R Fundamentals and Programming Techniques

Thomas Lumley

R Core Development Team
and UW Dept of Biostatistics

Birmingham — 2006–2–27/28

What are R and S-PLUS?

- R is a free implementation of a dialect of the S language, the statistics and graphics environment for which John Chambers won the ACM Software Systems award. S was consciously designed to blur the distinction between users and programmers. S-PLUS is a commercial system (Insightful Co) based on Bell Labs' S.

- S is a system for interactive data analysis. It has always been designed with interactive use in mind.

- S is a high-level programming language, with similarities to Scheme and Python. It is a good system for rapid development of statistical applications. For example, the survey package was developed by one person, part time, and is about 1.5% the size of the Census Bureau's VPLX package (which does less).

Why not S?

R (and S) are accused of being slow, memory-hungry, and able to handle only small data sets.

This is completely true.

Fortunately, computers are fast and have lots of memory. Data sets with a few tens of thousands of observations can be handled in 256Mb of memory, and quite large data sets with 1Gb of memory. Workstations with 32Gb or more to handle millions of observations are still expensive (but in a few years Moore's Law should catch up).

Tools for interfacing R with databases allow very large data sets, but this isn't transparent to the user.

Why not R?

21 CFR 11, Basel 2, and similar regulatory guidelines require tedious effort to document and verify things. Doing this for another system is unappealing (although not intrinsically harder than for commercial software)

R has no GUI and no commercial hand-holding services. S-PLUS does. S-PLUS also has a number of specialised add-ons (I would buy S-PLUS just to run S+SeqTrial if I were designing clinical trials).

How similar are R and S-PLUS?

- For basic command-line data analysis they are very similar

- Most programs written in one dialect can be translated straightforwardly to the other

- Most large programs will need some translation

- R has a very successful package system for distributing code and data.

R GUIs

Not GUIs for statistics, but for files/scripts/windows etc

- built-in: Windows, Mac

- cross-platform: JGR (`http://www.rosudo.org/JGR`), Emacs/ESS (`http://ess.r-projet.org`).

Outline

- Reading data
- Simple data summaries
- Graphics
- Scripts, Transcripts, ESS.
- Stratified data summaries
- Sweave
- More on functions (example: bootstrap)
- Debugging and optimization
- A little on objects
- Regression models: `lm`, `glm`, `coxph`
- Packages

Reading data

- Text files

- Stata datasets

- Web pages

- (Databases)

Much more information is in the Data Import/Export manual.

Reading text data

The easiest format has variable names in the first row

```
case     id gender    deg yrdeg field startyr year    rank admin
   1      1       F Other     92 Other       95    95 Assist       0
   2      2       M Other     91 Other       94    94 Assist       0
   3      2       M Other     91 Other       94    95 Assist       0
   4      4       M   PhD     96 Other       95    95 Assist       0
```

and fields separated by spaces. In R, use

```
salary <- read.table("salary.txt", header=TRUE)
```

to read the data from the file salary.txt into the data frame salary.

Syntax notes

- Spaces in commands don't matter (except for readability), but Capitalisation Does Matter.

- `TRUE` (and `FALSE`) are logical constants

- Unlike many systems, R does not distinguish between commands that do something and commands that compute a value. Everything is a function: ie returns a value.

- Arguments to functions can be named (`header=TRUE`) or unnamed (`"salary.txt"`)

- A whole data set (called a `data frame` is stored in a variable (`salary`), so more than one dataset can be available at the same time.

Reading text data

Sometimes columns are separated by commas (or tabs)

```
Ozone,Solar.R,Wind,Temp,Month,Day
41,190,7.4,67,5,1
36,118,8,72,5,2
12,149,12.6,74,5,3
18,313,11.5,62,5,4
NA,NA,14.3,56,5,5
```

Use

```
ozone <- read.table("ozone.csv", header=TRUE, sep=",")
```

or

```
ozone <- read.csv("ozone.csv")
```

Syntax notes

- Functions can have optional arguments (`sep` wasn't used the first time). Use `help(read.table)` for a complete description of the function and all the arguments.

- There's more than one way to do it.

- `NA` is the code for missing data. Think of it as "Don't Know". R handles it sensibly in computations: eg 1+NA, NA & FALSE, NA & TRUE. You cannot test `temp==NA` (Is temperature equal to some number I don't know?), so there is a function `is.na()`.

Reading text data

Sometime the variable names aren't included

1	0.2	115	90	1	3	68	42	yes
2	0.7	193	90	3	1	61	48	yes
3	0.2	58	90	1	3	63	40	yes
4	0.2	5	80	2	3	65	75	yes
5	0.2	8.5	90	1	2	64	30	yes

and you have to supply them

```
psa <- read.table("psa.txt", col.names=c("ptid","nadirpsa",
       "pretxpsa", "ps","bss","grade","age",
       "obstime","inrem"))
```

or

```
psa <- read.table("psa.txt")
names(psa) <- c("ptid","nadirpsa","pretxpsa", "ps",
        "bss","grade","age","obstime","inrem"))
```

Syntax notes

- Assigning a single vector (or anything else) to a variable uses the same syntax as assigning a whole data frame.

- `c()` is a function that makes a single vector from its arguments.

- `names` is a function that accesses the variable names of a data frame

- Some functions (such as `names`) can be used on the LHS of an assignment.

Fixed-format data

Two functions `read.fwf` and `read.fortran` read fixed-format data.

```
i1.3<-read.fortran("sipp87x.dat",c("f1.0","f9.0",
                    "f2.0","f3.0", "4f1.0", "15f1.0",
                    "2f12.7", "f1.0","f2.0" "2f1.0",
                    "f2.0", "15f1.0", "15f2.0",
                    "15f1.0","4f2.0", "4f1.0","4f1.0",
                    "15f1.0","4f8.0","4f7.0","4f8.0",
                     "4f5.0","15f1.0"), col.names=i1.3names,
                    buffersize=200)
```

Here `i1.3names` is a vector of names we created earlier. `buffersize` says how many lines to read in one gulp — small values can reduce memory use

Other statistical packages

```
library(foreign)
stata <- read.dta("salary.dta")
spss <- read.spss("salary.sav", to.data.frame=TRUE)
sasxport <- read.xport("salary.xpt")
epiinfo <- read.epiinfo("salary.rec")
```

Notes:

- Many functions in R live in optional packages. The `library()` function lists packages, shows help, or loads packages from the package library.

- The foreign package is in the standard distribution. It handles import and export of data. Hundreds of extra packages are available at `http://cran.us.r-project.org`.

The web

Files for `read.table` can live on the web

```
fl2000<-read.table("http://faculty.washington.edu/tlumley/
data/FLvote.dat", header=TRUE)
```

It's also possible to read from more complex web databases (such as the genome databases)

Operating on data

As R can have more than one data frame available you need to specify where to find a variable. The syntax `antibiotics$duration` means the variable `duration` in the data frame `antibiotics`.

```
## This is a comment
## Convert temperature to real degrees
antibiotics$tempC <- (antibiotics$temp-32)*5/9
## display mean, quartiles of all variables
summary(antibiotics)
```

Subsets

Everything in R is a vector (but some have only one element).
Use [] to extract subsets

```
## First element
antibiotics$temp[1]
## All but first element
antibiotics$temp[-1]
## Elements 5 through 10
antibiotics$temp[5:10]
## Elements 5 and 7
antibiotics$temp[c(5,7)]
## People who received antibiotics (note ==)
antibiotics$temp[ antibiotics$antib==1 ]
## or
with(antibiotics, temp[antib==1])
```

Notes

- Positive indices select elements, negative indices drop elements

- `5:10` is the sequence from 5 to 10

- You need `==` to test equality, not just `=`

- `with()` temporarily sets up a data frame as the default place to look up variables. You can do this longer-term with `attach()`, but I don't know any long-term R users who do this. It isn't as useful as it initial seems.

More subsets

For data frames you need two indices

```
## First row
antibiotics[1,]
## Second column
antibiotics[,2]
## Some rows and columns
antibiotics[3:7, 2:4]
## Columns by name
antibiotics[, c("id","temp","wbc")]
## People who received antibiotics
antibiotics[antibiotics$antib==1, ]
## Put this subset into a new data frame
yes <- antibiotics[antibiotics$antib==1,]
```

Computations

```
mean(antibiotics$temp)
median(antibiotics$temp)
var(antibiotics$temp)
sd(antibiotics$temp)
mean(yes$temp)
mean(antibiotics$temp[antibiotics$antib==1]
with(antibiotics, mean(temp[sex==2]))
toohot <- with(antibiotics, temp>99)
mean(toohot)
```

Factors

Factors represent categorical variables. You can't do mathematical operations on them (except for `==`)

```
> table(salary$rank,salary$field)


         Arts Other Prof
  Assist   668 2626   754
  Assoc   1229 4229  1071
  Full     942 6285  1984
> antibiotics$antib<-factor(antibiotics$antib,
                          labels=c("Yes","No"))
> antibiotics$agegp<-cut(antibiotics$antib, c(0,18,65,100))
> table(antibiotics$agegp)
  (0,18]  (18,65] (65,100]
       2       19        4
```

Strings

Functions for searching and modifying strings are based on regular expressions, which are like simple programs for representing strings

`"a"` matches the letter a

`"TATAA"` matches the string TATAA

`"ACGT.C"` matches ACGT followed by any single character, followed by C

`"a*b*"` matches zero or more as followed by zero or more bs

`"[A-Za-z]+"` matches one or more characters between A and Z or between a and z (in English, any letter).

`"[[:alpha:]]+"` matches one or more letters, defined according to the current locale

`"\\.csv$"` matches the string .csv at the end of a string

`"([[:alpha:]]+)[[:space:][:punct:]]+\\1[[:space:][:punct:]]+"` matches any repeated word.

Functions for regular expressions

- `grep(regexp, vector)` finds all the strings in the vector that contain a substring matching the regular expression
- `sub(regexp, replacement, vector)` replaces the first substring matching the regular expression with the replacement (for each element of the vector). `gsub` does the same thing but can make more than one replacement per string
- `regexpr(regexp, vector)` returns the position of the first match within each string, `gregexpr` is the same except that it returns all matches.
- `strsplit()` splits a string at each match to a regular expression
- `glob2rx()` converts filename wildcard specifications to regular expressions

There is also support for Perl-style regular expressions with similar functionality but different syntax

Help

- `help(fn)` for help on fn

- `help.search("topic")` for help pages related to "topic

- `apropos("tab")` for functions whose names contain "tab"

- Search function on the `http://www.r-project.org` web site.

Graphics

R (and S-PLUS) can produce graphics in many formats, including:

- on screen
- PDF files for LATEX or emailing to people
- PNG or JPEG bitmap formats for web pages (or on non-Windows platforms to produce graphics for MS Office). PNG is also useful for graphs of large data sets.
- On Windows, metafiles for Word, Powerpoint, and similar programs

Setup

Graphs should usually be designed on the screen and then may be replotted on eg a PDF file (for Word/Powerpoint you can just copy and paste)

For printed graphs, you will get better results if you design the graph at the size it will end up, eg:

```
## on Windows
windows(height=4,width=6)
## on Unix
x11(height=4,width=6)
```

Word or LaTeX can rescale the graph, but when the graph gets smaller, so do the axis labels...

Finishing

After you have the right commands to draw the graph you can produce it in another format: eg

```
## start a PDF file
pdf("picture.pdf",height=4,width=6)
## your drawing commands here
...
### close the PDF file
dev.off()
```

Drawing

Usually use plot() to create a graph and then lines(), points(), legend(), text(), and other commands to annotate it.

plot() is a generic function: it does appropriate things for different types of input

```
## scatterplot
plot(salary$year, salary$salary)
## boxplot
plot(salary$rank, salary$salary)
## stacked barplot
plot(salary$field, salary$rank)
```

and others for other types of input. This is done by magic (actually, by advanced technology).

Formula interface

The plot() command can be written

```
plot(salary~rank, data=salary)
```

introducing the formula system that is also used for regression models. The variables in the formula are automatically looked up in the data= argument.

Designing graphs

Two important aspects of designing a graph

- It should have something to say

- It should be legible

Having something to say is your problem; software can help with legibility.

Designing graphs

Important points

- Axes need labels (with units, large enough to read)

- Color can be very helpful (but not if the graph is going to be printed in black and white).

- Different line or point styles usually should be labelled.

- Points plotted on top of each other won't be seen

After these are satisfied, it can't hurt to have the graph look nice.

Options

Set up a data set: daily ozone concentrations in New York, summer 1973

```
data(airquality)
names(airquality)
airquality$date<-with(airquality, ISOdate(1973,Month,Day))
```

