displayed but not actually run, you insert a code chunk but then take out the `{r}`, so that it looks like this:

```
```
for (i in 1:10)
{
 print(i)
}
```
```

The programmers among you will be able to deduce what that does.

I mentioned earlier that you can put \LaTeX formulas and equations in an R Markdown script, and they get formatted nicely when you Knit HTML. This doesn't work right away in `pandoc` (and thus in your Slidy presentation or Word doc), but it can be fixed. If you have equations in your R Markdown file, you can get `pandoc` to handle them like this:

```
pandoc --mathjax -t slidy -s test.md -o test_pres.html
```

which I like best, though it's a bit slow. Or you can try

```
pandoc --latexmathml -t slidy -s test.md -o test_pres.html
```

which is quicker, though I don't like it so much.

7.6 Sweave and Statweave

I am not going to teach \LaTeX in this course, though if your writing has any mathematics in it, you should certainly consider learning and using it instead of Word or similar tools.

Recall that you can construct HTML or other output by using R Markdown. The idea there is that you insert a code chunk with some R code to run, and in the final document, that code is replaced with the output it produces. Sweave and Statweave use the same idea of code chunk being replaced by output, but they do it in a different way.

³⁴Probably the character hiding on the key to the left of `1` and below `Esc` on your keyboard.

Sweave works with R, and is handled by R Studio. To try it out, select File, New and R Sweave. At the top left, you'll see a skeleton \LaTeX file. You can put any \LaTeX commands or text in here. To insert a code chunk, click on Chunks at the top right of the window and select Insert Chunk (or press Control-Alt-I). Some peculiar symbols appear in green among the \LaTeX , with a grey background. These mark the beginning and end of the code chunk. In between them, you can put any R code you like. If you want to add any special options to a chunk, you put them in between the `<<` and `>>` of the `<<>>` at the top of the code chunk. The one you'll probably use most often is `FIG=TRUE`, which you need to put at the top of any chunk that has a graph in it.³⁵

When you want to turn your \LaTeX plus R code chunks into a PDF file, click on Compile PDF at the top of the window. This will first run Sweave, which replaces your code chunks by their output to produce a `.tex` file, and then runs `pdflatex` on that to produce a PDF file. A preview of that PDF file will appear (provided there were no errors).

Statweave handles R and SAS (and many other things besides). It doesn't work in R Studio, though, and you need to have SAS installed on your own computer. First you have to download the `statweave.jar` file from

<http://homepage.stat.uiowa.edu/~rlenth/StatWeave/>

and save it. You'll probably also want to grab the manual. Then follow the instructions in the manual to install it. (Unpacking the `.jar` file to the folder containing your document seems to work, if you don't have permissions to put it anywhere else.) Once you have it installed, you use your favourite editor³⁶ to create a file with an `.swv` extension that contains your \LaTeX and code chunks. Code chunks in Statweave look like ordinary \LaTeX environments with `begin` and `end`; an R code chunk is enclosed in an environment called `Rcode`, and a SAS code chunk is enclosed in an environment called `SAScode`. You can guess what kind of environment any other kind of code chunk lives in.

To make a PDF file, out of let's say a file called `myfile.swv`, first run `statweave myfile.swv`, which will produce `myfile.tex` with all the code chunks replaced by their output, and then run `pdflatex myfile.tex`. I'm using `statweave` to make these notes, and then I'm editing them to produce the slides with the \LaTeX package `beamer`.

As for Sweave, if you have a chunk with a graph in it, you have to begin it with `\begin{Rcode}{fig=TRUE}`.

³⁵There is a limit of one graph per chunk. If you have more than one graph, you need more than one chunk.

³⁶Mine is `emacs`.

Chapter 8

More examples

8.1 The windmill data

An engineer was interested in whether the amount of electricity generated by a windmill had anything to do with how strongly the wind was blowing. So the engineer took some measurements of the wind speed and DC current generated at various times.

We're going to assume that these times either were randomly selected, or they behave as if they were, since we are interested in generalizing from these data to something like "this type of windmill at all times".

Our research questions are something like these:

1. Is there a relationship between wind speed and current generated?
2. If so, what kind of relationship is it?
3. Can we model the relationship, in such a way that we can do predictions?

Let's do the analysis in R. Our data¹ look like this:

```
R> windmill=read.csv("windmill.csv",header=T)
R> head(windmill,n=10)
```

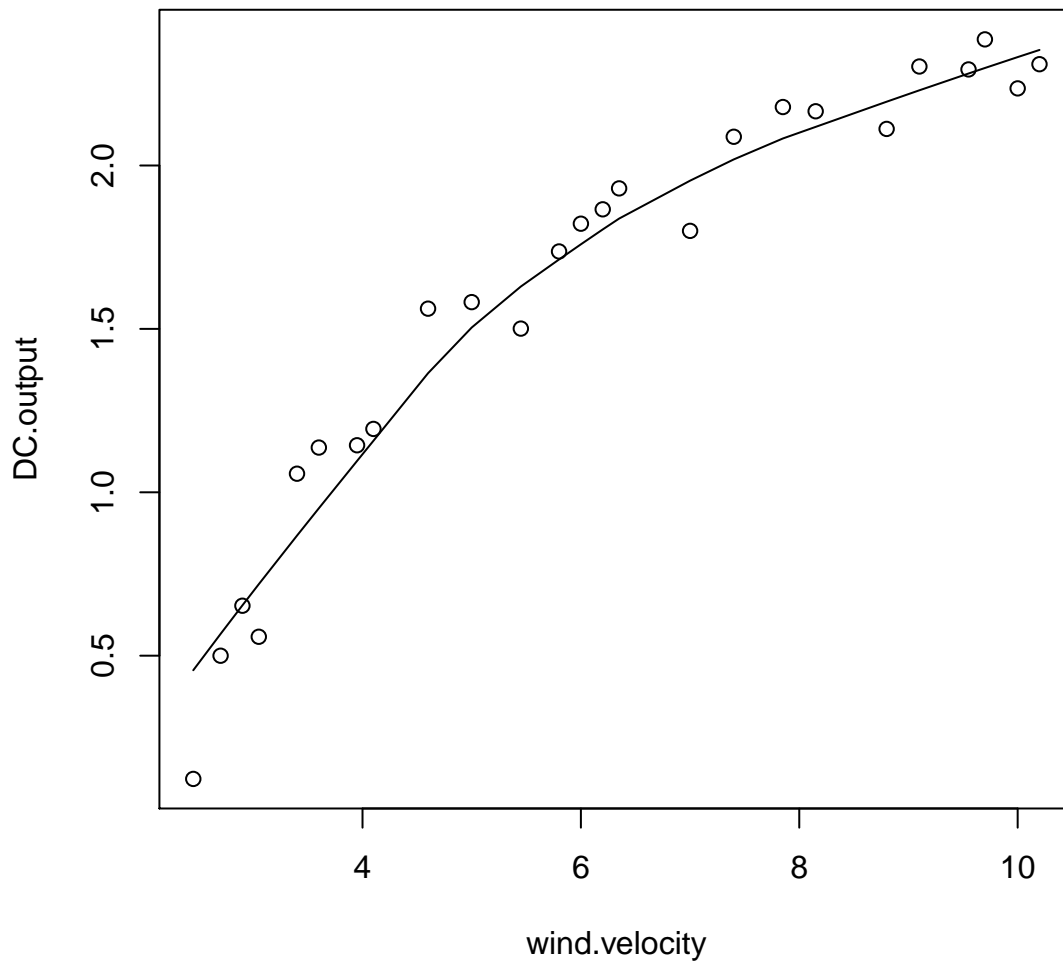
| | wind.velocity | DC.output |
|---|---------------|-----------|
| 1 | 5.00 | 1.582 |
| 2 | 6.00 | 1.822 |
| 3 | 3.40 | 1.057 |
| 4 | 2.70 | 0.500 |
| 5 | 10.00 | 2.236 |
| 6 | 9.70 | 2.386 |

¹Well, some of it.

| | | |
|----|------|-------|
| 7 | 9.55 | 2.294 |
| 8 | 3.05 | 0.558 |
| 9 | 8.15 | 2.166 |
| 10 | 6.20 | 1.866 |

We have two quantitative variables, so the techniques we will need to use are those related to regression. To start, we should have a look at a picture of the data, the appropriate picture being a scatterplot. We have two variables, `DC.output` and `wind.velocity`; the first of those is the output or response, and the second is the input or explanatory variable. So we draw our scatterplot with `DC.output` on the vertical scale, attaching the data frame first:

```
R> attach(windmill)
R> plot(DC.output~wind.velocity)
R> lines(lowess(DC.output~wind.velocity))
```



There is definitely a relationship: as the wind velocity goes up, the amount of electricity generated goes up as well. This is what you'd guess: more wind makes the blades of the windmill turn faster, and therefore makes the turbine turn faster, generating more electricity.

Now, the next question is whether the relationship is linear. To help assess that, I've added a lowess curve. You can see that the trend is more or less linear for a while, and then begins to curve downwards. So a straight line won't do so well here.

Before we get to fixing this, let's see what happens if we fit a straight line anyway:

```
R> windmill.lm=lm(DC.output~wind.velocity)
R> summary(windmill.lm)
```

Call:

```
lm(formula = DC.output ~ wind.velocity)
```

Residuals:

| | Min | 1Q | Median | 3Q | Max |
|--|----------|----------|---------|---------|---------|
| | -0.59869 | -0.14099 | 0.06059 | 0.17262 | 0.32184 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|---------------|----------|------------|---------|--------------|
| (Intercept) | 0.13088 | 0.12599 | 1.039 | 0.31 |
| wind.velocity | 0.24115 | 0.01905 | 12.659 | 7.55e-12 *** |

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

Residual standard error: 0.2361 on 23 degrees of freedom

Multiple R-squared: 0.8745, Adjusted R-squared: 0.869

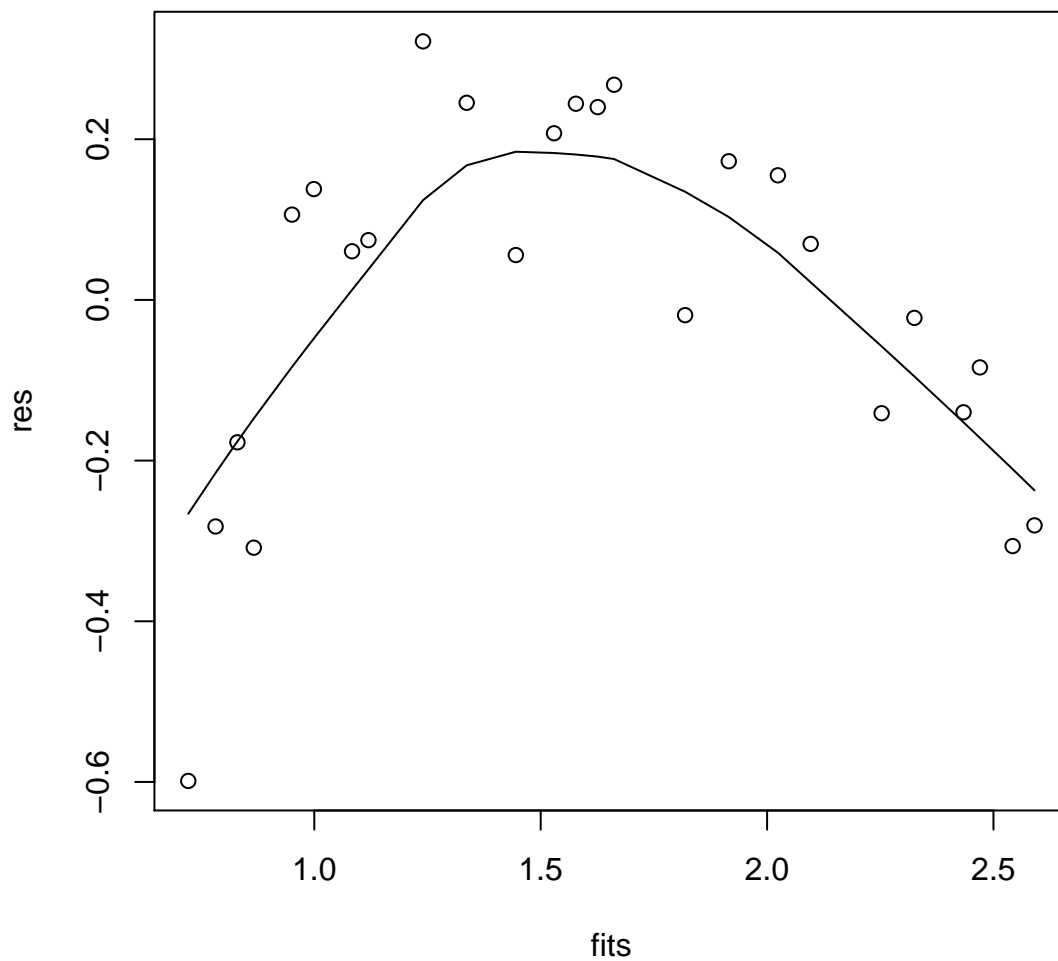
F-statistic: 160.3 on 1 and 23 DF, p-value: 7.546e-12

This looks pretty good, actually: `wind.velocity` is strongly significant, and the R-squared of around 87% is pretty high. You would think that a straight line is pretty good, and if you stopped here, you would be none the wiser.

How do we check whether a linear relationship is appropriate? Look at the residuals. Plotting the fitted model object gets you four plots, some of which will be a mystery to you until you take a Regression course. So I'll reproduce the first one of these, which is the most important: a plot of residuals against fitted values.² There's a little preparatory work. First we need to lay our hands on the fitted values and residuals. They are gotten directly from the fitted model object as shown. Then we plot them, and finally I've added a lowess curve again, for reasons that will become apparent in a moment.

```
R> fits=fitted(windmill.lm)
R> res=resid(windmill.lm)
R> plot(fits,res)
R> lines(lowess(fits,res))
```

²When you only have one explanatory variable, it makes no difference whether you plot residuals against the explanatory variable or against the fitted values, since they have a linear relationship between them. With more than one explanatory variable, though, it makes a difference. So we'll get into a good habit now.



Now, the reason for looking at a residual plot is that we should see *no pattern*. If we see a pattern, something is wrong. There ought to be no relationship between the residuals and anything else.

Is that the case here? Well, the residuals are mainly negative for small wind speeds, positive for middling wind speeds, and negative again for large wind speeds. The residuals, and perhaps more clearly the lowess, show a *curve*. That means that the relationship itself is not a line but a curve. So that's how we would have known to fit a curve even if we hadn't guessed to do so earlier.

There are³ three ways to fit a curved relationship that are within our abilities to do. One is to add a squared term in the explanatory variable, which we'll try first. The second is to transform the response, which here doesn't work very well, so we won't pursue it for this example. The third is to see if there is any math that applies, and to consider what that implies for the model-fitting.

All right, adding a squared term. Why does this help? Well, a curve of the form $y = ax^2 + bx + c$, called a **parabola**, is about the simplest form possible for a curve that isn't a straight line. For us statisticians, this is often the starting point if a straight line is no good.

To make this happen, there are a couple of ways, but I think this is easiest: define a new variable to be the wind velocity squared, and add that to the regression as another explanatory variable. Let's call this one the **parabola model**. It goes like this:

```
R> vel2=wind.velocity^2
R> windmill2.lm=lm(DC.output~wind.velocity+vel2)
R> summary(windmill2.lm)
```

Call:

```
lm(formula = DC.output ~ wind.velocity + vel2)
```

Residuals:

| | Min | 1Q | Median | 3Q | Max |
|--|----------|----------|---------|---------|---------|
| | -0.26347 | -0.02537 | 0.01264 | 0.03908 | 0.19903 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|---------------|-----------|------------|---------|--------------|
| (Intercept) | -1.155898 | 0.174650 | -6.618 | 1.18e-06 *** |
| wind.velocity | 0.722936 | 0.061425 | 11.769 | 5.77e-11 *** |
| vel2 | -0.038121 | 0.004797 | -7.947 | 6.59e-08 *** |

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

Residual standard error: 0.1227 on 22 degrees of freedom

Multiple R-squared: 0.9676, Adjusted R-squared: 0.9646

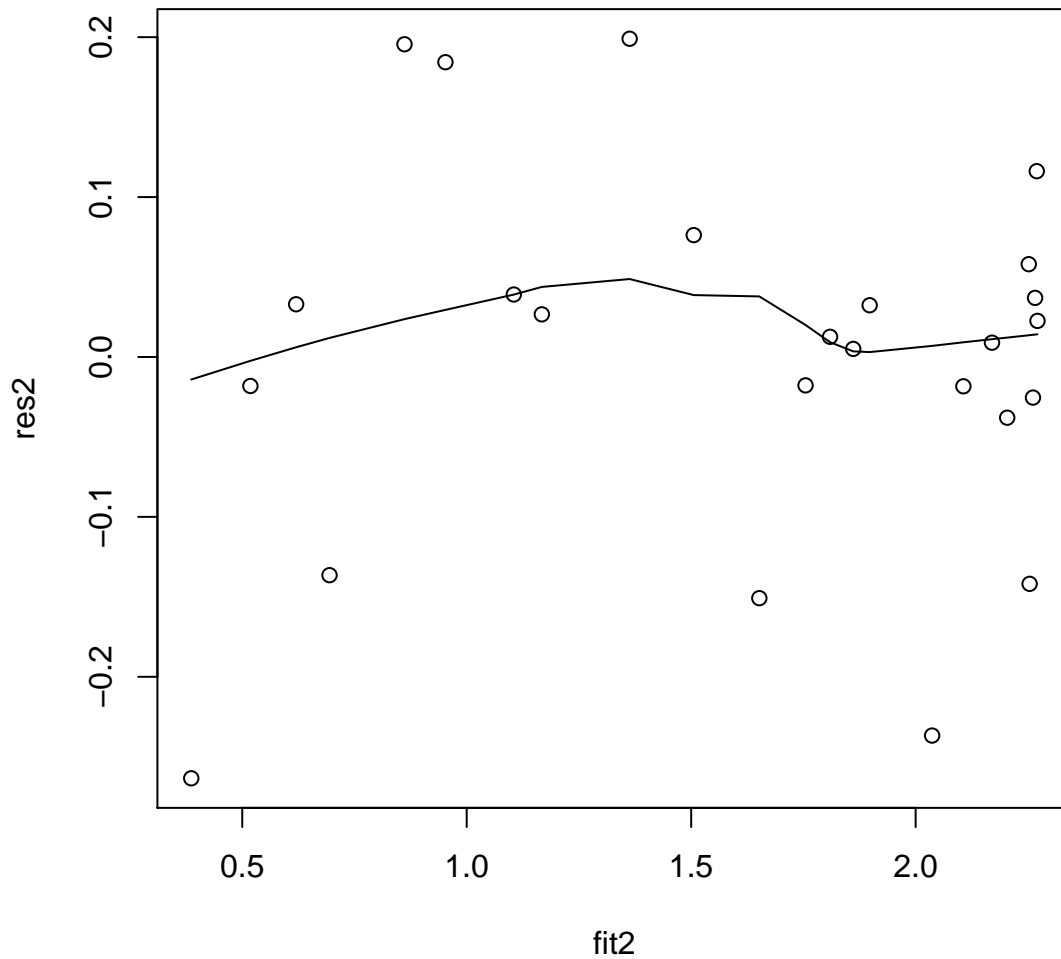
F-statistic: 328.3 on 2 and 22 DF, p-value: < 2.2e-16

The first check is to see whether the square term `vel2` is significant. It most certainly is, so this curve is an improvement over the straight line. As I like to say it, `vel2` has something to *add* to the regression containing just `wind.velocity`. But we should look at the residual plot for this regression, to make sure nothing is wrong. We get the fitted values and residuals from the fitted model object `windmill2.lm`, in the same way as before:

```
R> fit2=fitted(windmill2.lm)
```

³That I can think of.

```
R> res2=resid(windmill2.lm)
R> plot(fit2,res2)
R> lines(lowess(fit2,res2))
```

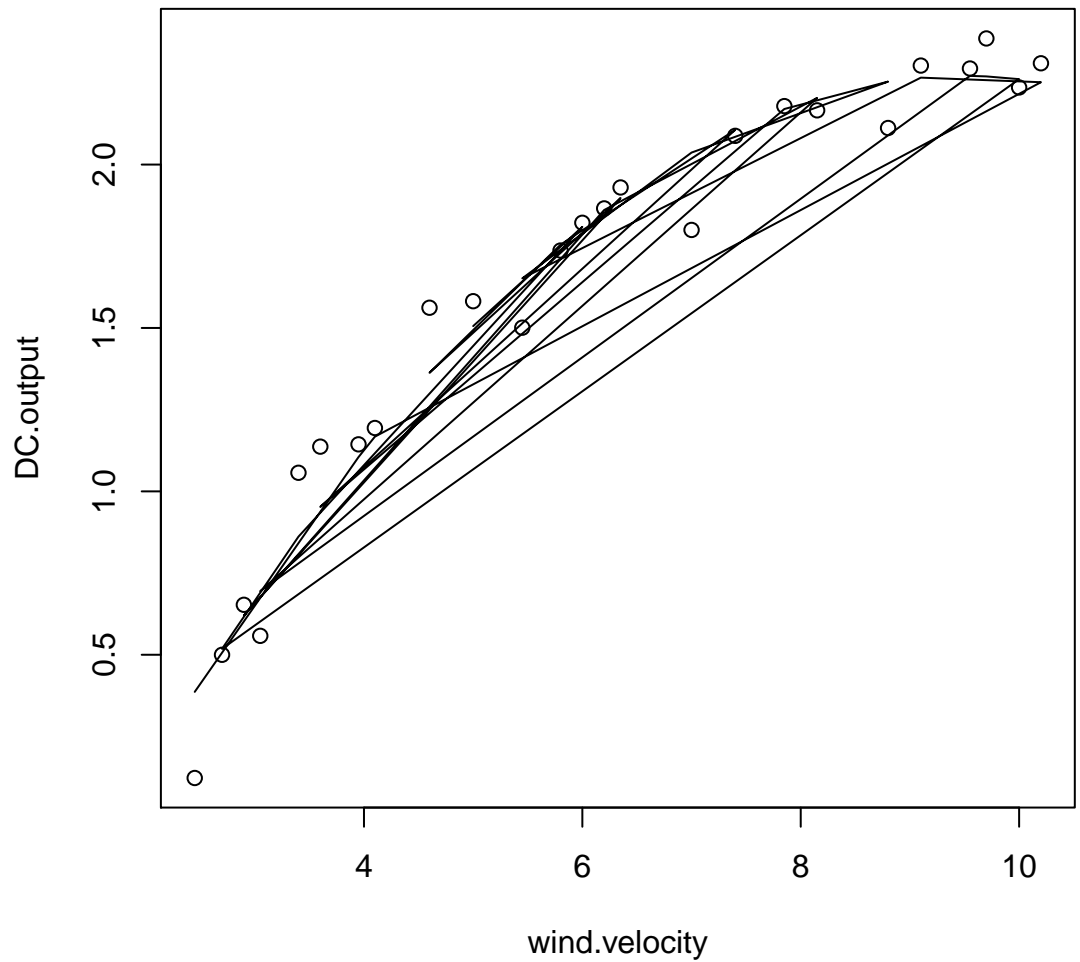


I'd say the residuals have a nice random pattern. Certainly the lowess curve is basically flat, with only a few inconsequential wiggles.

Let's take the original scatterplot and plot the fitted line and curve on it. Adding the line is as simple as `abline(windmill1.lm)`, but the curve is a bit more difficult to handle. Let's try and plot the curve on the scatterplot first to see

what goes wrong:

```
R> plot(DC.output~wind.velocity)
R> lines(wind.velocity,fit2)
```

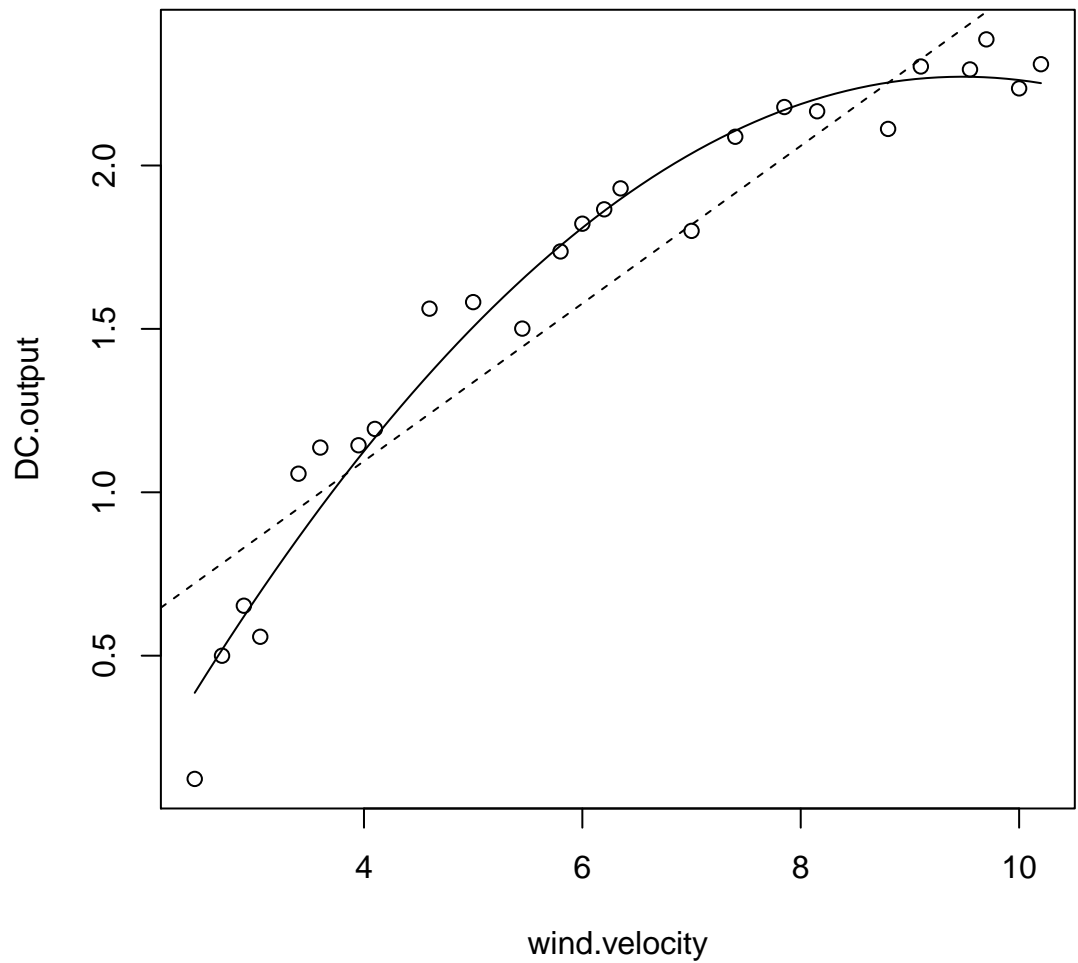


We got something like the smooth curve, but what are all those lines under it? The problem is that `lines` joins up the neighbouring data points in the file, but the `wind.velocity` values are not in order, so sometimes an observation with a low `wind.velocity` will be joined to one that has a high `wind.velocity`.

There's a way around this.⁴ The function `spline` puts a smooth curve through the (x, y) values you feed it, which in this case are going to be the observed wind velocities and the fitted `DC.output` values. This is different from `lowess`, which isn't guaranteed to go exactly through anything. Here's how it works, with the straight-line fit added on:

```
R> plot(DC.output~wind.velocity)
R> lines(spline(wind.velocity,fit2))
R> abline(windmill.lm,lty="dashed")
```

⁴Actually two. Another way is to sort the observations in order by `wind.velocity`.



The parabola model (solid) is clearly a better fit to the data than the line (dashed). It captures the increasing-then-levelling off trend nicely, at least for the data we have.

But there is one problem, which always happens with parabolas. Any curve $y = ax^2 + bx + c$, with a negative like this one, goes up, reaches a maximum, and then comes down again. This may or may not be reasonable in this case, but it illustrates that we should be very rigorous about only doing predictions in

the range of the data. Extrapolation is *very* dangerous with parabolas⁵ because they are just so darn *bendy*!

At this point, we go back to our engineer and report on our findings. In particular, we ask her what should happen as the wind velocity increases. “Well,” she says, scratching her head, “I think there’d be an upper limit to how much electricity is produced, since the blades of the windmill and the turbine can only spin so fast, but apart from that, the faster the wind blows, the higher the current output should be.”

This, in mathematical terms, sounds like an *asymptote*: a limit which is approached but never reached. Straight lines and parabolas don’t have asymptotes. But think of the function $y = 1/x$, with $x > 0$: as x gets bigger, y gets closer and closer to zero without ever reaching zero.

An idea is that we can use the function $y = 1/x$ to build a relationship with an asymptote in it. For example, if

$$y = a + b(1/x),$$

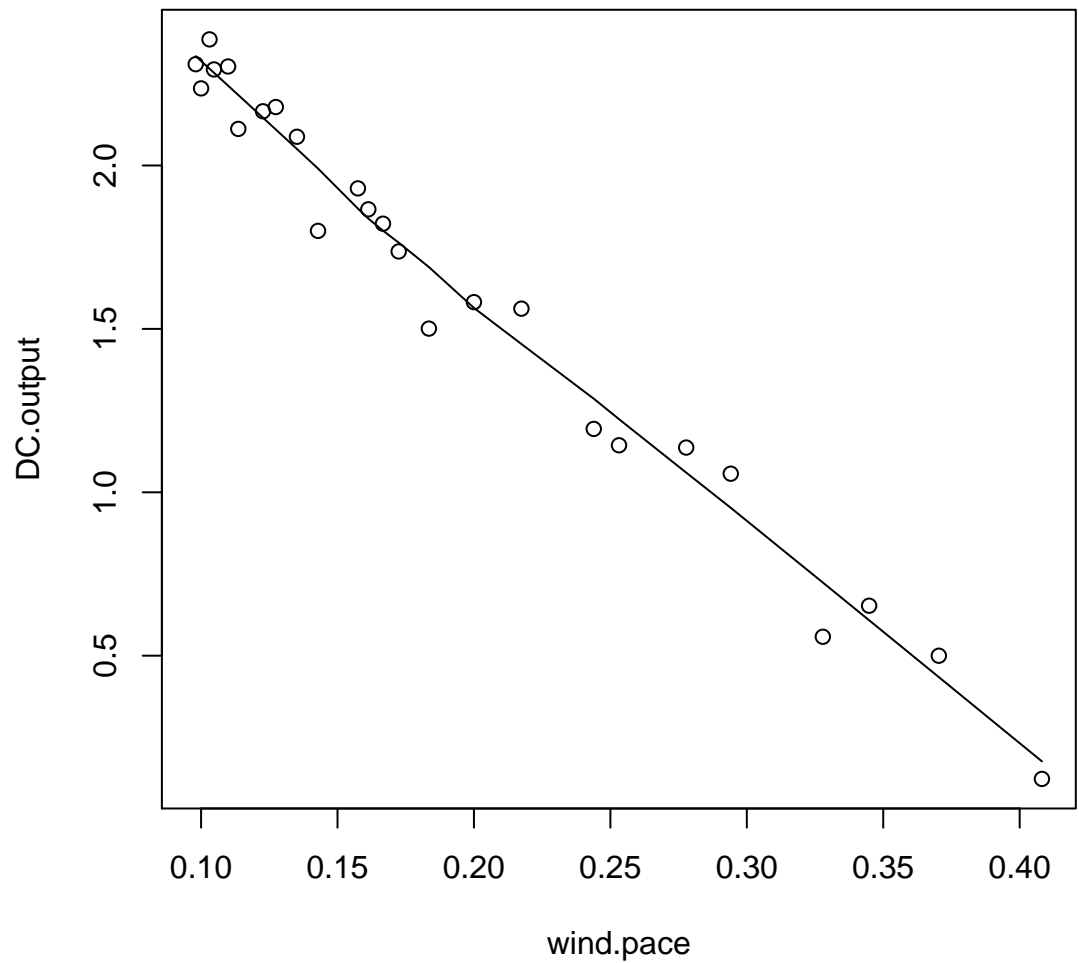
then as x gets larger, y gets closer and closer to a .⁶

So how you fit a model like $y = a + b(1/x)$? The same way as we fitted the parabola, really: define a new variable to be $1/x$ and use that as the explanatory variable in the regression. I didn’t have a good name for the inverse of velocity, until I thought of measuring how fast you walk. Say you walk 5 km/h; this means that you would take 12 minutes to walk 1 kilometre. In the walking world, time over distance is called **pace**. So, `wind.pace` it is. We should look at the scatterplot first and ask ourselves whether it is straight enough:

```
R> wind.pace=1/wind.velocity
R> plot(DC.output~wind.pace)
R> lines(lowess(DC.output~wind.pace))
```

⁵And other polynomials, too.

⁶The mathematicians among you may also be thinking of something like $y = a + be^{-x}$, which would also work, though it approaches its asymptote faster because e^{-x} approaches zero faster than $1/x$ does.



Doesn't get much straighter than that. The regression follows. Let's call this the **asymptote model**.⁷

```
R> windmill3.lm=lm(DC.output~wind.pace)
R> summary(windmill3.lm)
```

```
Call:
lm(formula = DC.output ~ wind.pace)
```

⁷Just because “asymptote” is a cool word.

Residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|----------|---------|---------|---------|
| -0.20547 | -0.04940 | 0.01100 | 0.08352 | 0.12204 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|-------------|----------|------------|---------|------------|
| (Intercept) | 2.9789 | 0.0449 | 66.34 | <2e-16 *** |
| wind.pace | -6.9345 | 0.2064 | -33.59 | <2e-16 *** |

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

Residual standard error: 0.09417 on 23 degrees of freedom

Multiple R-squared: 0.98, Adjusted R-squared: 0.9792

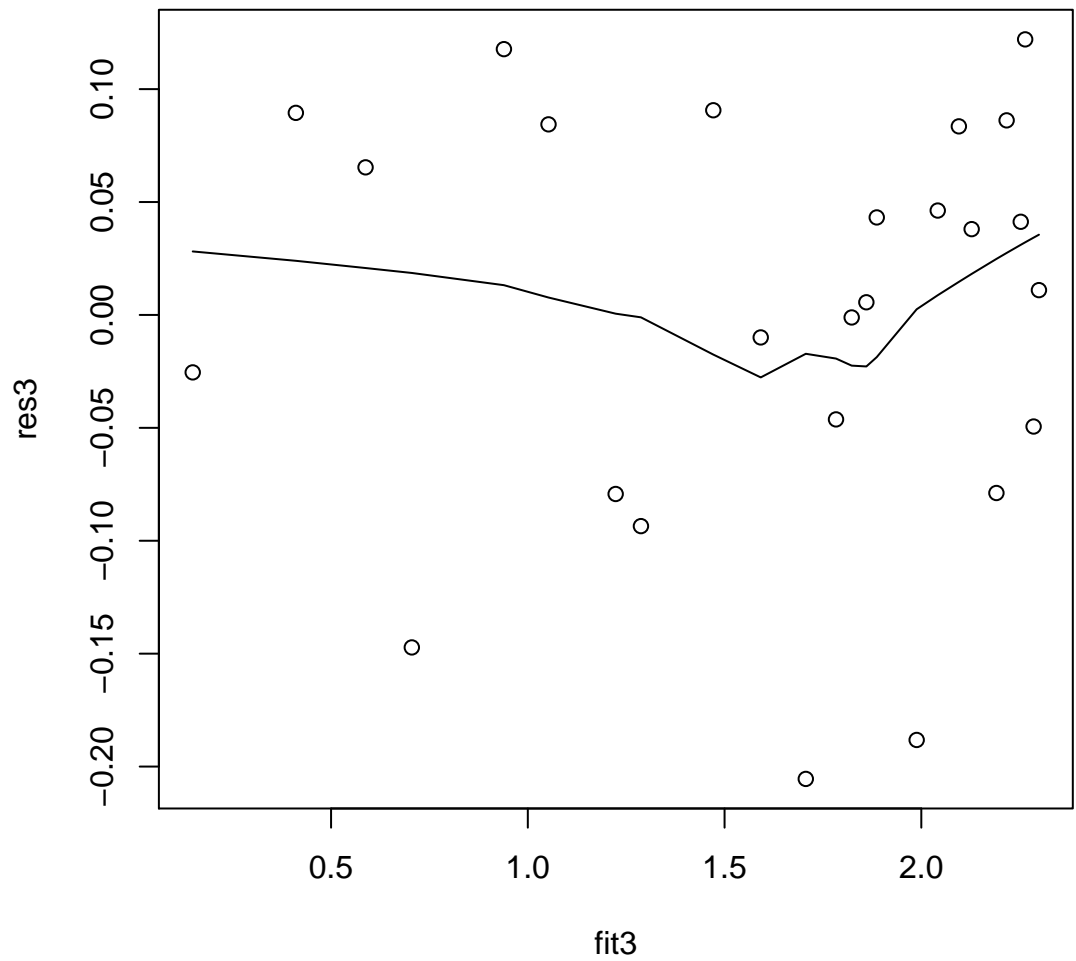
F-statistic: 1128 on 1 and 23 DF, p-value: < 2.2e-16

R-squared, at 98%, is even higher than for our parabola model. This is almost a perfect fit, and the benefit is that this model has an intercept and *one* slope, not *two* slopes as the parabola model had. We are looking for the *simplest model* that *adequately describes the data*⁸ and we seem to have found it.

Before we get too excited, though, we should check the residual plot:

```
R> fit3=fitted(windmill3.lm)
R> res3=resid(windmill3.lm)
R> plot(fit3,res3)
R> lines(lowess(fit3,res3))
```

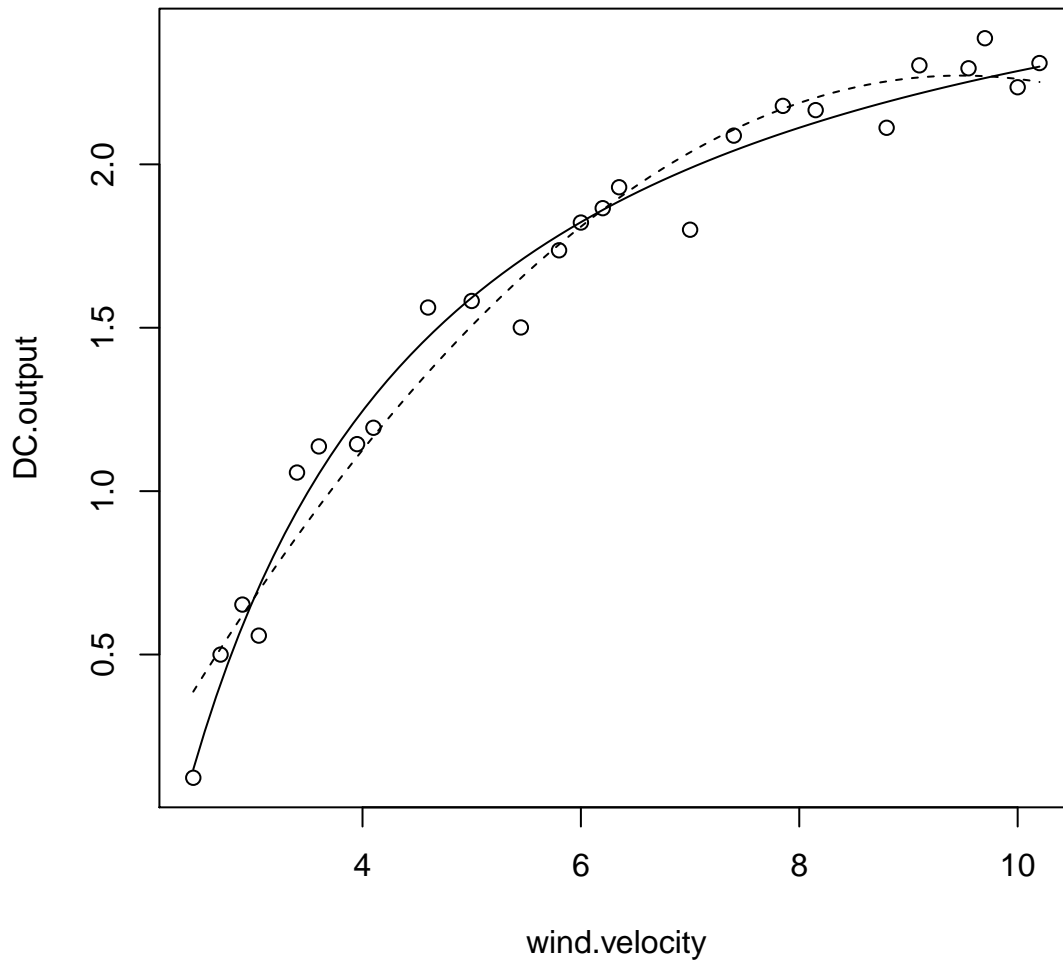
⁸This is called “Occam’s razor”.



I don't see too much to get worried about there. There are a lot of occasions where the predicted `DC.output` was near 2. What concerns me a little is that the residuals go up to about 0.10, but they go down all the way to -0.20 . In other words, the distribution of residuals is skewed, and not normal as we would like.

Let's plot the trend on the scatterplot, and see how it looks. I'll do the `spline` thing again, since the relationship is a curve, and I'll also plot our parabola again for comparison, dashed:

```
R> plot(DC.output~wind.velocity)
R> lines(spline(wind.velocity,fit3))
R> lines(spline(wind.velocity,fit2),lty="dashed")
```



There isn't much to choose between the two predictions, but the ones coming from 1 over `wind.velocity` are as good, and from a simpler model, so that's what I'd choose. Let's go back to the model `summary` and see what we have:

```
R> summary(windmill3.lm)
```

```

Call:
lm(formula = DC.output ~ wind.pace)

Residuals:
    Min       1Q   Median       3Q      Max
-0.20547 -0.04940  0.01100  0.08352  0.12204

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   2.9789     0.0449   66.34  <2e-16 ***
wind.pace     -6.9345     0.2064  -33.59  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.09417 on 23 degrees of freedom
Multiple R-squared:  0.98,    Adjusted R-squared:  0.9792
F-statistic: 1128 on 1 and 23 DF,  p-value: < 2.2e-16

```

The intercept in this model is about 3, which is the upper limit of `DC.output`. According to this model, the wind could get stronger than we observed, and there is still appreciably more electricity that could be generated. We're not close to the asymptote yet. You could eyeball a 95% confidence interval for the asymptote as $3 \pm 2(0.05)$: from 2.9 to 3.1.

The slope is about -7 . Why negative? Well, as the `wind.velocity` increases, the `wind.pace` goes down, and as the `wind.pace` goes down, the `DC.output` goes up. So this is all right. The actual number is hard to interpret, though.

Let's check back in with our research questions:

1. Is there a relationship between wind speed and current generated?
2. If so, what kind of relationship is it?
3. Can we model the relationship, in such a way that we can do predictions?

Our answers: yes, one with an asymptote, and yes (see `windmill3.lm`). Good. Job done.

I wanted to point out something else before we leave these data, though. Just because the parabola model and the asymptote model agree over the range of the data doesn't mean that they agree everywhere. Let's extend the range of the data a bit and see what we get. I'm going to try from 1 to 16. There are some points of R technique along the way, which I'll explain as we go.

The major problem is that we are predicting for *new* data, not just the values that we had in our data. R has a handy function called `predict` which will predict for new data based on a fitted model, but we have to get the new data set up right.

First, the new values of `wind.velocity`, 1 to 16 in steps of 0.5:


```
R> wv=seq(1,16,0.5)
R> wv

 [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0
[16] 8.5 9.0 9.5 10.0 10.5 11.0 11.5 12.0 12.5 13.0 13.5 14.0 14.5 15.0 15.5
[31] 16.0
```

Now, our parabola model contained two explanatory variables, `wind.velocity` and `vel2`, which was the velocity squared. We have to create a data frame with those two things as columns, filled with the values of velocity and velocity-squared from `wv`. That looks like this:

```
R> wind.vel.pred=data.frame(wind.velocity=wv,vel2=wv^2)
R> head(wind.vel.pred)

  wind.velocity  vel2
1           1.0  1.00
2           1.5  2.25
3           2.0  4.00
4           2.5  6.25
5           3.0  9.00
6           3.5 12.25
```

For our asymptote model, the input is `wind.pace`, which was 1 over wind velocity. So we need to make a data frame with a column called `wind.pace`, created from the values in `wv`:

```
R> wind.pace.pred=data.frame(wind.pace=1/wv)
R> head(wind.pace.pred)

  wind.pace
1 1.0000000
2 0.6666667
3 0.5000000
4 0.4000000
5 0.3333333
6 0.2857143
```

Now, `predict` needs two things: a fitted model object like `windmill2.lm` from above, and a data frame of values to predict from, like the `wind.vel.pred` that we just created:

```
R> pred.2=predict(windmill2.lm,wind.vel.pred)
R> pred.3=predict(windmill3.lm,wind.pace.pred)
```

`pred.2` contains predictions from the parabola model for our new data, and `pred.3` contains predictions from the asymptote model for the new data.

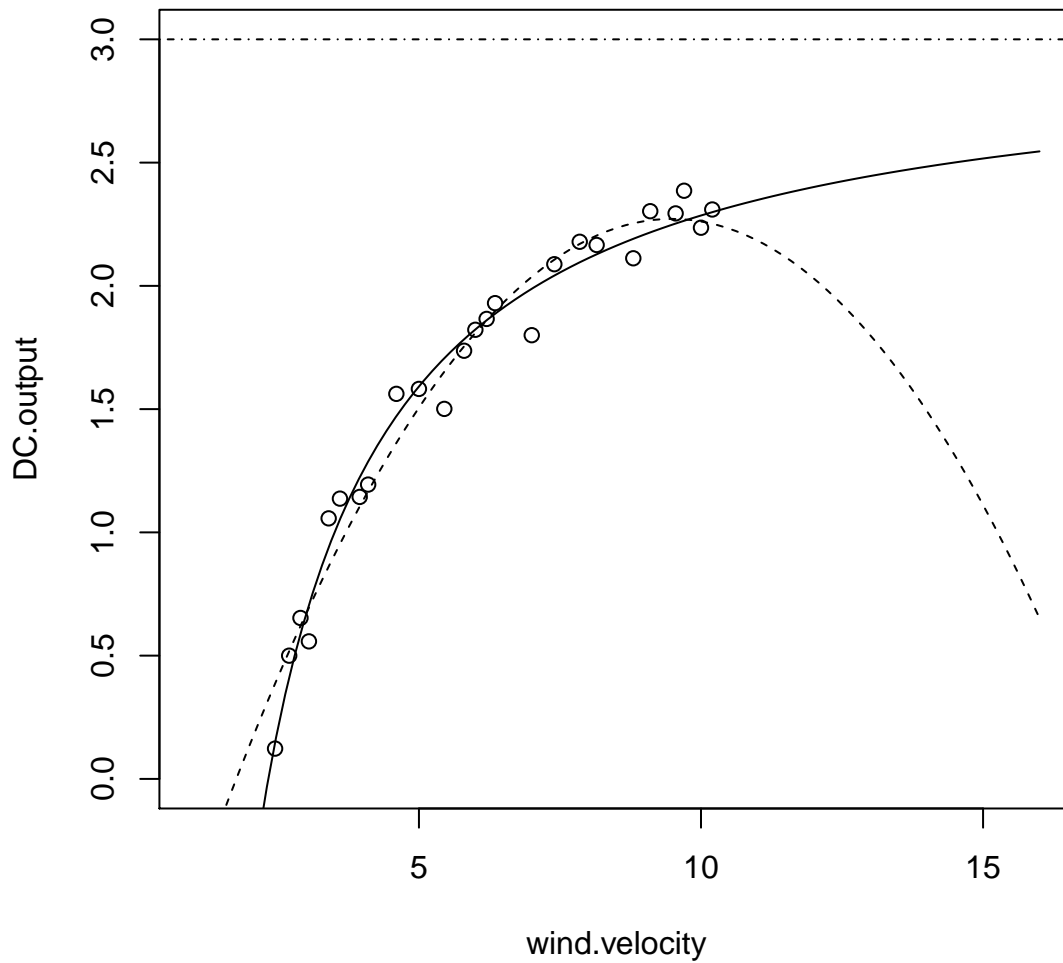
All right, now we want to plot these and see how they compare. First, let's plot the data, but we want to extend the horizontal scale to encompass all the values

in `wv`, and we want to make the vertical scale go up to 3 because that's what we thought the upper limit on `DC.output` was:

```
R> plot(DC.output~wind.velocity,xlim=range(wv),ylim=c(0,3))
```

Then, for each of the parabola model and the asymptote model, we make a smooth curve (using `spline`) out of the predictions. The asymptote model uses a solid line (default), while the parabola model uses a dashed line. Also, let's put in a horizontal line at 3, our guess at where the asymptote is. The whole thing looks like this:

```
R> plot(DC.output~wind.velocity,xlim=range(wv),ylim=c(0,3))
R> lines(spline(wv,pred.3))
R> lines(spline(wv,pred.2),lty="dashed")
R> abline(h=3,lty="dotdash")
```



Both models agree with the data, and more or less with each other, over the range of the data. But outside of the data, they disagree violently! For larger values of `wind.velocity`, it's the asymptote model that behaves reasonably. Not only is the asymptote model simpler, but it behaves more reasonably where we don't have data: it behaves as we think it should.

It struck me that the parabola model did the minimum necessary to fit the data: as soon as there was no data any more, it took off like a crazy thing!

Now, we haven't thought about what should happen as the `wind.velocity`

heads to zero. Logically, the `DC.output` should also be zero: if there is no wind, the windmill blades are not turning, and there should be no electricity generated. How do our models behave for zero `wind.velocity`?

For the parabola model, when `wind.velocity` is zero, the predicted `DC.output` is the intercept, which therefore should be zero. But it isn't:

```
R> summary(windmill12.lm)
```

Call:

```
lm(formula = DC.output ~ wind.velocity + vel2)
```

Residuals:

| | Min | 1Q | Median | 3Q | Max |
|--|----------|----------|---------|---------|---------|
| | -0.26347 | -0.02537 | 0.01264 | 0.03908 | 0.19903 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|---------------|-----------|------------|---------|--------------|
| (Intercept) | -1.155898 | 0.174650 | -6.618 | 1.18e-06 *** |
| wind.velocity | 0.722936 | 0.061425 | 11.769 | 5.77e-11 *** |
| vel2 | -0.038121 | 0.004797 | -7.947 | 6.59e-08 *** |

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

Residual standard error: 0.1227 on 22 degrees of freedom

Multiple R-squared: 0.9676, Adjusted R-squared: 0.9646

F-statistic: 328.3 on 2 and 22 DF, p-value: < 2.2e-16

It's negative, and significantly so. Not so good. What about the asymptote model?

```
R> summary(windmill13.lm)
```

Call:

```
lm(formula = DC.output ~ wind.pace)
```

Residuals:

| | Min | 1Q | Median | 3Q | Max |
|--|----------|----------|---------|---------|---------|
| | -0.20547 | -0.04940 | 0.01100 | 0.08352 | 0.12204 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|-------------|----------|------------|---------|------------|
| (Intercept) | 2.9789 | 0.0449 | 66.34 | <2e-16 *** |
| wind.pace | -6.9345 | 0.2064 | -33.59 | <2e-16 *** |

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

Residual standard error: 0.09417 on 23 degrees of freedom

```
Multiple R-squared:  0.98,          Adjusted R-squared:  0.9792  
F-statistic:  1128 on 1 and 23 DF,  p-value: < 2.2e-16
```

When `wind.velocity` is zero, `wind.pace` is infinite! Therefore the predicted `DC.output` is minus infinity! That makes no sense at all.

A way of fixing this would be to model the *log* of `DC.output` as a function of `wind.pace`. Then, as `wind.velocity` heads to zero, `wind.pace` would head to infinity, and *log* of `DC.output` would head to minus infinity, and `DC.output` itself would be heading to zero. But that might spoil the nice straight relationship between `DC.output` and `wind.pace` otherwise.⁹

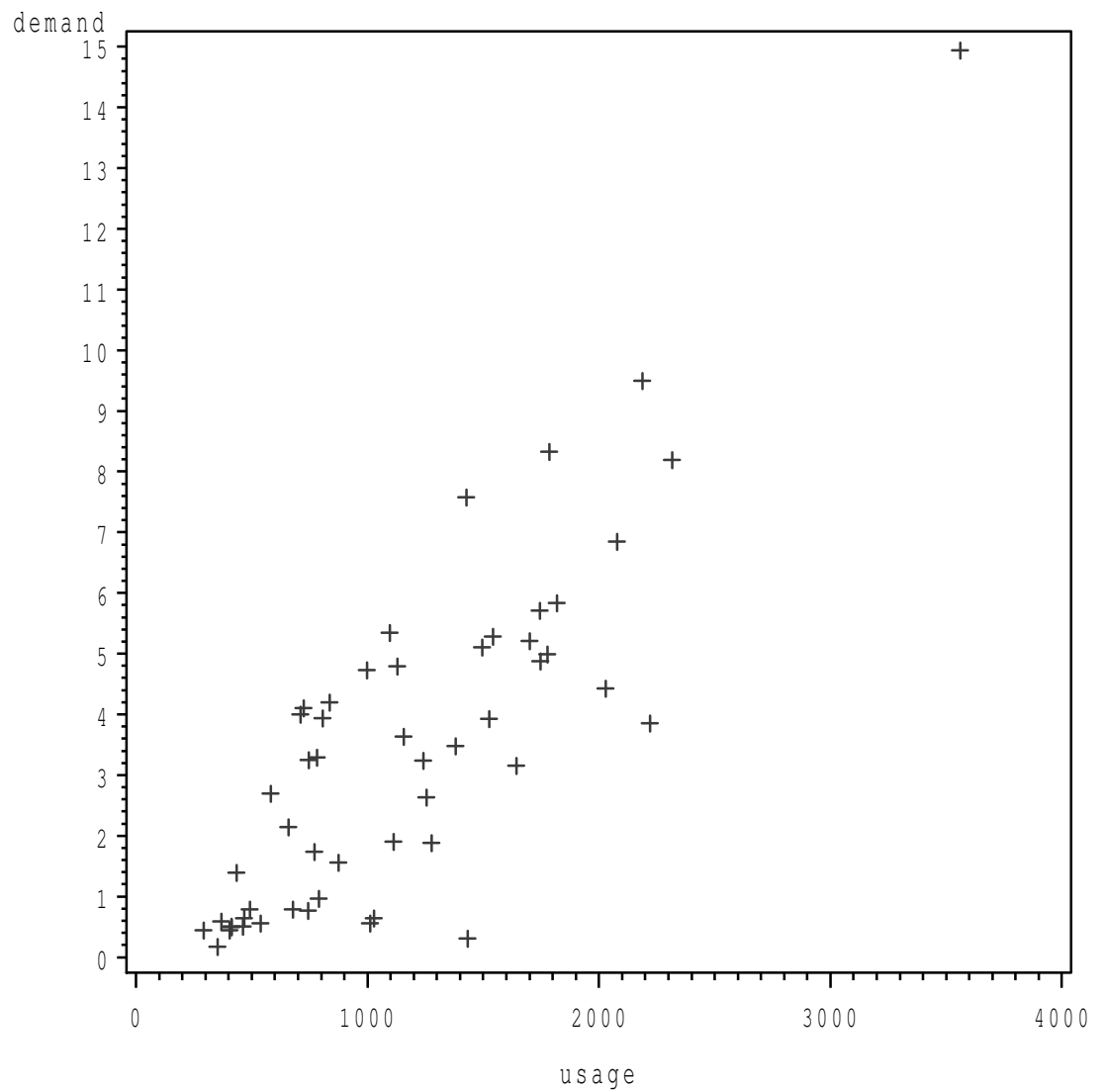
Data analyses rarely reach a thoroughly satisfactory conclusion; often, you have to settle for a model that works reasonably well. But there's always something else you can try, and at some point you have to say "here I stop".

8.2 Another regression example in SAS

An electric utility company is interested in relating peak-hour demand (kW) to total energy usage (kWh) during a month. This is an important planning problem, because the generation system must be large enough to meet the maximum demand imposed. Data were obtained from 53 residential customers for the month of August. Let's read in the data and draw a scatterplot, in SAS this time:

```
SAS> data util;  
SAS>   infile 'utility.txt';  
SAS>   input usage demand;  
SAS>  
SAS> proc gplot;  
SAS>   plot demand*usage;
```

⁹It does. I tried it.



This looks straight enough, though I'm concerned by the customer with high usage and high demand. This could be influential on where a regression line goes.

Let's try fitting a regression anyway:

```
SAS> proc reg;
SAS>   model demand=usage;
```

The REG Procedure
Model: MODEL1

Dependent Variable: demand
 Number of Observations Read 53
 Number of Observations Used 53

| Analysis of Variance | | | | | |
|----------------------|----------|----------------|-------------|---------|--------|
| Source | DF | Sum of Squares | Mean Square | F Value | Pr > F |
| Model | 1 | 302.63314 | 302.63314 | 121.66 | <.0001 |
| Error | 51 | 126.86602 | 2.48757 | | |
| Corrected Total | 52 | 429.49915 | | | |
| Root MSE | 1.57720 | R-Square | 0.7046 | | |
| Dependent Mean | 3.41321 | Adj R-Sq | 0.6988 | | |
| Coeff Var | 46.20882 | | | | |

| Parameter Estimates | | | | | |
|---------------------|----|--------------------|----------------|---------|---------|
| Variable | DF | Parameter Estimate | Standard Error | t Value | Pr > t |
| Intercept | 1 | -0.83130 | 0.44161 | -1.88 | 0.0655 |
| usage | 1 | 0.00368 | 0.00033390 | 11.03 | <.0001 |

This looks pretty good, with an R-squared of 70% and a strongly significant slope. But we know we are not done yet: we should take a look at the residuals. There are a couple of ways to do that. One way is to obtain an output data set with the residuals and fitted values in it, and then plot these. This is kind of a SAS way of doing things: an output data set contains extra stuff that didn't appear on the output. Here's how it goes:

```
SAS> proc reg;
SAS>   model demand=usage;
SAS>   output out=util2 r=res p=fit;
```

The REG Procedure

Model: MODEL1

Dependent Variable: demand
 Number of Observations Read 53
 Number of Observations Used 53

| Analysis of Variance | | | | | |
|----------------------|---------|----------------|-------------|---------|--------|
| Source | DF | Sum of Squares | Mean Square | F Value | Pr > F |
| Model | 1 | 302.63314 | 302.63314 | 121.66 | <.0001 |
| Error | 51 | 126.86602 | 2.48757 | | |
| Corrected Total | 52 | 429.49915 | | | |
| Root MSE | 1.57720 | R-Square | 0.7046 | | |
| Dependent Mean | 3.41321 | Adj R-Sq | 0.6988 | | |

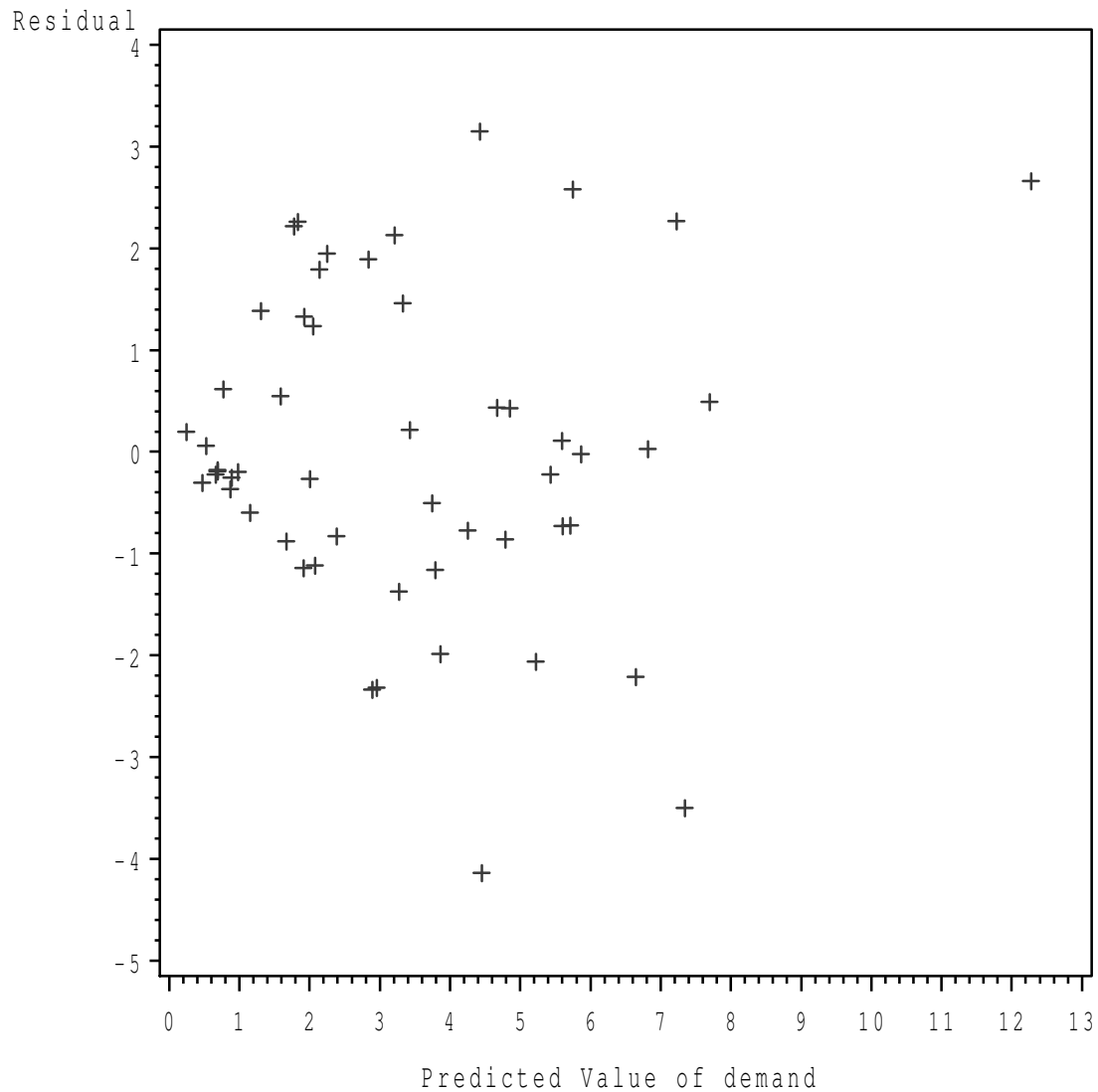
Coeff Var 46.20882

| | | Parameter Estimates | | | |
|-----------|----|---------------------|------------|---------|---------|
| | | Parameter | Standard | | |
| Variable | DF | Estimate | Error | t Value | Pr > t |
| Intercept | 1 | -0.83130 | 0.44161 | -1.88 | 0.0655 |
| usage | 1 | 0.00368 | 0.00033390 | 11.03 | <.0001 |

This will create a data set with the residuals (called **res**) and the fitted values (called **fit**) in it, along with all the original variables. You can verify that via `proc print`, should you wish. Or you can take my word for it.

Now we make a scatterplot of the newly-calculated residuals against the fitted values:

```
SAS> proc gplot;
SAS>   plot res*fit;
```

There doesn't seem to be a trend, but there is a “fanning-out”: as the fitted value gets larger, the residuals tend to get further away from zero — that is, larger *in size* even if their mean is still zero. This is not good, because one of the assumptions hiding behind regression is that the variability around the line should be the same all the way along.

The usual fix for this kind of thing is to transform the response variable. The best way to do *that* is to consult with the person who brought you the data and figure out what would be sensible. Only we can't do that here, and I have no idea what transformation would be sensible.

So let's try a "Box-Cox transformation", where we let SAS figure out what kind of transformation would be good. The scale is the same one as the "ladder of powers" that you might have seen in STAB22, where the number λ that says which transformation is best is a power, except $\lambda = 0$, which means "take logs". SAS has a `proc` for doing this, called `proc transreg`, which does other things as well, so we have to invoke it as below. We are contemplating a transformation of `demand` without changing `usage`, hence the `model` line. I also took us back to our original data set, since the residuals, fitted values and such in `util2` would just get in the way:

```
SAS> proc transreg data=util;
SAS>   model boxcox(demand)=identity(usage);
```

The TRANSREG Procedure

Box-Cox Transformation Information for demand

| Lambda | R-Square | Log Like |
|--------|----------|-----------|
| -3.00 | 0.05 | -285.806 |
| -2.75 | 0.05 | -256.529 |
| -2.50 | 0.06 | -227.764 |
| -2.25 | 0.07 | -199.652 |
| -2.00 | 0.09 | -172.389 |
| -1.75 | 0.12 | -146.248 |
| -1.50 | 0.15 | -121.590 |
| -1.25 | 0.20 | -98.852 |
| -1.00 | 0.25 | -78.489 |
| -0.75 | 0.32 | -60.876 |
| -0.50 | 0.39 | -46.203 |
| -0.25 | 0.46 | -34.473 |
| 0.00 | 0.53 | -25.618 |
| 0.25 | 0.59 | -19.687 |
| 0.50 + | 0.65 | -17.023 < |
| 0.75 | 0.69 | -18.287 * |
| 1.00 | 0.70 | -24.150 |
| 1.25 | 0.70 | -34.686 |
| 1.50 | 0.68 | -49.133 |
| 1.75 | 0.64 | -66.342 |
| 2.00 | 0.60 | -85.312 |
| 2.25 | 0.56 | -105.376 |
| 2.50 | 0.52 | -126.141 |
| 2.75 | 0.49 | -147.382 |
| 3.00 | 0.46 | -168.971 |

< - Best Lambda

* - 95% Confidence Interval

+ - Convenient Lambda

SAS marks the "best" λ , which is the one where Log Like is maximum (least negative). This value is $\lambda = 0.5$, or square root. The * marks the upper end of

a confidence interval for λ .¹⁰

When we're choosing a transformation guided by `proc transreg`, consider that you will have to justify the transformation you use.¹¹ So it's a good idea to choose a "round number" like 0.5 (square root) rather than 0.59, which says "raise the `demand` values to the 0.59 power". So let's try the square root transformation and see how that looks.

Creating new variables belongs in a `data` step. But we already read in the data (into `util`), so we don't need to read the data file again. The `set` statement below builds the new data set `trans` using the variables in `util` as a starting point, and then adds a new variable that is the square root of `demand`:

```
SAS> data trans;
SAS>   set util;
SAS>   rtdemand=sqrt(demand);
```

Now, `trans` is the most recently-created data set, so that will be the default one when we run `proc reg` again:

```
SAS> proc reg;
SAS>   model rtdemand=usage;
SAS>   output out=outrt r=res p=fit;
```

The REG Procedure

Model: MODEL1

Dependent Variable: rtdemand

Number of Observations Read 53

Number of Observations Used 53

Analysis of Variance

| Source | DF | Sum of Squares | Mean Square | F Value | Pr > F |
|-----------------|----|----------------|-------------|---------|--------|
| Model | 1 | 20.25850 | 20.25850 | 94.08 | <.0001 |
| Error | 51 | 10.98219 | 0.21534 | | |
| Corrected Total | 52 | 31.24069 | | | |

| | | | |
|----------------|----------|----------|--------|
| Root MSE | 0.46404 | R-Square | 0.6485 |
| Dependent Mean | 1.68040 | Adj R-Sq | 0.6416 |
| Coeff Var | 27.61503 | | |

Parameter Estimates

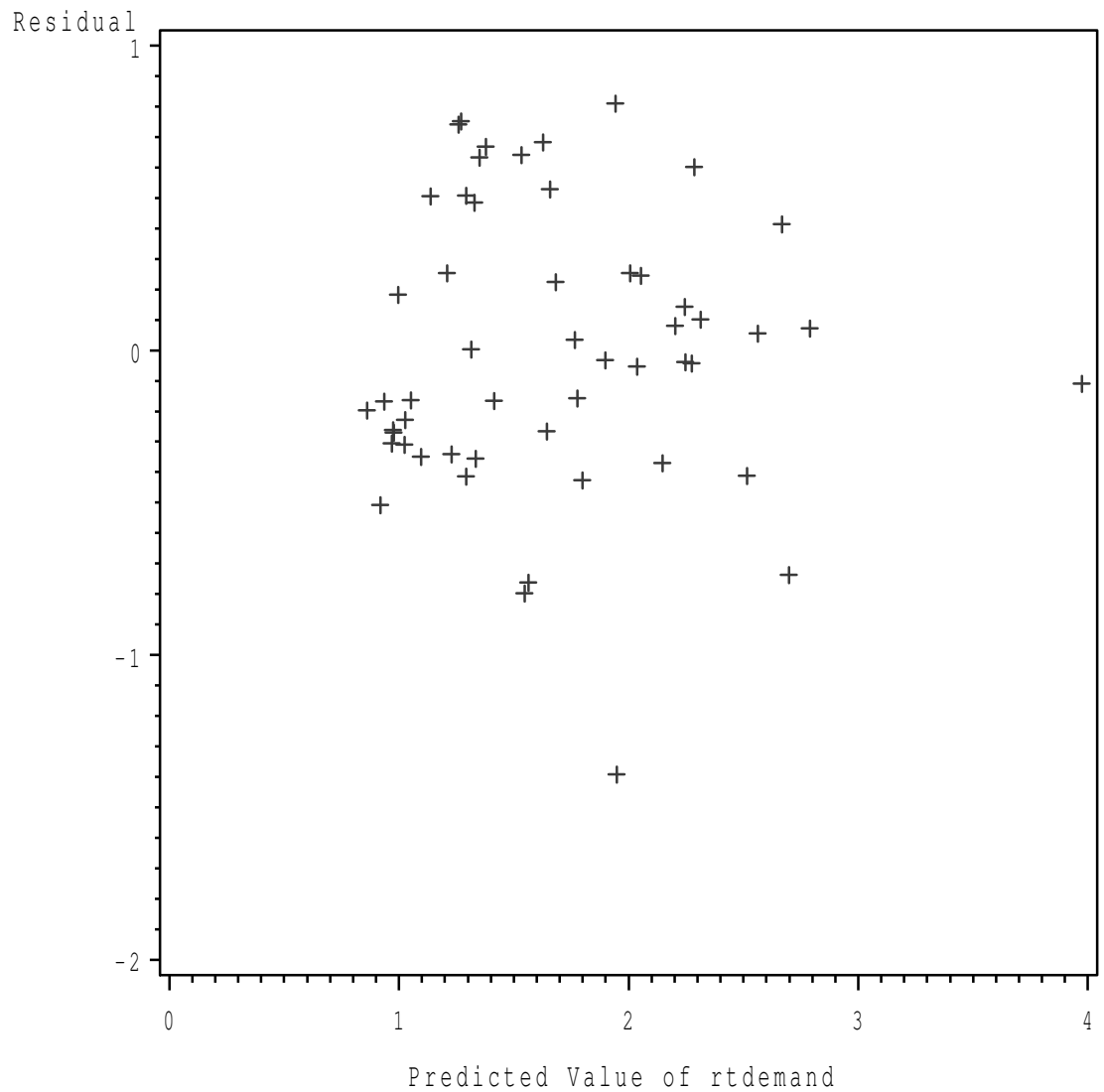
| Variable | DF | Parameter Estimate | Standard Error | t Value | Pr > t |
|-----------|----|--------------------|----------------|---------|---------|
| Intercept | 1 | 0.58223 | 0.12993 | 4.48 | <.0001 |
| usage | 1 | 0.00095286 | 0.00009824 | 9.70 | <.0001 |

¹⁰The lower end got hidden under the < on the 0.5 line.

¹¹When you come to write the report.

Does the residual plot look better now? Again, `outtrt` is the most recently-created data set:

```
SAS> proc gplot;  
SAS> plot res*fit;
```

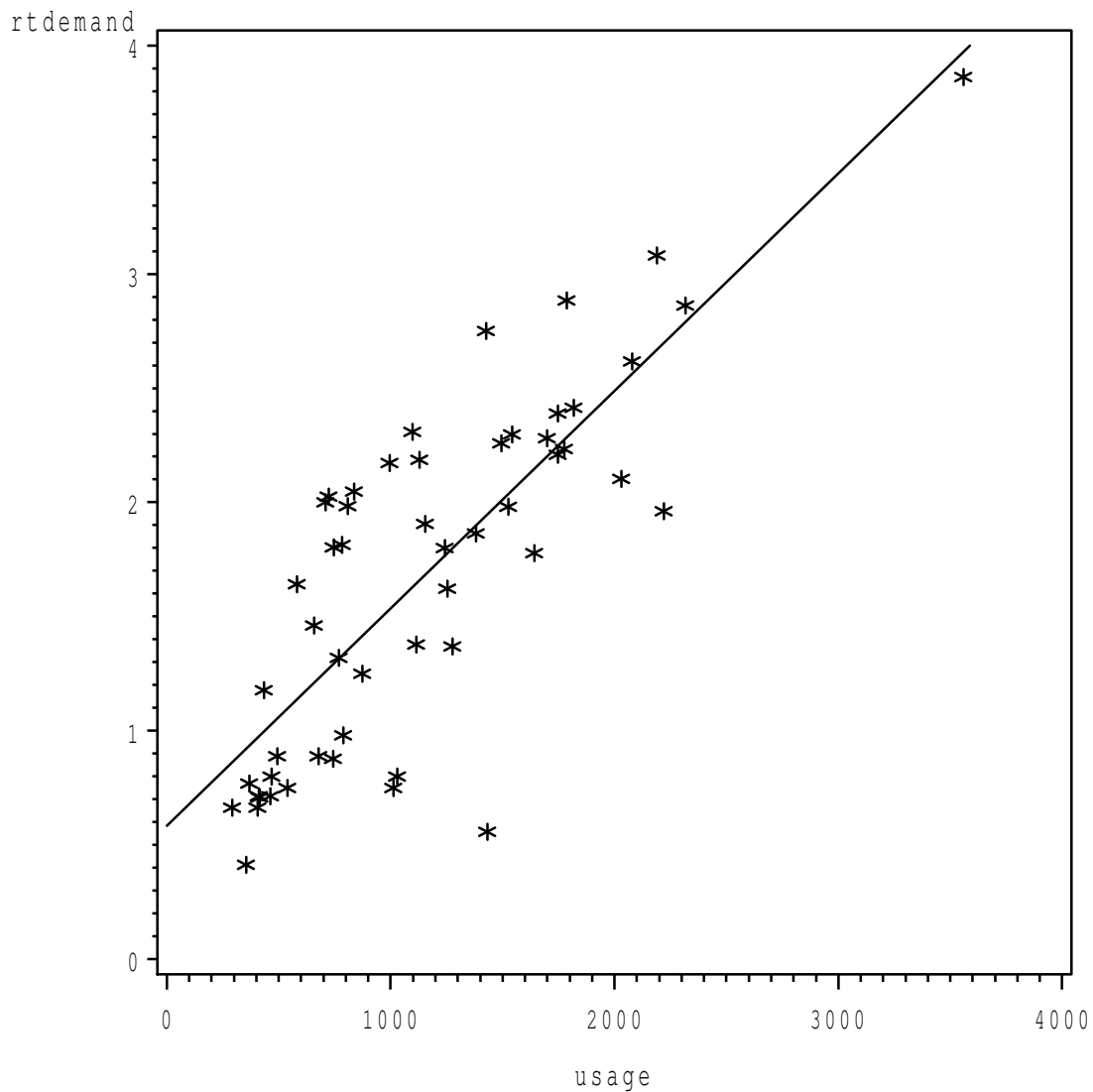


That looks a whole lot better. No trends, and constant variability all the way along. I think we can trust this regression.

The output from `proc reg` shows that R-squared has actually gone down a bit (from 70% to 65%), but it's better to have a lower R-squared from a regression

that we can trust, than a higher one from one we can't. The slope for `usage` may be tiny, but it is strongly significant. Let's see if we can make a scatterplot of `square-root-demand` against `usage` and put the regression line on it, using ideas from Section 10.7.2:

```
SAS> symbol1 c=black i=rl l=1 v=star;  
SAS> proc gplot;  
SAS>   plot rtdemand*usage;
```



Pretty good. The one observation with high `usage` might be an influential point, but it seems to lie pretty much on the trend, so I don't think we need to worry

about it.

When we have transformed the response variable, we have to think a bit more carefully about predictions. Let's start with a usage of 1000. If we just plug this into the regression equation we get this:¹²

```
R> int=0.58223
R> slope=0.00095286
R> pred=int+slope*1000
R> pred

[1] 1.53509
```

This is a prediction all right, but it's a prediction of the response variable in the regression, which was the *square root* of **demand**. To get a predicted actual demand, we need to undo the transformation. Undoing a square root is *squaring*, so the predicted demand is

```
R> pred^2

[1] 2.356501
```

Let's do 1000, 2000 and 3000 now, taking advantage of R's vector calculation to do all three at once:

```
R> usage=c(1000,2000,3000)
R> rt.demand=int+slope*usage
R> demand=rt.demand^2
R> demand

[1] 2.356501 6.189895 11.839173
```

Transformations are non-linear changes. The way that shows up here is: the **usage** values are equally spaced, but the predicted demand values are not. There's a larger gap between the second and third than between the first and second.¹³

8.3 The asphalt data

***** intro here

The data set contains several variables:

pct.a.surf The percentage of asphalt in the surface layer

pct.a.base The percentage of asphalt in the base layer

fines The percentage of fines in the surface layer

¹²Using R as a calculator.

¹³The **rt.demand** values *are* equally spaced.

voids The percentage of voids in the surface layer

rut.depth The change in rut depth per million vehicle passes

viscosity The viscosity of the asphalt

run There were two data collection periods. **run** is 1 for run 1, 0 for run 2.

The variable **rut.depth** is the response, and we are interested in whether it depends on any of the other variables, and if so, how. I'm doing to do this analysis in R.

Let's start by reading in the data:

```
R> asphalt=read.table("asphalt.txt",header=T)
R> head(asphalt)
```

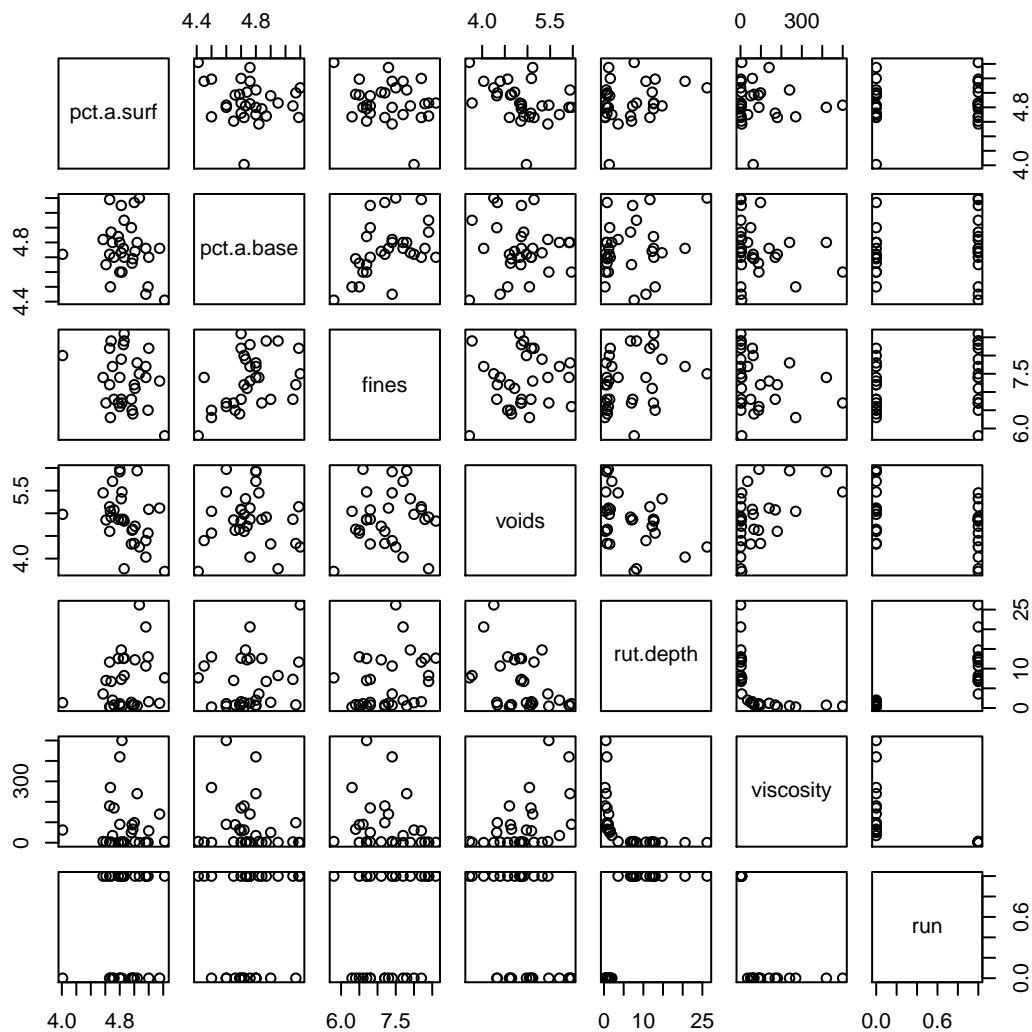
| | pct.a.surf | pct.a.base | finer | voids | rut.depth | viscosity | run |
|---|------------|------------|-------|-------|-----------|-----------|-----|
| 1 | 4.68 | 4.87 | 8.4 | 4.916 | 6.75 | 2.8 | 1 |
| 2 | 5.19 | 4.50 | 6.5 | 4.563 | 13.00 | 1.4 | 1 |
| 3 | 4.82 | 4.73 | 7.9 | 5.321 | 14.75 | 1.4 | 1 |
| 4 | 4.85 | 4.76 | 8.3 | 4.865 | 12.60 | 3.3 | 1 |
| 5 | 4.86 | 4.95 | 8.4 | 3.776 | 8.25 | 1.7 | 1 |
| 6 | 5.16 | 4.45 | 7.4 | 4.397 | 10.67 | 2.9 | 1 |

We have a bunch of quantitative variables here, with one response variable, so we'll use a multiple regression approach here. There are some issues that come up here that don't come up in "simple" regression, but we'll deal with those as we go along.¹⁴

We have too many variables to do a scatterplot, so let's do *lots*! If you plot a data frame***** ref, you get this:

```
R> plot(asphalt)
```

¹⁴If you've taken STAB27, you'll probably be familiar with these.



This is called a **pairs plot**: it contains the scatter plots of all possible pairs of variables. It's a way of seeing how things seem to be related.¹⁵

Look along the fifth row of the pairs plot to see how things are related to the response variable `rut.depth`. The relationships with `rut.depth` are mostly weak at best. The strongest ones appear to be the last two: there's a downward

¹⁵If you're working in R Studio, this plot might come out too small to see properly. You can click on the Zoom button to make a new window, with the whole plot, bigger. Maximize this window, and you might be able to see what's going on.

but very non-linear trend with `viscosity`, and the `rut.depth` is much bigger (and more variable) when `run=0` than when `run=1`.

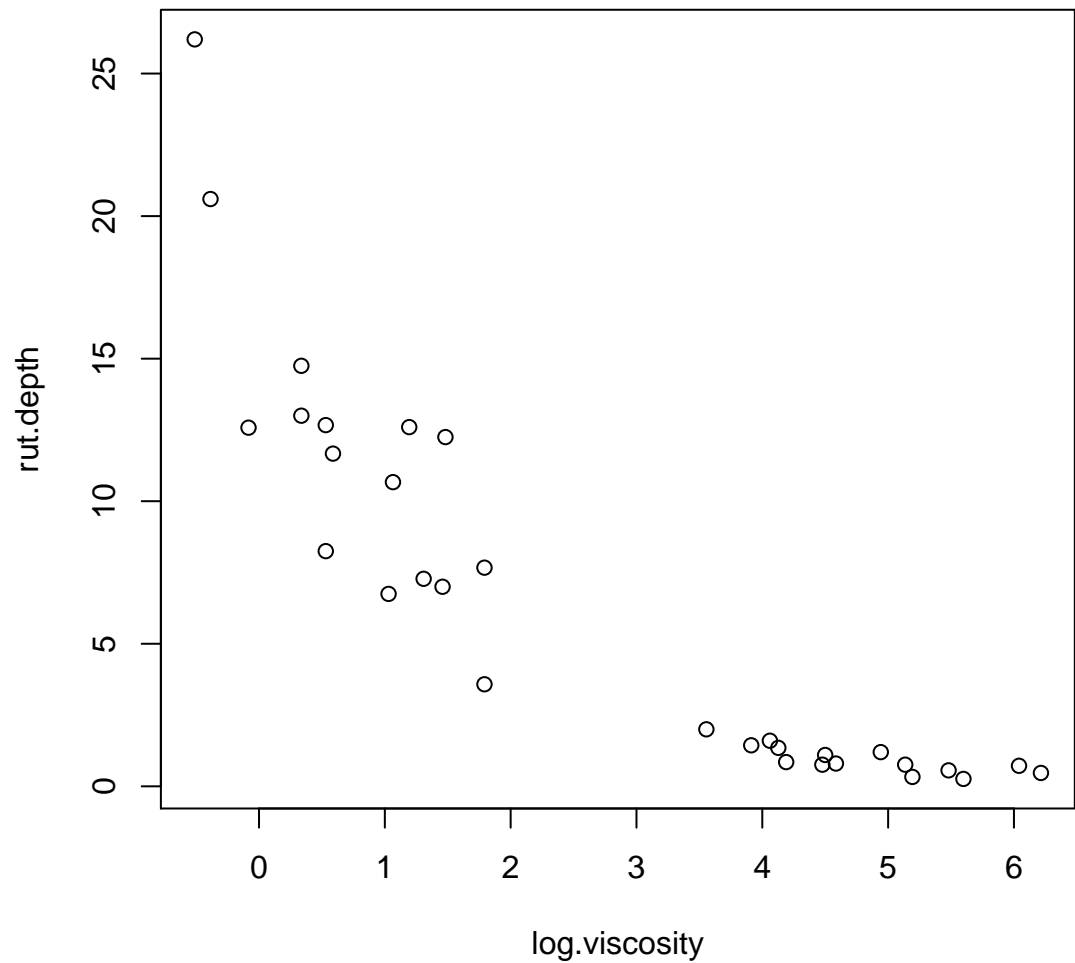
Something else to consider is the correlation *between the two explanatory variables* `viscosity` and `run`: when `run=0`, the viscosity is larger and more variable than when `run=1`. That is to say, `run=0` contained most of the high-viscosity observations, so it's hard to know whether `rut.depth` really depends on the `run` or the viscosity. This is what we know as “confounding”.