All these graphs were designed at 4in×6in and stored as PDF files

```
plot(Ozone~date, data=airquality)
```
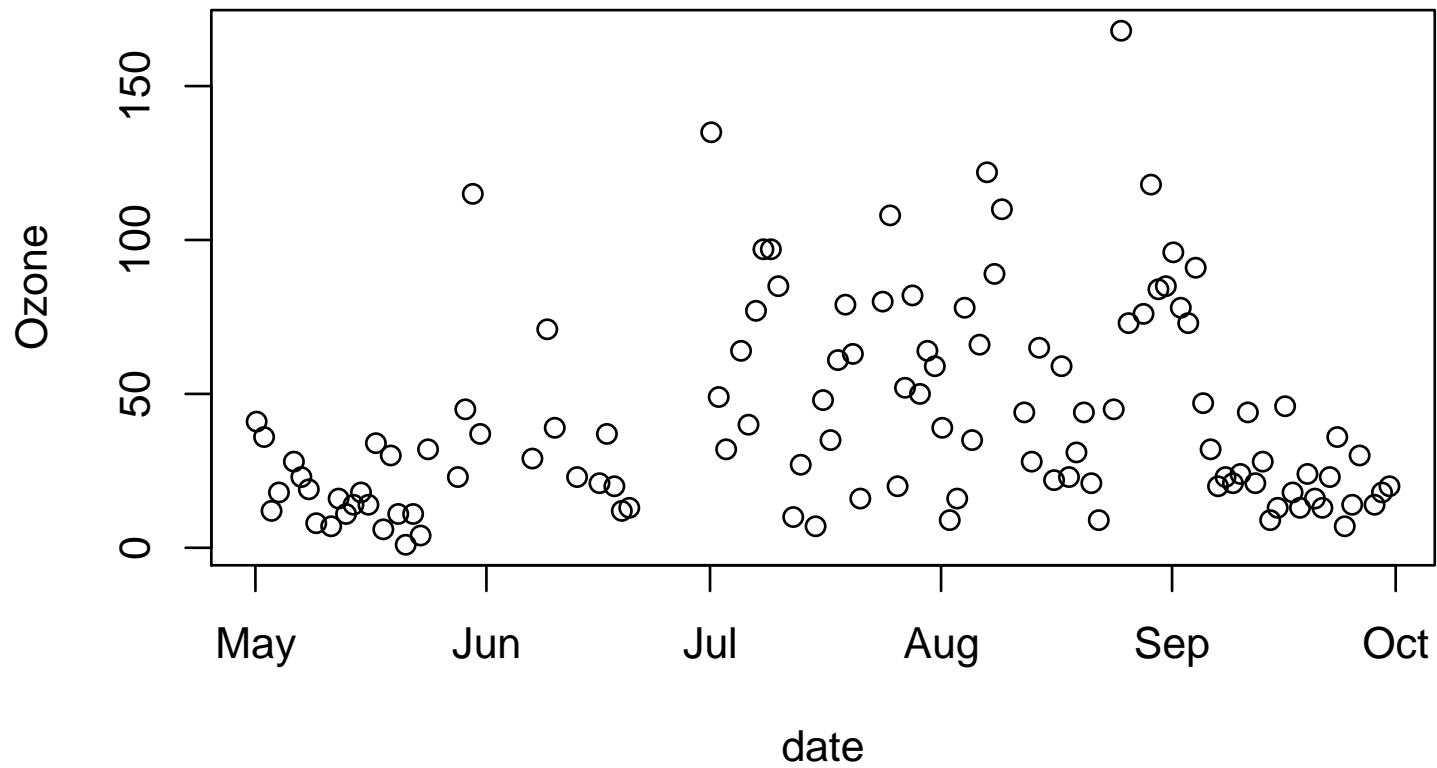
```
plot(Ozone~date, data=airquality,type="l")
```
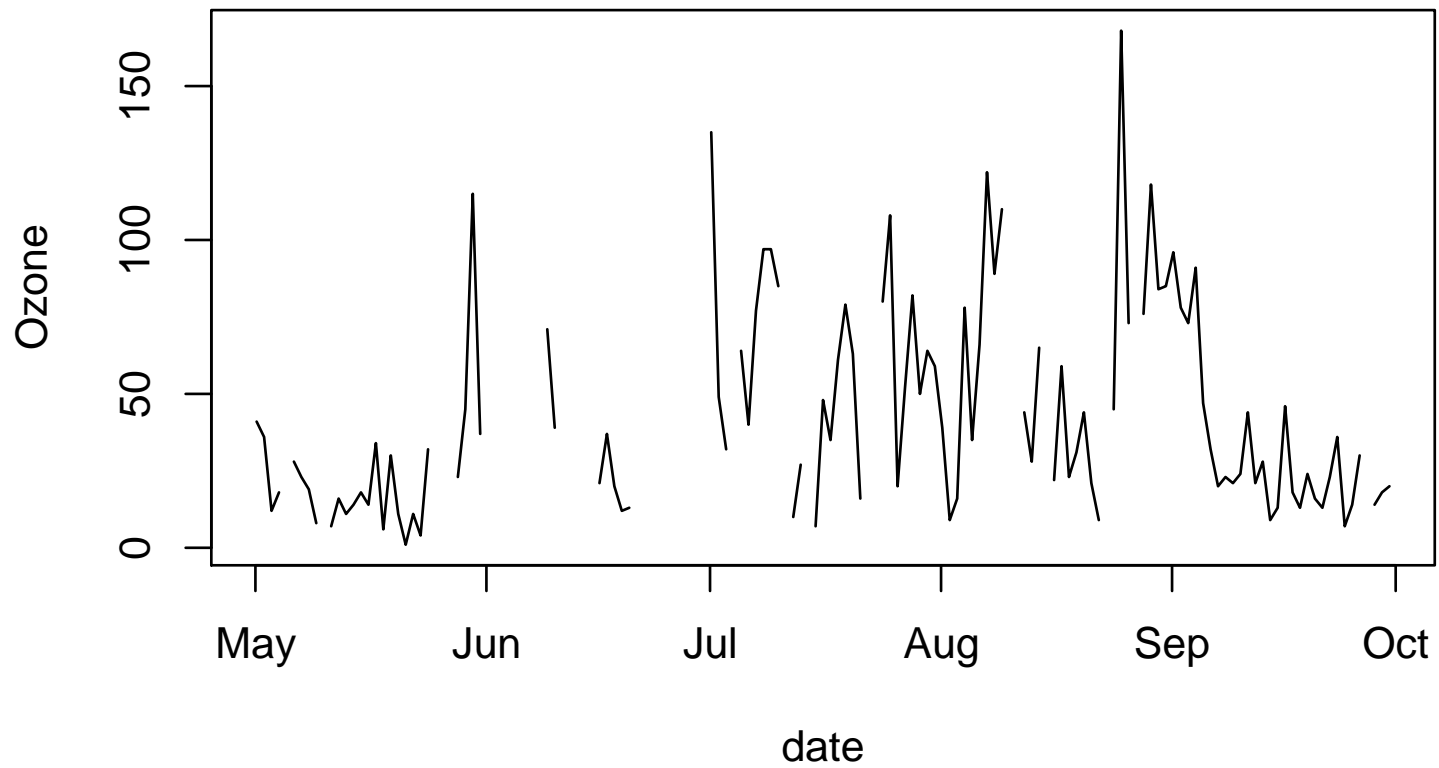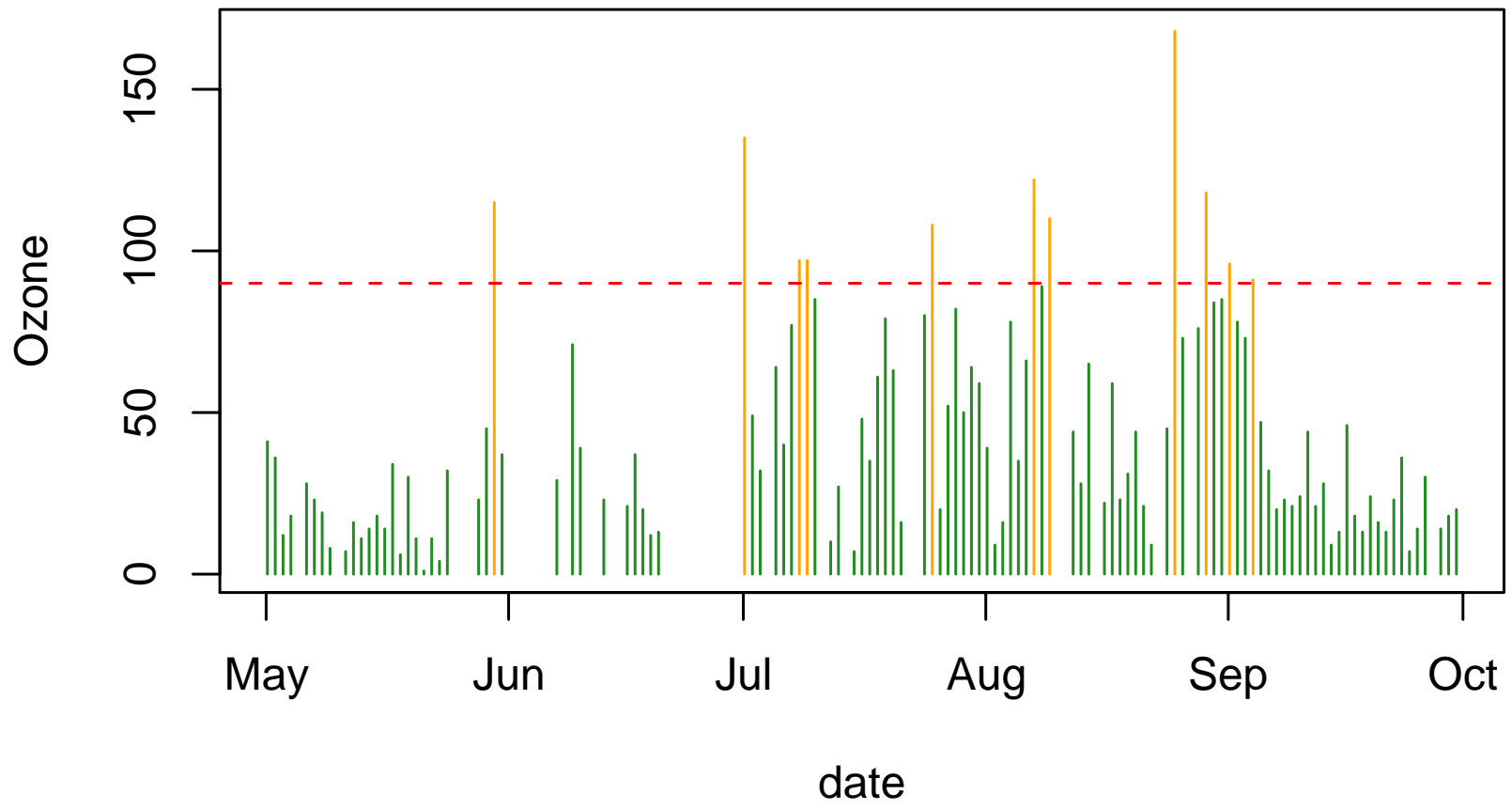
```
plot(Ozone~date, data=airquality,type="h")
```
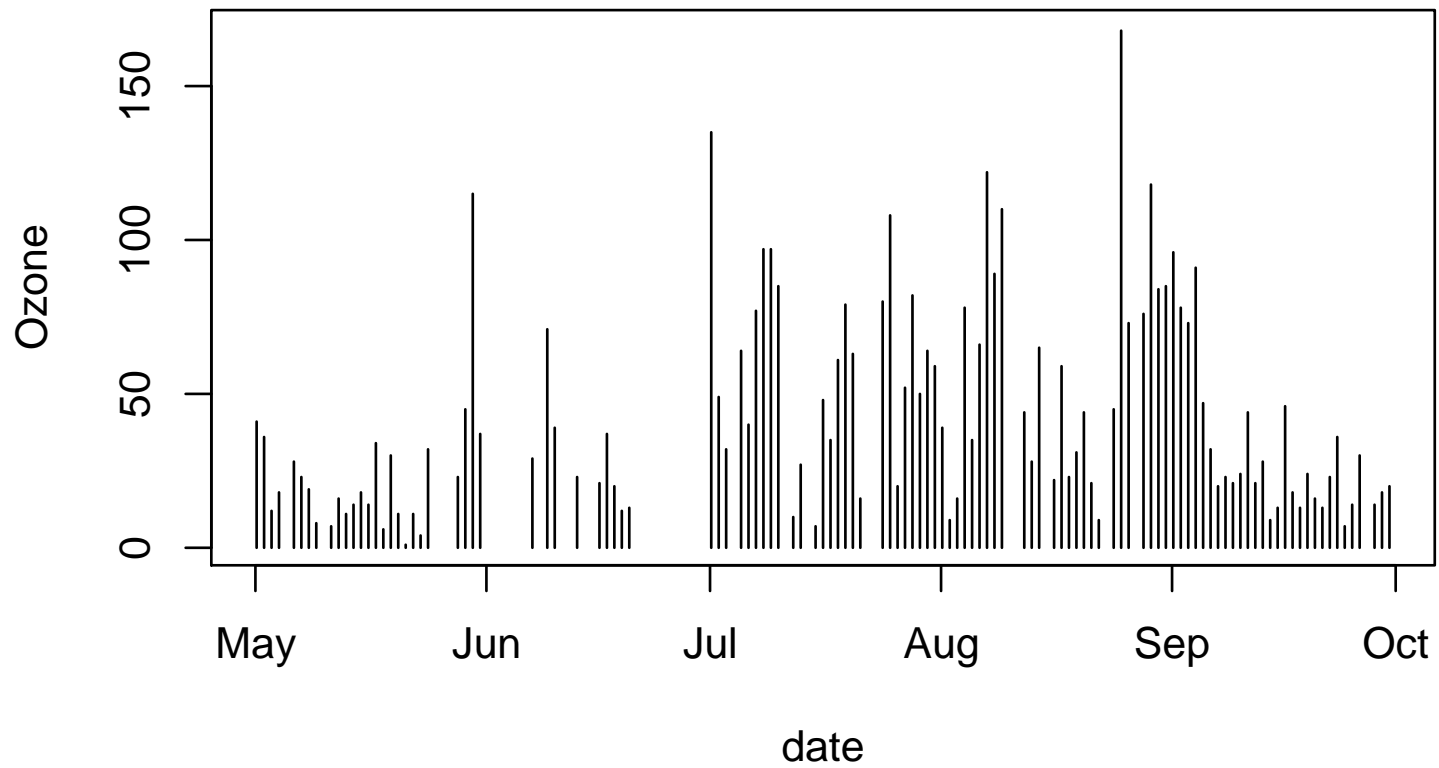
```
plot(Ozone~date, data=airquality,type="n")
```
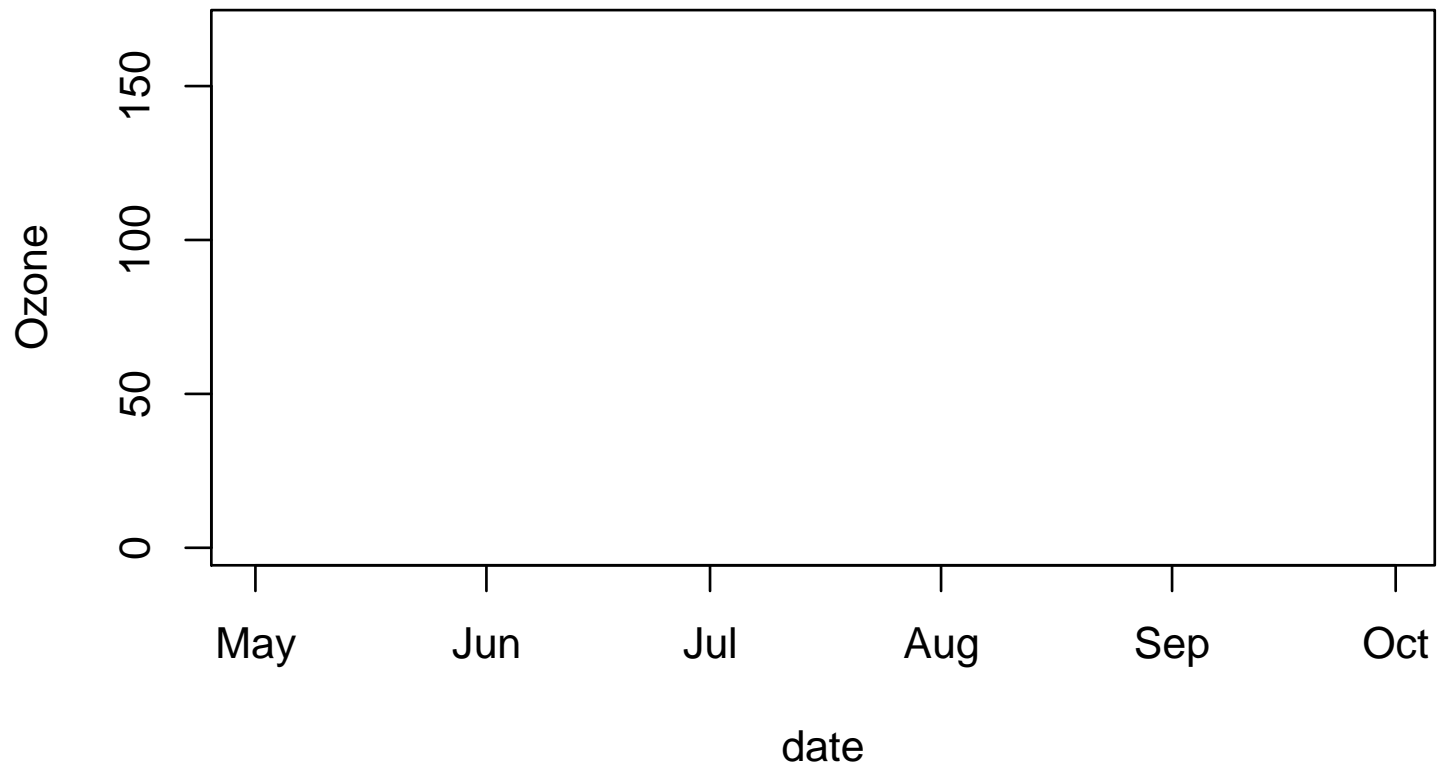
```
bad<-ifelse(airquality$Ozone>=90, "orange","forestgreen")
plot(Ozone~date, data=airquality,type="h",col=bad)
abline(h=90,lty=2,col="red")
```
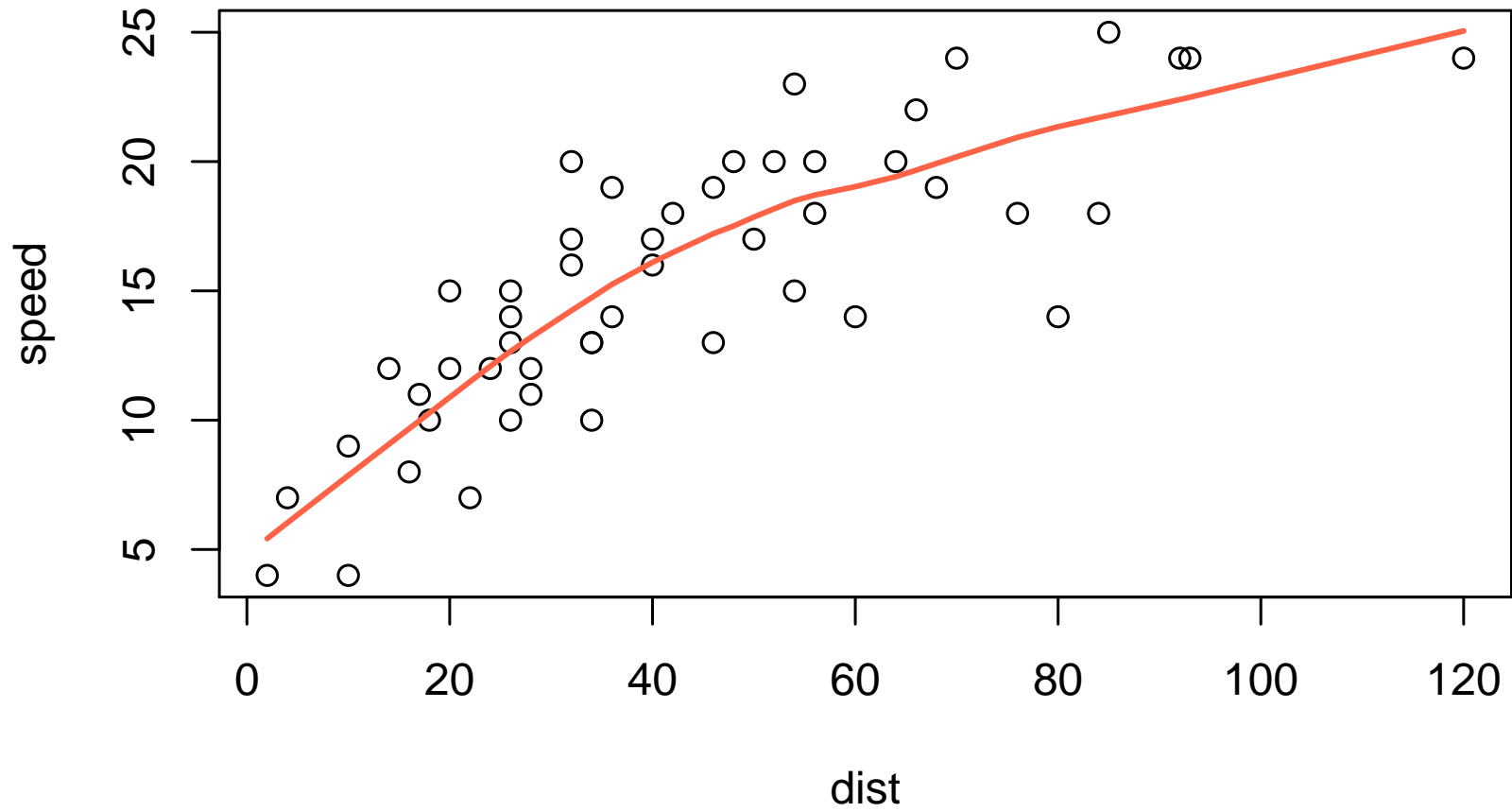
Notes

- `type=` controls how data are plotted. `type="n"` is not as useless as it looks: it can set up a plot for latter additions.
- Colors can be specified by name (the `colors()` function gives all the names), by red/green/blue values (#rrggbb with six base-sixteen digits) or by position in the standard palette of 8 colors. For `pdf()` and `quartz()`, partially transparent colors can be specified by $\#rrggbbaa$.
- `abline` draws a single straight line on a plot
- `ifelse()` selects between two vectors based on a logical variable.
- `lty` specifies the line type: 1 is solid, 2 is dashed, 3 is dotted, then it gets more complicated.

Adding to a plot

```
data(cars)
plot(speed~dist,data=cars)
with(cars, lines(lowess(dist,speed), col="tomato", lwd=2))
plot(speed~dist,data=cars, log="xy")
with(cars, lines(lowess(dist,speed), col="tomato", lwd=2))
with(cars, lines(supsmu(dist,speed), col="purple", lwd=2))
legend(2,25, legend=c("lowess","supersmoother"),bty="n", lwd=2,
   col=c("tomato","purple"))
```
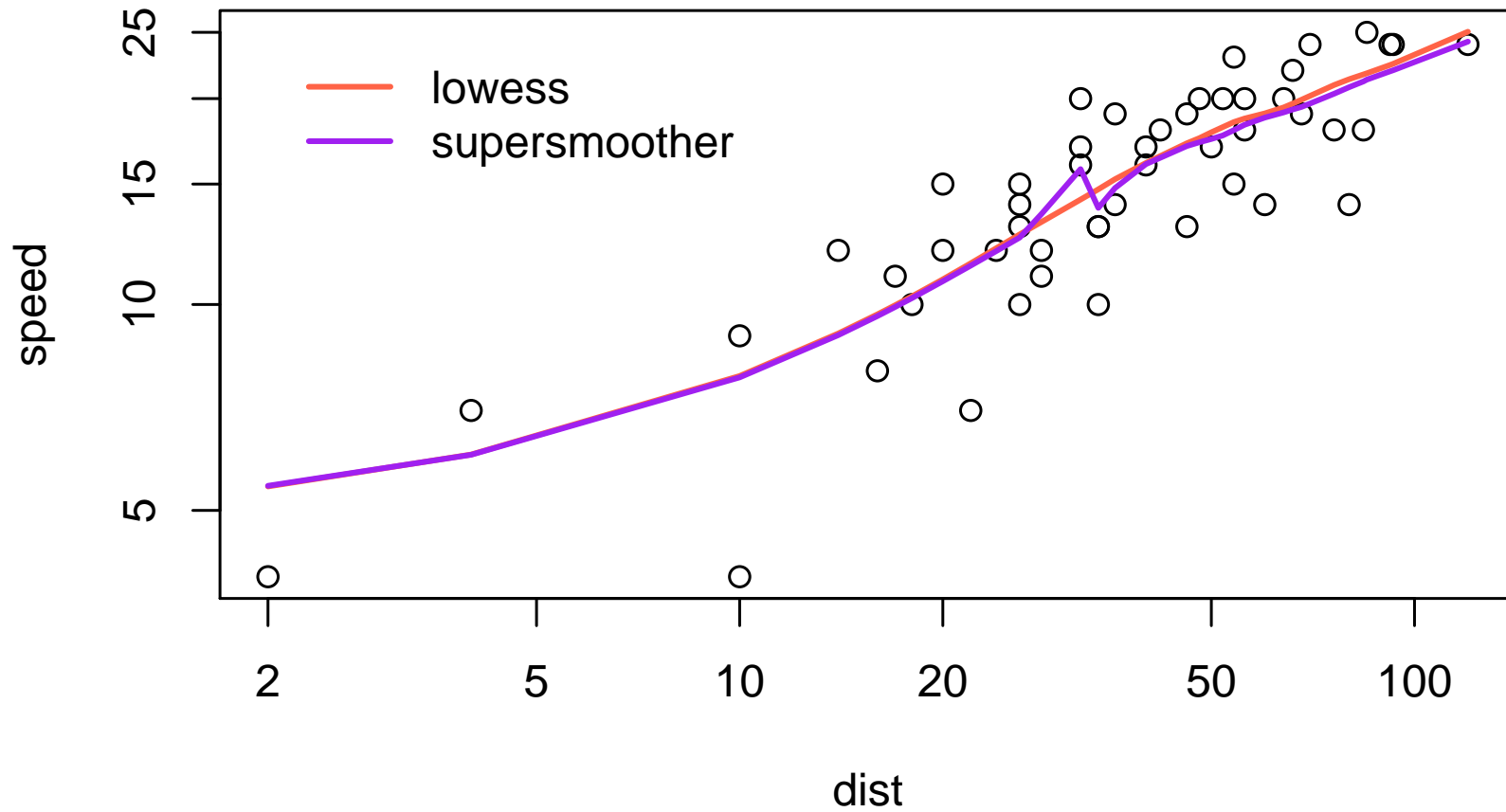
Adding to a plot

# Adding to a plot
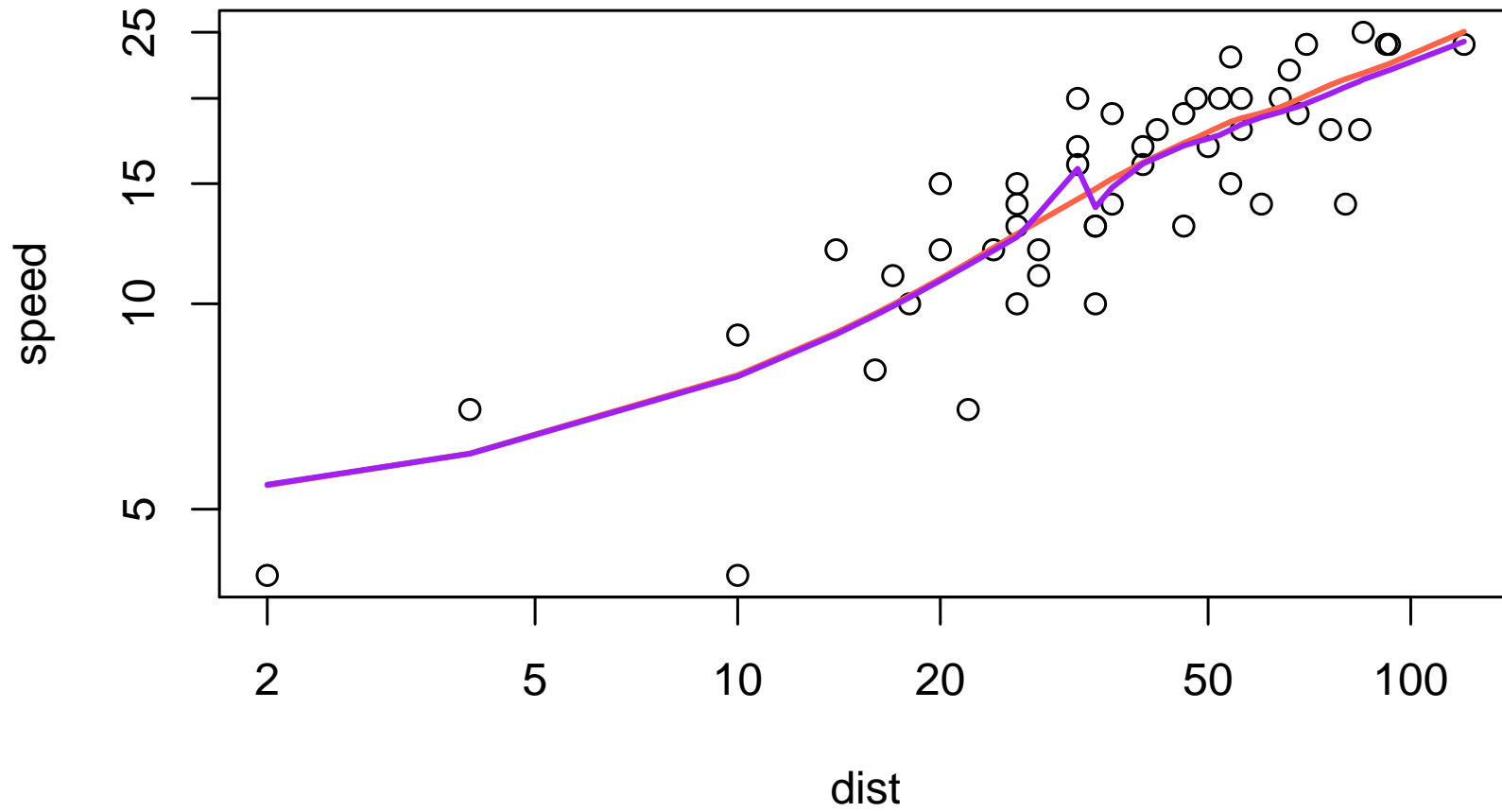
Adding to a plot

Notes

- `lines` adds lines to an existing plot (`points()` adds points).

- `lowess()` and `supsmu()` are scatterplot smoothers. They draw smooth curves that fit the relationship between y and x locally.

- `log="xy"` asks for both axes to be logarithm (`log="x"` would just be the x-axis)

- `legend()` adds a legend

Boxplots

```
data(api, package="survey")
boxplot(mobility~stype,data=apipop, horizontal=TRUE)
```

Boxplots
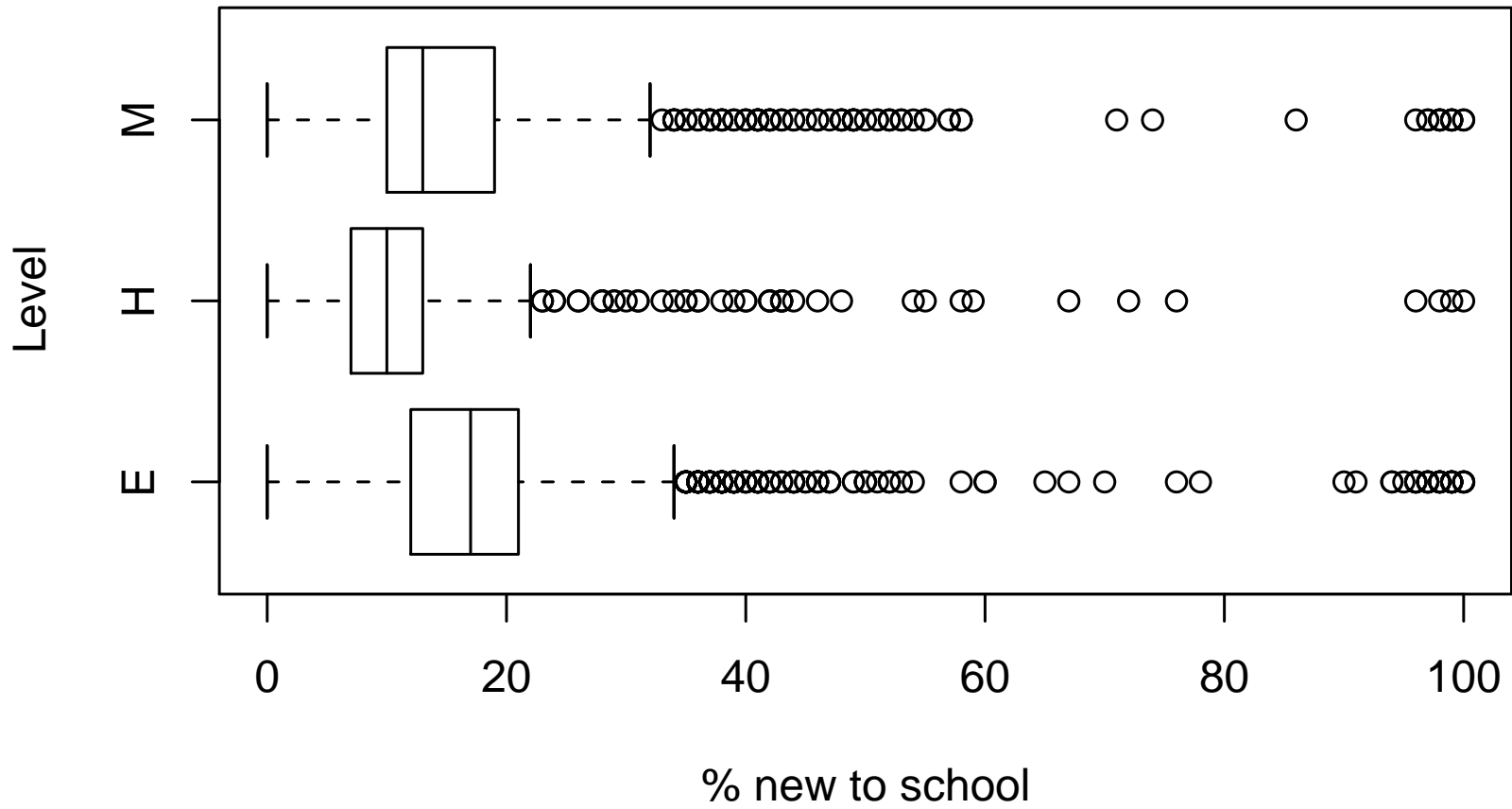
Notes

- `boxplot` computes and draws boxplots.

- `horizontal=TRUE` turns a boxplot sideways

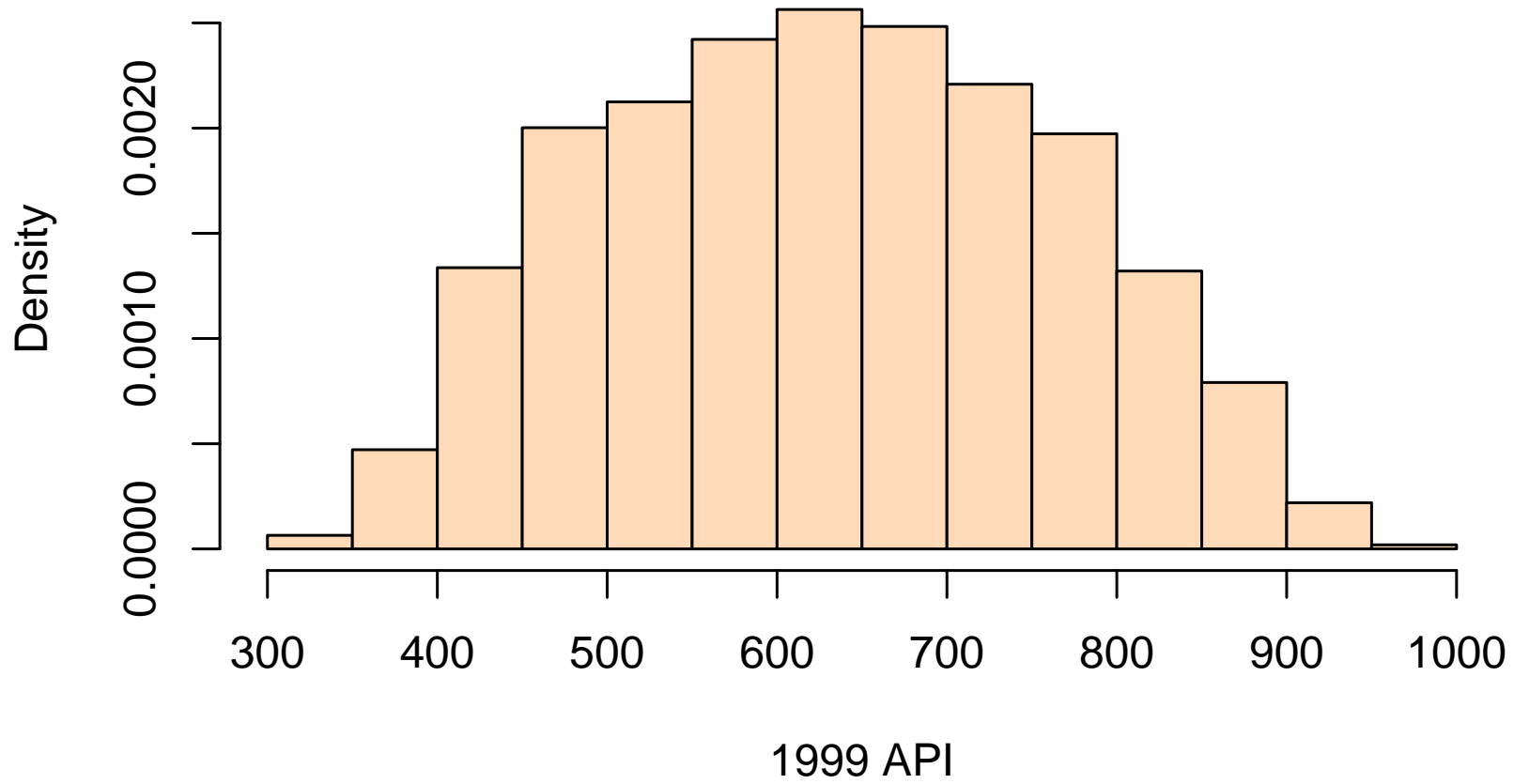- `xlab` and `ylab` are general options for x and y axis labels.