Now, the non-linearity of the relationship between `rut.depth` and `viscosity` is something we should be greatly concerned about, because any kind of regression is based on linearity. This would be a good point to go back to the asphalt engineer with this page of plots. The engineer plays with his beard a little,¹⁶ and says “We often find that the *log* of the viscosity tends to have linear relationships with things.”

So let's replace `viscosity` by its log, and see whether that works any better. The first thing to check is that the relationship with `rut.depth` is more nearly linear:

```
R> attach(asphalt)
R> log.viscosity=log(viscosity)
R> plot(rut.depth~log.viscosity)
```

¹⁶We have a male engineer this time.



Not great, but at least better than we had before.

One of the issues in multiple regression is that it's hard to guess which explanatory variables have an effect on the response, just from the pairs plot. So the typical way to start is to try to predict the response variable from *everything* else, and see how that looks. A multiple regression model formula has all the explanatory variables on the right of the squiggle, joined by plusses:

```
R> rut.lm=lm(rut.depth~pct.a.surf+pct.a.base+fines+voids+log.viscosity+run)
R> summary(rut.lm)
```

Call:

```
lm(formula = rut.depth ~ pct.a.surf + pct.a.base + fines + voids +
    log.viscosity + run)
```

Residuals:

| | Min | 1Q | Median | 3Q | Max |
|--|---------|---------|---------|--------|--------|
| | -4.1211 | -1.9075 | -0.7175 | 1.6382 | 9.5947 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|---------------|----------|------------|---------|-----------|
| (Intercept) | -12.9937 | 26.2188 | -0.496 | 0.6247 |
| pct.a.surf | 3.9706 | 2.4966 | 1.590 | 0.1248 |
| pct.a.base | 1.2631 | 3.9703 | 0.318 | 0.7531 |
| fines | 0.1164 | 1.0124 | 0.115 | 0.9094 |
| voids | 0.5893 | 1.3244 | 0.445 | 0.6604 |
| log.viscosity | -3.1515 | 0.9194 | -3.428 | 0.0022 ** |
| run | -1.9655 | 3.6472 | -0.539 | 0.5949 |

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

Residual standard error: 3.324 on 24 degrees of freedom

Multiple R-squared: 0.806, Adjusted R-squared: 0.7575

F-statistic: 16.62 on 6 and 24 DF, p-value: 1.743e-07

The place to start is at the bottom. R-squared is 81%, which is perhaps better than I was expecting. The next thing to look at is the F -statistic in the bottom row, and its P-value. This is testing the null hypothesis that *nothing* helps to predict the response. Here, that is resoundingly rejected, so *something* has an effect. To find out what, look at the table of coefficients. Each explanatory variable has a slope, and each slope has a P-value, in the right-most column of the table. The only P-value that is small belongs to `log.viscosity`. So that appears to be the only helpful explanatory variable.

We should look at the residual plot now, but I'm going to deliberately not do that in order to show you something else. Let's first see what happens when we throw out everything that isn't significant:

```
R> rut2.lm=lm(rut.depth~log.viscosity)
R> summary(rut2.lm)
```

Call:

```
lm(formula = rut.depth ~ log.viscosity)
```

Residuals:

| | Min | 1Q | Median | 3Q | Max |
|--|--------|--------|--------|-------|--------|
| | -5.487 | -1.635 | -0.538 | 1.624 | 10.816 |

Coefficients:

```

              Estimate Std. Error t value Pr(>|t|)
(Intercept)   13.9827     0.9315   15.01 3.29e-15 ***
log.viscosity -2.7434     0.2688  -10.21 4.15e-11 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

```

Residual standard error: 3.204 on 29 degrees of freedom
Multiple R-squared:  0.7822,    Adjusted R-squared:  0.7747
F-statistic: 104.2 on 1 and 29 DF,  p-value: 4.152e-11

```

Only `log.viscosity` is left. The `summary` shows that this variable is *strongly* significant, and also that R-squared hasn't dropped much at all (from 81% to 78%). That suggests that, in the context of this model at least, the other variables really don't have anything to add. But we haven't done a test for that; we've just removed them, shrugged our shoulders, and said "well, that didn't affect R-squared much". The *t*-tests only test for *one variable at a time*; they don't test for the significance of a bunch of variables all at once.

What we need to do is to compare the two models: does the bigger one fit significantly better than the small one,¹⁷ in which case we really do need the extra complication that the big model brings, or not, in which case we apply Occam's Razor and go with the smaller, simpler model. R provides a general mechanism for comparing two models and making this kind of decision, via the function `anova`. What you do is to feed `anova` two fitted model objects, the simpler one first, and it tests whether the bigger, more complicated model is a significantly better fit. Like this:

```
R> anova(rut2.lm,rut.lm)
```

Analysis of Variance Table

```

Model 1: rut.depth ~ log.viscosity
Model 2: rut.depth ~ pct.a.surf + pct.a.base + fines + voids + log.viscosity +
run
  Res.Df    RSS Df Sum of Sq    F Pr(>F)
1      29 297.62
2      24 265.10  5    32.525 0.5889 0.7084

```

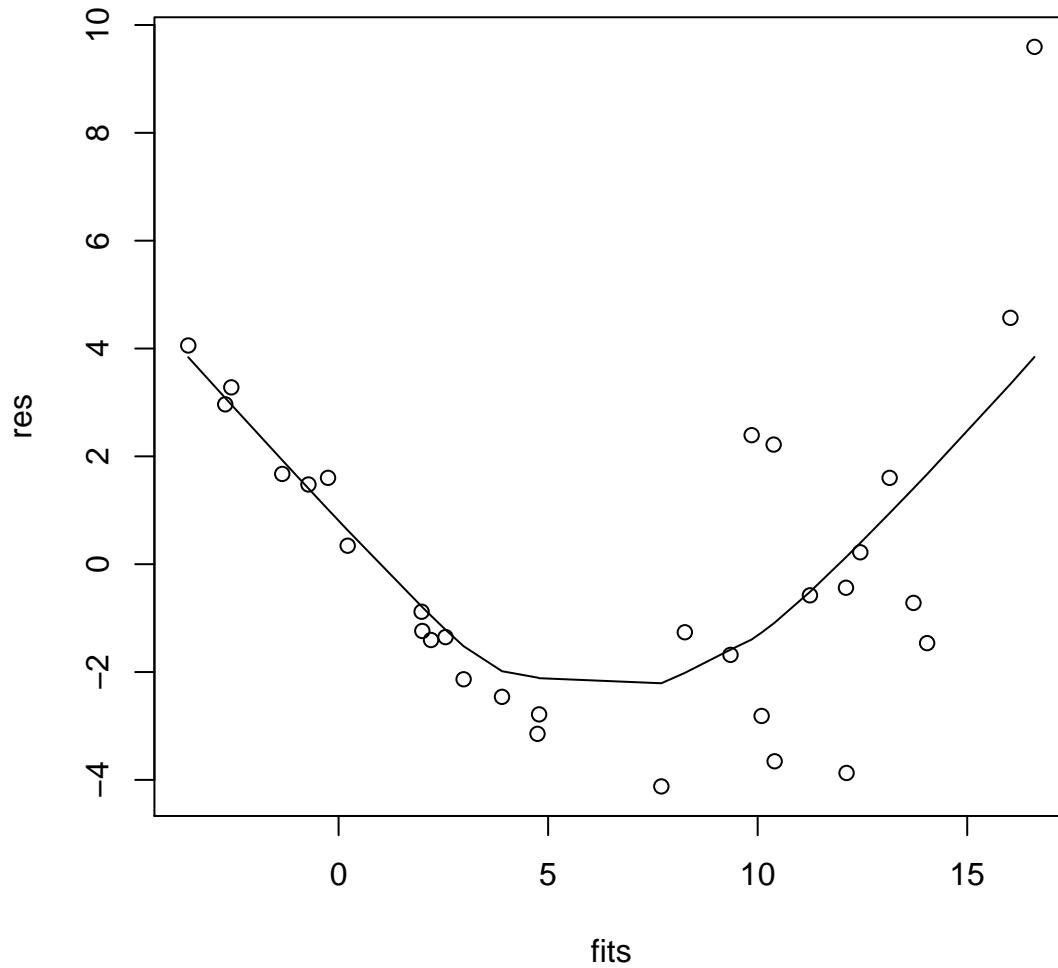
The null hypothesis here is that the two models fit equally well. The P-value is by no means small, so we cannot reject that null. Therefore, our decision would be to go with the simpler, smaller model.

I wanted to show you this mechanism, so that you can use it when you need to. But, we didn't do something very important: we didn't check the residuals for our big model. We should do that now.

```
R> fits=fitted(rut.lm)
R> res=resid(rut.lm)
```

¹⁷Or, equivalently, "does the small model fit significantly worse than the big one?"

```
R> plot(fits,res)
R> lines(lowess(fits,res))
```



Oh no, bendiness! We have a problem with a curved relationship again. In multiple regression, though, there are two possible remedies and they are different: either we can transform the response variable, or we can transform some of the explanatory variables.

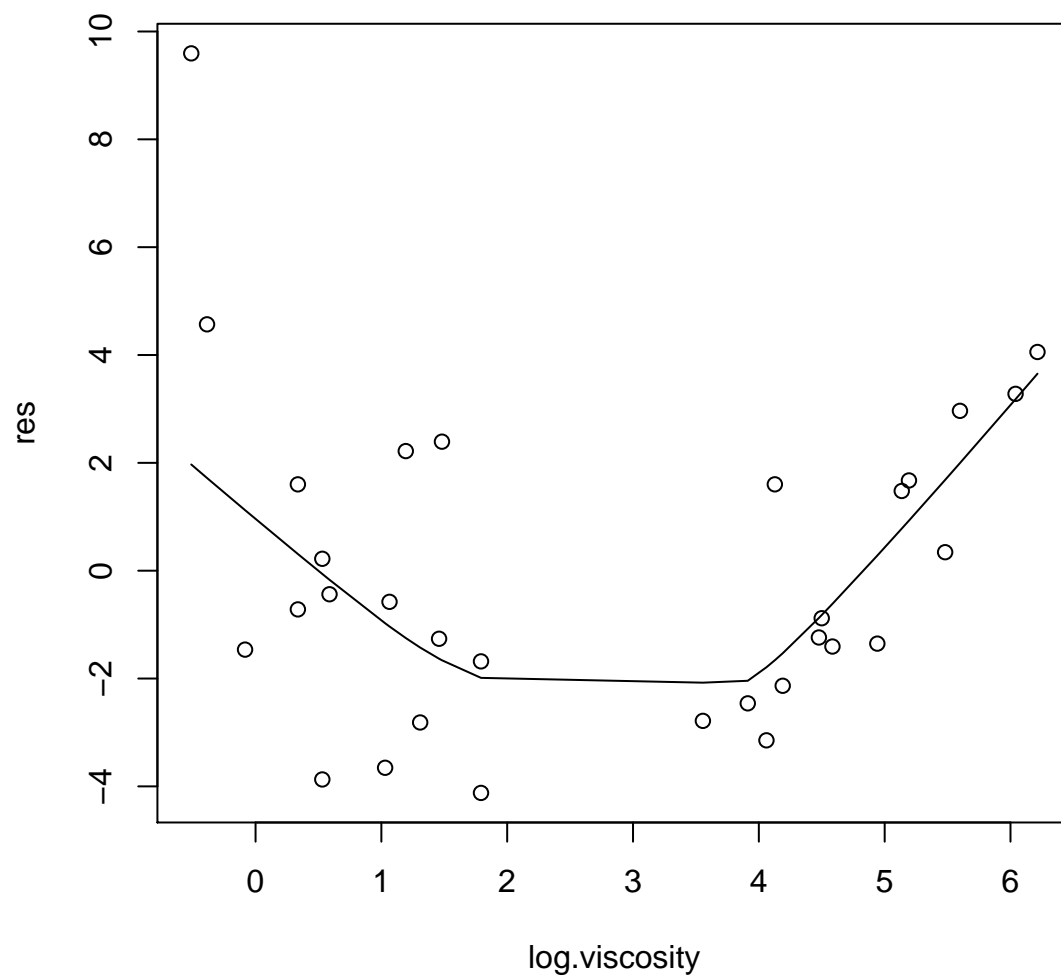
The plot of residuals against fitted values, above, has a problem, and it suggests that we fix it by transforming the response. But, we can also plot the residuals

against the explanatory variables.¹⁸ If *some* of those plots indicate a problem, you might be able to transform just those x 's where the problems are, and leave the response alone. It's an art, rather than a science, and there may well be no one best way to go.

We have six explanatory variables, so we should look at all six. I'll just show you a few, though. First, residuals against `log.viscosity`:

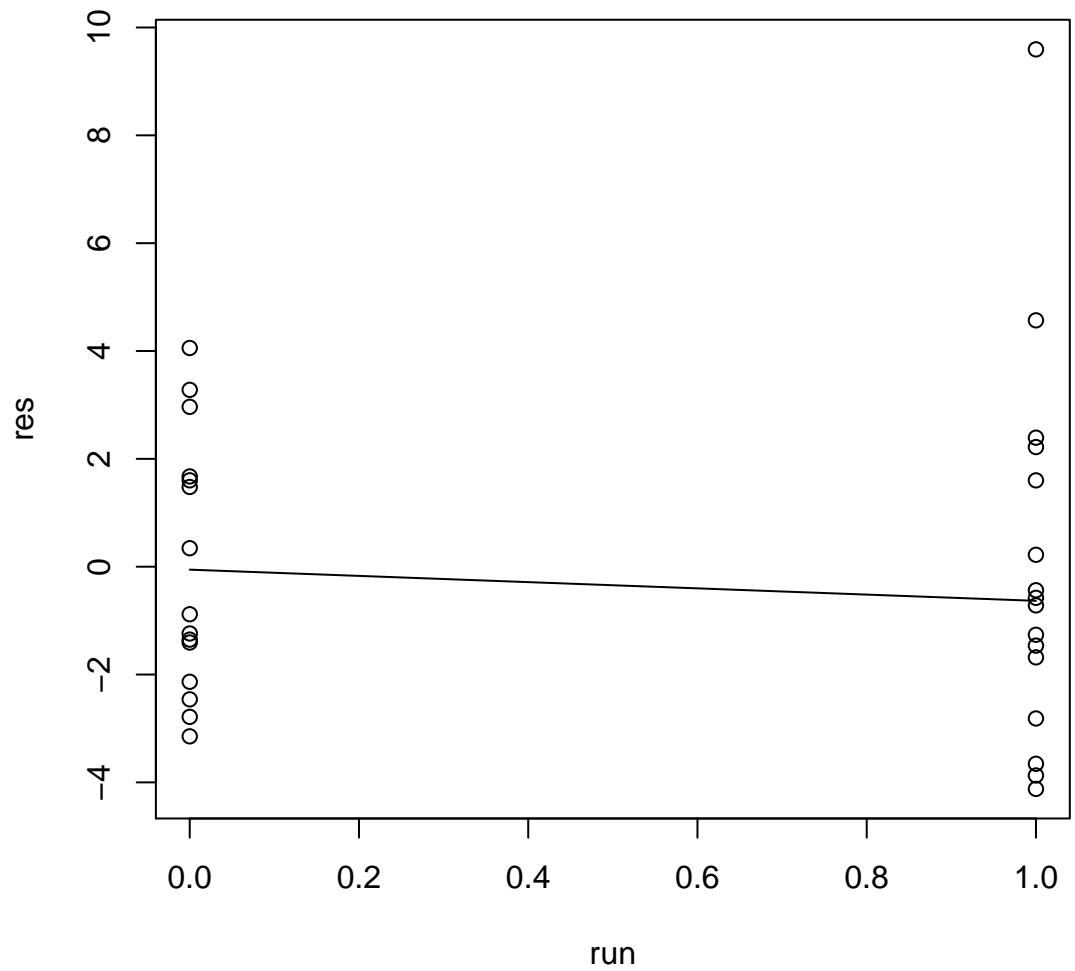
```
R> plot(log.viscosity,res)
R> lines(lowess(log.viscosity,res))
```

¹⁸Actually, you can plot the residuals against whatever you like.



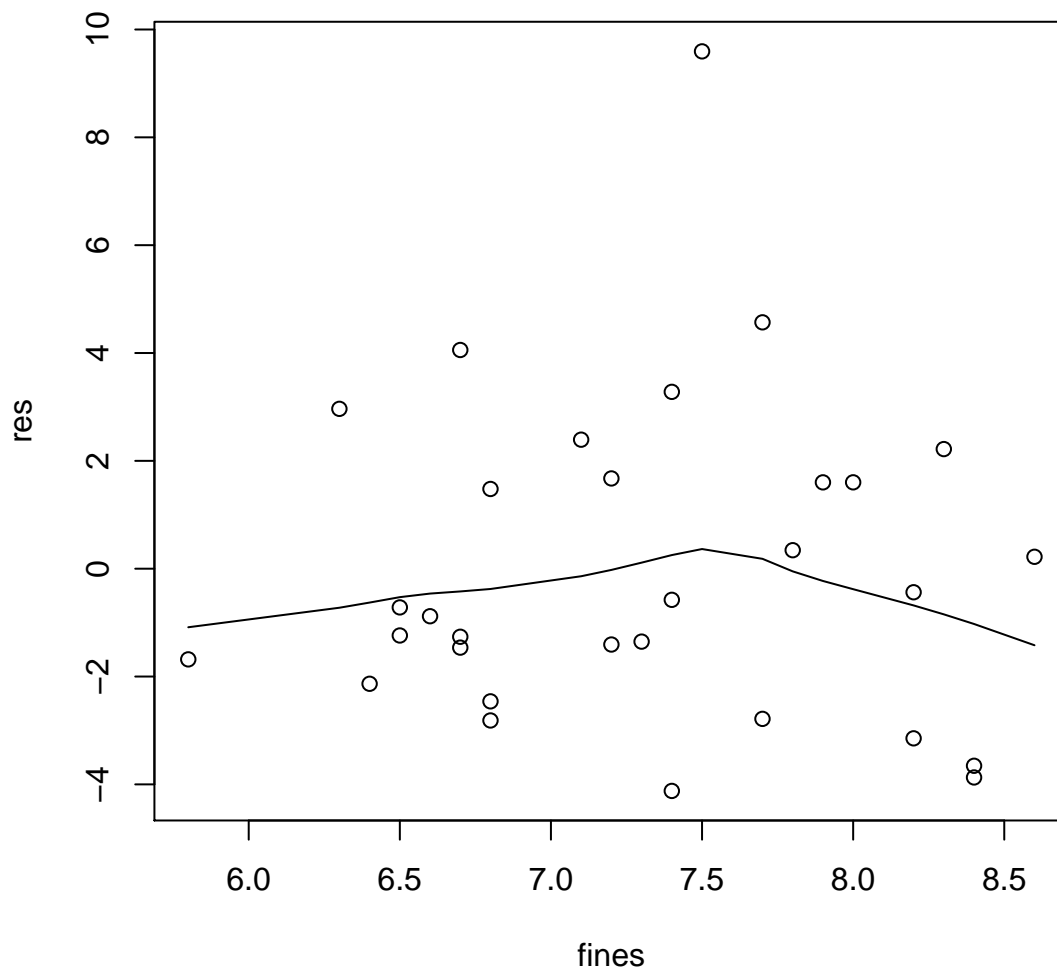
This shows the same curved trend. Next, residuals against run:

```
R> plot(run,res)
R> lines(lowess(run,res))
```



This one is not bad, though the trend goes a little bit down. It's hard to detect a curved trend with `run`, since it has only two values. Finally, `fines` against residuals:

```
R> plot(fines,res)
R> lines(lowess(fines,res))
```

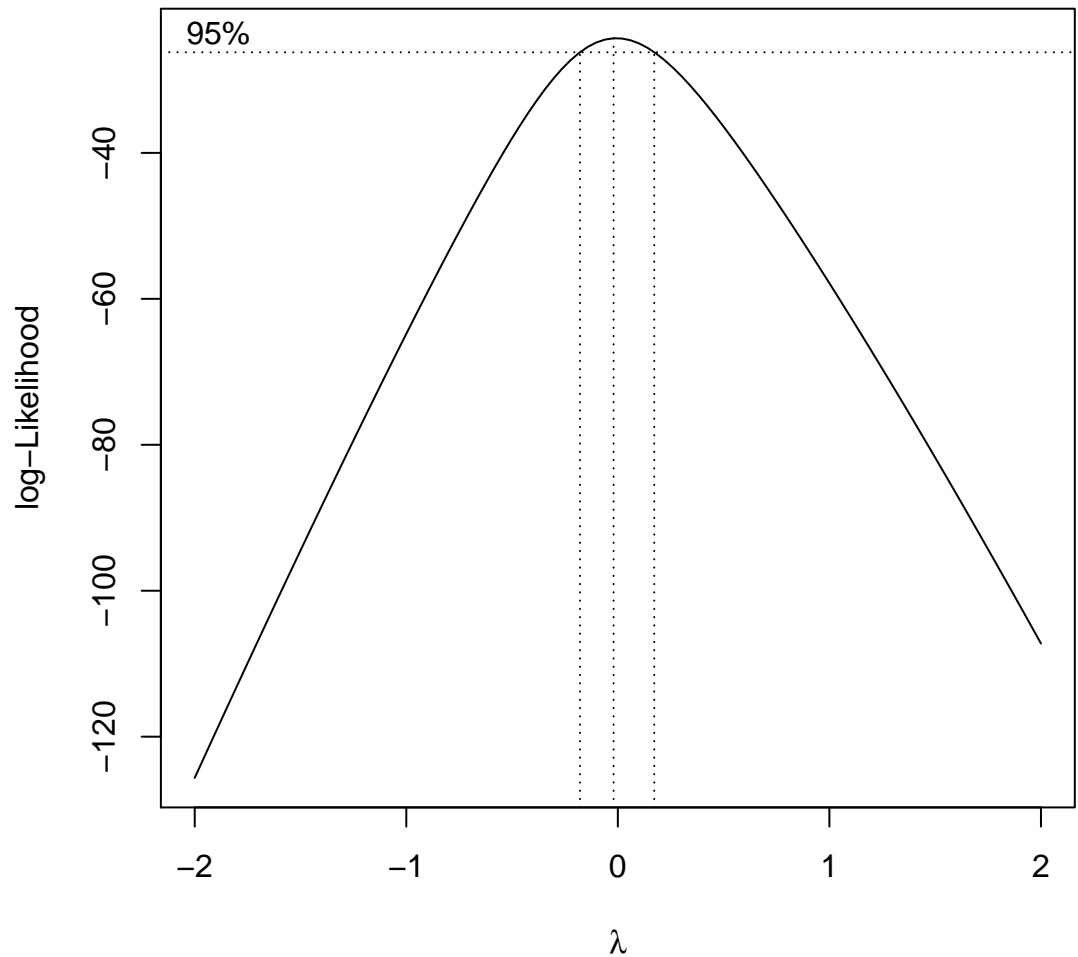



This one has a bit of an up-and-down on the lowess curve. Remember that the lowess curve is not unduly affected by that outlier, so this trend is indicative of an *overall* problem caused by a bunch of points, not just one.

My sense is that there are problems with *everything*, so it's the kind of thing that a transformation of the *response* variable might fix. Since we took logs of the *viscosity* values, taking logs of *rut.depth* is something we could try. We can also do Box-Cox again:

```
R> library(MASS)
```

```
R> boxcox(rut.depth~pct.a.surf+pct.a.base+finest+voids+viscosity+run)
```



You see that a log transformation of the response ($\lambda = 0$) is very much supported by the data.

All right, so \log^{19} of rut depth it is:

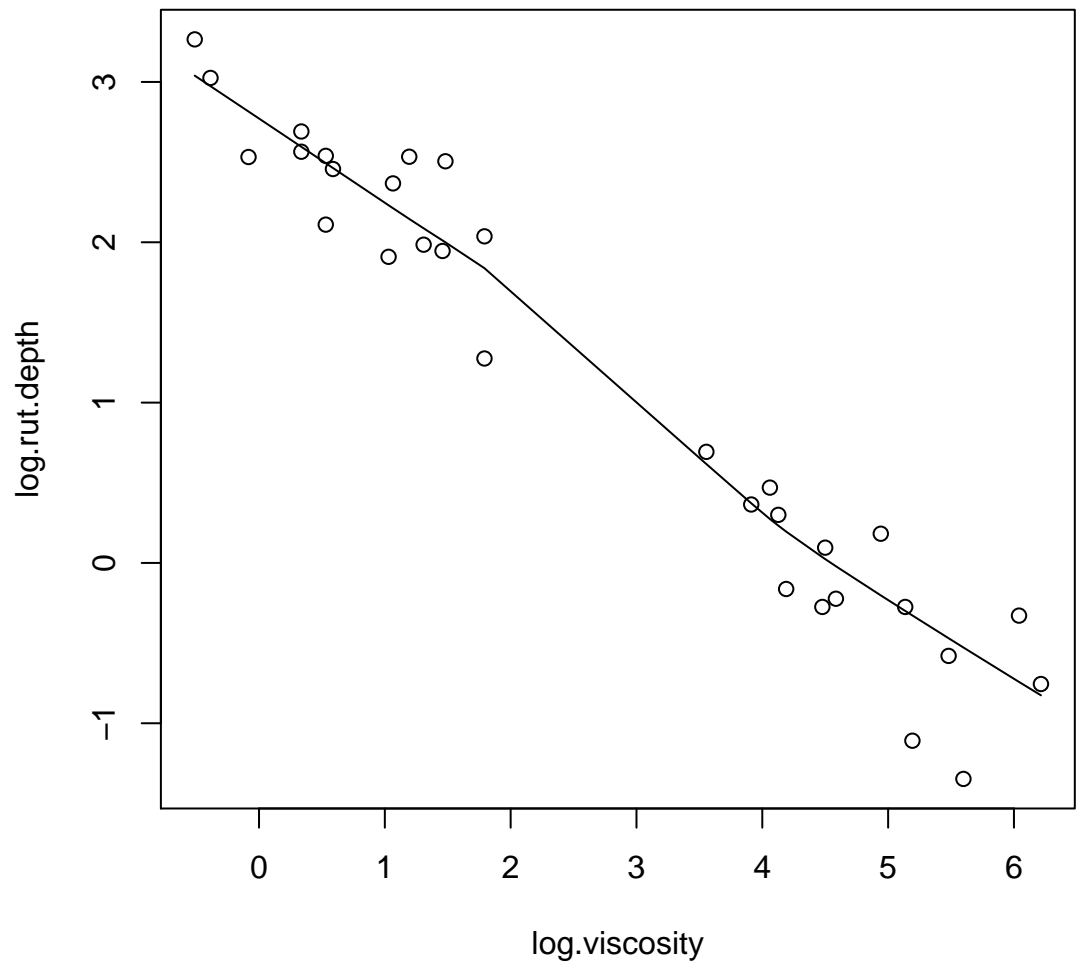
```
R> log.rut.depth=log(rut.depth)
```

¹⁹“Natural”, base e log for R.

Now we have to start all over again. First, we fit the model with all the explanatory variables in it. Then we check the residual plots and convince ourselves that we are happy with them. Then we do some variable selection: we decide which of the explanatory variables ought to stay and which of them can go.

Before we get to that, though: one of the things we were worried about before was the linearity of the relationship between the response and `log.viscosity`. Have we straightened that out?

```
R> plot(log.viscosity, log.rut.depth)
R> lines(lowess(log.viscosity, log.rut.depth))
```



That is *seriously* straighter. All right, let's go.

First, fit everything:

```
R> log.rut.lm=lm(log.rut.depth~pct.a.surf+pct.a.base+fines+voids
R> +log.viscosity+run)
R> summary(log.rut.lm)
```

Call:

```
lm(formula = log.rut.depth ~ pct.a.surf + pct.a.base + fines +
```

```

voids + log.viscosity + run)

Residuals:
      Min       1Q   Median       3Q      Max
-0.53072 -0.18563 -0.00003  0.20017  0.55079

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)   -1.57299     2.43617   -0.646    0.525
pct.a.surf      0.58358     0.23198    2.516    0.019 *
pct.a.base     -0.10337     0.36891   -0.280    0.782
fines           0.09775     0.09407    1.039    0.309
voids           0.19885     0.12306    1.616    0.119
log.viscosity  -0.55769     0.08543   -6.528 9.45e-07 ***
run             0.34005     0.33889    1.003    0.326
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3088 on 24 degrees of freedom
Multiple R-squared:  0.961,    Adjusted R-squared:  0.9512
F-statistic: 98.47 on 6 and 24 DF,  p-value: 1.059e-15

```

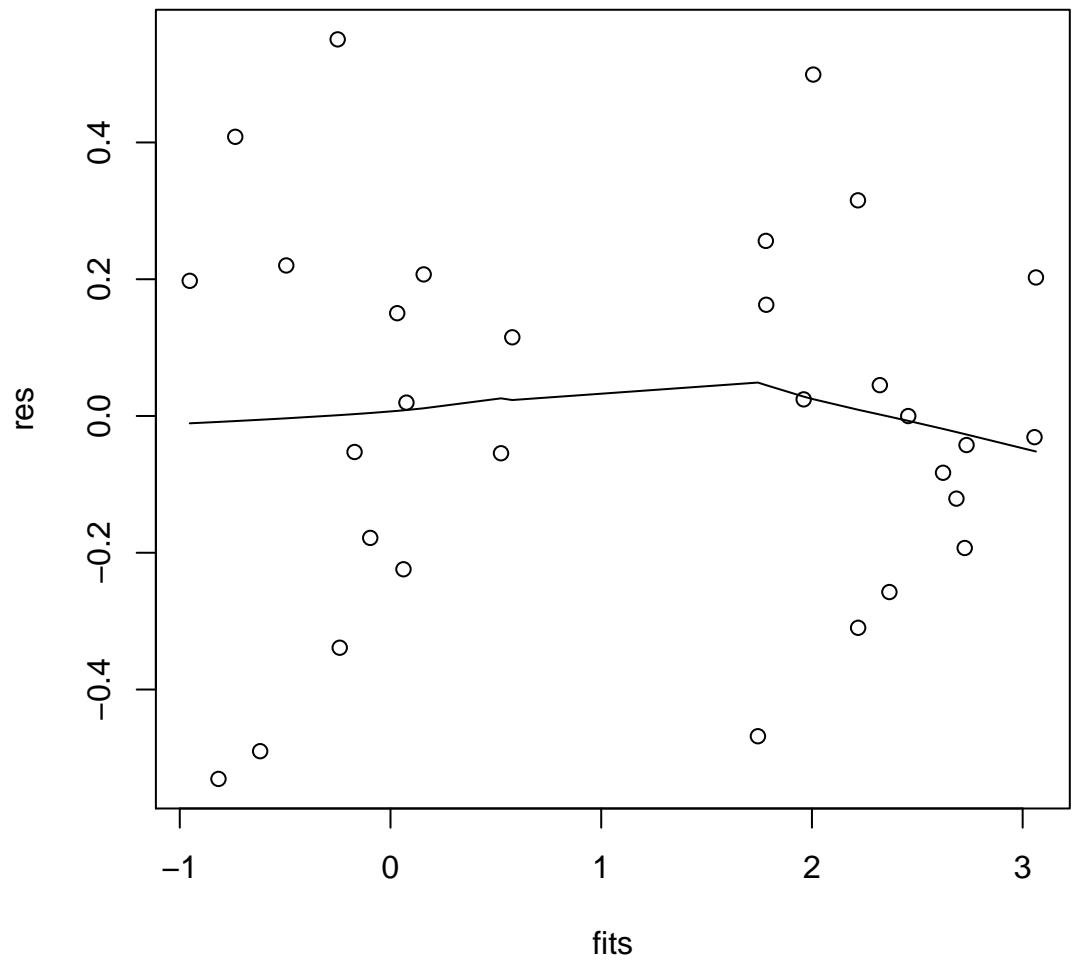
This has a pleasantly high R-squared of 96%. This is better than before we took logs (of `rut.depth`). Even though we are not comparing like with like, the increase in R-squared suggests that we have straightened things out, not just for the relationship with `log.viscosity` but for everything.

The next thing we need to do is to look at a residual plot. We start with residuals against fitted values:

```

R> fits=fitted(log.rut.lm)
R> res=resid(log.rut.lm)
R> plot(fits,res)
R> lines(lowess(fits,res))

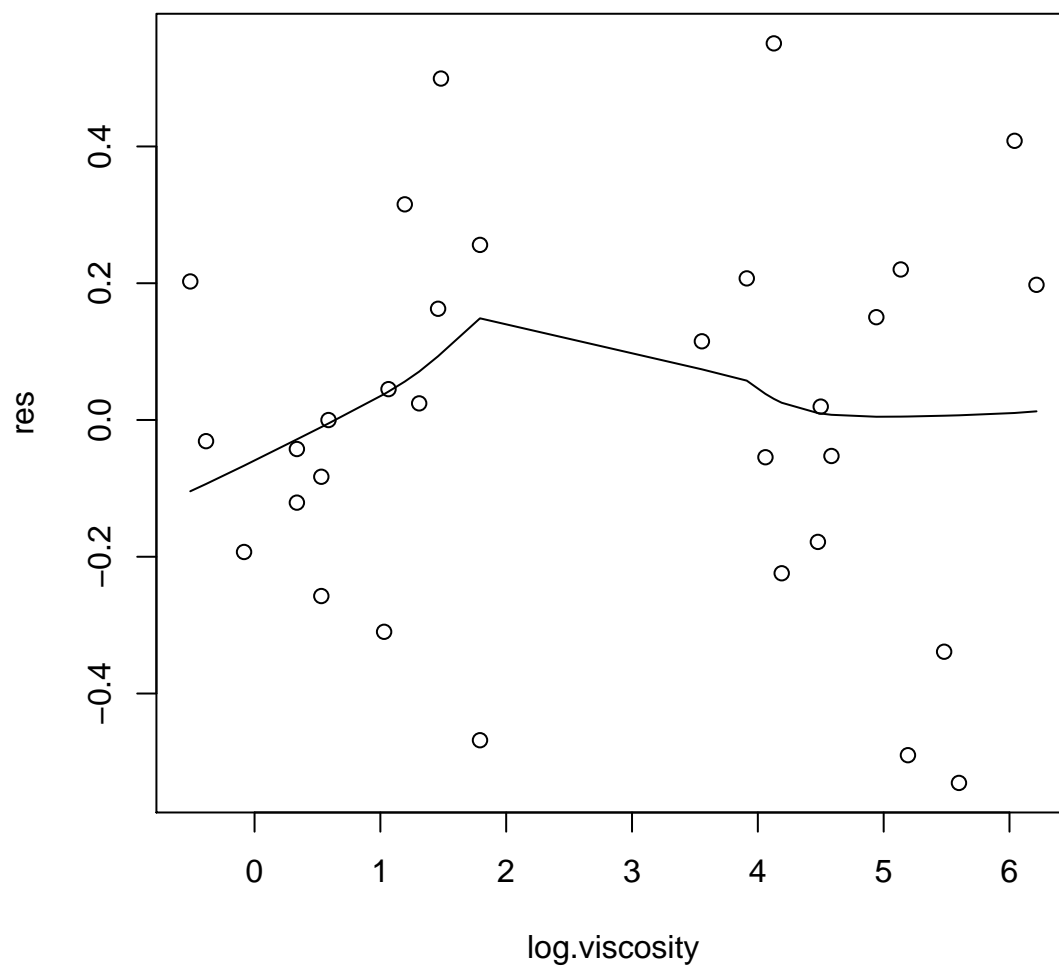
```



That's a whole lot better. Pretty much no trend.

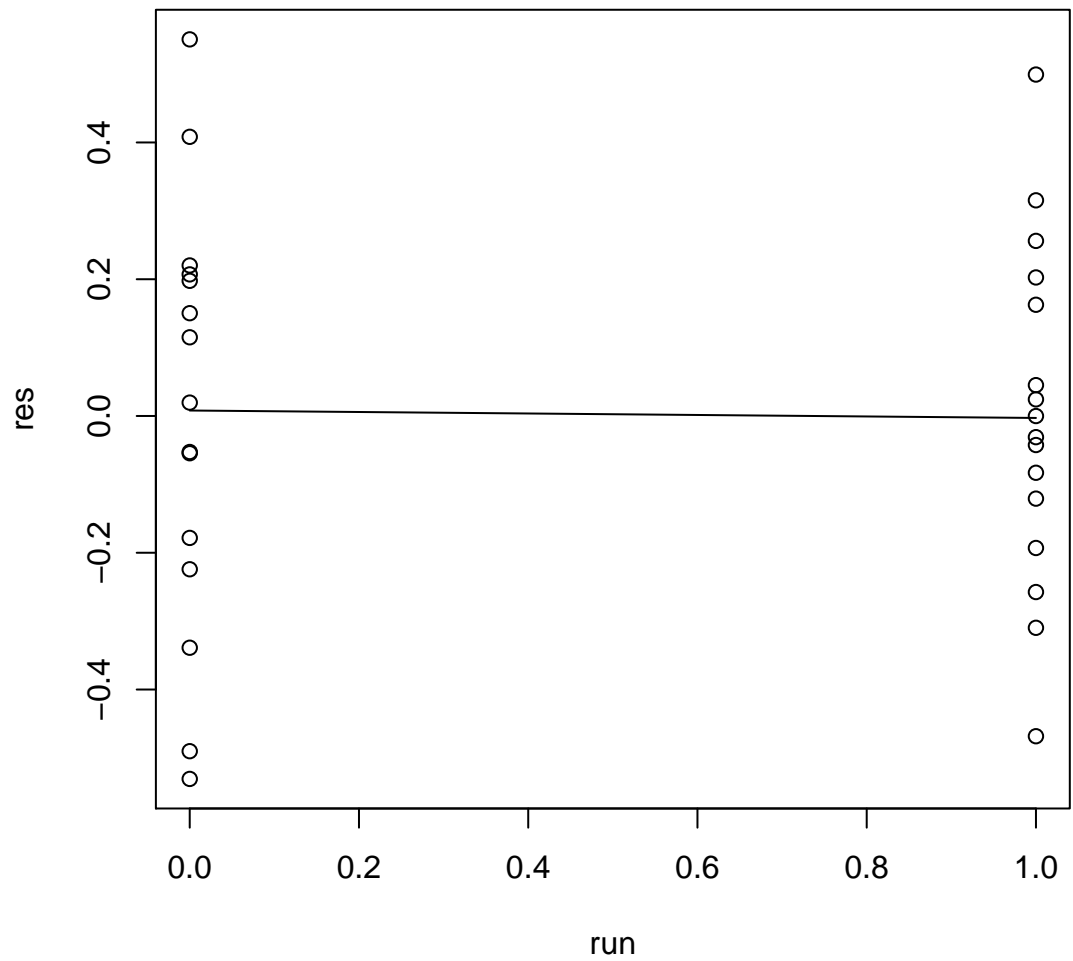
Now, we can plot the residuals against the explanatory variables, or at least the ones we did before. Let's start with `log.viscosity`:

```
R> plot(log.viscosity,res)
R> lines(lowess(log.viscosity,res))
```



I'd say no curves here, though feel free to disagree. `run`:

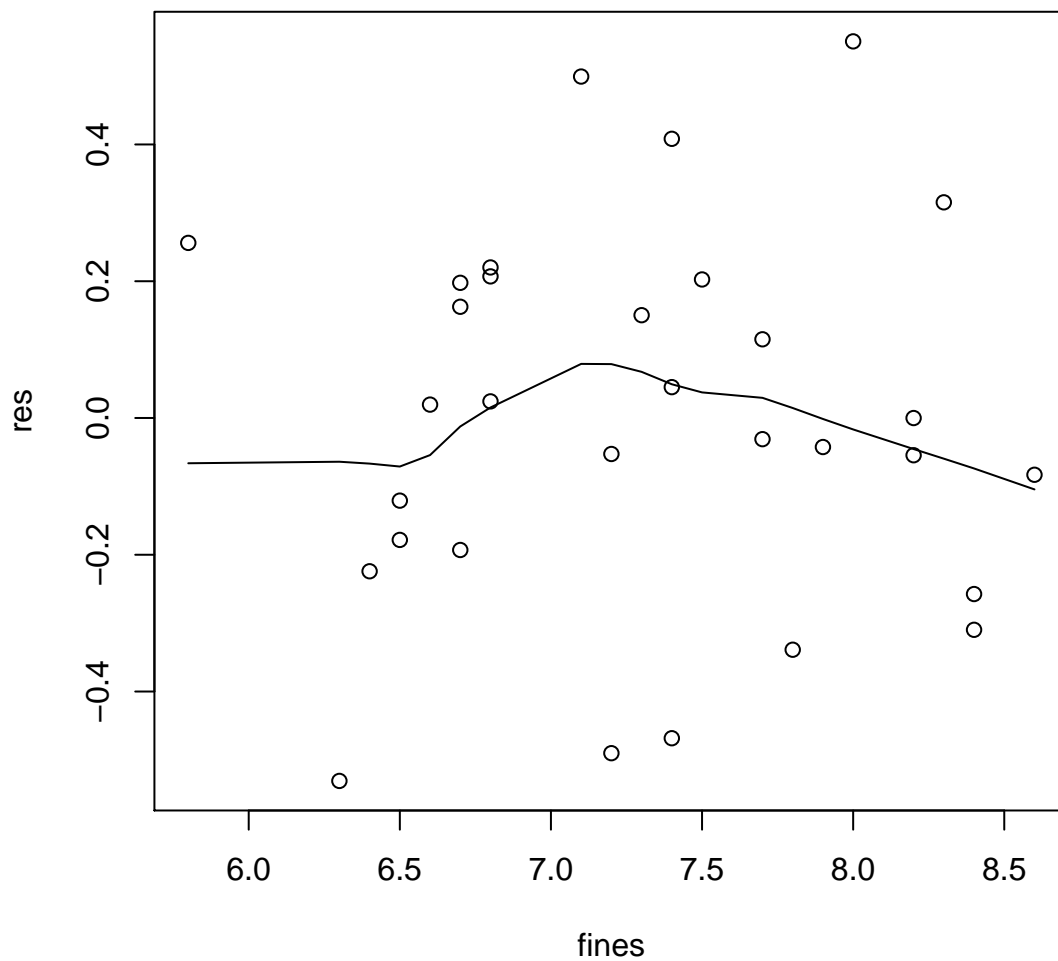
```
R> plot(run,res)
R> lines(lowess(run,res))
```



As flat as can be. The mean residual for each run is very close to zero, and also the spread of residuals looks about the same each time.

And the one we did before (which showed a bit of a curved pattern then) was `fines`:

```
R> plot(fines,res)
R> lines(lowess(fines,res))
```

I'd call that a minor wiggle, no more.

In principle, I do this for all the explanatory variables. This is necessary even for the ones that are not significant, because there could be a non-linear relationship with the response²⁰ but no linear one. I leave it to the reader to check the others.²¹

So, we seem to have the right form of relationship: a log transformation for the

²⁰Like a down-and-up curve, for example.

²¹Yes, I'm being lazy here.

response `rut.depth` and for `viscosity` yields linear relationships with at least some of the other variables.

The next stage is to decide which of the explanatory variables we need. The way I generally do this is to start with the model that has “everything” in it, and see what I can take out. Bear in mind that the tests in `summary` are for each explanatory variable *on its own*, and if you want to test more than one at the same time, you have to do the `anova` thing: fit a model *without* the variables you want to test, and see whether removing them all was a mistake.

Let’s remind ourselves of the model we’re at right now:

```
R> summary(log.rut.lm)
```

Call:

```
lm(formula = log.rut.depth ~ pct.a.surf + pct.a.base + fines +
    voids + log.viscosity + run)
```

Residuals:

| | Min | 1Q | Median | 3Q | Max |
|--|----------|----------|----------|---------|---------|
| | -0.53072 | -0.18563 | -0.00003 | 0.20017 | 0.55079 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|---------------|----------|------------|---------|--------------|
| (Intercept) | -1.57299 | 2.43617 | -0.646 | 0.525 |
| pct.a.surf | 0.58358 | 0.23198 | 2.516 | 0.019 * |
| pct.a.base | -0.10337 | 0.36891 | -0.280 | 0.782 |
| fines | 0.09775 | 0.09407 | 1.039 | 0.309 |
| voids | 0.19885 | 0.12306 | 1.616 | 0.119 |
| log.viscosity | -0.55769 | 0.08543 | -6.528 | 9.45e-07 *** |
| run | 0.34005 | 0.33889 | 1.003 | 0.326 |

Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.3088 on 24 degrees of freedom

Multiple R-squared: 0.961, Adjusted R-squared: 0.9512

F-statistic: 98.47 on 6 and 24 DF, p-value: 1.059e-15

There are a couple of ways to go now. One way is to take out everything that is not significant, which here leaves only `pct.a.surf` and `log.viscosity`:

```
R> log.rut2.lm=lm(log.rut.depth~pct.a.surf+log.viscosity)
```

If you do this, you need to check that the stuff you took out wasn’t too much to take out at once, which is a job for `anova`:

```
R> anova(log.rut2.lm,log.rut.lm)
```

Analysis of Variance Table

```

Model 1: log.rut.depth ~ pct.a.surf + log.viscosity
Model 2: log.rut.depth ~ pct.a.surf + pct.a.base + fines + voids + log.viscosity +
run
  Res.Df    RSS Df Sum of Sq    F Pr(>F)
1      28 2.8809
2      24 2.2888  4   0.59216 1.5523 0.2191

```

This time, we were safe, but we might not have been, which is why we needed to check. If you take some x 's out, others might become significant, especially if the x 's are intercorrelated enough.

The other way to go is to start with the biggest model:

```

R> summary(log.rut.lm)

Call:
lm(formula = log.rut.depth ~ pct.a.surf + pct.a.base + fines +
    voids + log.viscosity + run)

Residuals:
    Min       1Q   Median       3Q      Max
-0.53072 -0.18563 -0.00003  0.20017  0.55079

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  -1.57299     2.43617  -0.646   0.525
pct.a.surf     0.58358     0.23198   2.516   0.019 *
pct.a.base    -0.10337     0.36891  -0.280   0.782
fines         0.09775     0.09407   1.039   0.309
voids         0.19885     0.12306   1.616   0.119
log.viscosity -0.55769     0.08543  -6.528 9.45e-07 ***
run           0.34005     0.33889   1.003   0.326
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

```

Residual standard error: 0.3088 on 24 degrees of freedom
Multiple R-squared:  0.961,    Adjusted R-squared:  0.9512
F-statistic: 98.47 on 6 and 24 DF,  p-value: 1.059e-15

```

and take out the least significant variable first, which is `pct.a.base`:

```

R> log.rut3.lm=lm(log.rut.depth~pct.a.surf+fines+voids+log.viscosity+run)
R> summary(log.rut3.lm)

Call:
lm(formula = log.rut.depth ~ pct.a.surf + fines + voids + log.viscosity +
    run)

Residuals:

```

| | Min | 1Q | Median | 3Q | Max |
|--|----------|----------|----------|---------|---------|
| | -0.51610 | -0.18785 | -0.02248 | 0.18364 | 0.57160 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|---------------|----------|------------|---------|--------------|
| (Intercept) | -2.07850 | 1.60665 | -1.294 | 0.2076 |
| pct.a.surf | 0.59299 | 0.22526 | 2.632 | 0.0143 * |
| finest | 0.08895 | 0.08701 | 1.022 | 0.3165 |
| voids | 0.20047 | 0.12064 | 1.662 | 0.1091 |
| log.viscosity | -0.55249 | 0.08184 | -6.751 | 4.48e-07 *** |
| run | 0.35977 | 0.32533 | 1.106 | 0.2793 |

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

Residual standard error: 0.3031 on 25 degrees of freedom

Multiple R-squared: 0.9608, Adjusted R-squared: 0.953

F-statistic: 122.7 on 5 and 25 DF, p-value: < 2.2e-16

Now finest is the next one to go:

```
R> log.rut4.lm=lm(log.rut.depth~pct.a.surf+voids+log.viscosity+run)
R> summary(log.rut4.lm)
```

Call:

```
lm(formula = log.rut.depth ~ pct.a.surf + voids + log.viscosity +
    run)
```

Residuals:

| | Min | 1Q | Median | 3Q | Max |
|--|----------|----------|---------|---------|---------|
| | -0.57275 | -0.20080 | 0.01061 | 0.17711 | 0.59774 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|---------------|----------|------------|---------|--------------|
| (Intercept) | -1.25533 | 1.39147 | -0.902 | 0.3753 |
| pct.a.surf | 0.54837 | 0.22118 | 2.479 | 0.0200 * |
| voids | 0.23188 | 0.11676 | 1.986 | 0.0577 . |
| log.viscosity | -0.58039 | 0.07723 | -7.516 | 5.59e-08 *** |
| run | 0.29468 | 0.31931 | 0.923 | 0.3646 |

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

Residual standard error: 0.3033 on 26 degrees of freedom

Multiple R-squared: 0.9592, Adjusted R-squared: 0.9529

F-statistic: 152.8 on 4 and 26 DF, p-value: < 2.2e-16

and then run:

```
R> log.rut5.lm=lm(log.rut.depth~pct.a.surf+voids+log.viscosity)
```

```
R> summary(log.rut5.lm)

Call:
lm(formula = log.rut.depth ~ pct.a.surf + voids + log.viscosity)

Residuals:
    Min       1Q   Median       3Q      Max
-0.53548 -0.20181 -0.01702  0.16748  0.54707

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)   -1.02079     1.36430   -0.748   0.4608
pct.a.surf      0.55547     0.22044    2.520   0.0180 *
voids          0.24479     0.11560    2.118   0.0436 *
log.viscosity  -0.64649     0.02879  -22.458  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3025 on 27 degrees of freedom
Multiple R-squared:  0.9579,    Adjusted R-squared:  0.9532
F-statistic: 204.6 on 3 and 27 DF,  p-value: < 2.2e-16
```

Everything that's left is significant, and would cause a significant worsening of the fit if it were removed. So here we stop. Log rut depth depends on three things: percent of asphalt in the surface layer, percent voids and log viscosity.

Note that we have a different final result by taking things out one at a time than we did by taking out four things at once:

```
R> summary(log.rut2.lm)

Call:
lm(formula = log.rut.depth ~ pct.a.surf + log.viscosity)

Residuals:
    Min       1Q   Median       3Q      Max
-0.61938 -0.21361  0.06635  0.14932  0.63012

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.90014     1.08059   0.833   0.4119
pct.a.surf     0.39115     0.21879   1.788   0.0846 .
log.viscosity -0.61856     0.02713  -22.797  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3208 on 28 degrees of freedom
```

Multiple R-squared: 0.9509, Adjusted R-squared: 0.9474
 F-statistic: 270.9 on 2 and 28 DF, p-value: < 2.2e-16

I prefer the answer that we got from removing things one at a time. In fact, I like one-at-a-time better as a general strategy. But the point is that it can make a difference which way we go.

The *best* way to decide whether an explanatory variable should stay in the regression is *expert knowledge*, in this case asking an asphalt engineer whether the variable should be important. The method I used above, “backward elimination”, is the best of the “automatic” methods, but the results it obtains may not be very reproducible.

You may also hear about “stepwise regression”. Do not even learn about this! Why not? See, for example,

<http://danielezrajohnson.com/stepwise.pdf>

In fact, we don’t even need to go as far as we did with eliminating variables. If our goal is *prediction*, it doesn’t matter so much whether we have useless explanatory variables in the regression or not. If our goal is *estimation*, then we need to make more of an effort in eliminating useless variables, since taking out anything can change the coefficients of the variables that are left. But we don’t necessarily need to go as far as eliminating *everything* that is not significant. See page 14 of the presentation cited above.

Let’s try a couple of the R shortcuts for backward elimination. One is `drop1`. This takes a fitted model object and instructions for performing a test, and gives us brief output telling us which variable to drop. Recalling that `log.rut.lm` is the regression of `log.rut.depth` on everything else, it goes like this:²²

```
R> drop1(log.rut.lm, test="F")
```

Single term deletions

Model:

```
log.rut.depth ~ pct.a.surf + pct.a.base + fines + voids + log.viscosity +
run
```

| | Df | Sum of Sq | RSS | AIC | F value | Pr(>F) |
|---------------|----|-----------|--------|---------|---------|---------------|
| <none> | | | 2.2888 | -66.785 | | |
| pct.a.surf | 1 | 0.6035 | 2.8923 | -61.530 | 6.3283 | 0.01898 * |
| pct.a.base | 1 | 0.0075 | 2.2963 | -68.684 | 0.0785 | 0.78173 |
| fines | 1 | 0.1030 | 2.3917 | -67.421 | 1.0799 | 0.30909 |
| voids | 1 | 0.2490 | 2.5378 | -65.584 | 2.6112 | 0.11918 |
| log.viscosity | 1 | 4.0637 | 6.3525 | -37.139 | 42.6125 | 9.445e-07 *** |
| run | 1 | 0.0960 | 2.3848 | -67.511 | 1.0069 | 0.32567 |

²²That F-in-quotes on `test=` means not that `test` should be `FALSE`, which would be `test=F`, but that we should do an *F* test.

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

This gives us a (slightly) more concise summary of the P-values for each explanatory variable, so that we can easily see which one to drop next, in this case `pct.a.base`. Then we have to fit the model without that variable and run `drop1` again.

If you want to do backward elimination automatically, there is a function to do that, called `step`. It also does stepwise regression, which I warned you off above, but you can make it do only backward elimination. The output is rather lengthy, as below:

```
R> step(log.rut.lm,direction="backward")
```

```
Start:  AIC=-66.79
```

```
log.rut.depth ~ pct.a.surf + pct.a.base + fines + voids + log.viscosity +
run
```

| | Df | Sum of Sq | RSS | AIC |
|-----------------|----|-----------|--------|---------|
| - pct.a.base | 1 | 0.0075 | 2.2963 | -68.684 |
| - run | 1 | 0.0960 | 2.3848 | -67.511 |
| - fines | 1 | 0.1030 | 2.3917 | -67.421 |
| <none> | | | 2.2888 | -66.785 |
| - voids | 1 | 0.2490 | 2.5378 | -65.584 |
| - pct.a.surf | 1 | 0.6035 | 2.8923 | -61.530 |
| - log.viscosity | 1 | 4.0637 | 6.3525 | -37.139 |

```
Step:  AIC=-68.68
```

```
log.rut.depth ~ pct.a.surf + fines + voids + log.viscosity +
run
```

| | Df | Sum of Sq | RSS | AIC |
|-----------------|----|-----------|--------|---------|
| - fines | 1 | 0.0960 | 2.3922 | -69.415 |
| - run | 1 | 0.1123 | 2.4086 | -69.203 |
| <none> | | | 2.2963 | -68.684 |
| - voids | 1 | 0.2536 | 2.5499 | -67.436 |
| - pct.a.surf | 1 | 0.6365 | 2.9328 | -63.099 |
| - log.viscosity | 1 | 4.1856 | 6.4819 | -38.514 |

```
Step:  AIC=-69.41
```

```
log.rut.depth ~ pct.a.surf + voids + log.viscosity + run
```

| | Df | Sum of Sq | RSS | AIC |
|--------------|----|-----------|--------|---------|
| - run | 1 | 0.0784 | 2.4706 | -70.415 |
| <none> | | | 2.3922 | -69.415 |
| - voids | 1 | 0.3629 | 2.7551 | -67.036 |
| - pct.a.surf | 1 | 0.5656 | 2.9578 | -64.836 |

```
- log.viscosity 1 5.1969 7.5891 -35.625
```

Step: AIC=-70.42

```
log.rut.depth ~ pct.a.surf + voids + log.viscosity
```

| | Df | Sum of Sq | RSS | AIC |
|-----------------|----|-----------|--------|---------|
| <none> | | | 2.471 | -70.415 |
| - voids | 1 | 0.410 | 2.881 | -67.652 |
| - pct.a.surf | 1 | 0.581 | 3.052 | -65.868 |
| - log.viscosity | 1 | 46.151 | 48.622 | 19.953 |

Call:

```
lm(formula = log.rut.depth ~ pct.a.surf + voids + log.viscosity)
```

Coefficients:

| (Intercept) | pct.a.surf | voids | log.viscosity |
|-------------|------------|--------|---------------|
| -1.0208 | 0.5555 | 0.2448 | -0.6465 |

The final regression is the same as we got “by hand” above. The output shows which explanatory variable was eliminated in turn, and you can see that the order was the same as we had. (The variable with the most negative AIC is the one that is eliminated.)

Another approach is to fit *all possible* regressions, and to see which one(s) you like best. This may seem like an immense task (for R, at least). With 6 explanatory variables as we have here, each of which can be either in or out of the regression, there are $2^6 = 64$ regressions to check. But there are some computational shortcuts that R knows about. First you need the package `leaps` (install it via `install.packages` if you don’t have it), and then this. The thing inside `regsubsets` is the same model formula as you used before (for fitting a model with all the x ’s). You can have a look at the thing I called `s`, but I like the output shown:

```
R> library(leaps)
```

```
R> leaps=regsubsets(log.rut.depth~pct.a.surf+pct.a.base+fines+voids+log.viscosity+run,
```

```
R> s=summary(leaps)
```

```
R> cbind(s$rsq,s$which)
```

| | (Intercept) | pct.a.surf | pct.a.base | fines | voids | log.viscosity | run |
|-------------|-------------|------------|------------|-------|-------|---------------|-----|
| 1 0.9452562 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 0.8624107 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 2 0.9508647 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 2 0.9479541 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 3 0.9578631 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 3 0.9534561 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 4 0.9591996 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 4 0.9589206 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 5 0.9608365 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |

| | | | | | | | | |
|---|-----------|---|---|---|---|---|---|---|
| 5 | 0.9593265 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 6 | 0.9609642 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

This shows the best two models with 1, 2, ... explanatory variables, and the R-squared for each. 1 means “in” and 0 means “out”. The point is that R-squared will *always* go up if you add an x , even if that x is worthless, so what you are looking for is where the R-squared doesn’t get much bigger.²³

In our case, you might say that there isn’t very much improvement over the model with just `log.viscosity` in it (the top line). Or you might look at the first line with 3 x ’s (R-squared 0.9578), and say that the regression with `pct.a.surf`, `voids` and `log.viscosity` is the best one. This is what we got from backward elimination. Alternatively, you can use Mallows’ C_p :

```
R> cbind(s$cp, s$which)
```

| | (Intercept) | pct.a.surf | pct.a.base | finest | voids | log.viscosity | run |
|---|-------------|------------|------------|--------|-------|---------------|-----|
| 1 | 6.657543 | 1 | 0 | 0 | 0 | | 1 0 |
| 1 | 57.592621 | 1 | 0 | 0 | 0 | | 0 1 |
| 2 | 5.209344 | 1 | 1 | 0 | 0 | | 1 0 |
| 2 | 6.998869 | 1 | 0 | 0 | 1 | | 1 0 |
| 3 | 2.906603 | 1 | 1 | 0 | 1 | | 1 0 |
| 3 | 5.616119 | 1 | 1 | 0 | 1 | 0 | 1 0 |
| 4 | 4.084907 | 1 | 1 | 0 | 0 | 1 | 1 1 |
| 4 | 4.256398 | 1 | 1 | 0 | 1 | 1 | 1 0 |
| 5 | 5.078511 | 1 | 1 | 0 | 1 | 1 | 1 1 |
| 5 | 6.006861 | 1 | 1 | 1 | 1 | 1 | 1 0 |
| 6 | 7.000000 | 1 | 1 | 1 | 1 | 1 | 1 1 |

The best C_p value is the smallest, as long as that is close to the number of explanatory variables in the model.²⁴ The smallest C_p value is 2.90, going with the same 3-variable model as above. And 2.90 is pretty close to 3, so that’s good. If you want to learn more about C_p , you’ll need to take STAC67.

²³If you’ve done STAB27, you might know about *adjusted R-squared*, which goes *down* if the added x is not worth adding. You can get this here via `s$adjr2`.

²⁴If it is too much smaller, it suggests that you have a fit that was good by chance, one that will not be reproducible.

Chapter 9

Intricacies of R

9.1 Introduction

R consists of a thousand tiny parts, all working together. These parts all have names, which you won't know until you discover them. I discovered most of these tiny parts by searching for things that I wanted to do in R, like

```
how to label vertical axis R
```

and found that somebody on some forum or mailing list somewhere had asked the same question, and *one of the people involved in R's development* replied with something like “Oh, you need `ylab`”. Then I went to R's help for `ylab`, figured out how it worked, and was able to use it myself. The name of the tiny part was all I needed, and I could take it from there.

That means we need to start with R's help files. After that, we'll have a look at what you can do with plots, and then we'll look into selecting and combining data, which includes things like calculating summaries for groups. Also, we'll take a look at packages, which are a way of using someone else's code that might happen to do exactly what we want.

To start, let's read in a data file that we can use to illustrate things. This data frame contains some information about 38 different cars:

```
R> cars=read.csv("cars.csv")
R> cars
```

| | Car | MPG | Weight | Cylinders | Horsepower | Country |
|---|----------------|------|--------|-----------|------------|---------|
| 1 | Buick Skylark | 28.4 | 2.67 | 4 | 90 | U.S. |
| 2 | Dodge Omni | 30.9 | 2.23 | 4 | 75 | U.S. |
| 3 | Mercury Zephyr | 20.8 | 3.07 | 6 | 85 | U.S. |
| 4 | Fiat Strada | 37.3 | 2.13 | 4 | 69 | Italy |

| | | | | | | |
|----|---------------------------|------|------|---|-----|---------|
| 5 | Peugeot 694 SL | 16.2 | 3.41 | 6 | 133 | France |
| 6 | VW Rabbit | 31.9 | 1.93 | 4 | 71 | Germany |
| 7 | Plymouth Horizon | 34.2 | 2.20 | 4 | 70 | U.S. |
| 8 | Mazda GLC | 34.1 | 1.98 | 4 | 65 | Japan |
| 9 | Buick Estate Wagon | 16.9 | 4.36 | 8 | 155 | U.S. |
| 10 | Audi 5000 | 20.3 | 2.83 | 5 | 103 | Germany |
| 11 | Chevy Malibu Wagon | 19.2 | 3.61 | 8 | 125 | U.S. |
| 12 | Dodge Aspen | 18.6 | 3.62 | 6 | 110 | U.S. |
| 13 | VW Dasher | 30.5 | 2.19 | 4 | 78 | Germany |
| 14 | Ford Mustang 4 | 26.5 | 2.59 | 4 | 88 | U.S. |
| 15 | Dodge Colt | 35.1 | 1.92 | 4 | 80 | Japan |
| 16 | Datsun 810 | 22.0 | 2.82 | 6 | 97 | Japan |
| 17 | VW Scirocco | 31.5 | 1.99 | 4 | 71 | Germany |
| 18 | Chevy Citation | 28.8 | 2.60 | 6 | 115 | U.S. |
| 19 | Olds Omega | 26.8 | 2.70 | 6 | 115 | U.S. |
| 20 | Chrysler LeBaron Wagon | 18.5 | 3.94 | 8 | 150 | U.S. |
| 21 | Datsun 510 | 27.2 | 2.30 | 4 | 97 | Japan |
| 22 | AMC Concord D/L | 18.1 | 3.41 | 6 | 120 | U.S. |
| 23 | Buick Century Special | 20.6 | 3.38 | 6 | 105 | U.S. |
| 24 | Saab 99 GLE | 21.6 | 2.80 | 4 | 115 | Sweden |
| 25 | Datsun 210 | 31.8 | 2.02 | 4 | 65 | Japan |
| 26 | Ford LTD | 17.6 | 3.73 | 8 | 129 | U.S. |
| 27 | Volvo 240 GL | 17.0 | 3.14 | 6 | 125 | Sweden |
| 28 | Dodge St Regis | 18.2 | 3.83 | 8 | 135 | U.S. |
| 29 | Toyota Corona | 27.5 | 2.56 | 4 | 95 | Japan |
| 30 | Chevette | 30.0 | 2.16 | 4 | 68 | U.S. |
| 31 | Ford Mustang Ghia | 21.9 | 2.91 | 6 | 109 | U.S. |
| 32 | AMC Spirit | 27.4 | 2.67 | 4 | 80 | U.S. |
| 33 | Ford Country Squire Wagon | 15.5 | 4.05 | 8 | 142 | U.S. |
| 34 | BMW 320i | 21.5 | 2.60 | 4 | 110 | Germany |
| 35 | Pontiac Phoenix | 33.5 | 2.56 | 4 | 90 | U.S. |
| 36 | Honda Accord LX | 29.5 | 2.14 | 4 | 68 | Japan |
| 37 | Mercury Grand Marquis | 16.5 | 3.96 | 8 | 138 | U.S. |
| 38 | Chevy Caprice Classic | 17.0 | 3.84 | 8 | 130 | U.S. |

9.2 Help!

At an R command prompt (such as in the Console in R Studio), type a question mark followed by the name of the function you want help with. The R help file will appear. In R Studio, it appears in the bottom right window where plots normally show up. Help files are not the friendliest reading, but with practice you can find out what you want to know.

Let's take a fairly simple help file to start with, the one for `median`:

median package:stats R Documentation

Median Value

Description:

 Compute the sample median.

Usage:

```
median(x, na.rm = FALSE)
```

Arguments:

 x: an object for which a method has been defined, or a numeric vector containing the values whose median is to be computed.

 na.rm: a logical value indicating whether 'NA' values should be stripped before the computation proceeds.

Details:

 This is a generic function for which methods can be written. However, the default method makes use of 'sort' and 'mean' from package 'base' both of which are generic, and so the default method will work for most classes (e.g. "Date") for which a median is a reasonable concept.

Value:

 The default method returns a length-one object of the same type as 'x', except when 'x' is integer of even length, when the result will be double.

 If there are no values or if 'na.rm = FALSE' and there are 'NA' values the result is 'NA' of the same type as 'x' (or more generally the result of 'x[FALSE][NA]').

References:

 Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also:

 'quantile' for general quantiles.

Examples:

```
median(1:4)           # = 2.5 [even number]
median(c(1:3, 100, 1000)) # = 3 [odd, robust]
```

The first thing in the help file, below the name of the function and the package¹ that it lives in² is the description of what it does. In this case it's very simple: "compute the sample median". Below that is the Usage, or how you use it. The basic usage is very simple: you feed `median` something (typically a list of numbers) and it gives you back their median. The section Arguments tells you a bit more about what can be fed into `median`; the important bit is "... or a numerical vector containing the values whose median is to be computed." The other argument tells R how to treat missing values (which, in R, are labelled `NA`). This is not typically important; the notation `na.rm = FALSE` in Usage with an equals sign in it shows that R has a **default**: it will supply a value for `na.rm` if you don't. So typically, all you need is to type something like `median(y)` and R will do the right thing.

The Details section provides more information on how the function works. I'll talk a little about "generic functions" later, but the interesting thing in this section is that you can also feed `median` a string of dates and R will do the right thing.

The Value section tells you what comes back from `median`. If you feed it an odd number of integers, you'll get back an integer³; if you feed it an *even* number of integers, you'll get back a decimal number, since the median is halfway between the two middle values, and that might be something-and-a-half. "Double" is R's name for a "double-precision floating-point number", or decimal number to you and me.⁴

Excel makes it virtually impossible to find out how it calculates things. R tells you exactly where to look, in References.

There is often a See Also section, pointing you towards other functions that might be useful to you, such as in this case `quantile`, which can be used to calculate a quartile or the 10th percentile or the 95th percentile or anything like that. If you are in R Studio, the help files have hyperlinks that you can just click on to go to the other help files.

Lastly, there are examples. The examples in the Help files are just code, which isn't often very helpful.⁵ But you can run the examples like this:

¹See later.

²The package `stats` is home to many of the standard functions.

³Whole number.

⁴In a computer, decimal numbers cannot be represented precisely, even with lots of digits of accuracy. See https://en.wikipedia.org/wiki/Floating_point, which has *lots* of detail. R uses 16 significant digits, which is generally accurate enough.

⁵Usually you want to see what the code *does*.

```
R> example(median)
```

```
median> median(1:4)           # = 2.5 [even number]
[1] 2.5
```

```
median> median(c(1:3, 100, 1000)) # = 3 [odd, robust]
[1] 3
```

R puts the example code behind a fake prompt, here `median>`, so you know it's not your code. Here, the median of the (integers) 1 through 4 is the decimal number 2.5, and the median of 1, 2, 3, 100, 1000 is 3, even though the last two numbers are really big.

Let's take a look at the help for `plot` next. The `plot` function does all manner of things, so you might think that it has a complicated help file. But it isn't really. The first part of it looks like this:

```
plot                                package:graphics                R Documentation
```

```
Generic X-Y Plotting
```

```
Description:
```

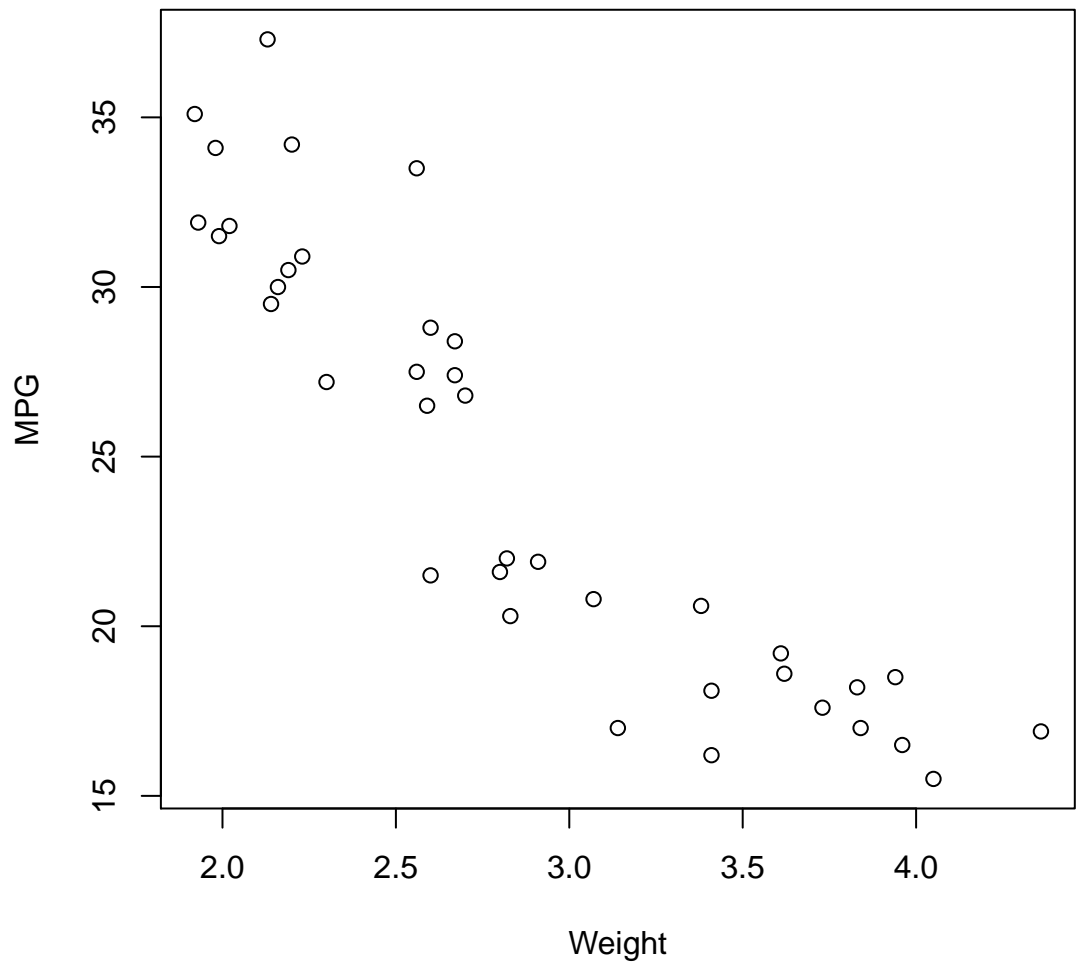
```
Generic function for plotting of R objects. For more details
about the graphical parameter arguments, see 'par'.
```

```
For simple scatter plots, 'plot.default' will be used. However,
there are 'plot' methods for many R objects, including
'function's, 'data.frame's, 'density' objects, etc. Use
'methods(plot)' and the documentation for these.
```

When you see “generic function”, this means that exactly what `plot` does depends on what you feed it. Let me illustrate with our cars data.

9.2.1 Making a scatterplot

```
R> attach(cars)
R> plot(Weight,MPG)
```

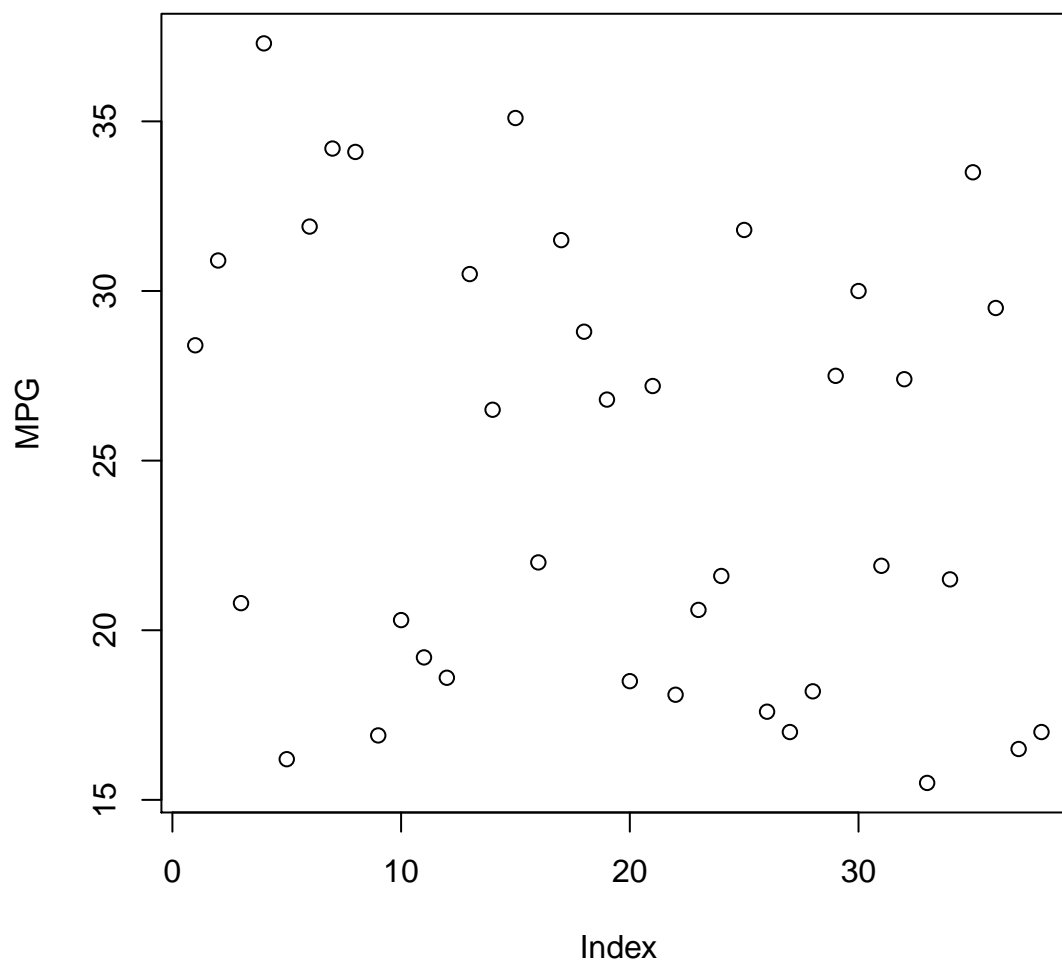


Plotting two numerical variables makes a scatter plot. In this case, it shows that as weight goes up, MPG goes down (no surprise).

9.2.2 Making a time (index) plot

What if you plot just one?

```
R> plot(MPG)
```

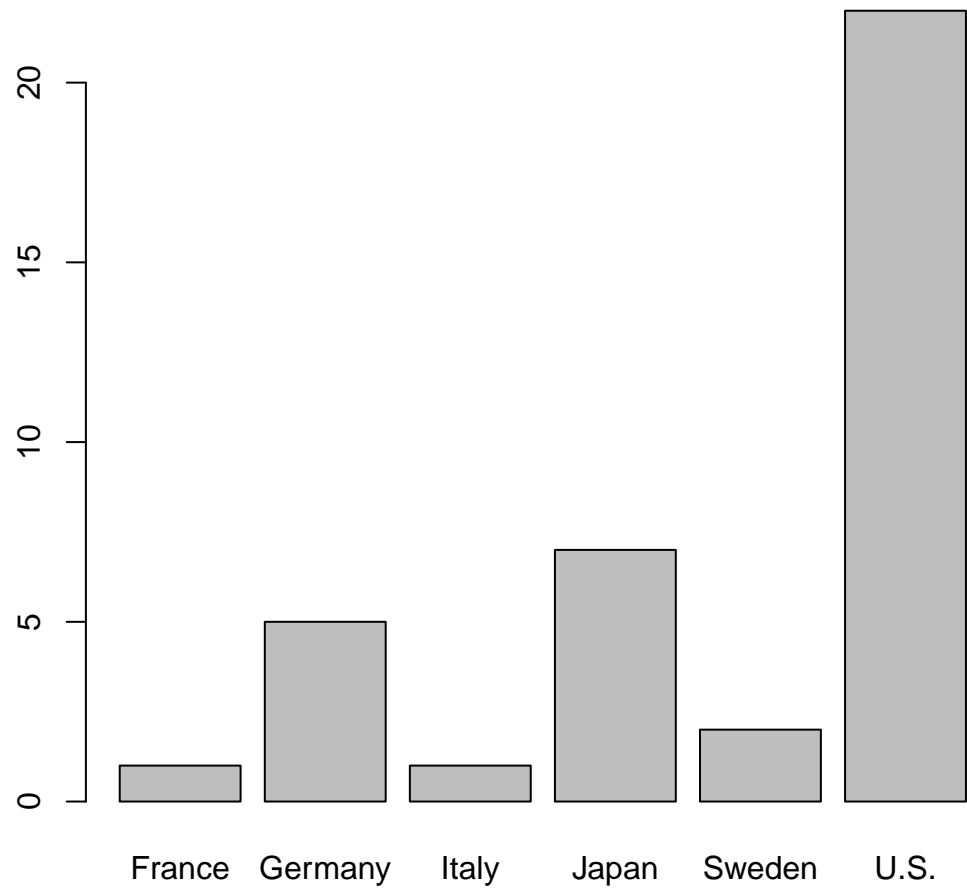
This is a plot of the gas mileages in the order that they appear in the file.⁶ There doesn't appear to be any trend with eg. lower-MPG cars coming later in the file, or anything like that.

9.2.3 Making a bar chart

How about plotting a categorical variable?

⁶A so-called **index plot**.

```
R> plot(Country)
```



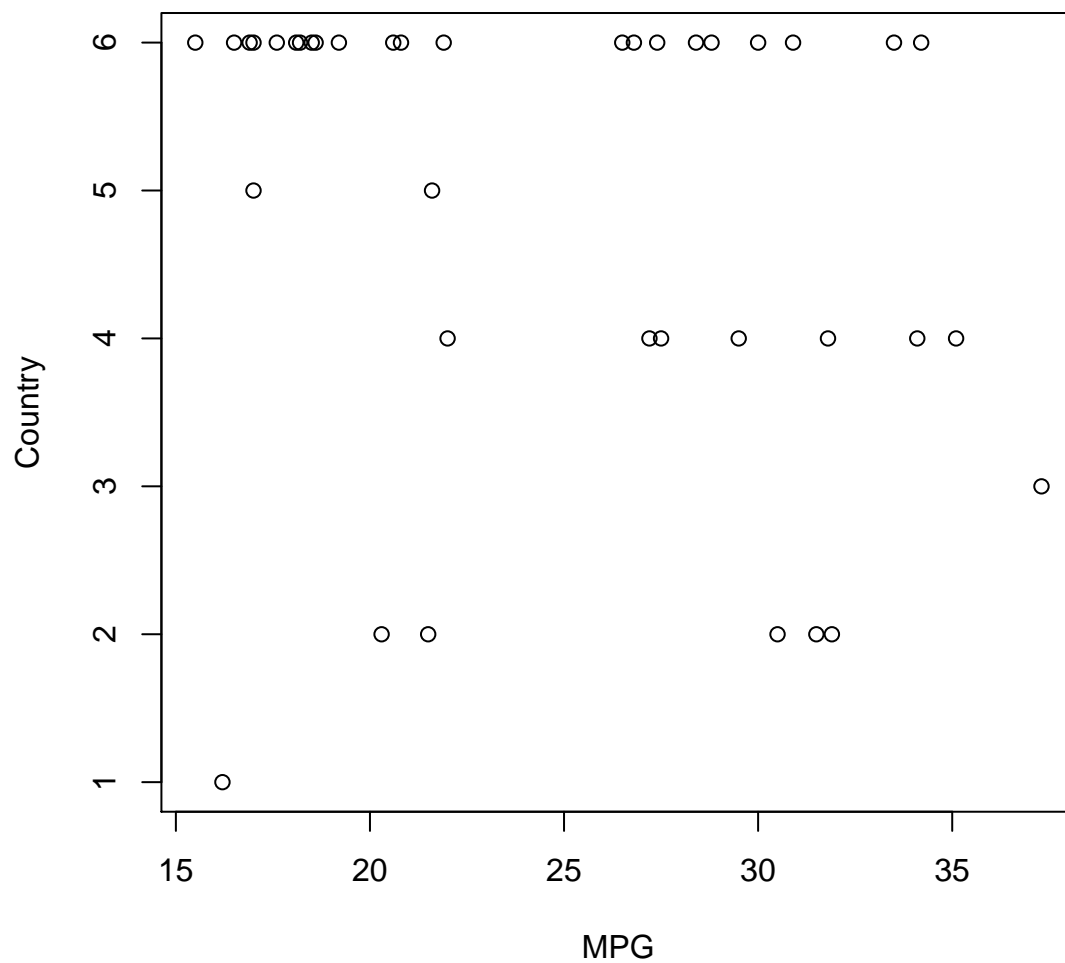
A bar chart. This one shows that most of the cars were American,⁷ and that France and Italy supplied only one car each.

⁷The data came from an American car magazine, so this is not a great surprise.

9.2.4 Making side-by-side boxplots

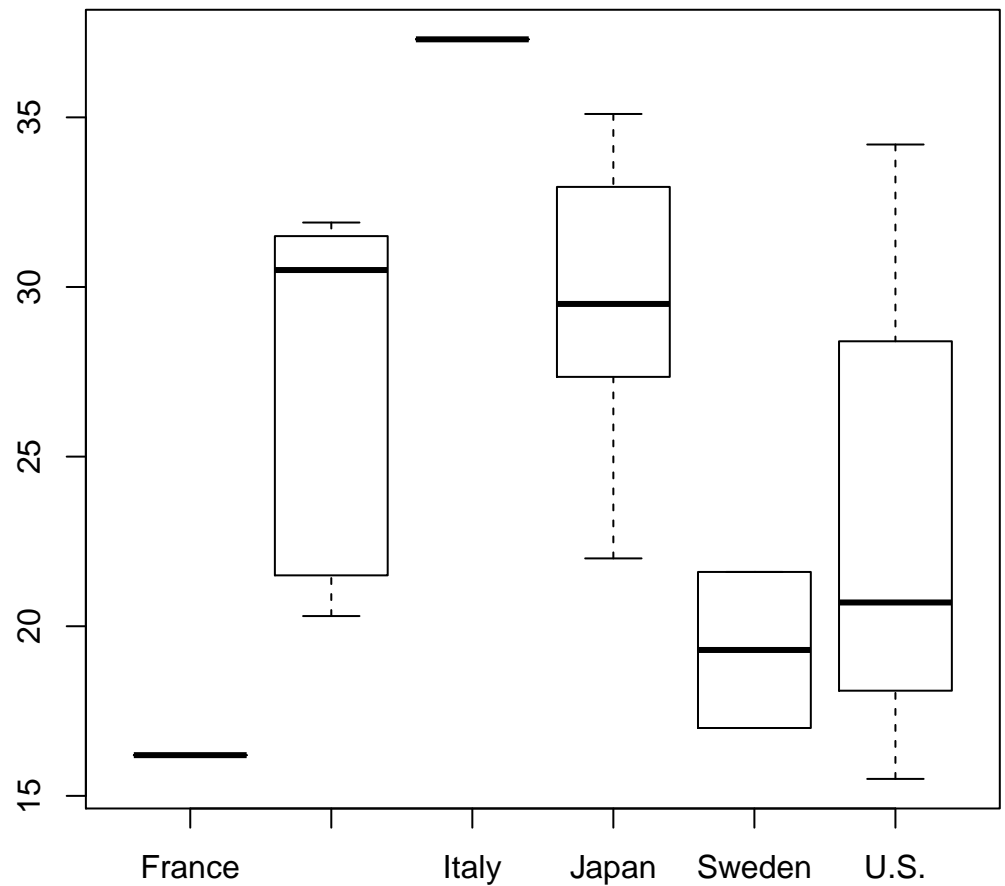
What if you plot a numerical variable against a categorical one?

```
R> plot(MPG, Country)
```



Not much. How about the other way around?

```
R> plot(Country, MPG)
```

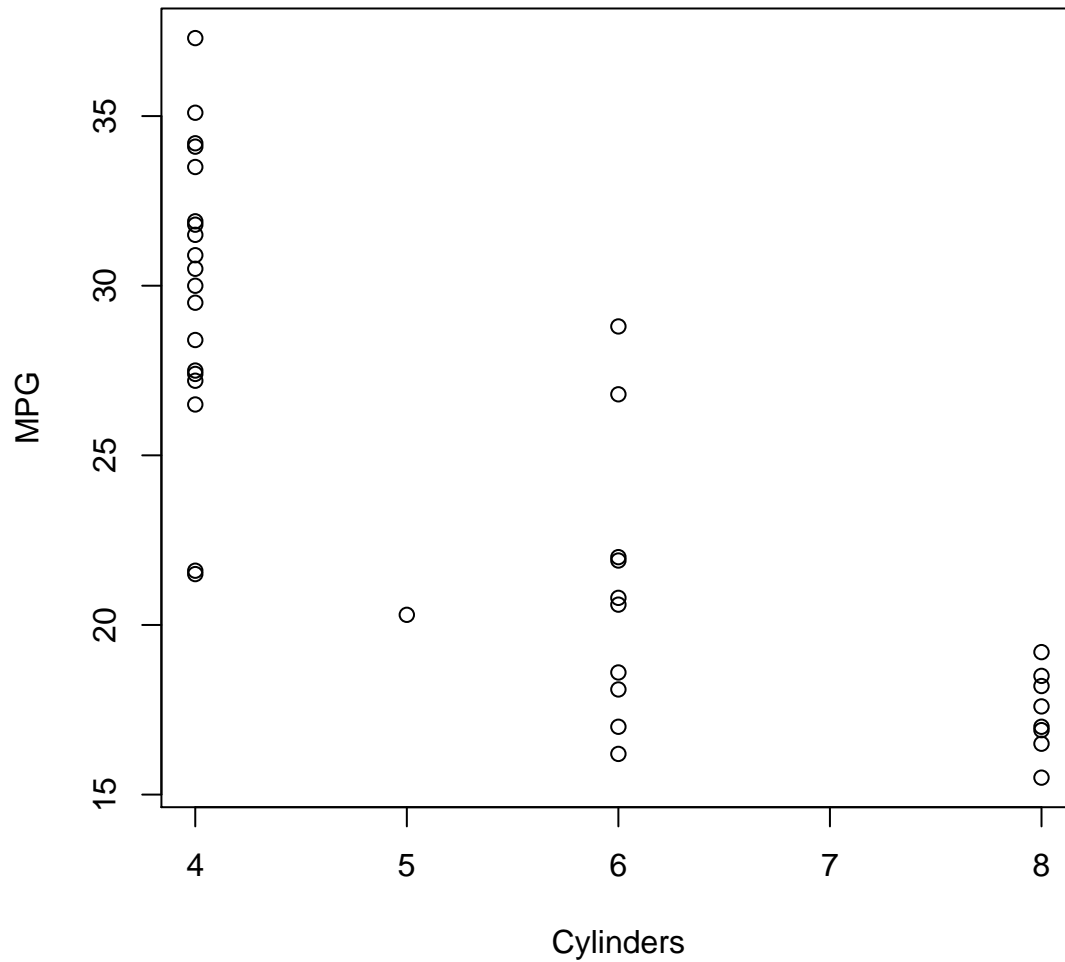


Now you see how the gas mileages vary by country. The US and Swedish cars tend to be low, while German⁸ and Japanese cars tend to be high. There isn't enough data from France and Italy to take those "boxplots" seriously. Recall from the bar chart of countries that France and Italy had only one car each.