Barplots

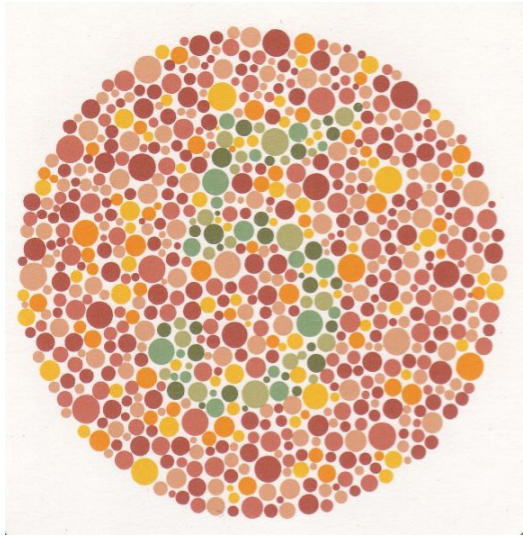Use `barplot` to draw barplots and `hist` to draw histograms:

```
barplot(VADeaths,beside=TRUE,legend=TRUE)
hist(apipop$api99,col="peachpuff",xlab="1999 API",
     main="",prob=TRUE)
```

- `main=` specifies a title for the top of the plot

- `prob=TRUE` asks for a real histogram with probability density rather than counts.
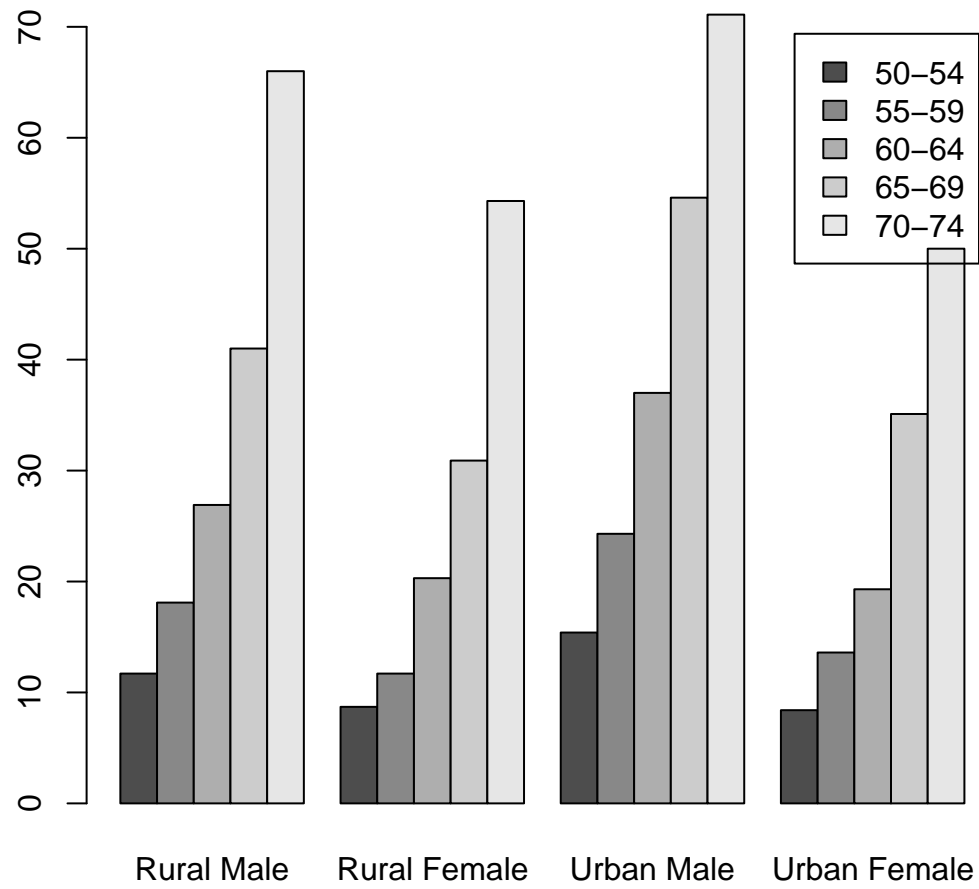
Scatterplot matrix and color-blindness

Shinobu Ishihara was a Japanese opthalmologist. While working in the Military Medical School he was asked to devise a testing system for color vision abnormalities. The original plates were hand-painted and had hiragana symbols as the patterns, a later 'international' edition used arabic numerals.

We will examine the colors on one of these plates.

Getting the data

The first step is to get colors into R. I converted the JPEG file to a PNG file, which can then be converted to a plain text format.

```
## read file in SNG format, converted via PNG from JPEG
disappearing<-readLines("ishihara-disappearing.sng")
start<-grep("IMAGE",disappearing)+2
end<-length(disappearing)+1
disappearing<-unlist(strsplit(disappearing[start:end],
                              "[[:space:]]"))

## remove very rare colors, which are just edge effects
disappearing<-names(table(disappearing))[table(disappearing)>6]

## change rrggbb to #RRGGBB
disappearing<-paste("#",toupper(disappearing),sep="")
```
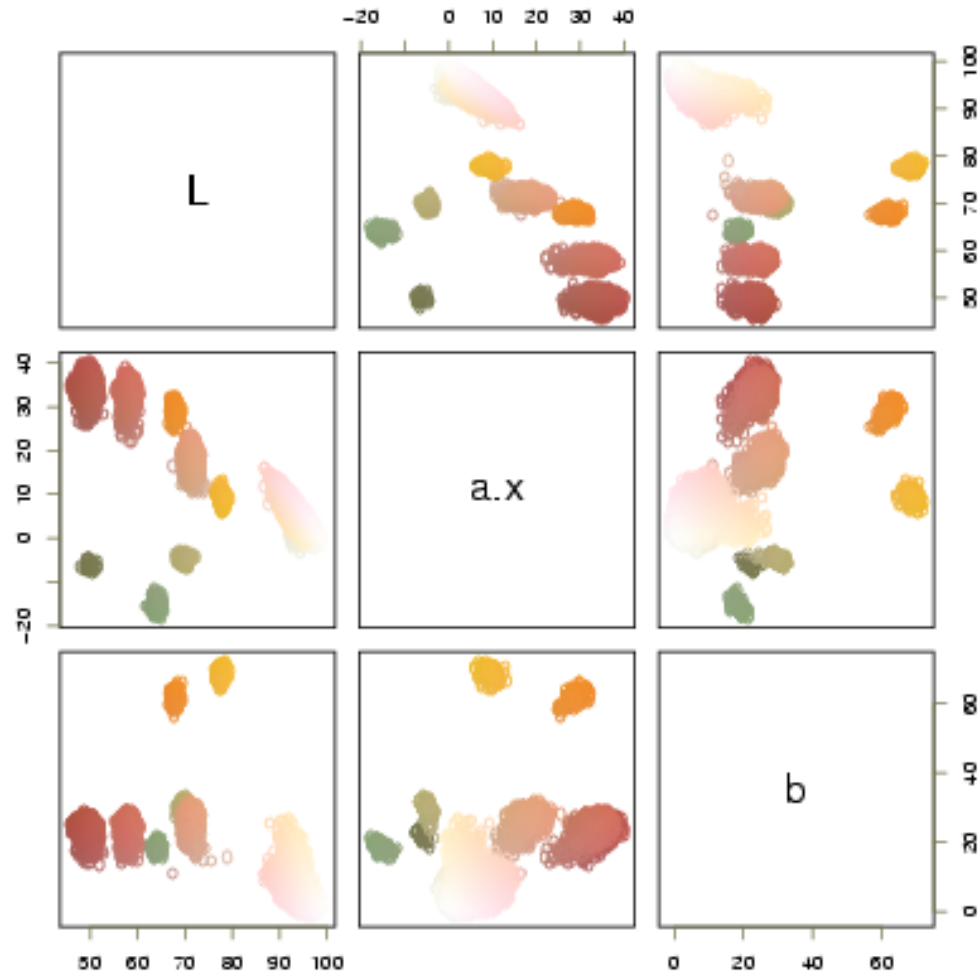
Color spaces

The `convertColor` function converts between coordinate systems for color (R 2.1.0-forthcoming). Here we

```
Lab.dis<-convertColor(matrix(disappearing), from=hexcolor,
                          to="Lab",scale.in=NULL)
```

The CIE Lab coordinate system is approximately perceptually uniform, with orthogonal light:dark, red:green, and blue:yellow axes.

Lab coordinates

Blue-green points are separated from the dark red points on the b axis but not on a or L.

Since the b axis is supposed to be aligned with exactly the color spectrum that is indistinguishable in red-green colorblindnesss, this explains why the tests work.

Remapping

The dichromat package in R implements a mapping that shows how colours appear to color-blind viewers.

We can look at the Lab space after the transformation for protanopia
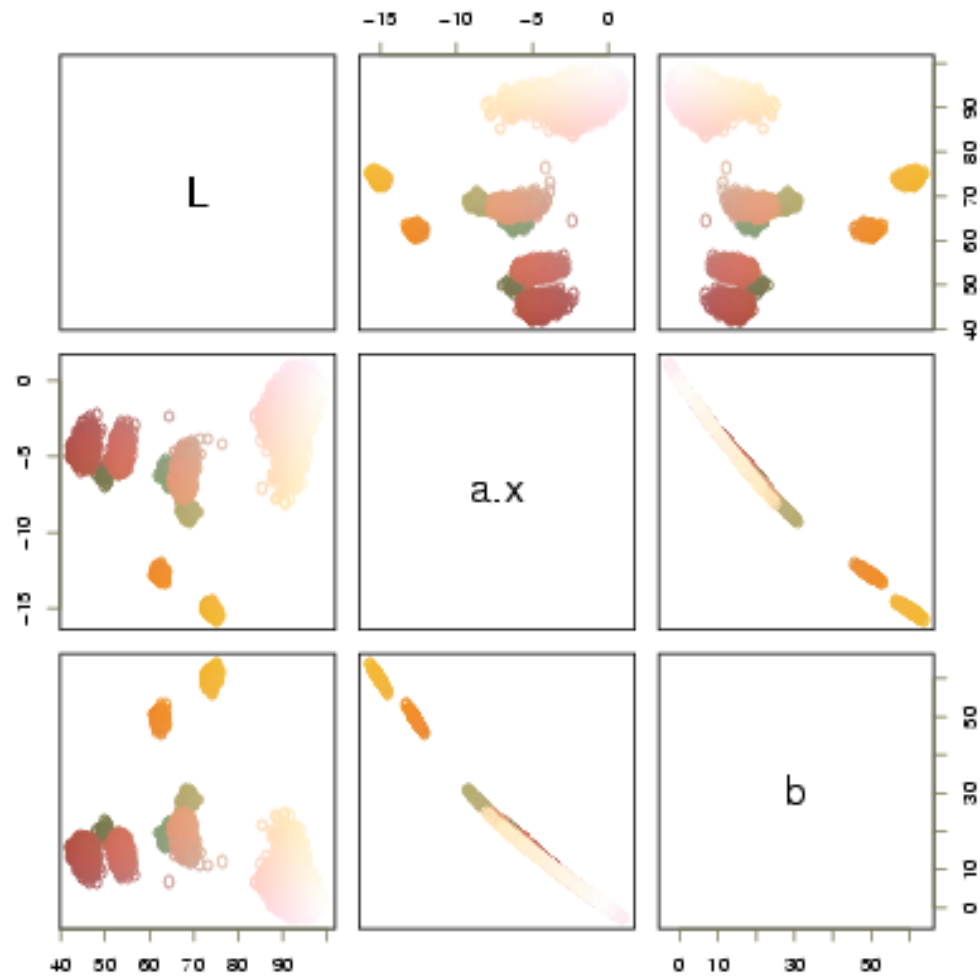
Protanopia

Image plots

`image()` plots colors to indicate values on a two-dimensional grid. A red-green spectrum is often used to code the values (especially in genomics) but this is a really bad idea.

Use the dichromat package to compare red:green and blue:yellow spectra.

```
par(mfrow=c(2,3))
 x<-matrix(rnorm(10*10),10)
 redgreen<-colorRampPalette(c("red","green3"))
 image(1:10,1:10,x, col=bluescale(10),
 main="blue-yellow scale")
 image(1:10,1:10,x, col=dichromat(bluescale(10)), main="deutan")
 image(1:10,1:10,x,col=dichromat(bluescale(10),"protan"), main="protan")

 image(1:10,1:10,x, col=redgreen(10),
 main="red-green scale")
 image(1:10,1:10,x, col=dichromat(redgreen(10)), main="deutan")
 image(1:10,1:10,x, col=dichromat(redgreen(10),"protan"), main="protan")
```

Image plots
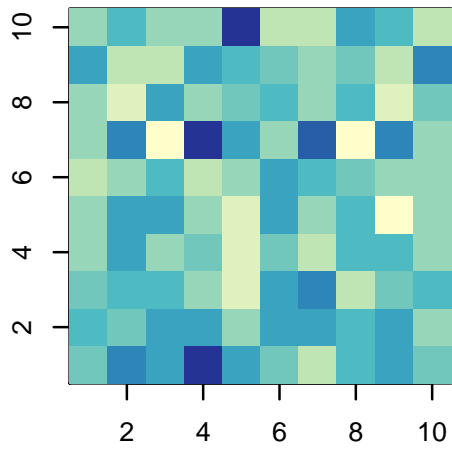
blue–yellow scale   deutan   protan
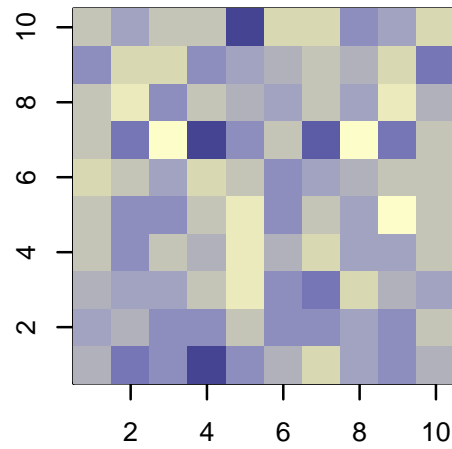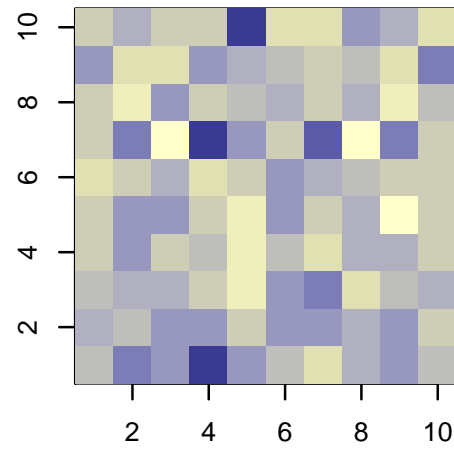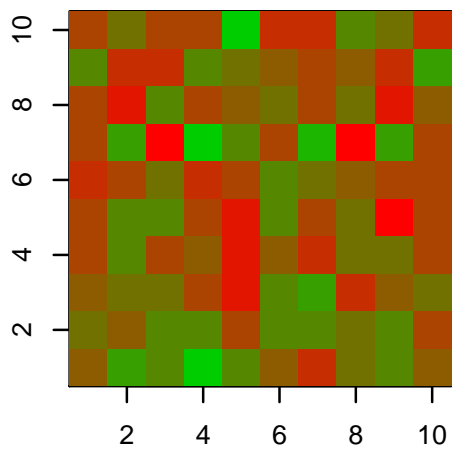
red–green scale   deutan   protan
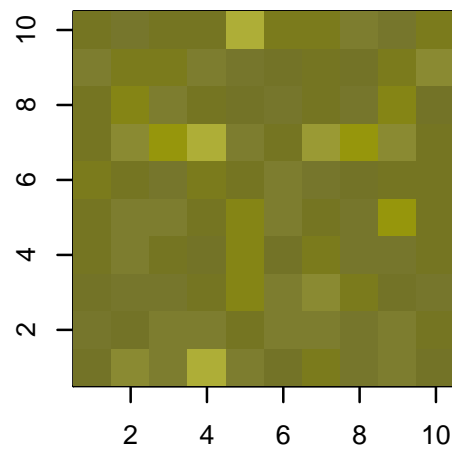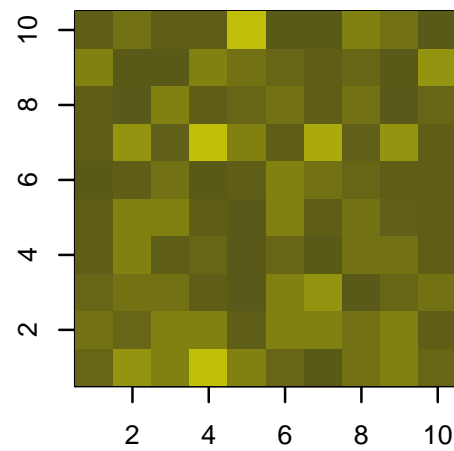
Large data sets

Scatterplots quickly get crowded. For example, the California Academic Performance Index is reported on 6194 schools

```
> plot(api00~api99,data=apipop)
> colors<-c("tomato","forestgreen","purple")[apipop$stype]
> plot(api00~api99,data=apipop,col=colors)
```

Large data sets
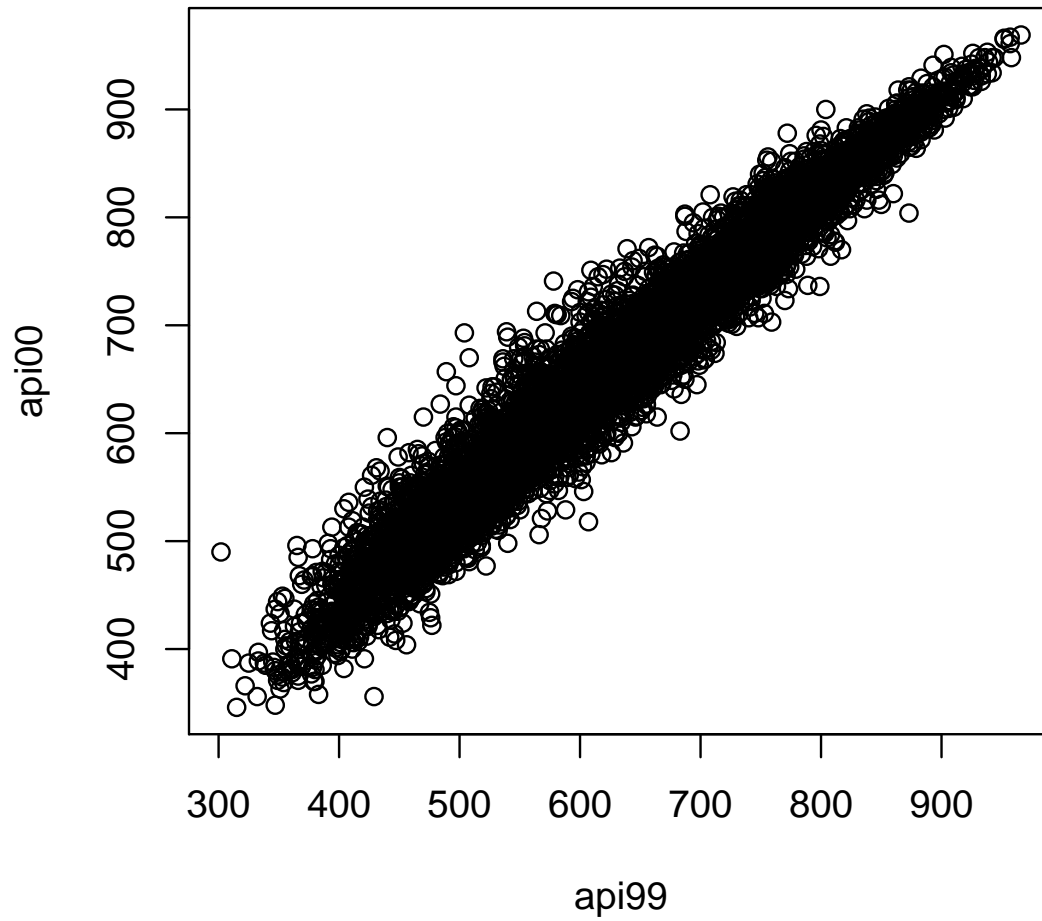
Large data sets
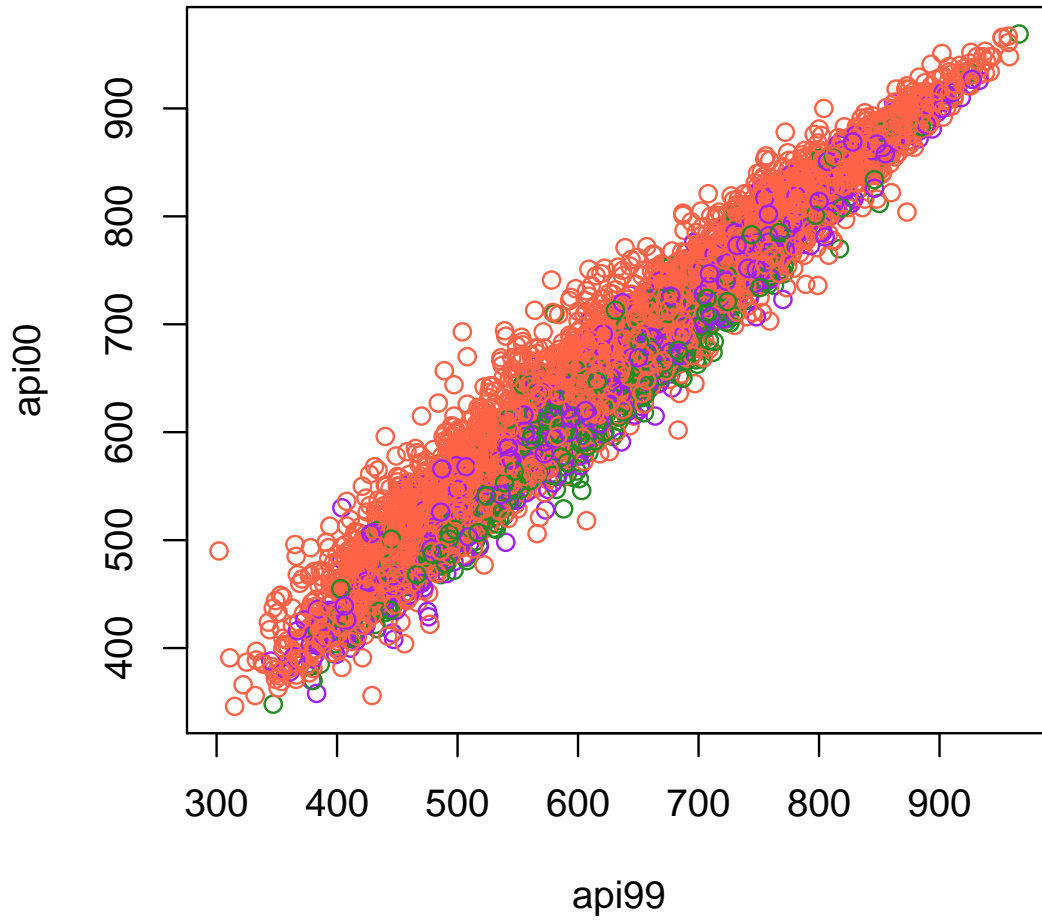
Density plots

For a single large scatterplot some form of density estimation is useful

```
library(MASS)
with(apipop, image(kde2d(api99,api00),
       col=grey(seq(1,0.2,length=20))))
library(hexbin)
with(apipop, plot(hexbin(api99,api00), style="centroids"))
```

- `kde2d` in the MASS package is a 2-dimensional kernel density estimate. It returns the density of points everywhere on a rectangular grid. Variants are `contour`, which draws contours and `filled.contour`, which does coloring and contours.

- `image` draws images from a rectangular grid of data

Density plots

- `hexbin` is in the hexbin package from the Bioconductor project. It computes the number of points in each hexagonal bin.

- The `style=centroids` plot puts a filled hexagon with size depending on the number of points at the centroid of the points in the bin.

Density plots
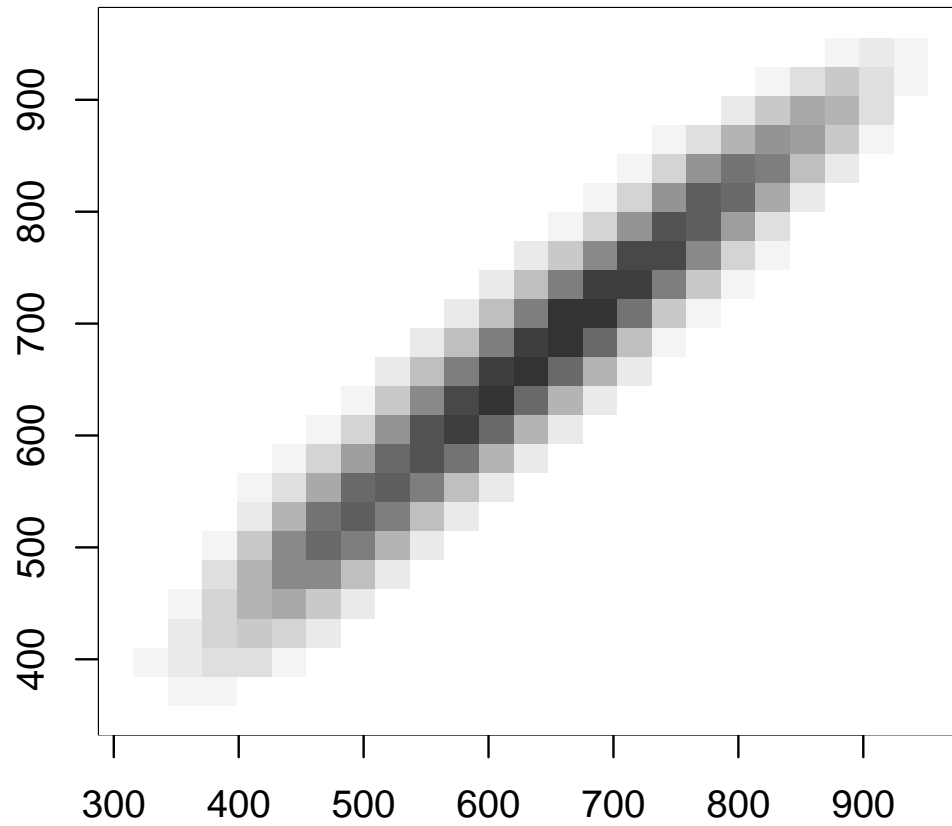
Density plots
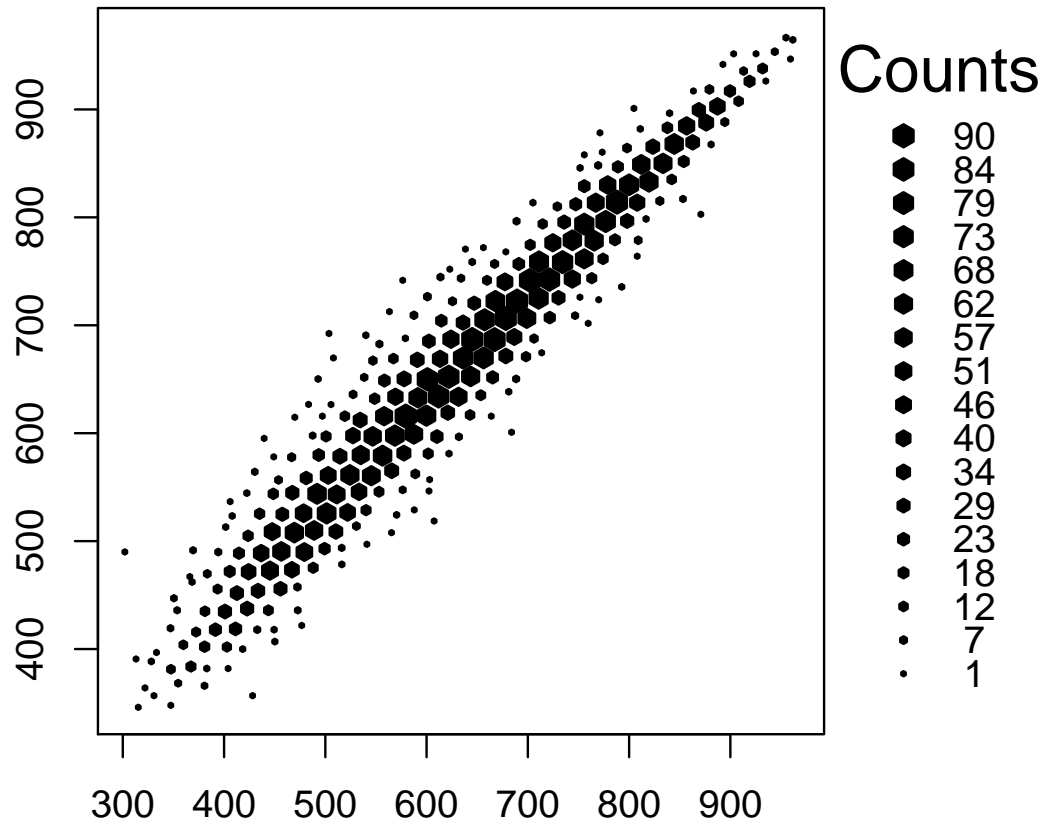
Smoothers

For showing multiple groups a scatterplot smoother or perhaps boxplots would be better.

```
> boxplot(api00~cut(api99,(3:10)*100), data=apipop)
> par(las=1)
> par(mar=c(5.1,10.1,2.1,2.1))
> boxplot(api00~interaction(stype,
                               cut(api99,(3:10)*100)),
          data=apipop, horizontal=TRUE,col=1:3)
 plot(api00~api99,data=apipop,type="n")
 with(subset(apipop, stype=="E"),
      lines(lowess(api99, api00), col="tomato"))
 with(subset(apipop, stype=="H"),
      lines(lowess(api99, api00), col="forestgreen"))
 with(subset(apipop, stype=="M"),
      lines(lowess(api99, api00), col="purple"))
```