Here's a plot of gas mileage against cylinders:

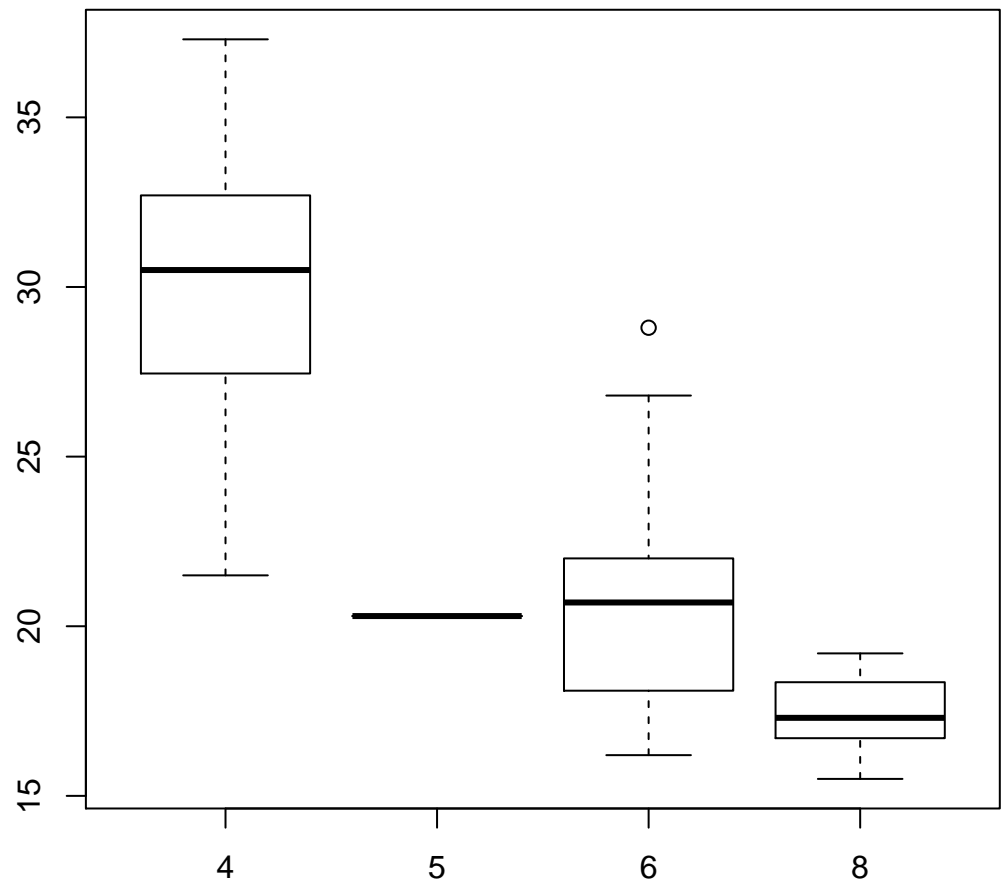
⁸The label is missing, because it is slightly too long to fit in the space.

```
R> plot(Cylinders,MPG)
```



This is really a scatterplot, since both variables are numerical. But `Cylinders` is discrete, which suggests that we could treat it as if it were a categorical one. R's term for a categorical variable is a **factor**, and you make a factor like this:

```
R> cyl.fac=factor(Cylinders)
R> plot(cyl.fac,MPG)
```



This is how to get a boxplot for each number of cylinders. You see, perhaps more clearly than from the scatterplot, that engines with more cylinders use more gas, and that 8-cylinder engines have uniformly bad gas mileage.

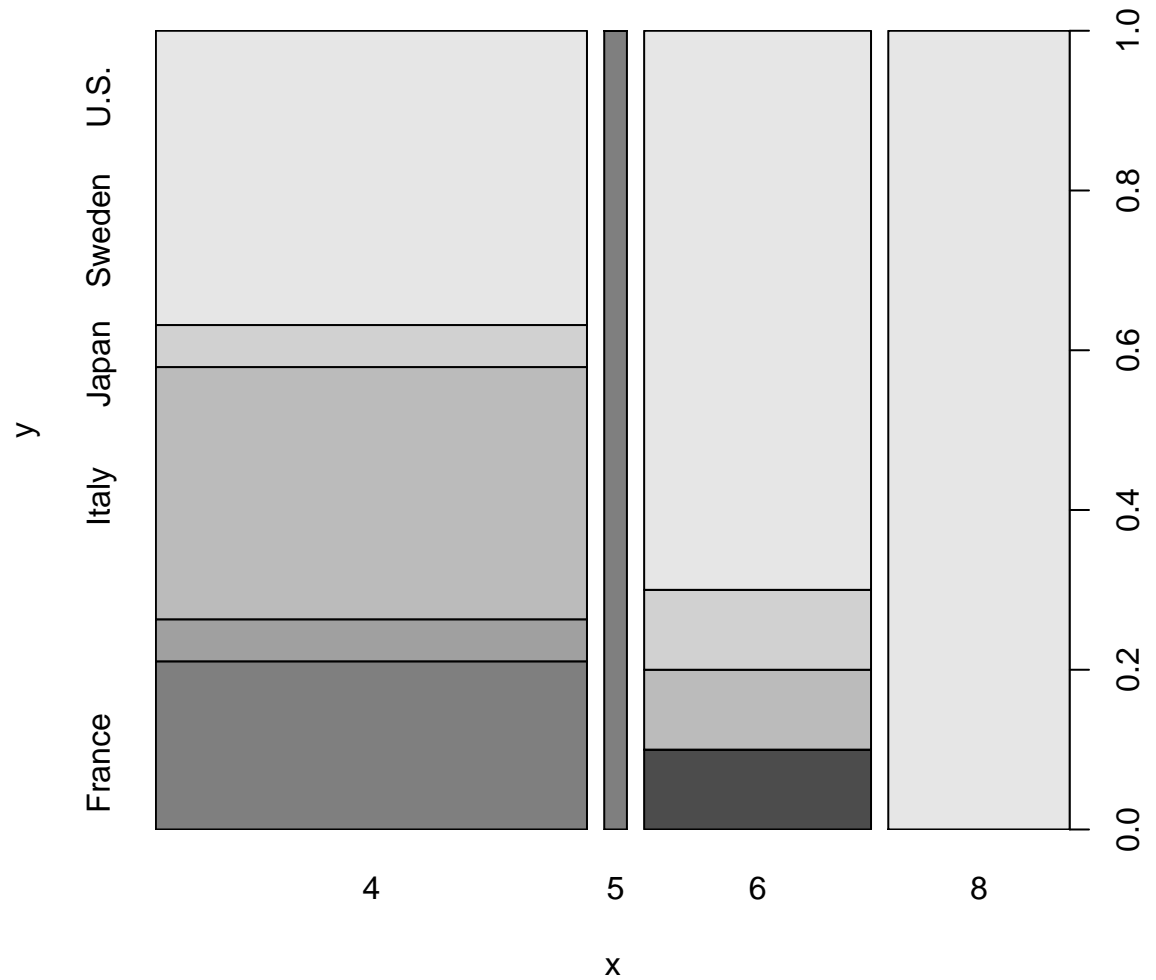
9.2.5 Plotting two factors against each other

What if you plot two factors against each other? This plot has a name, but I don't know what it is⁹. I made a contingency table above for comparison, which is actually upside down compared to the plot.

```
R> table(Country,cyl.fac)
R> plot(cyl.fac,Country)
```

| | cyl.fac | | | |
|---------|---------|---|---|---|
| Country | 4 | 5 | 6 | 8 |
| France | 0 | 0 | 1 | 0 |
| Germany | 4 | 1 | 0 | 0 |
| Italy | 1 | 0 | 0 | 0 |
| Japan | 6 | 0 | 1 | 0 |
| Sweden | 1 | 0 | 1 | 0 |
| U.S. | 7 | 0 | 7 | 8 |

⁹I discovered that this is called a spineplot, from looking at the help for `plot.factor`.



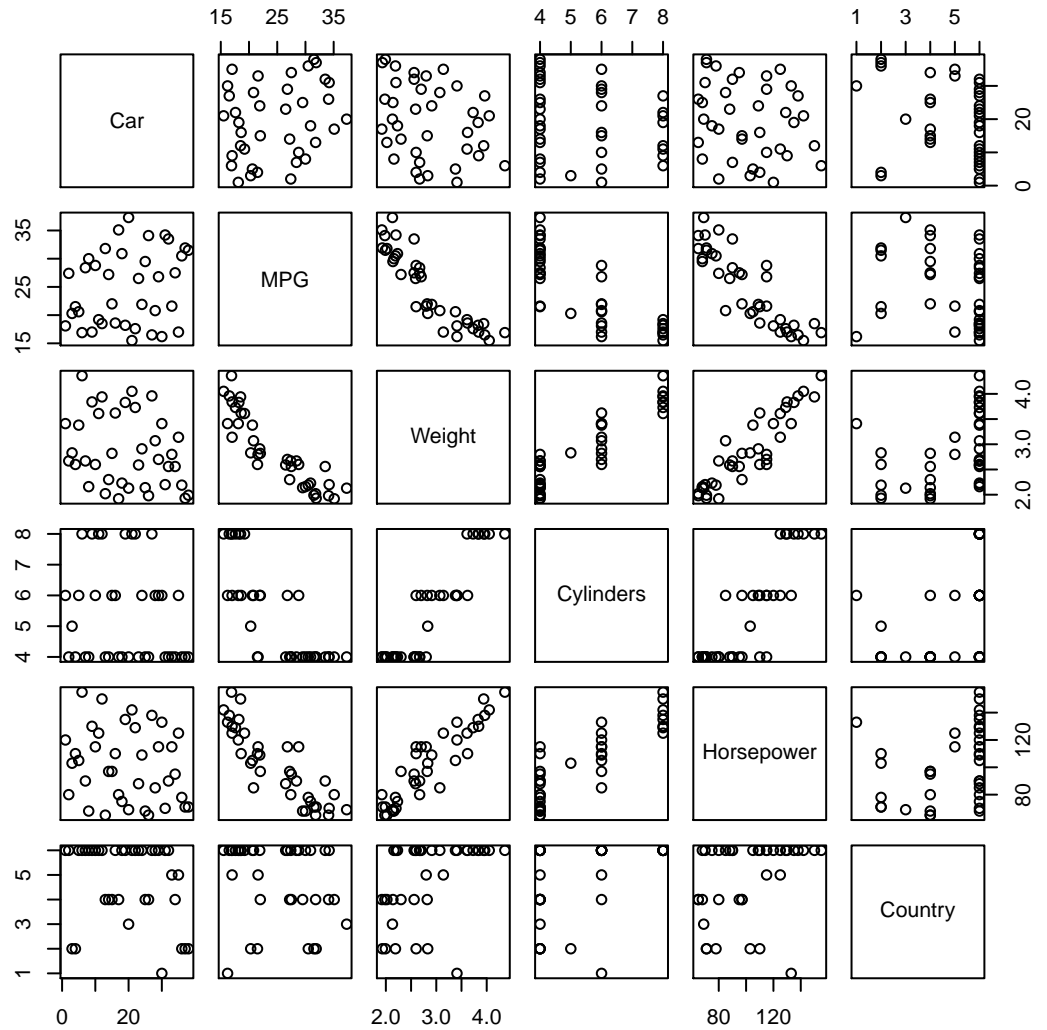
The size of the box on the plot reflects how many cars there are from that country with that many cylinders. The countries go from France (darkest) at the bottom to the US (lightest) at the top. The one French car has 6 cylinders (see the very dark rectangle above 6). Germany is next darkest; Germany has some 4-cylinder cars (confusingly next to the word “France” above 4) and the one 5-cylinder car, an Audi. The US has some of the 4-cylinder cars, most of the 6-cylinder cars and *all* of the 8-cylinder cars.

I don't know whether this plot really offers anything beyond what the contingency table does, but that's not really the point here: the point is that what you feed into `plot` determines the nature of what you get out.

9.2.6 Making a pairs plot

As a final fling, what happens if you `plot` a data frame?

```
R> plot(cars)
```

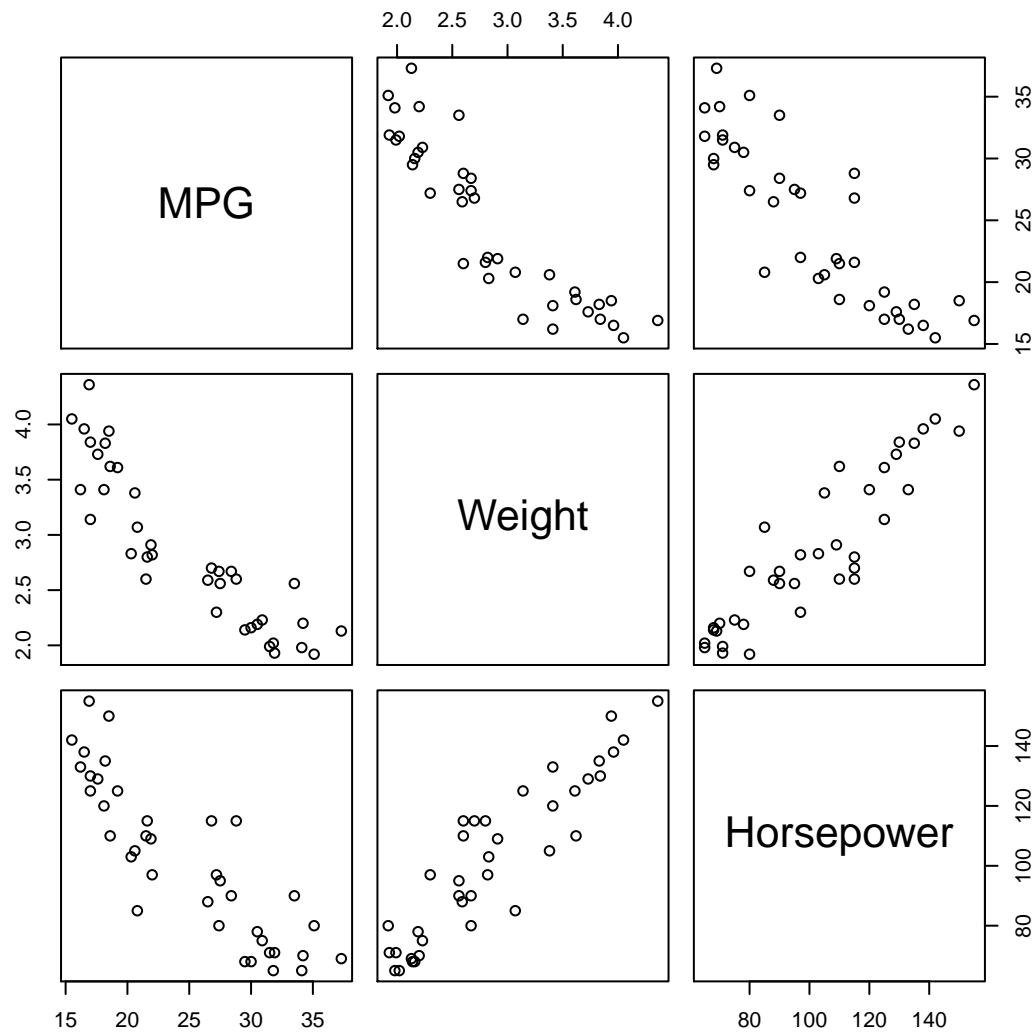


R turns all the categorical variables into numbers¹⁰ and makes a “scatterplot matrix”, which is an array of scatterplots of each variable against each other variable. Thus, for example (looking in the second row), MPG goes down as Weight or Cylinders or Horsepower goes up.

¹⁰Specifically, R takes the alphabetical list of categories, turns the first one into 1, the second into 2, and so on. For our countries, France becomes 1 and the US becomes 6. This isn't usually very meaningful.

If you select just some columns from a data frame, you still have a data frame, so plotting that will give you a pairs plot also:

```
R> plot(cars[,c(2,3,5)])
```



There's a strong positive relationship between `Weight` and `Horsepower`, as you would guess.

Let's tidy up after ourselves:

```
R> detach(cars)
```

9.2.7 Different plot methods

There are actually a whole lot of different functions that R will call depending on what kind of thing you feed into `plot`. To see what they're called, you can type

```
R> methods(plot)

[1] plot.acf*           plot.data.frame*    plot.decomposed.ts*
[4] plot.default        plot.dendrogram*    plot.density
[7] plot.ecdf           plot.factor*        plot.formula*
[10] plot.function       plot.hclust*        plot.histogram*
[13] plot.HoltWinters*    plot.isoreg*        plot.lm
[16] plot.medpolish*     plot.mlm            plot.ppr*
[19] plot.prcomp*        plot.princomp*      plot.profile.nls*
[22] plot.spec           plot.stepfun        plot.stl*
[25] plot.table*         plot.ts             plot.tskernel*
[28] plot.TukeyHSD
```

Non-visible functions are asterisked

You can get help on specific ones by asking for help on the full name. This is the help for `plot.default`.¹¹ It is rather long!

```
plot.default          package:graphics          R Documentation
```

The Default Scatterplot Function

Description:

Draw a scatter plot with decorations such as axes and titles in the active graphics window.

Usage:

```
## Default S3 method:
plot(x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
     log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
     ann = par("ann"), axes = TRUE, frame.plot = axes,
     panel.first = NULL, panel.last = NULL, asp = NA, ...)
```

Arguments:

`x`, `y`: the 'x' and 'y' arguments provide the x and y coordinates for the plot. Any reasonable way of defining the coordinates is acceptable. See the function 'xy.coords' for details. If

¹¹Which is what gets called if none of the other plots apply.

supplied separately, they must be of the same length.

type: 1-character string giving the type of plot desired. The following values are possible, for details, see 'plot': "p" for points, "l" for lines, "b" for both points and lines, "c" for empty points joined by lines, "o" for overplotted points and lines, "s" and "S" for stair steps and "h" for histogram-like vertical lines. Finally, "n" does not produce any points or lines.

xlim: the x limits (x1, x2) of the plot. Note that 'x1 > x2' is allowed and leads to a 'reversed axis'.

The default value, 'NULL', indicates that the range of the finite values to be plotted should be used.

ylim: the y limits of the plot.

log: a character string which contains "x" if the x axis is to be logarithmic, "y" if the y axis is to be logarithmic and "xy" or "yx" if both axes are to be logarithmic.

main: a main title for the plot, see also 'title'.

sub: a sub title for the plot.

xlab: a label for the x axis, defaults to a description of 'x'.

ylab: a label for the y axis, defaults to a description of 'y'.

ann: a logical value indicating whether the default annotation (title and x and y axis labels) should appear on the plot.

axes: a logical value indicating whether both axes should be drawn on the plot. Use graphical parameter "xaxt" or "yaxt" to suppress just one of the axes.

frame.plot: a logical indicating whether a box should be drawn around the plot.

panel.first: an 'expression' to be evaluated after the plot axes are set up but before any plotting takes place. This can be useful for drawing background grids or scatterplot smooths. Note that this works by lazy evaluation: passing this argument from other 'plot' methods may well not work since it may be evaluated too early.

`panel.last`: an expression to be evaluated after plotting has taken place but before the axes, title and box are added. See the comments about `'panel.first'`.

`asp`: the y/x aspect ratio, see `'plot.window'`.

`...`: other graphical parameters (see `'par'` and section 'Details' below).

Details:

Commonly used graphical parameters are:

`'col'` The colors for lines and points. Multiple colors can be specified so that each point can be given its own color. If there are fewer colors than points they are recycled in the standard fashion. Lines will all be plotted in the first colour specified.

`'bg'` a vector of background colors for open plot symbols, see `'points'`. Note: this is *not* the same setting as `'par("bg")'`.

`'pch'` a vector of plotting characters or symbols: see `'points'`.

`'cex'` a numerical vector giving the amount by which plotting characters and symbols should be scaled relative to the default. This works as a multiple of `'par("cex")'`. `'NULL'` and `'NA'` are equivalent to `'1.0'`. Note that this does not affect annotation: see below.

`'lty'` a vector of line types, see `'par'`.

`'cex.main'`, `'col.lab'`, `'font.sub'`, etc settings for main- and sub-title and axis annotation, see `'title'` and `'par'`.

`'lwd'` a vector of line widths, see `'par'`.

Note:

The presence of `'panel.first'` and `'panel.last'` is a historical anomaly: default plots do not have 'panels', unlike e.g. 'pairs' plots. For more control, use lower-level plotting functions: `'plot.default'` calls in turn some of `'plot.new'`, `'plot.window'`, `'plot.xy'`, `'axis'`, `'box'` and `'title'`, and plots can be built up by

calling these individually, or by calling `'plot(type = "n")'` and adding further elements.

References:

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Cleveland, W. S. (1985) *The Elements of Graphing Data*. Monterey, CA: Wadsworth.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also:

`'plot'`, `'plot.window'`, `'xy.coords'`.

Examples:

```
Speed <- cars$speed
Distance <- cars$dist
plot(Speed, Distance, panel.first = grid(8, 8),
     pch = 0, cex = 1.2, col = "blue")
plot(Speed, Distance,
     panel.first = lines(stats::lowess(Speed, Distance), lty = "dashed"),
     pch = 0, cex = 1.2, col = "blue")

## Show the different plot types
x <- 0:12
y <- sin(pi/5 * x)
op <- par(mfrow = c(3,3), mar = .1+ c(2,2,3,1))
for (tp in c("p","l","b", "c","o","h", "s","S","n")) {
  plot(y ~ x, type = tp, main = paste0("plot(*, type = \"", tp, "\")"))
  if(tp == "S") {
    lines(x, y, type = "s", col = "red", lty = 2)
    mtext("lines(*, type = \"s\", ...)", col = "red", cex = 0.8)
  }
}
par(op)

##--- Log-Log Plot with custom axes
lx <- seq(1, 5, length = 41)
yl <- expression(e^{-frac(1,2) * {log[10](x)}^2})
y <- exp(-.5*lx^2)
op <- par(mfrow = c(2,1), mar = par("mar")+c(0,1,0,0))
plot(10^lx, y, log = "xy", type = "l", col = "purple",
     main = "Log-Log plot", ylab = yl, xlab = "x")
```

```

plot(10^lx, y, log = "xy", type = "o", pch = ".", col = "forestgreen",
     main = "Log-Log plot with custom axes", ylab = yl, xlab = "x",
     axes = FALSE, frame.plot = TRUE)
my.at <- 10^(1:5)
axis(1, at = my.at, labels = formatC(my.at, format = "fg"))
}
par(op)

##--- Log-Log Plot with custom axes
lx <- seq(1, 5, length = 41)
yl <- expression(e^{-frac(1,2) * {log[10](x)}^2})
y <- exp(-.5*lx^2)
op <- par(mfrow = c(2,1), mar = par("mar")+c(0,1,0,0))
plot(10^lx, y, log = "xy", type = "l", col = "purple",
     main = "Log-Log plot", ylab = yl, xlab = "x")
plot(10^lx, y, log = "xy", type = "o", pch = ".", col = "forestgreen",
     main = "Log-Log plot with custom axes", ylab = yl, xlab = "x",
     axes = FALSE, frame.plot = TRUE)
my.at <- 10^(1:5)
axis(1, at = my.at, labels = formatC(my.at, format = "fg"))
at.y <- 10^(-5:-1)
axis(2, at = at.y, labels = formatC(at.y, format = "fg"), col.axis = "red")
par(op)

```

The point is not to read the whole help file, since that will only get you confused. Usually, you will have some kind of question in mind, like “how do I set the limits of the axes to be something other than what R chooses?” You might find something that helps you in the examples (in this case, I couldn’t), or you can go down the list of “arguments” (things that can be fed in) and find `xlim` and `ylim`. These have a default value of `NULL`, as you see from the Usage; in this case, that means “let R choose”. We’ll investigate this more later when we look specifically at plotting.

9.2.8 Finding function names

Another useful function is `apropos`. This can give you a hint when you’ve forgotten the name of a function. Let’s say you want to draw a histogram. Is the function `histogram` or something else? Try the full name first:

```

R> apropos("histogram")

character(0)

```

This doesn’t produce anything. Let’s try `hist`:¹²

¹²Trying the shortest name you think it might be improves your chances of finding something.


```
R> apropos("hist")
```

```
[1] "hist"          "hist.default" "history"      "loadhistory"  "savehistory"
```

The first one of those looks the most plausible. Let's try looking at the help for `hist` and see whether that's the one. Here's the top of the help file:

```
hist                                package:graphics                R Documentation
```

```
Histograms
```

```
Description:
```

```
The generic function 'hist' computes a histogram of the given data
values.  If 'plot = TRUE', the resulting object of class
'histogram' is plotted by 'plot.histogram', before it is
returned.
```

```
Usage:
```

```
hist(x, ...)

## Default S3 method:
hist(x, breaks = "Sturges",
     freq = NULL, probability = !freq,
     include.lowest = TRUE, right = TRUE,
     density = NULL, angle = 45, col = NULL, border = NULL,
     main = paste("Histogram of" , xname),
     xlim = range(breaks), ylim = NULL,
     xlab = xname, ylab,
     axes = TRUE, plot = TRUE, labels = FALSE,
     nclass = NULL, warn.unused = TRUE, ...)
```

That's the one.¹³

R being R, there are many things you can change, but they all have defaults, so simply saying something like `hist(x)` will get you a histogram. We'll look at that later too.

¹³Or you can throw something like `draw histogram r` into your favourite search engine. When I did this, the first link that came back was <http://www.harding.edu/fmccown/r/>, which showed me how to draw histograms, after I paged down a bit.

9.3 Making and annotating plots

9.3.1 Introduction

While we were looking at the help file for `plot`, I showed you a lot of different plots, all obtained with `plot`. Even though they were a lot of these, they don't cover everything that you might want to do on a plot, like:

- adding a title to a plot
- changing the axis labels
- adding points to a plot
- adding lines or curves to a plot
- changing line types
- changing the size of the axes
- adding text to a plot, eg. to label points
- colouring groups of points
- using different plotting symbols
- changing the size of text
- types of plot
- plotting nothing to add lines/points
- arrays of plots

Let's use our car MPG vs. `Weight` as the basis, and illustrate how these things might go. They can also be combined with each other, as you need. I'll illustrate them mainly one at a time, though, so that you can see the effects.

First, we need to `attach` the data frame again:

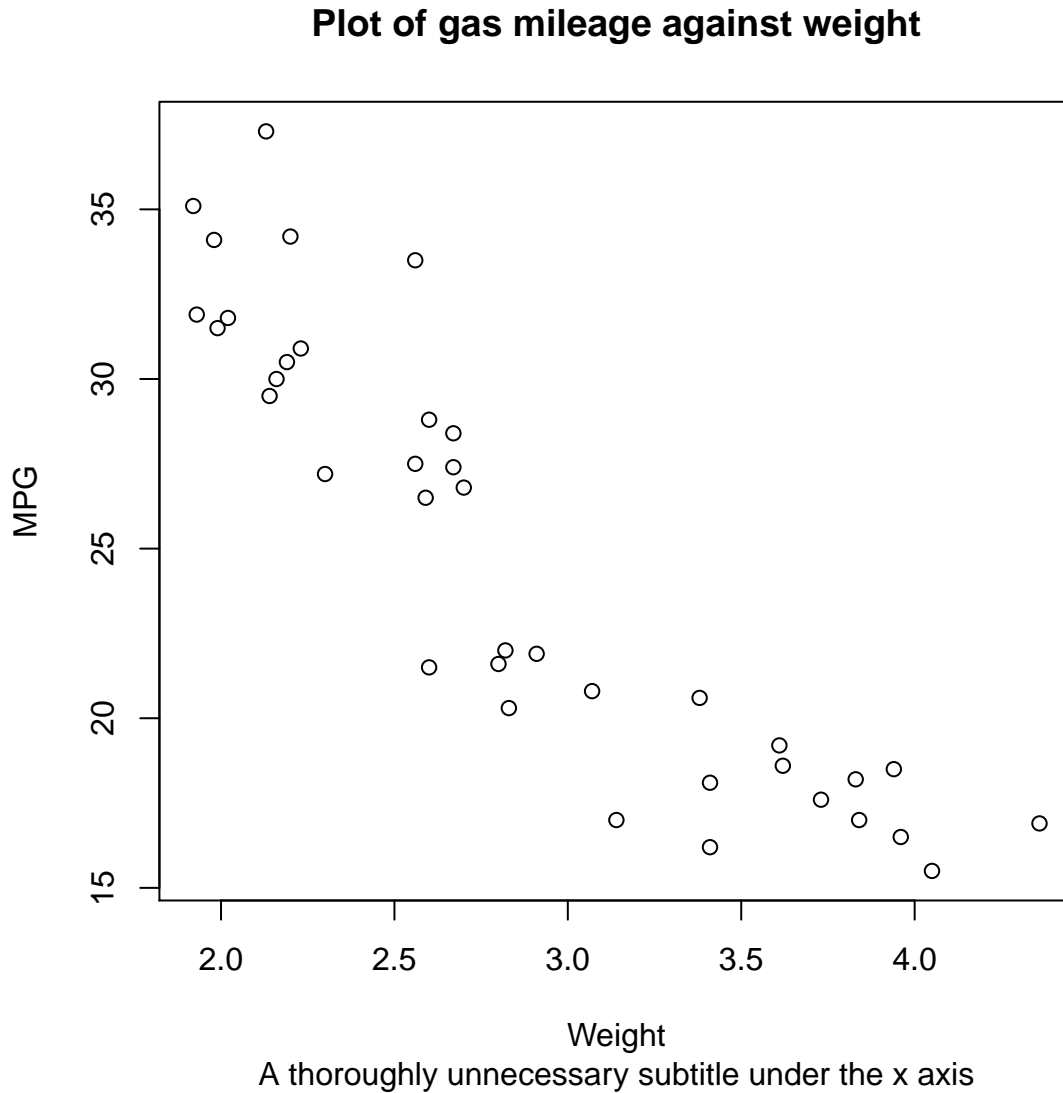
```
R> attach(cars)
```

To get help for these, you need to look under the help for `par` (these are so-called "graphical parameters").

9.3.2 Adding a title to a plot

Starting with titles:

```
R> plot(Weight,MPG,main="Plot of gas mileage against weight",  
R>      sub="A thoroughly unnecessary subtitle under the x axis")
```



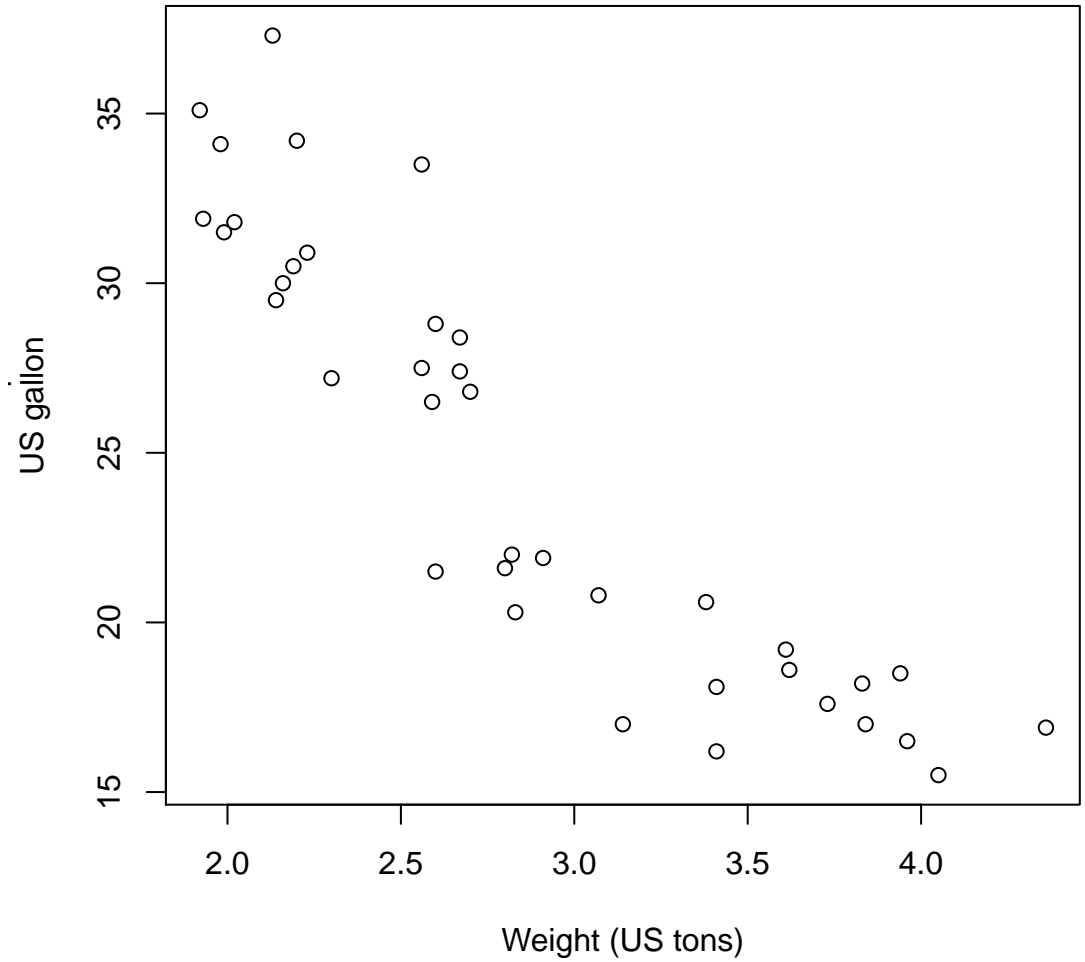
In a report, you'd usually put a plot in a Figure and give it a caption rather than a title; in that case, you would do the captioning using eg. `Word` and not use R's `main` at all.

9.3.3 Axis labels

R uses the names of the variables as axis labels, which is quite often perfectly good, but `xlab` and `ylab` allow you to change them. Beware of making the new

labels too long, or else parts of them can disappear:

```
R> plot(Weight,MPG,xlab="Weight (US tons)",ylab="Miles per  
R> US gallon")
```

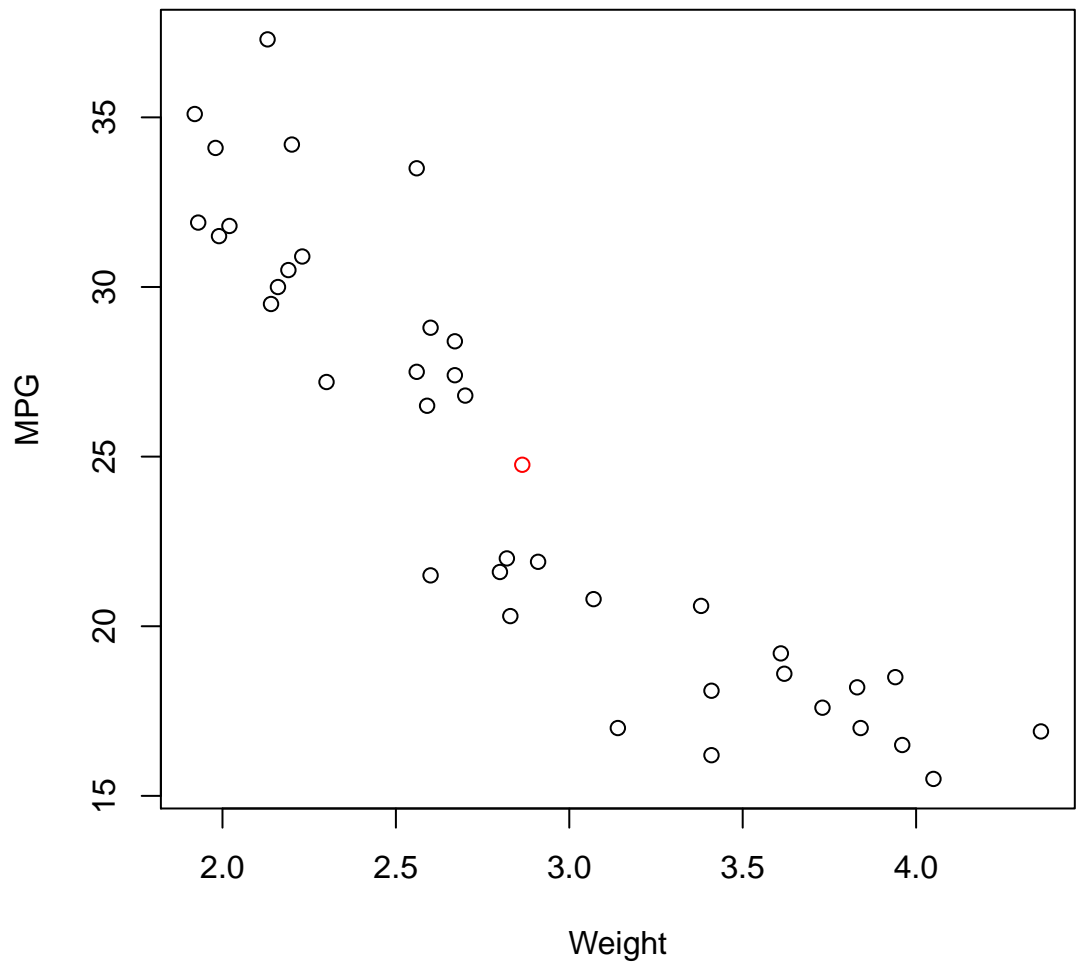


9.3.4 Adding points to a plot

Now, let's work out the mean `Weight` and `MPG`, and put that on the plot in red.¹⁴

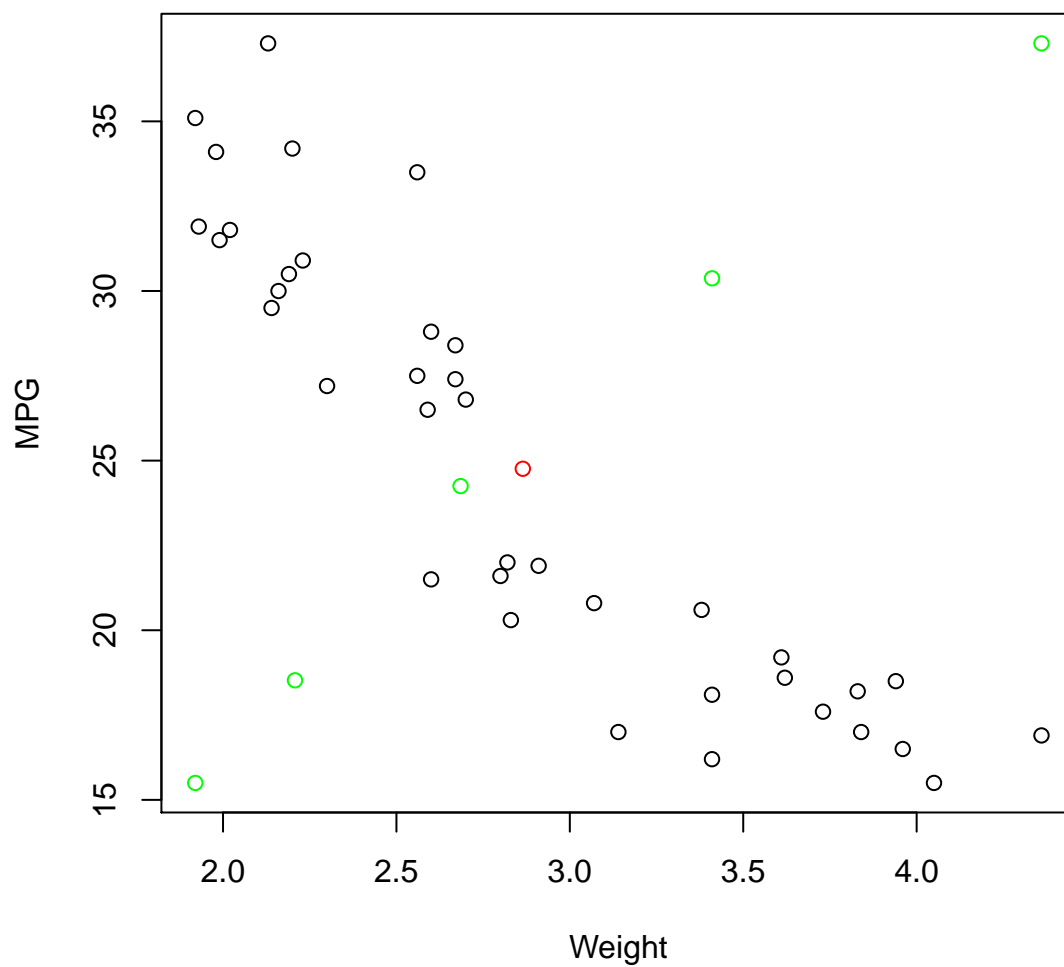
```
R> mean.weight=mean(Weight)
R> mean.MPG=mean(MPG)
R> plot(Weight,MPG)
R> points(mean.weight,mean.MPG,col="red")
```

¹⁴Yes, I'm getting ahead of myself with the colour thing, but otherwise you wouldn't be able to see where the point goes.



The `quantile` function works out (by default) the five-number summary for the variable you feed it. Let's add them to the scatterplot, in green. `col` does colour, as you have no doubt already figured out:

```
R> q.weight=quantile(Weight)
R> q.MPG=quantile(MPG)
R> plot(Weight,MPG)
R> points(mean.weight,mean.MPG,col="red")
R> points(q.weight,q.MPG,col="green")
```



This makes no sense! But the point was to illustrate that you can feed `points` two vectors of values and it will plot them. There were five numbers in each five-number summary, and therefore five green points.

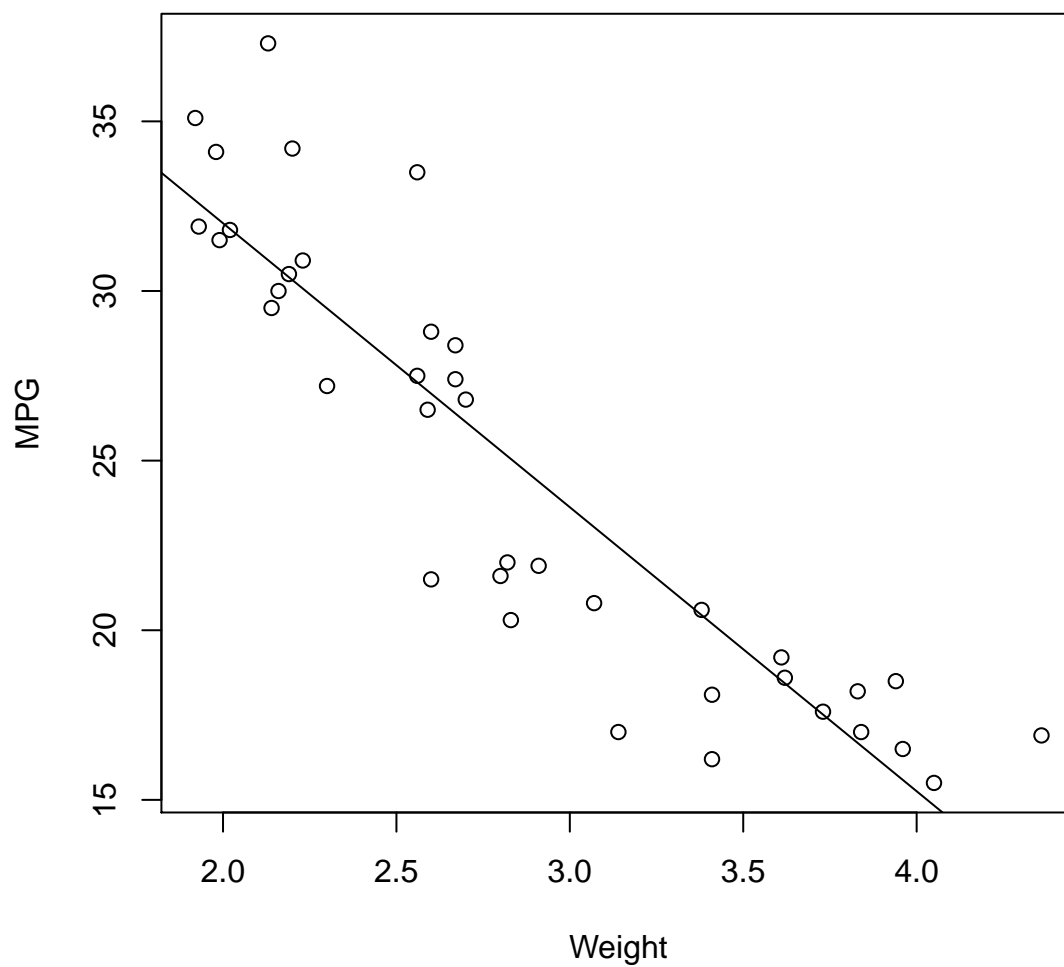
As you see, you can have as many `points` lines as you want.

9.3.5 Adding a regression line to a plot

An obvious thing to want to add to a plot like this is a regression line. R appears to make this more work than needed, but there is a common theme here (which we see later when we look at regression): obtain a “fitted model object”, and then use the fitted model object in a suitable way. Here `lm` stands for “linear model”, which includes linear and multiple regression. The bit inside the `lm` with a squiggle in it is called a **model formula**: the response goes on the left of the squiggle, and the explanatory variable(s) go on the right.

`abline` is used to add straight lines to a plot:

```
R> plot(Weight,MPG)
R> MPG.lm=lm(MPG~Weight)
R> abline(MPG.lm)
```

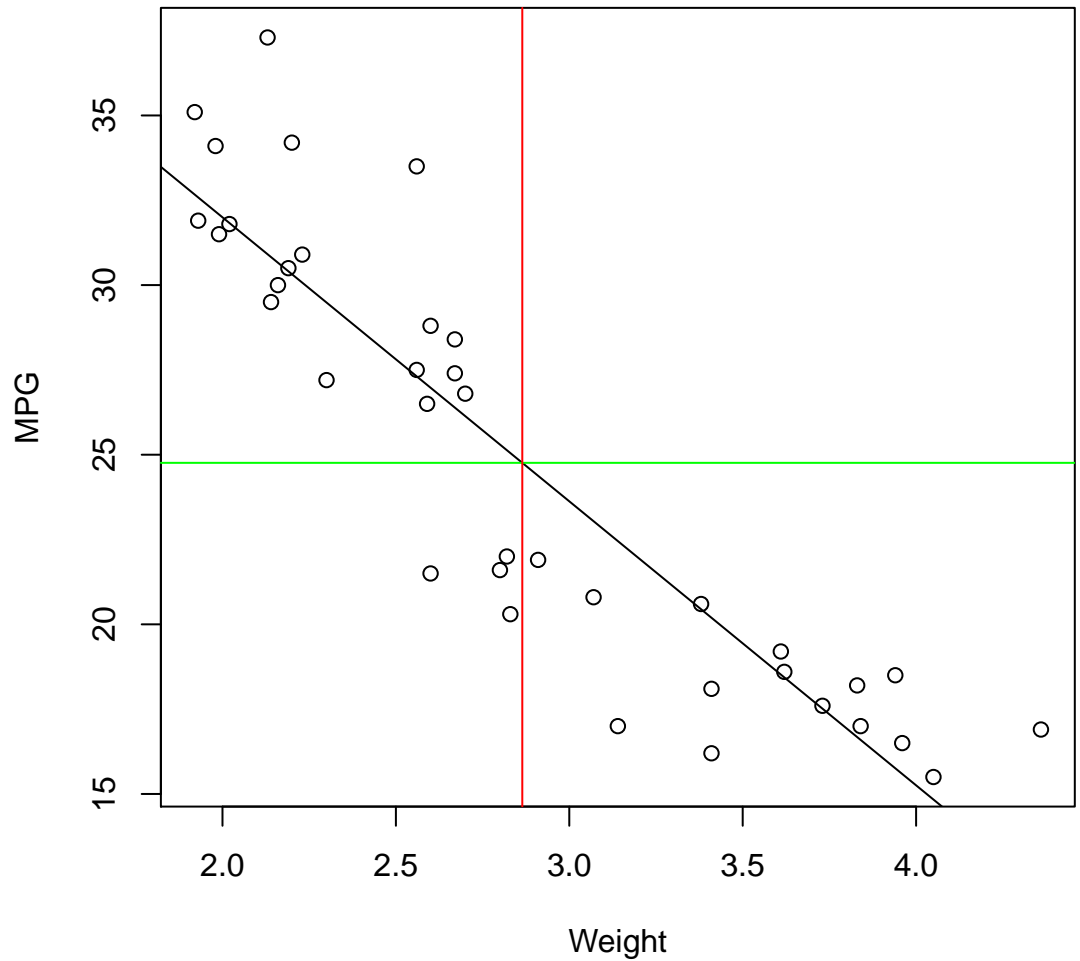



9.3.6 Horizontal and vertical lines

`abline` can also be used to add horizontal and vertical lines, like this:

```
R> plot(Weight, MPG)
R> MPG.lm=lm(MPG~Weight)
R> abline(MPG.lm)
R> abline(h=mean.MPG, col="green")
```

```
R> abline(v=mean.weight,col="red")
```



I've added a horizontal line through the mean of MPG (using the variable I calculated earlier) and a vertical line through the mean of `Weight`, in red. The place where these lines meet is on the regression line. This is a consequence of how the regression line always passes through the “point of averages” (\bar{x}, \bar{y}) . You also see that most of the data points are top left and bottom right, which would indicate a negative correlation. In fact, the correlation is

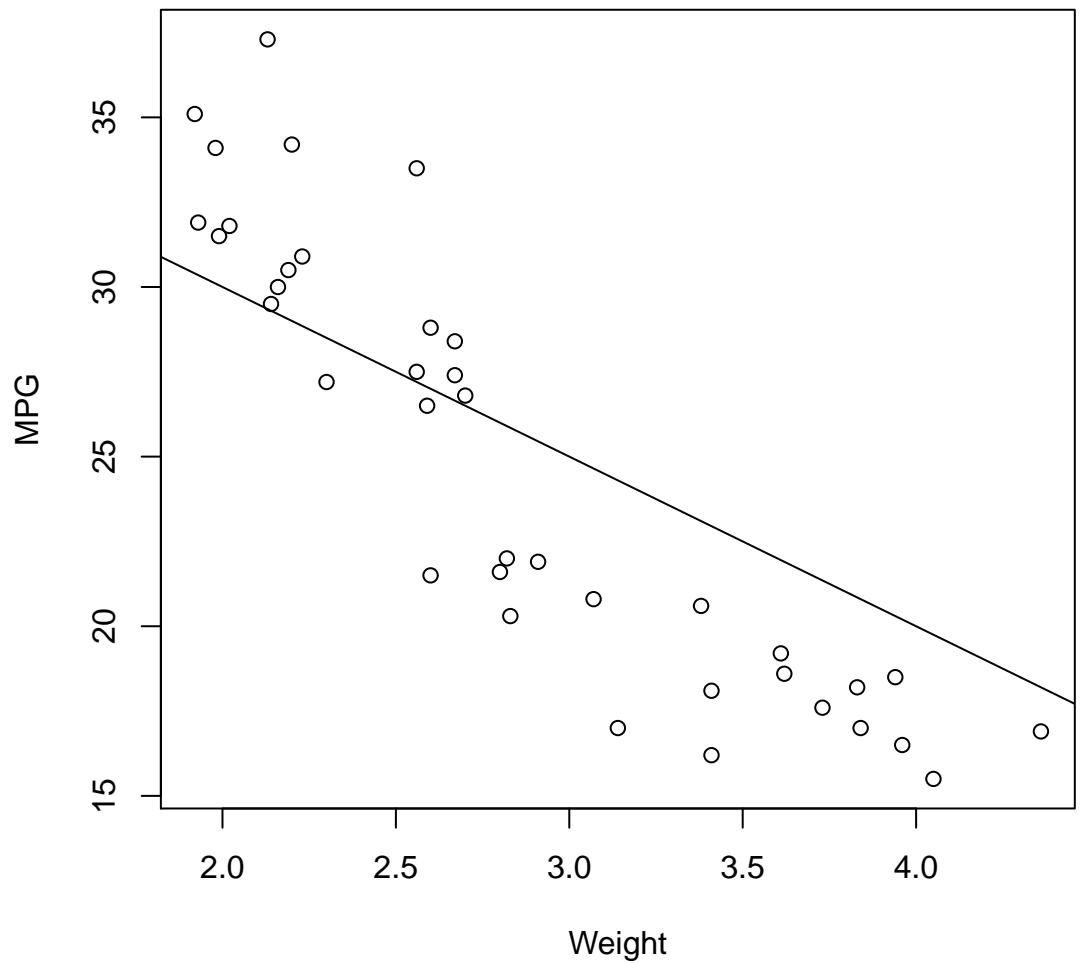
```
R> cor(Weight,MPG)
```

```
[1] -0.9030111
```

9.3.7 Adding lines with given intercept and slope

`abline` can also handle lines of any intercept and slope. Say we wanted to look at a line with intercept 40 and slope -5 :

```
R> plot(Weight,MPG)  
R> abline(a=40,b=-5)
```

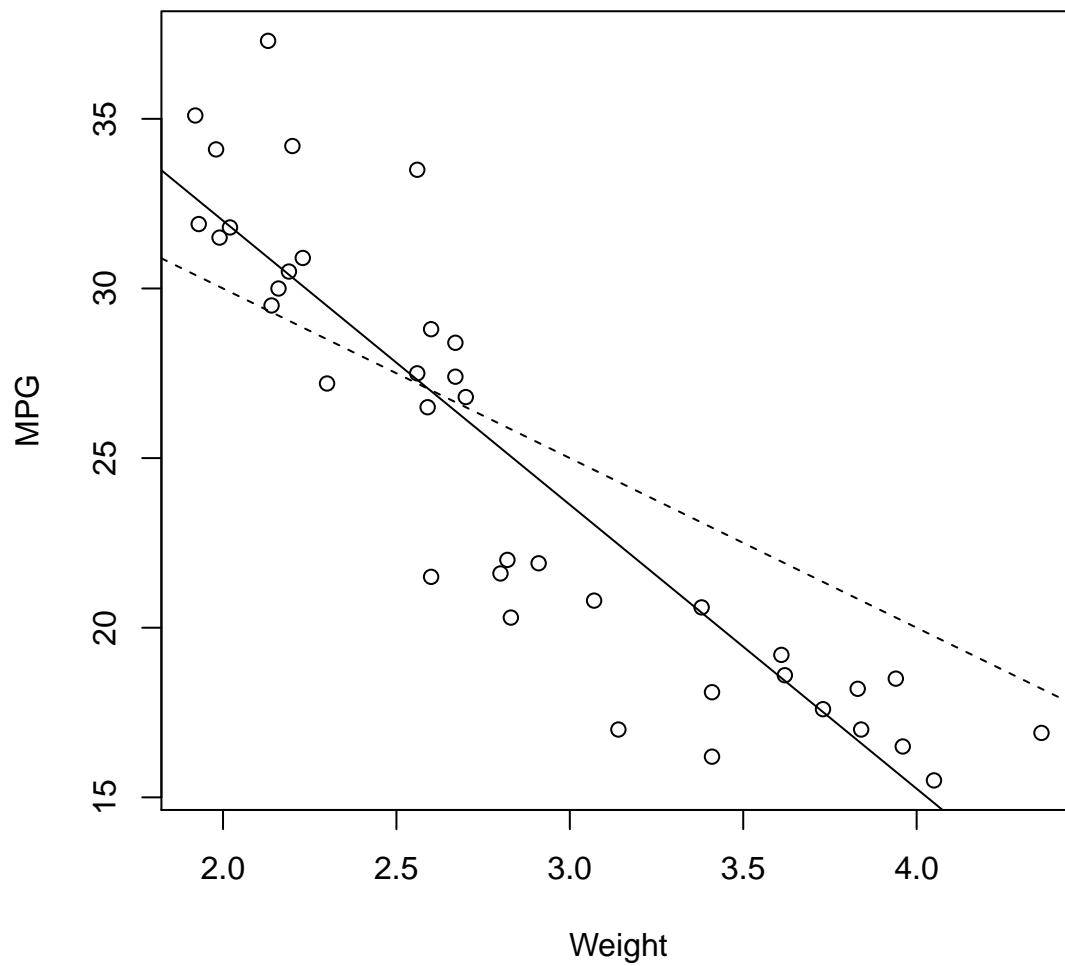


This also goes through the data reasonably well.

9.3.8 Different line types

Let's plot both lines from above at once, but we'll need to distinguish them. One way would be to make them different colours, but we'll illustrate how to make the regression line solid (which is the default) and the other line dashed. Note how I'm re-using my fitted model object from earlier:

```
R> plot(Weight,MPG)
R> abline(MPG.lm)
R> abline(a=40,b=-5,lty="dashed")
```



The regression line is quite a bit steeper (goes down faster) than the line with slope -5 .

Choices for `lty` include `dashed`, `dotted` and `dotdash`. Or you can specify the line type as a number, where 1 is `solid`, 2 is `dashed` and so on.

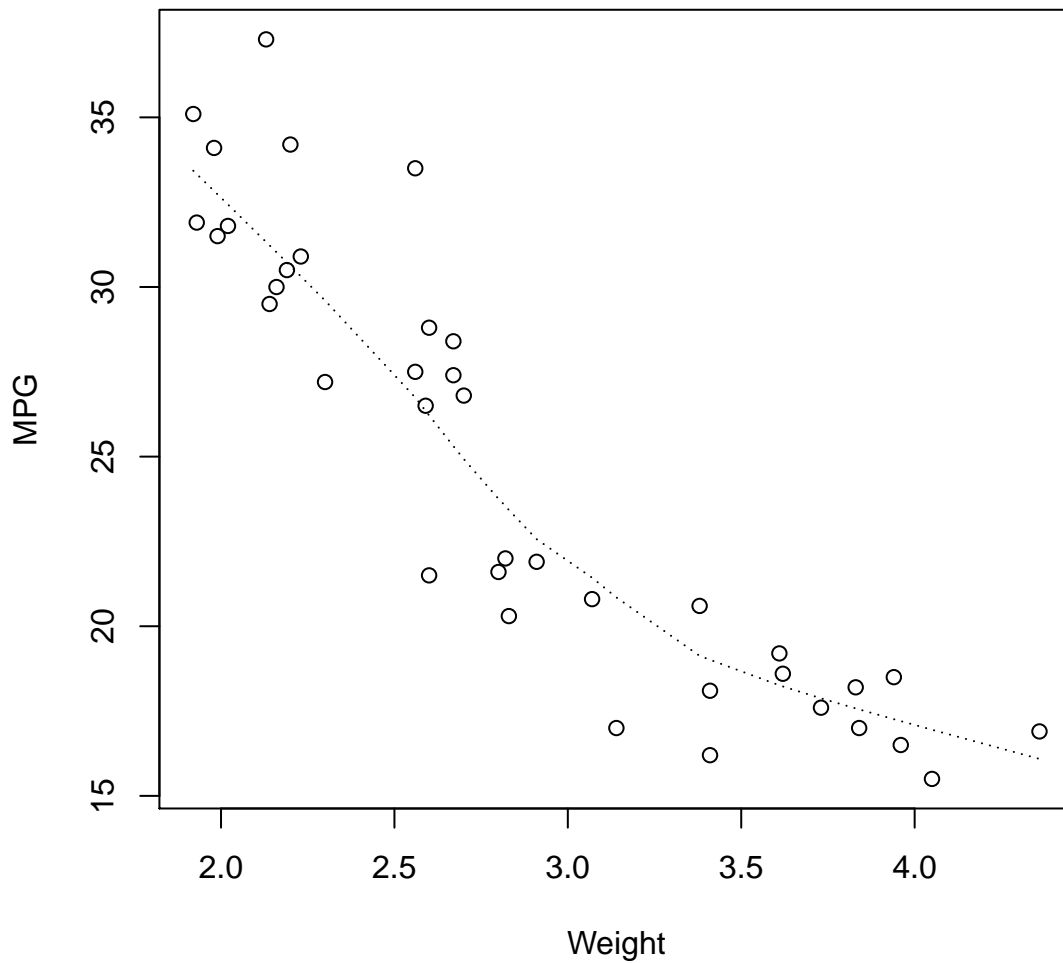
9.3.9 Plotting curves

For plotting curves, you use `lines`.¹⁵ One of my favourite things to plot on a scatter plot is a **lowess curve**. This is a non-linear trend — “lowess” stands for locally-weighted least squares — which downweights points far away from where you are now, and also downweights outliers. So it’s kind of like a trend version of the median. It’s a nice thing to put on a scatter plot to get a sense of where the trend is going, without constraining it to be linear.

Here’s a lowess curve on our scatterplot, drawn dotted, just because.

```
R> plot(Weight,MPG)
R> lines(lowess(Weight,MPG),lty="dotted")
```

¹⁵Yes, I know.



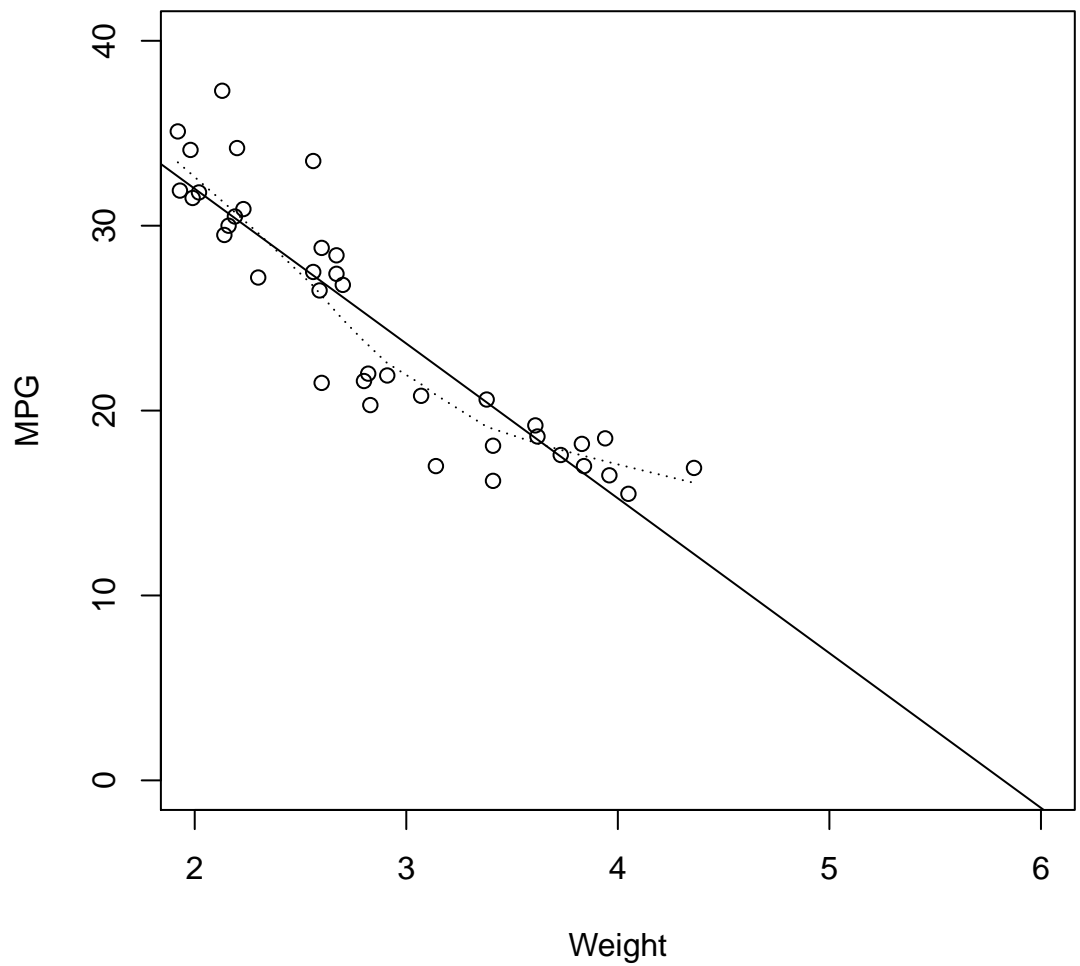
According to the lowess curve, the trend is more or less linear up to a weight of about 3 tons, and then the decrease slows down.

9.3.10 Extending (or shrinking) axes

Let's compare the lowess curve with the regression line. I use this example in STAB22 to illustrate the perils of extrapolation, by asking "what would we predict the gas mileage to be for a car that weighs 6 tons?" So let's extend the

x axis out to 6 tons, and extend the gas mileage down to 0, which illustrates the next thing. Axes are extended (or shrunk) by adding `xlim` and/or `ylim` to the plot:

```
R> plot(Weight,MPG,xlim=c(2,6),ylim=c(0,40))  
R> abline(MPG.lm)  
R> lines(lowess(Weight,MPG),lty="dotted")
```



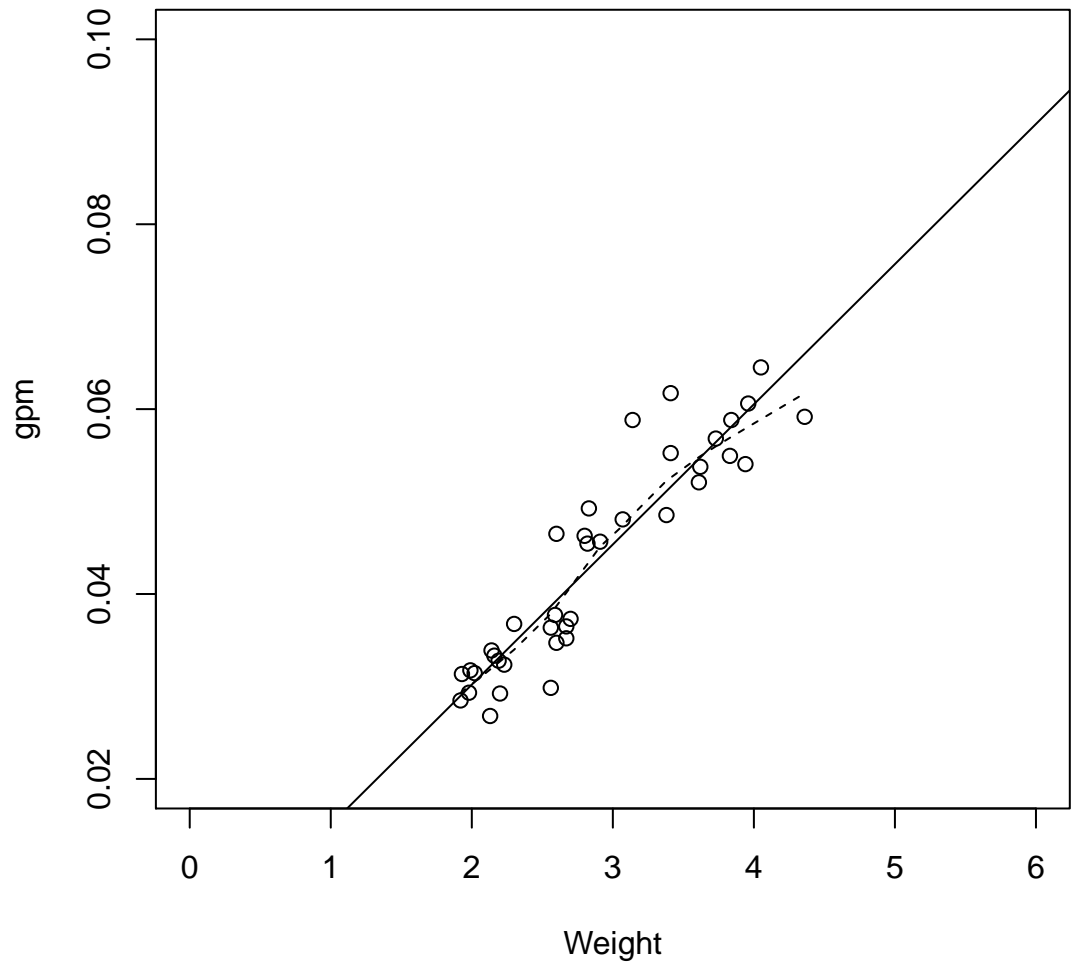
So what do *you* think should be a sensible prediction for MPG when `Weight` is 6 tons? The line keeps on going down, and at 6 tons goes below 0, which makes

no sense at all. The lowess curve stops at the last data point, which echoes the STAB22 response: “we should not do a prediction at all”. Maybe we should instead fit a curve that gradually levels off.¹⁶ Or maybe doing a transformation, predicting some *function* of MPG from `weight`. The lowess curve gives us the idea at least that we can improve upon the straight line.

One possible transformation is to predict the *reciprocal* of gas mileage, that is, gallons per mile:

```
R> gpm=1/MPG
R> plot(Weight,gpm,xlim=c(0,6),ylim=c(0.02,0.1))
R> gpm.lm=lm(gpm~Weight)
R> abline(gpm.lm)
R> lines(lowess(Weight,gpm),lty="dashed")
```

¹⁶Like a hyperbola, if you know about those.



I made the lowess curve dashed this time to make it easier to see. I also experimented a bit with the `gpm` scale to make it look better. A predicted `gpm` for a vehicle weighing 6 tons is still extrapolation, but the lowess curve looks nearly linear, so we can have at least a little confidence that the linear relationship will continue. The predicted `gpm` value for a weight of 6 tons is a bit more than 0.09, so the predicted miles per gallon is a bit less than $1/0.09 = 11$. That at least has the merit of not being obviously meaningless.

9.3.11 Labelling points

Which car is the heaviest? Which has the best gas mileage? Can we label on the plot which cars those are? The key is the `text` function. First, though, we have to find the rows of the data frame containing the heaviest and most fuel-efficient car. The code below will seem like black magic until we have studied selection, in Section 9.6. That's OK.¹⁷

```
R> cars[Weight==max(Weight),]
```

| | Car | MPG | Weight | Cylinders | Horsepower | Country |
|---|--------------------|------|--------|-----------|------------|---------|
| 9 | Buick Estate Wagon | 16.9 | 4.36 | 8 | 155 | U.S. |

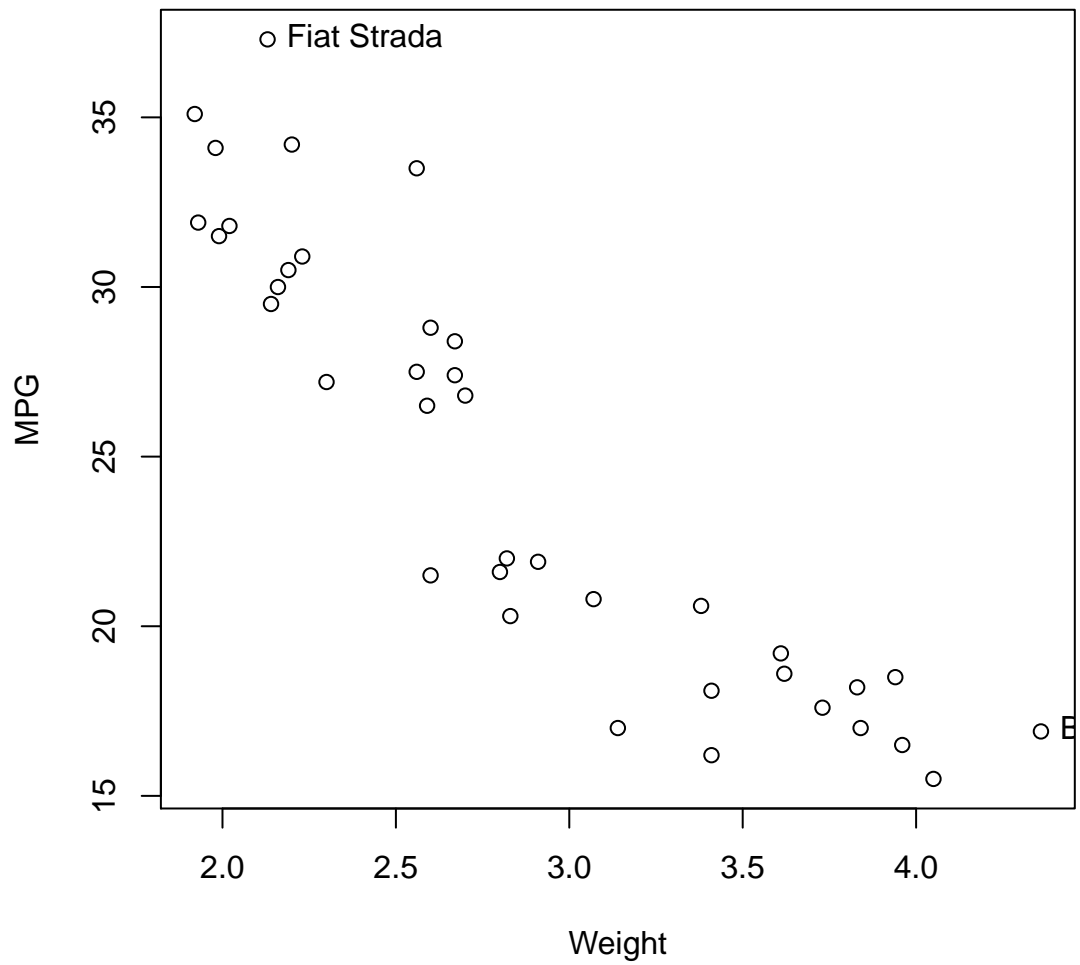
```
R> cars[MPG==max(MPG),]
```

| | Car | MPG | Weight | Cylinders | Horsepower | Country |
|---|-------------|------|--------|-----------|------------|---------|
| 4 | Fiat Strada | 37.3 | 2.13 | 4 | 69 | Italy |

The Buick Estate Wagon, car #9, is the heaviest, and the Fiat Strada, car #4, has the best gas mileage. Here's how we label them on the plot:

```
R> plot(Weight,MPG)
R> text(Weight[9],MPG[9],Car[9],pos=4)
R> text(Weight[4],MPG[4],Car[4],pos=4)
```

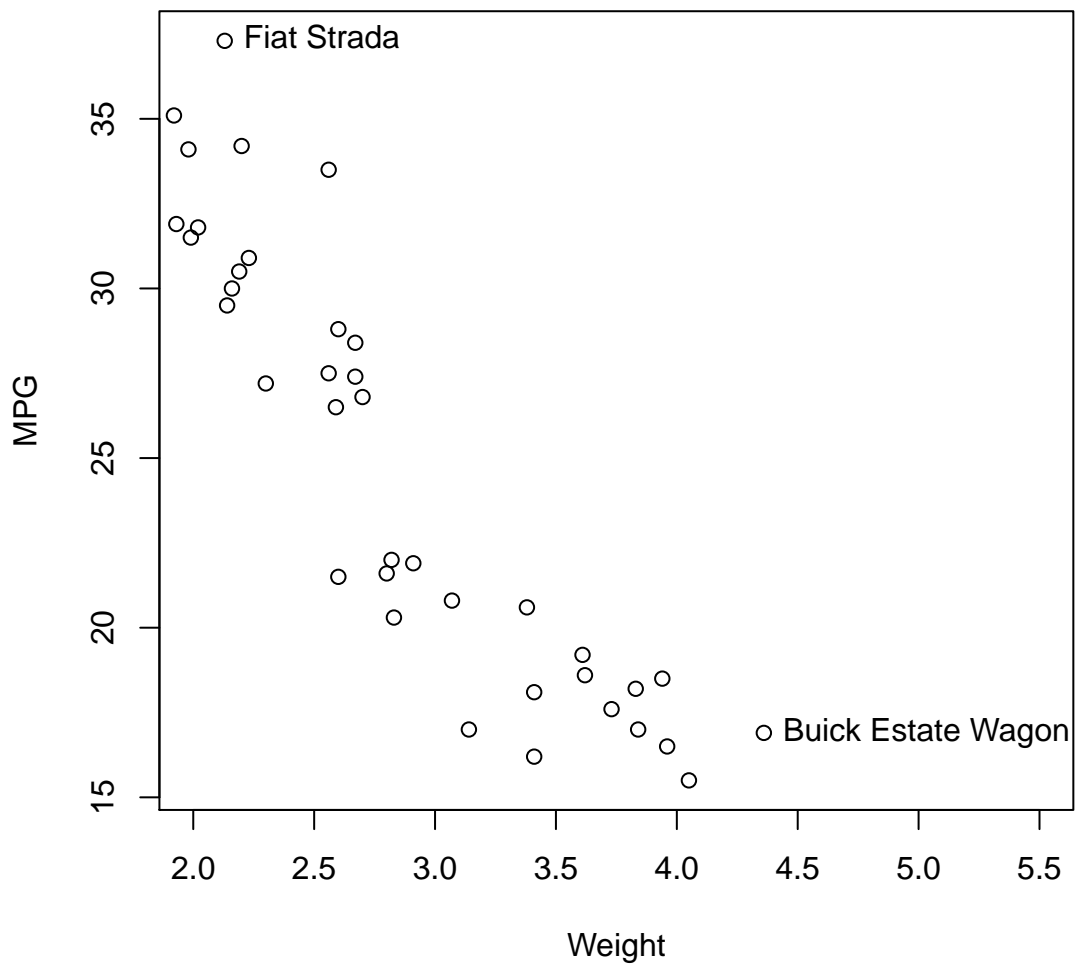
¹⁷For now.



`text` requires three things: the x and y coordinates of the place to put the text, and the text to put there (in this case, the name of the car). By default, the text gets plotted (centred) right at the coordinates you give. This is sometimes what you want. But here, we already have a circle there, so we want the text offset a bit so that it's next to the point, rather than on top of it. `pos` arranges that, working clockwise from 6 o'clock (`pos=1`), so that `pos=4` puts the text to the right.

Unfortunately, when R drew the plot, it had no idea that it needed to allow space for the Buick Estate Wagon. That can be fixed, though, by extending the `Weight` scale up to 5.5 or so:

```
R> plot(Weight,MPG,xlim=c(2,5.5))
R> text(Weight[9],MPG[9],Car[9],pos=4)
R> text(Weight[4],MPG[4],Car[4],pos=4)
```



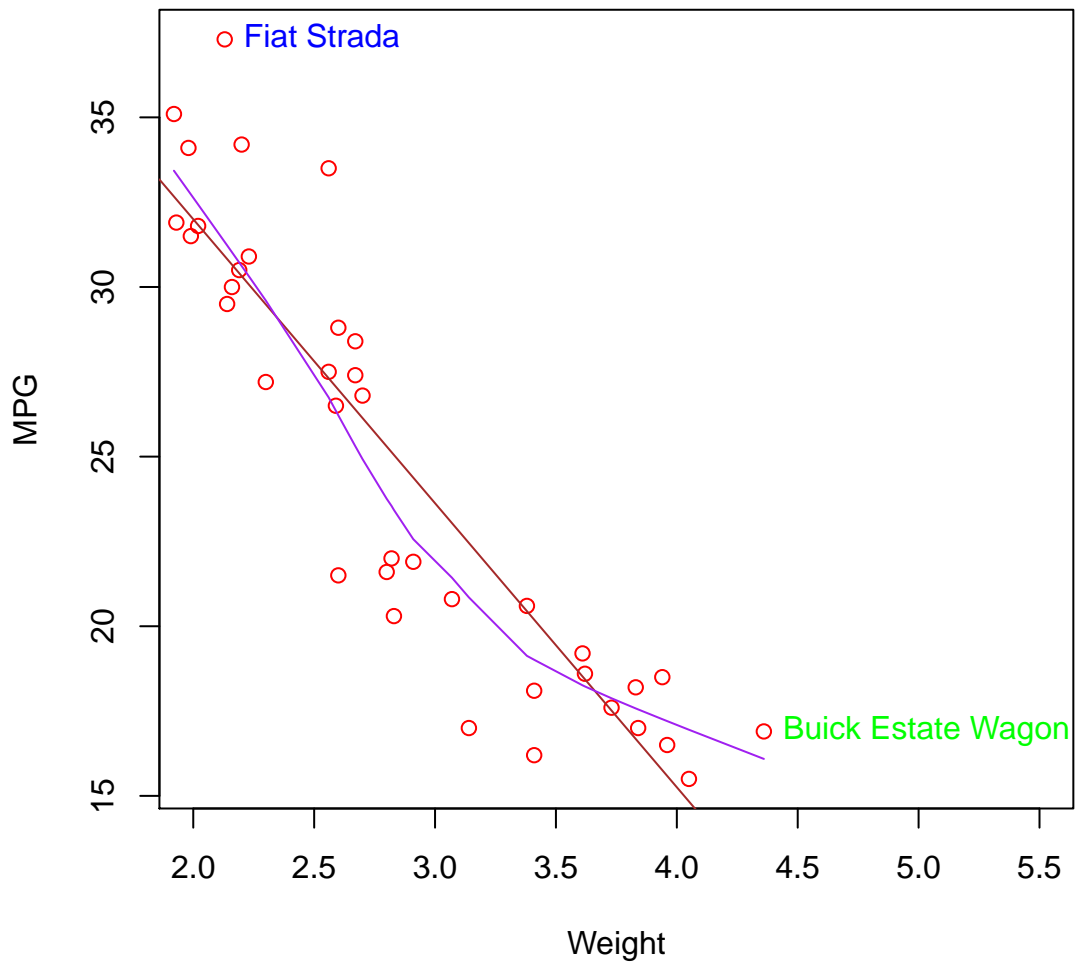
I got the value 5.5 by trial and error. I tried 5 first, but that wasn't wide enough. An alternative, which we'll see in a moment, is to make the font size of the text

smaller.

9.3.12 Colours

We already figured out how to make things different colours. You can use `col=` on pretty much anything. Here's the last plot again, but with colours on, as well as some extra lines for fun:

```
R> plot(Weight,MPG,xlim=c(2,5.5),col="red")
R> text(Weight[9],MPG[9],Car[9],pos=4,col="green")
R> text(Weight[4],MPG[4],Car[4],pos=4,col="blue")
R> abline(MPG.lm,col="brown")
R> lines(lowess(Weight,MPG),col="purple")
```



I make no claims about artistic value, though!

Colours can be given by name (as we have done). If you type `colors()`, you'll see a list of the 657 colour names that R understands. Some of these are hard to tell apart, though.¹⁸ Colours can also be referred to by numbers. The colour that a number refers to can be found by consulting the `palette`:

```
R> palette()
```

¹⁸I am reminded of <http://thedoghouseidiaries.com/1406>. See also <http://blog.xkcd.com/2010/05/03/color-survey-results/>, from where I found the preceding link.