Smoothers
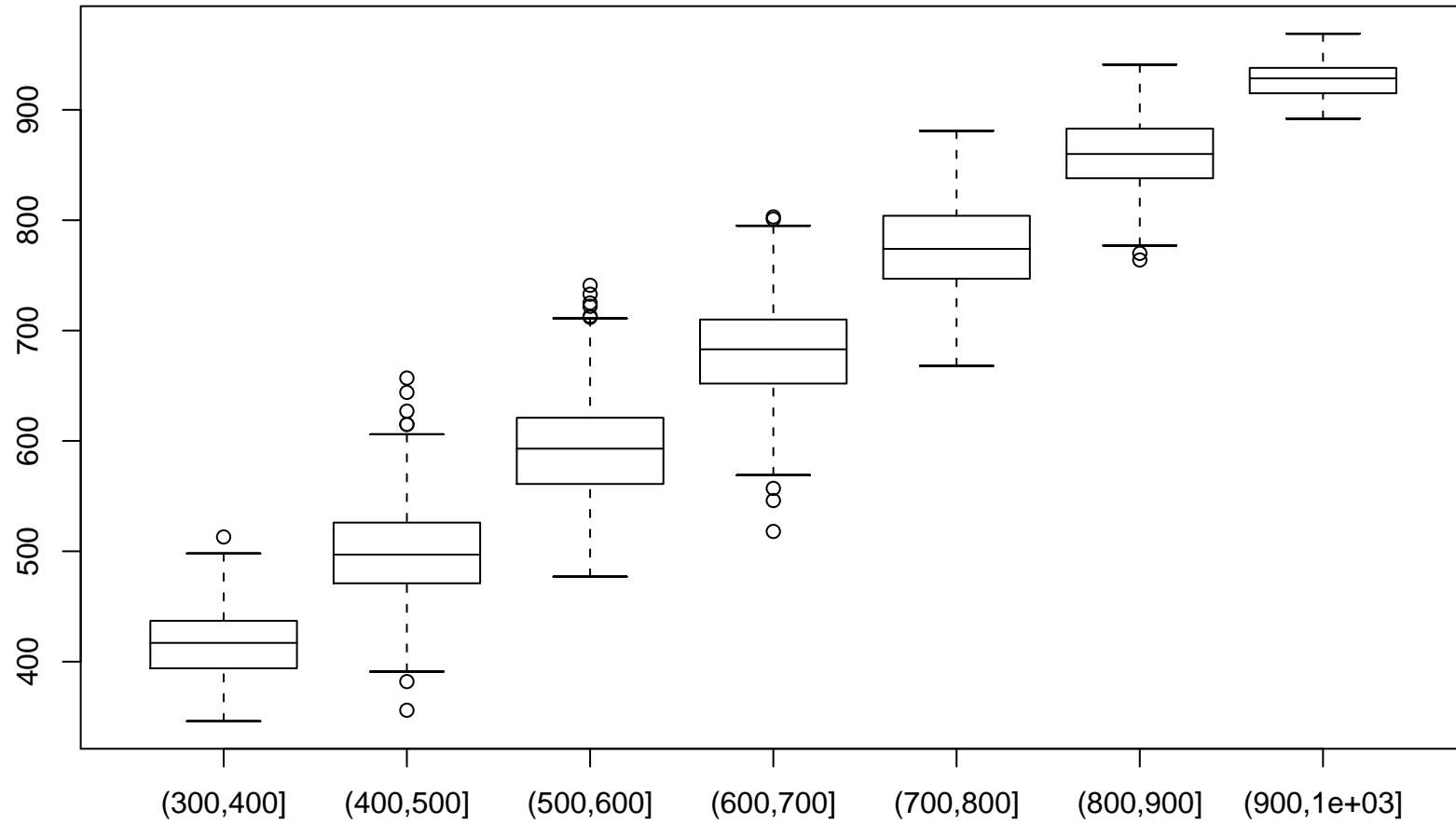
Smoothers
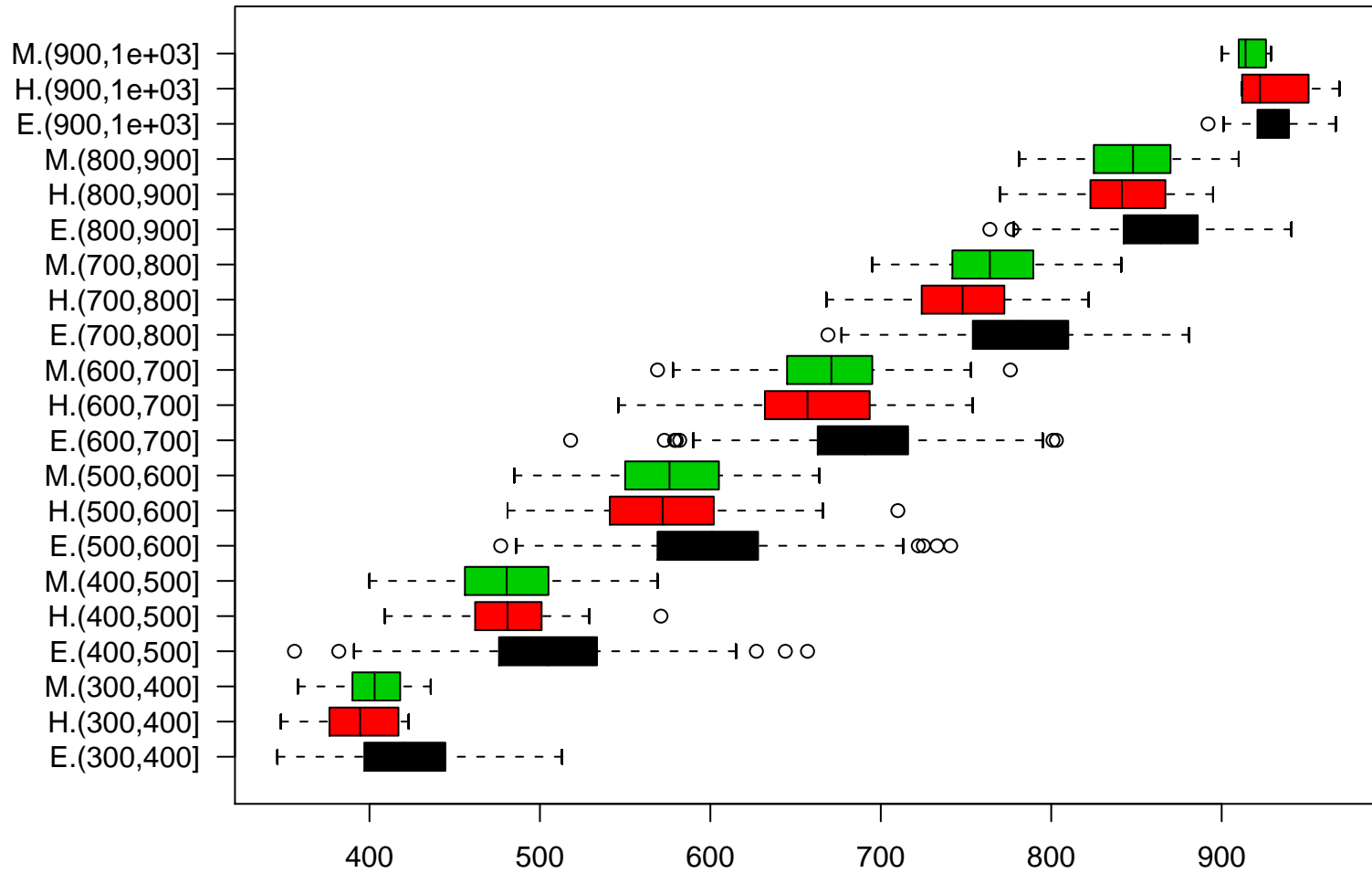
Smoothers
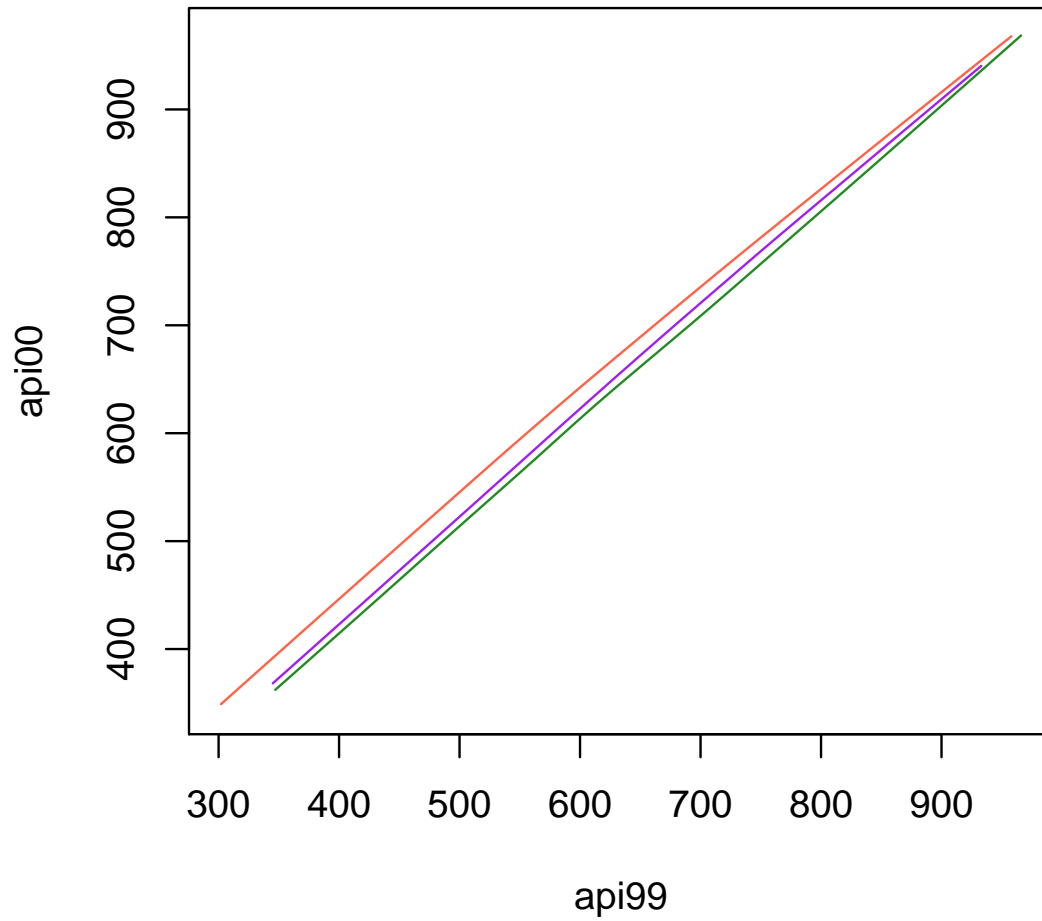
Notes

- `cut` turns a variable into a factor by cutting it at the specified points.

- Note the use of `type="n"`

- `par(mar=)` sets the margins around the plot. We need a large left margin for the labels.

- `subset` takes a subset of a data frame.

Example: Confidence intervals

A popular plot in the lab sciences looks like:

Death Rates in Virginia
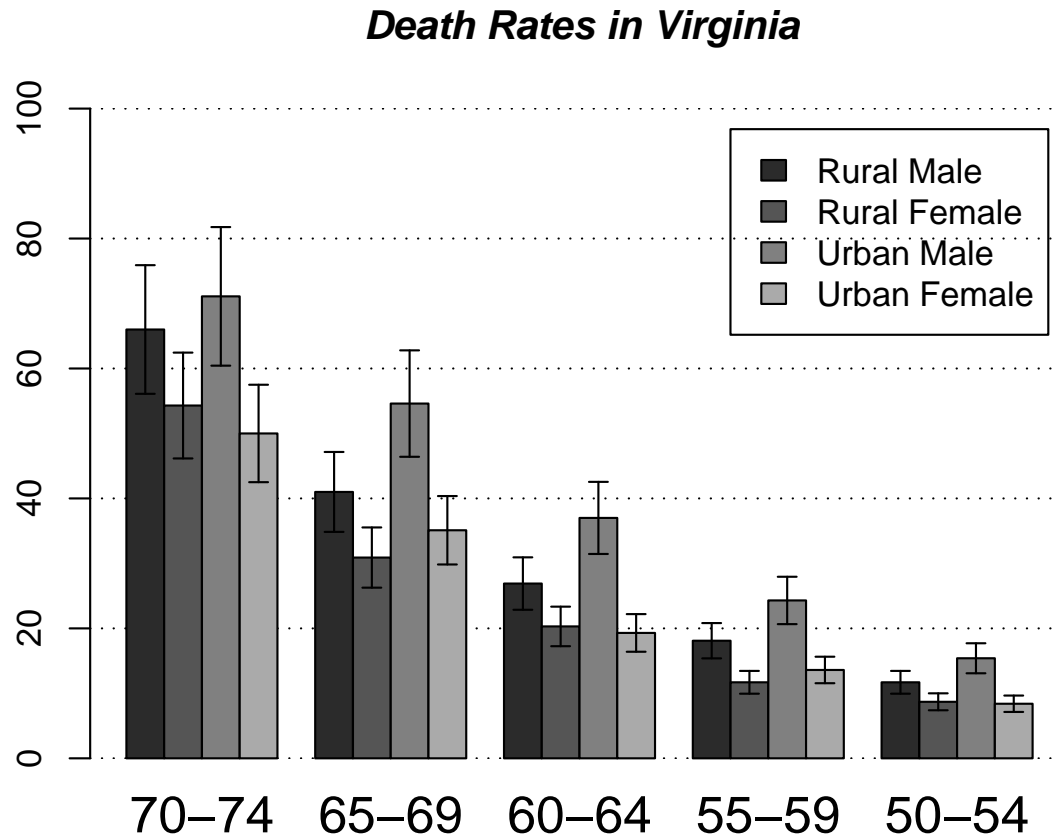
and can be created with `gplots::barplot2`

Example: Confidence intervals

Line plots are ordinarily preferable to bar plots,
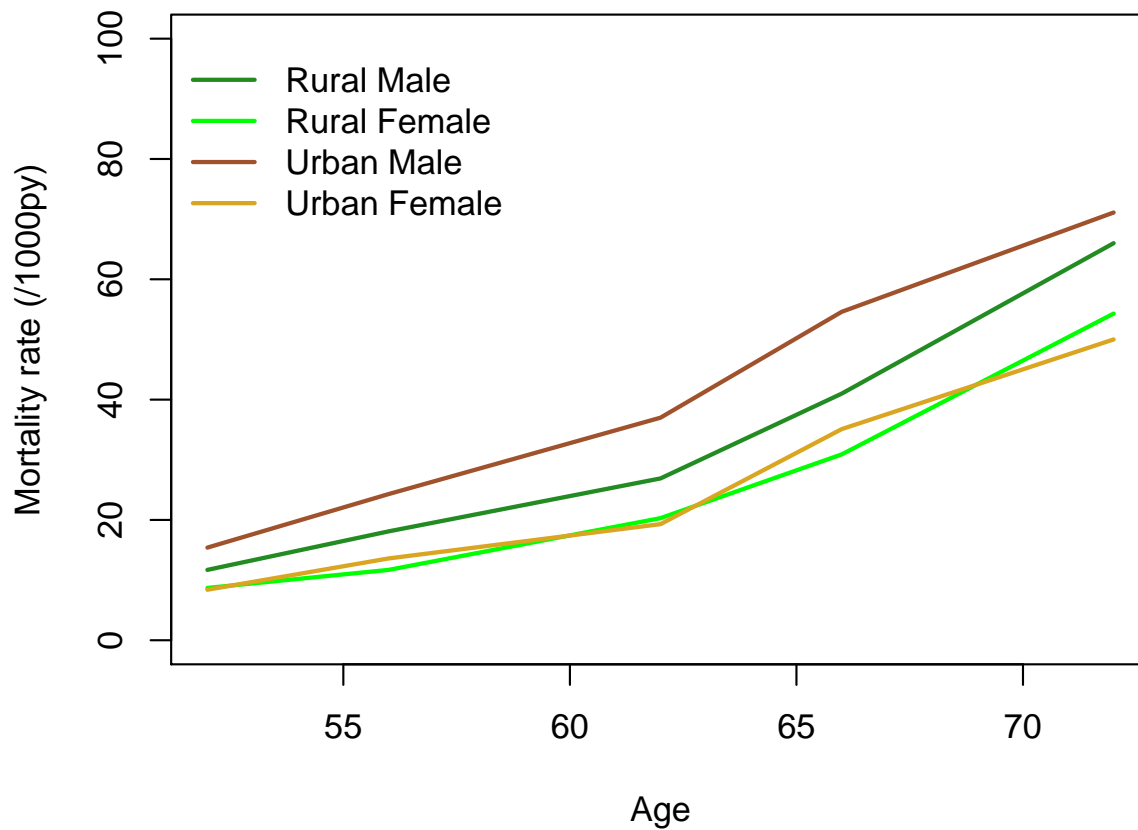
Example: Confidence intervals

but the confidence intervals get in the way
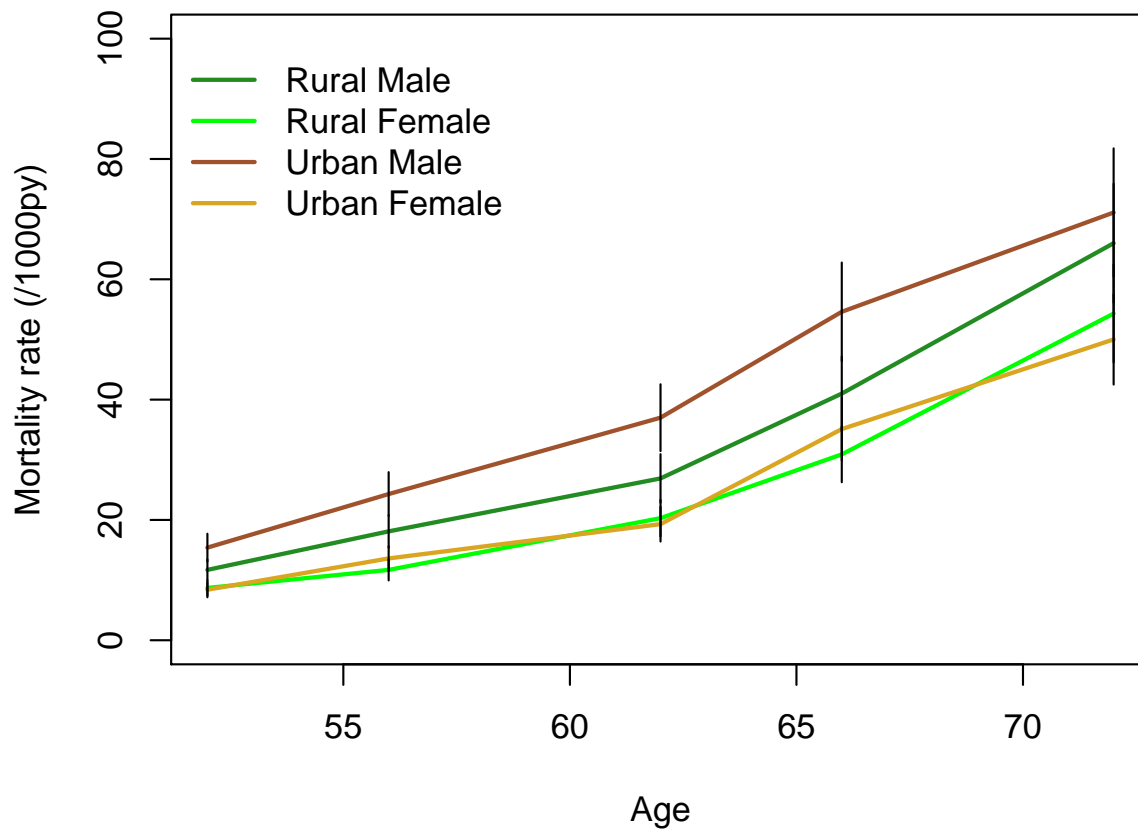
Example: Confidence intervals