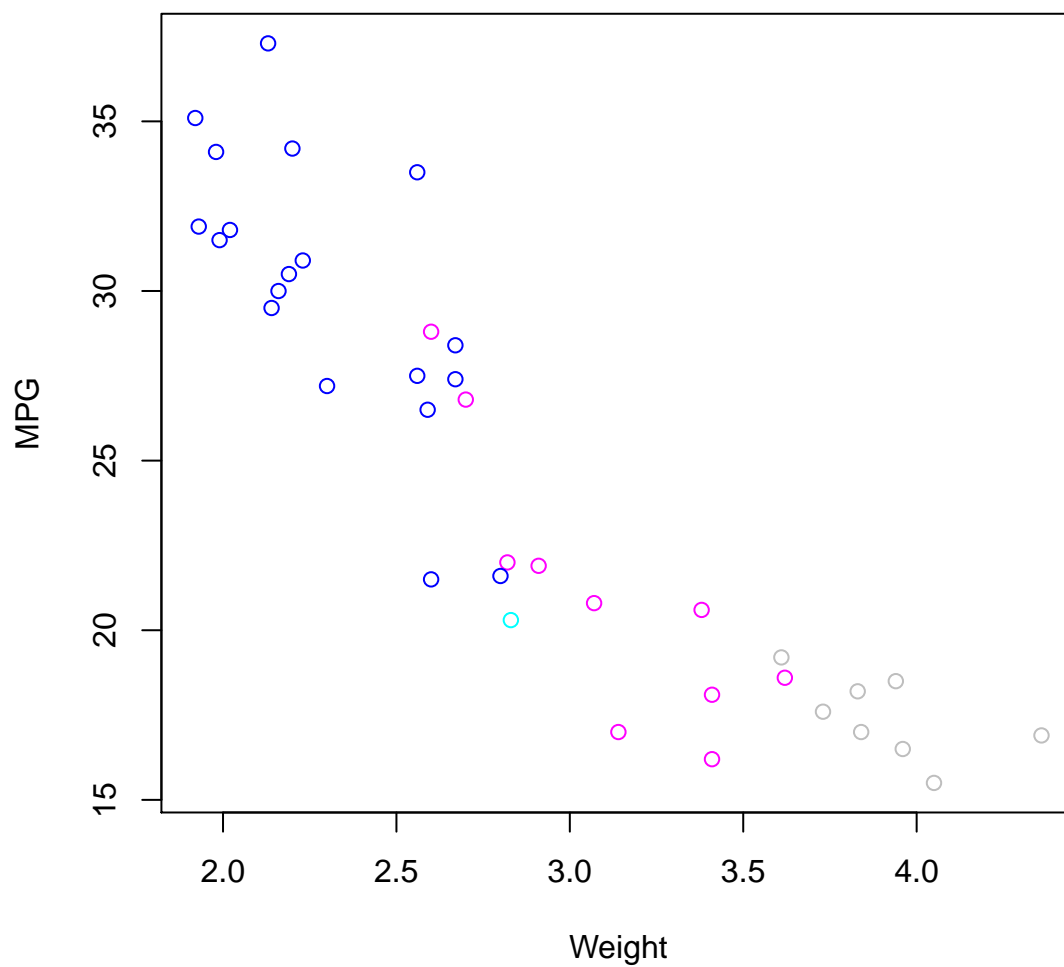
```
[1] "black"  "red"    "green3" "blue"   "cyan"   "magenta" "yellow"
[8] "gray"
```

After `gray`, the colours go around again starting at `black`.

The default way of plotting points, via `plot`, is to have them all the same colour.¹⁹ But how can we plot the cars a different colour according to what number of cylinders they have? `Cylinders` is already a number, so we feed *that* into `col`:

```
R> plot(Weight,MPG,col=Cylinders)
```

¹⁹Black, unless you specify `col=` in `plot`.

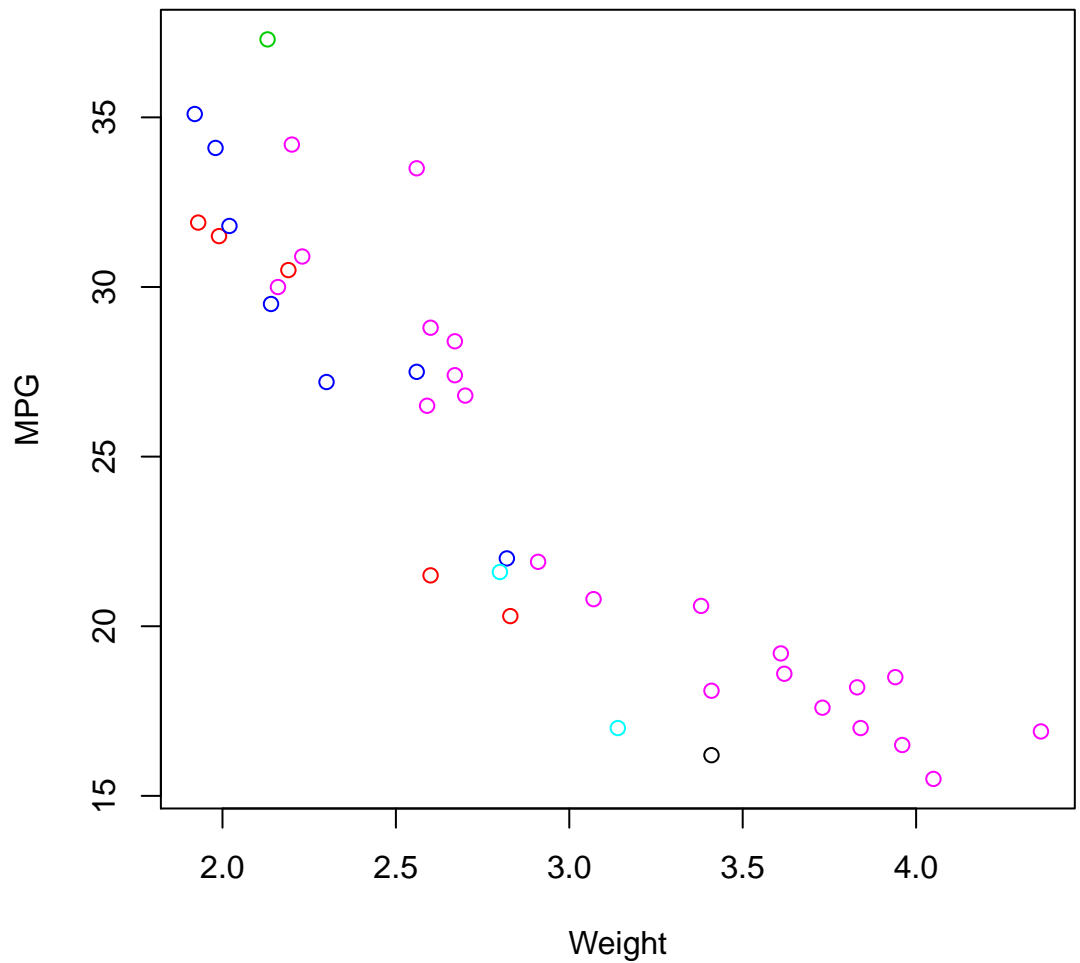


This uses colours 4, 5, 6 and 8 of the palette.²⁰

A more common problem is that your categorical variable that defines the colours is something, like `Country`, that is not a number.

```
R> plot(Weight, MPG, col=Country)
```

²⁰Which actually came out rather pleasingly.



R converted the countries into numbers, and then into colours. `Country`, in R terms, is a **factor**, whose categories are these, in order:

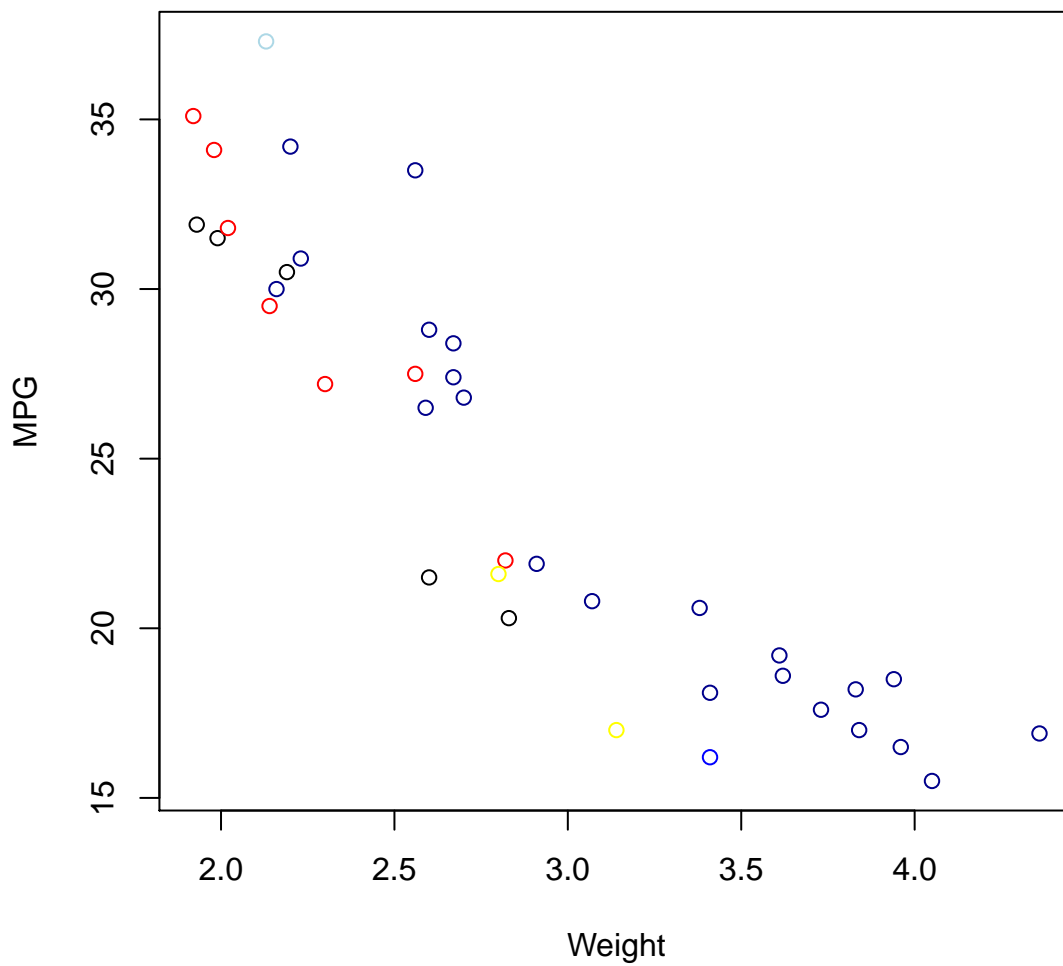
```
R> levels(Country)
```

```
[1] "France" "Germany" "Italy" "Japan" "Sweden" "U.S."
```

so that France is plotted in colour 1, Germany in colour 2, and so on. These colours are not very mnemonic. We can do better by making a list of colours in the same order as the countries. I'm picking a colour from the countries' soccer

team jerseys.²¹ In the `plot` command, I use the `[]` notation to pick out the right colour for the country by number. For example, Italy is 3, so I pick out the 3rd colour, light blue.

```
R> colours=c("blue","black","lightblue","red","yellow","darkblue")
R> plot(Weight,MPG,col=colours[Country])
```



²¹I had to cheat for Germany.

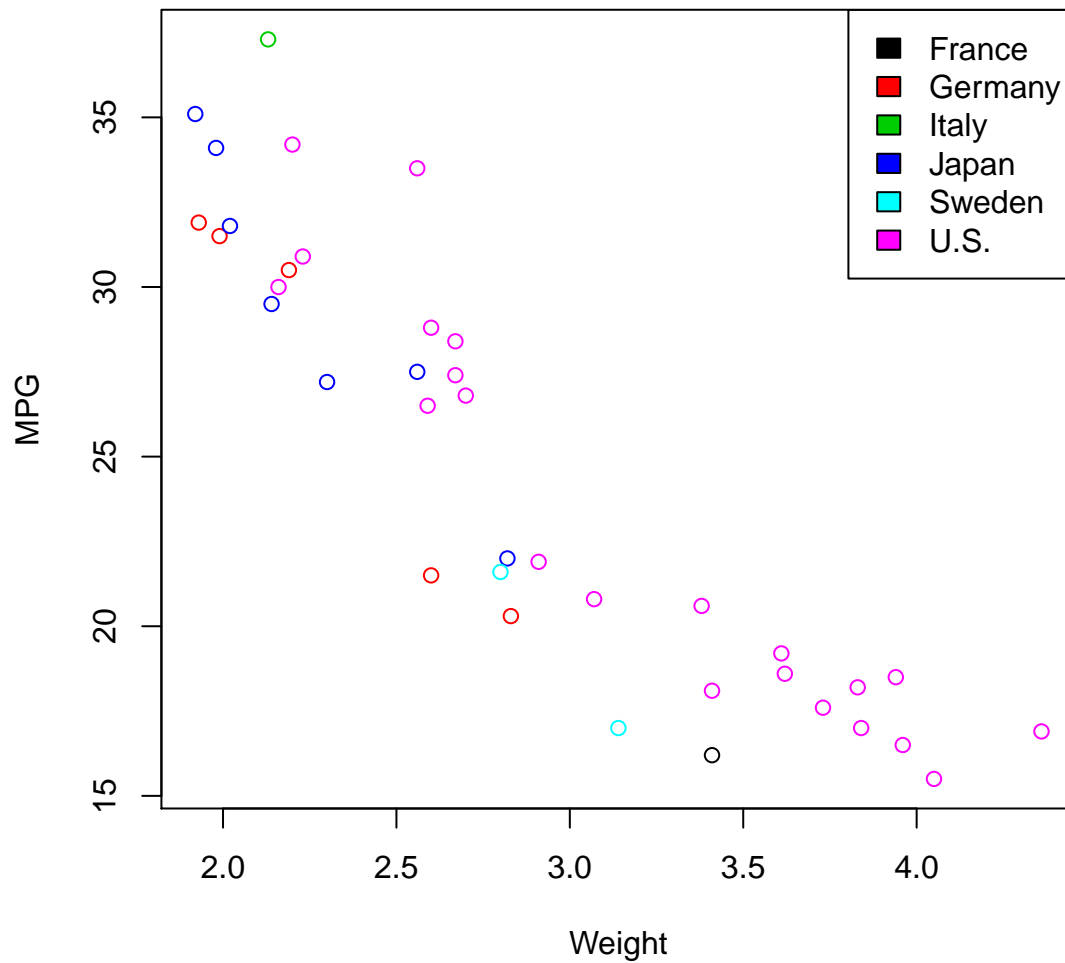
Not bad. The blues are a bit confusing. The darkest blue is the US; the mid-blue is France, and the almost-invisible light blue is Italy. And can you tell the blacks (Germany) from the dark blues? On the previous plot, the colours were easier to tell apart; we just couldn't tell which countries they referred to!

9.3.13 Legends

Let's go back to our previous plot and add a legend. The difficult part of this is to make sure that each country appears only once in the legend. I did this by creating a list of countries via `levels(Country)`. The `legend` function needs three things fed into it. The first thing is where to place the legend; there are handy names for the corners of the plot and the middles of the sides, `bottom`, `left`, `top` and `right`, which can be combined. You can also give x and y coordinates, via `x=` and `y=`; these tell where to put the *top-left* corner of the legend box. Next comes a categorical variable containing the names of the things to appear in the legend.²² Finally, we need some kind of indication of how the categories are distinguished. Here it's by colour, so I use `fill=` with the colours referred to by number.

```
R> plot(Weight,MPG,col=Country)
R> countries=levels(Country)
R> legend("topright",legend=countries,fill=1:6)
```

²²If you use `Country` for this, you get a list of 38 countries, with repeats, one for each car. You need a categorical variable that contains the names of the categories *once each*.



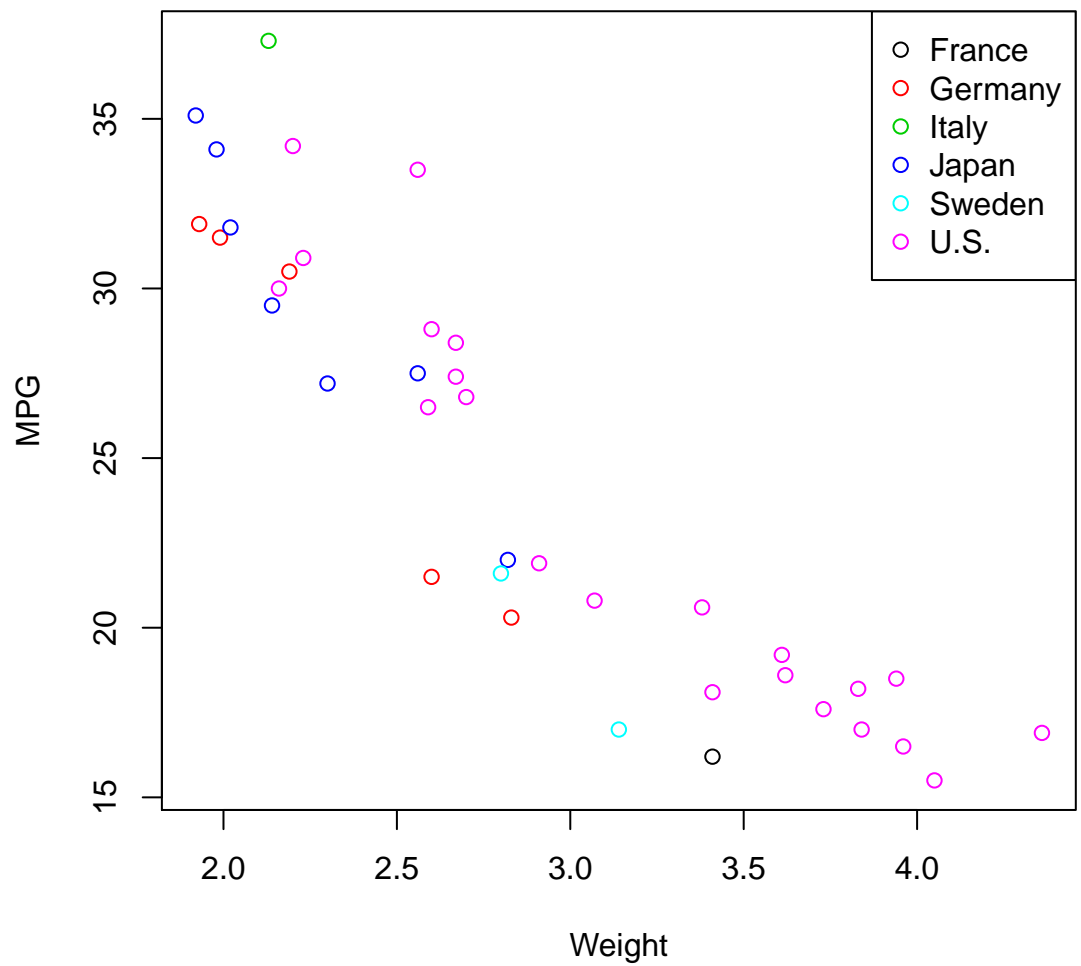
You can see huge number of purple (actually magenta) cars from the US, the one Italian car in green at top left, the one French car at the bottom, and a gaggle of German and Japanese cars mainly top left.

9.3.14 Plotting symbols

Since the points above were plotted by circles of different colours, I could have the legend contain those, like this. I changed `fill` to `col`, and I also had to

add `pch=1`:²³

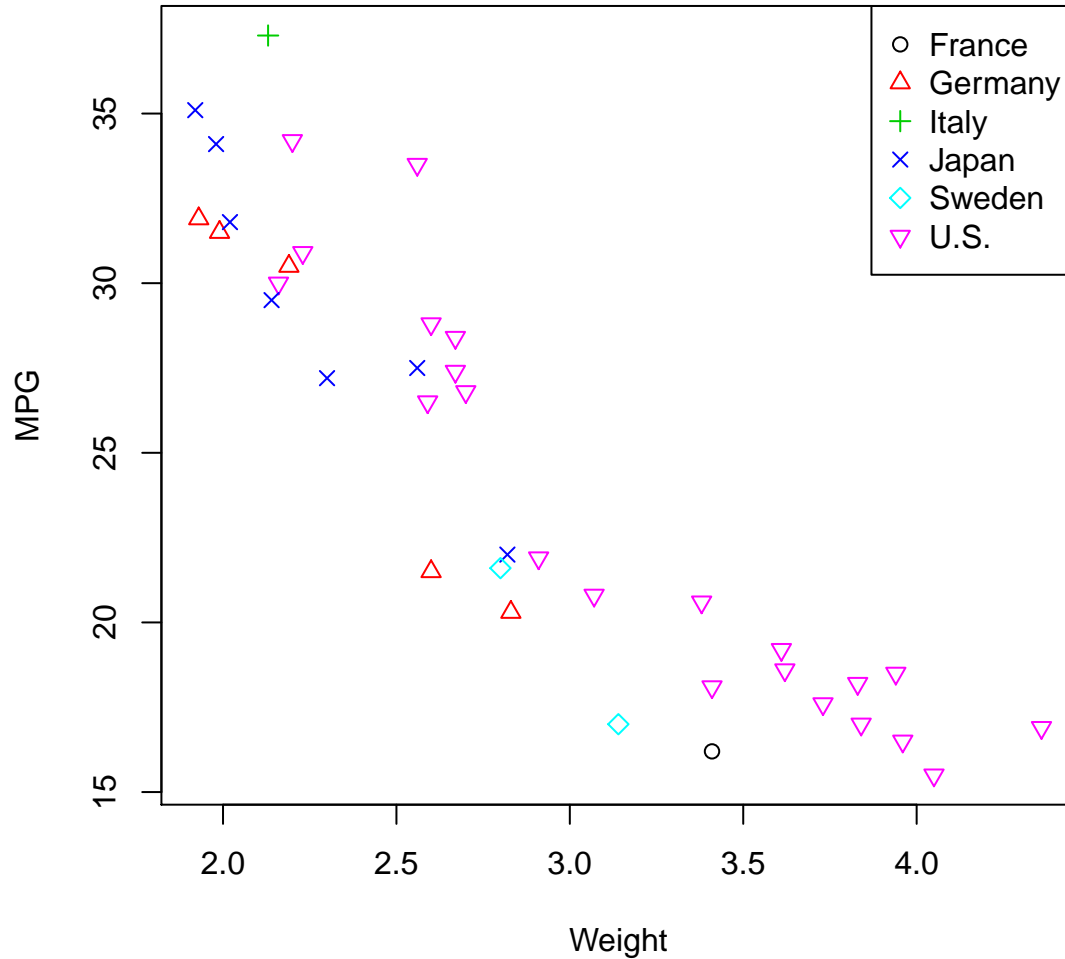
```
R> plot(Weight,MPG,col=Country)
R> countries=levels(Country)
R> legend("topright",legend=countries,pch=1,col=1:6)
```



²³`pch` says what character you plot to represent a point. The first “plotting character” is number 1, a circle.

The plotting characters are listed in the help for `points`. There are about 25 of them. You specify a particular one of them by number. For example, suppose we reproduce the above plot, but distinguishing countries by colour *and* plotting character:

```
R> plot(Weight,MPG,col=Country,pch=as.numeric(Country))
R> countries=levels(Country)
R> legend("topright",legend=countries,pch=1:6,col=1:6)
```

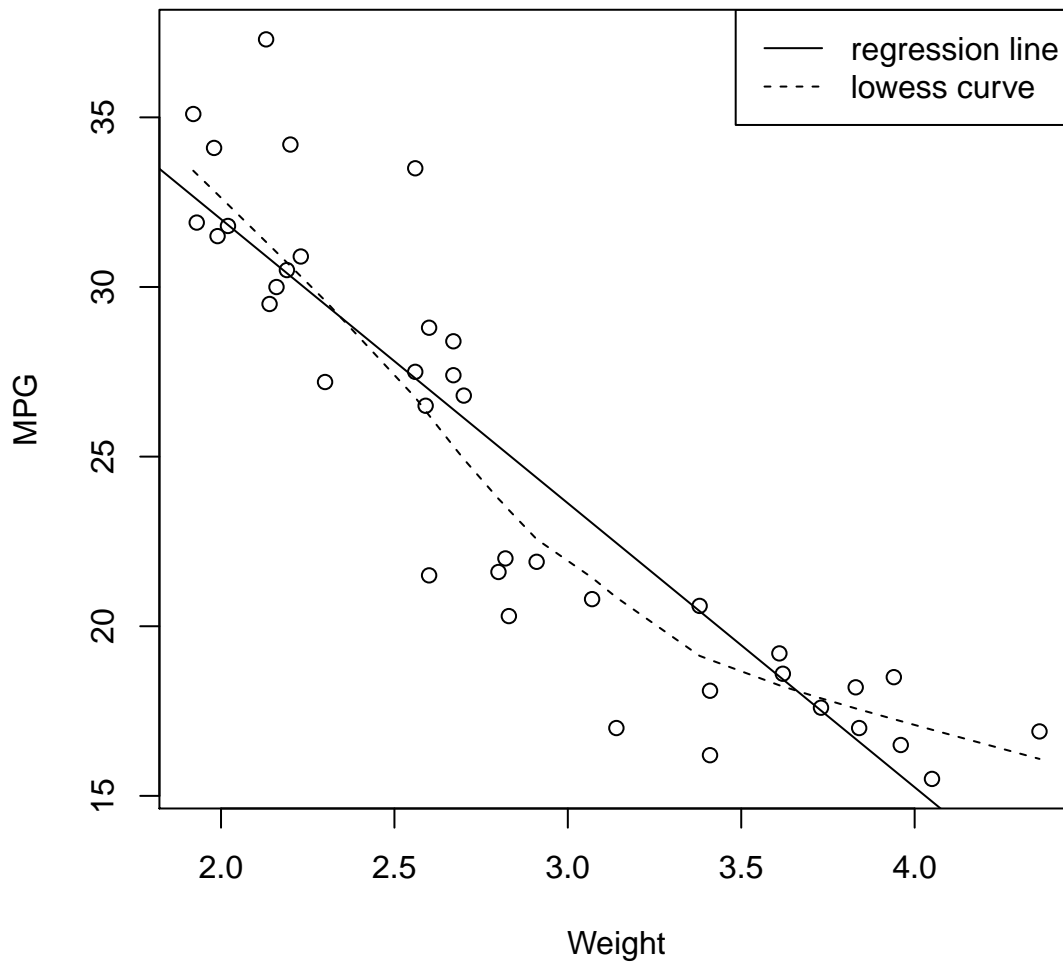


I think this makes the countries easier to distinguish. There's one quirk here: while `col` seems quite happy to turn a categorical factor into a number, `pch` requires an actual number. The way to turn a factor into a number is `as.numeric`. For the legend, I just want my six countries; plotting character 1 (circle) goes with colour 1 (black), and so on, up to plotting character 6 (triangle upside down) going with colour 6 (magenta).

9.3.15 More about legends

In the same kind of way, a legend can feature line types as well. Let's plot our data with the regression line and lowess curve, using different line types, labelled appropriately:

```
R> plot(Weight,MPG)
R> curve.types=c("regression line","lowess curve")
R> line.types=c("solid","dashed")
R> abline(MPG.lm,lty=line.types[1])
R> lines(lowess(Weight,MPG),lty=line.types[2])
R> legend("topright",legend=curve.types,lty=line.types)
```

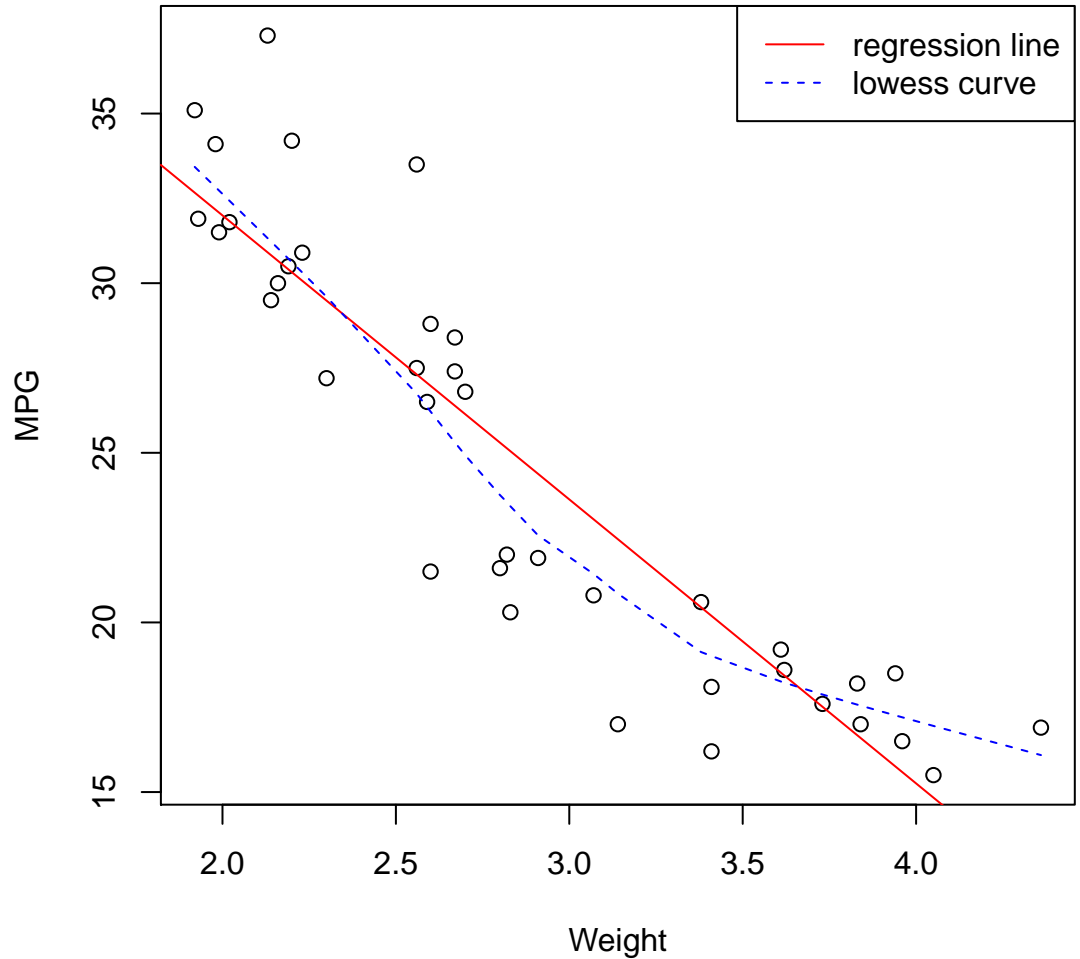



I defined some extra variables, mainly to help with the legend. I made a list of the curve types I was going to draw, and I chose line types to go with them. Then I drew the regression line, explicitly using the first line type (`solid`).²⁴ Next I drew the lowess curve, using the second line type. Last, I made the legend, using the `curve.types` variable to list the types of curve I drew and identifying them with the right one of `line.types`.

There is no difficulty in adding colours to these:

²⁴Though I didn't need to do this, since it is the default.

```
R> plot(Weight,MPG)
R> curve.types=c("regression line","lowess curve")
R> line.types=c("solid","dashed")
R> colours=c("red","blue")
R> abline(MPG.lm,lty=line.types[1],col=colours[1])
R> lines(lowess(Weight,MPG),lty=line.types[2],col=colours[2])
R> legend("topright",legend=curve.types,lty=line.types,col=colours)
```



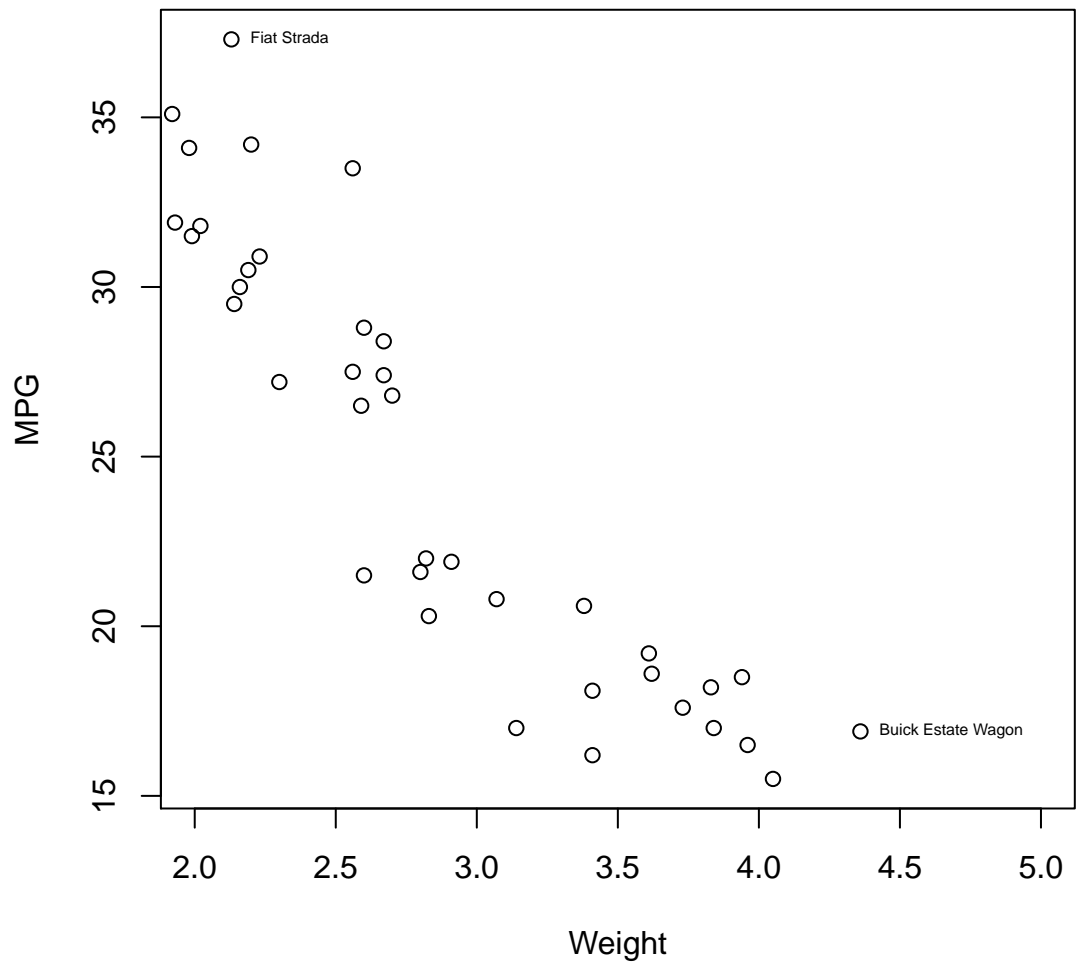
I did this by defining my `colours`, and then using them via `col=` all over the place.

9.3.16 Changing text size

Sometimes text is too big. Or too small. The basic way to fix this is via `cex`, which is a multiplier to change the size of text on the plot: `cex=2` makes the text twice as big as normal, `cex=0.5` makes it half as big.

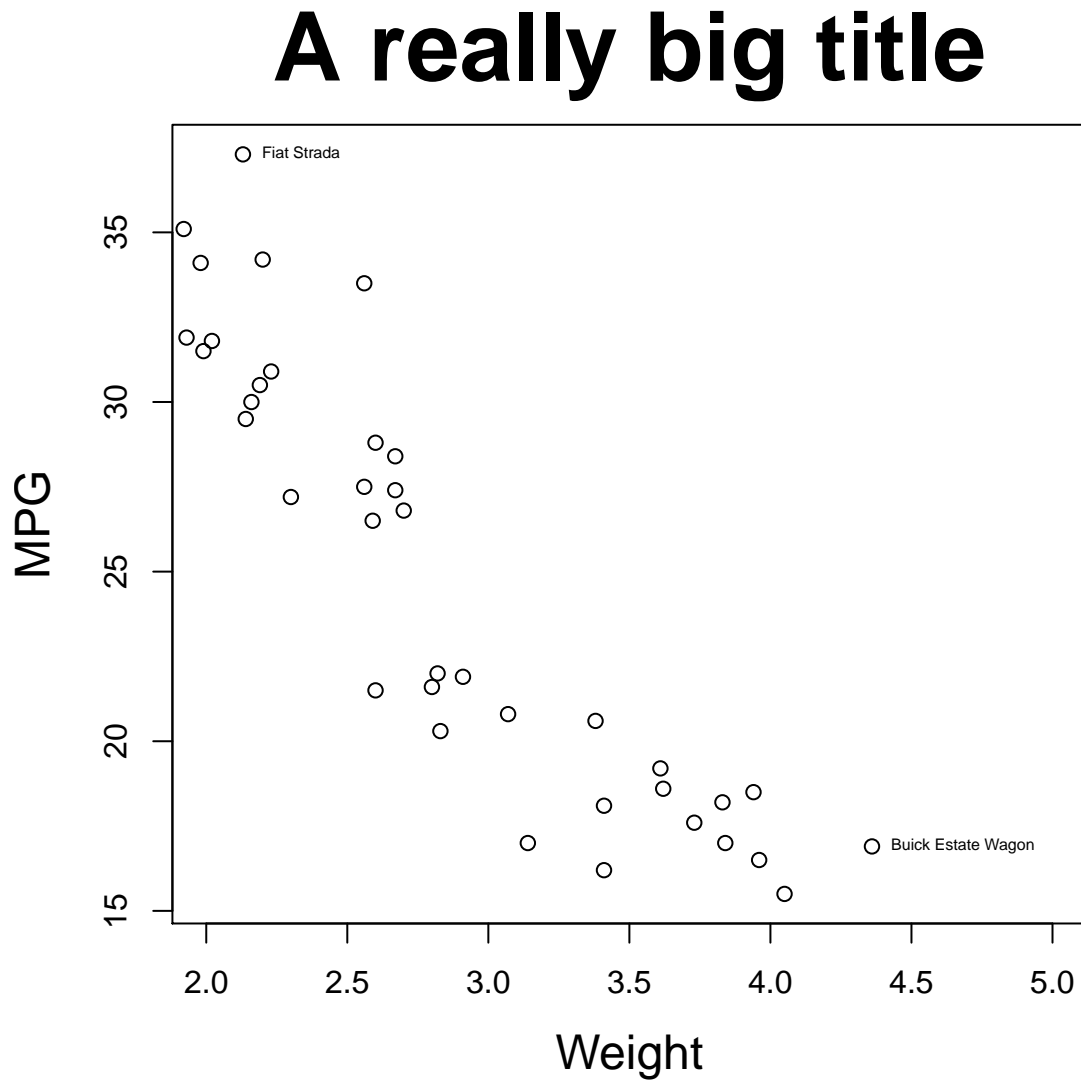
Recall the plot before where we wanted to label the Fiat Strada (best gas mileage) and Buick Estate Wagon (heaviest)? Let's make the labels half normal size:

```
R> plot(Weight,MPG,xlim=c(2,5))
R> text(Weight[9],MPG[9],Car[9],pos=4,cex=0.5)
R> text(Weight[4],MPG[4],Car[4],pos=4,cex=0.5)
```



There are other variations on `cex` that affect text on axes, text on axis labels, text in main titles, and text in subtitles. Let's play at taking the previous plot and making the axis labels and the main title *really big*. These go on the original plot command, since that sets up the main title and axes:

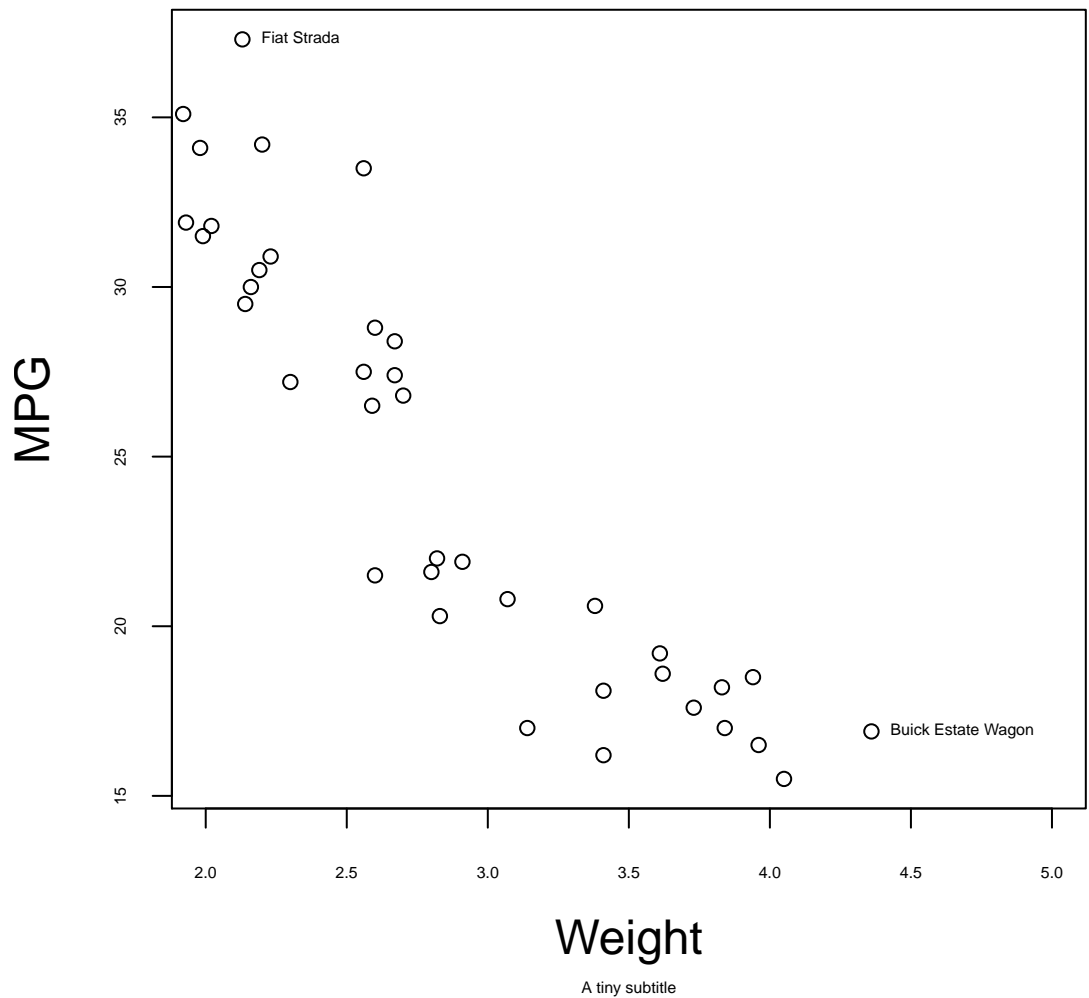
```
R> plot(Weight,MPG,xlim=c(2,5),cex.lab=1.5,cex.main=3,main="A really big title")
R> text(Weight[9],MPG[9],Car[9],pos=4,cex=0.5)
R> text(Weight[4],MPG[4],Car[4],pos=4,cex=0.5)
```



and this time we'll make the stuff on the axes really small, and add a tiny subtitle:

```
R> plot(Weight,MPG,xlim=c(2,5),cex.lab=1.5,cex.main=3,
R>   main="A really big title",cex.axis=0.5,cex.sub=0.5,sub="A tiny subtitle")
R> text(Weight[9],MPG[9],Car[9],pos=4,cex=0.5)
R> text(Weight[4],MPG[4],Car[4],pos=4,cex=0.5)
```

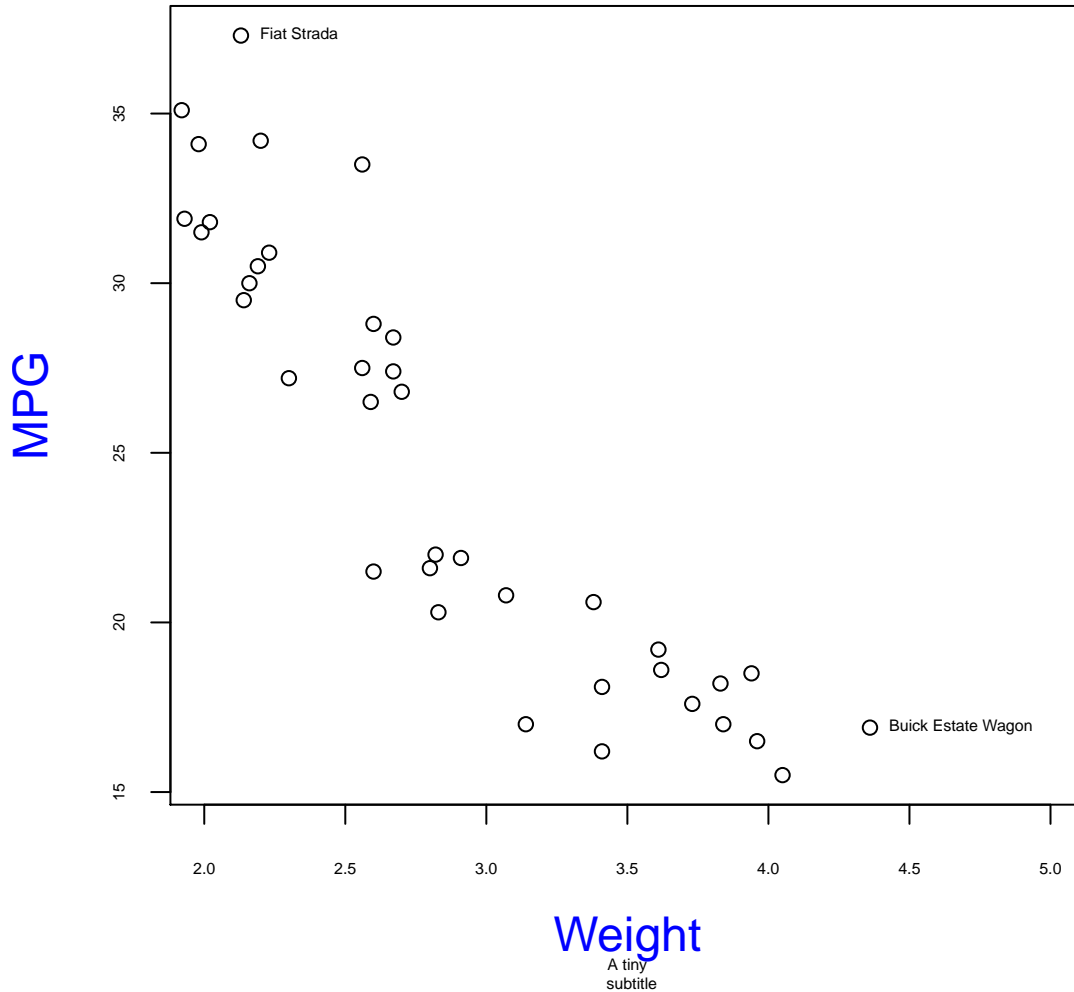
A really big title



of it:

```
R> plot(Weight,MPG,xlim=c(2,5),cex.lab=1.5,cex.main=3,  
R>   main="A really big title",cex.axis=0.5,cex.sub=0.5,sub="A tiny  
R>   subtitle",col.main="green",col.lab="blue")  
R> text(Weight[9],MPG[9],Car[9],pos=4,cex=0.5)  
R> text(Weight[4],MPG[4],Car[4],pos=4,cex=0.5)
```

A really big title

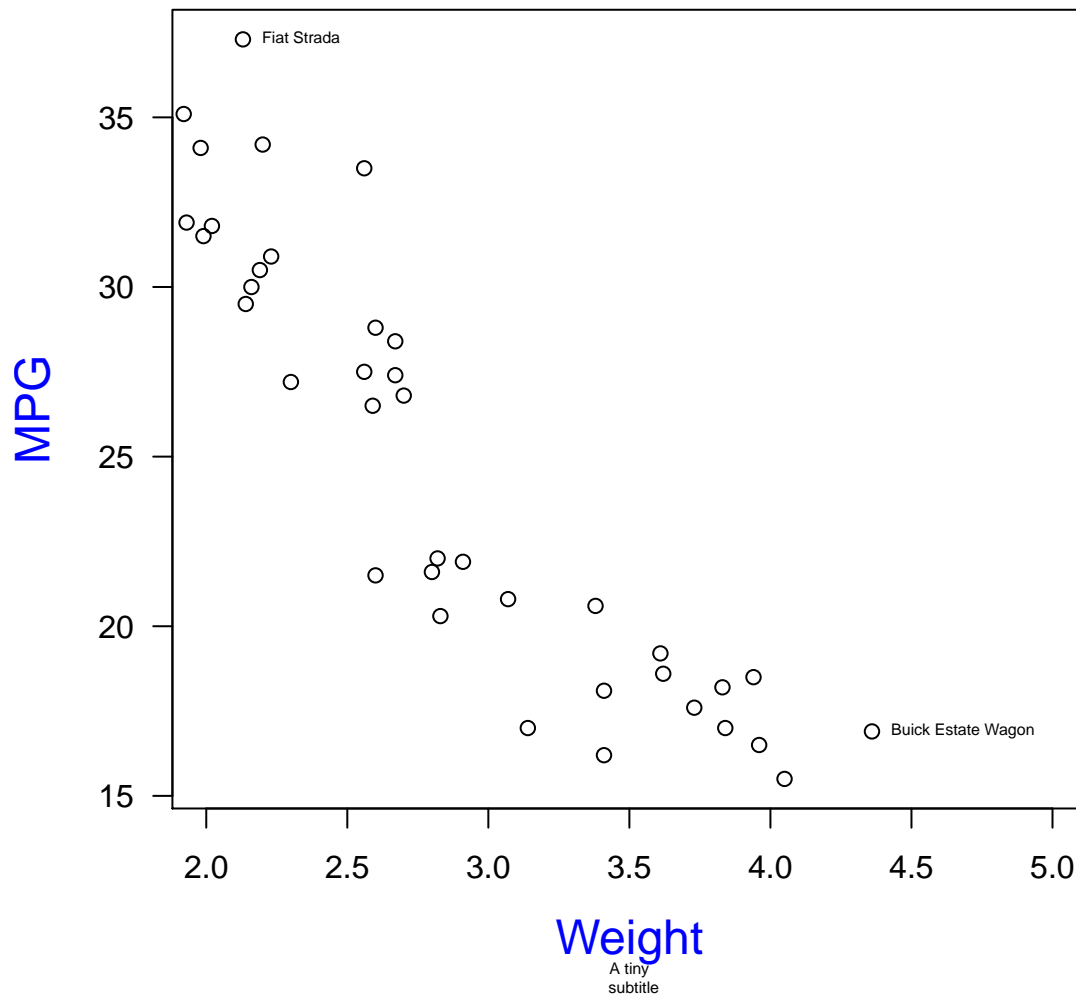


9.3.18 Appearance of numbers on axes

I wanted to mention one other command that I sometimes use. R's default is to put the numbers on the axes *perpendicular to the axes*, so the numbers on the x axis are horizontal and the numbers on the y axis are sideways. Sometimes you would like to change this. The commonest change is that the numbers on the y axis should be horizontal too. This is accomplished by `las`. The help for `par` tells you how it works, but `las=1` makes all the axis labels horizontal. Let's do that to our plot, taking out the `cex.axis` so that you can actually see the results:

```
R> plot(Weight,MPG,xlim=c(2,5),cex.lab=1.5,cex.main=3,  
R>   main="A really big title",las=1,cex.sub=0.5,sub="A tiny  
R>   subtitle",col.main="green",col.lab="blue")  
R> text(Weight[9],MPG[9],Car[9],pos=4,cex=0.5)  
R> text(Weight[4],MPG[4],Car[4],pos=4,cex=0.5)
```


A really big title



See now the MPG numbers on the y -axis are now horizontal?

9.4 Other types of plots

9.4.1 Time series plots

The usual way of using `plot` is to draw scatterplots, in which case plotting just the points, which is the default, is what you want. Sometimes, though, you want to join the points by lines. Often, this is because you have data in time order, and you want to see if there's a trend. I got a time series of annual household electricity usage in Britain, which I saved in a data frame called `uk.energy`.

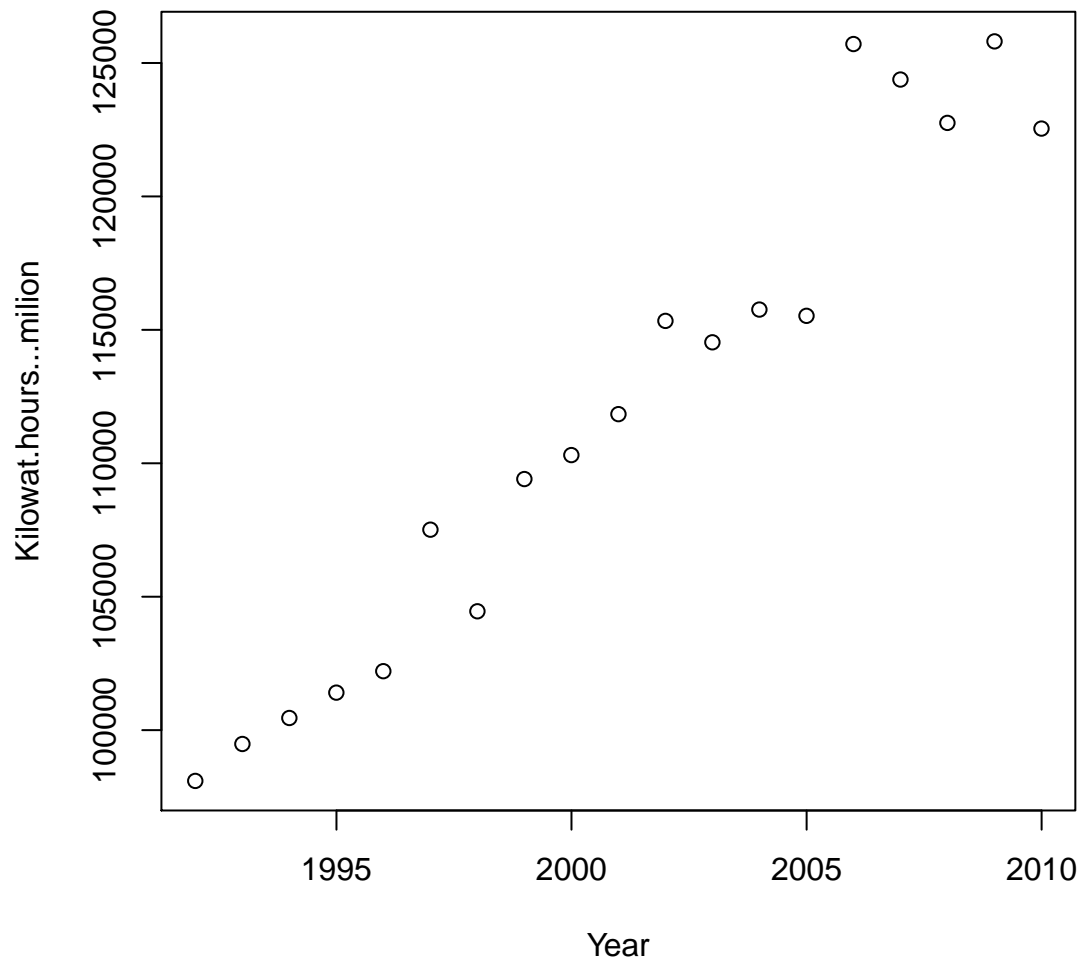
```
R> uk.energy=read.table('uk-energy.txt',header=T,
R>   colClasses=c('Year'='Date'))
```

This data frame has two variables, the first of which is the year. The reason for the `colclasses` thing was that I wanted `Year` to be treated as a date, and not a text string.

If you just plot the data frame, the two variables will be plotted against each other.

```
R> uk.energy
R> plot(uk.energy)
```

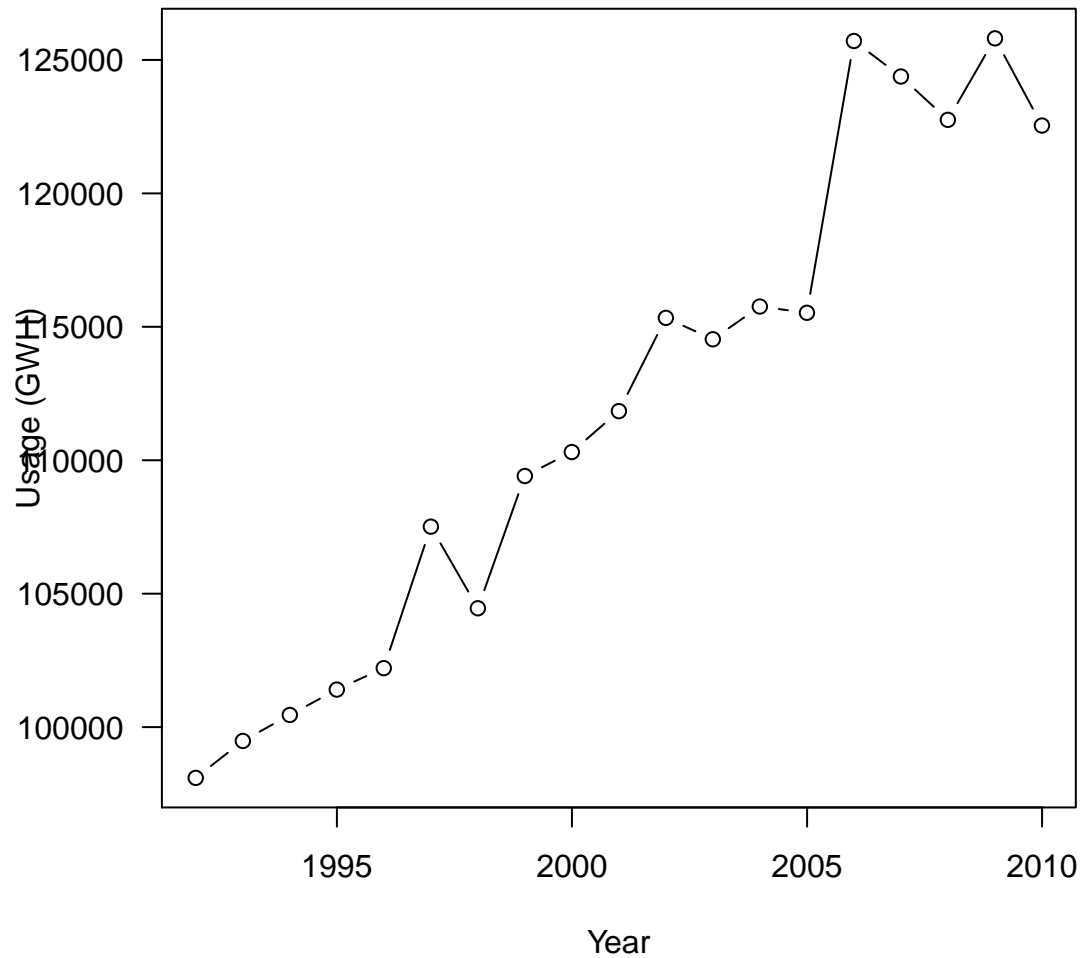
| | Year | Kilowat.hours...million |
|----|------------|-------------------------|
| 1 | 2009-12-31 | 122543 |
| 2 | 2008-12-31 | 125811 |
| 3 | 2007-12-31 | 122756 |
| 4 | 2006-12-31 | 124381 |
| 5 | 2005-12-31 | 125711 |
| 6 | 2004-12-31 | 115526 |
| 7 | 2003-12-31 | 115761 |
| 8 | 2002-12-31 | 114534 |
| 9 | 2001-12-31 | 115337 |
| 10 | 2000-12-31 | 111842 |
| 11 | 1999-12-31 | 110308 |
| 12 | 1998-12-31 | 109410 |
| 13 | 1997-12-31 | 104455 |
| 14 | 1996-12-31 | 107513 |
| 15 | 1995-12-31 | 102210 |
| 16 | 1994-12-31 | 101407 |
| 17 | 1993-12-31 | 100456 |
| 18 | 1992-12-31 | 99482 |
| 19 | 1991-12-31 | 98098 |



We can improve this plot in a couple of ways. First, we can join the points by lines. The `type=` argument to `plot` controls this. The default is `type="p"`, “points only”. You can also have `type="l"`, “lines only”, or `type="b"`, “both points and lines”. Second, we can change the *y*-axis label to something more meaningful via `ylab`. Third, those big numbers on the *y*-axis would be better horizontal; as they are, they are taking up too much space. This is `las`. Fourth, let’s add a big red title:

```
R> plot(uk.energy, type="b", ylab="Usage (GWH)", las=1, cex.main=2, col.main="red",  
R>      main="UK Household Electricity Usage")
```

UK Household Electricity Usage



This isn't quite right, because the horizontal GWH figures overwrite the axis label. Maybe we should divide the numbers by 1000 and call them TWH instead.²⁵ But anyway, the plot clearly shows the increasing trend. There was a big jump from 2005 to 2006, but things seem to have levelled off since then. Is that meaningful? Who knows.

²⁵Terawatt hours, like terabyte hard disks.

9.4.2 Plotting multiple series on one plot

The next thing illustrates what you might think of as plotting multiple “series” in a spreadsheet.

Imagine that you have a sample of five orange trees, and you measure their circumference at various times as they grow. Here are the times:

```
R> ages
```

```
[1] 118 484 664 1004 1231 1372 1582
```

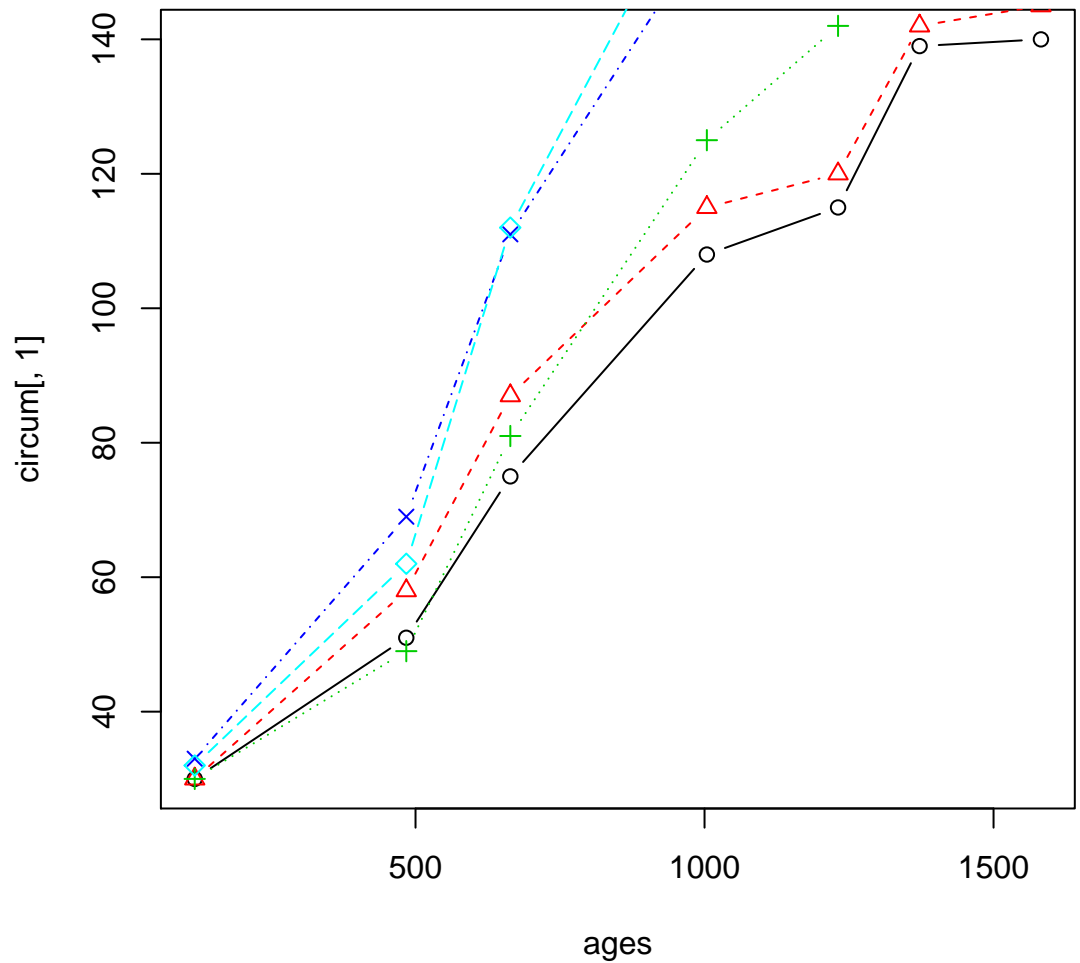
and here are the circumferences at those times for each of the trees, labelled A through E:

```
R> circum
```

| | A | B | C | D | E |
|---|-----|-----|-----|-----|-----|
| 1 | 30 | 30 | 30 | 33 | 32 |
| 2 | 51 | 58 | 49 | 69 | 62 |
| 3 | 75 | 87 | 81 | 111 | 112 |
| 4 | 108 | 115 | 125 | 156 | 167 |
| 5 | 115 | 120 | 142 | 172 | 179 |
| 6 | 139 | 142 | 174 | 203 | 209 |
| 7 | 140 | 145 | 177 | 203 | 214 |

We want to make a plot of the growth of those trees over time. We can’t just do this in one `plot`, since we have five things to plot. A handy trick is to use the `plot` command to plot *nothing* (`type="n"`), and then use `lines` or `points` to add things to the plot one at a time. The original `plot` needs something to not-plot, though, because it uses this to set up the axes. Here’s our first go, using a loop to go through the columns of `circum` one at a time. The notation `x[,i]` means “the *i*-th column of *x*”:

```
R> plot(ages, circum[,1], type="n")
R> for (i in 1:5)
R> {
R>   lines(ages, circum[,i], type="b", col=i, lty=i, pch=i)
R> }
```



In case you're wondering, I could have used either `lines` or `points`, since I was actually plotting both lines and points.

I used different line types, colours and plotting characters for each tree. I took advantage of the fact that the numbers 1 through 5 can refer to colours from the palette, line types *and* plotting characters. You might not want to go to this length. I did it because I could.²⁶

²⁶“Why do dogs lick their balls?” “Because they can.”

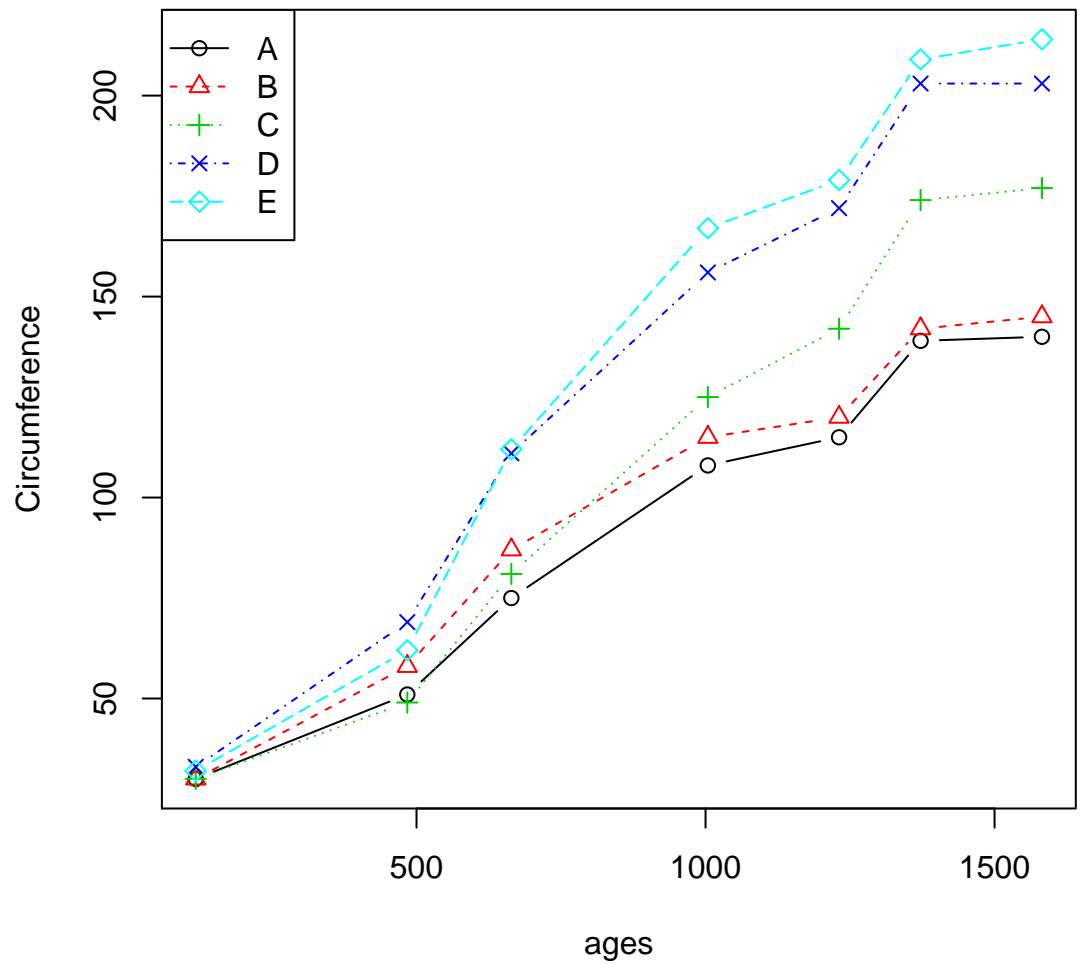
There are a couple of problems with this plot. The simple one to fix is the *y*-axis label. The more pressing problem is that some of the curves go off the top of the plot! The reason for that is that we used the first tree to set the vertical scale, and some of the other trees grow bigger than that. One way to fix this is to go back and look at the circumference data frame `circum`, pick out the smallest and largest values (30 and 214) and set those as our `ylim`. R makes that easier still:

```
R> circum.range=range(circum)
R> circum.range
```

```
[1] 30 214
```

We also need a legend. Here's how all of that looks:

```
R> plot(ages, circum[,1], type="n", ylim=circum.range, ylab="Circumference")
R> for (i in 1:5)
R> {
R>   lines(ages, circum[,i], type="b", col=i, lty=i, pch=i)
R> }
R> legend("topleft", legend=colnames(circum), col=1:5, lty=1:5, pch=1:5)
```



I think that looks good. Note how the legend shows the plotting symbols, line types *and* colours.

The trees exhibited a steady growth until the last time period, when they all stopped growing any more.

9.4.3 Multiple plots on one page

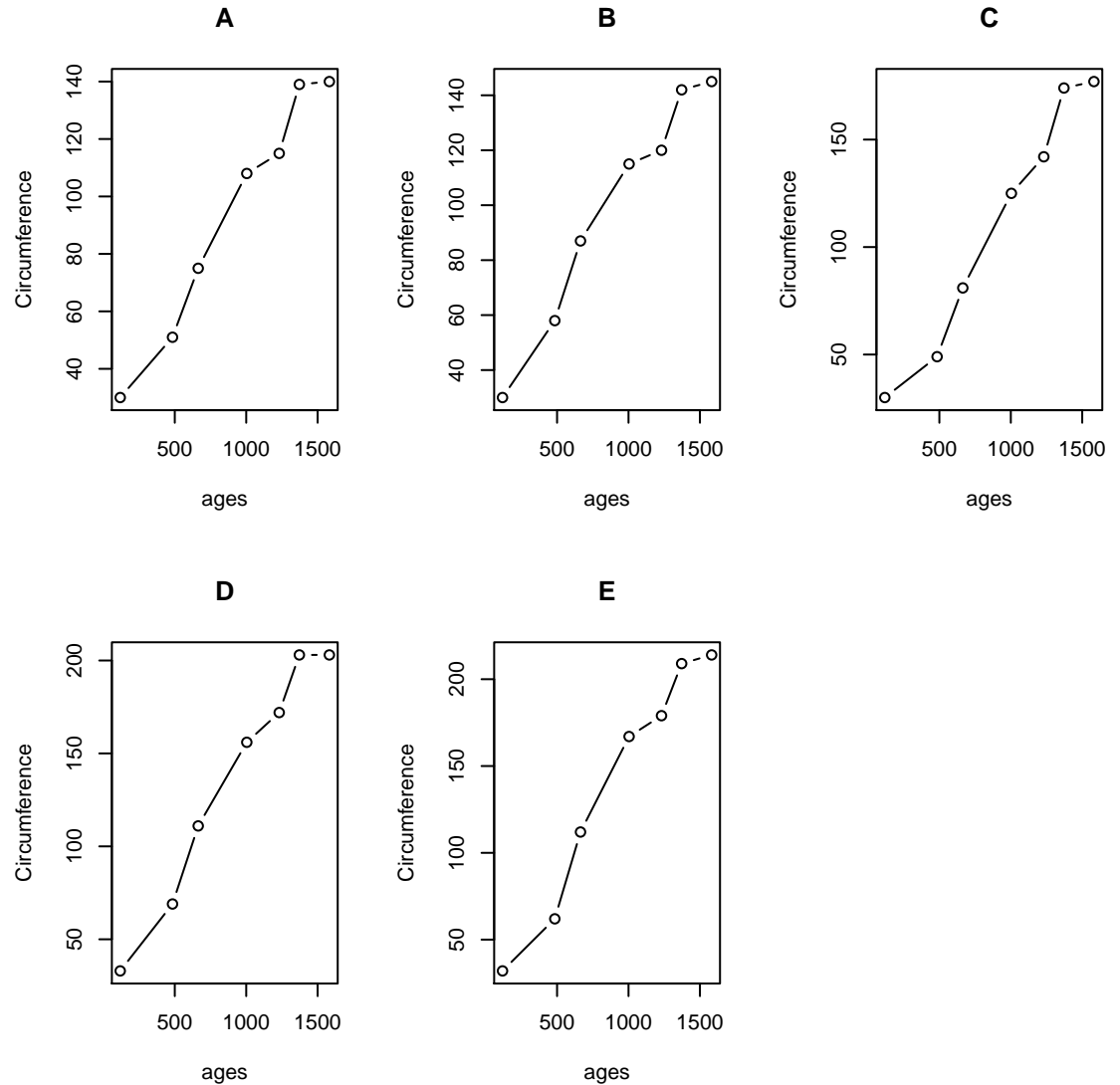
I want to show you one more thing about plots. Sometimes, you want to have multiple plots on one page. For example, instead of plotting those five tree growth curves on one plot, you might want to have them on separate plots all on one page.

The way to arrange that is via `mfrow`, which sorts out an array of plots on the page. The idea is that you set up `mfrow` first, and then each time you say `plot`, R moves on to the next plot in the array. It's actually amusing²⁷ to run the commands below one line at a time in R Studio, and you'll see the page filling up with plots.

Now, we have five growth curves to plot, and five isn't a very handy number. Let's make a plotting area with two rows and three columns, so that we'll have one of them blank at the end. The way to set `mfrow` is to put it inside a call to `par`, as shown below:

```
R> par(mfrow=c(2,3))
R> plot(ages, circum[,1], ylab="Circumference", main="A", type="b")
R> plot(ages, circum[,2], ylab="Circumference", main="B", type="b")
R> plot(ages, circum[,3], ylab="Circumference", main="C", type="b")
R> plot(ages, circum[,4], ylab="Circumference", main="D", type="b")
R> plot(ages, circum[,5], ylab="Circumference", main="E", type="b")
```

²⁷For me it is, anyway.

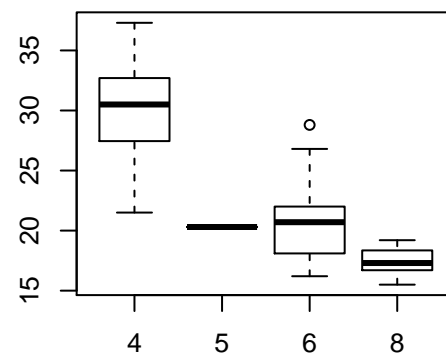
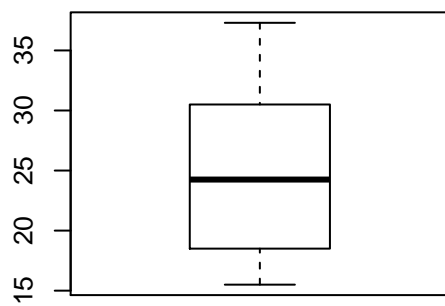
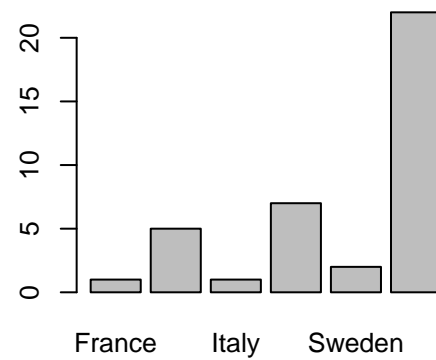
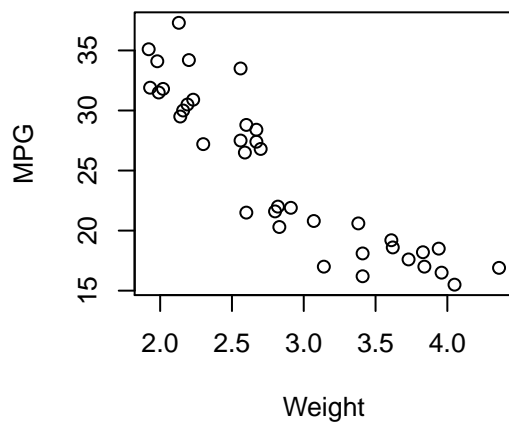


I could have done the plots in a `for (i in 1:5)` loop, but then I would have needed some trickery to get the titles right. In this case, I like the five curves on one plot better than the five individual plots, but in other circumstances it might be different.

When you make an array of plots with `mfrow`, each plot can contain anything; they don't have to be all the same kind of thing, as they were above. They don't even have to be all scatterplots. Anything that appears in the Plots window in R Studio counts as a plot.

A rather pointless example using the cars data:

```
R> par(mfrow=c(2,2))
R> plot(Weight,MPG)
R> plot(Country)
R> boxplot(MPG)
R> boxplot(MPG~Cylinders)
R> par(mfrow=c(1,1))
```



The last `mfrow` resets the Plots window to go back to one plot at a time. If you

don't do that, future plots will still be in the 2×2 array, the next plot blanking the array and taking the top left spot.

In the plots above, the last three plots could do with some axis labels, and the bar chart could do with having the country names vertical, so that they will all go in. These fixes are left as an exercise for the reader.²⁸

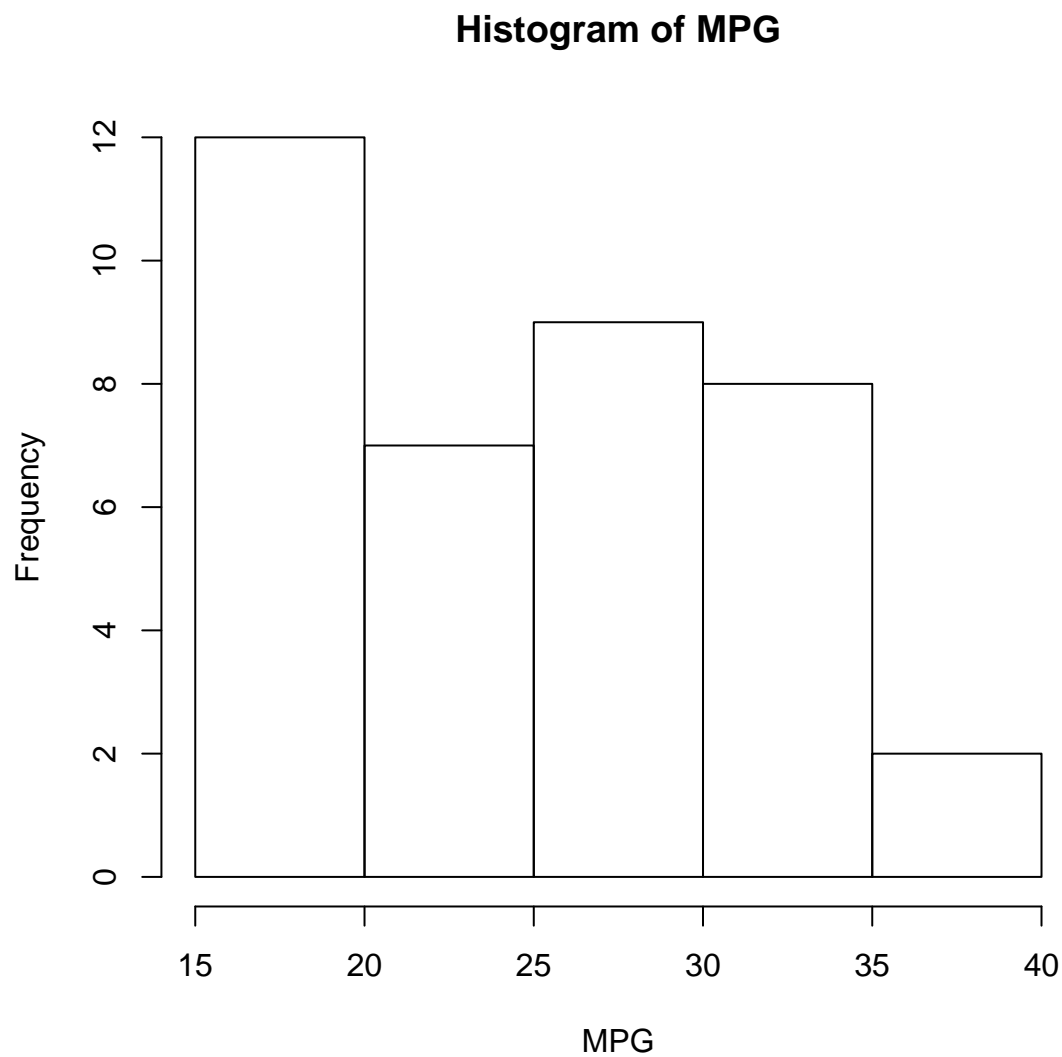
9.5 Histograms and normal quantile plots

9.5.1 Introduction

Histograms, or any other plots, can be annotated as well. I wanted especially to illustrate one thing, though. On a scatterplot, we plotted a lowess curve, which gave a more or less assumption-free picture of the trend. In the same way, we can put a “smooth curve” on a histogram, to get a sense of what the underlying distribution might be. Let's illustrate with our car gas mileages and weights.

```
R> hist(MPG)
```

²⁸One of the other options on `las` will fix the last one.



9.5.2 Changing bar width

I think there are not enough bars²⁹, but we know how to fix that, because we saw the help file before. The key is `breaks`, but I am lazy, so I am going to set up “14 to 40 by 2’s” beforehand:

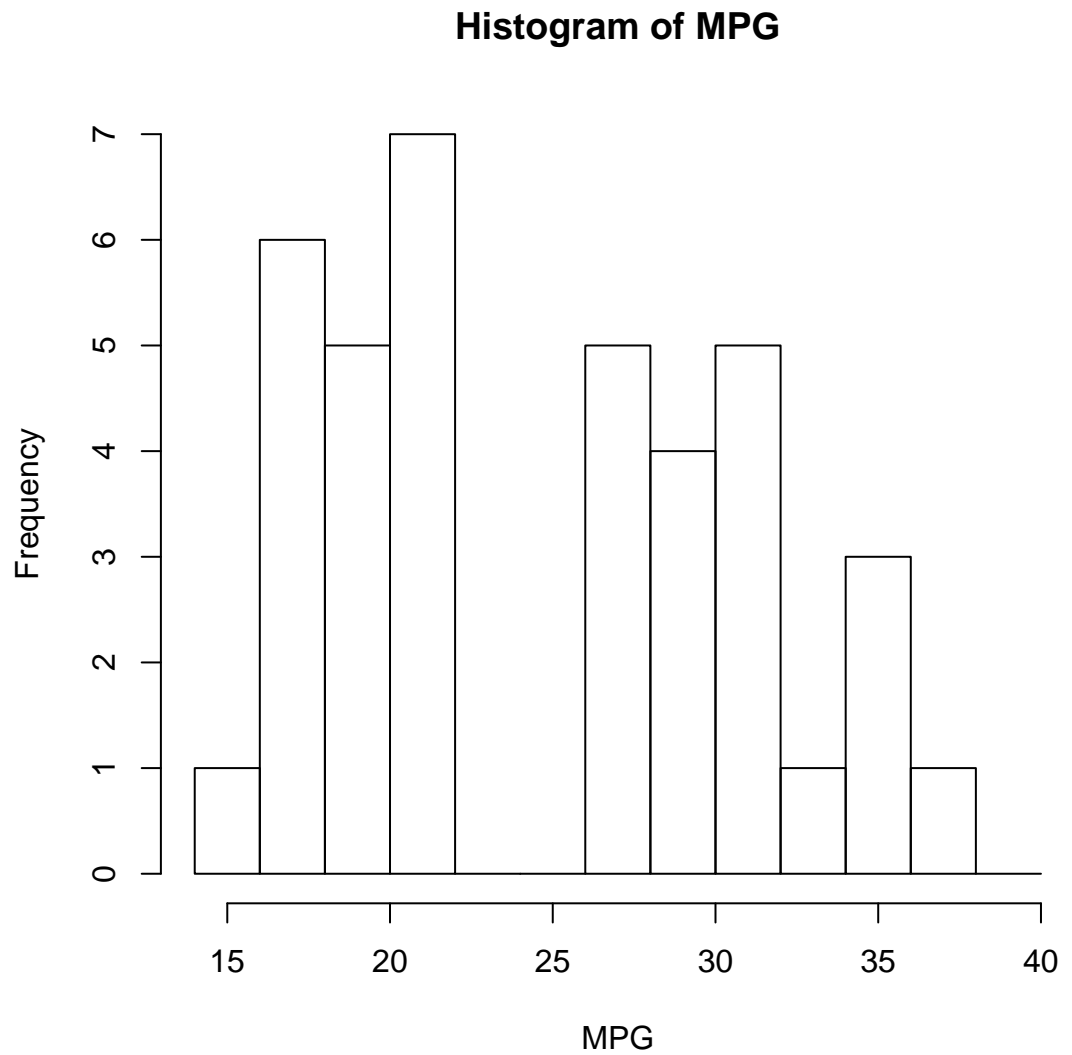
²⁹Because I think this is an over-smoothed picture of the distribution.

```
R> mybreaks=seq(14,40,2)
R> mybreaks

[1] 14 16 18 20 22 24 26 28 30 32 34 36 38 40

and then30

R> hist(MPG,breaks=mybreaks)
```



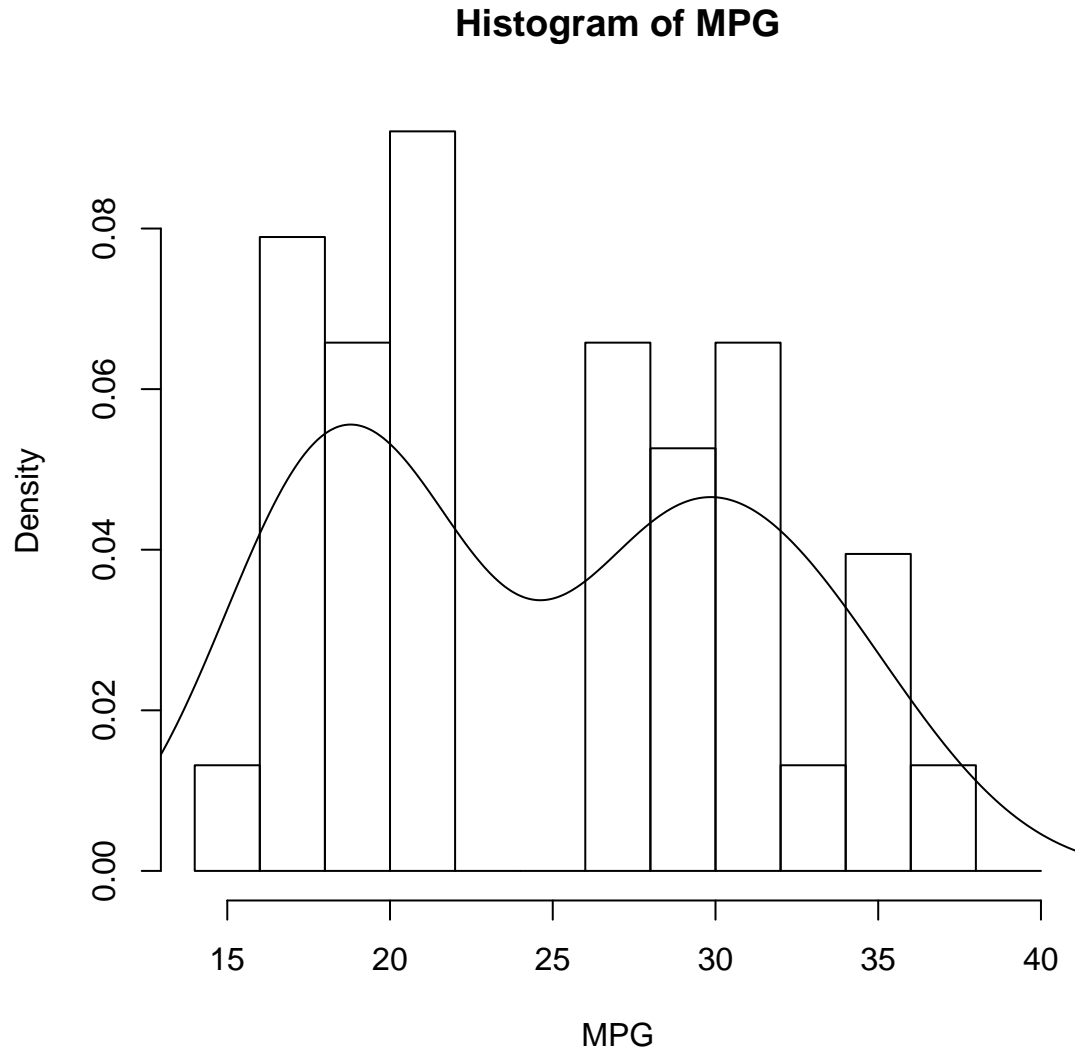
³⁰And this has too many, but I did it for dramatic effect.

This appears to have a “hole” in it, as the cars divide into low-mpg and high-mpg vehicles.

9.5.3 Kernel density curves

What we are going to do now is to put a **kernel density curve** on this, which is a smoothed version of the histogram, designed to show us what the underlying distribution might look like. There is one piece of work we have to do first, though: we have to make the histogram show not frequencies but *relative* frequencies (out of the total), and *then* we can put the kernel density curve on the histogram with `lines`:

```
R> hist(MPG,breaks=mybreaks,probability=T)
R> lines(density(MPG))
```



Just so that you know what the vertical scale represents: there were 6 cars out of 38 with `mpg` between 16 and 18, which is a fraction

```
R> 6/38
```

```
[1] 0.1578947
```

of the whole. But the interval is $18 - 16 = 2$ mpg wide, so to get the “density” we have to divide by 2 as well:

```
R> (6/38)/2
```



```
[1] 0.07894737
```

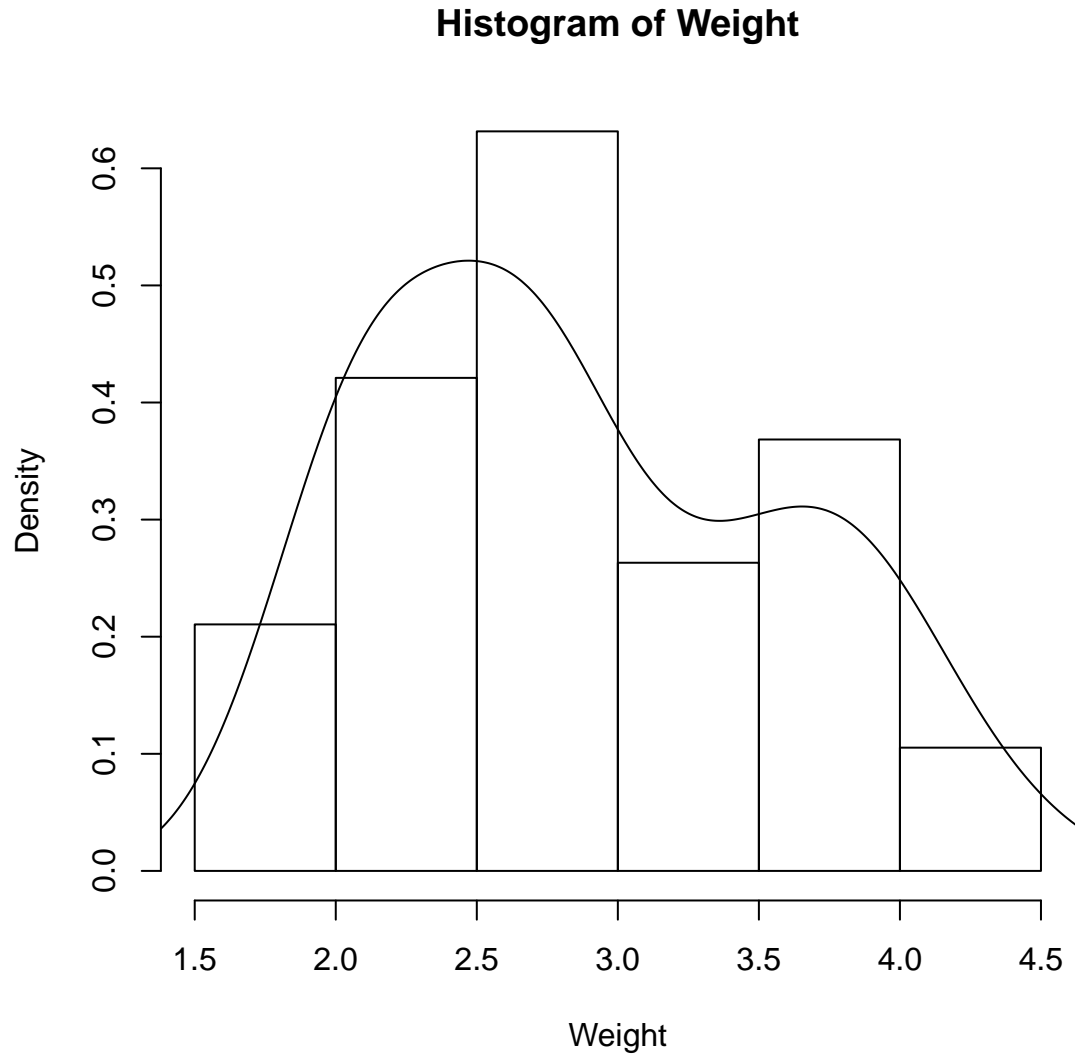
which is how high that bar goes on the second histogram.

The density curve says that we really do have two peaks;³¹ the hole in the middle is really too prominent to be just chance. Note how the curve smooths out those fake non-peaks 18–20 and 28–30, which are lower than their neighbours.

The same thing for the weights. Do we have two peaks there as well?

```
R> hist(Weight,probability=T)
R> lines(density(Weight))
```

³¹A “bimodal” distribution.



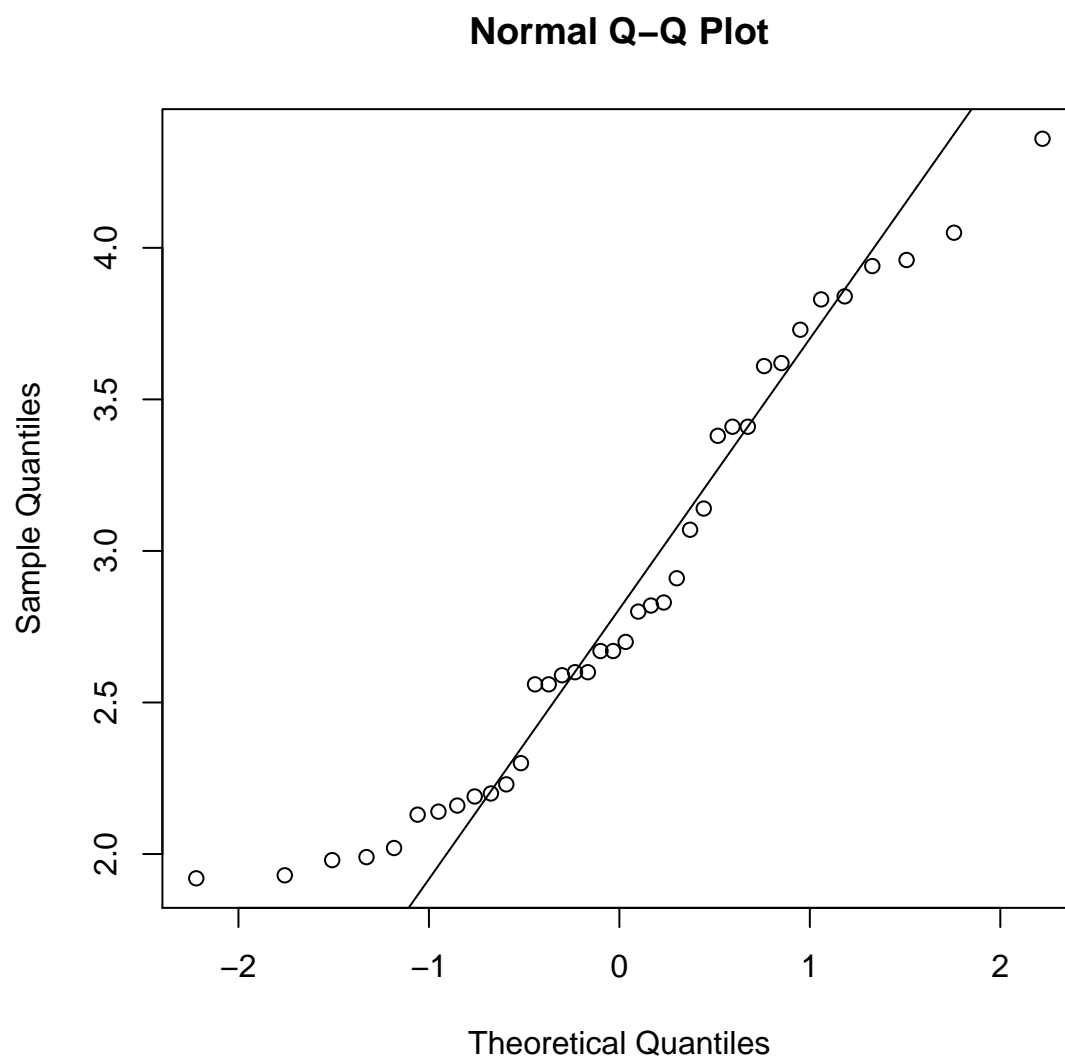
The kernel density curve is not so convinced that there is a “hole” in this one. It looks more as if the distribution is skewed to the right.

9.5.4 Assessing normality

Histograms with kernel density curves are a good way to get a sense of what kind of distribution shape we have, but they are a poor answer to another question: “are my data normally distributed”? Sometimes this matters, as for

example when you're doing a piece of inference³² that was derived from normally distributed data, like a *t*-test.³³ The best way to tackle *that* is via a **normal quantile plot**. Here's how it goes for the weights:

```
R> qqnorm(Weight)
R> qqline(Weight)
```



³²Confidence interval or hypothesis test.

³³This often matters less than you'd think.

This is a plot of the data in order (vertical scale) against the smallest, second-smallest, third-smallest, . . . values that you'd expect from a normal distribution. If the data were *exactly* normal, they'd follow the straight line on the plot.³⁴ Of course, real data are never *exactly* normal, in the same way that scatterplots are never *exactly* linear, so we have to ask ourselves how much non-linearity we're willing to tolerate.

What you're looking for on one of these plots is any obvious departures from the points lying on a line. Often the largest or smallest data values give a clue. Here the smallest ones, at the bottom left, curve away from the line. If you pause to think about it, this means the smallest values are *not small enough* to be normal: they are too bunched up together.

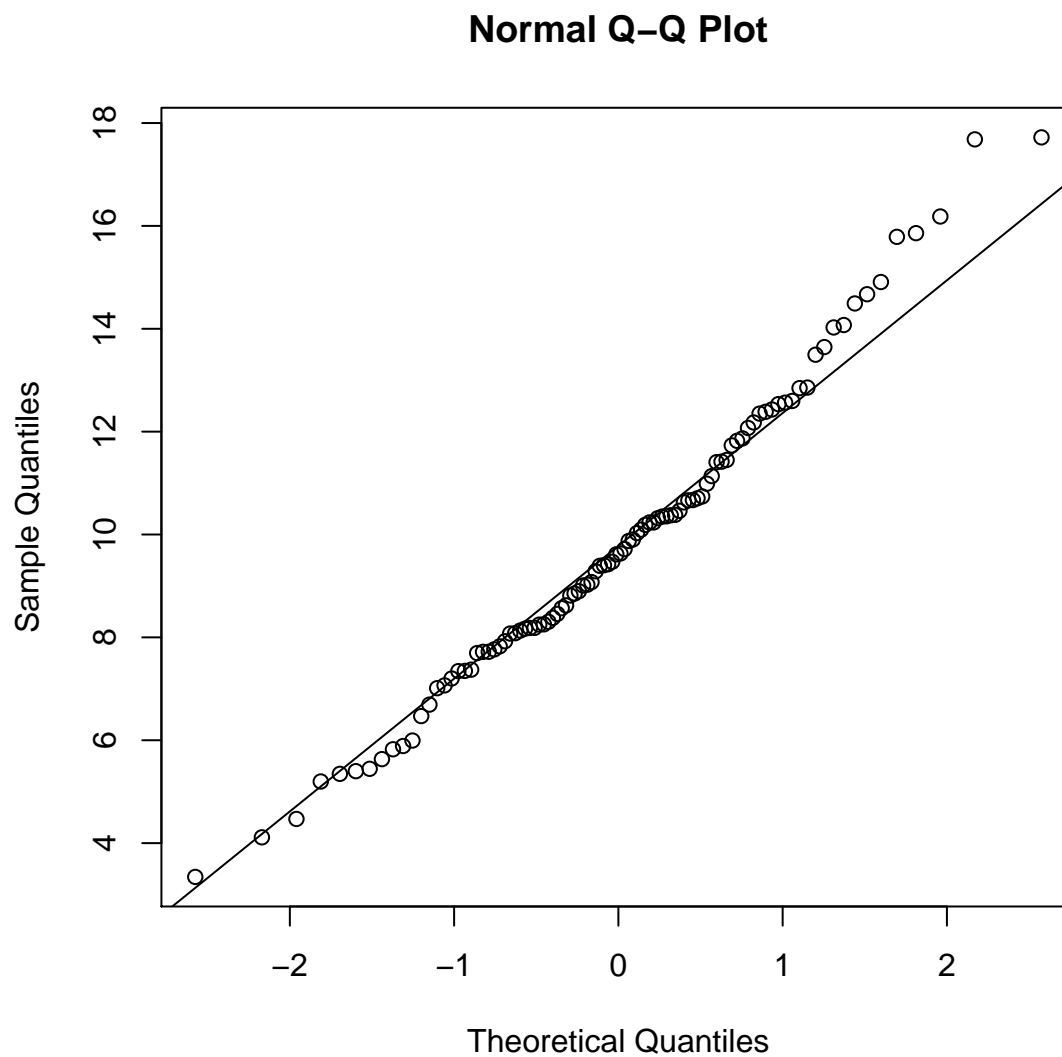
There is a small deviation from the line at the top as well. The largest values are actually a tiny bit too small, so that (oddly) there is bunching up at both ends.

We can get a sense of what a normal quantile plot might look like by generating random data from different distributions that are or are not normal, and see what we get.

First, an example of what the plot looks like when the data actually *are* normal, so the points should follow the line:

```
R> set.seed(457297)
R> z=rnorm(100,10,3)
R> qqnorm(z)
R> qqline(z)
```

³⁴Which was drawn by the `qqline`.

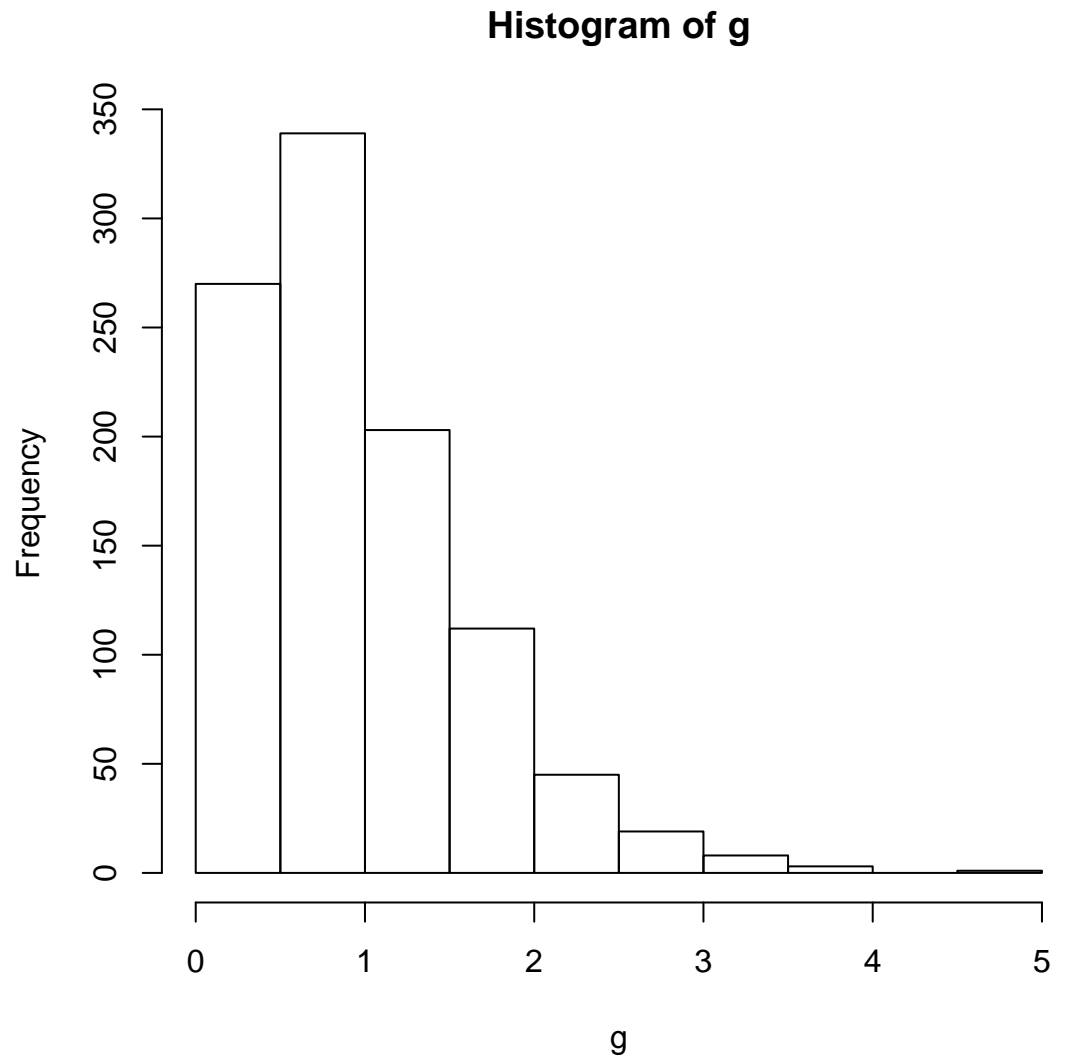


These follow the line pretty well, and the pattern of circles is not obviously curved. The high ones are a bit too high, but we know that this actually *is* chance, because we know what the true distribution of the data values is.

Let's make something right-skewed next:³⁵

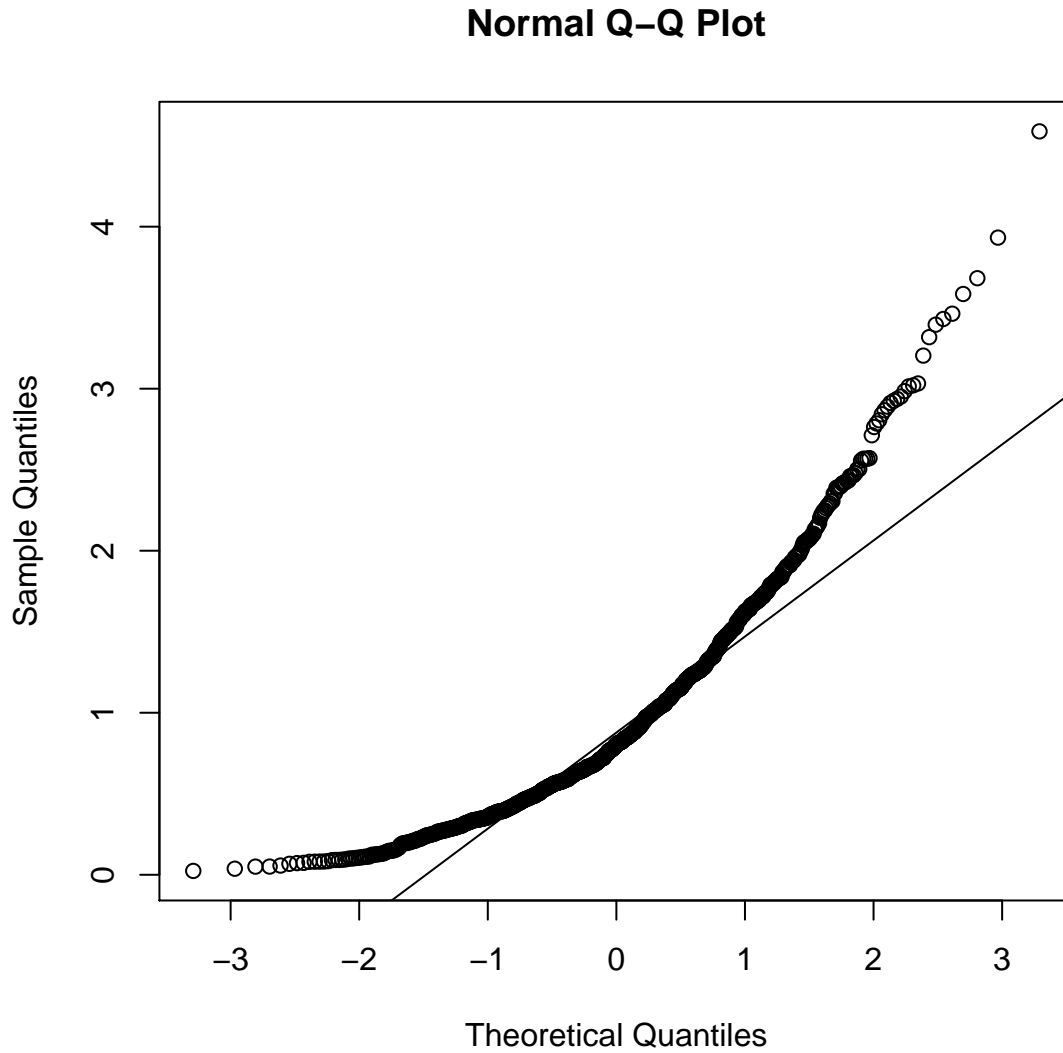
```
R> g=rgamma(1000,2,2)
R> hist(g)
```

³⁵Don't worry if you don't know what the gamma distribution is. Just think of it as a machine for producing right-skewed data.



The normal quantile plot should show a problem here:

```
R> qqnorm(g)
R> qqline(g)
```



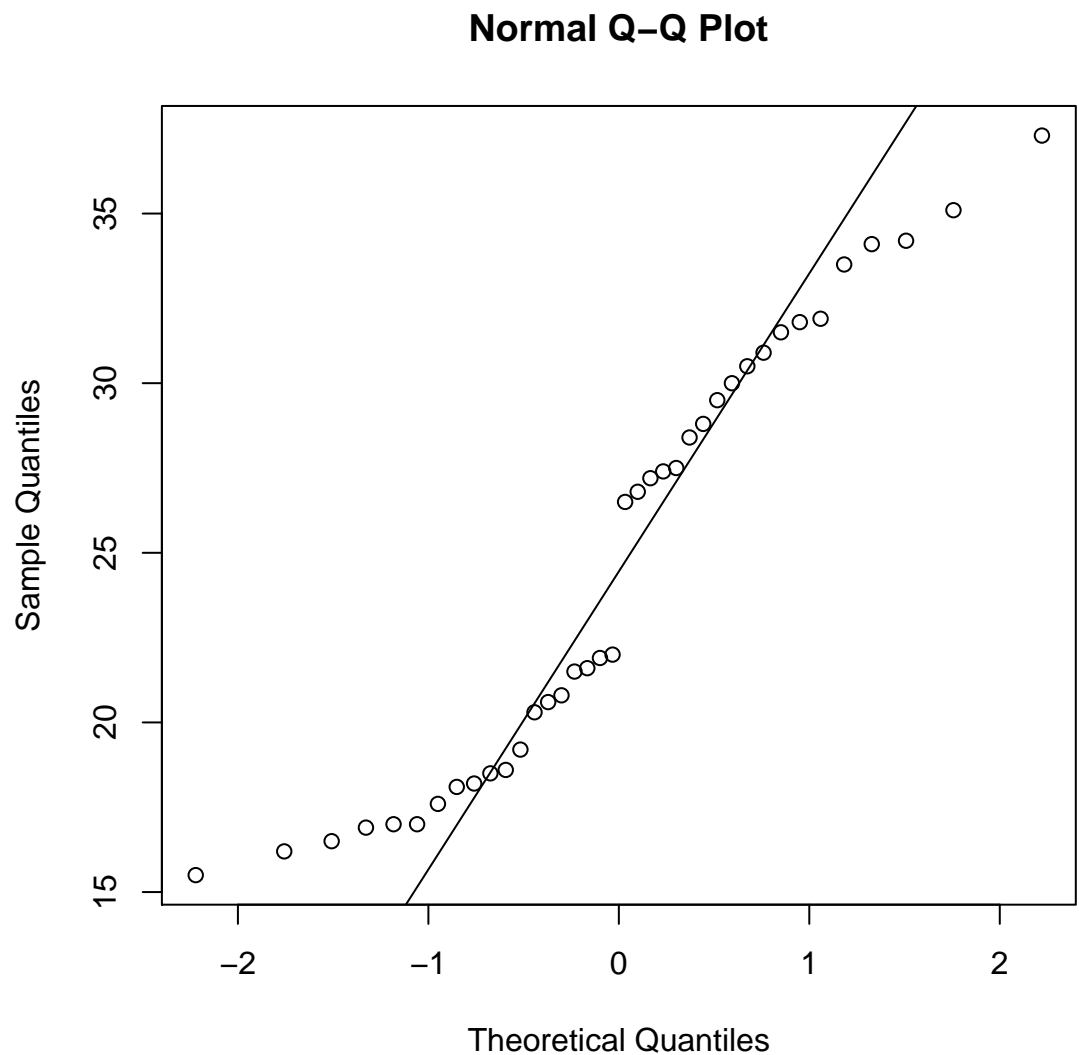
This shows a big-time curve, and doesn't follow the line at all. A little investigation reveals why it's not normal: the values at the bottom are not small enough (too bunched up), and the values at the top are too big (too spread out) for the normal. So this points to the data being skewed to the right.

You might guess what a normal quantile plot of skewed-left data would look like: curved the other way, so that the small values are spread out and the large ones are bunched up.

When you draw a normal quantile plot for actual data, it won't be nearly as clear-cut as this one. I had to go all the way up to a sample size of 1000 to get this, because the picture even for $n = 100$ wasn't clear enough to illustrate what was going on.

All right, let's look at our car `mpgs`, with two peaks. That isn't normally distributed. How does that show up on a normal quantile plot?

```
R> qqnorm(MPG)  
R> qqline(MPG)
```



You can see the hole in the middle of the data from the large vertical gap in the circles. If you look at the extreme values, both the top and bottom ends are *too bunched up* to be normal. This isn't quite what the histogram suggests: there appear to be normal-looking upper and lower tails. But R is testing the fit of a normal distribution with a certain mean and SD to the *whole* data. What mean and SD? Well, the sample mean and SD for the `mpg` data are these:

```
R> xbar=mean(MPG)
R> xbar

[1] 24.76053

R> s=sd(MPG)
R> s

[1] 6.547314
```

Now, there's a rule³⁶ that connects a normal distribution's mean and SD to how much of it is where. Let's take the 95 part of it: 95% of a normal distribution with this mean and SD should be between 11.7 and 37.9, and the other 5% should be outside that interval.

How many of the actual data values are outside? Let's count them and see. I've sorted the values to make the counting easier.

```
R> sort(MPG)

[1] 15.5 16.2 16.5 16.9 17.0 17.0 17.6 18.1 18.2 18.5 18.6 19.2 20.3 20.6 20.8
[16] 21.5 21.6 21.9 22.0 26.5 26.8 27.2 27.4 27.5 28.4 28.8 29.5 30.0 30.5 30.9
[31] 31.5 31.8 31.9 33.5 34.1 34.2 35.1 37.3
```

Nothing below 11.7 and nothing above 37.9. A normal distribution would say that 95% of values are between 11.7 and 37.9, but for our `mpg` figures 100% of them are! That is, our data are too bunched up to be normal, which is what the normal quantile plot said.

Let's tidy up after ourselves:

```
R> detach(cars)
```

9.6 Selecting and combining data

9.6.1 Selecting rows and columns of a data frame

The basic way of arranging data is the **data frame**, like our `cars` from above. Here are the first few rows:

```
R> head(cars)
```

³⁶The 68-95-99.7 rule or "empirical rule".

| | Car | MPG | Weight | Cylinders | Horsepower | Country |
|---|----------------|------|--------|-----------|------------|---------|
| 1 | Buick Skylark | 28.4 | 2.67 | 4 | 90 | U.S. |
| 2 | Dodge Omni | 30.9 | 2.23 | 4 | 75 | U.S. |
| 3 | Mercury Zephyr | 20.8 | 3.07 | 6 | 85 | U.S. |
| 4 | Fiat Strada | 37.3 | 2.13 | 4 | 69 | Italy |
| 5 | Peugeot 694 SL | 16.2 | 3.41 | 6 | 133 | France |
| 6 | VW Rabbit | 31.9 | 1.93 | 4 | 71 | Germany |

You can refer to things in a data frame in a couple of ways. If you want a variable, you can refer to it using \$:

```
R> cars$Cylinders
```

```
[1] 4 4 6 4 6 4 4 4 8 5 8 6 4 4 4 6 4 6 6 8 4 6 6 4 4 8 6 8 4 4 6 4 8 4 4 4 8 8
```

You can also refer to things in a data frame by picking out which number row and column they are. For example:

```
R> cars[4,2]
```

```
[1] 37.3
```

This is the 4th row and 2nd column. That is, it's the Fiat Strada's (4th row) MPG (2nd column).

You can select all of a row of a data frame as below. This one is all the information for the Fiat Strada:

```
R> cars[4,]
```

| | Car | MPG | Weight | Cylinders | Horsepower | Country |
|---|-------------|------|--------|-----------|------------|---------|
| 4 | Fiat Strada | 37.3 | 2.13 | 4 | 69 | Italy |

and as well as selecting a column by name, you can select it by number. Here's another way to select all the `Cylinder` values:

```
R> cars[,4]
```

```
[1] 4 4 6 4 6 4 4 4 8 5 8 6 4 4 4 6 4 6 6 8 4 6 6 4 4 8 6 8 4 4 6 4 8 4 4 4 8 8
```

As we saw before, just `Cylinders` won't work for this, but

```
R> attach(cars)
```

```
R> Cylinders
```

```
[1] 4 4 6 4 6 4 4 4 8 5 8 6 4 4 4 6 4 6 6 8 4 6 6 4 4 8 6 8 4 4 6 4 8 4 4 4 8 8
```

will, because `attach`ing a data frame makes the names of the columns available as variables.

You can also select several rows or columns. Bear in mind that `6:9` means "6 through 9" and `c(3,5)` means "just 3 and 5":

```
R> cars[6:9,c(3,5)]
```

```

      Weight Horsepower
6    1.93           71
7    2.20           70
8    1.98           65
9    4.36          155

R> cars[6:9,c("Car","Cylinders")]

```

```

      Car Cylinders
6      VW Rabbit      4
7  Plymouth Horizon      4
8      Mazda GLC      4
9 Buick Estate Wagon      8

```

9.6.2 Selecting based on conditions

The logical vector

Perhaps the most useful reason for knowing how to select things is *selecting things based on conditions*, such as the cars that have 8 cylinders, or the cars whose gas mileage is bigger than 30. The starting point for this is the **logical vector**. Let's start with the cars that have 8 cylinders:

```

R> Cylinders==8
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
[25] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
[37]  TRUE  TRUE

```

This is a vector of 38 trues and falses, one for each car. There's a **TRUE** for each car that has 8 cylinders and a **FALSE** for each car that has some other number of cylinders.

Note the double equals `==` in the above. This is a "logical equals": it produces a **TRUE** if the statement is logically true,³⁷ and a **FALSE** otherwise. What is potentially opaque³⁸ is if you want to store this logical vector in a variable. You get something like this:

```
R> has8=Cylinders==8
```

which is way too confusing for me. I'd rather write it like this:

```
R> has8=(Cylinders==8)
```

which emphasizes what's going on: first we create a logical vector according to whether each car has 8 cylinders or not, and then we store it in **has8**.

³⁷That is, if the car actually *did* have 8 cylinders.

³⁸Meaning, literally, that you can't see through it.

The nice thing about logical vectors is that you can use them to select things. Let's say we wanted to know where the 8-cylinder cars come from. This means that we need to look at the `Country` just for the 8-cylinder cars. That is done just like this:

```
R> Country[has8]
```

```
[1] U.S. U.S. U.S. U.S. U.S. U.S. U.S. U.S.
Levels: France Germany Italy Japan Sweden U.S.
```

They are *all* from the US. What about the gas mileages of those cars?

```
R> MPG[has8]
```

```
[1] 16.9 19.2 18.5 17.6 18.2 15.5 16.5 17.0
```

They are pretty uniformly bad.

This is, in probability terms, the conditional distribution of gas mileage given that a car has an 8 cylinder engine.

Or we could look at all the variables for the 8-cylinder cars by selecting the rows of `cars` for which `has8` is TRUE, and selecting all the columns:

```
R> cars[has8,]
```

| | Car | MPG | Weight | Cylinders | Horsepower | Country |
|----|---------------------------|------|--------|-----------|------------|---------|
| 9 | Buick Estate Wagon | 16.9 | 4.36 | 8 | 155 | U.S. |
| 11 | Chevy Malibu Wagon | 19.2 | 3.61 | 8 | 125 | U.S. |
| 20 | Chrysler LeBaron Wagon | 18.5 | 3.94 | 8 | 150 | U.S. |
| 26 | Ford LTD | 17.6 | 3.73 | 8 | 129 | U.S. |
| 28 | Dodge St Regis | 18.2 | 3.83 | 8 | 135 | U.S. |
| 33 | Ford Country Squire Wagon | 15.5 | 4.05 | 8 | 142 | U.S. |
| 37 | Mercury Grand Marquis | 16.5 | 3.96 | 8 | 138 | U.S. |
| 38 | Chevy Caprice Classic | 17.0 | 3.84 | 8 | 130 | U.S. |

This way you get to see the names of the cars.

Less than or greater than

Another kind of logical condition is a less-than or a greater-than. Which cars have MPG of 30 or more? There are two ways to do it: a direct way and a step-by-step way. The direct way is:

```
R> cars[MPG>=30,]
```

| | Car | MPG | Weight | Cylinders | Horsepower | Country |
|---|------------------|------|--------|-----------|------------|---------|
| 2 | Dodge Omni | 30.9 | 2.23 | 4 | 75 | U.S. |
| 4 | Fiat Strada | 37.3 | 2.13 | 4 | 69 | Italy |
| 6 | VW Rabbit | 31.9 | 1.93 | 4 | 71 | Germany |
| 7 | Plymouth Horizon | 34.2 | 2.20 | 4 | 70 | U.S. |

| | | | | | | |
|----|-----------------|------|------|---|----|---------|
| 8 | Mazda GLC | 34.1 | 1.98 | 4 | 65 | Japan |
| 13 | VW Dasher | 30.5 | 2.19 | 4 | 78 | Germany |
| 15 | Dodge Colt | 35.1 | 1.92 | 4 | 80 | Japan |
| 17 | VW Scirocco | 31.5 | 1.99 | 4 | 71 | Germany |
| 25 | Datsun 210 | 31.8 | 2.02 | 4 | 65 | Japan |
| 30 | Chevette | 30.0 | 2.16 | 4 | 68 | U.S. |
| 35 | Pontiac Phoenix | 33.5 | 2.56 | 4 | 90 | U.S. |

The step-by-step way, which I find is nicer for thinking about things, is to create a logical vector first which is `TRUE` for the things you want to select, and then use that to do the selection:³⁹

```
R> hi.mpg=(MPG>=30)
R> cars[hi.mpg,]
```

| | Car | MPG | Weight | Cylinders | Horsepower | Country |
|----|------------------|------|--------|-----------|------------|---------|
| 2 | Dodge Omni | 30.9 | 2.23 | 4 | 75 | U.S. |
| 4 | Fiat Strada | 37.3 | 2.13 | 4 | 69 | Italy |
| 6 | VW Rabbit | 31.9 | 1.93 | 4 | 71 | Germany |
| 7 | Plymouth Horizon | 34.2 | 2.20 | 4 | 70 | U.S. |
| 8 | Mazda GLC | 34.1 | 1.98 | 4 | 65 | Japan |
| 13 | VW Dasher | 30.5 | 2.19 | 4 | 78 | Germany |
| 15 | Dodge Colt | 35.1 | 1.92 | 4 | 80 | Japan |
| 17 | VW Scirocco | 31.5 | 1.99 | 4 | 71 | Germany |
| 25 | Datsun 210 | 31.8 | 2.02 | 4 | 65 | Japan |
| 30 | Chevette | 30.0 | 2.16 | 4 | 68 | U.S. |
| 35 | Pontiac Phoenix | 33.5 | 2.56 | 4 | 90 | U.S. |

Or you might like to ask “how many cylinders do the cars have whose MPG is 30 or more?” Using the `hi.mpg` just defined:

```
R> Cylinders[hi.mpg]

[1] 4 4 4 4 4 4 4 4 4 4 4
```

They all have 4 cylinders. This is scarcely a surprise.⁴⁰

This, in probability terms, is the conditional distribution of number of cylinders given that the MPG is 30 or more.

And, or, not

How many cars in the dataset are *not* from the US? To answer this, we first create a logical vector that is `TRUE` when a car is *not* from the US and `FALSE` otherwise. R uses `!` to mean “not”:

³⁹This is greater than or equal to; strictly greater than would be `MPG>30`.

⁴⁰Compare the `Horsepower` of these cars with the cars that have 8-cylinder engines, above.

```
R> not.us=(Country!="U.S.")
R> not.us
```

```
[1] FALSE FALSE FALSE TRUE TRUE TRUE FALSE TRUE FALSE TRUE FALSE FALSE
[13] TRUE FALSE TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE FALSE TRUE
[25] TRUE FALSE TRUE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE TRUE
[37] FALSE FALSE
```

This needs exact matching: the letters have to be uppercase and the dots have to be there. To count them, use `table`:

```
R> table(not.us)
```

```
not.us
FALSE  TRUE
    22    16
```

16 of the 38 cars in the data set are not from the US.

Another way of doing this is simply to use `table` on the `Country` variable:

```
R> table(Country)
```

```
Country
France Germany  Italy  Japan  Sweden  U.S.
      1      5      1      7      2     22
```

and with a little work you find that 22 of the 38 cars *are* from the US, so the rest are not, or you can get it by adding up: $1 + 5 + 1 + 7 + 2 = 16$.

Which cars are from the US *and* have 4-cylinder engines? In one step like this:

```
R> cars[Country=="U.S." & Cylinders==4,]
```

| | Car | MPG | Weight | Cylinders | Horsepower | Country |
|----|------------------|------|--------|-----------|------------|---------|
| 1 | Buick Skylark | 28.4 | 2.67 | 4 | 90 | U.S. |
| 2 | Dodge Omni | 30.9 | 2.23 | 4 | 75 | U.S. |
| 7 | Plymouth Horizon | 34.2 | 2.20 | 4 | 70 | U.S. |
| 14 | Ford Mustang 4 | 26.5 | 2.59 | 4 | 88 | U.S. |
| 30 | Chevette | 30.0 | 2.16 | 4 | 68 | U.S. |
| 32 | AMC Spirit | 27.4 | 2.67 | 4 | 80 | U.S. |
| 35 | Pontiac Phoenix | 33.5 | 2.56 | 4 | 90 | U.S. |

You can see that both conditions are satisfied for these cars.

Or you can define two logical variables, one for “country is US” and one for “cylinders is 4”, and then plug them in with an `&` between. Or you can do it like this:

```
R> i.want=(Country=="U.S." & Cylinders==4)
R> cars[i.want,]
```

| | Car | MPG | Weight | Cylinders | Horsepower | Country |
|----|------------------|------|--------|-----------|------------|---------|
| 1 | Buick Skylark | 28.4 | 2.67 | 4 | 90 | U.S. |
| 2 | Dodge Omni | 30.9 | 2.23 | 4 | 75 | U.S. |
| 7 | Plymouth Horizon | 34.2 | 2.20 | 4 | 70 | U.S. |
| 14 | Ford Mustang 4 | 26.5 | 2.59 | 4 | 88 | U.S. |
| 30 | Chevette | 30.0 | 2.16 | 4 | 68 | U.S. |
| 32 | AMC Spirit | 27.4 | 2.67 | 4 | 80 | U.S. |
| 35 | Pontiac Phoenix | 33.5 | 2.56 | 4 | 90 | U.S. |

R's term for "or" is `|`. So this is how you find the cars that either come from Japan or have 6 cylinders:⁴¹

```
R> cars[Country=="Japan" | Cylinders==6,]
```

| | Car | MPG | Weight | Cylinders | Horsepower | Country |
|----|-----------------------|------|--------|-----------|------------|---------|
| 3 | Mercury Zephyr | 20.8 | 3.07 | 6 | 85 | U.S. |
| 5 | Peugeot 694 SL | 16.2 | 3.41 | 6 | 133 | France |
| 8 | Mazda GLC | 34.1 | 1.98 | 4 | 65 | Japan |
| 12 | Dodge Aspen | 18.6 | 3.62 | 6 | 110 | U.S. |
| 15 | Dodge Colt | 35.1 | 1.92 | 4 | 80 | Japan |
| 16 | Datsun 810 | 22.0 | 2.82 | 6 | 97 | Japan |
| 18 | Chevy Citation | 28.8 | 2.60 | 6 | 115 | U.S. |
| 19 | Olds Omega | 26.8 | 2.70 | 6 | 115 | U.S. |
| 21 | Datsun 510 | 27.2 | 2.30 | 4 | 97 | Japan |
| 22 | AMC Concord D/L | 18.1 | 3.41 | 6 | 120 | U.S. |
| 23 | Buick Century Special | 20.6 | 3.38 | 6 | 105 | U.S. |
| 25 | Datsun 210 | 31.8 | 2.02 | 4 | 65 | Japan |
| 27 | Volvo 240 GL | 17.0 | 3.14 | 6 | 125 | Sweden |
| 29 | Toyota Corona | 27.5 | 2.56 | 4 | 95 | Japan |
| 31 | Ford Mustang Ghia | 21.9 | 2.91 | 6 | 109 | U.S. |
| 36 | Honda Accord LX | 29.5 | 2.14 | 4 | 68 | Japan |

9.7 The apply family

9.7.1 Introduction

Sometimes you don't need to select anything to get what you want. R is very good at applying things to entire data frames and vectors, for example calculating means by data frame column or by subgroup. If you're an old-fashioned kind of computer programmer like me,⁴² you'll be used to using loops for this kind of thing. Loops are actually slower than the do-it-all-at-once functions in R (though you can still use them if you want).

⁴¹Or both. R also has a function `xor` that does "exclusive or": "either A or B but not both".

⁴²I grew up with FORTRAN.

9.7.2 `apply`

The first of these functions is `apply`. Let's go back to the orange tree circumferences for this:

```
R> circum
```

| | A | B | C | D | E |
|---|-----|-----|-----|-----|-----|
| 1 | 30 | 30 | 30 | 33 | 32 |
| 2 | 51 | 58 | 49 | 69 | 62 |
| 3 | 75 | 87 | 81 | 111 | 112 |
| 4 | 108 | 115 | 125 | 156 | 167 |
| 5 | 115 | 120 | 142 | 172 | 179 |
| 6 | 139 | 142 | 174 | 203 | 209 |
| 7 | 140 | 145 | 177 | 203 | 214 |

```
R> apply(circum,1,mean)
```

```
[1] 31.0 57.8 93.2 134.2 145.6 173.4 175.8
```

This gets you the row means, in other words, the mean circumference of all the trees at each time. The increasing values show that all the trees were growing. The anatomy of `apply`⁴³ is that it needs three things: a data frame⁴³ to work on, a dimension to work along, 1 being rows and 2 being columns, and finally a function to calculate. This could be `mean`, `median`, `sd`, `IQR`, or anything else that calculates a number. Column means don't make much sense here, but we can do it anyway:

```
R> apply(circum,2,mean)
```

| A | B | C | D | E |
|----------|----------|-----------|-----------|-----------|
| 94.00000 | 99.57143 | 111.14286 | 135.28571 | 139.28571 |

This the mean circumference over all the times for each tree.

Here's a funky⁴⁴ one:

```
R> apply(circum,1,quantile,probs=0.25)
```

```
[1] 30 51 81 115 120 142 145
```

This one finds the first quartile Q1 of each row. The way you'd do that for an ordinary vector of numbers, such as the first column of `circum`, is

```
R> quantile(circum[,1],probs=0.25)
```

```
25%
63
```

⁴³Or R matrix.

⁴⁴"Funky" originally meant "smelly".

This finds Q1 for the first row. The problem is that `quantile` needs extra stuff fed to it, since it needs to know whether you want Q1, Q3 or some other percentile. The way you handle this with `apply` is that after you say what function you want to calculate for each row,⁴⁵ you put in any extra stuff. Anything that `apply` doesn't know how to handle gets handed on to `quantile`⁴⁶ to deal with. Since `quantile` knows what `probs=0.25` means,⁴⁷ all is good.

There are several other variations on `apply`. We'll take a look at them next.

9.7.3 `tapply` and `aggregate`

The one I use most is `tapply`, which works like `aggregate`: it calculates something for each of a number of groups. Let's go back to our cars to illustrate how it works. For example, you might want the mean gas mileage by country:

```
R> tapply(MPG, Country, mean)

      France  Germany    Italy   Japan   Sweden    U.S. 
16.20000 27.14000 37.30000 29.60000 19.30000 22.99545
```

R makes you a nice table showing the results. Again, it doesn't have to be `mean`; you can use any function that gives you back a number, and if you're using something like `quantile` that requires extra stuff, you put that on the end, as for `apply`.

Sometimes you want to calculate means for variable combinations. Here's how you find the mean for each combination of `Country` and `Cylinders`:

```
R> tapply(MPG, list(Country, Cylinders), mean)

           4      5      6      8
France      NA      NA 16.20000    NA
Germany 28.85000 20.3      NA      NA
Italy    37.30000      NA      NA      NA
Japan    30.86667      NA 22.00000      NA
Sweden   21.60000      NA 17.00000      NA
U.S.      30.12857      NA 22.22857 17.425
```

There are lots of NA results, which is R's way of saying that there were no data, so it couldn't find a mean. For example, all the 8-cylinder cars are American, so there are no 8-cylinder means for any of the other countries.

What happens if you feed `tapply` a function that returns more than one value? Let's try it and see:

```
R> tapply(MPG, Country, quantile)
```

⁴⁵Or column.

⁴⁶Or whatever.

⁴⁷"Calculate Q1".

```

$France
  0%  25%  50%  75% 100%
16.2 16.2 16.2 16.2 16.2

$Germany
  0%  25%  50%  75% 100%
20.3 21.5 30.5 31.5 31.9

$Italy
  0%  25%  50%  75% 100%
37.3 37.3 37.3 37.3 37.3

$Japan
  0%  25%  50%  75% 100%
22.00 27.35 29.50 32.95 35.10

$Sweden
  0%  25%  50%  75% 100%
17.00 18.15 19.30 20.45 21.60

$U.S.
  0%  25%  50%  75% 100%
15.500 18.125 20.700 28.150 34.200

```

We get what in R terms is called a **list**: the five-number summary for each country separately. All the information is there, but it doesn't look very nice.

9.7.4 `sapply` and `split`

A nicer look is obtained with uglier code and a different function `sapply`:

```

R> sapply(split(MPG, Country), quantile)
      France Germany Italy Japan Sweden  U.S.
0%      16.2    20.3  37.3 22.00  17.00 15.500
25%      16.2    21.5  37.3 27.35  18.15 18.125
50%      16.2    30.5  37.3 29.50  19.30 20.700
75%      16.2    31.5  37.3 32.95  20.45 28.150
100%     16.2    31.9  37.3 35.10  21.60 34.200

```

What does `split` do? This:

```

R> split(MPG, Country)

$France
[1] 16.2

$Germany

```

```
[1] 31.9 20.3 30.5 31.5 21.5
```

```
$Italy
[1] 37.3
```

```
$Japan
[1] 34.1 35.1 22.0 27.2 31.8 27.5 29.5
```

```
$Sweden
[1] 21.6 17.0
```

```
$U.S.
[1] 28.4 30.9 20.8 34.2 16.9 19.2 18.6 26.5 28.8 26.8 18.5 18.1 20.6 17.6 18.2
[16] 30.0 21.9 27.4 15.5 33.5 16.5 17.0
```

It makes a list of the MPG values for each Country. `sapply` then applies a function, here `quantile`, to each element of the list. The nice thing about this is that the results get arranged into a pretty table.

`tapply` does the same kind of thing as `aggregate`, which we saw earlier. The data frame `cars` has to be specified explicitly:

```
R> aggregate(MPG~Country,data=cars,mean)
```

| | Country | MPG |
|---|---------|----------|
| 1 | France | 16.20000 |
| 2 | Germany | 27.14000 |
| 3 | Italy | 37.30000 |
| 4 | Japan | 29.60000 |
| 5 | Sweden | 19.30000 |
| 6 | U.S. | 22.99545 |

A data frame, a column of countries and a column of mean MPGs.

```
R> aggregate(MPG~Country+Cylinders,data=cars,mean)
```

| | Country | Cylinders | MPG |
|----|---------|-----------|----------|
| 1 | Germany | 4 | 28.85000 |
| 2 | Italy | 4 | 37.30000 |
| 3 | Japan | 4 | 30.86667 |
| 4 | Sweden | 4 | 21.60000 |
| 5 | U.S. | 4 | 30.12857 |
| 6 | Germany | 5 | 20.30000 |
| 7 | France | 6 | 16.20000 |
| 8 | Japan | 6 | 22.00000 |
| 9 | Sweden | 6 | 17.00000 |
| 10 | U.S. | 6 | 22.22857 |
| 11 | U.S. | 8 | 17.42500 |

Two columns of categories, and one of means. Not a table, but a data frame containing all combinations of `Country` and `Cylinders` that were actually observed.⁴⁸

```
R> aggregate(MPG~Country,data=cars,quantile)
```

| | Country | MPG.0% | MPG.25% | MPG.50% | MPG.75% | MPG.100% |
|---|---------|--------|---------|---------|---------|----------|
| 1 | France | 16.200 | 16.200 | 16.200 | 16.200 | 16.200 |
| 2 | Germany | 20.300 | 21.500 | 30.500 | 31.500 | 31.900 |
| 3 | Italy | 37.300 | 37.300 | 37.300 | 37.300 | 37.300 |
| 4 | Japan | 22.000 | 27.350 | 29.500 | 32.950 | 35.100 |
| 5 | Sweden | 17.000 | 18.150 | 19.300 | 20.450 | 21.600 |
| 6 | U.S. | 15.500 | 18.125 | 20.700 | 28.150 | 34.200 |

One column of countries, then five columns of output from `quantile`. R makes column names out of the variable name `MPG` and what `quantile` produces in its output.

`aggregate` also gracefully handles two categories and a vector-valued function:

```
R> aggregate(MPG~Country+Cylinders,data=cars,quantile)
```

| | Country | Cylinders | MPG.0% | MPG.25% | MPG.50% | MPG.75% | MPG.100% |
|----|---------|-----------|--------|---------|---------|---------|----------|
| 1 | Germany | 4 | 21.500 | 28.250 | 31.000 | 31.600 | 31.900 |
| 2 | Italy | 4 | 37.300 | 37.300 | 37.300 | 37.300 | 37.300 |
| 3 | Japan | 4 | 27.200 | 28.000 | 30.650 | 33.525 | 35.100 |
| 4 | Sweden | 4 | 21.600 | 21.600 | 21.600 | 21.600 | 21.600 |
| 5 | U.S. | 4 | 26.500 | 27.900 | 30.000 | 32.200 | 34.200 |
| 6 | Germany | 5 | 20.300 | 20.300 | 20.300 | 20.300 | 20.300 |
| 7 | France | 6 | 16.200 | 16.200 | 16.200 | 16.200 | 16.200 |
| 8 | Japan | 6 | 22.000 | 22.000 | 22.000 | 22.000 | 22.000 |
| 9 | Sweden | 6 | 17.000 | 17.000 | 17.000 | 17.000 | 17.000 |
| 10 | U.S. | 6 | 18.100 | 19.600 | 20.800 | 24.350 | 28.800 |
| 11 | U.S. | 8 | 15.500 | 16.800 | 17.300 | 18.275 | 19.200 |

Whether this last one is actually useful is another matter, however.

9.8 Vectors and matrices in R

9.8.1 Vector addition

Next, let's have a look at how R combines vectors (and matrices).

```
R> u=c(2,3,6,5,7)
```

```
R> k=2
```

⁴⁸The missing combinations don't appear anywhere.

```
R> u+k
```

```
[1] 4 5 8 7 9
```

This adds 2 to each element of **x**. Likewise, this works as you'd expect:

```
R> u
```

```
[1] 2 3 6 5 7
```

```
R> v=c(1,8,3,4,2)
```

```
R> u+v
```

```
[1] 3 11 9 9 9
```

R adds the two vectors element by element.⁴⁹

9.8.2 Scalar multiplication

Likewise:

```
R> k
```

```
[1] 2
```

```
R> u
```

```
[1] 2 3 6 5 7
```

```
R> k*u
```

```
[1] 4 6 12 10 14
```

multiplies all the elements of **x** by 2.⁵⁰

9.8.3 “Vector multiplication”

What about this?

```
R> u
```

```
[1] 2 3 6 5 7
```

```
R> v
```

```
[1] 1 8 3 4 2
```

```
R> u*v
```

```
[1] 2 24 18 20 14
```

⁴⁹This is MATA23's vector addition.

⁵⁰Scalar multiplication.

Figuring this one out depends on how good your times tables are. What has happened is that each element of `u` has been multiplied by the corresponding element of `v`. This is called *elementwise multiplication*.

9.8.4 Combining different-length vectors

What happens if you combine two vectors, but they are different lengths? You might think that R would give you an error, but it doesn't:

```
R> u
[1] 2 3 6 5 7

R> w=c(1,2)
R> u+w
[1] 3 5 7 7 8
Warning message:
In u + w : longer object length is not a multiple of shorter object length
```

R took the first element of `u` and added 1 to it, and the second, and added 2 to it. Then it realized that it had run out of values in `w`, so it went back to the beginning (of `w`) again. So the 3rd element of `u` had 1 added to it, and the 4th had 2 added to it. Lastly the 5th element of `u` has 1 added to it again, and there we stop. R calls this process *recycling*: we re-use the shorter vector to make it match the length of the longer one.

If you think about it, this same recycling idea is applied when you add or multiply a vector by a number: the number is repeated enough times to make it as long as the vector, and then you add or multiply elementwise.

9.8.5 Making matrices

This bit will only make sense to you if you know about matrices. If you don't, skip it!

A matrix is a rectangular array of numbers. In R it looks like, but is subtly different from, a data frame. Here's how you make one:

```
R> A=matrix(1:4,nrow=2,ncol=2)
R> A
      [,1] [,2]
[1,]    1    3
[2,]    2    4
```

The first thing fed into `matrix` is the stuff to make the matrix out of, then we need to say how many rows and columns there are. Strictly, we only need one

of `nrow` and `ncol`, since R already knows the matrix is going to have (here) 4 values.

Notice that the values have been filled in *down the columns*. If you want them to go along the rows instead, do this:

```
R> B=matrix(5:8,nrow=2,ncol=2,byrow=T)
R> B

      [,1] [,2]
[1,]     5     6
[2,]     7     8
```

Again, I could have left out either `nrow` or `ncol`.

9.8.6 Adding matrices

What happens if you add two matrices?

```
R> A

      [,1] [,2]
[1,]     1     3
[2,]     2     4

R> B

      [,1] [,2]
[1,]     5     6
[2,]     7     8

R> A+B

      [,1] [,2]
[1,]     6     9
[2,]     9    12
```

Nothing surprising here. This is matrix addition as we know it.

9.8.7 Multiplying matrices

Now, what has happened here?

```
R> A

      [,1] [,2]
[1,]     1     3
[2,]     2     4

R> B
```

```

      [,1] [,2]
[1,]    5    6
[2,]    7    8

```

```
R> A*B
```

```

      [,1] [,2]
[1,]    5   18
[2,]   14   32

```

This is *not* matrix multiplication. You can do a little detective work, or you can guess based on how R “multiplied” two vectors. It is *elementwise* multiplication: numbers in corresponding places in the two matrices are multiplied.⁵¹

Actual matrix multiplication as you know it⁵² is done like this:

```
R> A
```

```

      [,1] [,2]
[1,]    1    3
[2,]    2    4

```

```
R> B
```

```

      [,1] [,2]
[1,]    5    6
[2,]    7    8

```

```
R> A %*% B
```

```

      [,1] [,2]
[1,]   26   30
[2,]   38   44

```

9.8.8 Linear algebra stuff in R

Also, some other MATA23 things can be done in R. To solve the system of linear equations $Ax = w$ for x , do this:

```
R> A
```

```

      [,1] [,2]
[1,]    1    3
[2,]    2    4

```

```
R> w
```

```
[1] 1 2
```

⁵¹This is otherwise known as the Hadamard product of **A** and **B**. It is the same as the Hadamard product of **B** and **A**. This is in contrast to standard matrix multiplication where $AB \neq BA$ in general.

⁵²Well, maybe.


```
R> solve(A,w)
```

```
[1] 1 0
```

and to find the inverse of a matrix, do this:

```
R> A
```

```
      [,1] [,2]
[1,]     1     3
[2,]     2     4
```

```
R> solve(A)
```

```
      [,1] [,2]
[1,]    -2  1.5
[2,]     1 -0.5
```

9.9 Writing functions in R

9.9.1 Introduction

The way in which R gets things done is the **function**. Every time you do something in R that has a name and something in brackets,⁵³ you are using a function. This includes `c()`, `mean()` and even something more complicated like `lm()`.

9.9.2 The anatomy of a function

A function has two parts: what you feed in, and what you get back out. Let's start with `c()`. This makes vectors out of stuff, like this:⁵⁴

```
R> c(1,2,5)
```

```
[1] 1 2 5
```

which returns (gives you back) the vector containing those three numbers. What goes into `c()` can be a vector, too, so that everything gets glued together:

```
R> u=c(1,2,5)
```

```
R> w=c(8,9)
```

```
R> c(u,w,6)
```

```
[1] 1 2 5 8 9 6
```

or even (but I hate doing it this way, because I find it too confusing):

⁵³() kind of brackets, I mean.

⁵⁴“Consult the Book of Armaments!”

```
R> c(c(1,2,5),c(8,9),6)
[1] 1 2 5 8 9 6
```

So the input to `c()` is one or more vectors, and the output is a vector also. What about `mean()`? This needs exactly one vector for input, like this:

```
R> mean(u)
[1] 2.666667
```

and produces a *number*.⁵⁵ Anything that makes a vector will do as input for `mean()`, such as this:

```
R> mean(c(8,9))
[1] 8.5
```

or even this:

```
R> mean(c(u,w,6))
[1] 5.166667
```

`lm()` is a bit trickier. It needs a *model formula*⁵⁶, an optional data frame, and some other optional stuff:

```
R> y=c(w,11)
R> lm(y~u)
```

Call:

```
lm(formula = y ~ u)
```

Coefficients:

```
(Intercept)          u
      7.3846       0.7308
```

So, what's the output of `lm()`? Well, what you saw above, right? Not so fast. When you put the name of something, or a function call like this, on a line by itself without storing it in a variable, what you actually do is to call that thing's **print method**. `print`, like `plot`, does different things according to what kind of thing you feed into it. `printing` an `lm` object actually calls up `print.lm`, which produces the display you see. To see what is *actually* inside, you can instead use `print.default`, like this:

```
R> print.default(lm(y~u))

$coefficients
(Intercept)          u
 7.3846154    0.7307692
```

⁵⁵The mean, of course.

⁵⁶The thing with a squiggle in it, with the response variable on the left and the explanatory variable(s) on the right.

```

$residuals
      1      2      3
-0.11538462  0.15384615 -0.03846154

$effects
(Intercept)      u
-16.1658075    2.1513264    0.1961161

$rank
[1] 2

$fitted.values
      1      2      3
 8.115385  8.846154 11.038462

$assign
[1] 0 1

$qr
$qr
      (Intercept)      u
1  -1.7320508 -4.618802
2   0.5773503  2.943920
3   0.5773503 -0.999815
attr("assign")
[1] 0 1

$graux
[1] 1.577350 1.019234

$pivot
[1] 1 2

$tol
[1] 1e-07

$rank
[1] 2

attr("class")
[1] "qr"

$df.residual
[1] 1

```

```

$levels
named list()

$call
lm(formula = y ~ u)

$terms
y ~ u
attr("variables")
list(y, u)
attr("factors")
      u
y 0
u 1
attr("term.labels")
[1] "u"
attr("order")
[1] 1
attr("intercept")
[1] 1
attr("response")
[1] 1
attr(".Environment")
<environment: R_GlobalEnv>
attr("predvars")
list(y, u)
attr("dataClasses")
      y      u
"numeric" "numeric"

$model
      y u
1   8 1
2   9 2
3  11 5

attr("class")
[1] "lm"

```

Now you see a whole bunch of stuff. `lm`'s output is a whole bunch of different things: the intercept and slope(s), the fitted values, the residuals, and some other things that we normally don't care about. The point is that the output from `lm` is a whole bunch of *different* things. The R data structure that handles "a whole bunch of different things" is called a `list`, and you access the individual elements of a list using the same `$` notation that you use for getting at columns

of data frames.⁵⁷

9.9.3 Writing a function

All right, let's see if we can write a function that calculates and outputs the range, highest value minus lowest value. You might, for example, be in the business of calculating a lot of ranges,⁵⁸ and you want to automate the process.

The function should have as its input a vector of numbers, and its output will be a single number. The strategy is to find the largest value in the vector (via `max`), the smallest value (via `min`), and subtract them.

The top line of our function looks something like this:

```
R> my.range=function(x)
```

`my.range` is the name I chose to give to the function. I didn't use `range` because there's already an R function called that, and I don't want to confuse things. Then an `=` and the word `function`. Then, in brackets, the input(s) to the function, which in this case is a single vector.

If you know something about programming: `x` is called a *function argument*. You will be able to use the function with any kind of input (a vector with a name, or a list of numbers), but whatever it's called outside the function, it's called `x` inside. Whatever we do to `x` in the function will be done to whatever input we give the function when we use it.

Next comes the actual doing-stuff part of the function. This is enclosed within curly brackets. I'm going to write this function in a fairly long-winded way.⁵⁹ First we have to find the maximum value and save it somewhere. Then find the minimum value and save *it* somewhere. Then take the difference, and return it.⁶⁰

```
R> my.range=function(x)
R> {
R>   a=max(x)
R>   b=min(x)
R>   a-b
R> }
```

What about that last line? I seem to be just printing out the value of `a-b`. This is R's mechanism for output: sending a function's value back out from the function into the rest of the world. An R function returns to the world the last statement that was evaluated but not saved into a variable. This is

⁵⁷Actually, a data frame *is* a list whose pieces all happen to be the same size.

⁵⁸I can't think why, but that's not my problem here!

⁵⁹The intention is to be clear.

⁶⁰Oh, I'm getting ahead of myself.

typically on the last line of the function.⁶¹ If this offends your programming sensitivities, R also has a `return` function, so my last line could just as well have been `return(a-b)`.

A consequence of this mechanism for output is that if you want to print out the value of something inside a function,⁶² you need to explicitly call on `print`.⁶³

If you're creating a function in R Studio, you type the lines into an R Script window. When you've finished, you copy all the lines right down to the closing curly bracket and run them (Control-Enter). There won't be any output: all you're doing is defining a function to use later.⁶⁴ If there are any errors, though, R will tell you about them, and you can go back and fix your function.

9.9.4 One-liners and curly brackets

A function is a kind of black box. Once you have it working, it doesn't matter what happens inside it: it's a machine for turning input into output.

I mentioned that the business end⁶⁵ of a function has to be enclosed in curly brackets. This is not strictly true; the exception is that if the function is only one line long, the curly brackets are optional. But I've written so many functions that started life as one-liners but ended up longer that I like to put the curly brackets in just to be safe.

If I really wanted, I would write the range function as a one-liner, like this:

```
R> my.r=function(x) max(x)-min(x)
```

I don't think it's quite so readable this way, though it's not bad. You'll see people straining to define functions as one-liners, as if curly brackets were \$100 a shot. What does this one do?

```
R> f=function(x,y) mean(resid(lm(y~x)))
```

Hint: work from the innermost brackets out.

9.9.5 Using a function you wrote

When you've successfully defined a function, you use it the same way as any other function that R provides. In fact, by writing a function, you have single-

⁶¹What else would you do after that?

⁶²Say, you want to print out the value of `b` to check that it really is what you thought it was.

⁶³In the example of the previous footnote, you'd have to say `print(b)`. Or maybe `print(c(a,b))` if you want to look at both variables.

⁶⁴I'm not calculating any ranges, yet. All I've done is to produce a machine for calculating ranges.

⁶⁵Officially called "body".

handedly extended what R can do! To exemplify, with our `my.range` function:

```
R> v=c(4,5,2,7)
R> my.range(v)
```

```
[1] 5
```

$7 - 2 = 5$. Or:

```
R> my.range(c(10,13,14))
```

```
[1] 4
```

Or even:

```
R> my.range(1)
```

```
[1] 0
```

Does that make sense?

9.9.6 Optional arguments

Functions can have optional arguments. These often need to be given a default value. This is done on the `function` line as shown below. Let's suppose we want to standardize a vector of values, but we want to have the option of passing in a mean and SD to standardize with. Here's how you might do that:

```
R> stand=function(x,calc=T,mean=0,std=1)
R> {
R>   if (calc)
R>     {
R>       mu=mean(x)
R>       sigma=sd(x)
R>       z=(x-mu)/sigma
R>     }
R>   else
R>     {
R>       z=(x-mean)/std
R>     }
R>   z
R> }
```

Here's how this goes: on the `function` line, the input parameters to the function (apart from `x`) all have defaults, meaning that specifying them is optional. If you don't, the default values will be used, exactly as if you had entered them yourself.

In this case, our options are `calc`, whether or not to calculate the mean and SD from the data (which is a logical value: either T or F, the former if we

don't say), and values of the mean and SD to use if we don't want to calculate them from the data. I gave these default values as well as a piece of "defensive programming": what if the user says `calc=F` but doesn't provide any mean and SD to standardize with?

The logic is this: first we check whether `calc` is `TRUE` or not. If it is, we calculate the mean and SD of the data, and use them to standardize with, saving the results in `z`. If not (that is, `calc` is `FALSE`), we use the input `mean` and `std` to calculate with, and get the standardized values that way, again saved in `z`. The final step is, whichever way we got `z`, to send it back to the outside world.

Here are some examples of the function in action:

```
R> v=c(2,3,6,8)
R> stand(v)

[1] -0.9986254 -0.6354889  0.4539206  1.1801937
```

Using all the defaults, so we standardize using mean and SD of the data.

```
R> stand(v,calc=T)

[1] -0.9986254 -0.6354889  0.4539206  1.1801937
```

Same thing. This time using a mean and SD we feed in:

```
R> stand(v,mean=6,std=2,calc=F)

[1] -2.0 -1.5  0.0  1.0
```

Notice that if you use the names, you can specify them in any order, but if you don't, you have to enter them in the order they appear:

```
R> stand(v,F,6,2)

[1] -2.0 -1.5  0.0  1.0
```

Same, but the value for `calc` had to come first. Look at this:

```
R> stand(5,mean=6,std=2,calc=F)

[1] -0.5
```

This shows that you can feed in either a vector of values or a single number, and the standardization works.⁶⁶ This is because the calculation in the function works just as well for a vector as it does for a number.

Here's why I did my defensive programming. What about this?

This makes no sense: why refuse to let R calculate the sample mean and SD and then not supply any values to calculate with? The *right* way to handle this is to have your function return an error, but that's something else to check for when you're writing it, and at this point makes things less clear. I decided to

⁶⁶Though if you feed in a single number, you have to give a mean and SD as well.

supply values for `mean` and `std` that are guaranteed to provide *some answer*⁶⁷ without giving an error.

9.9.7 Geometric distribution

You might recall the *binomial distribution*. This gives the probability for things like this: toss a coin 10 times and count the number X of heads.

Another way of looking at this kind of thing is to ask yourself “how many tosses might it take me to get my first head?”. Here, the binomial distribution doesn’t apply, because you no longer have a fixed number of coin tosses (“trials”, more generally). As long as we have Bernoulli trials⁶⁸, though, the distribution of “time to first success” is something we can work out.

Let’s suppose the probability of a success is p .⁶⁹ and let X be the number of tosses/trials altogether until your first success. X is a random variable. Well, you could get a success first time:

$$P(X = 1) = p$$

since there’s only one way to do it. To get your first success on the *second* trial, you have to fail first time, and then succeed:

$$P(X = 2) = p(1 - p)$$

To get your first success on the *third* trial, you have to fail the first two times:

$$P(X = 3) = p(1 - p)^2$$

and by now you get the idea: to see the first success after n trials, you have to observe $n - 1$ failures and then a success:

$$P(X = n) = p(1 - p)^{n-1}$$

Let’s see if we can implement this in R.⁷⁰ What do we need to feed in? The number of trials we want the probability of (let’s call that x), and the probability of any one trial being a success, which we’ll call p . The calculation itself is a one-liner, so there’s no need for any curly brackets. Unless you want them, of course. I do.

⁶⁷Actually, the input, unchanged. You can decide for yourself whether that’s a bug or a feature!

⁶⁸Independent trials with constant probability of success.

⁶⁹A success could be “heads”. We’re not assuming a fair coin.

⁷⁰Even though R already has a perfectly good geometric distribution calculator called `dgeom`.

```
R> geometric=function(x,p)
R> {
R>   p*(1-p)^(x-1)
R> }
```

Not much here. \wedge raises to a power, as on a spreadsheet.

Let's check this for $p = 0.5$, like tossing a fair coin:

```
R> geometric(1,0.5)
[1] 0.5
```

The chance of succeeding first time is the same as p , so that's good.

```
R> geometric(2,0.5)
[1] 0.25
```

To get your first success on the second time, you have to fail and then succeed, both of which have probability 0.5. If you test some more, you'll find that each probability is half of the one before, as it should be.

What happens if somebody feeds in a probability outside of $[0, 1]$, or a number of trials less than 1? The function dies with an error. Not very helpful. What we can do is to catch the error ourselves, before the function gives us a nonsense answer. For example:

```
R> geometric(0,0.5)
[1] 1
```

and

```
R> geometric(2,1.1)
[1] -0.11
```

neither of which make any sense. Let's check that the input is sane before we calculate anything. R provides a handy mechanism for this called `stopifnot`: you feed this function a string of logical conditions. If they're all true, it does nothing, but if any of them are false, it tells you which one and stops.

Implementing that looks as below. We have three things to check: that the number of trials x is at least 1, that p is no smaller than zero, and that p is no bigger than 1:

```
R> geometric=function(x,p)
R> {
R>   stopifnot(p>=0,p<=1,x>=1)
R>   p*(1-p)^(x-1)
R> }
```

Let's try it out:

```
R> geometric(2,0.5)
```

```
[1] 0.25
```

That's supposed to work, and does.

```
R> geometric(0,0.5)
```

```
Error: x >= 1 is not TRUE
```

```
R> geometric(2,1.1)
```

```
Error: p < 1 is not TRUE
```

These both fail, and `stopifnot` tells you why.

9.9.8 Functions calling other functions

Here's a question: what happens if you call `geometric` with a *vector* for `x` instead of a number? As ever, try it and see:

```
R> geometric(1:5,0.5)
```

```
[1] 0.50000 0.25000 0.12500 0.06250 0.03125
```

This gives the probabilities of the first success taking exactly 1, 2, 3, ... successes. This is kind of a nice freebie of R: because of the way vector arithmetic works, you can often get vector output from vector input without having to do any extra coding.⁷¹

If you look at the output above, you'll see that it gives you everything you need to figure out the probability that the first success comes in the first *five trials or less*. Just add up all those five probabilities:

```
R> sum(geometric(1:5,0.5))
```

```
[1] 0.96875
```

This is the cumulative distribution function of a geometric random variable. That might be a useful thing to have a function to do for us: the probability that the first success comes in `x` trials or less in a geometric distribution with success probability `p`. These two are the inputs:

```
R> c.geometric=function(x,p)
R> {
R>   probs=geometric(1:x,p)
R>   sum(probs)
R> }
```

⁷¹You can also feed in a vector for `p`, and you get back the probability of the first success taking so-many trials if the success probability is each of those values of `p`. But I can't think of a good use for that.

See how easy that was to write? Get a vector of geometric probabilities up to `x`, then add them up and return the answer. Does it work?

```
R> c.geometric(5,0.5)
```

```
[1] 0.96875
```

That's the answer we got before.

```
R> c.geometric(20,0.1)
```

```
[1] 0.8784233
```

When the success probability is smaller, it'll take you longer to get the first success, at least on average. This shows that there is an appreciable probability of having to wait longer than 20 trials for the first success, when p is as small as 0.1. Since R actually already has a geometric-distribution calculator, we can check our answers, but I had to read the R help first to make sure I was asking R to find the right thing:

```
R> pgeom(4,0.5)
```

```
[1] 0.96875
```

```
R> pgeom(19,0.1)
```

```
[1] 0.8784233
```

The answers are the same, but the numbers of trials are off by one. This is because R's `dgeom` (probability) and `pgeom` (cumulative probability) only count the numbers of failures *before* the first success, and not the final trial that actually *is* the first success. So R's functions have to be called with a number of trials that's one less than we had.

Now, `c.geometric` calls on `geometric`, so you might be wondering “what happens if I change the definition of `geometric`?” Say, you might put in some extra error-checking. The good news is that `c.geometric` will always use the *newest version of `geometric`*, without you having to do anything to keep things in sync.

Suppose you hadn't had the bright idea of feeding `geometric` a vector for `x`. How might you write `c.geometric` then? Well, all is not lost. Say you want to find the probability that the first success takes 5 trials or less, when the probability of success is 0.5? Here's what you can do:

- Find the probability that the first success occurs on the first trial exactly.
- Find the probability that the first success occurs on the second trial exactly.
- Find the probability that the first success occurs on the third trial exactly.
- Find the probability that the first success occurs on the fourth trial exactly.
- Find the probability that the first success occurs on the fifth trial exactly.

Then you add these all up.

The right way to handle a repetitive task like this is via a *loop*. We'll have a variable `total` that contains the total so far (it starts out at zero). Every time we find a probability (using `geometric`) we'll add it to the total. Then we return whatever the total is. Like this:

```
R> c2.geometric=function(x,p)
R> {
R>   total=0
R>   for (i in 1:x)
R>     {
R>       prob=geometric(i,p)
R>       total=total+prob
R>     }
R>   total
R> }
```

This isn't quite as slick as before, but it gets the job done. Checking:

```
R> c2.geometric(5,0.5)
[1] 0.96875
R> c2.geometric(20,0.1)
[1] 0.8784233
```

and these are the same as before.

9.9.9 Passing on arguments

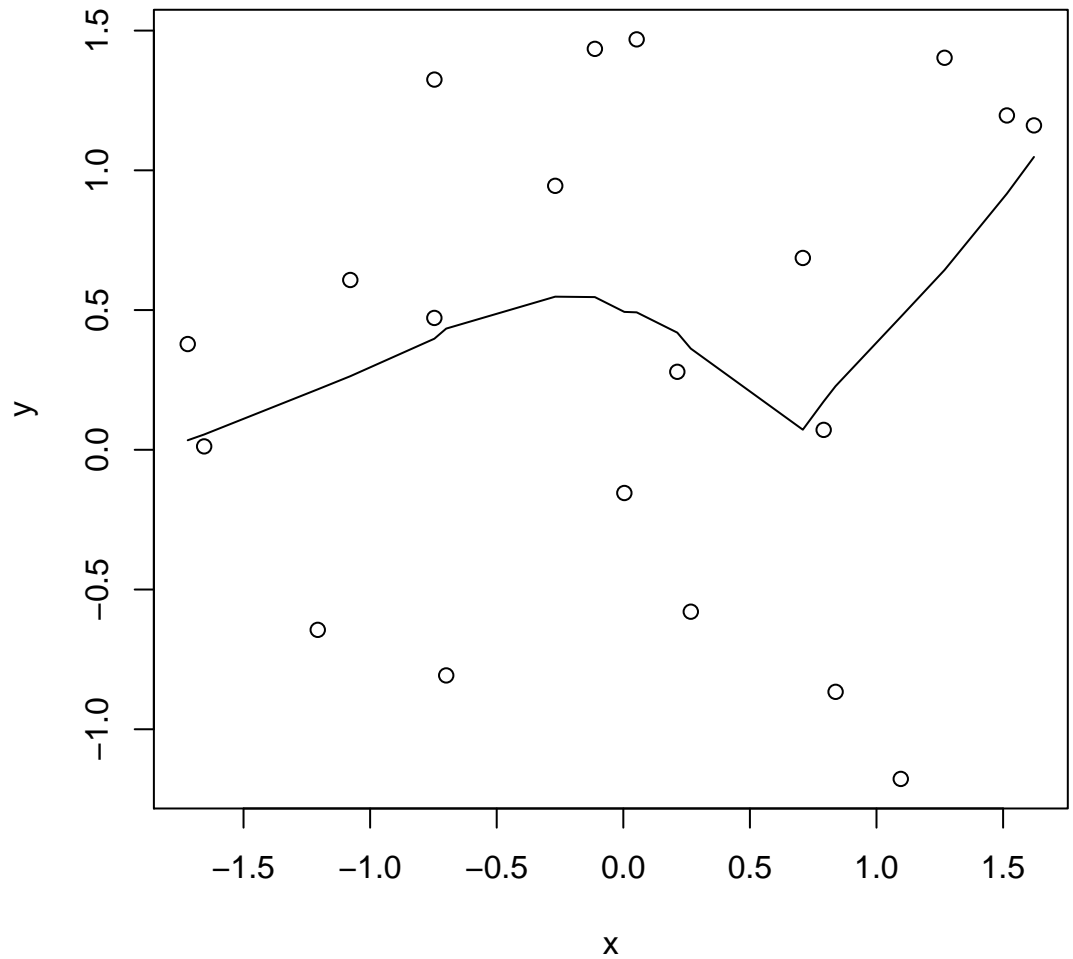
See Section 9.3.8, Section 9.3.12 and Section 9.3.14 for some background to this.

Let's take another example. This time we're going to make a function that draws a scatterplot for input variables `x` and `y`, and then puts a lowess curve on the plot. Here's the basic function (there's not much to it):

```
R> scatter=function(x,y)
R> {
R>   plot(x,y)
R>   lines(lowess(x,y))
R> }
```

Let's make some random data to test it on:

```
R> set.seed(457299)
R> x=rnorm(20)
R> y=rnorm(20)
R> scatter(x,y)
```



OK, that works. Now, the instinct of a programmer who has a working function is to fiddle with it. What would be a nice feature to add? How about making the colour, line type etc. of the lowess curve configurable?

The obvious thing to try is this:

```
R> scatter(x,y,col="green")
```

```
Error in scatter(x, y, col = "green") : unused argument (col = "green")
```

Oops. `scatter` is not expecting anything called `col`. If you think about it, it

was optimistic to hope that this would work, because there's no way the green colour can get through to the `lowess`.⁷²

One way to handle this is to have arguments `col`, `lty` etc. on `scatter`, with defaults.⁷³ But R has a more flexible mechanism, which saves us having to think about all the many graphical parameters we might want to pass to `lines(lowess())`. It looks like this:

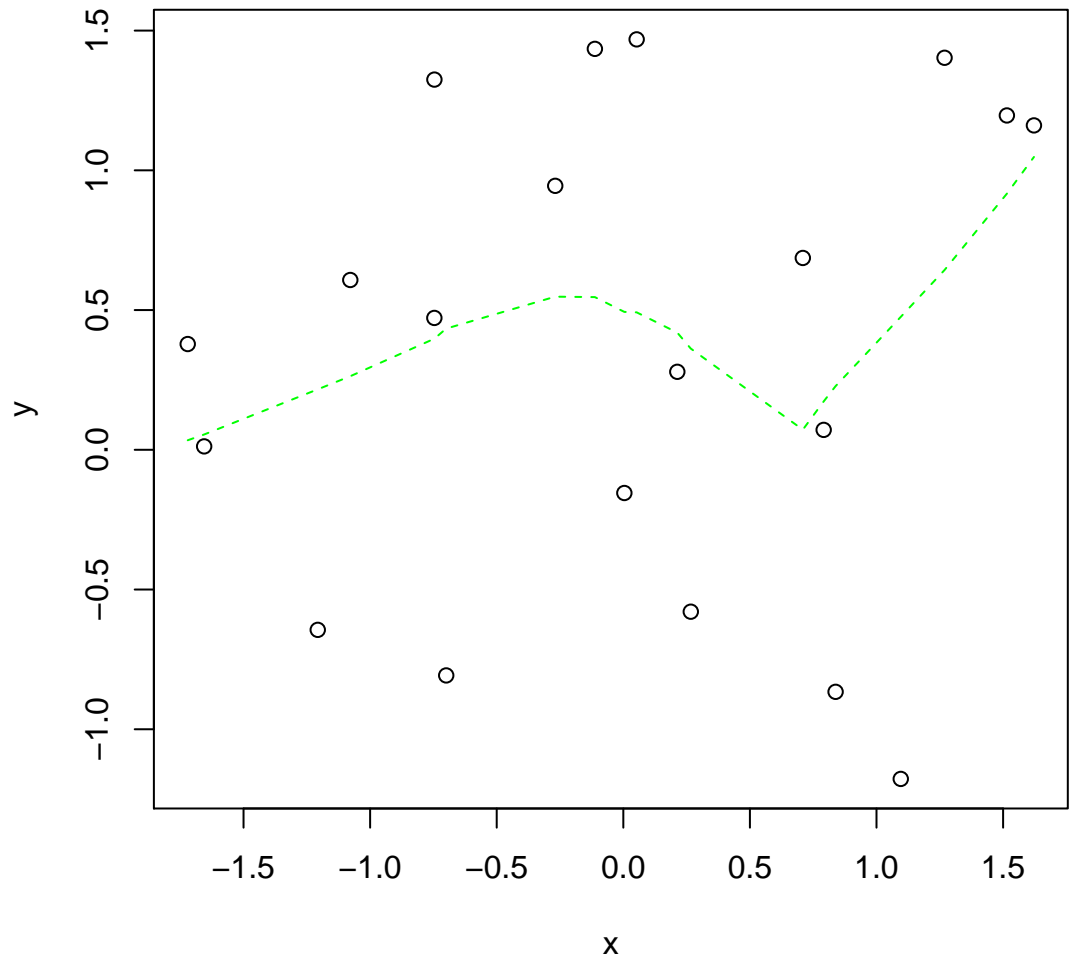
```
R> scatter2=function(x,y,...)
R> {
R>   plot(x,y)
R>   lines(lowess(x,y),...)
R> }
```

The `...` means “other arguments, unspecified”. The function `scatter2` doesn't even have to know what they will be. Anything that `scatter2` doesn't recognize as needing for itself will get passed on to `lines`. As long as `lines` knows what it means, everything is good. Here's how you might use it:

```
R> scatter2(x,y,col="green",lty="dashed")
```

⁷²Or was it the points on the plot that we wanted to make green?

⁷³`black` and `solid` respectively, presumably.



This produces a scatterplot with a green dashed lowess curve. The way `scatter2` is coded, the defaults on `plot` cannot be changed, so that the only configurable thing is the lines.