Offsetting the intervals slightly and coloring them helps a bit
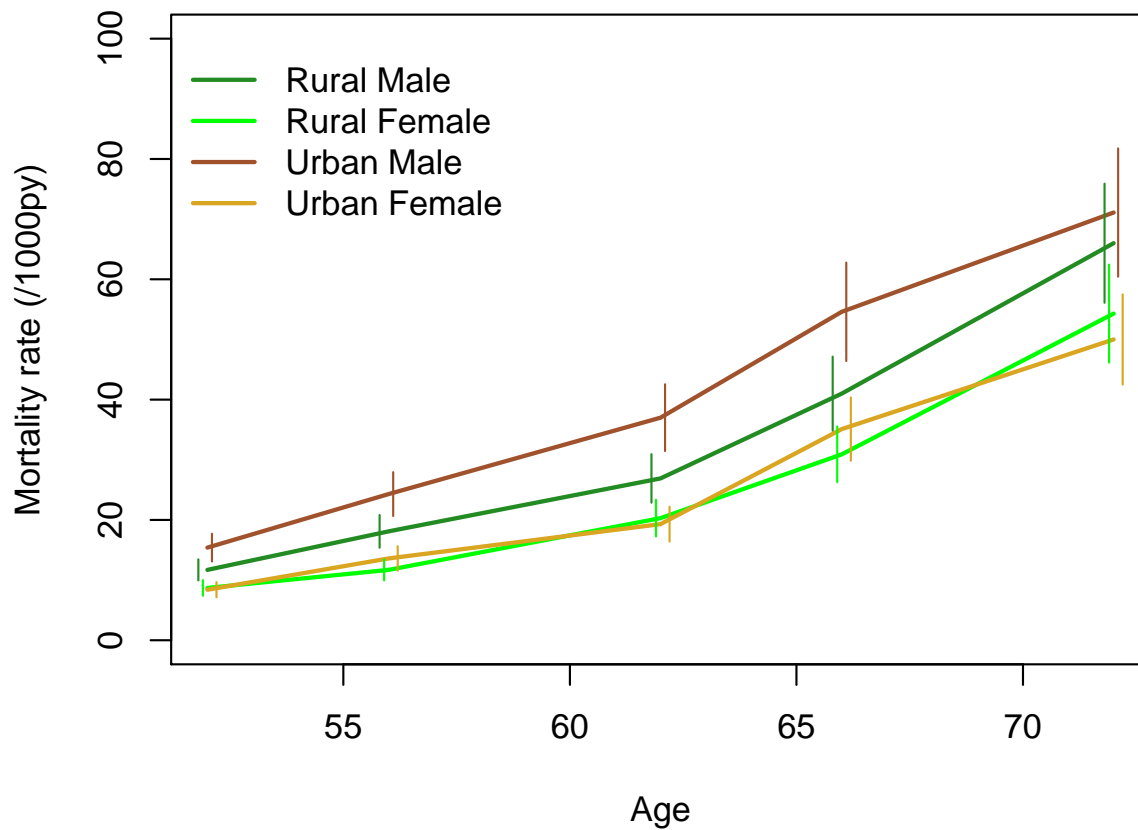
Example: Confidence intervals

We draw the confidence intervals with the `segments()` function.

A common strategy is to plot the error bars leading up from upper curves and down from lower curves, but this is where they are least useful.

Transparency is useful to allow overlap, but requires bands rather than intervals

Only a few formats support transparent colors (eg PNG and PDF) and software may not support it (R does only for PDF). Colors are specified as RGBA, where the A or α channel is 1 for completely opaque and 0 for completely transparent.

Example: Confidence intervals
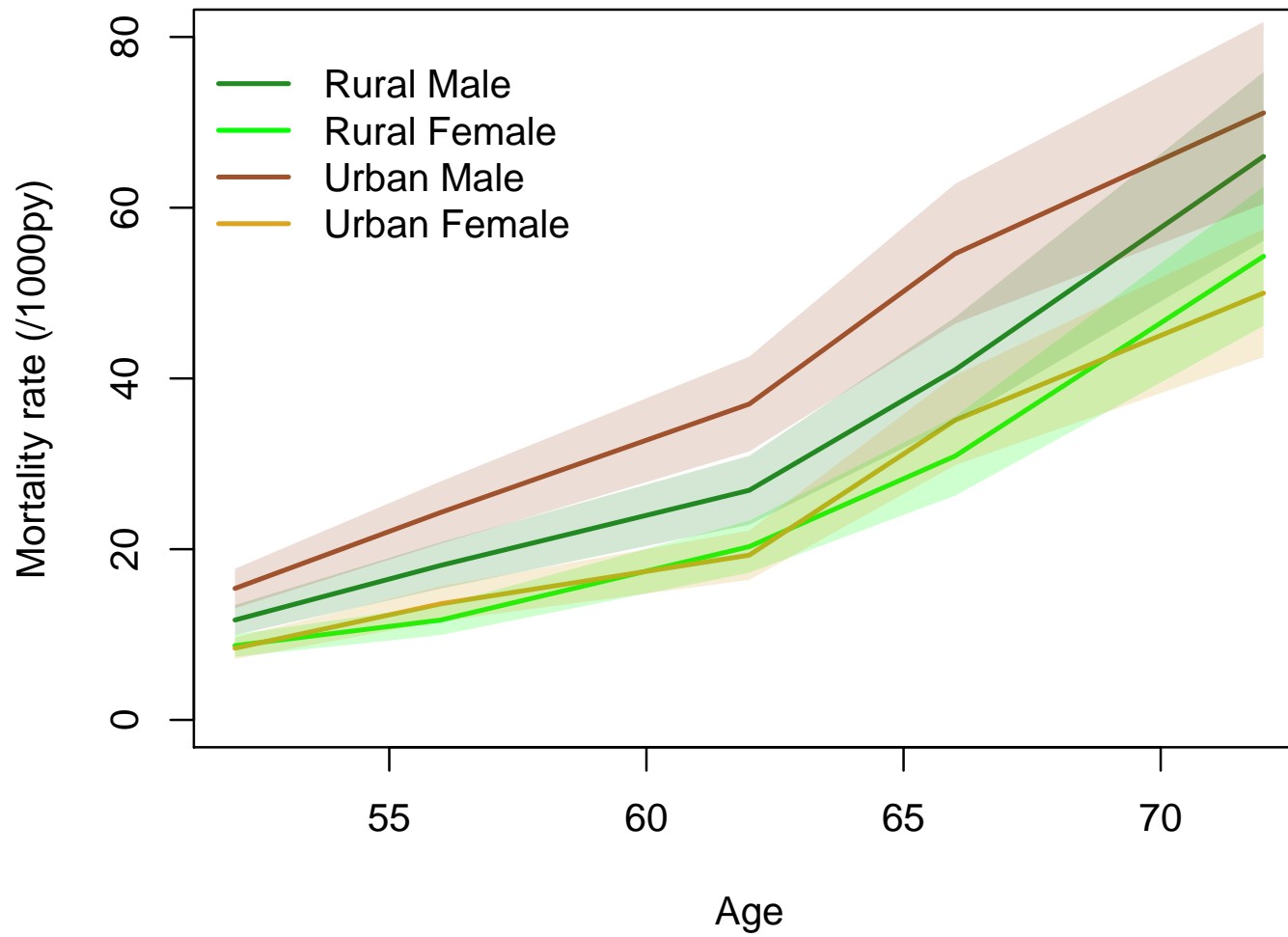
Confidence intervals and even simultaneous confidence bands do not necessarily reflect uncertainty in the shape of a curve correctly.

For example, in a plot of heart disease risk vs blood pressure we might see an increased risk for the lowest blood pressures and want to know if this increase is real.

This requires simulating or bootstrapping realisations of the curve to compare to the actual curve. Simulation is most useful in a testing context, where we can generate curves under the null hypothesis.
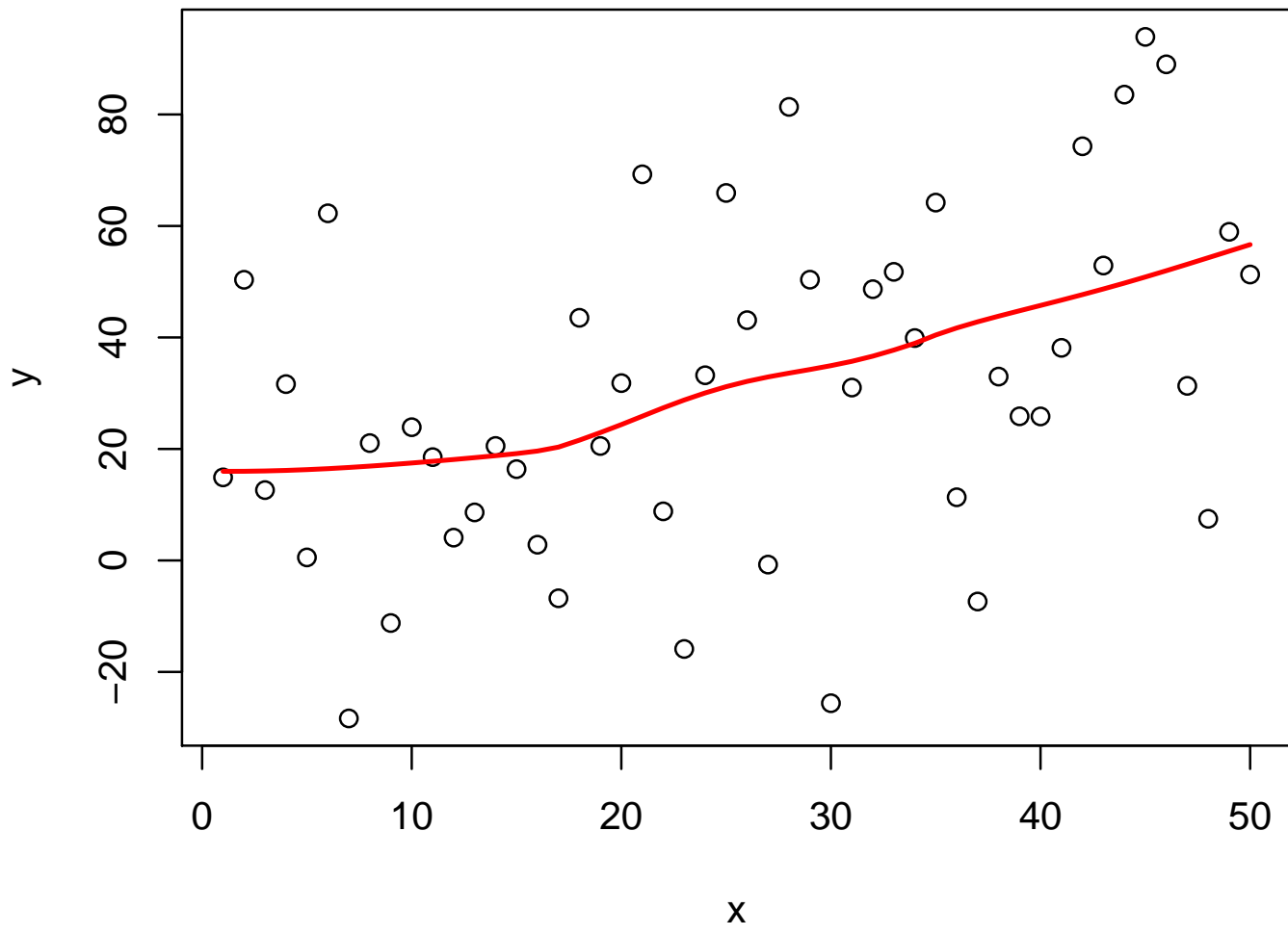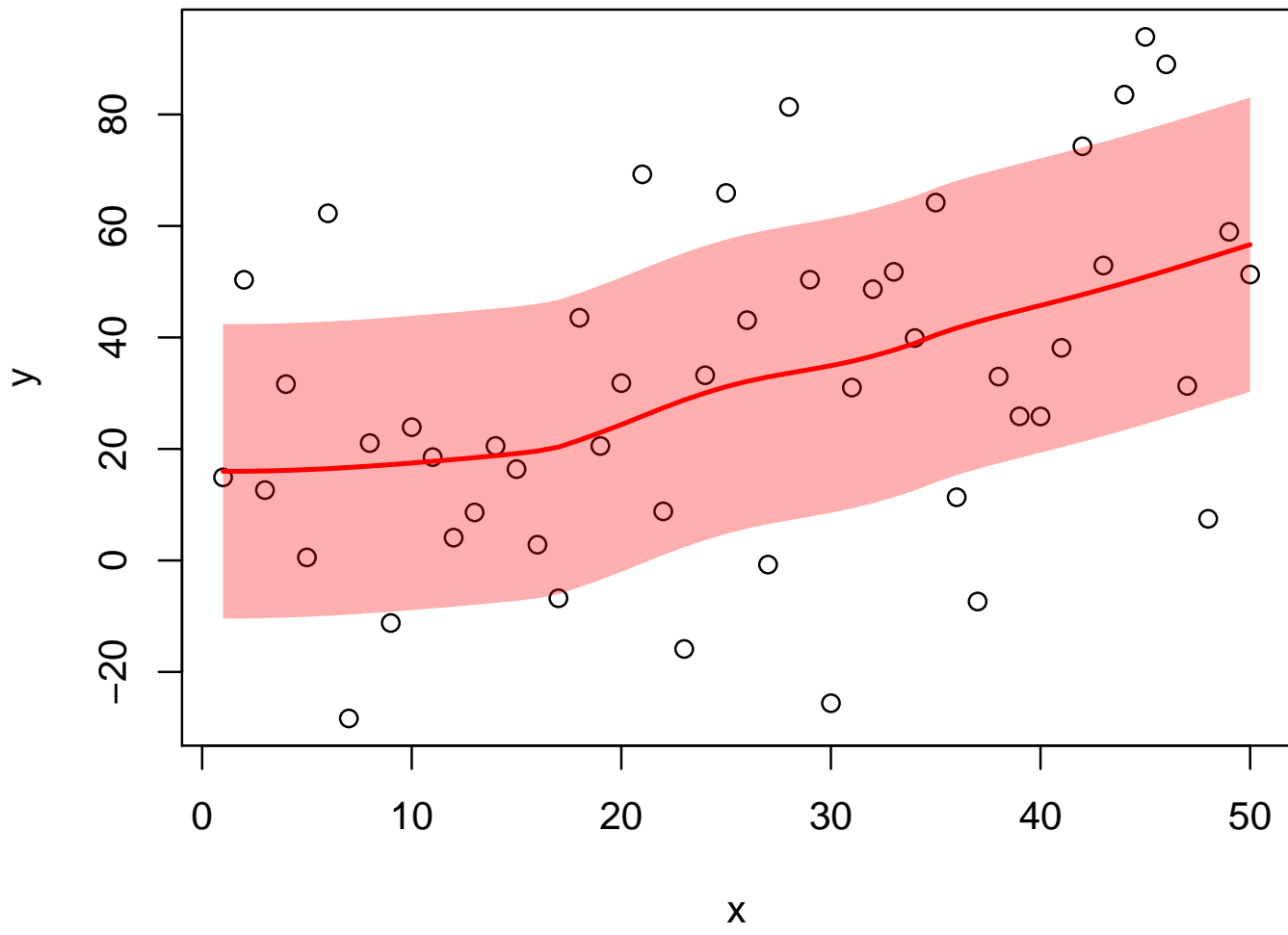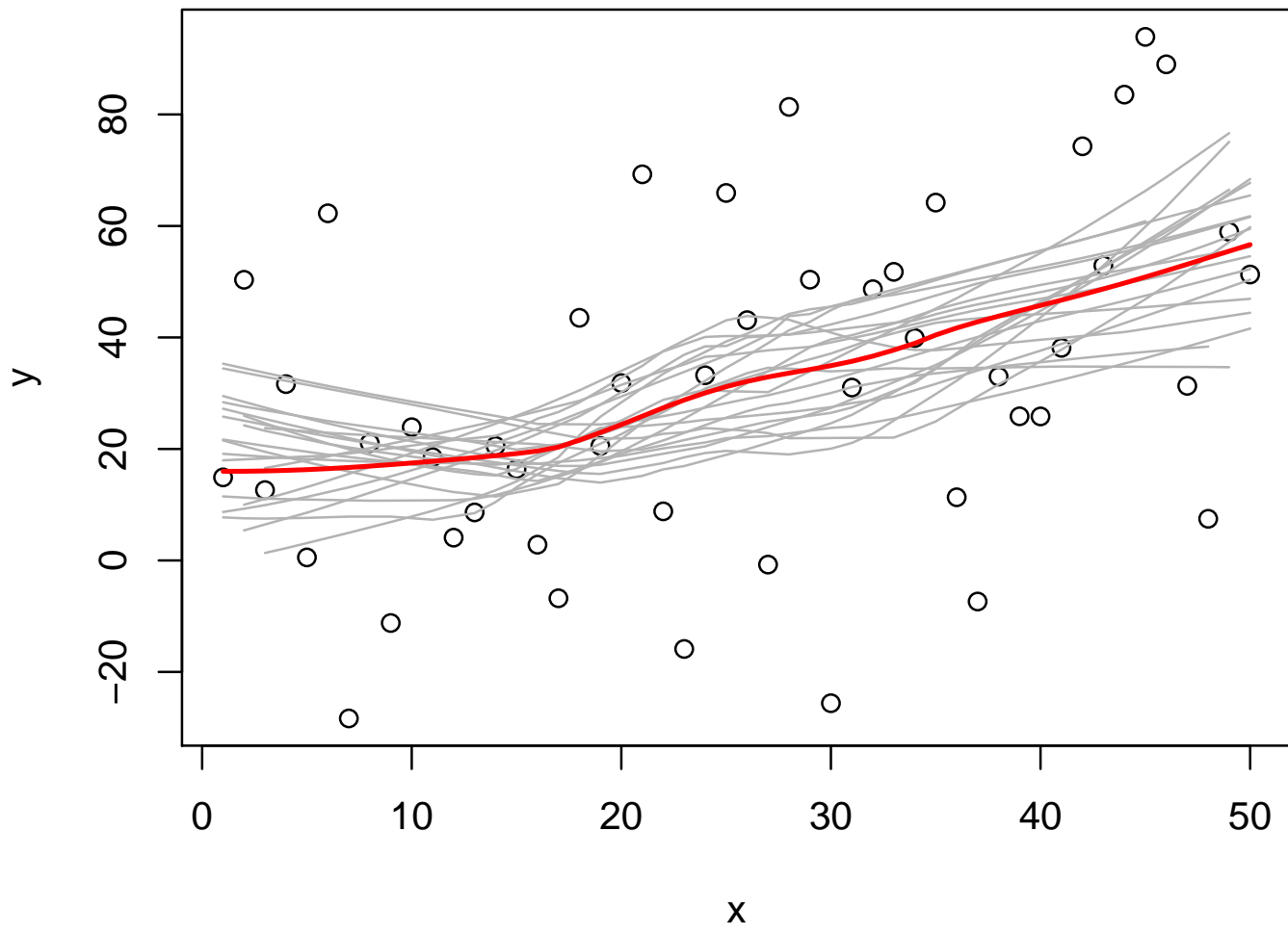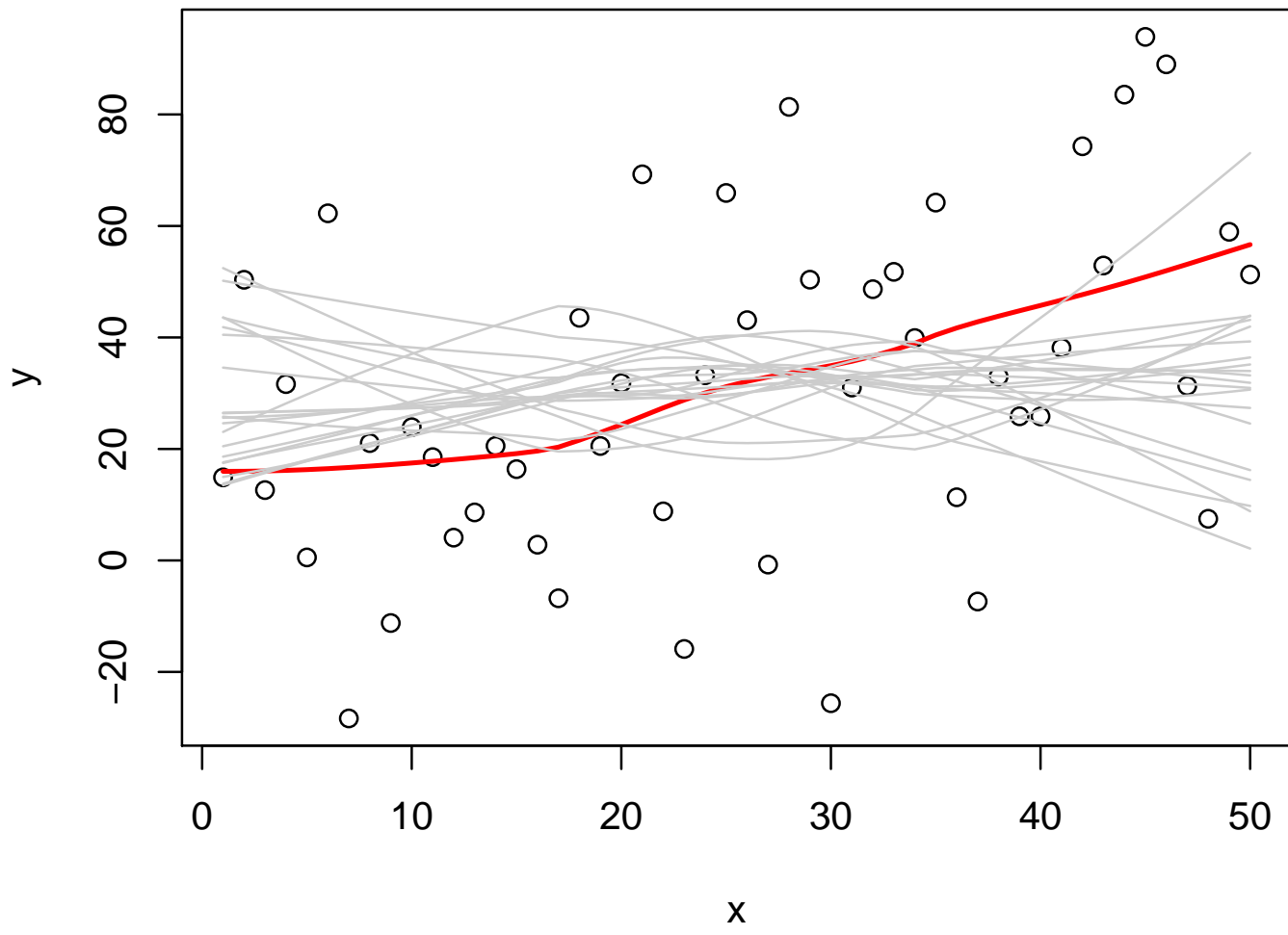
Code

```r
pdf("uncertainty.pdf",height=6,width=7,version="1.4")
x<-runif(50)
y<-rnorm(50)+2*x
ii<-order(x)
x<-x[ii]
y<-y[ii]
ll<-lowess(x,y)
plot(x,y,ylim=c(-3,6))
lines(ll,col="red",lwd=2)
s<-sqrt(var(ll$y-y))
polygon(c(x,rev(x)), c(ll$y+2*s,rev(ll$y-2*s)),
        col="#FF000070", border=NA)
plot(x,y,ylim=c(-3,6))
lines(ll,col="red",lwd=2)
replicate(10, {index<-sample(50,replace=TRUE);
              lines(lowess(x[index],y[index]),col="grey")})
```