If you want to have the points be configurable as well, you can do this:

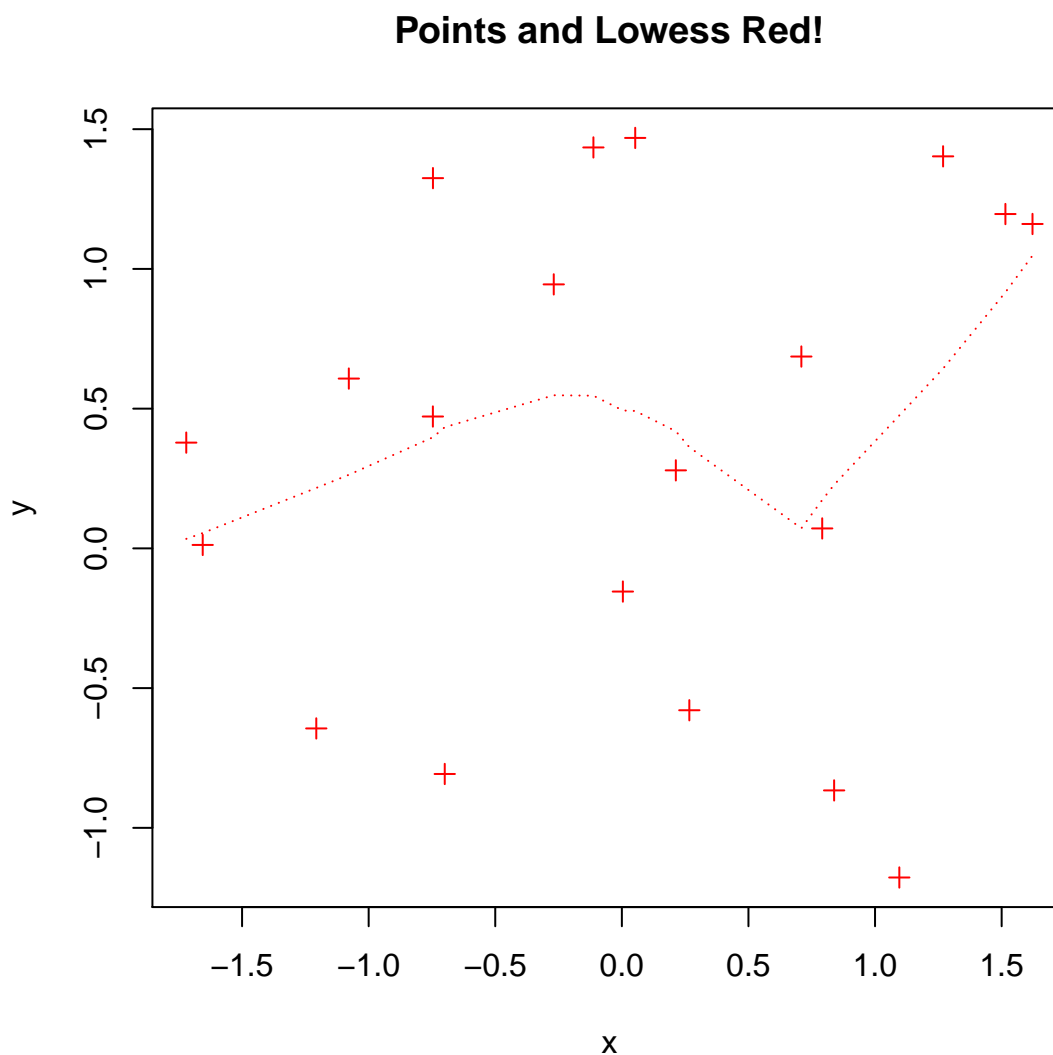
```
R> scatter3=function(x,y,...)
R> {
R>   plot(x,y,...)
R>   lines(lowess(x,y),...)
```



```
R> }
```

This time, the extra arguments get passed to both `plot` and `lines`. For example:

```
R> scatter3(x,y,col="red",lty="dotted",pch=3,main="Points and Lowess Red!")
```



Since `col` gets passed on to both `plot` and `lines`, *everything*⁷⁴ comes out red.

⁷⁴Well, not *everything*. The title didn't come out red, for example. That's because things like title and axis colours are governed not by `col` but by things like `col.main`. See Sec-

But things like `lty` that only make sense for lines are ignored by `plot`. We are now passing things into `plot`, so anything that `plot` can take, like `main` for a main title, is good.

On the other hand, if you pass in something that `plot` and `lines` don't know about, even though you don't get an error, you do get a bunch of warnings:

```
R> scatter3(x,y,col="red",hair="spiky")
```

Warning messages:

```
1: In plot.window(...) : "hair" is not a graphical parameter
2: In plot.xy(xy, type, ...) : "hair" is not a graphical parameter
3: In axis(side = side, at = at, labels = labels, ...) :
   "hair" is not a graphical parameter
4: In axis(side = side, at = at, labels = labels, ...) :
   "hair" is not a graphical parameter
5: In box(...) : "hair" is not a graphical parameter
6: In title(...) : "hair" is not a graphical parameter
7: In plot.xy(xy.coords(x, y), type = type, ...) :
   "hair" is not a graphical parameter
```

Chapter 10

Intricacies of SAS

10.1 Introduction

In R, we read our data in from a file, and do things with it and compute new variables as we go. The SAS philosophy is different; SAS splits things between “data stuff”, which lives in a `data` step, and “analysis stuff”, which lives in a `proc` step.

R is relatively restrictive in what it will allow you to read in from a file, so you might have to do some pre-processing to get things into the right format. This might involve using a spreadsheet or some other tool. SAS can read in things in almost any format; you just have to figure out how!

10.2 Reading stuff from a file

10.2.1 The basics

We saw the basic layout earlier. This file:

```
a 20
a 21
a 16
b 11
b 14
b 17
b 15
c 13
c 9
c 12
```

c 13

got read in using this code:

```
SAS> data groups;
SAS>   infile 'threegroups.dat';
SAS>   input group $ y;
SAS>
SAS> run;
```

This works with (a) one observation on each line, (b) separated by whitespace. The dollar sign marked the variable that was characters rather than numbers.

10.2.2 More than one observation on each line

Sometimes you have more than one observation on each line. Typically, this happens when you have only a small number of variables:

```
3 4 5 6 7 7
8 9 3 4
```

These are, let's say, all values of a variable *x*, in which case you'd read them like this:

```
SAS> data xonly;
SAS>   infile 'one.dat';
SAS>   input x @@;
SAS>
SAS> proc print;
```

| Obs | x |
|-----|---|
| 1 | 3 |
| 2 | 4 |
| 3 | 5 |
| 4 | 6 |
| 5 | 7 |
| 6 | 7 |
| 7 | 8 |
| 8 | 9 |
| 9 | 3 |
| 10 | 4 |

The @@ tells SAS to keep reading on the same line if there are any more values. What happens if you leave off the @@?

```
SAS> data xonly;
SAS>   infile 'one.dat';
SAS>   input x;
```

```
SAS>
SAS> proc print;
```

```
Obs      x
  1       3
  2       8
```

Only the first value on each line gets read into `x`.

The `@@` idea applies to more than one variable as well. Suppose now that the values in the file are a value of `x` followed by a value of `y`, followed by an `x` and a `y` and so on:

```
SAS> data xonly;
SAS>   infile 'one.dat';
SAS>   input x y @@;
SAS>
SAS> proc print;
```

```
Obs      x      y
  1       3      4
  2       5      6
  3       7      7
  4       8      9
  5       3      4
```

Next, some miscellaneous things for reading file data.

10.2.3 Skipping over header lines

Suppose you have a header line with names of variables and stuff, like this:

```
x y
3 4
5 6
7 7
8 9
3 4
```

Now, in R you read this with `header=T`. In SAS, your `input` line contains the names of your variables, so you need to skip over that line and start on line 2, like this:

```
SAS> data xy;
SAS>   infile 'two.dat' firstobs=2;
SAS>   input xx yy;
SAS>
SAS> proc print;
```

| Obs | xx | yy |
|-----|----|----|
| 1 | 3 | 4 |
| 2 | 5 | 6 |
| 3 | 7 | 7 |
| 4 | 8 | 9 |
| 5 | 3 | 4 |

Note that the `firstobs` goes on the `infile` line, since it's a property of the *file* that you want to start reading on line 2. Likewise, if you have something like a title and then some variable names, you might want to start reading data from line 3, and then `firstobs=3` is what you want. Also, the variable names that SAS uses are the ones in the `input` statement, not the ones in the file, which are completely ignored.

10.2.4 Delimited data

Sometimes your data come separated not by whitespace, but by something else. For example, one way of reading spreadsheet data into SAS is to save the values as a `.csv` file,¹ and then tell SAS that the values are separated by commas. So if the data file is like this:

```
3,4
5,6
7,7
8,9
3,4
```

this will read the values in:

```
SAS> data xy;
SAS>   infile 'three.dat' dlm=',';
SAS>   input x y;
SAS>
SAS> proc print;
```

| Obs | x | y |
|-----|---|---|
| 1 | 3 | 4 |
| 2 | 5 | 6 |
| 3 | 7 | 7 |
| 4 | 8 | 9 |
| 5 | 3 | 4 |

As another example, I made a list of 11 female singers in a spreadsheet, and saved this as a `.csv` file. Here's how I can read it in:

```
SAS> data singers;
SAS>   infile 'singers.csv' dlm=',';
```

¹This stands for “comme-separated values”.

```
SAS>    input number name $;
SAS>
SAS> proc print;
```

| Obs | number | name |
|-----|--------|----------|
| 1 | 1 | Bessie S |
| 2 | 2 | Peggy Le |
| 3 | 3 | Aretha F |
| 4 | 4 | Diana Ro |
| 5 | 5 | Dolly Pa |
| 6 | 6 | Tina Tur |
| 7 | 7 | Madonna |
| 8 | 8 | Mary J B |
| 9 | 9 | Salt n P |
| 10 | 10 | Aaliyah |
| 11 | 11 | Beyonce |

Almost right. The problem is that we only got the first 8 characters of each singer's name. The reason is that SAS reads only the first 8 characters of a text string by default. We fix that next.

10.2.5 Reading text

We can fix that by telling SAS to make these strings, let's say, 20 characters long, like this:

```
SAS> data singers;
SAS>    infile 'singers.csv' dlm=',';
SAS>    input number name $20.;
SAS>
SAS> proc print;
```

| Obs | number | name |
|-----|--------|-----------------|
| 1 | 1 | Bessie Smith |
| 2 | 2 | Peggy Lee |
| 3 | 3 | Aretha Franklin |
| 4 | 4 | Diana Ross |
| 5 | 5 | Dolly Parton |
| 6 | 6 | Tina Turner |
| 7 | 7 | Madonna |
| 8 | 8 | Mary J Blige |
| 9 | 9 | Salt n Pepa |
| 10 | 10 | Aaliyah |
| 11 | 11 | Beyonce |

The extra number and period are what is known to SAS as an **informat**. This tells SAS to read in this variable in a non-default way.

You'll notice that the singers' names have spaces in them. Some of the singers have just a single word in their name, and some have two or three. That doesn't matter to SAS, because the delimiter² is no longer a space but a comma. The only thing you have to beware of is commas *inside* the names, like **Robert Downey, Jr.**³ You can get around this by adding `dsd` to the `infile` line. I made another file of singers by adding Mr. Downey to the end, which I can read in like this:

```
SAS> data singers2;
SAS>   infile 'singers2.csv' dlm=',' dsd;
SAS>   input number name $20.;
SAS>
SAS> proc print;
```

| Obs | number | name |
|-----|--------|--------------------|
| 1 | 1 | Bessie Smith |
| 2 | 2 | Peggy Lee |
| 3 | 3 | Aretha Franklin |
| 4 | 4 | Diana Ross |
| 5 | 5 | Dolly Parton |
| 6 | 6 | Tina Turner |
| 7 | 7 | Madonna |
| 8 | 8 | Mary J Blige |
| 9 | 9 | Salt n Pepa |
| 10 | 10 | Aaliyah |
| 11 | 11 | Beyonce |
| 12 | 12 | Robert Downey, Jr. |

There is a gotcha here that might have grabbed you, if you tried this before reading this far. Your singer names have to be *at least 20 characters long* in the file. If they are not, you need to add enough spaces after them to make them 20 characters long. Let's find out what happens if you don't. I made a new file `singers3.csv` with all the extra spaces after Tina Turner and Aaliyah removed. Then I ran this code:

```
SAS> data singers3;
SAS>   infile 'singers3.csv' dlm=',' dsd;
SAS>   input number name $20.;
SAS>
SAS> proc print;
```

| Obs | number | name |
|-----|--------|-----------------|
| 1 | 1 | Bessie Smith |
| 2 | 2 | Peggy Lee |
| 3 | 3 | Aretha Franklin |
| 4 | 4 | Diana Ross |

²This is what `dlm` is short for.

³Who isn't female, but *is* a singer, having begun his career as an actor.

| | | |
|----|----|--------------------|
| 5 | 5 | Dolly Parton |
| 6 | 6 | 7, Madonna |
| 7 | 8 | Mary J Blige |
| 8 | 9 | Salt n Pepa |
| 9 | 10 | 11, Beyonce |
| 10 | 12 | Robert Downey, Jr. |

Hmm, odd.

Let's try to work out what happened. First off, the two short⁴ singers have disappeared. On the Tina Turner line, SAS correctly read number 6, but the name of the singer wasn't 20 characters long, so SAS skipped over the rest of this line and read the first 20 characters from the next line, which were 7, Madonna and a bunch of spaces. Then it failed to find a number on that line (*after* Madonna), so it went to the next line and started reading from there. The same happened on the line containing Aaliyah; SAS got the 20-character name it was looking for from the next line, 11, Beyonce plus spaces, and then grabbed a number 12 from the line after that.

The moral of this story is that it is *awfully* easy to fail in some way to get SAS to read your data in properly. So `proc print` is often a good idea, at least until you are sure that everything has worked. You ought to do at least some basic sanity checking, such as having the right number of lines⁵ and having sensible names.⁶

10.2.6 Windows and Linux/Unix file formats

I mention in passing that adding spaces is a cure for a lot of ills. One of the biggest apparently insoluble causes of trouble is when you copy a Windows file over to `cms-chorus`, which runs Linux. Windows and Unix/Linux have different conventions for how you determine the end of a line. So when you run SAS, on Linux, it gets very confused about where the line-ends are, and makes a right mess of reading the file.⁷ A quick fix for this is to go back to your data file, and add a space to the end of each line. Then SAS appears to be just fine about reading the data. I guess, by the time it gets to the space, SAS has gotten everything it wants from the line, so that the off-kilter end-of-line characters don't bother it so much.

⁴In name, not in stature.

⁵We knew that there were 12 singers in our data file, so having only 10 singers in the data set `singers3` should set the alarm bells ringing.

⁶Who is 7, Madonna anyway?

⁷You'll have most or all of your variables, but the values will be missing.

10.2.7 Another way to read spreadsheet data

Here's another way to read in spreadsheet data. You can copy a group of cells and paste it into the SAS program editor. From there, you save it as a file on Matlab, and then you use that as your `infile`. There are a couple of problems, one of which is when you copy spreadsheet cells, they get pasted in with *tabs* in between them.

To make this work, it seems to be better to rearrange the spreadsheet so that the names are first and the values second. I copied and pasted the data into the SAS Program Editor and saved it in `singsing.dat`. Then this works:

```
SAS> data sing;
SAS>   infile "singsing.dat" expandtabs;
SAS>   input singer $20. value;
SAS>
SAS> proc print;
```

| Obs | singer | value |
|-----|-----------------|-------|
| 1 | Bessie Smith | 1 |
| 2 | Peggy Lee | 2 |
| 3 | Aretha Franklin | 3 |
| 4 | Diana Ross | 4 |
| 5 | Dolly Parton | 5 |
| 6 | Tina Turner | 6 |
| 7 | Madonna | 7 |
| 8 | Mary J Blige | 8 |
| 9 | Salt n Pepa | 9 |
| 10 | Aaliyah | 10 |
| 11 | Beyonce | 11 |

The informat seems to be necessary, so that the whole names get read in. Otherwise you'll only get the first 8 characters and then SAS will get very confused about wanting a number and seeing more text.⁸ The `expandtabs` part gets around there being tabs in the data file (as a result of the values having been copied from a spreadsheet). This turns each tab into 8 literal spaces, so that the values are now delimited by spaces and we don't need any other special treatment.

10.2.8 Yet another way to read in a spreadsheet

There is another way of reading an Excel spreadsheet in "whole", without any copying, pasting, saving as `.csv` required. I saved my singers data into an `.xls` file⁹ and read it in like as below. Note that we are *not* using a data step this

⁸I tried 20 and it worked. You might not be so lucky. Be prepared to try different values at least as long as the longest name.

⁹I use LibreOffice, but that can save a spreadsheet as an `.xls` file too.

time. On the `proc import` line go four things: (a) a name for the data set, (b) the Excel file where the data live, which *needs to be transferred to Matlab first*, (c) the kind of thing it is (an Excel file), and (d) whether or not to overwrite an existing data set. Then we need to say which worksheet we want,¹⁰ and finally, whether the first row is variable names (like `header=T` in R). I broke up the `proc import` line since it was rather long. SAS doesn't care; it just goes looking for the semicolon.

Here we go:

```
SAS> proc import out=singers
SAS>          datafile='sing.xls'
SAS>          dbms=xls
SAS>          replace;
SAS>    sheet="sheet1";
SAS>    getnames=yes;
SAS>
SAS> proc print;
```

| Obs | singer | number |
|-----|-----------------|--------|
| 1 | Bessie Smith | 1 |
| 2 | Peggy Lee | 2 |
| 3 | Aretha Franklin | 3 |
| 4 | Diana Ross | 4 |
| 5 | Dolly Parton | 5 |
| 6 | Tina Turner | 6 |
| 7 | Madonna | 7 |
| 8 | Mary J Blige | 8 |
| 9 | Salt n Pepa | 9 |
| 10 | Aaliyah | 10 |
| 11 | Beyonce | 11 |

It worked!

10.3 Dates

I like this date format best. Think of the other column as some other variable measured on each date:

```
27may2013 857373
01apr2013 34757
03jan2013 38784
```

You read it into SAS this way, using an informat:

¹⁰I didn't re-name any of my worksheets, so it's the default `sheet1` here.

```

SAS> data mydates;
SAS>   infile 'dates.dat';
SAS>   input thedate date9. x;
SAS>
SAS> proc print;
SAS>
SAS> proc print;
SAS>   format thedate date9.;

```

| Obs | thedata | x |
|-----|---------|--------|
| 1 | 19505 | 857373 |
| 2 | 19449 | 34757 |
| 3 | 19361 | 38784 |

| Obs | thedata | x |
|-----|-----------|--------|
| 1 | 27MAY2013 | 857373 |
| 2 | 01APR2013 | 34757 |
| 3 | 03JAN2013 | 38784 |

Note the difference between the two `prints`: the first displays the dates in SAS-encoded form.¹¹ To display the dates as they were entered, you put a `format` with `proc print` with what you might call an “outformat”.¹²

There are other formats you might use for dates. Probably the commonest is `mm/dd/yyyy`, which is known to SAS as `mmddyy10`. The version with 8 replacing 10 has two-digit years, but you shouldn’t be using those.¹³ If you are not American, you might prefer the day first and then the month. You can use these both as formats for both input and output. Our dates look like this:

```

SAS> proc print;
SAS>   format thedate mmddyy10.;
SAS>
SAS> proc print;
SAS>   format thedate ddmmyy10.;

```

| Obs | thedata | x |
|-----|------------|--------|
| 1 | 05/27/2013 | 857373 |
| 2 | 04/01/2013 | 34757 |
| 3 | 01/03/2013 | 38784 |

| Obs | thedata | x |
|-----|------------|--------|
| 1 | 27/05/2013 | 857373 |
| 2 | 01/04/2013 | 34757 |
| 3 | 03/01/2013 | 38784 |

¹¹Days since Jan 1, 1960.

¹²SAS just calls this a “format”, which is kind of a dull name.

¹³Unless you are *sure* that none of your data will be from another century.

The point about these is that the dates are stored the same way *internally to SAS*; the right dates are “in there somewhere”. The different looks on the `proc print` are *only* because we are printing out the data in different ways. The underlying data are the same.

10.4 Creating permanent data sets and referring to data sets

You might find this business of data steps with `infile` and `input` statements to be rather tiresome. What about reading in the data once and then using the SAS dataset created after that?

Creating a permanent SAS data set is as simple as using a filename rather than just a plain name next to `data`. Here’s how we might make a permanent data set from the cars data. The car names have spaces in them, so I need to use an informat to read them in properly. The longest car name is 25 characters long. Also, in my file `cars.txt`, the first line is variable names:

```
SAS> options ls=70;
SAS> data 'cars';
SAS>   infile 'cars.txt' firstobs=2;
SAS>   input car $25. mpg weight cylinders hp country $;
SAS>
SAS> proc print;
```

| Obs | car | mpg | weight | cylinders | hp | country |
|-----|--------------------|------|--------|-----------|-----|---------|
| 1 | Buick Skylark | 28.4 | 2.670 | 4 | 90 | U.S. |
| 2 | Dodge Omni | 30.9 | 2.230 | 4 | 75 | U.S. |
| 3 | Mercury Zephyr | 20.8 | 3.070 | 6 | 85 | U.S. |
| 4 | Fiat Strada | 37.3 | 2.130 | 4 | 69 | Italy |
| 5 | Peugeot 694 SL | 16.2 | 3.410 | 6 | 133 | France |
| 6 | VW Rabbit | 31.9 | 1.925 | 4 | 71 | Germany |
| 7 | Plymouth Horizon | 34.2 | 2.200 | 4 | 70 | U.S. |
| 8 | Mazda GLC | 34.1 | 1.975 | 4 | 65 | Japan |
| 9 | Buick Estate Wagon | 16.9 | 4.360 | 8 | 155 | U.S. |
| 10 | Audi 5000 | 20.3 | 2.830 | 5 | 103 | Germany |
| 11 | Chevy Malibu Wagon | 19.2 | 3.605 | 8 | 125 | U.S. |
| 12 | Dodge Aspen | 18.6 | 3.620 | 6 | 110 | U.S. |
| 13 | VW Dasher | 30.5 | 2.190 | 4 | 78 | Germany |
| 14 | Ford Mustang 4 | 26.5 | 2.585 | 4 | 88 | U.S. |
| 15 | Dodge Colt | 35.1 | 1.915 | 4 | 80 | Japan |
| 16 | Datsun 810 | 22.0 | 2.815 | 6 | 97 | Japan |
| 17 | VW Scirocco | 31.5 | 1.990 | 4 | 71 | Germany |
| 18 | Chevy Citation | 28.8 | 2.595 | 6 | 115 | U.S. |
| 19 | Olds Omega | 26.8 | 2.700 | 6 | 115 | U.S. |

| | | | | | | |
|----|---------------------------|------|-------|---|-----|---------|
| 20 | Chrysler LeBaron Wagon | 18.5 | 3.940 | 8 | 150 | U.S. |
| 21 | Datsun 510 | 27.2 | 2.300 | 4 | 97 | Japan |
| 22 | AMC Concord D/L | 18.1 | 3.410 | 6 | 120 | U.S. |
| 23 | Buick Century Special | 20.6 | 3.380 | 6 | 105 | U.S. |
| 24 | Saab 99 GLE | 21.6 | 2.795 | 4 | 115 | Sweden |
| 25 | Datsun 210 | 31.8 | 2.020 | 4 | 65 | Japan |
| 26 | Ford LTD | 17.6 | 3.725 | 8 | 129 | U.S. |
| 27 | Volvo 240 GL | 17.0 | 3.140 | 6 | 125 | Sweden |
| 28 | Dodge St Regis | 18.2 | 3.830 | 8 | 135 | U.S. |
| 29 | Toyota Corona | 27.5 | 2.560 | 4 | 95 | Japan |
| 30 | Chevette | 30.0 | 2.155 | 4 | 68 | U.S. |
| 31 | Ford Mustang Ghia | 21.9 | 2.910 | 6 | 109 | U.S. |
| 32 | AMC Spirit | 27.4 | 2.670 | 4 | 80 | U.S. |
| 33 | Ford Country Squire Wagon | 15.5 | 4.054 | 8 | 142 | U.S. |
| 34 | BMW 320i | 21.5 | 2.600 | 4 | 110 | Germany |
| 35 | Pontiac Phoenix | 33.5 | 2.556 | 4 | 90 | U.S. |
| 36 | Honda Accord LX | 29.5 | 2.135 | 4 | 68 | Japan |
| 37 | Mercury Grand Marquis | 16.5 | 3.955 | 8 | 138 | U.S. |
| 38 | Chevy Caprice Classic | 17.0 | 3.840 | 8 | 130 | U.S. |

The only difference here is the quotes on `'cars'`. That will store a file improbably called `cars.sas7bdat`¹⁴ on Mathlab.¹⁵

Now you can do the conjuror's "there is nothing up my sleeve" trick by closing SAS and opening it again. That will break SAS's connection with any data sets you read in using `data` steps. To get `those` back, you'll have to run the `data` steps again.

Let's run `proc means` on the numerical variables in our `cars` data, like this. Note the single quotes around `cars`, and the utter absence of a `data` step:

```
SAS> proc means data='cars';
SAS>   var mpg weight cylinders hp;
```

The MEANS Procedure

| Variable | N | Mean | Std Dev | Minimum | Maximum |
|-----------|----|-------------|------------|------------|-------------|
| ----- | | | | | |
| mpg | 38 | 24.7605263 | 6.5473138 | 15.5000000 | 37.3000000 |
| weight | 38 | 2.8628947 | 0.7068704 | 1.9150000 | 4.3600000 |
| cylinders | 38 | 5.3947368 | 1.6030288 | 4.0000000 | 8.0000000 |
| hp | 38 | 101.7368421 | 26.4449292 | 65.0000000 | 155.0000000 |
| ----- | | | | | |

¹⁴To verify this, go back to the Putty/command window where you typed `sas&`, and type `ls`. You'll a list of all your files on Mathlab. `cars.sas7bdat` should be among them if you ran the commands above.

¹⁵You can store files in sub-folders as well, by using the appropriate folder convention for the system that SAS is running on. Mathlab runs Linux, so storing the data set in a folder `mydata` would be done like `data 'mydata/cars'`. If you were running SAS on Windows, you could do something like `data 'mydata\cars'` or supply the whole thing beginning with `c:\`.

Or, we can find the mean gas mileage by country. SAS makes this kind of thing easy (it is one of its strengths). No messing around with `aggregate`, just this:

```
SAS> proc means;
SAS>   var mpg;
SAS>   class country;
```

The MEANS Procedure

| Analysis Variable : mpg | | | | | |
|-------------------------|-----|----|------------|-----------|------------|
| country | N | | | | |
| | Obs | N | Mean | Std Dev | Minimum |
| France | 1 | 1 | 16.2000000 | . | 16.2000000 |
| Germany | 5 | 5 | 27.1400000 | 5.7348060 | 20.3000000 |
| Italy | 1 | 1 | 37.3000000 | . | 37.3000000 |
| Japan | 7 | 7 | 29.6000000 | 4.5328431 | 22.0000000 |
| Sweden | 2 | 2 | 19.3000000 | 3.2526912 | 17.0000000 |
| U.S. | 22 | 22 | 22.9954545 | 6.0542372 | 15.5000000 |

| Analysis Variable : mpg | | |
|-------------------------|-----|------------|
| country | N | |
| | Obs | Maximum |
| France | 1 | 16.2000000 |
| Germany | 5 | 31.9000000 |
| Italy | 1 | 37.3000000 |
| Japan | 7 | 35.1000000 |
| Sweden | 2 | 21.6000000 |
| U.S. | 22 | 34.2000000 |

The one Italian car has the highest “mean” mpg of all, followed by the seven Japanese cars.

10.5 How SAS knows which data set to use

You might have been wondering how SAS knows which data set to apply a `proc` to. There are two rules:¹⁶

1. Any `proc` can have a `data=` statement attached to it. That tells SAS to use that particular data set. It can be an unquoted data set name, which is one that you created earlier in this SAS session using a `data` step, or a quoted one like `'cars'`, which means to use the permanent data set in that file on disk.

¹⁶“There is *no* rule six!”

2. In the absence of a data statement, SAS will use the most recently created data set. This is typically a data set created by a `data` step, though it could also be a spreadsheet read in using `proc import`, as with the singers spreadsheet above.

The second rule is what has enabled us to avoid using `data=` statements so far. It corresponds to what we usually do: read in some data, then do some things with it. The most recently-created data set, which is the one we typically want to use, is the one we read in before. I discovered, with the calculations above for the mean `mpg` for the cars, that this extends to “most recently used”: I didn’t need to specify `data='cars'` a second time for the second `proc means`.¹⁷

Sometimes, though, there is more than one data set floating about. For example, some `procs` produce an output data set with an `out=` option. *This* one counts as the most recent one, technically, but it is a good idea to make sure you have the right one with a `data=` on the `proc` line. After all, somebody else might be reading your code, and you don’t want *them* to be confused!

10.6 Creating new data sets from old ones

10.6.1 Selecting individuals

Data sets can be large, and we might only want to look at part of the data. We can handle this a couple of ways: either when we read the data in in a `data` step, or by taking a subset of a SAS data set that already exists.

Let’s illustrate the first one with our singers. This was our original data step, with the `informat` because the singers’ names were long:

```
SAS> data singers;
SAS>   infile 'singers.csv' dlm=',';
SAS>   input number name $20.;
SAS>
SAS> proc print;
```

| Obs | number | name |
|-----|--------|-----------------|
| 1 | 1 | Bessie Smith |
| 2 | 2 | Peggy Lee |
| 3 | 3 | Aretha Franklin |
| 4 | 4 | Diana Ross |
| 5 | 5 | Dolly Parton |
| 6 | 6 | Tina Turner |
| 7 | 7 | Madonna |
| 8 | 8 | Mary J Blige |
| 9 | 9 | Salt n Pepa |

¹⁷I didn’t actually know that this would work, so I tried it to see, and it did.

| | | |
|----|----|---------|
| 10 | 10 | Aaliyah |
| 11 | 11 | Beyonce |

Let's say we only wanted to read in singers numbered 1 through 6. We handle that by putting an `if` line in the `data` step, like this:

```
SAS> data singers;
SAS>   infile 'singers.csv' dlm=',';
SAS>   input number name $20.;
SAS>   if number<=6;
SAS>
SAS> proc print;
```

| Obs | number | name |
|-----|--------|-----------------|
| 1 | 1 | Bessie Smith |
| 2 | 2 | Peggy Lee |
| 3 | 3 | Aretha Franklin |
| 4 | 4 | Diana Ross |
| 5 | 5 | Dolly Parton |
| 6 | 6 | Tina Turner |

Sometimes it's easier to think about the observations you want to leave out, which is done like this:

```
SAS> data singers;
SAS>   infile 'singers.csv' dlm=',';
SAS>   input number name $20.;
SAS>   if number>6 then delete;
SAS>
SAS> proc print;
```

| Obs | number | name |
|-----|--------|-----------------|
| 1 | 1 | Bessie Smith |
| 2 | 2 | Peggy Lee |
| 3 | 3 | Aretha Franklin |
| 4 | 4 | Diana Ross |
| 5 | 5 | Dolly Parton |
| 6 | 6 | Tina Turner |

You can also select on a text variable. "Less than" for these means "earlier alphabetically". This selects the singers before M in the alphabet:

```
SAS> data singers;
SAS>   infile 'singers.csv' dlm=',';
SAS>   input number name $20.;
SAS>   if name<'M';
SAS>
SAS> proc print;
```

| Obs | number | name |
|-----|--------|------|
|-----|--------|------|

| | | |
|---|----|-----------------|
| 1 | 1 | Bessie Smith |
| 2 | 3 | Aretha Franklin |
| 3 | 4 | Diana Ross |
| 4 | 5 | Dolly Parton |
| 5 | 10 | Aaliyah |
| 6 | 11 | Beyonce |

and this selects singer number 7:

```
SAS> data singers;
SAS>   infile 'singers.csv' dlm=',';
SAS>   input number name $20.;
SAS>   if number=7;
SAS>
SAS> proc print;
```

| Obs | number | name |
|-----|--------|---------|
| 1 | 7 | Madonna |

You can also use **and** and **or** to make complicated selections. Here's how you select those singers whose number is greater than 6, and whose name begins with something later than K:

***** but I can't make it work!

10.6.2 Selecting variables

The above **if** is used to select individuals. If you want to select *variables*, you need **drop** or **keep**. Let me illustrate:

```
SAS> data singers;
SAS>   infile 'singers.csv' dlm=',';
SAS>   input number name $20.;
SAS>   keep name;
SAS>
SAS> proc print;
```

| Obs | name |
|-----|-----------------|
| 1 | Bessie Smith |
| 2 | Peggy Lee |
| 3 | Aretha Franklin |
| 4 | Diana Ross |
| 5 | Dolly Parton |
| 6 | Tina Turner |
| 7 | Madonna |
| 8 | Mary J Blige |
| 9 | Salt n Pepa |

```

10    Aaliyah
11    Beyonce

SAS> data singers;
SAS>   infile 'singers.csv' dlm=',';
SAS>   input number name $20.;
SAS>   drop number;
SAS>
SAS> proc print;

Obs    name
  1    Bessie Smith
  2    Peggy Lee
  3    Aretha Franklin
  4    Diana Ross
  5    Dolly Parton
  6    Tina Turner
  7    Madonna
  8    Mary J Blige
  9    Salt n Pepa
 10    Aaliyah
 11    Beyonce

```

These both have the effect of making a data set with only `name`, either by specifying which variable(s) to `keep` or which ones to `drop`.

10.6.3 New data sets from old ones

What would be more interesting is to see the effect of `drop` and `keep` on a bigger data set. We'll demonstrate with the cars data set in a minute. But first, I want to explain how to create a new data set using (some of) the values in the old one.

Here's a pointless cloning of the singers data:

```

SAS> data singers;
SAS>   infile 'singers.csv' dlm=',';
SAS>   input number name $20.;
SAS>
SAS> data singers2;
SAS>   set singers;
SAS>
SAS> proc print data=singers2;

Obs    number    name
  1         1    Bessie Smith
  2         2    Peggy Lee
  3         3    Aretha Franklin

```

| | | |
|----|----|--------------|
| 4 | 4 | Diana Ross |
| 5 | 5 | Dolly Parton |
| 6 | 6 | Tina Turner |
| 7 | 7 | Madonna |
| 8 | 8 | Mary J Blige |
| 9 | 9 | Salt n Pepa |
| 10 | 10 | Aaliyah |
| 11 | 11 | Beyonce |

You wouldn't normally make a copy of a data set without doing anything else. The point of this is that you can use `if` and `keep` and `drop` to decide which variables you keep in your new data set. Let's have some fun with the cars:

```
SAS> data mycars;
SAS>   set 'cars';
SAS>   if mpg>30;
SAS>
SAS> proc print;
```

| Obs | car | mpg | weight | cylinders | hp | country |
|-----|------------------|------|--------|-----------|----|---------|
| 1 | Dodge Omni | 30.9 | 2.230 | 4 | 75 | U.S. |
| 2 | Fiat Strada | 37.3 | 2.130 | 4 | 69 | Italy |
| 3 | VW Rabbit | 31.9 | 1.925 | 4 | 71 | Germany |
| 4 | Plymouth Horizon | 34.2 | 2.200 | 4 | 70 | U.S. |
| 5 | Mazda GLC | 34.1 | 1.975 | 4 | 65 | Japan |
| 6 | VW Dasher | 30.5 | 2.190 | 4 | 78 | Germany |
| 7 | Dodge Colt | 35.1 | 1.915 | 4 | 80 | Japan |
| 8 | VW Scirocco | 31.5 | 1.990 | 4 | 71 | Germany |
| 9 | Datsun 210 | 31.8 | 2.020 | 4 | 65 | Japan |
| 10 | Pontiac Phoenix | 33.5 | 2.556 | 4 | 90 | U.S. |

This makes the new data set `mycars` contain just those cars whose gas mileage is over 30.

```
SAS> data mycars;
SAS>   set 'cars';
SAS>   keep car mpg;
SAS>
SAS> proc print;
```

| Obs | car | mpg |
|-----|------------------|------|
| 1 | Buick Skylark | 28.4 |
| 2 | Dodge Omni | 30.9 |
| 3 | Mercury Zephyr | 20.8 |
| 4 | Fiat Strada | 37.3 |
| 5 | Peugeot 694 SL | 16.2 |
| 6 | VW Rabbit | 31.9 |
| 7 | Plymouth Horizon | 34.2 |
| 8 | Mazda GLC | 34.1 |

| | | |
|----|---------------------------|------|
| 9 | Buick Estate Wagon | 16.9 |
| 10 | Audi 5000 | 20.3 |
| 11 | Chevy Malibu Wagon | 19.2 |
| 12 | Dodge Aspen | 18.6 |
| 13 | VW Dasher | 30.5 |
| 14 | Ford Mustang 4 | 26.5 |
| 15 | Dodge Colt | 35.1 |
| 16 | Datsun 810 | 22.0 |
| 17 | VW Scirocco | 31.5 |
| 18 | Chevy Citation | 28.8 |
| 19 | Olds Omega | 26.8 |
| 20 | Chrysler LeBaron Wagon | 18.5 |
| 21 | Datsun 510 | 27.2 |
| 22 | AMC Concord D/L | 18.1 |
| 23 | Buick Century Special | 20.6 |
| 24 | Saab 99 GLE | 21.6 |
| 25 | Datsun 210 | 31.8 |
| 26 | Ford LTD | 17.6 |
| 27 | Volvo 240 GL | 17.0 |
| 28 | Dodge St Regis | 18.2 |
| 29 | Toyota Corona | 27.5 |
| 30 | Chevette | 30.0 |
| 31 | Ford Mustang Ghia | 21.9 |
| 32 | AMC Spirit | 27.4 |
| 33 | Ford Country Squire Wagon | 15.5 |
| 34 | BMW 320i | 21.5 |
| 35 | Pontiac Phoenix | 33.5 |
| 36 | Honda Accord LX | 29.5 |
| 37 | Mercury Grand Marquis | 16.5 |
| 38 | Chevy Caprice Classic | 17.0 |

This keeps just the variables `car` and `mpg` and drops the rest. Or we can drop a couple of variables and keep the rest:

```
SAS> data mycars;
SAS>   set 'cars';
SAS>   drop cylinders hp;
SAS>
SAS> proc print;
```

| Obs | car | mpg | weight | country |
|-----|----------------|------|--------|---------|
| 1 | Buick Skylark | 28.4 | 2.670 | U.S. |
| 2 | Dodge Omni | 30.9 | 2.230 | U.S. |
| 3 | Mercury Zephyr | 20.8 | 3.070 | U.S. |
| 4 | Fiat Strada | 37.3 | 2.130 | Italy |
| 5 | Peugeot 694 SL | 16.2 | 3.410 | France |
| 6 | VW Rabbit | 31.9 | 1.925 | Germany |

| | | | | |
|----|---------------------------|------|-------|---------|
| 7 | Plymouth Horizon | 34.2 | 2.200 | U.S. |
| 8 | Mazda GLC | 34.1 | 1.975 | Japan |
| 9 | Buick Estate Wagon | 16.9 | 4.360 | U.S. |
| 10 | Audi 5000 | 20.3 | 2.830 | Germany |
| 11 | Chevy Malibu Wagon | 19.2 | 3.605 | U.S. |
| 12 | Dodge Aspen | 18.6 | 3.620 | U.S. |
| 13 | VW Dasher | 30.5 | 2.190 | Germany |
| 14 | Ford Mustang 4 | 26.5 | 2.585 | U.S. |
| 15 | Dodge Colt | 35.1 | 1.915 | Japan |
| 16 | Datsun 810 | 22.0 | 2.815 | Japan |
| 17 | VW Scirocco | 31.5 | 1.990 | Germany |
| 18 | Chevy Citation | 28.8 | 2.595 | U.S. |
| 19 | Olds Omega | 26.8 | 2.700 | U.S. |
| 20 | Chrysler LeBaron Wagon | 18.5 | 3.940 | U.S. |
| 21 | Datsun 510 | 27.2 | 2.300 | Japan |
| 22 | AMC Concord D/L | 18.1 | 3.410 | U.S. |
| 23 | Buick Century Special | 20.6 | 3.380 | U.S. |
| 24 | Saab 99 GLE | 21.6 | 2.795 | Sweden |
| 25 | Datsun 210 | 31.8 | 2.020 | Japan |
| 26 | Ford LTD | 17.6 | 3.725 | U.S. |
| 27 | Volvo 240 GL | 17.0 | 3.140 | Sweden |
| 28 | Dodge St Regis | 18.2 | 3.830 | U.S. |
| 29 | Toyota Corona | 27.5 | 2.560 | Japan |
| 30 | Chevette | 30.0 | 2.155 | U.S. |
| 31 | Ford Mustang Ghia | 21.9 | 2.910 | U.S. |
| 32 | AMC Spirit | 27.4 | 2.670 | U.S. |
| 33 | Ford Country Squire Wagon | 15.5 | 4.054 | U.S. |
| 34 | BMW 320i | 21.5 | 2.600 | Germany |
| 35 | Pontiac Phoenix | 33.5 | 2.556 | U.S. |
| 36 | Honda Accord LX | 29.5 | 2.135 | Japan |
| 37 | Mercury Grand Marquis | 16.5 | 3.955 | U.S. |
| 38 | Chevy Caprice Classic | 17.0 | 3.840 | U.S. |

and you see that the resulting data set has everything *except* cylinders and hp.

`keep` implies that the variables not mentioned are automatically dropped; `drop` implies that the variables not mentioned are automatically kept. So you don't need both. But you could have both an `if` and a `keep` or `drop`:

```
SAS> data mycars;
SAS>   set 'cars';
SAS>   keep car mpg;
SAS>   if weight>4;
SAS>
SAS> proc print;
```

```
Obs          car          mpg
```

```

1      Buick Estate Wagon      16.9
2      Ford Country Squire Wagon 15.5

```

You can see that we've trimmed things down a lot.

10.7 Plots

10.7.1 Histograms, normal quantile plots and boxplots

There are a number of plots you might want to make. SAS can handle them. Unlike R though, `proc plot` *only* does scatterplots; if you want something else, you have to find the right `proc` for it. We'll illustrate with the cars data, because it's handy.

Let's start with histograms, normal quantile plots and boxplots. Histograms are made by running `proc univariate` on the variable(s) you want, and asking for histograms of them, like this for `mpg`:

```

SAS> proc univariate data='cars';
SAS>   var mpg;
SAS>   histogram;

```

The UNIVARIATE Procedure

Variable: mpg

| | | Moments | |
|-----------------|------------|------------------|------------|
| N | 38 | Sum Weights | 38 |
| Mean | 24.7605263 | Sum Observations | 940.9 |
| Std Deviation | 6.54731385 | Variance | 42.8673186 |
| Skewness | 0.19678199 | Kurtosis | -1.3715426 |
| Uncorrected SS | 24883.27 | Corrected SS | 1586.09079 |
| Coeff Variation | 26.4425472 | Std Error Mean | 1.06211456 |

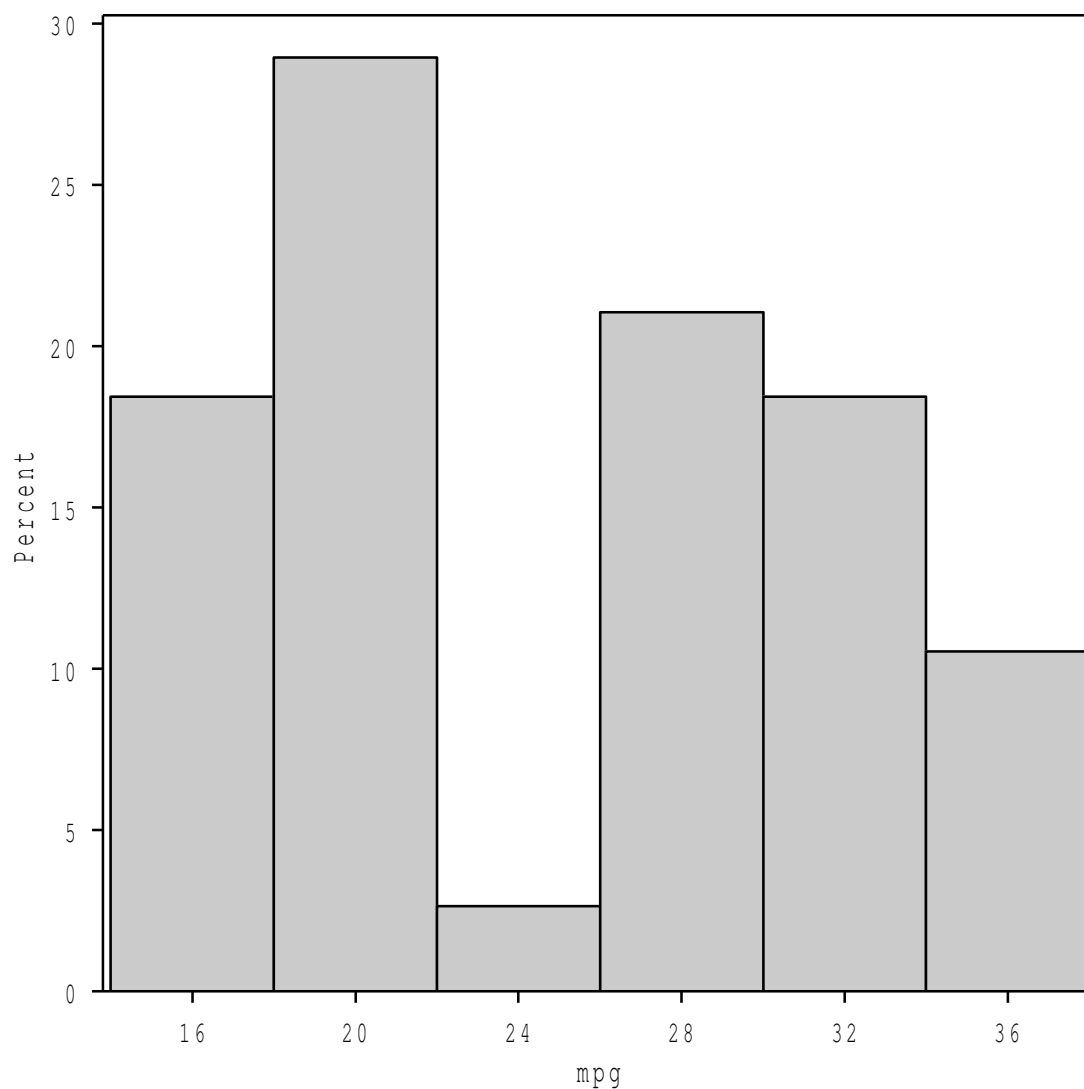
| Basic Statistical Measures | | | |
|----------------------------|----------|---------------------|----------|
| Location | | Variability | |
| Mean | 24.76053 | Std Deviation | 6.54731 |
| Median | 24.25000 | Variance | 42.86732 |
| Mode | 17.00000 | Range | 21.80000 |
| | | Interquartile Range | 12.00000 |

| Tests for Location: Mu0=0 | | | |
|---------------------------|-------------|-------------------|--------|
| Test | -Statistic- | -----p Value----- | |
| Student's t | t 23.31248 | Pr > t | <.0001 |
| Sign | M 19 | Pr >= M | <.0001 |
| Signed Rank | S 370.5 | Pr >= S | <.0001 |

Quantiles (Definition 5)

| Quantile | Estimate |
|------------|----------|
| 100% Max | 37.30 |
| 99% | 37.30 |
| 95% | 35.10 |
| 90% | 34.10 |
| 75% Q3 | 30.50 |
| 50% Median | 24.25 |
| 25% Q1 | 18.50 |
| 10% | 16.90 |
| 5% | 16.20 |
| 1% | 15.50 |
| 0% Min | 15.50 |

| Extreme Observations | | | |
|----------------------|-----|----------------|-----|
| ----Lowest---- | | ----Highest--- | |
| Value | Obs | Value | Obs |
| 15.5 | 33 | 33.5 | 35 |
| 16.2 | 5 | 34.1 | 8 |
| 16.5 | 37 | 34.2 | 7 |
| 16.9 | 9 | 35.1 | 15 |
| 17.0 | 38 | 37.3 | 4 |



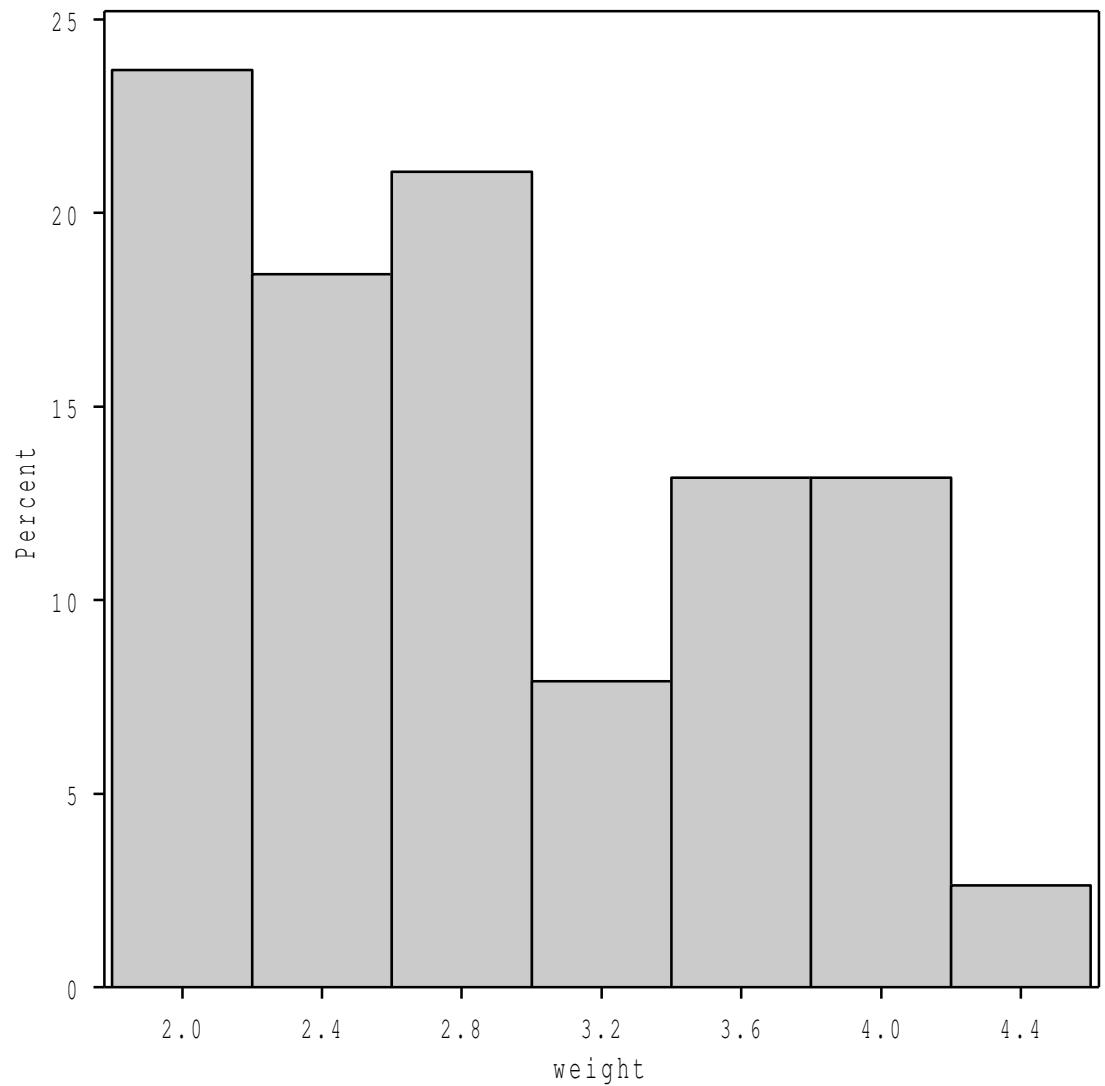
As usual for SAS, you get a whole bunch of output you don't really want (the numerical output from `proc univariate`), then the histograms. If you got the HTML output working, these will be shown after the numerical data for the variable concerned. If not, you'll get a separate plot window, *one at a time*, for each histogram, and you'll have to get rid of the first one before you'll see the second one.

In this case, we see the “gap” on the `mpg` histogram between the low-mpg cars and the high-mpg ones.

Putting an option `noprint` gets rid of the printed output. I'll do **weight** this

time:

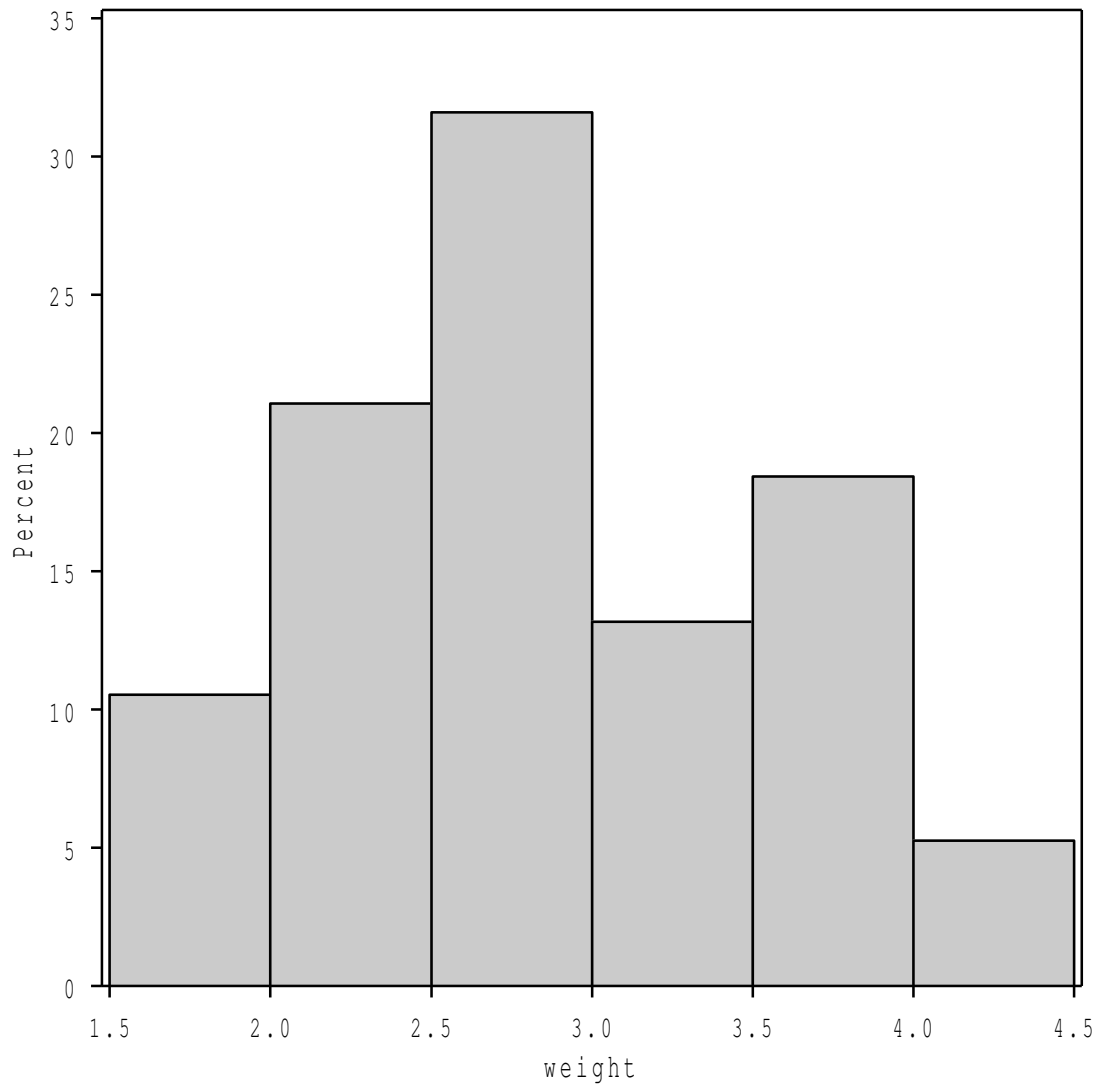
```
SAS> proc univariate data='cars' noprint;  
SAS>   var weight;  
SAS>   histogram;
```



The **weight** histogram also has a smaller bar in it (at 3.2 tons), but not as pronounced a “gap” as the **mpg** histogram does.

We can change the intervals for the histogram by supplying an option on **histogram**:

```
SAS> proc univariate data='cars' noprint;  
SAS>   var weight;  
SAS>   histogram / endpoints=1.5 2.0 2.5 3.0 3.5 4.0 4.5;
```

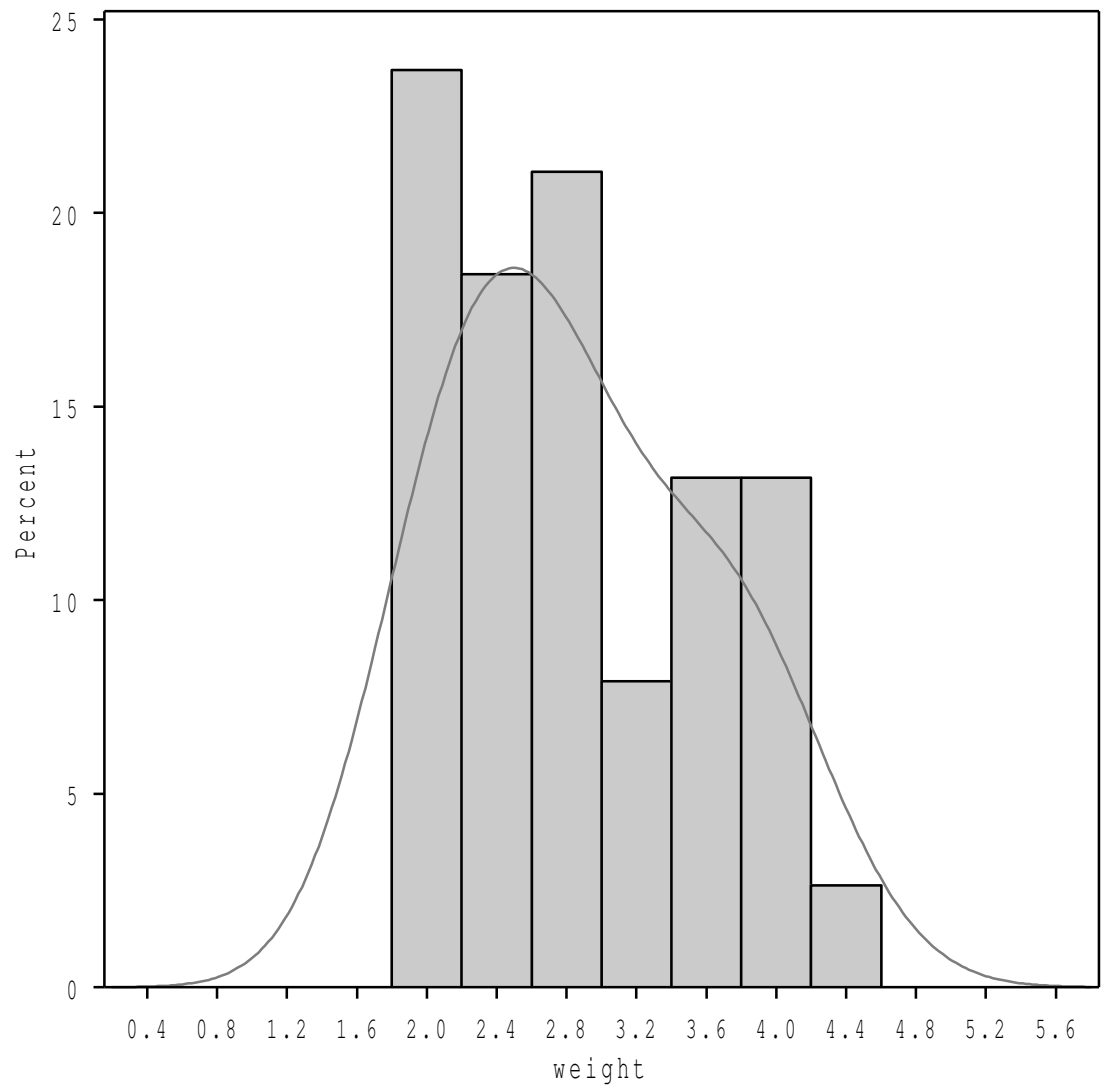


Or you can use `midpoint` to *centre* the bars on the values you give. The previous histogram had `midpoints=2.0 2.4 2.8 3.2 3.6 4.0 4.4`, only we didn't need to say that since it was the default.

You can plot things like a normal curve overlaid on the plot. I'm going to show you something else. It's called a **kernel density estimator**. It's a way of smoothing out the bumps of the scatterplot so that you see what the shape is.

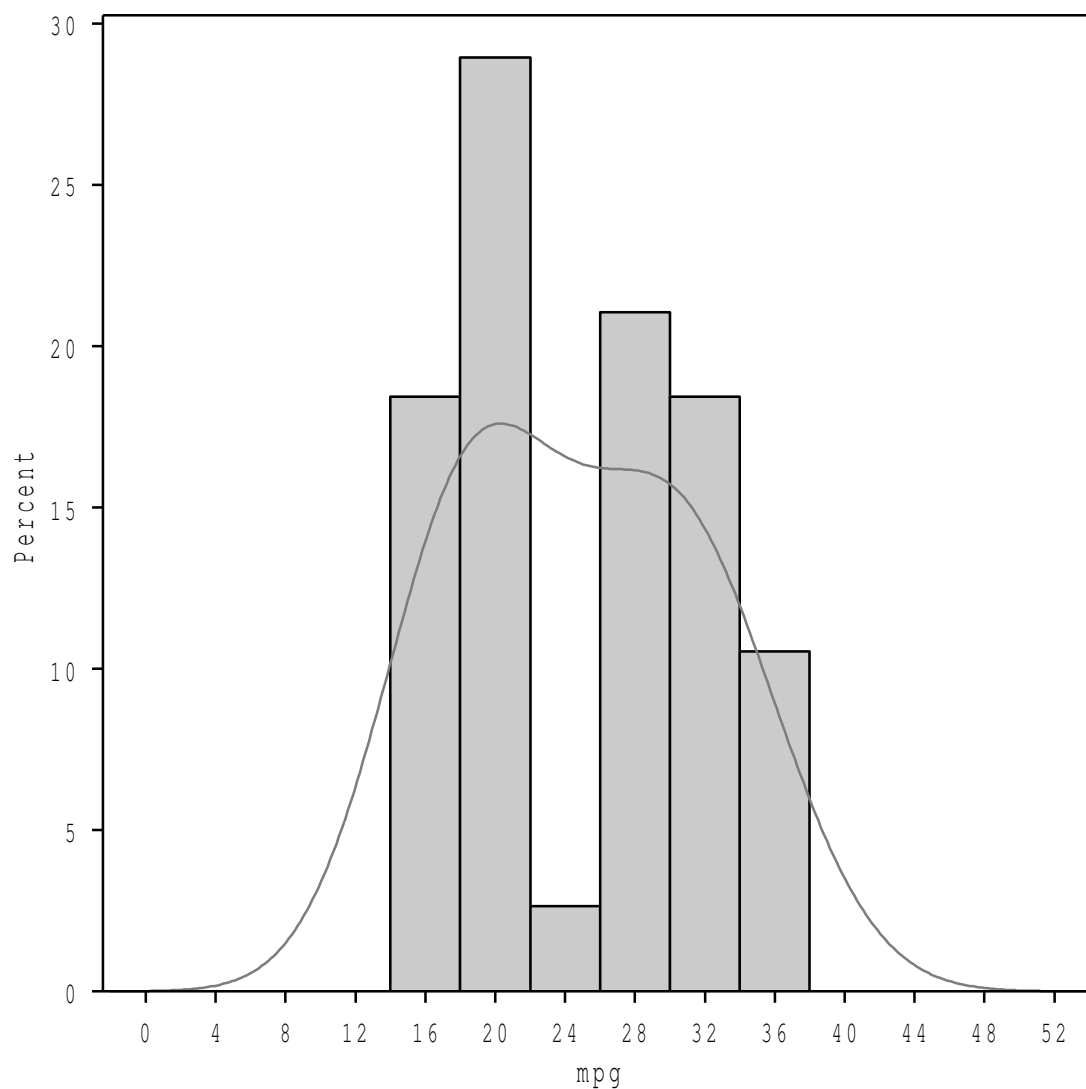
Here's how you get it:

```
SAS> proc univariate data='cars' noprint;  
SAS>   var weight;  
SAS>   histogram / kernel;
```



Here's the same thing for the mpg's:

```
SAS> proc univariate data='cars' noprint;  
SAS>   var mpg;  
SAS>   histogram / kernel;
```



If you want a normal curve overlaid instead, replace `kernel` by `normal`.

You can control the amount of smoothing the kernel density estimation uses, but we used the default here. More smoothing means that you might miss a peak that is important; less smoothing produces a rather bumpy “smooth”, making it difficult to see what the overall pattern is.

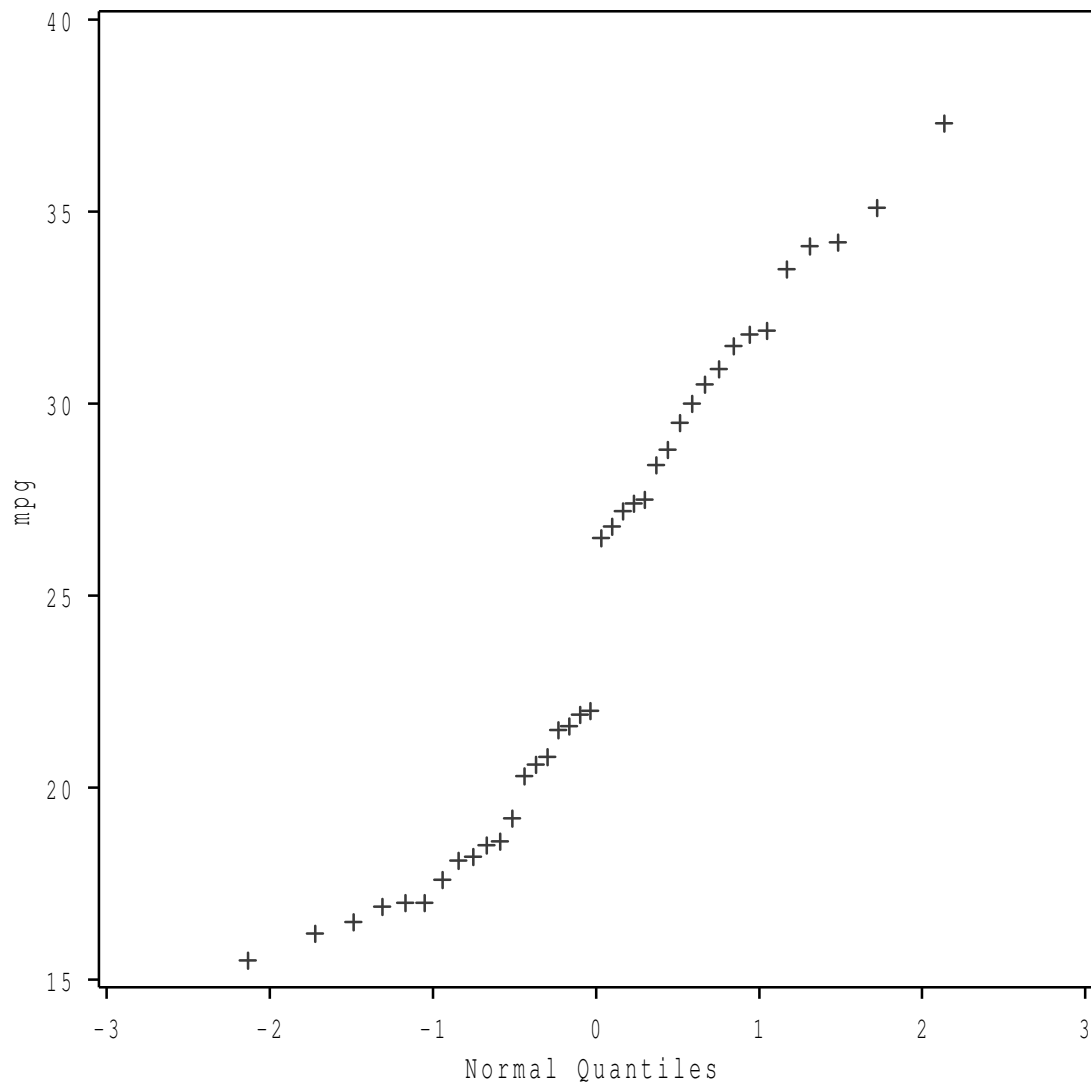
In this case, the SAS smooth curves are not very impressed by the small bars; they are taking the point of view that these could be chance. The one for `mpg`

actually looks quite normal¹⁸, while the one for `weight` is a bit skewed to the right.

The best way to assess normality, though, is not to look at a histogram but a normal quantile plot. This is accomplished the same kind of way as a histogram: run `proc univariate` and ask for it. The `mpg` figures looked not so far from normal, apart from a possible hole in the middle:

```
SAS> proc univariate data='cars' noprint;  
SAS>   var mpg;  
SAS>   qqplot;
```

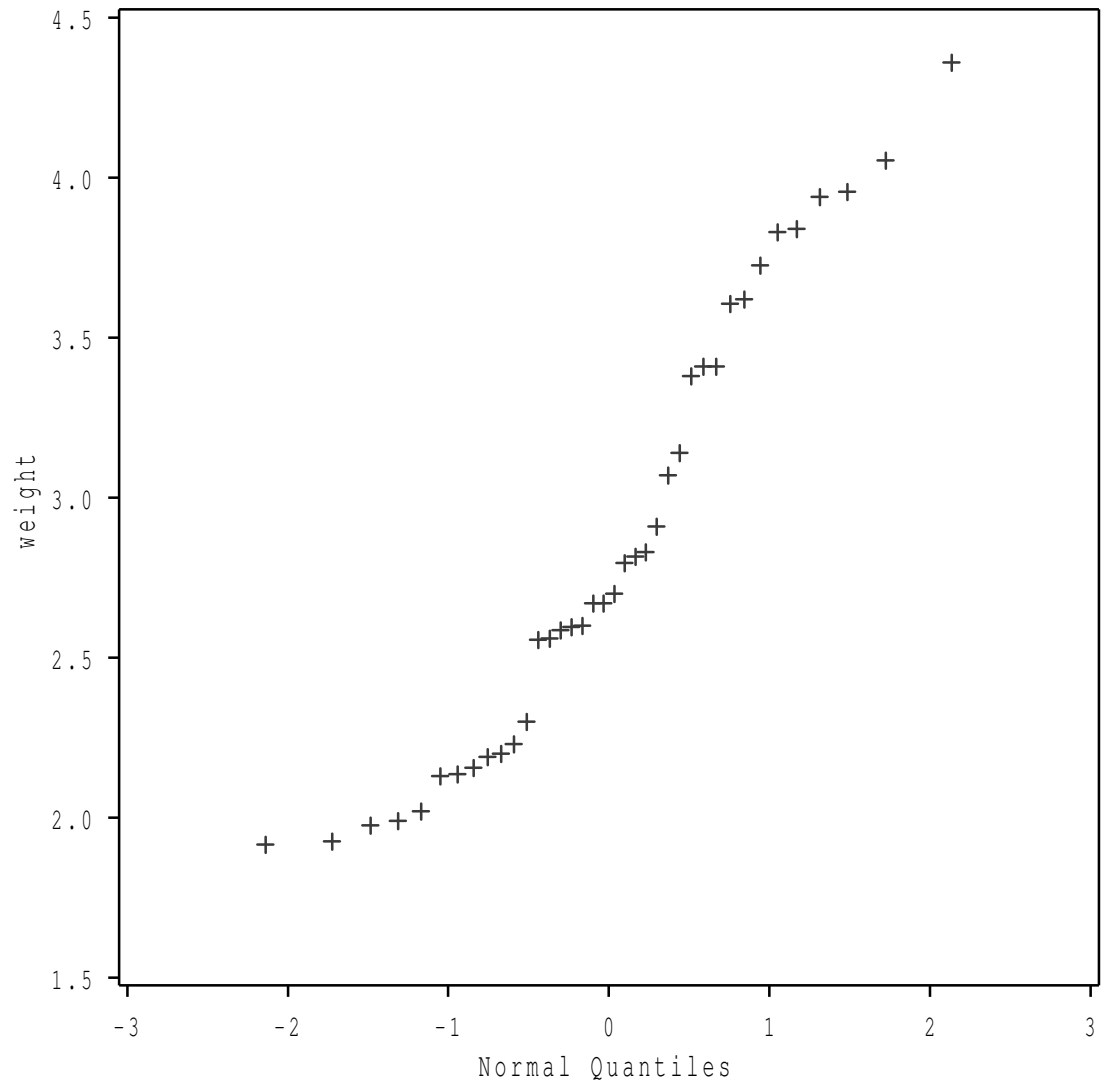
¹⁸With a bit of a flat top.



The hole in the middle shows up rather more clearly here as a break in the plot. We are looking for the points to form a more or less straight line, which they don't (largely because of the break in the middle). The higher values are more or less straight, but the lower ones are definitely curved. This kind of shows up on the histogram too, but it's hard to be sure, because there aren't enough bars on the histogram, and *that* is because there aren't enough values in the data set to justify using more.

What about the weights? The histogram looked a bit skewed to the right, so the normal quantile plot ought to show a curve:

```
SAS> proc univariate data='cars' noprint;  
SAS>   var weight;  
SAS>   qqplot;
```

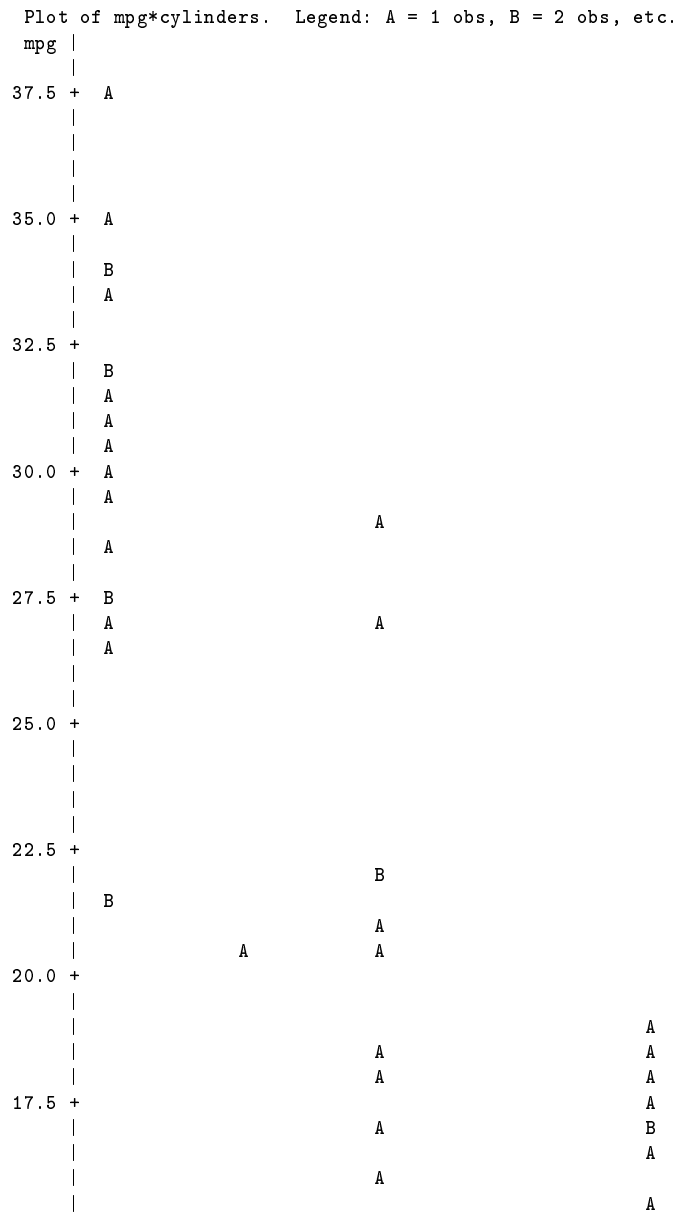


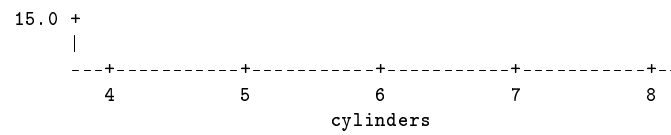
I see a curve here, especially at the lower end. If you look on the vertical (**weight**) axis, you'll see that the lowest values don't go down as far as you'd expect for a normal. This would suggest skewed to the right, as the histogram does. But the normal quantile plot almost curves back the other way, suggesting that the top values, except for the very highest one, are too bunched up for a normal. Not clear. The weights are not normal, but describing *how* they're not

normal is more of a challenge.

Here's a question: how does gas mileage vary with the number of cylinders? There are a couple of ways of handling this. Either you treat **cylinders** as a numerical variable, in which case a scatter plot is the way to go:

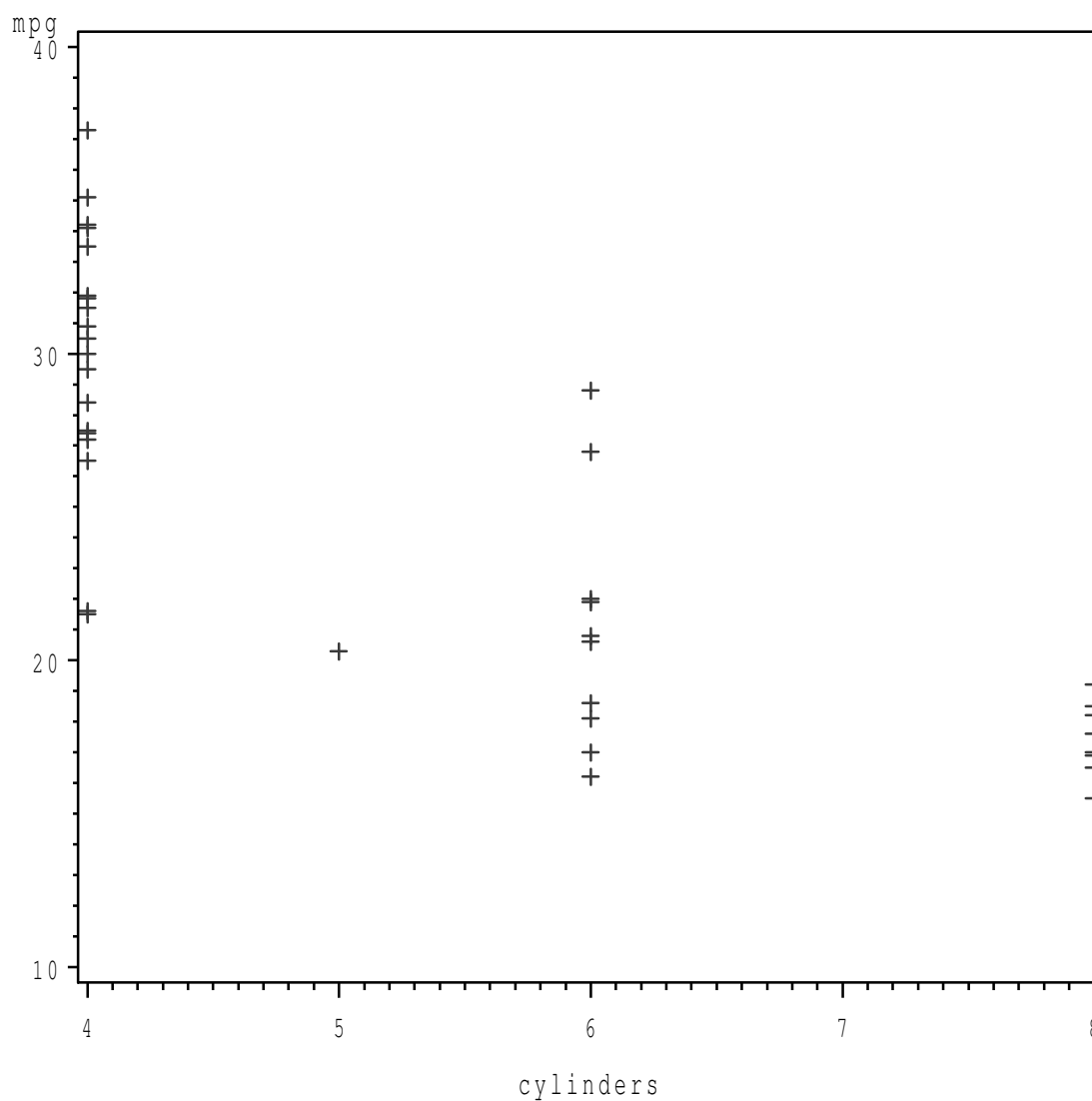
```
SAS> proc plot data='cars';
SAS>   plot mpg*cylinders;
```





This isn't very pretty. A means a single car, B means two cars that plotted in the same place. This is an example of the old-fashioned "line-printer graphics". In the old days, for graphics, this is all there was. But now, we can do better by adding a "g":

```
SAS> proc gplot data='cars';  
SAS>   plot mpg*cylinders;
```

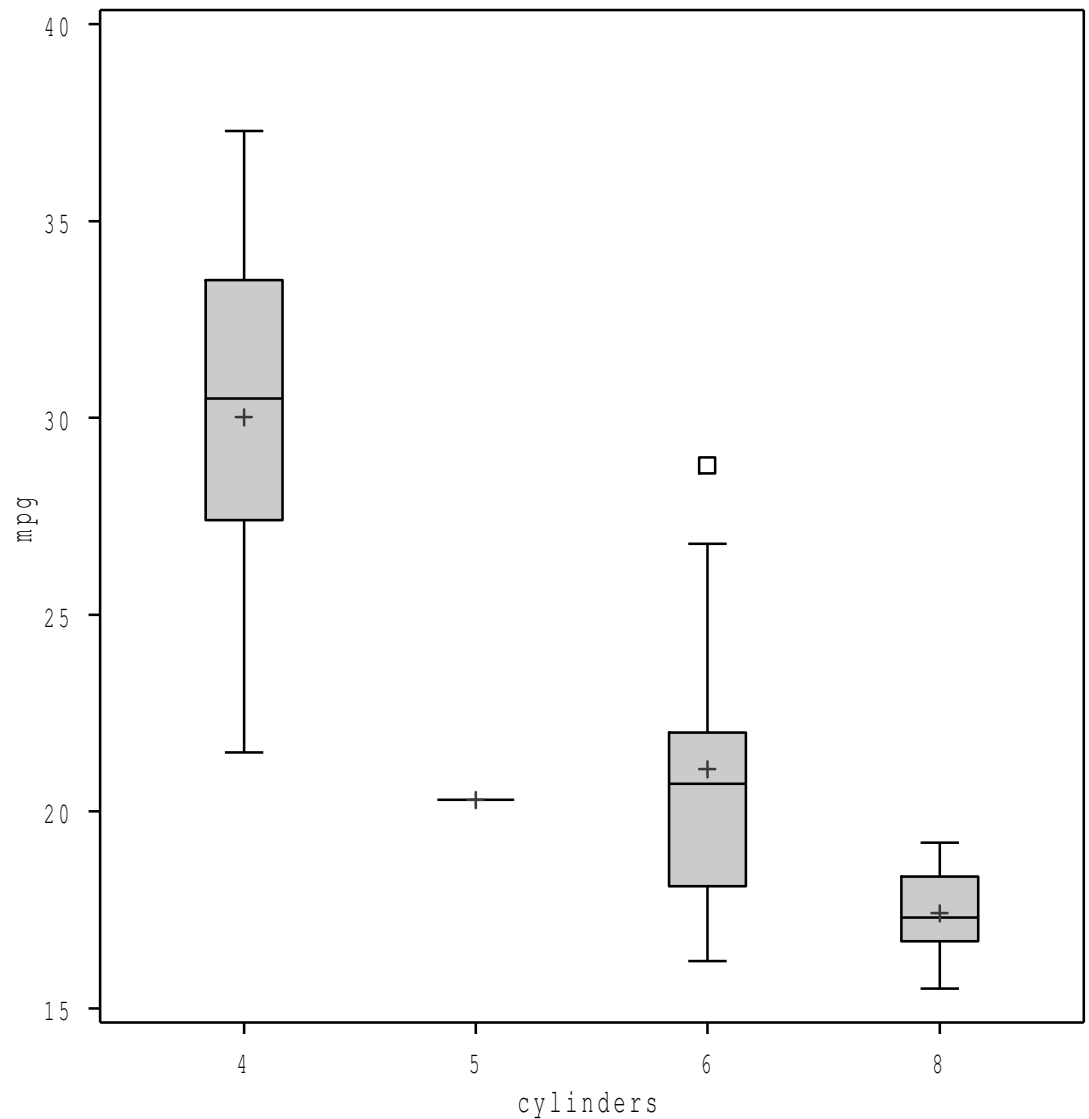


Much better. This shows us that gas mileage tends to come down as the number of cylinders goes up.¹⁹

The other way of handling this is to make side-by-side boxplots, treating `cylinders` as a categorical variable. The code for this actually looks alarmingly similar, but before we get to that, SAS requires us to have the `cylinders` values in order, so we have to `sort` the data first. Also, the purpose of the `boxstyle` option is to get the boxplot we know, with outliers plotted separately.

¹⁹Rather as we'd expect.

```
SAS> proc sort data='cars' out=sorted;  
SAS>   by cylinders;  
SAS>  
SAS> proc boxplot data=sorted;  
SAS>   plot mpg*cylinders / boxstyle=schematic;
```



This is identical to the previous boxplot.

What we see is that average mpg comes down as the number of cylinders increases, but there is a fair bit of variability, as shown by the overlapping of the boxplots. The variability is quite a bit less for the 8-cylinder engines.

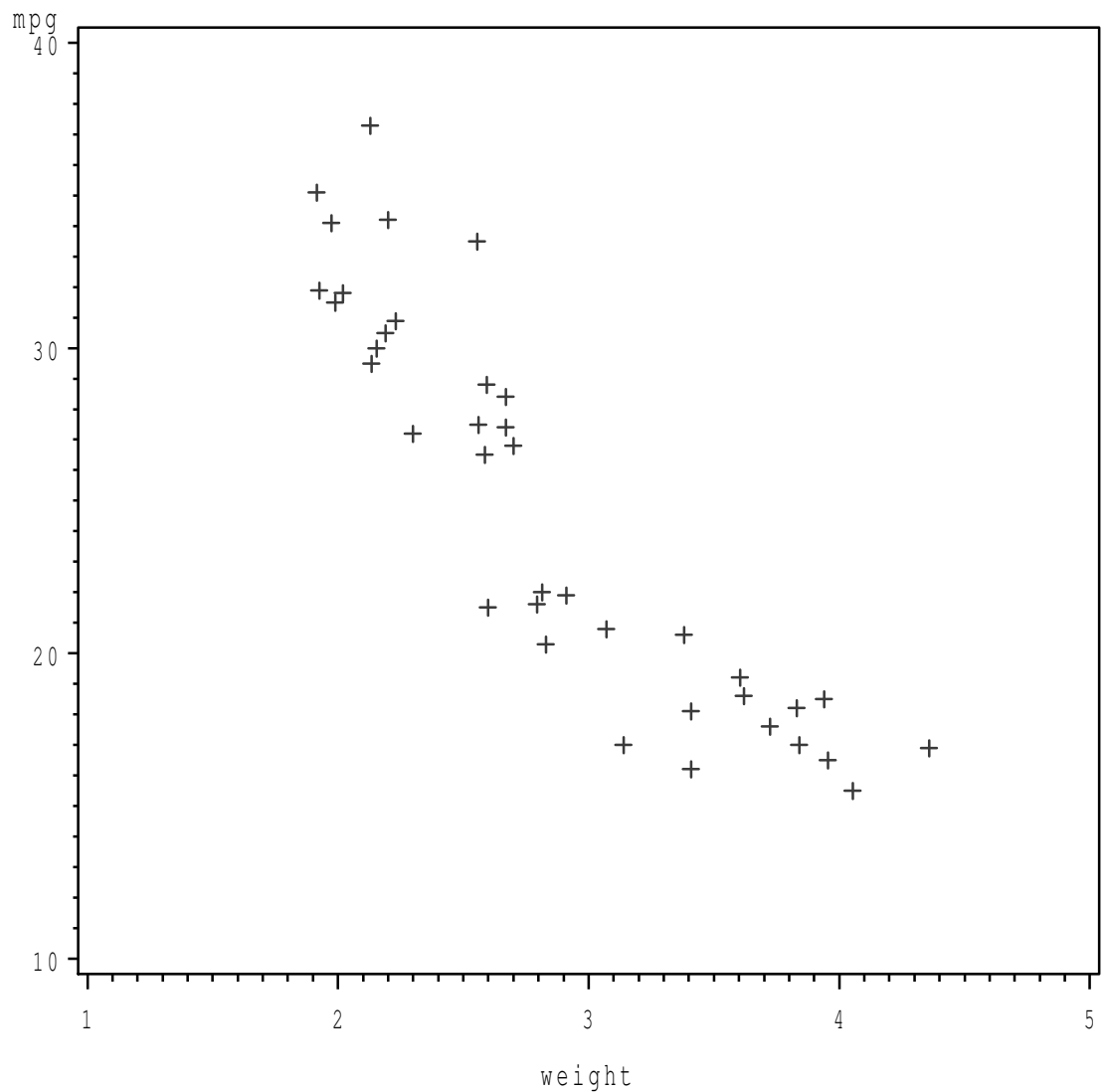
The plus signs on the boxes are the means. This is something that R doesn't show. The 6-cylinder cars have a high outlier, which is probably why the mean is bigger than the median for those.

10.7.2 Scatterplots

Introduction

We already mentioned `proc gplot` to make scatterplots. I wanted to talk more about that, and about how to annotate them (as we did for R). Let's start with a scatterplot of gas mileage against weight:

```
SAS> proc gplot data='cars';  
SAS>   plot mpg*weight;
```



Note that the vertical-scale variable goes *first*.

There is a clear downward trend: a heavier vehicle tends to have a worse gas mileage. This makes sense, because the heavier a car is, the more gas it should take to keep it moving.²⁰

²⁰If there were no friction between the car and the road, *no* fuel would be needed to keep it moving at a constant speed; the only use of fuel would be for acceleration, principally to start the car moving in the first place. But there *is* friction, so keeping a heavier car moving (overcoming this friction) takes more fuel. Plus, “typical” driving involves about the same number of starts and stops regardless of the size of vehicle you have.

Adding a regression line

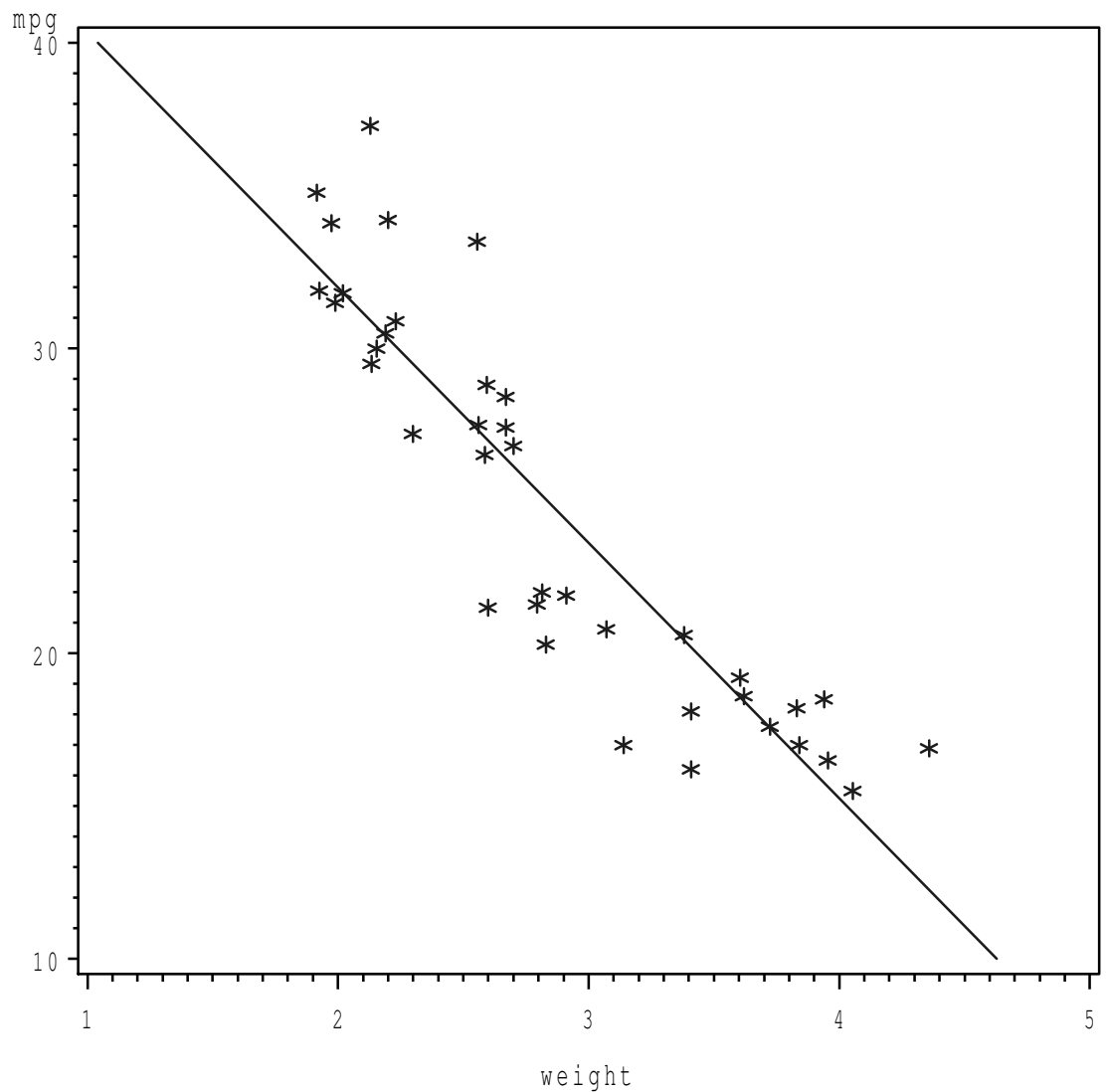
This trend looks more or less linear, so you might think of putting a regression line on the plot. This is a bit more awkward in SAS than it is in R. The key is `symbol`. With `symbol` and a number you specify all of these:²¹

- `c`= Colour. Use a colour name like `red` or `green` or `black` after the equals. No quotes needed.
- `i`= Draw lines. Commonest one is `rl` for regression line. There is also `rq` for a quadratic (parabola) regression.
- `l`= Line type. 1 is solid, 2 is dots, and so on.
- `v`= Plotting symbol. By name, for example `square`, `star`, `circle`, `plus`, `dot`.

All right, let's try this out. I'll make the plotting points stars, make everything blue, and draw a regression line. Note how we define our `symbol` first:

```
SAS> symbol1 c=blue i=rl l=1 v=star;  
SAS> proc gplot;  
SAS>   plot mpg*weight;
```

²¹Maybe not all, but if you don't specify all the ones you need, you'll get an error. SAS doesn't appear to have defaults to enable you to specify some and use the defaults for others.



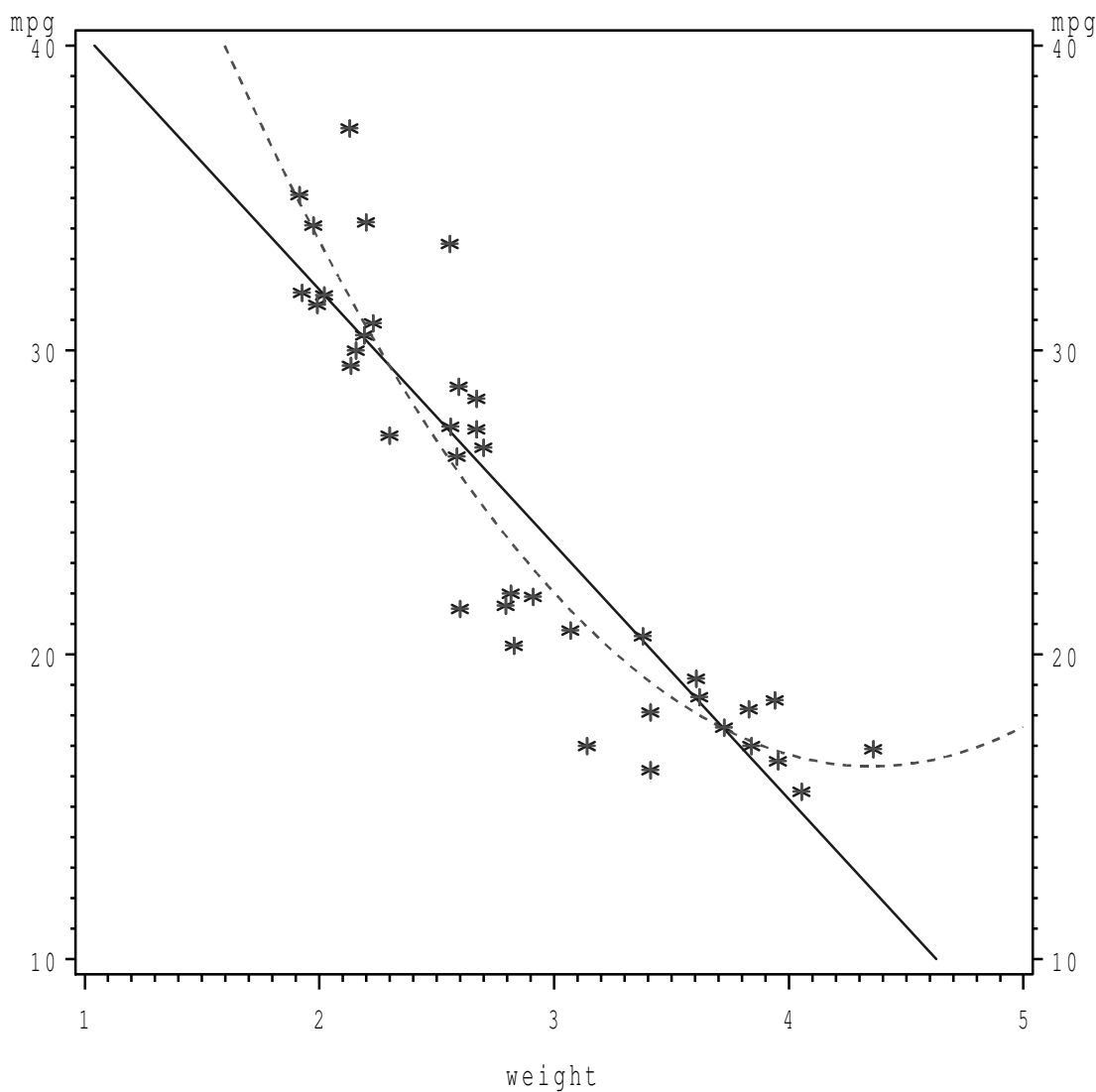
That didn't come out blue here, but that's a consequence of the method I'm using to get my SAS output in these notes. I just tried it the regular way, and it *did* come out blue.

Adding a parabola

Now, there's a bunch of cars with weight about 3 tons that all come out *below* the line, so that their gas mileage is worse than the line would predict. Would fitting a curve be better? Let's try a quadratic curve as well. I would like to

have the line and the curve on the same plot. That can be accomplished like this:

```
SAS> symbol1 c=blue i=r1 l=1 v=star;
SAS> symbol2 c=red i=rq l=2 v=;
SAS> proc gplot;
SAS>   plot mpg*weight;
SAS>   plot2 mpg*weight;
```



A couple of pieces of strategy here: define *two* symbols, one of which uses `i=r1` to get the line, and the other uses `i=rq` to get the curve. Then, inside

`proc gplot`, we add a `plot2` command, which adds another plot on top of the one generated by `plot`. What actually happens is that the points would get overplotted (which is why I used `v=` to not plot the points the second time), but the line that is “interpolated” is different: a solid regression line and a dashed quadratic regression curve.

I think this business of having to specify everything each time is a lot more clunky than R’s `points` and `lines`, but there you go.

As far as interpreting the plot is concerned: the curve goes a little closer to the cars that weigh about 3 tons, but not much. More worryingly, once you get to a weight of a bit less than 4.5 tons, the predicted `mpg` actually starts *increasing*, which makes no sense at all.²²

Smooth curve through scatter plot

SAS has a procedure called `proc loess` that I seem to be unable to demonstrate for you here. But if you run it from SAS directly (with HTML graphics) like this:

```
SAS> proc loess data='cars';
SAS>   model mpg=weight;
```

substituting your own preferred data set and variables, you’ll see a largish amount of output²³, including the following:

- Smoothing parameter selection: any smoothing method has a smoothing parameter that you can either choose, or let SAS²⁴ estimate from the data. Depending on the value of the smoothing parameter, your smoothed trend can be anything between very smooth and very bumpy. SAS shows you how well some values for the smoothing parameter fit to the data, and marks the best one with a star.
- The most important picture is the Fit Plot. This is the scatter plot with the smoothed curve superimposed, as you would get from `lines(lowess(x,y))` in R.
- The next batch of plots think of the loess curve as a model fit (like a regression). This means you can think of residuals (data minus smoothed curve) and assess the residuals for (in sequence along the first row) randomness, normality (via histogram and normal quantile plot). I haven’t figured out the two plots in the second row!

The way to get only the fit plot is this:

²²Consult the physics I outlined above if you have doubts about this.

²³As usual for SAS

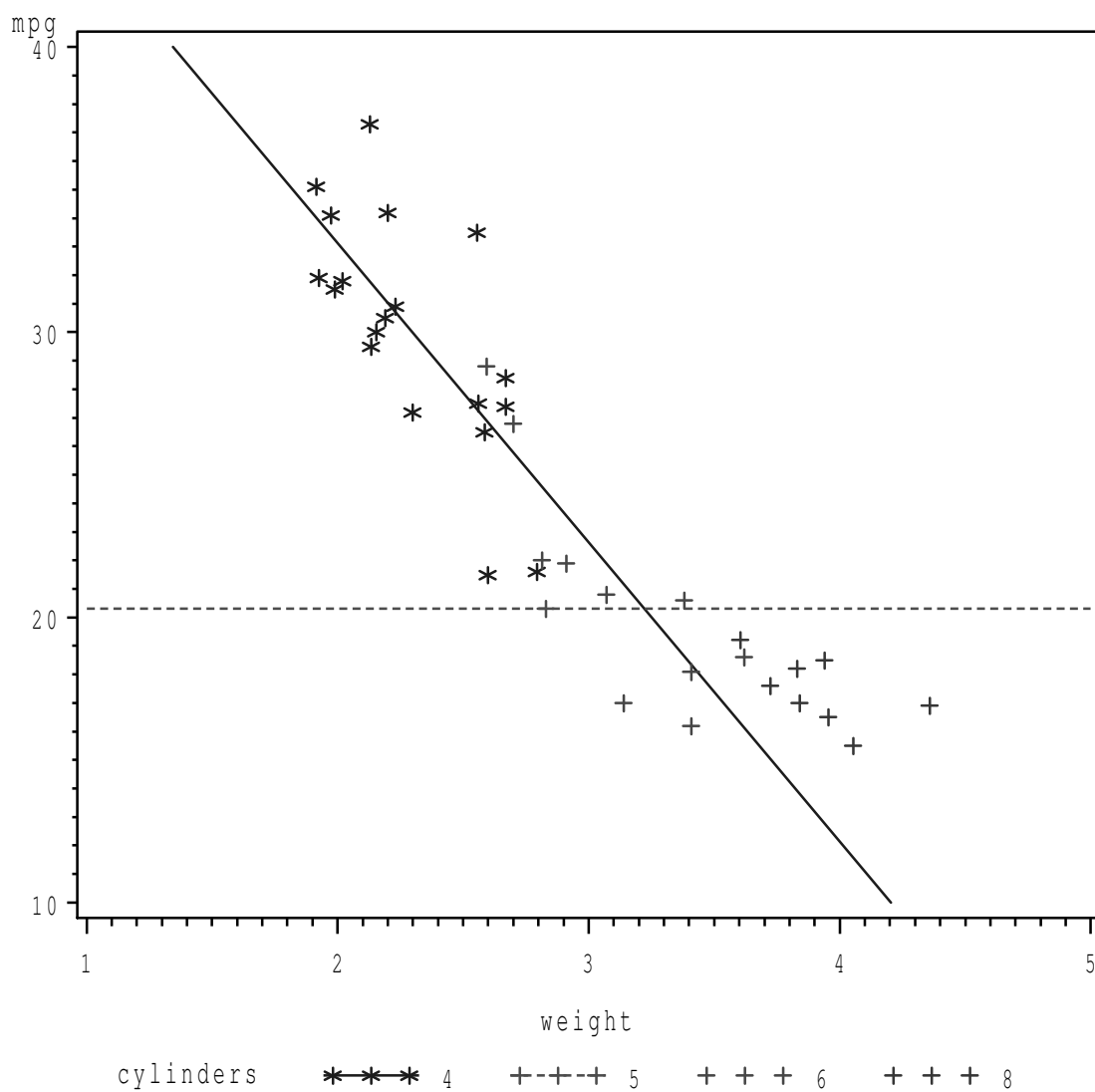
²⁴Or R, via `lowess`.

```
SAS> proc loess data='cars' plots(only)=fit;  
SAS>   model mpg=weight;
```

Distinguishing points by colours or symbols

Another kind of plot you might want to do has different colour or symbols for groups of points. For example, we might plot gas mileage by weight, but distinguishing cars with different numbers of cylinders. That goes like this:

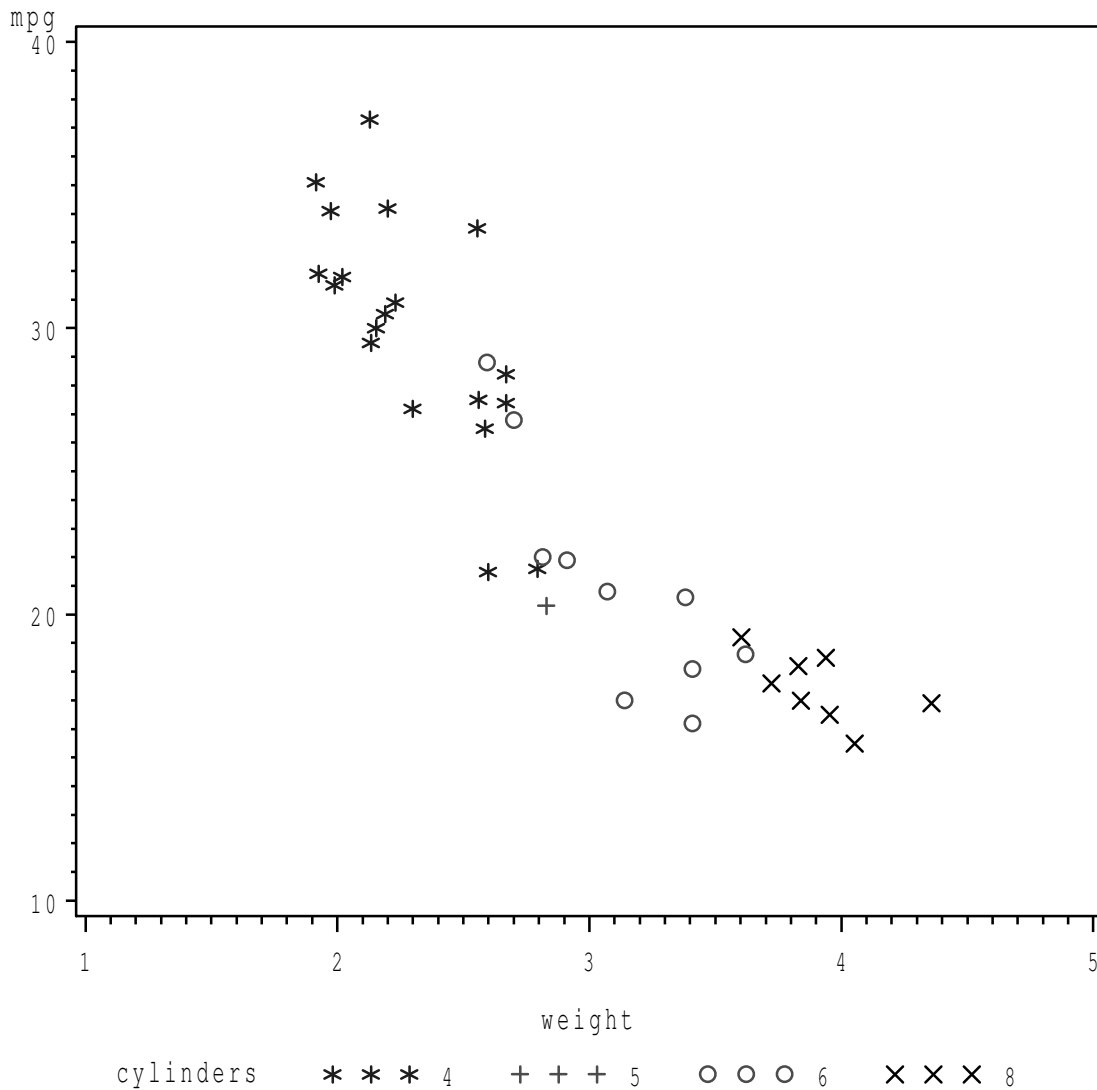
```
SAS> proc gplot data='cars';  
SAS>   plot mpg*weight=cylinders;
```



This re-uses the `symbols` that we defined before, which is how come we got the lines. Since the points are not joined by lines, we can't distinguish 4 cylinders from 5 or 6 from 8. So we'll define new symbols. There are 4 values of `cylinders`, so we need 4 symbols:

```
SAS> symbol1 c=blue i=v=star l=;
SAS> symbol2 c=red i=v=plus l=;
SAS> symbol3 c=green i=v=circle l=;
SAS> symbol4 c=black i=v=x l=;
SAS>
```

```
SAS> proc gplot data='cars';
SAS>   plot mpg*weight=cylinders;
```



As before, the colours don't work here, but I did get different symbols, as shown in the legend at the bottom of the plot. As a point of technique, I found out how to set as option as "I don't care": leave it empty²⁵. But it has to be there, or else you'll get an error. The `symbol` things are "sticky": they will stay the same until you restart SAS, explicitly re-define them²⁶ or reset them with `goptions`

²⁵If you look in the Log window, you'll see that this has given you a warning.

²⁶Which is what we did here.

```
reset=all.
```

Multiple series on one plot

Let's illustrate this with the oranges data set from Chapter 9. First, let's read it in from the data file, which looks like this:

```
row ages A B C D E
1 118 30 30 30 33 32
2 484 51 58 49 69 62
3 664 75 87 81 111 112
4 1004 108 115 125 156 167
5 1231 115 120 142 172 179
6 1372 139 142 174 203 209
7 1582 140 145 177 203 214
```

Remember, in SAS, that the variable names go on the `input` line, so we have to skip over the first line of the file. Let's also create a permanent data file while we're about it, which is just a matter of putting quotes in the right places:

```
SAS> data 'oranges';
SAS>   infile "oranges.txt" firstobs=2;
SAS>   input row age a b c d e;
```

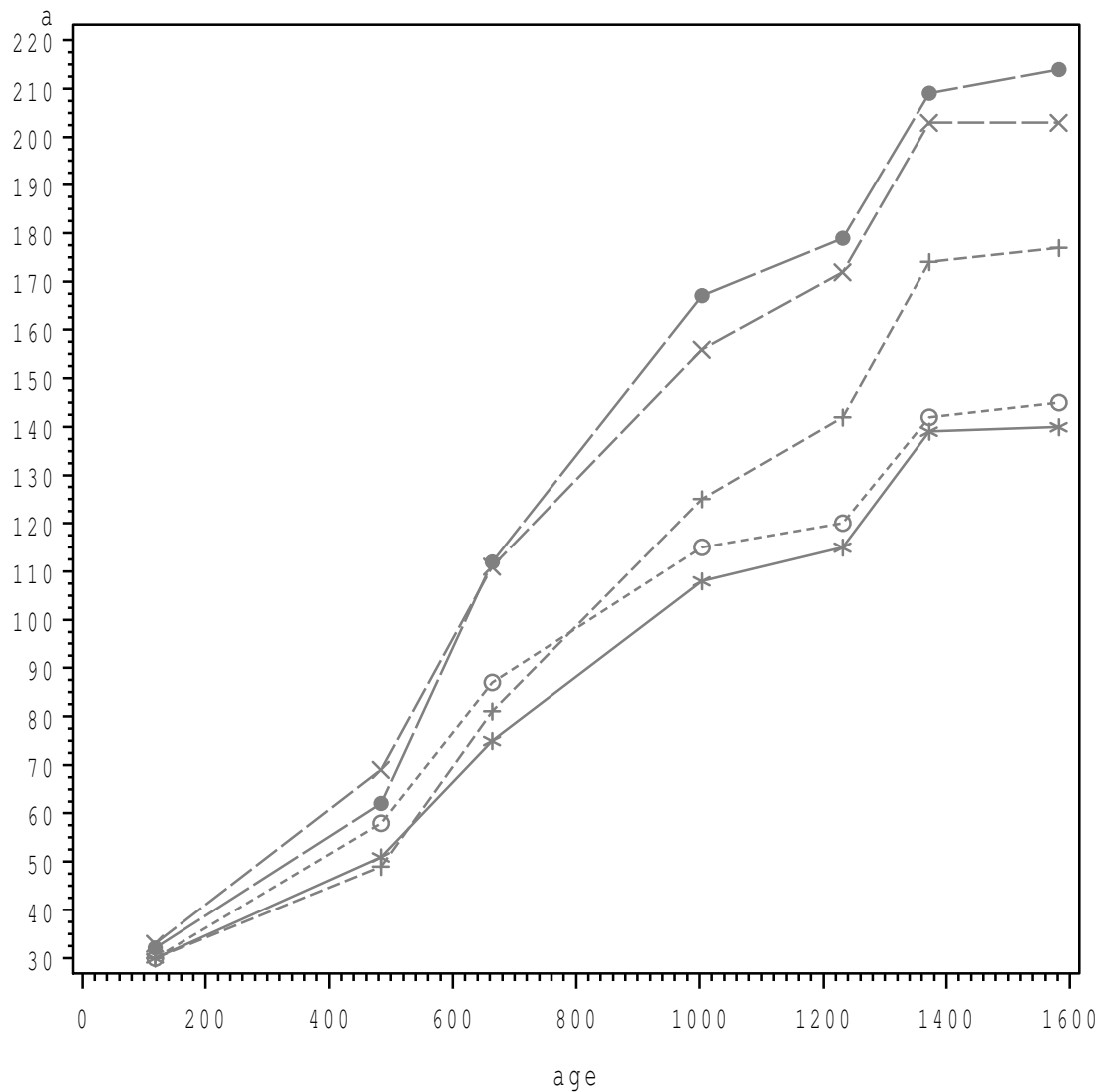
Now, the way to plot multiple series is to put them one after another on the plot line. Also, we want the five series to come out different, joined by lines, so we have to define five symbols first:

```
SAS> symbol1 c= v=star i=join l=1;
SAS> symbol2 c= v=circle i=join l=2;
SAS> symbol3 c= v=plus i=join l=3;
SAS> symbol4 c= v=x i=join l=4;
SAS> symbol5 c= v=dot i=join l=5;
```

Since I don't have colour here, I left all the `c=` blank, but I filled in everything else. `v=` is the plotting symbol; `i=join` means to join the successive points by lines (rather than, say, putting a regression line through them all), and `l=` is the line type.

Now we have to make the plot:

```
SAS> proc gplot;
SAS>   plot a*age b*age c*age d*age e*age / overlay;
```



Note the use of the `overlay` option to get all the plots to appear on the one set of axes. If you forget this,²⁷ you'll only get one plot (probably the `a*age` one).

This could use a legend, and the vertical axis. Let's see if we can manage that. In the code below, the new bits are `axis2` and `legend1`.

The `axis2` specifies something non-default for an axis (we link it with the `y`-axis below). The non-default thing is that the axis label should be "circumferer-

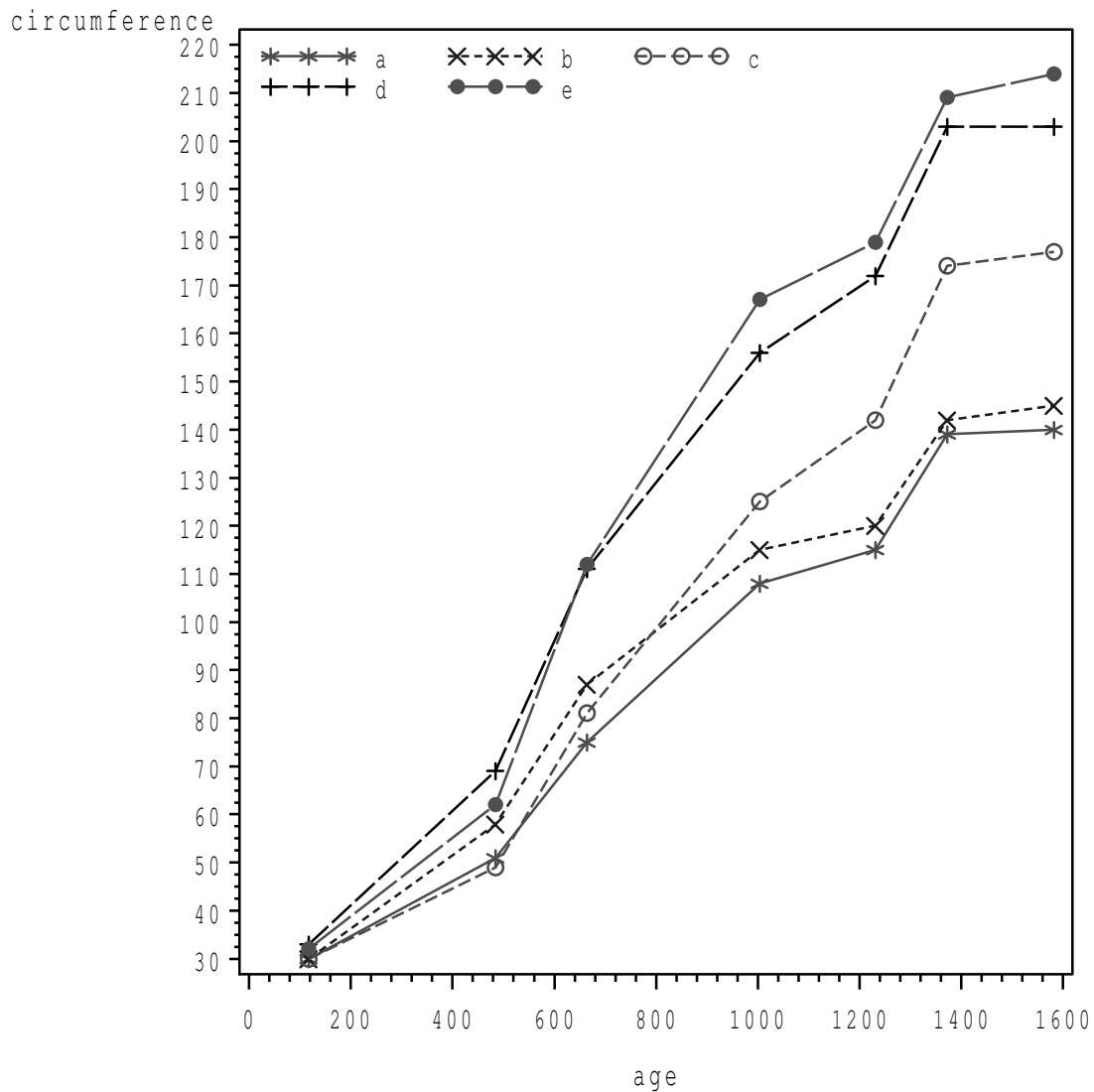
²⁷I did, the first time.

ence”.

The legend is made by `legend1`. There appears to be some necessary black magic: `label=none` removes the default label that SAS puts on the legend (which, I discovered, is the word `PLOT`), the next line says where to put the legend, and the last line has no effect that I can see, but is supposed to allow the graph and legend to share space. Note that the three lines in the `legend1` piece are really just one line of code that’s a bit long, so only the last one has a semicolon. There’s nothing about what to put in the legend; SAS will figure that out.

Lastly, we have to add something to the `proc gplot` to make sure that these axis and legend modifications actually happen. After the slash and the `overlay` that we had before, the `legend=legend1` and `vaxis=axis2` get SAS to use the modifications to the legend and vertical axis that we described in `legend1` and `axis2` respectively. I think we had to put them in `legend` with a number and `axis` with a number, as we do for `symbol`, but they could have been `legend2` or `axis1` without bothering anything.

```
SAS> symbol1 c=red v=star i=j l=1;
SAS> symbol2 c=blue v=x i=j l=2;
SAS> symbol3 c=green v=circle i=j l=3;
SAS> symbol4 c=black v=plus i=j l=4;
SAS> symbol5 c=brown v=dot i=j l=5;
SAS>
SAS> axis2 label=('circumference');
SAS> legend1 label=none
SAS>   position=(top left inside)
SAS>   mode=share;
SAS>
SAS> proc gplot;
SAS>   plot a*age b*age c*age d*age e*age /
SAS>     overlay legend=legend1 vaxis=axis2;
```

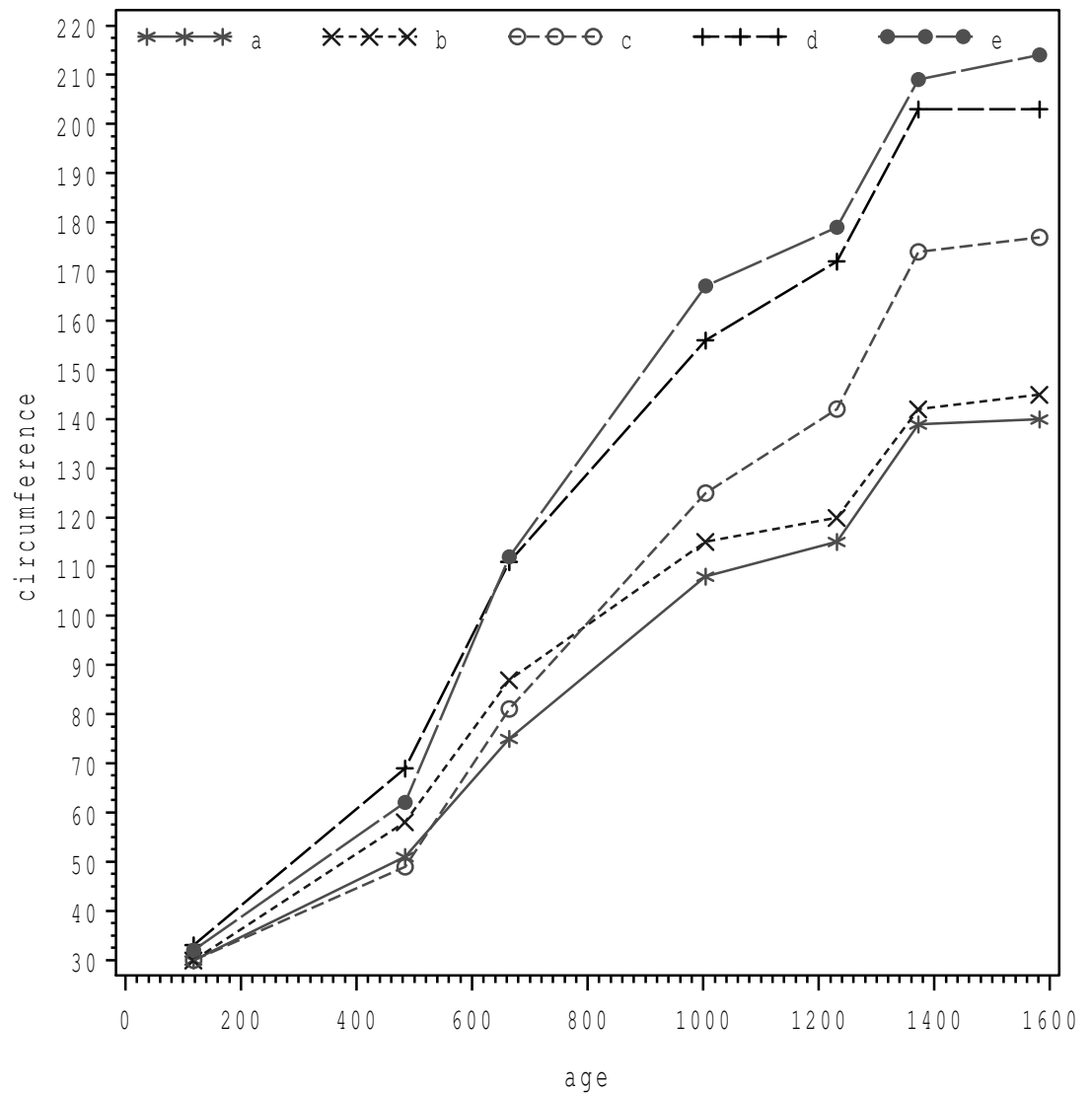
The word “circumference” is printed (horizontally) top left, outside the graph, which has made the graph a bit narrower than we might like. What we can do to fix this is to rotate the text used on the axis label through 90 degrees (counterclockwise), which is done by changing the `axis2` line to read as below. Note that I don’t need the `symbol1` lines, since SAS will re-use the ones I defined before.

```
SAS> axis2 label=(angle=90 'circumference');
SAS> legend1 label=none
SAS> position=(top left inside)
```

```

SAS> mode=share;
SAS>
SAS> proc gplot;
SAS> plot a*age b*age c*age d*age e*age /
SAS> overlay legend=legend1 vaxis=axis2;

```



and I think we can be happy with that.

An easy-to-follow how-to for these things is at <http://www2.sas.com/proceedings/sugi31/239-31.pdf>. The first part of Chapter 14 of the SAS book is also worth consulting on this.

Labelling points on a plot

Before we do any more plotting in SAS, we should “undo” our re-definitions of symbols. This can be done by

```
SAS> goptions reset=all;
```

or by re-defining `symbol1`²⁸ back to what you want. In our case, the symbol can be a black plus, with no lines joining it to anything else:

```
SAS> symbol1 c=black v=plus i= 1=;
```

Section 16 of the above how-to shows one example of labelling points on a plot using the `annotate` facility.²⁹ The process is to define a new data set containing the text to be plotted and the locations at which it should be plotted.³⁰

Let’s go back to our cars, and the scatterplot of `mpg` against `weight`. Here’s a silly one to begin with: we label each point with the word “hello”. That goes as shown below.

The first step is to define a new data set (which I called `mytext1`) that contains all the variables in the original data set plus some new ones. The `retain` line defines `xsys` and `ysys` to be the text string 2 for each car.³¹ Then, after bringing in all the variables in `cars`, we define `x` and `y` to be the *x* and *y* coordinates of where the text is going to go, and the variable `text` contains the actual text.

The second stage is to draw the plot, but to include some extra stuff that includes the labels, which is a `/` (meaning “options coming”), and then `annotate=` and the name of your labels data set. That all produces this:

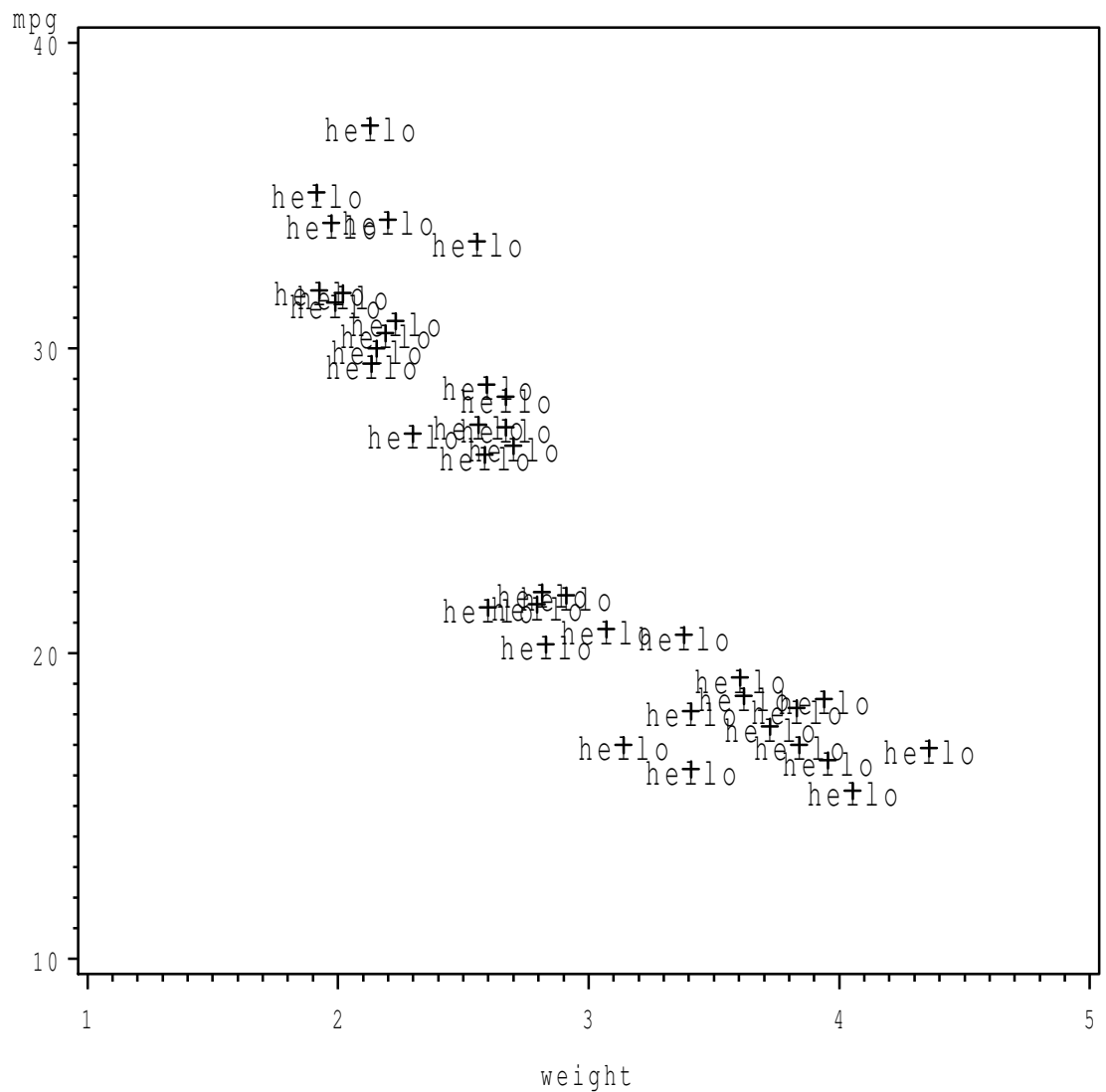
```
SAS> data mytext1;
SAS>   retain xsys ysys '2';
SAS>   set 'cars';
SAS>   x=weight;
SAS>   y=mpg;
SAS>   text='hello';
SAS>
SAS> proc gplot data='cars';
SAS>   plot mpg*weight / annotate=mytext1;
```

²⁸And any other symbols you might want to use again.

²⁹A “facility” is “something that permits the easier performance of an action”. I leave it to you to decide whether `annotate` actually does that.

³⁰Like `text` in R.

³¹Don’t ask me, but it appears to be necessary, otherwise the text comes out on the vertical axis.



Is that confusing? Most certainly. The how-to, which is from `sas.com`, is honest enough to say this:

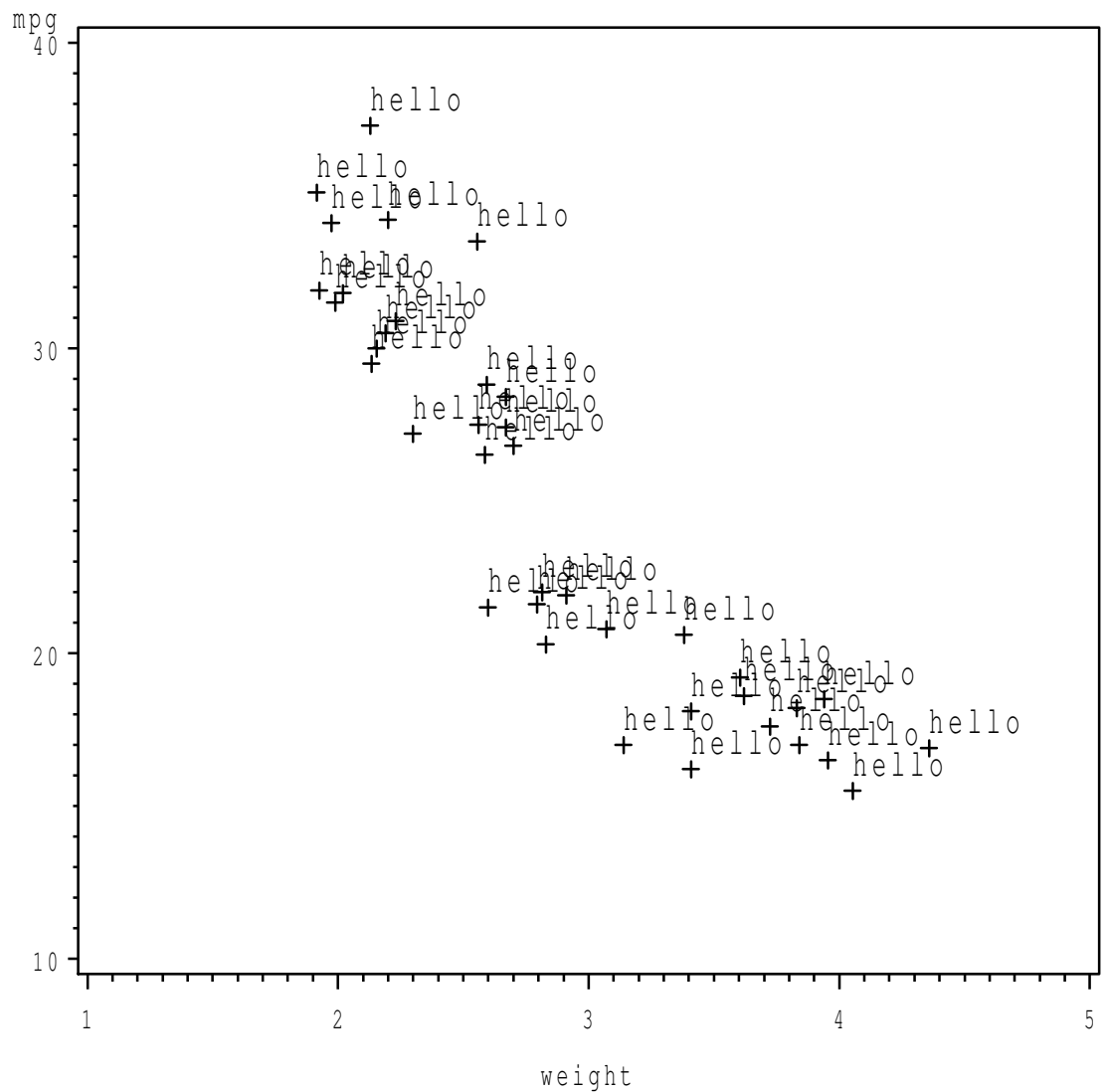
If you are really new to SAS/GRAPH (or SAS in general), the following may seem a little obtuse. Do not worry, it's not just you, ANNOTATE is obtuse.

The text labels overwrite the +’s that mark where they belong. We need something like R’s `pos` to put the labels next to the points, not on top of them. This

is a `position` thing that goes on the `retain` line.³² The only thing that has changed is that `retain` line. The `position '3'` puts the text above and to the right of the point it belongs to. If you go looking for the SAS help “Effect of POSITION values on text strings”, you’ll see all 16 possibilities.

```
SAS> data mytext2;
SAS>   retain xsys ysys '2' position '3';
SAS>   set 'cars';
SAS>   x=weight;
SAS>   y=mpg;
SAS>   text='hello';
SAS>
SAS> proc gplot data='cars';
SAS>   plot mpg*weight / annotate=mytext2;
```

³²Why it goes there I have no idea, but it does.



It's rather silly to label all the points with "hello". Usually, you want to label them with the value of a variable, like `country`. This means changing the `text=` line in the new data set to the name of a variable.³³:

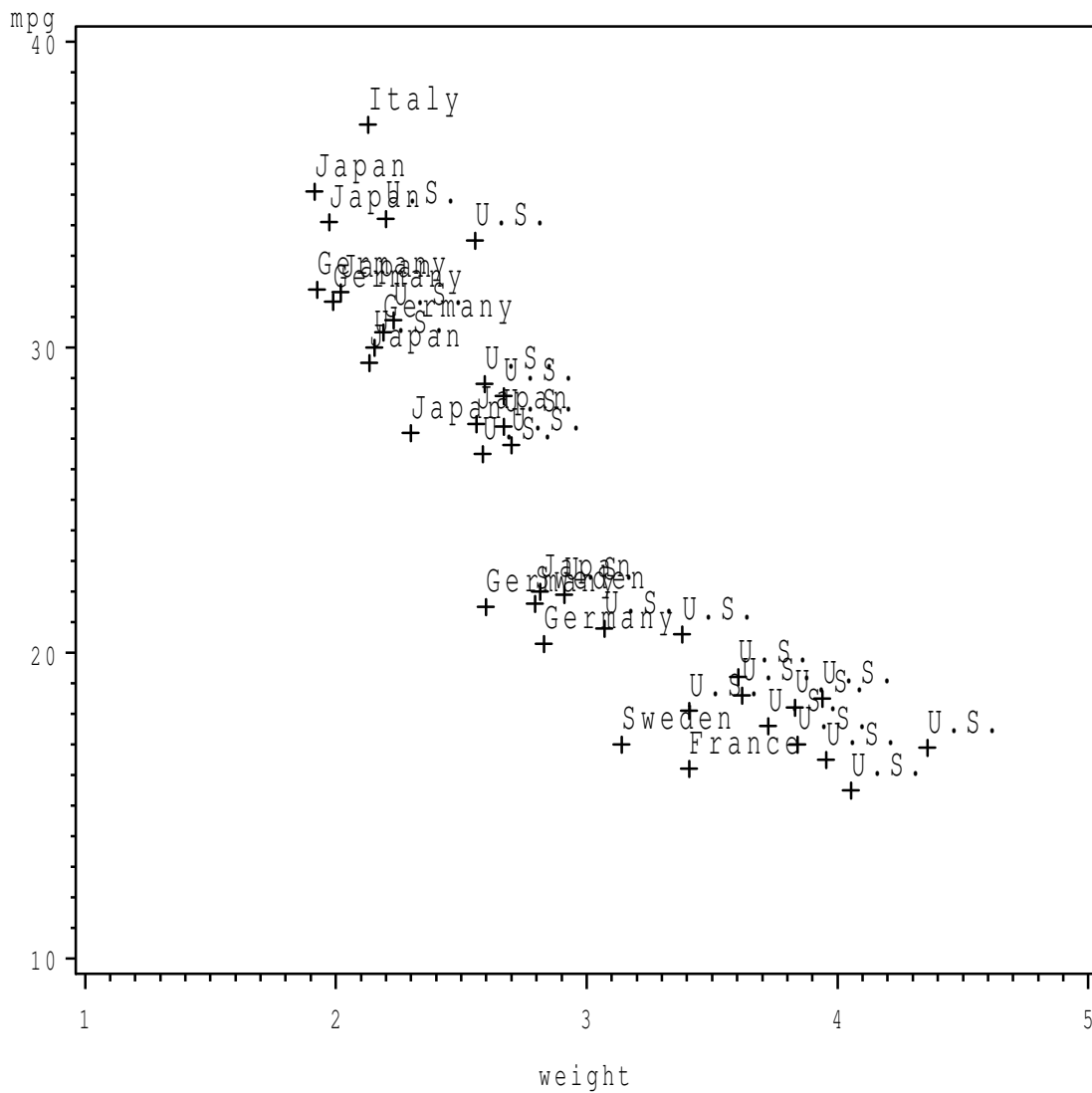
```
SAS> data mytext3;
SAS>   retain xsys ysys '2' position '3';
SAS>   set 'cars';
SAS>   x=weight;
SAS>   y=mpg;
```

³³A text one, not something like `cylinders`.

```

SAS> text=country;
SAS>
SAS> proc gplot data='cars';
SAS> plot mpg*weight / annotate=mytext3;

```



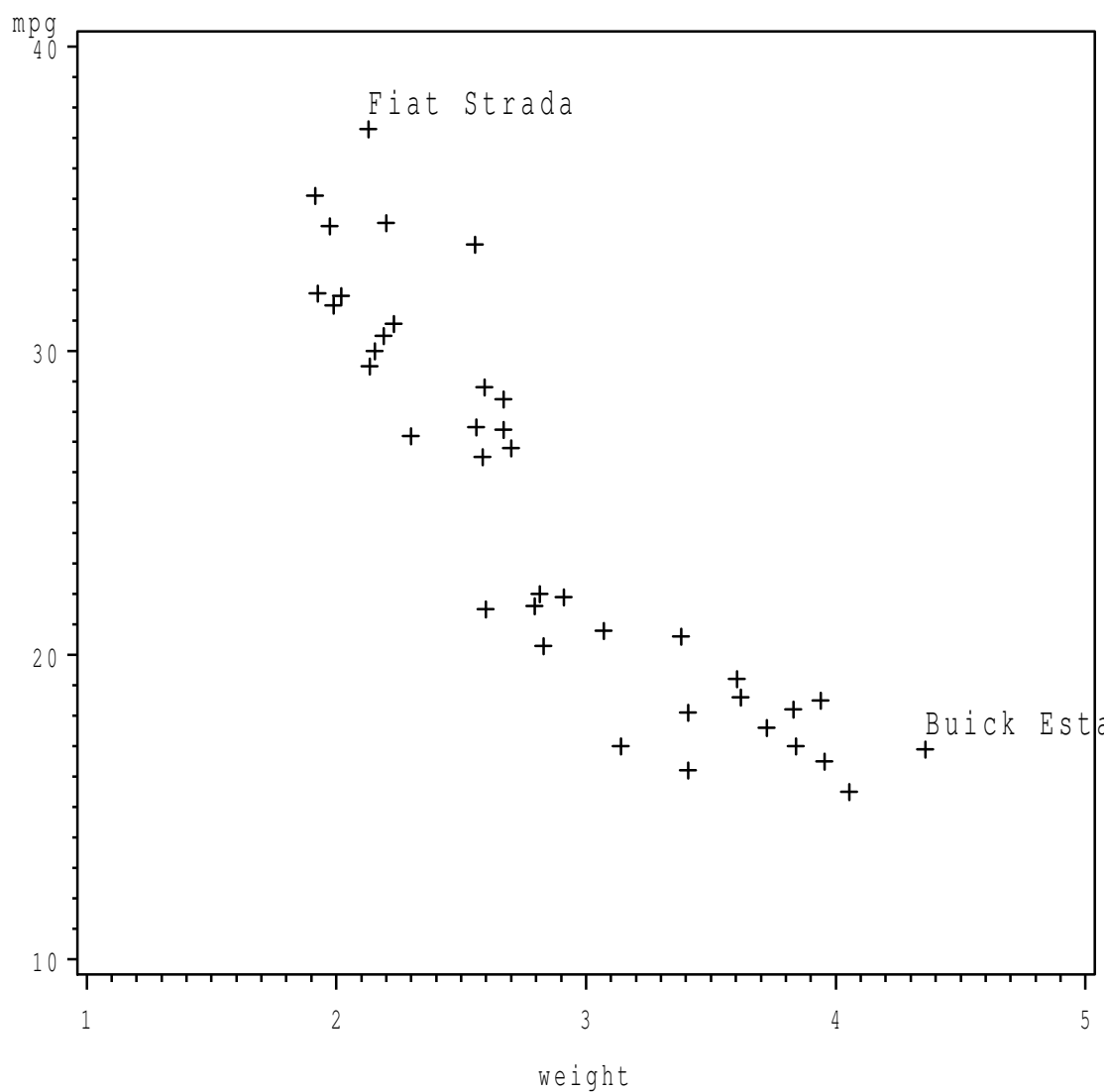
The labels overwrite each other quite a lot, but because you know they are above and to the right of the point they belong to, you can usually figure out which label belongs to which point. We see that the heaviest cars are all American.

What if we want to label only *some* of the cars? Then, when we define our `mytext` data set, we need to pick out just those cars we want. A plot we did in R was to label just the heaviest car and the most fuel-efficient car, which were cars 9 and 4 respectively in the data set. To pick out just them, we need some cleverness in defining our new data set, as below. There are several things we need to worry about:

- “Observation number” or “line” in the data set is known by the SAS special variable `_n_`.
- We can use `|` to represent “or”.
- To select just some observations, we use `if`.
- When several things have to be done if the `if` is true, we have to use the form shown below. End the `if` line with `then do;`, then do all the things that have to be done, which in this case means setting `x`, `y` and `text`,³⁴ and then end with `end;`.

```
SAS> data mytext4;
SAS> retain xsys ysys '2' position '3';
SAS> set 'cars';
SAS> if (_n_ eq 4 | _n_ eq 9) then do;
SAS>     x=weight;
SAS>     y=mpg;
SAS>     text=car;
SAS> end;
SAS>
SAS> proc gplot data='cars';
SAS> plot mpg*weight /
SAS>     annotate=mytext4;
```

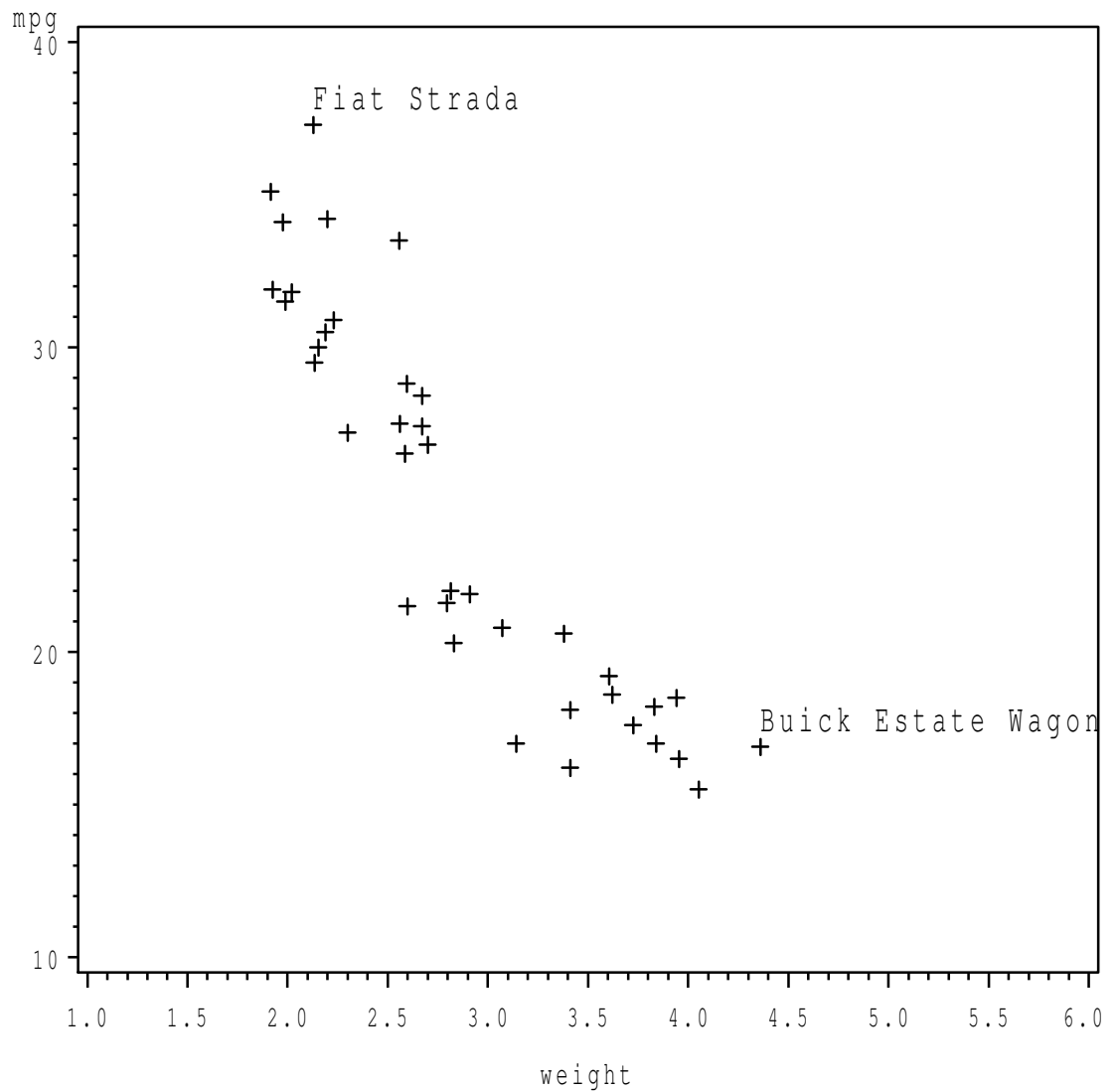
³⁴This time we want the text to be the name of the car, not which country it comes from.



As happened with R, the Buick Estate Wagon's name doesn't fit on the graph. We can extend the x axis. This is done in SAS by defining a new axis, and then using it on the plot. Let's make the axis go from 1 to 6, with the major tick marks being at intervals of 0.5. We can use the `mytext4` again that we defined before, so we don't need to re-define that.

```
SAS> axis1 order=(1 to 6 by 0.5);
SAS>
SAS> proc gplot data='cars';
SAS>   plot mpg*weight /
```

```
SAS>   annotate=mytext4
SAS>   haxis=axis1;
```



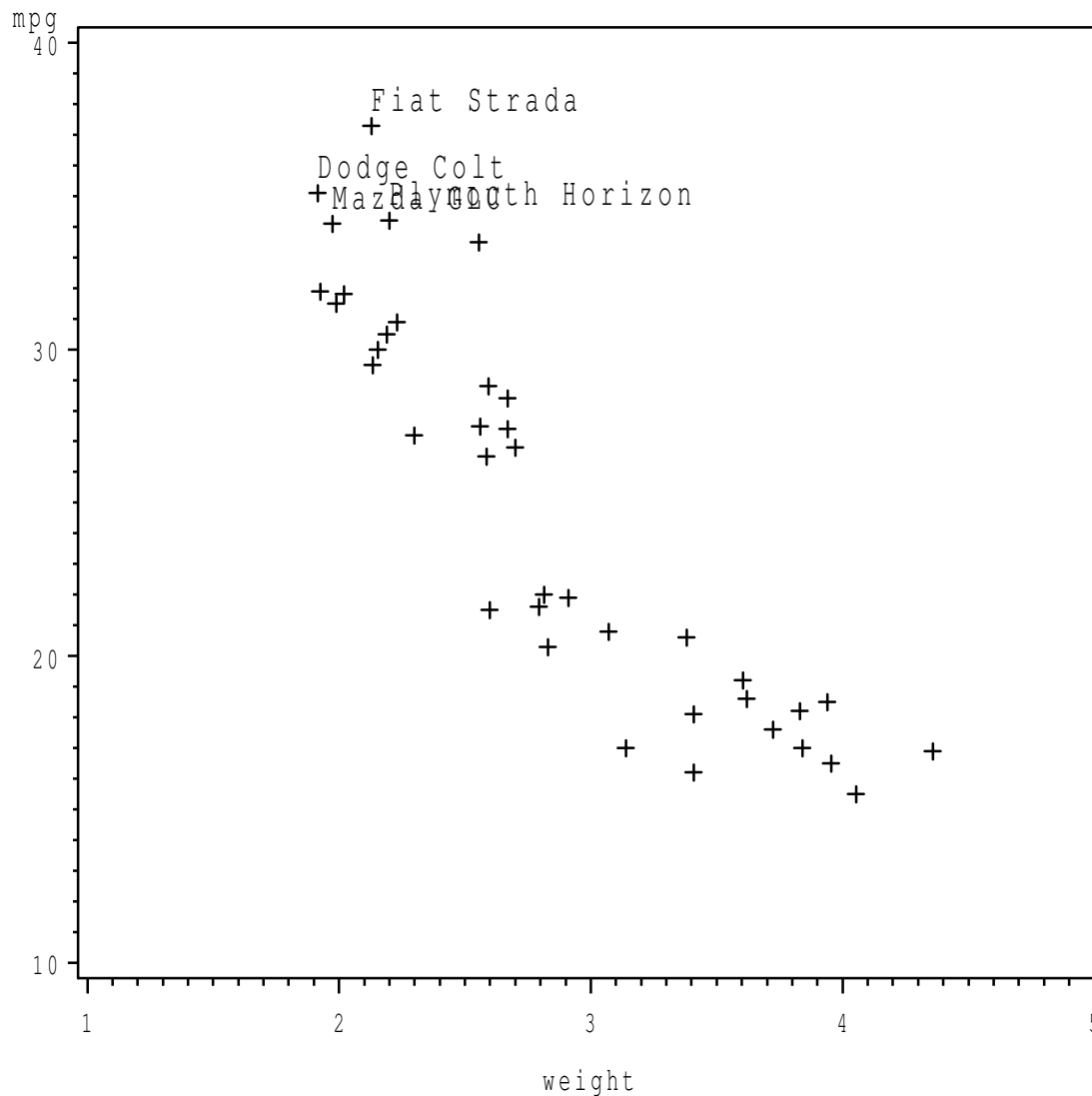
The same kind of logic as above would enable us to label the cars with gas mileage bigger than 34:

```
SAS>   data mytext5;
SAS>   retain xsys ysys '2' position '3';
SAS>   set 'cars';
SAS>   if mpg>34 then do;
SAS>       x=weight;
```

```

SAS>      y=mpg;
SAS>      text=car;
SAS>      end;
SAS>
SAS> proc gplot data='cars';
SAS>      plot mpg*weight / annotate=mytext5;

```



If you look carefully, you can see that the left-hand car with mpg just bigger than 34 is the Mazda GLC, and the right-hand one is the Plymouth Horizon. You might be surprised that two of the four high-MPG cars are American. I

had a friend who owned a Dodge Colt, and he told me it was built by Mitsubishi Motors.³⁵

³⁵According to Wikipedia, the Plymouth Horizon was developed by Simca, the French arm of Chrysler Europe. So those “American” cars were not so American after all.

Chapter 11

Data munging

11.1 Introduction

Data rarely come to us in the form that we wish to use them. There could be errors, or the values are not in the right format, or we need things in rows that are actually in columns, or anything like that. In computer science, “munging” means re-arranging data into the form you want, so I’ve borrowed the term to describe what we often need to do before we can do an analysis.

R and SAS both have tools to check and re-arrange data. We’ll take a look at some of these.

11.2 Handling data

The data, as they come to you, are sacrosanct. That means that you should never edit the data values that you receive; you should always work with a copy. That way, if you mess something up, you always have the original data to work with.

One way of handling this is to create a new folder for each project. Within that folder, you create a folder `data` that you received, *which you are not allowed to change*. You also create folders `R` or `SAS` (as appropriate), which contain commands to read the data from your `data` folder and to perform analyses. As the analysis progresses, you’ll move from having snippets of code to do various things to having complete scripts that take your data, process it as necessary, produce the final output and even (if you’re using something like R Markdown) make the final report. A good folder layout helps with keeping everything together. I like the one described at <http://nicercode.github.io/blog/2013-04-05-projects/>. This blog post talks about R functions, for

which see Section 9.9. The basic setup here is: snippets of R/SAS code belong in the main `proj` folder; as they turn into complete programs/functions you can move them to the `R` (or `SAS`) folder; `doc` contains the report; `figs` contains graphs and such that you could reproduce if you needed to; `output` contains output that took some time to create, so that you *could* reproduce it if you needed to, but you'd rather not. When you've completed the analysis, you make a file called `analysis.R` (or `analysis.sas`) that, when run, reproduces the whole analysis, so that you can always obtain the most recent version of everything.¹

Computer Science people always ask you to comment everything, and that advice applies here as well. One blog I saw suggested “metafiles” (using the name `README`) in appropriate places that document where everything came from.² Another tendency we all have is to create files with names containing `new` or `old`: what happens when you have a new `new` file?³ If that is happening to you, you can create a subfolder with the current date in the format `2013-06-25` and copy the previous stuff there before you start modifying it. The reason for specifying the date this way is that the folders will get sorted into chronological order when you look at them, so that it's immediately obvious which is the most recent one.

11.3 Checking data

11.3.1 Introduction

The most important step to take *first*, before we do any analysis, is to check that our data are correct. There are several issues here:

- Directly checking that the data values are correct.
- Looking for outliers (suspicious values).
- Examining missing values.

Let's examine these in turn:

11.3.2 Direct checking

The best way to ensure that our data are correct is to look at the data in our software, via `proc print` in SAS or by looking at the data frame in R, and

¹You, or your collaborator, will always have thoughts like “wouldn't it be nice if...” or “maybe we should put *this* in”, just when you think you are finished.

²And how to get it back if you lose it.

³Or an old `old` one. Or even a new `old` one.

physically checking each value against the original source. Errors in transcription or data entry are easy to make, and this, for small data sets, is the best way to verify things.

If the data set is too large to make this doable, you can take a random sample of rows and check those. This becomes a case of balancing the value of checking (more rows gives a better chance of finding errors) against your time (more rows take longer to check).

Just to illustrate this, let's randomly select 5 rows of our `cars` data file, first in SAS, then in R.

In SAS, `proc surveyselect` does random sampling, like this:

```
SAS> proc surveyselect data='cars' sampsize=5 out=cars1;
SAS>   id _all_;
SAS>
SAS> proc print;
```

```
The SURVEYSELECT Procedure
Selection Method      Simple Random Sampling
Input Data Set        CARS
Random Number Seed    832049497
Sample Size           5
Selection Probability  0.131579
Sampling Weight        7.6
Output Data Set       CARS1
```

| Obs | car | mpg | weight | cylinders | hp | country |
|-----|------------------|------|--------|-----------|-----|---------|
| 1 | Plymouth Horizon | 34.2 | 2.200 | 4 | 70 | U.S. |
| 2 | Ford Mustang 4 | 26.5 | 2.585 | 4 | 88 | U.S. |
| 3 | Datsun 810 | 22.0 | 2.815 | 6 | 97 | Japan |
| 4 | Dodge St Regis | 18.2 | 3.830 | 8 | 135 | U.S. |
| 5 | AMC Spirit | 27.4 | 2.670 | 4 | 80 | U.S. |

The `proc print` prints out the data set that comes *out* of `proc surveyselect`, since that was the most recently created one.

In R, I think the task is easier:

```
R> myrows=sample(nrow(cars),5)
R> myrows
R> cars[myrows,]
```

```
[1] 11 26 30 34 24
```

| | Car | MPG | Weight | Cylinders | Horsepower | Country |
|----|--------------------|------|--------|-----------|------------|---------|
| 11 | Chevy Malibu Wagon | 19.2 | 3.61 | 8 | 125 | U.S. |
| 26 | Ford LTD | 17.6 | 3.73 | 8 | 129 | U.S. |
| 30 | Chevette | 30.0 | 2.16 | 4 | 68 | U.S. |

```

34          BMW 320i 21.5   2.60          4          110 Germany
24          Saab 99 GLE 21.6   2.80          4          115  Sweden

```

In each case, you then check all the variables in the five rows selected against the original source (in this case, a car magazine).

Your data source might also contain summaries of the data, like means or standard deviations, which you can check against your data.⁴ Having the right mean and standard deviation isn't *proof* that your values are correct, but it does at least increase your confidence.

I might know, for example, that the mean MPG is 24.76, the smallest value is 15.5 and the largest is 37.3.

Simple tools like `proc means` are good for this:

```
SAS> proc means data='cars';
```

The MEANS Procedure

| Variable | N | Mean | Std Dev | Minimum | Maximum |
|-----------|----|-------------|------------|------------|-------------|
| mpg | 38 | 24.7605263 | 6.5473138 | 15.5000000 | 37.3000000 |
| weight | 38 | 2.8628947 | 0.7068704 | 1.9150000 | 4.3600000 |
| cylinders | 38 | 5.3947368 | 1.6030288 | 4.0000000 | 8.0000000 |
| hp | 38 | 101.7368421 | 26.4449292 | 65.0000000 | 155.0000000 |

from which I can easily check that the MPG statistics are correct, and

```
R> mean(cars$MPG)
```

```
[1] 24.76053
```

```
R> quantile(cars$MPG)
```

```

      0%      25%      50%      75%     100%
15.500 18.525 24.250 30.375 37.300

```

likewise. (The largest and smallest are easy to find from the original data source, even if they are not given to you: you just scan down the columns and keep track of the largest and smallest you've seen so far.)

11.3.3 Looking for outliers

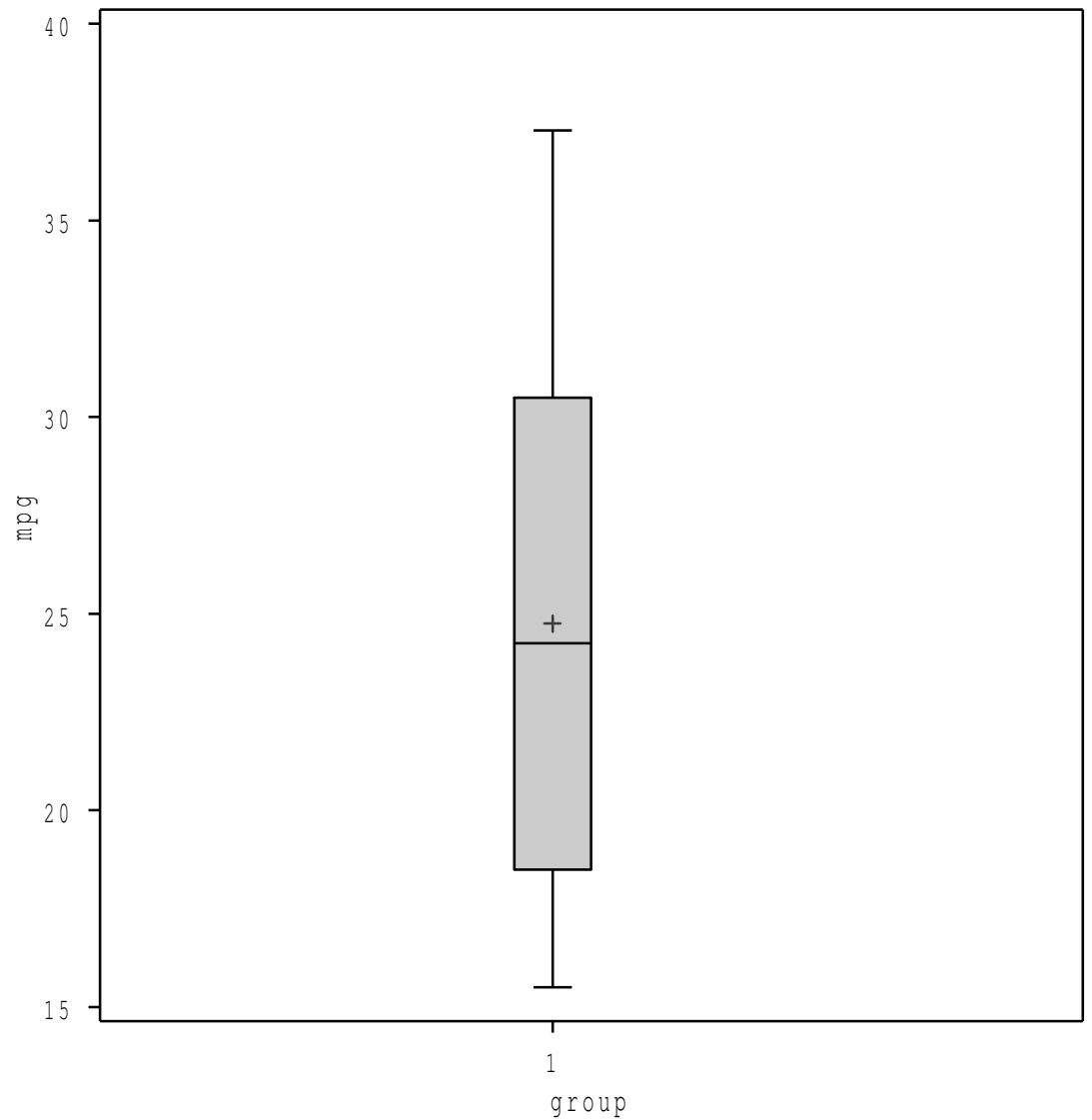
Outliers, as we know, can distort statistics such as means, or things calculated from means, such as regression slopes. One of the ways in which errant values often show up is as outliers: values that are much larger or smaller than you would have expected. For example, values that got mis-typed often end up too big or too small.

⁴I often use this when copying data from textbooks.

Of course, perfectly legitimate values might turn out to be outliers, but the point is that any value that shows up as an outlier is worth checking.

The simple tool for detecting outliers is the boxplot. You check the variables one at a time. In SAS, you have to remember to create a fake “group” variable if you don’t have one already, for example for the car MPG values:

```
SAS> data newcars;  
SAS>   set 'cars';  
SAS>   group=1;  
SAS>  
SAS> proc boxplot data=newcars;  
SAS>   plot mpg*group / boxtype=schematic;
```



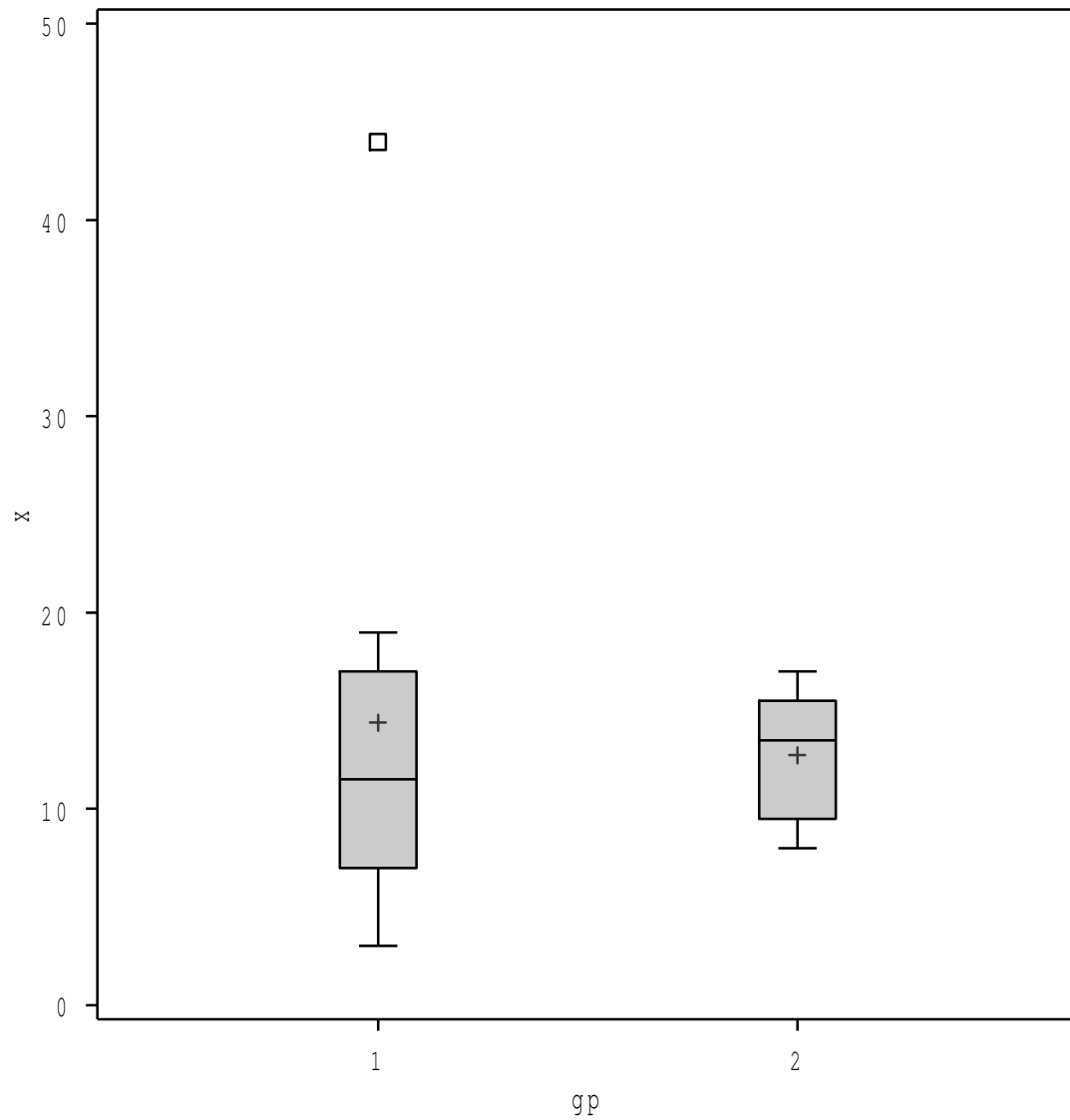
There are no outliers here.⁵

When you have data in groups, the idea is to check *each group* for outliers. Here's an example in SAS, with one data value deliberately wrong. You can probably guess, looking at the data, which one it is.

```
SAS> data mydata;  
SAS>   input gp x @@;  
SAS>   cards;
```

⁵Don't forget the `boxtype=schematic` or else any outliers won't get plotted at all!

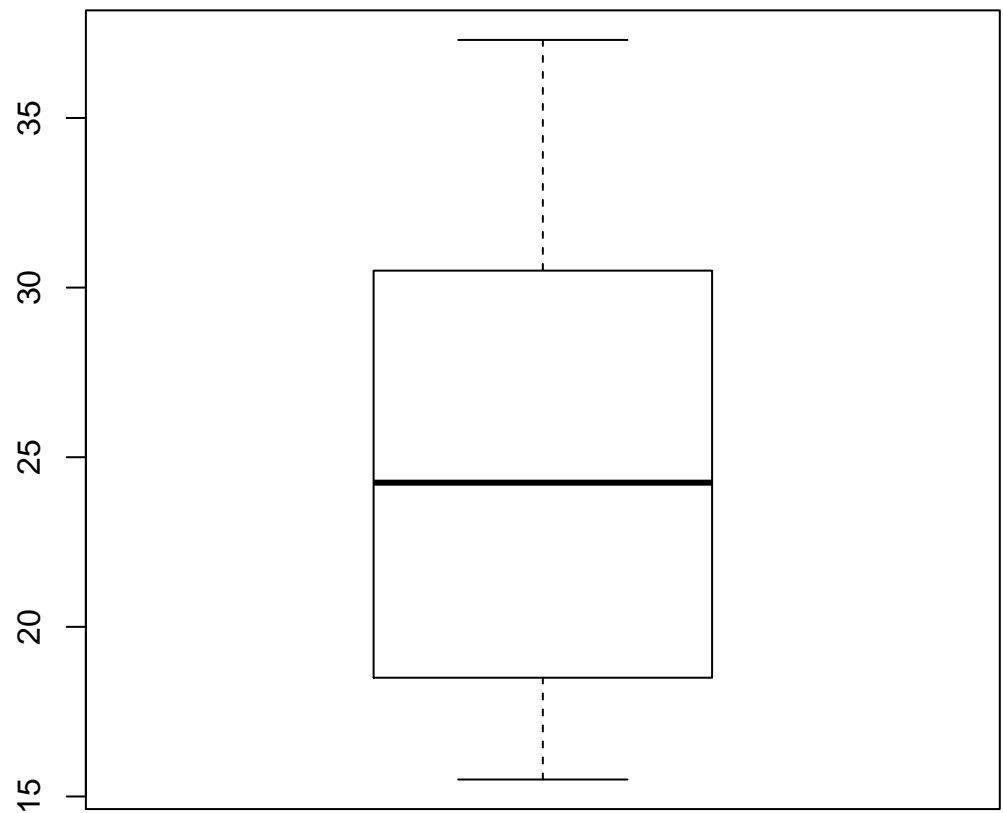
```
SAS> 1 3 1 5 1 17 1 16 1 11 1 44 1 10 1 7 1 12 1 19
SAS> 2 10 2 13 2 15 2 14 2 17 2 9 2 8 2 16
SAS> ;
SAS>
SAS> proc boxplot;
SAS> plot x*gp / boxtype=schematic;
```



Yes, that value 44 should have been 14.

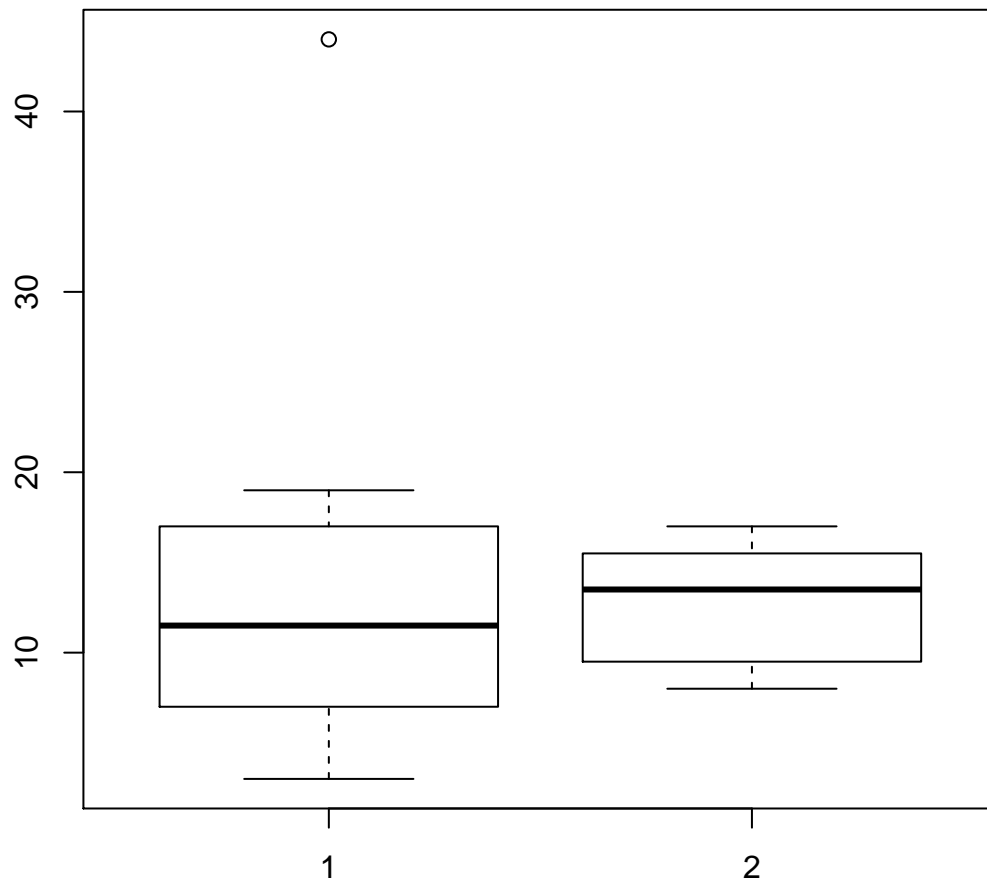
R has no need to create fake groups:

```
R> boxplot(cars$MPG)
```



The MPG values have no outliers.

```
R> od=read.table("outlier.txt",header=T)
R> boxplot(x~g,data=od)
```



This is the same data as above, with group 1 having an outlier.

The procedure after discovering an outlier is to go back and check the original data source. It may be that the suspect point is wrong (as in my example), but it may be that it is correct and we just have to live with it in the modelling.

If you've studied multiple regression, you might know about a concept called "leverage". This is a way of assessing outlierness in multiple dimensions: it asks "is the *combination* of variable values unusual?". It might be that an observation is not an outlier on any of the variables individually, but it *is* for the combination

of them. Here's an example of the kind of thing I mean. There are two variables called `u` and `v`:

```
u v
0 4
1 5
2 5
3 7
5 9
6 12
7 13
7 12
2 12
```

Take a look at the last observation. Neither its `u` nor its `v` are unusual; in fact, neither of them are the biggest and smallest values in the data set. But when `u` is 2, you'd expect `v` to be about 5, not as big as 12, and when `v` is 12, you'd expect `u` to be 6 or 7, not as small as 2. So the *combination* 2,12 is unusual.

To detect this, we're going to do a multiple regression with *all the variables we have as explanatory variables*. So what's the response variable? Doesn't matter. Make one up. Here's how it goes in R, with `z` being our made-up response:

```
R> uv=read.table("leverage.txt",header=T)
R> z=1:9
R> uv.lm=lm(z~u+v,data=uv)
R> h=hatvalues(uv.lm)
R> cbind(uv,h)
```

| | u | v | h |
|---|---|----|-----------|
| 1 | 0 | 4 | 0.3653835 |
| 2 | 1 | 5 | 0.2572723 |
| 3 | 2 | 5 | 0.2688846 |
| 4 | 3 | 7 | 0.1504384 |
| 5 | 5 | 9 | 0.1894560 |
| 6 | 6 | 12 | 0.2197643 |
| 7 | 7 | 13 | 0.3171921 |
| 8 | 7 | 12 | 0.3164954 |
| 9 | 2 | 12 | 0.9151135 |

That variable `h`, the result of running `hatvalues` on the fitted model object, measures the multivariable outlierness of the (`u`,`v`) combinations. The last one, you see, is quite a bit bigger than the others. This flags $u = 2, v = 12$ as a possible outlier.

Same kind of thing in SAS:

```
SAS> data uv;
```

```

SAS> infile 'leverage.txt' firstobs=2;
SAS> input u v;
SAS> z=1;
SAS>
SAS> proc reg;
SAS> model z=u v / influence;

```

The REG Procedure

Model: MODEL1

Dependent Variable: z

Number of Observations Read 9

Number of Observations Used 9

Analysis of Variance

| Source | DF | Sum of Squares | Mean Square | F Value | Pr > F |
|-----------------|----|----------------|-------------|---------|--------|
| Model | 2 | 0 | 0 | . | . |
| Error | 6 | 0 | 0 | | |
| Corrected Total | 8 | 0 | | | |

| | | | |
|----------------|---------|----------|---|
| Root MSE | 0 | R-Square | . |
| Dependent Mean | 1.00000 | Adj R-Sq | . |
| Coeff Var | 0 | | |

Parameter Estimates

| Variable | DF | Parameter Estimate | Standard Error | t Value | Pr > t |
|-----------|----|--------------------|----------------|---------|---------|
| Intercept | 1 | 1.00000 | 0 | Infty | <.0001 |
| u | 1 | 0 | 0 | . | . |
| v | 1 | 0 | 0 | . | . |

The REG Procedure

Model: MODEL1

Dependent Variable: z

Output Statistics

| Obs | Residual | RStudent | Hat Diag H | Cov Ratio | DFFITS | -----DFBETAS----- | |
|-----|----------|----------|------------|-----------|--------|-------------------|---|
| | | | | | | Intercept | u |
| 1 | 0 | . | 0.3654 | . | . | . | . |
| 2 | 0 | . | 0.2573 | . | . | . | . |
| 3 | 0 | . | 0.2689 | . | . | . | . |
| 4 | 0 | . | 0.1504 | . | . | . | . |
| 5 | 0 | . | 0.1895 | . | . | . | . |
| 6 | 0 | . | 0.2198 | . | . | . | . |
| 7 | 0 | . | 0.3172 | . | . | . | . |
| 8 | 0 | . | 0.3165 | . | . | . | . |
| 9 | 0 | . | 0.9151 | . | . | . | . |

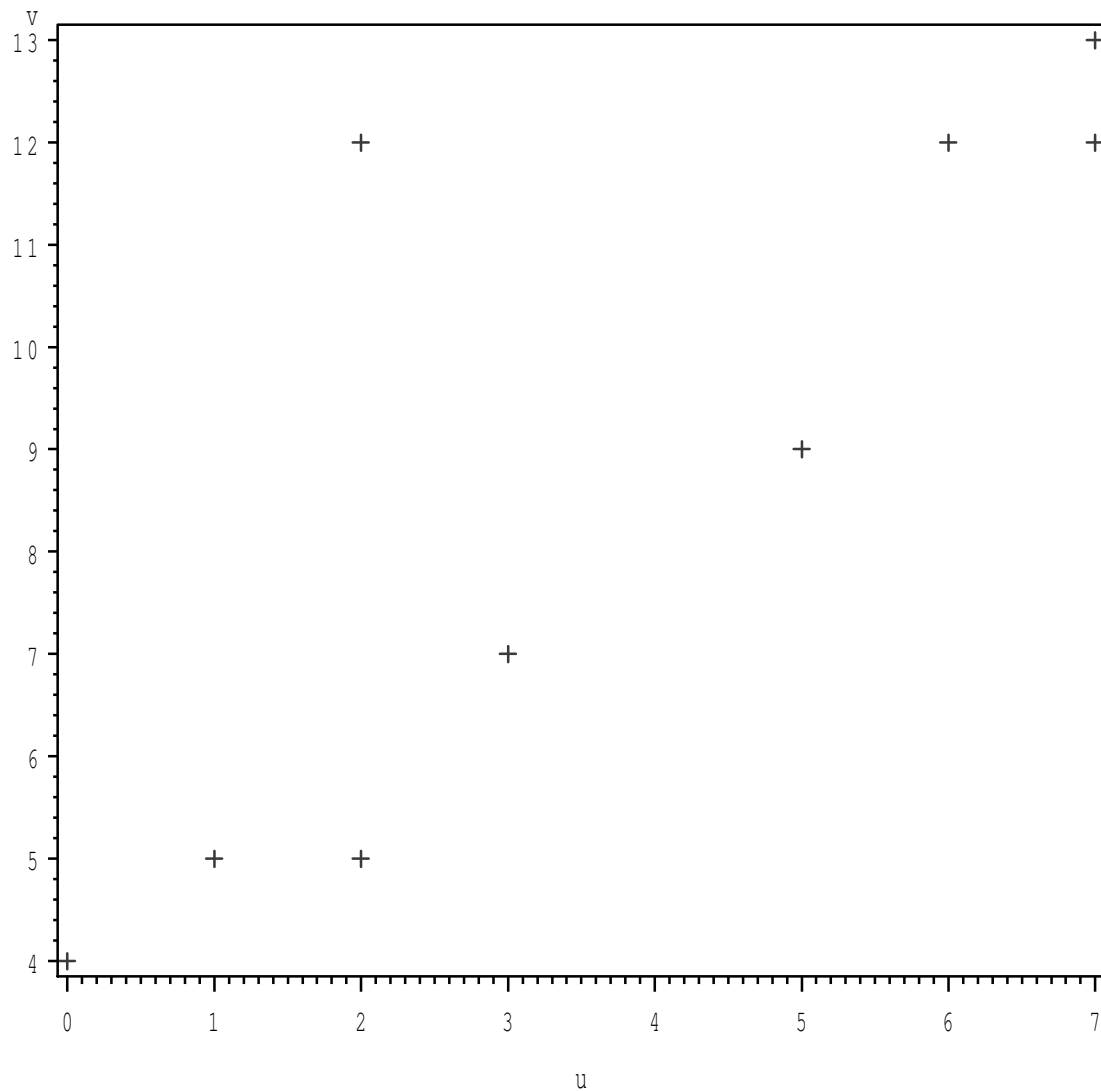
| | |
|-------------------------------|---|
| Sum of Residuals | 0 |
| Sum of Squared Residuals | 0 |
| Predicted Residual SS (PRESS) | 0 |

Most of the regression output is blank (because my fake “response” variable was constant), but the values in the `Hat Diag` column are as R produced, the clearly largest one being for the 9th observation, the one with $u = 2, v = 12$.

How big does one of these `h` values have to be for us to be worried? Some people like the standard $L = 3(k+1)/n$, where k is the number of variables you have, and n is the number of observations. In this case, $k = 2$ and $n = 9$, so $L = 3(3)/9 = 1$. Our observation 9 doesn’t quite reach this, but it is close.

In our case, we had two variables, so we could have plotted them against each other (either way around):

```
SAS> proc gplot;  
SAS>   plot v*u;
```

Most of the points lie close to a trend, except the one unusual one top left. With two variables, this is a nice way of spotting unusual observations, but with more than two, it doesn't work very well.⁶

Does this “leverage” idea apply to one variable as well? Yes, it does, and it also applies to groups, but we have to be a bit careful to make it all fly. Recall our data `mydata` where group 1 had an outlier in it? First, we have to create a new data set with a fake “response” variable in it, and just pick out group 1.⁷ Then

⁶Unless you're good at drawing scatter plots in high dimensions.

⁷Group 2 would work by the same idea.

we run that through a regression using our fake response as the response and our real variable as explanatory. The advantage, in SAS, of using a constant fake response variable is that most of the output is blank:

```
SAS> data mydata2;
SAS>   set mydata;
SAS>   z=1;
SAS>   if gp eq 1;
SAS>
SAS> proc print;
SAS>
SAS> proc reg;
SAS>   model z=x / influence;
```

| Obs | gp | x | z |
|-----|----|----|---|
| 1 | 1 | 3 | 1 |
| 2 | 1 | 5 | 1 |
| 3 | 1 | 17 | 1 |
| 4 | 1 | 16 | 1 |
| 5 | 1 | 11 | 1 |
| 6 | 1 | 44 | 1 |
| 7 | 1 | 10 | 1 |
| 8 | 1 | 7 | 1 |
| 9 | 1 | 12 | 1 |
| 10 | 1 | 19 | 1 |

The REG Procedure

Model: MODEL1

Dependent Variable: z

Number of Observations Read 10

Number of Observations Used 10

Analysis of Variance

| Source | DF | Sum of Squares | Mean Square | F Value | Pr > F |
|-----------------|----|----------------|-------------|---------|--------|
| Model | 1 | 0 | 0 | . | . |
| Error | 8 | 0 | 0 | | |
| Corrected Total | 9 | 0 | | | |

| | | | |
|----------------|---------|----------|---|
| Root MSE | 0 | R-Square | . |
| Dependent Mean | 1.00000 | Adj R-Sq | . |
| Coeff Var | 0 | | |

Parameter Estimates

| Variable | DF | Parameter Estimate | Standard Error | t Value | Pr > t |
|-----------|----|--------------------|----------------|---------|---------|
| Intercept | 1 | 1.00000 | 0 | Infty | <.0001 |

```
x          1          0          0          .          .
```

The REG Procedure

Model: MODEL1

Dependent Variable: z

| | | | Output Statistics | | | -----DFBETAS----- | |
|-------------------------------|----------|----------|-------------------|--------------|--------|-------------------|---|
| Obs | Residual | RStudent | Hat Diag
H | Cov
Ratio | DFFITS | Intercept | x |
| 1 | 0 | . | 0.2068 | . | . | . | . |
| 2 | 0 | . | 0.1726 | . | . | . | . |
| 3 | 0 | . | 0.1056 | . | . | . | . |
| 4 | 0 | . | 0.1021 | . | . | . | . |
| 5 | 0 | . | 0.1095 | . | . | . | . |
| 6 | 0 | . | 0.8203 | . | . | . | . |
| 7 | 0 | . | 0.1159 | . | . | . | . |
| 8 | 0 | . | 0.1450 | . | . | . | . |
| 9 | 0 | . | 0.1047 | . | . | . | . |
| 10 | 0 | . | 0.1174 | . | . | . | . |
| Sum of Residuals | | | 0 | | | | |
| Sum of Squared Residuals | | | 0 | | | | |
| Predicted Residual SS (PRESS) | | | 0 | | | | |

We should be worried by any h values bigger than $L = 3(1 + 1)/10 = 0.6$. The sixth one is. Looking back at the data, we see that this corresponds to the erroneous x value of 44.

Likewise, in R. First, we select just those x 's that belong to group 1:

```
R> attach(od)
R> is1=(g==1)
R> x1=x[is1]
R> x1

[1]  3  5 17 16 11 44 10  7 12 19
```

Then we make a fake response variable which is a string of 1's as long as $x1$ is, fit the regression, and list out the h -values (side by side with the original data). By the same logic as above, 44 is an outlier.

```
R> fake=rep(1,length(x1))
R> fake.lm=lm(fake~x1)
R> cbind(x1,hatvalues(fake.lm))

x1
1  3 0.2068399
2  5 0.1726406
3 17 0.1055574
4 16 0.1021046
```

```

5  11 0.1095035
6  44 0.8202894
7  10 0.1159158
8   7 0.1450181
9  12 0.1047353
10 19 0.1173956

```

11.3.4 Missing values

Introduction

Despite our best efforts, data can be unavailable for whatever reason. A person drops out of a study, something can happen to an experimental animal, a plant dies for a reason that's nothing to do with the study. It pays to design a study carefully so as to minimize the risk of getting missing data, but there is always that risk.

Missing at random

When you have missing values, the first question to ask is “*why* are they missing?”. The best, or at least “least bad” situation, is that the values are **missing at random**: that is, whether or not they are missing has nothing to do with anything else.

If the data values appear to be missing at random, things are usually all right. You can just throw away the observations with missing values, and act as if they never existed in the first place. For the kinds of models we consider here, things like regression or simple analysis of variance, this causes no problems.⁸

Many procedures in software will handle missing values automatically, so that you don't even need to worry about removing them. For our two-group data, let's change the apparently-wrong value 44 to an appropriate “missing” for our software:

```

SAS> data mydata;
SAS>   input gp x @@;
SAS>   cards;
SAS>   1 3 1 5 1 17 1 16 1 11 1 . 1 10 1 7 1 12 1 19
SAS>   2 10 2 13 2 15 2 14 2 17 2 9 2 8 2 16
SAS> ;
SAS>
SAS> proc means;

```

⁸More complex analyses of variance go more smoothly if you have equal numbers of observations in each group, but even then, things are usually OK.

```
SAS> var x;
SAS> class gp;
```

The MEANS Procedure

| Analysis Variable : x | | | | | | | |
|-----------------------|----|----------|---|------------|-----------|-----------|------------|
| | gp | N
Obs | N | Mean | Std Dev | Minimum | Maximum |
| ----- | 1 | 10 | 9 | 11.1111111 | 5.5100918 | 3.0000000 | 19.0000000 |
| ----- | 2 | 8 | 8 | 12.7500000 | 3.3700360 | 8.0000000 | 17.0000000 |
| ----- | | | | | | | |

In SAS, . means “missing”. You’ll see that `proc means` found 10 observations in group 1, but only used 9 of them to calculate the mean and SD (throwing the missing one away).

In R, NA is the code for “missing”:

```
R> od$x[6]=NA
R> aggregate(x~g,od,mean)
```

```
  g      x
1 1 11.11111
2 2 12.75000
```

giving the same results as SAS. But look at this. The first line picks out the `x`-values for group 1.

```
R> x1=od$x[is1]
R> x1

[1]  3  5 17 16 11 NA 10  7 12 19
```

This calculates the mean of all the values *including* the NA:

```
R> mean(x1)

[1] NA
```

and this removes any NAs before finding the mean:

```
R> mean(x1,na.rm=T)

[1] 11.11111
```

`aggregate` automatically removes any missing values before `mean` even gets its hands on them, explaining why the `aggregate` code above doesn’t require an `na.rm=T` on the end.

Informative missing values

All of the above assumes that observations are missing at random, that is, that whether or not an observation is missing does not depend on any other variable in the data set. But the missingness of an observation might be informative. Would this affect the results?

Take a look at this⁹ example. Suppose we observe these values for a variable `x`:

```
R> x=c(10,13,14,15,19,21,23)
R> mean(x)

[1] 16.42857
```

Now suppose there is a problem with the recording device, so that observations greater than 20 come out as missing:

```
R> x[x>20]=NA
R> x

[1] 10 13 14 15 19 NA NA
```

What does that do to the mean?

```
R> mean(x,na.rm=T)

[1] 14.2
```

The mean has gone down. Obviously, you say. But things might not be as obvious as this. For example, `x` might be positively correlated with some other variable `y`, and high values of `y` cannot be observed. This means that, on average, high values of `x` won't be observed either, and so the mean of `x` will again be pulled down from where it should be.

The point about informative, non-random missing values is rather like what happens when you draw a sample by some non-random method like convenience sampling. The observations that end up in your data set got there by some reason other than chance, and so there really isn't much you can conclude.

One thing you *can* do is to see whether the missing observations are different (in terms of the variables you *did* observe) from the complete ones. This is done by treating “missing” and “non-missing” as a grouping variable. In R you do something like this:

```
R> x=c(3,3,4,5,6,7,8)
R> y=c(10,10,11,12,NA,NA,12)
R> z=is.na(y)
R> tapply(x,z,mean)
```

⁹Admittedly rather contrived.

```
FALSE  TRUE
4.6    6.5
```

`z` is a logical variable that is `TRUE` when the observation has a missing `y` and `FALSE` otherwise. Therefore the mean of the `x`-values when `y` is missing is *higher* (6.5) than the mean of the `x`-values when `y` is present. This suggests that if those missing `y`'s had been observed, they would have been higher than the average `y`. This is supported by the correlation between `x` and `y`, where both are observed:

```
R> cor(x,y,use="complete.obs")
[1] 0.8439249
```

This is positive, so high values of `x` and `y` go together. Therefore the missing values of `y`, which went with high values of `x`, are probably higher than average themselves.

Imputation

Sometimes you have a lot of variables, and it seems very wasteful to throw away a whole observation just because one of them is missing. Is it possible to estimate what the missing value might have been? This is the point behind imputation, where you supply a guess as to what the missing value is, and then proceed as if that were data that you actually observed. Then you don't have to worry about whether the data were missing at random or not.

There is a number of methods of imputation, of varying sophistication:

The simplest is to replace the missing values by the means of the variables that are missing. In the case of our `y` above, we'd do this:

```
R> mean(y,na.rm=T)
[1] 11
```

and replace the missing values by 11. This might strike you as too small a value, since the missing `y`'s go with two of the bigger `x`'s.

Which leads us to suggest a regression idea: use the variable with missing values as the response, and predict what the missing values should be given their `x`'s.¹⁰

Here's how that goes for our data. SAS first (which is actually easier, or at least less fiddly, than R for this). We start by entering the data, with missing values, then do a regression to predict `y` (the variable with missing values) from `x` (which is not missing anything). The idea is that when you run `proc reg` this way, you obtain an output data set that has all the original variables, plus the

¹⁰If you have lots of variables, you use them all as explanatory variables in the regression.

predicted values (\hat{y}) saved in the variable `pred`. This does the predictions for *all* the x 's, including the ones that went with a missing y .

```
SAS> data xy;
SAS>   input x y @@;
SAS>   cards;
SAS>    3 10 3 10 4 11 5 12 6 . 7 . 8 12
SAS> ;
SAS>
SAS> proc reg;
SAS>   model y=x;
SAS>   output out=xy2 p=pred;
SAS>
SAS> proc print;
```

The REG Procedure

Model: MODEL1

Dependent Variable: y

Number of Observations Read 7

Number of Observations Used 5

Number of Observations with Missing Values 2

| Analysis of Variance | | | | | |
|----------------------|----------|----------------|-------------|---------|--------|
| Source | DF | Sum of Squares | Mean Square | F Value | Pr > F |
| Model | 1 | 2.84884 | 2.84884 | 7.42 | 0.0723 |
| Error | 3 | 1.15116 | 0.38372 | | |
| Corrected Total | 4 | 4.00000 | | | |
| Root MSE | 0.61945 | R-Square | 0.7122 | | |
| Dependent Mean | 11.00000 | Adj R-Sq | 0.6163 | | |
| Coeff Var | 5.63138 | | | | |

| Parameter Estimates | | | | | |
|---------------------|----|--------------------|----------------|---------|---------|
| Variable | DF | Parameter Estimate | Standard Error | t Value | Pr > t |
| Intercept | 1 | 9.12791 | 0.74082 | 12.32 | 0.0012 |
| x | 1 | 0.40698 | 0.14936 | 2.72 | 0.0723 |

| Obs | x | y | pred |
|-----|---|----|---------|
| 1 | 3 | 10 | 10.3488 |
| 2 | 3 | 10 | 10.3488 |
| 3 | 4 | 11 | 10.7558 |
| 4 | 5 | 12 | 11.1628 |
| 5 | 6 | . | 11.5698 |
| 6 | 7 | . | 11.9767 |
| 7 | 8 | 12 | 12.3837 |

You can see that the predictions for the missing values are 11.57 and 11.98,

In R, we again fit a regression predicting y from x . This successfully handles the missing data (by omitting the observations with missing y from the analysis entirely). You can verify this by looking at the `model` component of the fitted model object:

```
R> y.lm=lm(y~x)
R> y.lm$model
```

```
      y x
1 10  3
2 10  3
3 11  4
4 12  5
7 12  8
```

which shows that only the 5 observations with non-missing y are included in the regression.

Now we want to predict for the missing y values. To do that, we want `predict` to use the *original* x 's, not the ones left after the containing-missings observations have been removed. That's done by using the `newdata` thing on `predict`, as shown, making sure that we put the original x -values into a data frame:

```
R> y.p=predict(y.lm,newdata=data.frame(x=x))
R> cbind(x,y,y.p)
```

```
      x y      y.p
1  3 10 10.34884
2  3 10 10.34884
3  4 11 10.75581
4  5 12 11.16279
5  6 NA 11.56977
6  7 NA 11.97674
7  8 12 12.38372
```

The missing y -values are again predicted to be 11.57 and 11.98.

These values seem a whole lot more sensible than the 11 we would get by just predicting the mean of y . But they are *too good*: just because that's what the regression predicts them to be, doesn't mean the missing data were *exactly* those values. There will always be random errors attached to real data. So if we take our missing values to be exactly 11.56 and 11.98, we will probably make our model fit better than it really should, and therefore be over-optimistic about our conclusions.

The Wikipedia article http://en.wikipedia.org/wiki/Imputation_%28statistics%29 suggests a modification called “stochastic regression” where a random quantity is added to the predictions before we use them, to simulate random error. This is rather beyond our scope, however.

Also described in the Wikipedia article (but, by now, rather old-fashioned) is a method called *hot-deck imputation*. The idea is that you first sort your data set by a relevant variable that has no missing values, in our case `x`:

```
SAS> proc sort data=xy;
SAS>   by x;
SAS>
SAS> proc print;
```

```
Obs    x    y
  1     3    10
  2     3    10
  3     4    11
  4     5    12
  5     6     .
  6     7     .
  7     8    12
```

When you run into a missing `y`, you replace the missing values by the last non-missing `y`, in this case (both times) the 12 that goes with `x=5`. This is not obviously as good as the values from the regression method, but the random error part is, in some sense, built in.

11.4 Re-arranging data

11.4.1 Keeping, dropping and creating variables

Sometimes the step between reading in the original data and the stuff you want to use in your analysis is a large one. It can be a good idea to put the variables you want to keep into a new data set or data frame.

Let's take our cars data as an example. Suppose we just want to keep `car`, `weight`, `mpg`, and `cylinders` (dropping `hp` and `country`) and add a new variable `gpm` that is the reciprocal of `mpg`. In R, you might create a new data frame like this, being careful about the capitalization:

```
R> cars=read.csv("cars.csv",header=T)
R> to.keep=c("Car","Weight","MPG","Cylinders")
R> my.cars=cars[,to.keep]
R> my.cars$gpm=1/cars$MPG
R> my.cars
```

| | Car | Weight | MPG | Cylinders | gpm |
|---|----------------|--------|------|-----------|------------|
| 1 | Buick Skylark | 2.67 | 28.4 | 4 | 0.03521127 |
| 2 | Dodge Omni | 2.23 | 30.9 | 4 | 0.03236246 |
| 3 | Mercury Zephyr | 3.07 | 20.8 | 6 | 0.04807692 |
| 4 | Fiat Strada | 2.13 | 37.3 | 4 | 0.02680965 |

| | | | | | |
|----|---------------------------|------|------|---|------------|
| 5 | Peugeot 694 SL | 3.41 | 16.2 | 6 | 0.06172840 |
| 6 | VW Rabbit | 1.93 | 31.9 | 4 | 0.03134796 |
| 7 | Plymouth Horizon | 2.20 | 34.2 | 4 | 0.02923977 |
| 8 | Mazda GLC | 1.98 | 34.1 | 4 | 0.02932551 |
| 9 | Buick Estate Wagon | 4.36 | 16.9 | 8 | 0.05917160 |
| 10 | Audi 5000 | 2.83 | 20.3 | 5 | 0.04926108 |
| 11 | Chevy Malibu Wagon | 3.61 | 19.2 | 8 | 0.05208333 |
| 12 | Dodge Aspen | 3.62 | 18.6 | 6 | 0.05376344 |
| 13 | VW Dasher | 2.19 | 30.5 | 4 | 0.03278689 |
| 14 | Ford Mustang 4 | 2.59 | 26.5 | 4 | 0.03773585 |
| 15 | Dodge Colt | 1.92 | 35.1 | 4 | 0.02849003 |
| 16 | Datsun 810 | 2.82 | 22.0 | 6 | 0.04545455 |
| 17 | VW Scirocco | 1.99 | 31.5 | 4 | 0.03174603 |
| 18 | Chevy Citation | 2.60 | 28.8 | 6 | 0.03472222 |
| 19 | Olds Omega | 2.70 | 26.8 | 6 | 0.03731343 |
| 20 | Chrysler LeBaron Wagon | 3.94 | 18.5 | 8 | 0.05405405 |
| 21 | Datsun 510 | 2.30 | 27.2 | 4 | 0.03676471 |
| 22 | AMC Concord D/L | 3.41 | 18.1 | 6 | 0.05524862 |
| 23 | Buick Century Special | 3.38 | 20.6 | 6 | 0.04854369 |
| 24 | Saab 99 GLE | 2.80 | 21.6 | 4 | 0.04629630 |
| 25 | Datsun 210 | 2.02 | 31.8 | 4 | 0.03144654 |
| 26 | Ford LTD | 3.73 | 17.6 | 8 | 0.05681818 |
| 27 | Volvo 240 GL | 3.14 | 17.0 | 6 | 0.05882353 |
| 28 | Dodge St Regis | 3.83 | 18.2 | 8 | 0.05494505 |
| 29 | Toyota Corona | 2.56 | 27.5 | 4 | 0.03636364 |
| 30 | Chevette | 2.16 | 30.0 | 4 | 0.03333333 |
| 31 | Ford Mustang Ghia | 2.91 | 21.9 | 6 | 0.04566210 |
| 32 | AMC Spirit | 2.67 | 27.4 | 4 | 0.03649635 |
| 33 | Ford Country Squire Wagon | 4.05 | 15.5 | 8 | 0.06451613 |
| 34 | BMW 320i | 2.60 | 21.5 | 4 | 0.04651163 |
| 35 | Pontiac Phoenix | 2.56 | 33.5 | 4 | 0.02985075 |
| 36 | Honda Accord LX | 2.14 | 29.5 | 4 | 0.03389831 |
| 37 | Mercury Grand Marquis | 3.96 | 16.5 | 8 | 0.06060606 |
| 38 | Chevy Caprice Classic | 3.84 | 17.0 | 8 | 0.05882353 |

The strategy is: make a list of the variables you want to keep, select them out of the old data frame, and create a column in the new data frame using variable(s) from the old data frame, with liberal use of \$ signs to make it clear which data frame each variable is coming from.¹¹ If you wanted to, you could just create a variable called `gpm`, but that would be a “free-floating” variable by that name, not one attached to a data frame. It is tidier to attach things to a data frame, really, though I have to admit that I often don’t do it.

The strategy in SAS is really the same, done in a `data` step:

¹¹Attaching the data frames probably wouldn’t help, since there are two of them, and variable names may well, indeed do, exist in both.

```

SAS> data mycars;
SAS>   set 'cars';
SAS>   gpm=1/mpg;
SAS>   keep car weight mpg cylinders gpm;
SAS>
SAS> proc print;

```

| Obs | car | mpg | weight | cylinders | gpm |
|-----|---------------------------|------|--------|-----------|----------|
| 1 | Buick Skylark | 28.4 | 2.670 | 4 | 0.035211 |
| 2 | Dodge Omni | 30.9 | 2.230 | 4 | 0.032362 |
| 3 | Mercury Zephyr | 20.8 | 3.070 | 6 | 0.048077 |
| 4 | Fiat Strada | 37.3 | 2.130 | 4 | 0.026810 |
| 5 | Peugeot 694 SL | 16.2 | 3.410 | 6 | 0.061728 |
| 6 | VW Rabbit | 31.9 | 1.925 | 4 | 0.031348 |
| 7 | Plymouth Horizon | 34.2 | 2.200 | 4 | 0.029240 |
| 8 | Mazda GLC | 34.1 | 1.975 | 4 | 0.029326 |
| 9 | Buick Estate Wagon | 16.9 | 4.360 | 8 | 0.059172 |
| 10 | Audi 5000 | 20.3 | 2.830 | 5 | 0.049261 |
| 11 | Chevy Malibu Wagon | 19.2 | 3.605 | 8 | 0.052083 |
| 12 | Dodge Aspen | 18.6 | 3.620 | 6 | 0.053763 |
| 13 | VW Dasher | 30.5 | 2.190 | 4 | 0.032787 |
| 14 | Ford Mustang 4 | 26.5 | 2.585 | 4 | 0.037736 |
| 15 | Dodge Colt | 35.1 | 1.915 | 4 | 0.028490 |
| 16 | Datsun 810 | 22.0 | 2.815 | 6 | 0.045455 |
| 17 | VW Scirocco | 31.5 | 1.990 | 4 | 0.031746 |
| 18 | Chevy Citation | 28.8 | 2.595 | 6 | 0.034722 |
| 19 | Olds Omega | 26.8 | 2.700 | 6 | 0.037313 |
| 20 | Chrysler LeBaron Wagon | 18.5 | 3.940 | 8 | 0.054054 |
| 21 | Datsun 510 | 27.2 | 2.300 | 4 | 0.036765 |
| 22 | AMC Concord D/L | 18.1 | 3.410 | 6 | 0.055249 |
| 23 | Buick Century Special | 20.6 | 3.380 | 6 | 0.048544 |
| 24 | Saab 99 GLE | 21.6 | 2.795 | 4 | 0.046296 |
| 25 | Datsun 210 | 31.8 | 2.020 | 4 | 0.031447 |
| 26 | Ford LTD | 17.6 | 3.725 | 8 | 0.056818 |
| 27 | Volvo 240 GL | 17.0 | 3.140 | 6 | 0.058824 |
| 28 | Dodge St Regis | 18.2 | 3.830 | 8 | 0.054945 |
| 29 | Toyota Corona | 27.5 | 2.560 | 4 | 0.036364 |
| 30 | Chevette | 30.0 | 2.155 | 4 | 0.033333 |
| 31 | Ford Mustang Ghia | 21.9 | 2.910 | 6 | 0.045662 |
| 32 | AMC Spirit | 27.4 | 2.670 | 4 | 0.036496 |
| 33 | Ford Country Squire Wagon | 15.5 | 4.054 | 8 | 0.064516 |
| 34 | BMW 320i | 21.5 | 2.600 | 4 | 0.046512 |
| 35 | Pontiac Phoenix | 33.5 | 2.556 | 4 | 0.029851 |
| 36 | Honda Accord LX | 29.5 | 2.135 | 4 | 0.033898 |
| 37 | Mercury Grand Marquis | 16.5 | 3.955 | 8 | 0.060606 |
| 38 | Chevy Caprice Classic | 17.0 | 3.840 | 8 | 0.058824 |

I discovered that you need to explicitly **keep** the new variable. It doesn't matter whether you define it first, or **keep** it first and then define it.¹² It works either way.

Perhaps a better way to handle this, since we are keeping most of our variables and getting rid of only a few, is to **drop** the ones we don't want:

```
SAS> data mycars;
SAS>   set 'cars';
SAS>   drop hp country;
SAS>   gpm=1/mpg;
```

How about using R to drop **Horsepower** and **Country**?¹³ This looks a bit awkward, but we'll build up to it gradually. Let's start by making a vector of the variables to drop, and take a look at the variable names we have:

```
R> mydrops=c("Horsepower","Country")
R> mynames=names(cars)
R> mynames

[1] "Car"          "MPG"          "Weight"       "Cylinders"    "Horsepower"
[6] "Country"
```

R has a handy function called **%in%** that tells you whether the things in one vector are in another. It goes like this:

```
R> mynames %in% mydrops

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

This goes through the list of things in the first vector and asks “do they appear anywhere in the second vector?”. The fifth thing in **mynames** is **Horsepower**, and that *is* one of the things in **mydrops**. Likewise the sixth thing in **mynames**, **Country**. So these both send back a **TRUE**. The other things in **mynames** do *not* appear anywhere in **mydrops**, so they return a **FALSE**.

The good thing about this is that it returns a logical vector of **TRUE**s and **FALSE**s, so it can be used for selecting the columns you want to drop, or, equivalently, picking out the ones you want to keep. Given the foregoing, the next bit is not supposed to be too scary:

```
R> mycars=cars[!(mynames %in% mydrops)]
R> head(mycars)
```

| | Car | MPG | Weight | Cylinders |
|---|----------------|------|--------|-----------|
| 1 | Buick Skylark | 28.4 | 2.67 | 4 |
| 2 | Dodge Omni | 30.9 | 2.23 | 4 |
| 3 | Mercury Zephyr | 20.8 | 3.07 | 6 |
| 4 | Fiat Strada | 37.3 | 2.13 | 4 |

¹²Which, to me, makes no logical sense.

¹³These are the variable names in the R data frame.

```
5 Peugeot 694 SL 16.2 3.41 6
6 VW Rabbit 31.9 1.93 4
```

In English, that first statement says “select the columns for which it is *not* true that the column name is in my list of columns to drop”, or in plainer terms, “drop those columns that were in my list of columns to drop”.

R has another way of doing this called `subset`. This requires two things: a data frame, and an argument called `select` that says which columns to keep (or, with some adjustment, which ones to drop):

```
R> mycars=subset(cars,select=c(Car,MPG,Weight,Cylinders))
R> head(mycars)
```

| | Car | MPG | Weight | Cylinders |
|---|----------------|------|--------|-----------|
| 1 | Buick Skylark | 28.4 | 2.67 | 4 |
| 2 | Dodge Omni | 30.9 | 2.23 | 4 |
| 3 | Mercury Zephyr | 20.8 | 3.07 | 6 |
| 4 | Fiat Strada | 37.3 | 2.13 | 4 |
| 5 | Peugeot 694 SL | 16.2 | 3.41 | 6 |
| 6 | VW Rabbit | 31.9 | 1.93 | 4 |

Note that the column names here have *no quotes*.¹⁴

In R, negative indices on a vector mean “not these”, so you make “drop” happen like this:

```
R> mycars=subset(cars,select=-c(Horsepower,Country))
R> head(mycars)
```

| | Car | MPG | Weight | Cylinders |
|---|----------------|------|--------|-----------|
| 1 | Buick Skylark | 28.4 | 2.67 | 4 |
| 2 | Dodge Omni | 30.9 | 2.23 | 4 |
| 3 | Mercury Zephyr | 20.8 | 3.07 | 6 |
| 4 | Fiat Strada | 37.3 | 2.13 | 4 |
| 5 | Peugeot 694 SL | 16.2 | 3.41 | 6 |
| 6 | VW Rabbit | 31.9 | 1.93 | 4 |

with the same result as above.

11.4.2 Dropping observations

In Section 11.3, we learned about finding observations that are in error.¹⁵ Observations that are mistakes need to be removed from the data set that you analyze.

¹⁴Some programming languages call this “syntactic sugar”, something that’s there not because it’s logically needed (you could type the quotes), but because it makes the programmer’s life easier.

¹⁵But remember, outliers can be correct observations; they need to be checked first.

The easiest way to get rid of observations is to find out what row number they are in the data set. For example, suppose the Fiat Strada (observation #4) had something wrong with it, and we wanted to do the analysis without it. Then, in R we'd make a new data frame like this:

```
R> mycars=cars[-4,]
```

and in SAS we'd make a new data set like this:

```
SAS> data mycars;
SAS>   set 'cars';
SAS>   if _n_=4 then delete;
```

With several observations to delete, like say #4 and #6, we do this:

```
R> mycars=cars[-c(4,6),]
```

or this:

```
SAS> data mycars;
SAS>   set 'cars';
SAS>   if _n_=4 | _n_=6 then delete;
```

To delete some observations *and* drop some variables, do both of the above (in R or in SAS):

```
SAS> data mycars;
SAS>   set 'cars';
SAS>   keep car mpg weight;
SAS>   if _n_=4 | _n_=6 then delete;
SAS>
SAS> proc print;
```

| Obs | car | mpg | weight |
|-----|--------------------|------|--------|
| 1 | Buick Skylark | 28.4 | 2.670 |
| 2 | Dodge Omni | 30.9 | 2.230 |
| 3 | Mercury Zephyr | 20.8 | 3.070 |
| 4 | Peugeot 694 SL | 16.2 | 3.410 |
| 5 | Plymouth Horizon | 34.2 | 2.200 |
| 6 | Mazda GLC | 34.1 | 1.975 |
| 7 | Buick Estate Wagon | 16.9 | 4.360 |
| 8 | Audi 5000 | 20.3 | 2.830 |
| 9 | Chevy Malibu Wagon | 19.2 | 3.605 |
| 10 | Dodge Aspen | 18.6 | 3.620 |
| 11 | VW Dasher | 30.5 | 2.190 |
| 12 | Ford Mustang 4 | 26.5 | 2.585 |
| 13 | Dodge Colt | 35.1 | 1.915 |
| 14 | Datsun 810 | 22.0 | 2.815 |
| 15 | VW Scirocco | 31.5 | 1.990 |
| 16 | Chevy Citation | 28.8 | 2.595 |
| 17 | Olds Omega | 26.8 | 2.700 |

| | | | |
|----|---------------------------|------|-------|
| 18 | Chrysler LeBaron Wagon | 18.5 | 3.940 |
| 19 | Datsun 510 | 27.2 | 2.300 |
| 20 | AMC Concord D/L | 18.1 | 3.410 |
| 21 | Buick Century Special | 20.6 | 3.380 |
| 22 | Saab 99 GLE | 21.6 | 2.795 |
| 23 | Datsun 210 | 31.8 | 2.020 |
| 24 | Ford LTD | 17.6 | 3.725 |
| 25 | Volvo 240 GL | 17.0 | 3.140 |
| 26 | Dodge St Regis | 18.2 | 3.830 |
| 27 | Toyota Corona | 27.5 | 2.560 |
| 28 | Chevette | 30.0 | 2.155 |
| 29 | Ford Mustang Ghia | 21.9 | 2.910 |
| 30 | AMC Spirit | 27.4 | 2.670 |
| 31 | Ford Country Squire Wagon | 15.5 | 4.054 |
| 32 | BMW 320i | 21.5 | 2.600 |
| 33 | Pontiac Phoenix | 33.5 | 2.556 |
| 34 | Honda Accord LX | 29.5 | 2.135 |
| 35 | Mercury Grand Marquis | 16.5 | 3.955 |
| 36 | Chevy Caprice Classic | 17.0 | 3.840 |

```
R> keep.vars=c("Car","MPG","Weight")
R> drop.obs=c(4,6)
R> mycars=cars[-drop.obs,keep.vars]
R> mycars
```

| | Car | MPG | Weight |
|----|------------------------|------|--------|
| 1 | Buick Skylark | 28.4 | 2.67 |
| 2 | Dodge Omni | 30.9 | 2.23 |
| 3 | Mercury Zephyr | 20.8 | 3.07 |
| 5 | Peugeot 694 SL | 16.2 | 3.41 |
| 7 | Plymouth Horizon | 34.2 | 2.20 |
| 8 | Mazda GLC | 34.1 | 1.98 |
| 9 | Buick Estate Wagon | 16.9 | 4.36 |
| 10 | Audi 5000 | 20.3 | 2.83 |
| 11 | Chevy Malibu Wagon | 19.2 | 3.61 |
| 12 | Dodge Aspen | 18.6 | 3.62 |
| 13 | VW Dasher | 30.5 | 2.19 |
| 14 | Ford Mustang 4 | 26.5 | 2.59 |
| 15 | Dodge Colt | 35.1 | 1.92 |
| 16 | Datsun 810 | 22.0 | 2.82 |
| 17 | VW Scirocco | 31.5 | 1.99 |
| 18 | Chevy Citation | 28.8 | 2.60 |
| 19 | Olds Omega | 26.8 | 2.70 |
| 20 | Chrysler LeBaron Wagon | 18.5 | 3.94 |
| 21 | Datsun 510 | 27.2 | 2.30 |
| 22 | AMC Concord D/L | 18.1 | 3.41 |
| 23 | Buick Century Special | 20.6 | 3.38 |

| | | | |
|----|---------------------------|------|------|
| 24 | Saab 99 GLE | 21.6 | 2.80 |
| 25 | Datsun 210 | 31.8 | 2.02 |
| 26 | Ford LTD | 17.6 | 3.73 |
| 27 | Volvo 240 GL | 17.0 | 3.14 |
| 28 | Dodge St Regis | 18.2 | 3.83 |
| 29 | Toyota Corona | 27.5 | 2.56 |
| 30 | Chevette | 30.0 | 2.16 |
| 31 | Ford Mustang Ghia | 21.9 | 2.91 |
| 32 | AMC Spirit | 27.4 | 2.67 |
| 33 | Ford Country Squire Wagon | 15.5 | 4.05 |
| 34 | BMW 320i | 21.5 | 2.60 |
| 35 | Pontiac Phoenix | 33.5 | 2.56 |
| 36 | Honda Accord LX | 29.5 | 2.14 |
| 37 | Mercury Grand Marquis | 16.5 | 3.96 |
| 38 | Chevy Caprice Classic | 17.0 | 3.84 |

In each case, there are 36 cars and three variables left. Don't be confused by the R output, because the row "names"¹⁶ in the new data frame inherited from the original data frame. If you scroll back up, you'll find that rows 4 and 6 are missing, as they should be.

11.4.3 Reshaping data

Sometimes you find your data are the wrong format. Take a look at the oranges data, for example.

```
SAS> proc print data='oranges';
```

| Obs | row | age | a | b | c | d | e |
|-----|-----|------|-----|-----|-----|-----|-----|
| 1 | 1 | 118 | 30 | 30 | 30 | 33 | 32 |
| 2 | 2 | 484 | 51 | 58 | 49 | 69 | 62 |
| 3 | 3 | 664 | 75 | 87 | 81 | 111 | 112 |
| 4 | 4 | 1004 | 108 | 115 | 125 | 156 | 167 |
| 5 | 5 | 1231 | 115 | 120 | 142 | 172 | 179 |
| 6 | 6 | 1372 | 139 | 142 | 174 | 203 | 209 |
| 7 | 7 | 1582 | 140 | 145 | 177 | 203 | 214 |

This is fine if you want to find the mean circumference for each tree, averaged over times. Note the use of - to select consecutive variables (columns) in the data set:

```
SAS> proc means data='oranges';
```

```
SAS> var a--e;
```

The MEANS Procedure

| Variable | N | Mean | Std Dev | Minimum | Maximum |
|----------|---|------|---------|---------|---------|
|----------|---|------|---------|---------|---------|

¹⁶Actually numbers.

| | | | | | |
|---|---|-------------|------------|------------|-------------|
| a | 7 | 94.0000000 | 42.9806158 | 30.0000000 | 140.0000000 |
| b | 7 | 99.5714286 | 43.2930215 | 30.0000000 | 145.0000000 |
| c | 7 | 111.1428571 | 58.8598011 | 30.0000000 | 177.0000000 |
| d | 7 | 135.2857143 | 66.3242396 | 33.0000000 | 203.0000000 |
| e | 7 | 139.2857143 | 71.8974137 | 32.0000000 | 214.0000000 |

I doubt this is very meaningful, but there you go. What would probably make more sense is to find the average for each *time* over *trees*. This means, for SAS, that rows should be columns and vice versa. There is a `proc transpose` that handles this:

```
SAS> proc transpose data='oranges' name=tree prefix=age;
SAS>   var a--e;
SAS>
SAS> proc print;
```

| Obs | tree | age1 | age2 | age3 | age4 | age5 | age6 | age7 |
|-----|------|------|------|------|------|------|------|------|
| 1 | a | 30 | 51 | 75 | 108 | 115 | 139 | 140 |
| 2 | b | 30 | 58 | 87 | 115 | 120 | 142 | 145 |
| 3 | c | 30 | 49 | 81 | 125 | 142 | 174 | 177 |
| 4 | d | 33 | 69 | 111 | 156 | 172 | 203 | 203 |
| 5 | e | 32 | 62 | 112 | 167 | 179 | 209 | 214 |

There is a measure of consistency between the circumferences, if you look at one age at a time. So finding the means might be useful:

```
SAS> proc means;
SAS>   var age1--age7;
```

The MEANS Procedure

| Variable | N | Mean | Std Dev | Minimum | Maximum |
|----------|---|-------------|------------|-------------|-------------|
| age1 | 5 | 31.0000000 | 1.4142136 | 30.0000000 | 33.0000000 |
| age2 | 5 | 57.8000000 | 8.1670068 | 49.0000000 | 69.0000000 |
| age3 | 5 | 93.2000000 | 17.2394896 | 75.0000000 | 112.0000000 |
| age4 | 5 | 134.2000000 | 25.9364608 | 108.0000000 | 167.0000000 |
| age5 | 5 | 145.6000000 | 29.2284108 | 115.0000000 | 179.0000000 |
| age6 | 5 | 173.4000000 | 32.8374786 | 139.0000000 | 209.0000000 |
| age7 | 5 | 175.8000000 | 33.2821273 | 140.0000000 | 214.0000000 |

The means get larger with age, but perhaps just as interesting is that the *standard deviations* do as well.

This same kind of stuff is much easier in R, because you can treat a data frame as a matrix, and work directly with rows or columns as appropriate.

The original data frame is this:

```
R> oranges=read.table("oranges.txt",header=T)
R> oranges
```

| | row | ages | A | B | C | D | E |
|---|-----|------|-----|-----|-----|-----|-----|
| 1 | 1 | 118 | 30 | 30 | 30 | 33 | 32 |
| 2 | 2 | 484 | 51 | 58 | 49 | 69 | 62 |
| 3 | 3 | 664 | 75 | 87 | 81 | 111 | 112 |
| 4 | 4 | 1004 | 108 | 115 | 125 | 156 | 167 |
| 5 | 5 | 1231 | 115 | 120 | 142 | 172 | 179 |
| 6 | 6 | 1372 | 139 | 142 | 174 | 203 | 209 |
| 7 | 7 | 1582 | 140 | 145 | 177 | 203 | 214 |

To find means (or SDs) by rows or columns, use `apply`, with the second argument being 1 for rows and 2 for columns. Thus, the mean circumferences by tree (averaging over ages) are as below, noting that I only need columns 3 through 7 of the data frame because the first two tell me nothing about circumferences:

```
R> apply(oranges[,3:7],2,mean)

      A      B      C      D      E
94.00000 99.57143 111.14286 135.28571 139.28571
```

and the means by age (averaging over trees) are:

```
R> apply(oranges[,3:7],1,mean)

[1] 31.0 57.8 93.2 134.2 145.6 173.4 175.8
```

In SAS, we found that the SDs also increased with age. In R, it's just this:

```
R> apply(oranges[,3:7],1,sd)

[1] 1.414214 8.167007 17.239490 25.936461 29.228411 32.837479 33.282127
```

and you can verify that the numbers coming out of R and SAS are the same.

If you really want rows and columns interchanged (to do something else with), R's version of `proc transpose` is simply called `t()`:

```
R> t(oranges[,3:7])

[,1] [,2] [,3] [,4] [,5] [,6] [,7]
A   30   51   75  108  115  139  140
B   30   58   87  115  120  142  145
C   30   49   81  125  142  174  177
D   33   69  111  156  172  203  203
E   32   62  112  167  179  209  214
```

and if you want, you can add the actual ages as column names, like this:

```
R> oo=t(oranges[,3:7])
R> colnames(oo)=oranges$ages
R> oo
```

| | | | | | | | |
|---|-----|-----|-----|------|------|------|------|
| | 118 | 484 | 664 | 1004 | 1231 | 1372 | 1582 |
| A | 30 | 51 | 75 | 108 | 115 | 139 | 140 |
| B | 30 | 58 | 87 | 115 | 120 | 142 | 145 |
| C | 30 | 49 | 81 | 125 | 142 | 174 | 177 |
| D | 33 | 69 | 111 | 156 | 172 | 203 | 203 |
| E | 32 | 62 | 112 | 167 | 179 | 209 | 214 |

Just as a warning, this is an R **matrix**, which does not behave quite the same as a data frame. For example, you can't **attach** a matrix and use its column names as variables. If you really want a data frame, though, it's no trouble to make one:

```
R> data.frame(oo)

  X118 X484 X664 X1004 X1231 X1372 X1582
A   30   51   75   108   115   139   140
B   30   58   87   115   120   142   145
C   30   49   81   125   142   174   177
D   33   69  111   156   172   203   203
E   32   62  112   167   179   209   214
```

Note that the column names have gained an **X** on the front, which means that they would be legitimate variable names if you were to **attach** this data frame.

Let's go back to the original data frame:

```
R> oranges

  row ages   A   B   C   D   E
1    1 118  30  30  30  33  32
2    2 484  51  58  49  69  62
3    3 664  75  87  81 111 112
4    4 1004 108 115 125 156 167
5    5 1231 115 120 142 172 179
6    6 1372 139 142 174 203 209
7    7 1582 140 145 177 203 214
```

This layout was fine for plotting; we used data laid out this way in Section 9.4.2. It's called **wide format**, since several observations of circumference appear on each line, one for each tree.

For modelling, though,¹⁷ R tends to like all the values of the response variable in *one* column, with other variables (columns) to indicate which wide-format column the observation came from. This is called **long format**.

To convert between the two, R has a command called **reshape**. Its operation is a bit fiddly. I'm going to reshape the oranges data in a minute, but first, let me explain how it works.

¹⁷Like regression or ANOVA.

We need to specify three things (at a minimum): First, which columns of the data frame need to be made into one. These can be specified by name or number. Here, we know they are columns 3 through 7. Second, we need a name for what these columns are all measurements of. Here, they are all circumferences (of different trees), so let's call the new variable `circum`. Third, what makes the columns different. They are different here because they are different *trees*.

These are fed in as `varying`, `v.names` and `timevar`¹⁸ respectively. The last thing is whether you want a wide or a long data frame; here, we already have a wide one, so we want to make it long. This is done using `direction`.

The first few times you do this, it won't work because you've forgotten something. Pretty much guaranteed. It took me about four goes to get this one right:¹⁹

```
R> reshape(oranges, varying=3:7, v.names="circum", timevar="tree", direction="long")
```

| | row | ages | tree | circum | id |
|-----|-----|------|------|--------|----|
| 1.1 | 1 | 118 | 1 | 30 | 1 |
| 2.1 | 2 | 484 | 1 | 51 | 2 |
| 3.1 | 3 | 664 | 1 | 75 | 3 |
| 4.1 | 4 | 1004 | 1 | 108 | 4 |
| 5.1 | 5 | 1231 | 1 | 115 | 5 |
| 6.1 | 6 | 1372 | 1 | 139 | 6 |
| 7.1 | 7 | 1582 | 1 | 140 | 7 |
| 1.2 | 1 | 118 | 2 | 30 | 1 |
| 2.2 | 2 | 484 | 2 | 58 | 2 |
| 3.2 | 3 | 664 | 2 | 87 | 3 |
| 4.2 | 4 | 1004 | 2 | 115 | 4 |
| 5.2 | 5 | 1231 | 2 | 120 | 5 |
| 6.2 | 6 | 1372 | 2 | 142 | 6 |
| 7.2 | 7 | 1582 | 2 | 145 | 7 |
| 1.3 | 1 | 118 | 3 | 30 | 1 |
| 2.3 | 2 | 484 | 3 | 49 | 2 |
| 3.3 | 3 | 664 | 3 | 81 | 3 |
| 4.3 | 4 | 1004 | 3 | 125 | 4 |
| 5.3 | 5 | 1231 | 3 | 142 | 5 |
| 6.3 | 6 | 1372 | 3 | 174 | 6 |
| 7.3 | 7 | 1582 | 3 | 177 | 7 |
| 1.4 | 1 | 118 | 4 | 33 | 1 |
| 2.4 | 2 | 484 | 4 | 69 | 2 |
| 3.4 | 3 | 664 | 4 | 111 | 3 |
| 4.4 | 4 | 1004 | 4 | 156 | 4 |
| 5.4 | 5 | 1231 | 4 | 172 | 5 |
| 6.4 | 6 | 1372 | 4 | 203 | 6 |

¹⁸It *isn't* time here, but it often is, which is why it's called this.

¹⁹And, I didn't like it when I had finished, so I had to do it again.

| | | | | | |
|-----|---|------|---|-----|---|
| 7.4 | 7 | 1582 | 4 | 203 | 7 |
| 1.5 | 1 | 118 | 5 | 32 | 1 |
| 2.5 | 2 | 484 | 5 | 62 | 2 |
| 3.5 | 3 | 664 | 5 | 112 | 3 |
| 4.5 | 4 | 1004 | 5 | 167 | 4 |
| 5.5 | 5 | 1231 | 5 | 179 | 5 |
| 6.5 | 6 | 1372 | 5 | 209 | 6 |
| 7.5 | 7 | 1582 | 5 | 214 | 7 |

Now you can see why this is called “long format”!

Let’s take a look at the things we have. The first seven values are the circumferences at the various **ages** of tree A. Then follow the seven circumferences of tree B. And so on.

In the output data frame, therefore, the variables **row** (and also **id**) represent which of the seven **ages** we’re looking at. These actual ages appear in the column **ages**. Then we need to identify which **tree** we’re talking about, which is in the column **tree**. The letters have gotten changed to numbers. Then, in **circum**, the circumference of that tree at that time. There were $5 \times 7 = 35$ circumferences in the original wide data frame,²⁰ so there are 35 *rows* in the long data frame, since the long data frame has only one circumference per line.

Let’s suppose we had given our wide-format data frame columns different names, like this:

```
R> names(oranges)=c("row","ages","circum.A","circum.B","circum.C","circum.D","circum.E")
R> oranges
```

| | row | ages | circum.A | circum.B | circum.C | circum.D | circum.E |
|---|-----|------|----------|----------|----------|----------|----------|
| 1 | 1 | 118 | 30 | 30 | 30 | 33 | 32 |
| 2 | 2 | 484 | 51 | 58 | 49 | 69 | 62 |
| 3 | 3 | 664 | 75 | 87 | 81 | 111 | 112 |
| 4 | 4 | 1004 | 108 | 115 | 125 | 156 | 167 |
| 5 | 5 | 1231 | 115 | 120 | 142 | 172 | 179 |
| 6 | 6 | 1372 | 139 | 142 | 174 | 203 | 209 |
| 7 | 7 | 1582 | 140 | 145 | 177 | 203 | 214 |

The columns that are going to be combined into one have two-part names: what they are a measurement of (circumference), and which tree that thing was measured on (separated, in this case, by a dot, which seems to work all right). Then, when you ask for a **reshape**, it’s simpler because you don’t have to specify **v.names**:

```
R> oranges.long=reshape(oranges,varying=3:7,timevar="tree",direction="long")
R> head(oranges.long,n=10)
```

| | row | ages | tree | circum | id |
|-----|-----|------|------|--------|----|
| 1.A | 1 | 118 | A | 30 | 1 |

²⁰5 trees at 7 times.

| | | | | | |
|-----|---|------|---|-----|---|
| 2.A | 2 | 484 | A | 51 | 2 |
| 3.A | 3 | 664 | A | 75 | 3 |
| 4.A | 4 | 1004 | A | 108 | 4 |
| 5.A | 5 | 1231 | A | 115 | 5 |
| 6.A | 6 | 1372 | A | 139 | 6 |
| 7.A | 7 | 1582 | A | 140 | 7 |
| 1.B | 1 | 118 | B | 30 | 1 |
| 2.B | 2 | 484 | B | 58 | 2 |
| 3.B | 3 | 664 | B | 87 | 3 |

This time, R split up the column names, so that we now have a column called `circum`²¹ and the column called `tree` has the letter designations of the trees.

11.4.4 Matched pairs and reshape in reverse

In R

A common experimental design is matched pairs, where, for example, you have before-and-after measurements on the *same people*. Let's say they were learning to speak Spanish: you might test them at the beginning of the course, to see how much they knew initially, and then test them again at the end, to see how much they learned during the course.

Logically, the right way to test data like this is to take differences, after minus before, and then run a one-sample *t*-test, using a null hypothesis that the mean difference is equal to zero, against the alternative that the mean difference is greater than zero.²²

Here's another example. A tire manufacturer was interested in how well a particular brand of tire helped drivers to stop on wet pavement (as compared to dry pavement). Tires of that brand were tested on 10 different models of car, once under wet conditions and once under dry conditions. The stopping distance, in feet, was measured each time. Since each car produced two measurements, we have matched pairs. The data, as they come to us, look like this:

```
R> braking=read.table("braking.txt",header=T)
```

```
R> braking
```

| | car | cond | dist |
|---|-----|------|------|
| 1 | 1 | dry | 150 |
| 2 | 1 | wet | 201 |
| 3 | 2 | dry | 147 |
| 4 | 2 | wet | 220 |
| 5 | 3 | dry | 136 |

²¹Without explicitly telling R to call it that.

²²Depending on the circumstances, you might use a two-sided alternative, but in this case you'd expect test scores to go *up* if a student has learned some Spanish.

```

6    3  wet  192
7    4  dry  134
8    4  wet  146
9    5  dry  130
10   5  wet  182
11   6  dry  134
12   6  wet  173
13   7  dry  134
14   7  wet  202
15   8  dry  128
16   8  wet  180
17   9  dry  136
18   9  wet  192
19  10  dry  158
20  10  wet  206

```

There's only one column of stopping distances, so this is long format. Usually R likes long format, but for the particular case of doing a paired *t*-test, it wants *wide* format. So this time, we have to use `reshape` to make a wide-format data frame out of a long-format one. What we are looking for is one line for each `car`, with both the `wet` and the `dry` stopping distance on it.

This way around is actually easier to contemplate. The long data frame needs to have three things in it: one, a response variable (here `dist`) that we are going to get several of on a row, two, a variable that describes what the different values of the response are going to be on each line (here `cond`), and three, a variable that says what each line of the wide data frame represents (here `car`). Add in `direction="wide"`, and you get this:

```

R> braking.wide=reshape(braking,v.names="dist",timevar="cond",
R>   idvar="car",direction="wide")
R> braking.wide

```

```

      car dist.dry dist.wet
1      1      150      201
3      2      147      220
5      3      136      192
7      4      134      146
9      5      130      182
11     6      134      173
13     7      134      202
15     8      128      180
17     9      136      192
19    10      158      206

```

R even constructs suitable column names for you. You can see that the two braking distances for car 1 are on the first line of `braking.wide`, first the dry-conditions one and then the wet-conditions one. And so on for the other

`cars`.

We can use `aggregate` on the long data frame to get the means by condition:

```
R> aggregate(dist~cond,data=braking,mean)

   cond  dist
1  dry 138.7
2  wet 189.4
```

Or we can just find the column means of the wide data frame:

```
R> apply(braking.wide[,-1],2,mean)

dist.dry dist.wet
  138.7    189.4
```

We get the same answer both ways. Note that I did the calculation without the first column,²³ since it doesn't make much sense to calculate the mean of the car numbers!

As a point of strategy, the wet-conditions braking distance mean is a lot bigger than the dry one, so that a hypothesis test is kind of pointless. Of more interest is a confidence interval for the mean difference, so that we have some sense of how much longer the stopping distance is in the wet than in the dry for a "typical" car²⁴ using this brand of tires. The P-value on the output from `t.test` will be, we can take for granted, very small. Note `paired=T` inside `t.test`. The calculation of `diff` is for later.

```
R> attach(braking.wide)
R> t.test(dist.dry,dist.wet,paired=T)
R> diff=dist.wet-dist.dry
R> detach(braking.wide)

      Paired t-test

data:  dist.dry and dist.wet
t = -9.6233, df = 9, p-value = 4.921e-06
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -62.61808 -38.78192
sample estimates:
mean of the differences
      -50.7
```

Since we fed in `dist.dry` first, the confidence interval contains negative values.²⁵

²³A negative index means "everything but this one". `braking.wide[,2:3]` would also have worked.

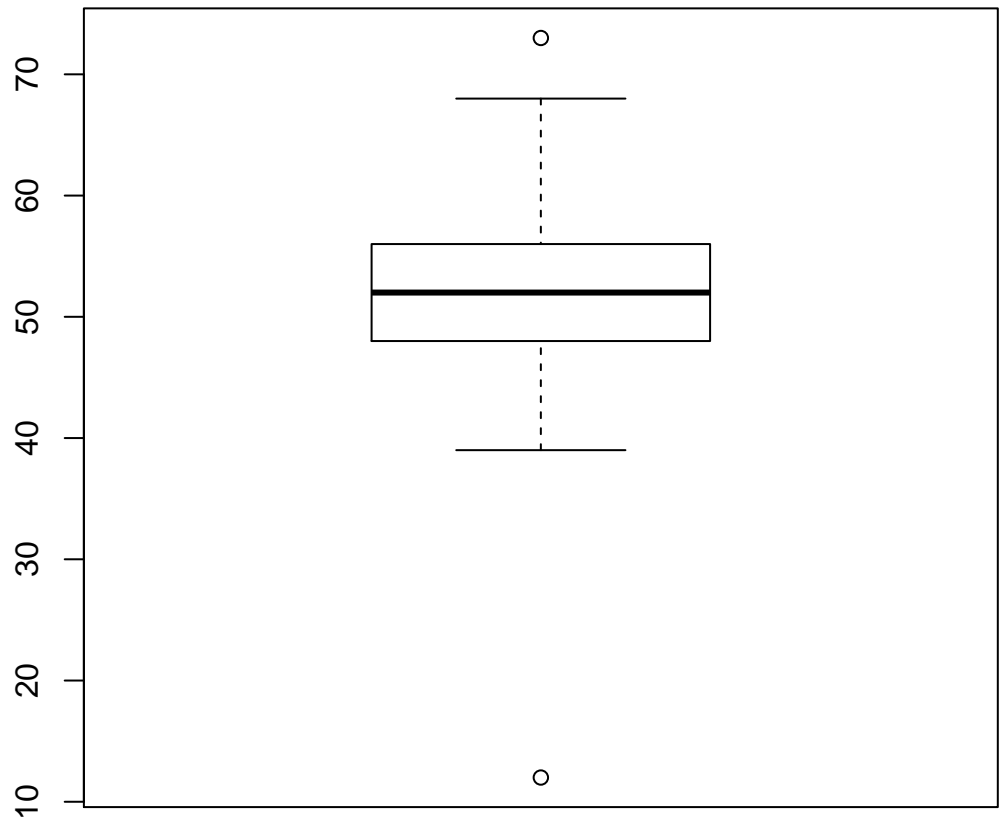
²⁴We're taking the attitude that these 10 cars are a random sample of "all possible cars".

²⁵Dry minus wet.

So, according to these data, the stopping distance for this tire is (likely) between about 40 and 60 feet longer in the wet than in the dry.

This is a small sample, and also the differences appear to be rather variable. What we probably should have done first is to take a look at something like a boxplot of the differences, to see whether there are any outliers:

```
R> boxplot(diff)
```



Oh dear. There's a serious outlier at the bottom, and apparently another at the

top. The actual box of the boxplot is quite compressed, though, so the values in the middle of the distribution are close together. It's just that the extreme values are more extreme than you would otherwise expect.

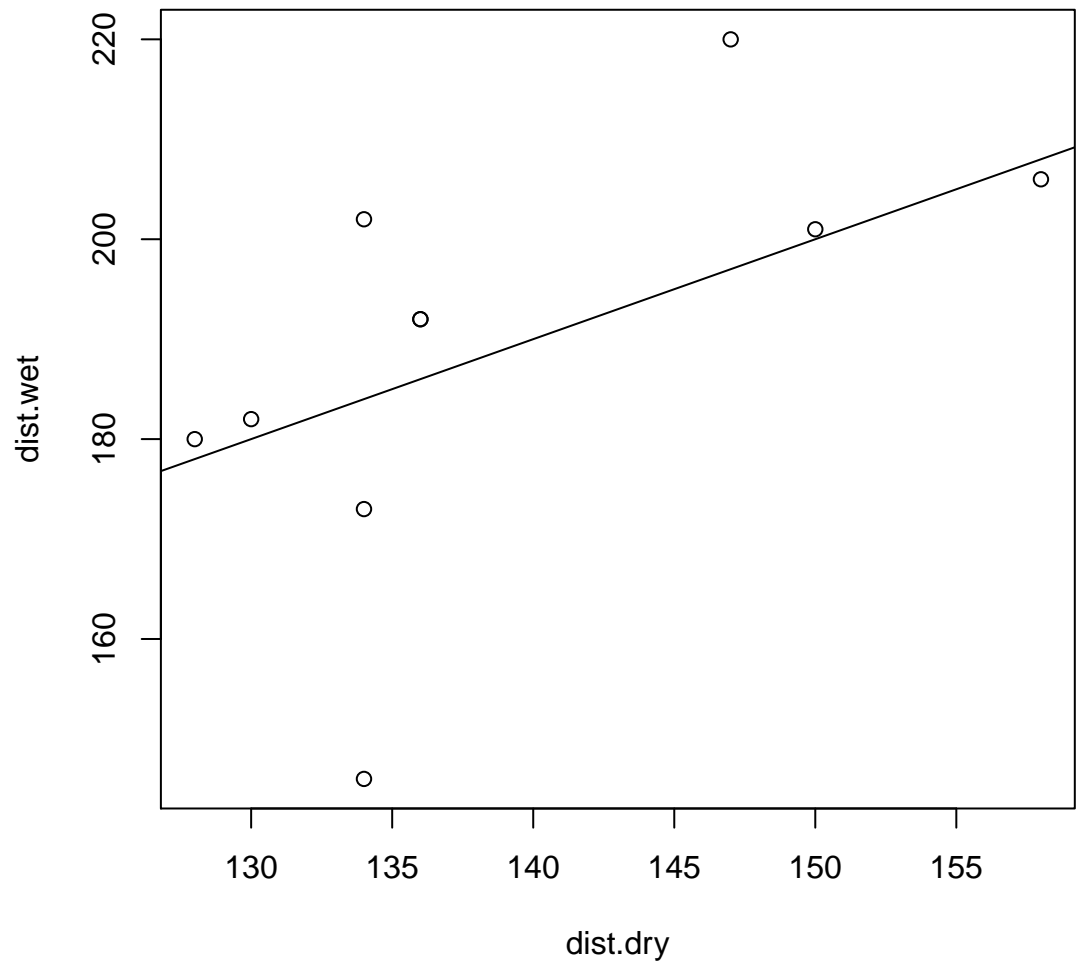
So I would urge caution in using the confidence interval, because the story (based on our small sample) seems to be that some models of car stop much better or worse in the wet, compared to how they stop in the dry, than you would expect.

You can always recommend collecting more data (which would help here), but another thing that could be done is to have several runs, both in the wet and the dry, which would help determine whether those unusually long or short stopping distances in the wet should be trusted or not.

I made another picture, just because I thought it was interesting. What I did was to plot the two stopping distances against each other for each of the 10 models of car. The middle of our confidence interval for the difference wet minus dry is about 50, so that on average you'd expect `dist.wet` to be `dist.dry` plus 50. That is, the points should be reasonably close to the line $y = 50 + x$. Let's plot that on the graph too. In `abline`, `a` is the intercept and `b` the slope. Note that you can feed `plot`²⁶ a two-column matrix, or two columns of a data frame, and it'll take the first column as x and the second as y .

```
R> plot(braking.wide[, -1])
R> abline(a=50, b=1)
```

²⁶Or lines or points or text.



You can see how the point at the bottom of the plot really stands out as unusual. Its `dist.wet` is about 150, when we would have expected to see about 180. This is the big outlier on the boxplot.

In SAS

Now, let's go back to the beginning again and try to do the analysis in SAS.

The actual running of the *t*-test is pretty straightforward: we need a variable

with the dry stopping distances and another with the wet ones. The difficult part is reading the data in, and there's another **data** step trick that we will use. As a reminder, here's what the data file looks like:

```
car cond dist
1 dry 150
1 wet 201
2 dry 147
2 wet 220
3 dry 136
3 wet 192
4 dry 134
4 wet 146
5 dry 130
5 wet 182
6 dry 134
6 wet 173
7 dry 134
7 wet 202
8 dry 128
8 wet 180
9 dry 136
9 wet 192
10 dry 158
10 wet 206
```

The key observation here is that to make our wide-format data set, we actually need to take values from *two* lines of the input file. So what we have to read in is something like this: "car number, word 'dry', dry distance, newline, car number again, word 'wet', wet distance". We persuade SAS of this by something like the following:

```
SAS> data braking;
SAS>   infile "braking.txt" firstobs=2;
SAS>   input car cond $ distdry / car2 cond2 $ distwet;
SAS>   drop car2 cond cond2;
SAS>
SAS> proc print;
```

| Obs | car | distdry | distwet |
|-----|-----|---------|---------|
| 1 | 1 | 150 | 201 |
| 2 | 2 | 147 | 220 |
| 3 | 3 | 136 | 192 |
| 4 | 4 | 134 | 146 |
| 5 | 5 | 130 | 182 |
| 6 | 6 | 134 | 173 |
| 7 | 7 | 134 | 202 |
| 8 | 8 | 128 | 180 |

| | | | |
|----|----|-----|-----|
| 9 | 9 | 136 | 192 |
| 10 | 10 | 158 | 206 |

Success! And now we have to run this through `proc ttest`:

```
SAS> proc ttest;
SAS>   paired distdry*distwet;
```

The TTEST Procedure

Difference: distdry - distwet

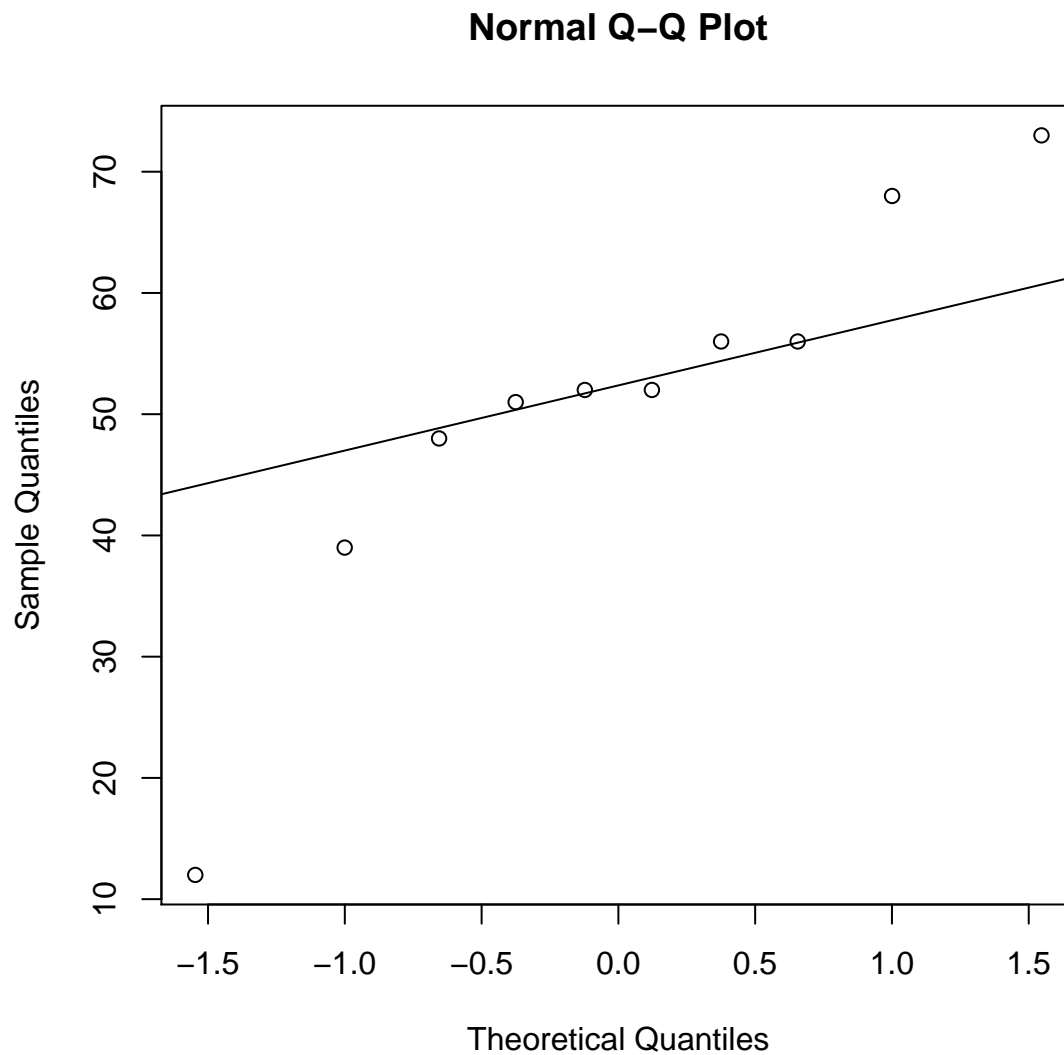
| N | Mean | Std Dev | Std Err | Minimum | Maximum |
|----------|-------------------|---------|-----------------|----------|----------|
| 10 | -50.7000 | 16.6603 | 5.2685 | -73.0000 | -12.0000 |
| Mean | 95% CL Mean | Std Dev | 95% CL Std Dev | | |
| -50.7000 | -62.6181 -38.7819 | 16.6603 | 11.4596 30.4153 | | |
| DF | t Value | Pr > t | | | |
| 9 | -9.62 | <.0001 | | | |

If you can get the HTML output, as well as the printed output above, you'll get a bunch of graphs, which are:

- A histogram (with kernel density estimator) and boxplot for the differences. These support what we saw before: the differences are very compact, but with some unusually large or small values given that.
- A plot with lines joining the wet and dry distances for each car. If the lines are more or less parallel, that means there's a more or less constant difference between the distances. This is not too bad, except for the one car with a wet-conditions stopping distance less than 150, which is the one that caught our attention before.
- A scatterplot of wet and dry distances against each other. This has a line marking wet equals dry; the points are top left of this, indicating that wet is greater than dry.
- A normal quantile plot of the differences. These, surprisingly, do not look too wildly non-normal. But then, there are only ten differences. And see later.

R's normal quantile plot tells apparently a very different story:

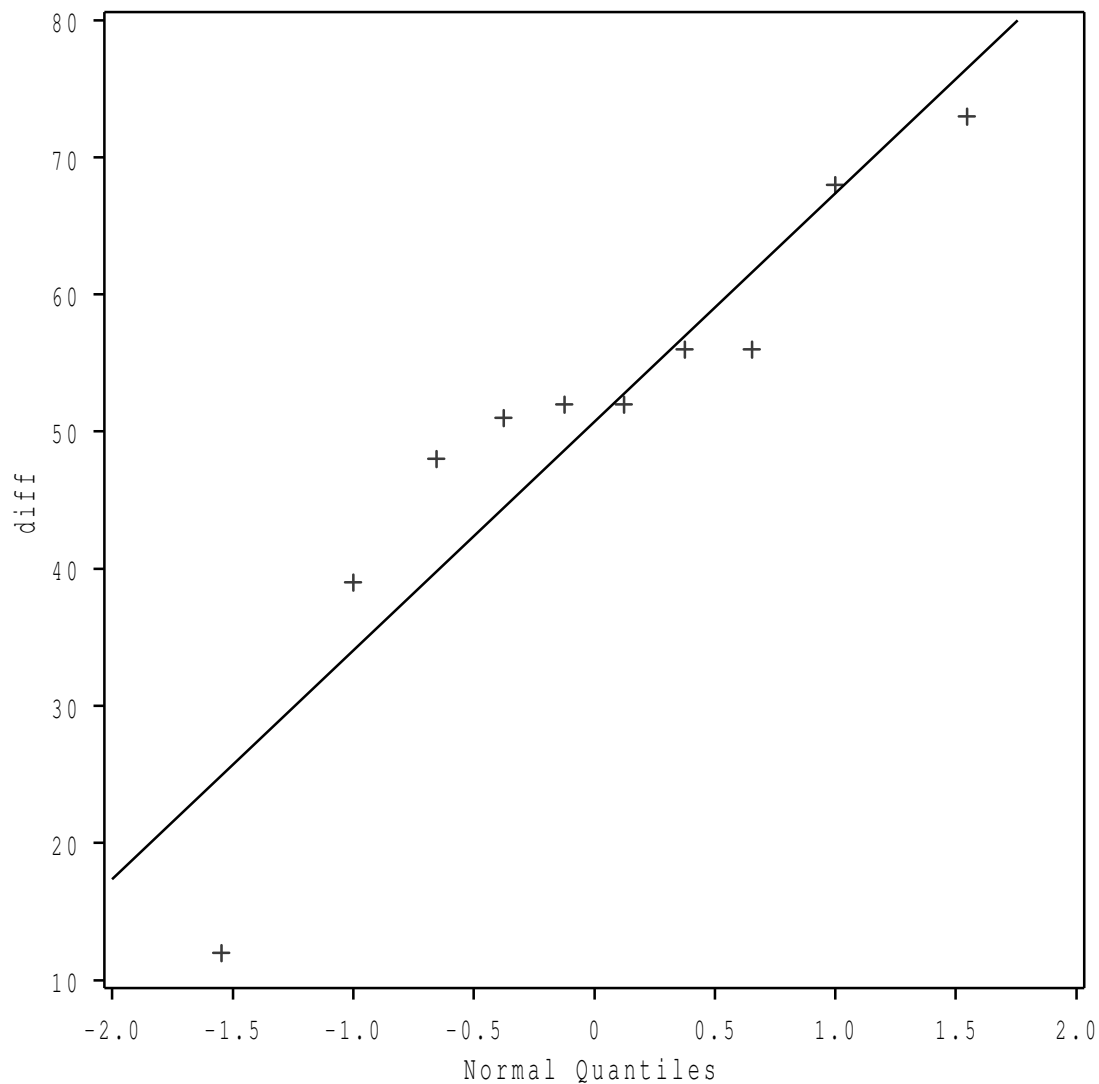
```
R> qqnorm(diff)
R> qqline(diff)
```



The point bottom left looks way too small to be part of a normal distribution. Let's do SAS's normal quantile plot on its own. First, though, we need to calculate the differences, which means defining a new data set:

```
SAS> data fred;  
SAS>   set braking;  
SAS>   diff=distwet-distdry;  
SAS>  
SAS> proc univariate noprint;
```

```
SAS> qqplot diff / normal(mu=50.7 sigma=16.67 color=black);
```



The patterns of points are the same, but the lines are very different. SAS uses the sample mean and SD to determine where the line goes²⁷, but when there are outliers, as there are here, the sample SD will be inflated, and outlying values will not look especially outlying. That appears to have happened here; on the SAS plot, the value 12 is too small, but not overwhelmingly too small.

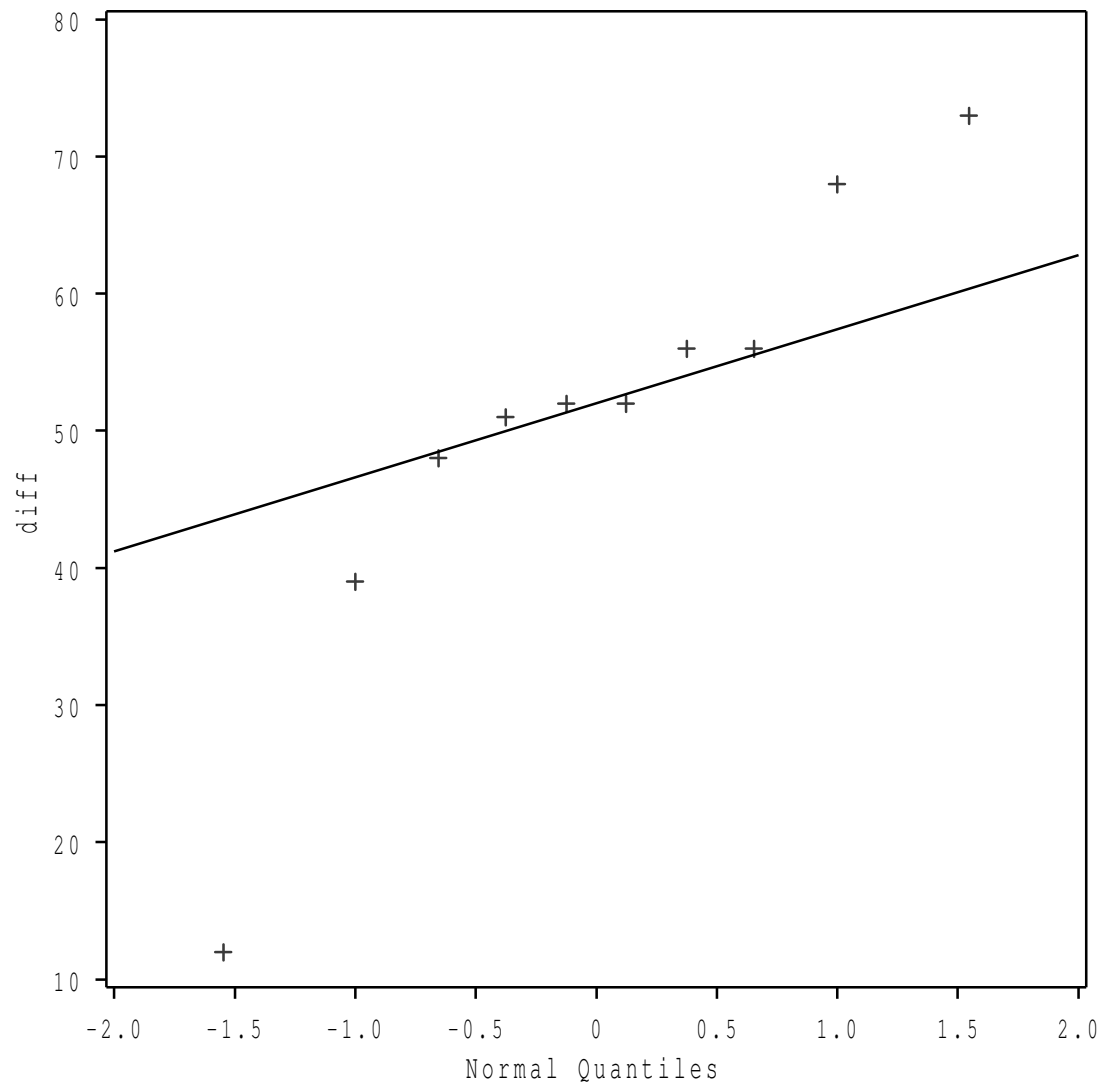
²⁷Actually, you can use whatever mean and SD you like, but to get the line, you have to specify *some* mean and SD, and I can't easily think of any better values to use than sample mean and SD.

R, instead of using the mean and SD, puts the line through observed and expected Q1 and Q3. Because these are not affected by outliers, any outliers you have in your data set (like 12) will end up a long way off the line. In fact, the line barely even seems to go through the points, and the conclusion to draw from this is that the data are not normal in the first place.

Another way to estimate the mean and SD is to recall that the quartiles for the standard normal are ± 0.67 .²⁸ So the interquartile range for a normal distribution with SD σ is 1.34σ . For our data, $IQR = 7.25$, which gives an estimated σ of 5.41, much smaller than the sample SD of 16.67. The corresponding estimate of μ ought to be the sample median, which is 52. Let's put these onto SAS's normal quantile plot:

```
SAS> proc univariate noprint;  
SAS> qqplot diff / normal(mu=52 sigma=5.41 color=black);
```

²⁸Check this from a normal table.



This is the same line `R` plotted with `qqline`. It shows that the values in the middle are passably normal, but the highest two values are a bit too high, and the lowest value is *way* too low.