Code

```
plot(x,y,ylim=c(-3,6))
lines(ll,col="red",lwd=2)
replicate(10, {xindex<-sample(50,replace=TRUE);
               yindex<-sample(50,replace=TRUE);
               lines(lowess(x[xindex],y[yindex]),col="grey")})
dev.off()
```

Conditioning plots

Ozone is a secondary pollutant, it is produced from organic compounds and atmostpheric oxygen in reactions catalyzed by nitrogen oxides and powered by su nlight.

However, looking at ozone concentrations in NY in summer we see a non-monotone relationship with sunlight

Conditioning plots
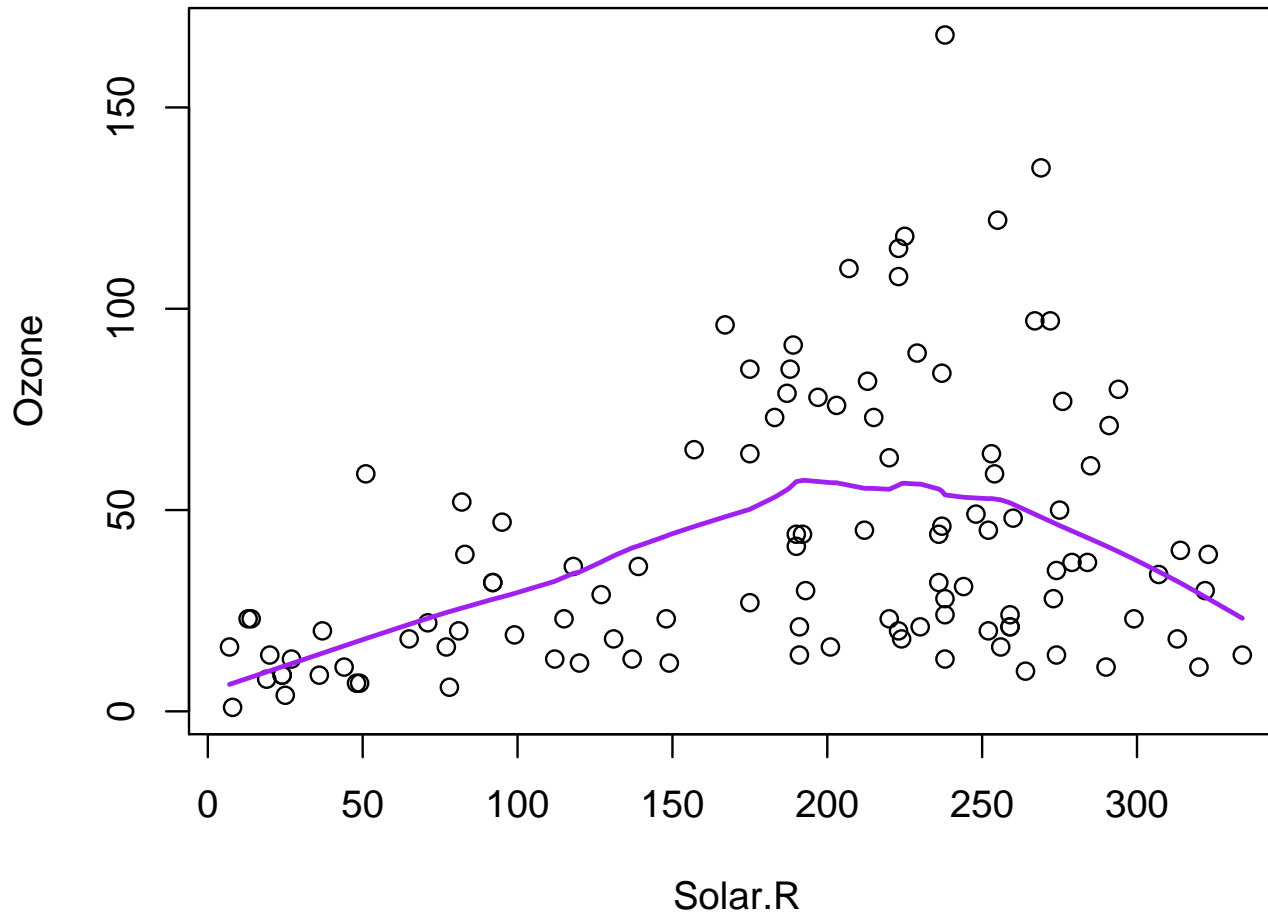
Conditioning plots

Here we draw a scatterplot of `Ozone` vs `Solar.R` for various subranges of `Temp` and `Wind`. A simple version of what is possible with the Trellis system.

```
data(airquality)
coplot(Ozone ~ Solar.R | Temp * Wind, number = c(4, 4),
     data = airquality,
    pch = 21, col = "goldenrod", bg = "goldenrod")
```

Conditioning plots
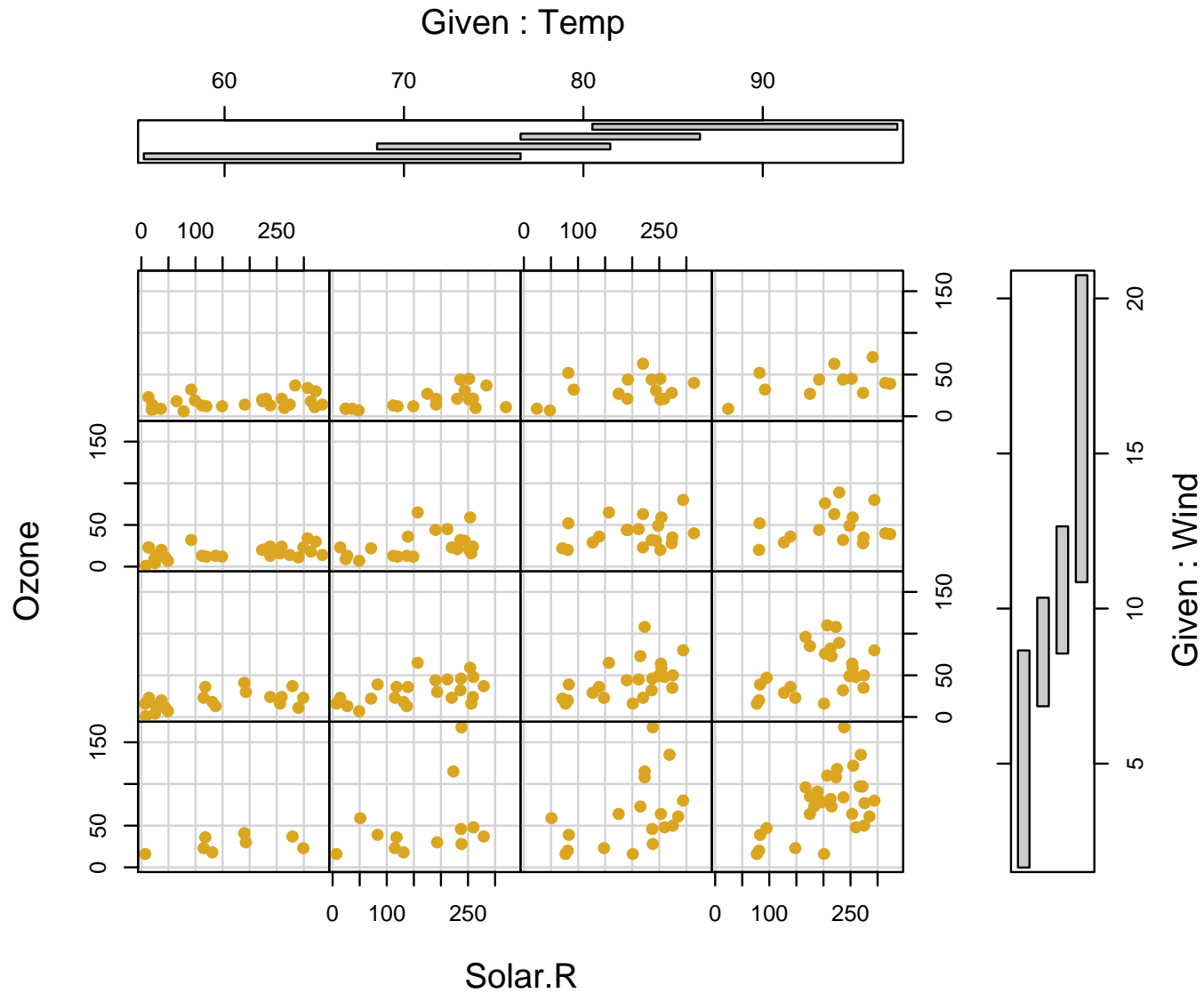
Trellis

The best-known system for producing conditioning plots is Cleveland's Trellis, in S-PLUS. There is a similar system based on Cleveland's research in R, called lattice.

Trellis is not restricted to conditioning scatterplots: it allows histograms, bo xplots, barcharts, and even 3-d plots.

One dramatic example of the usefulness of these plots is the following graph of some repeatedly analyzed data on field trials of barley varieties.

```
library(lattice)
data(barley)
dotplot(variety ~ yield | site, data = barley, groups = year,
    key = simpleKey(levels(barley$year), space = "right"))
```

Dotplots: barley varieties
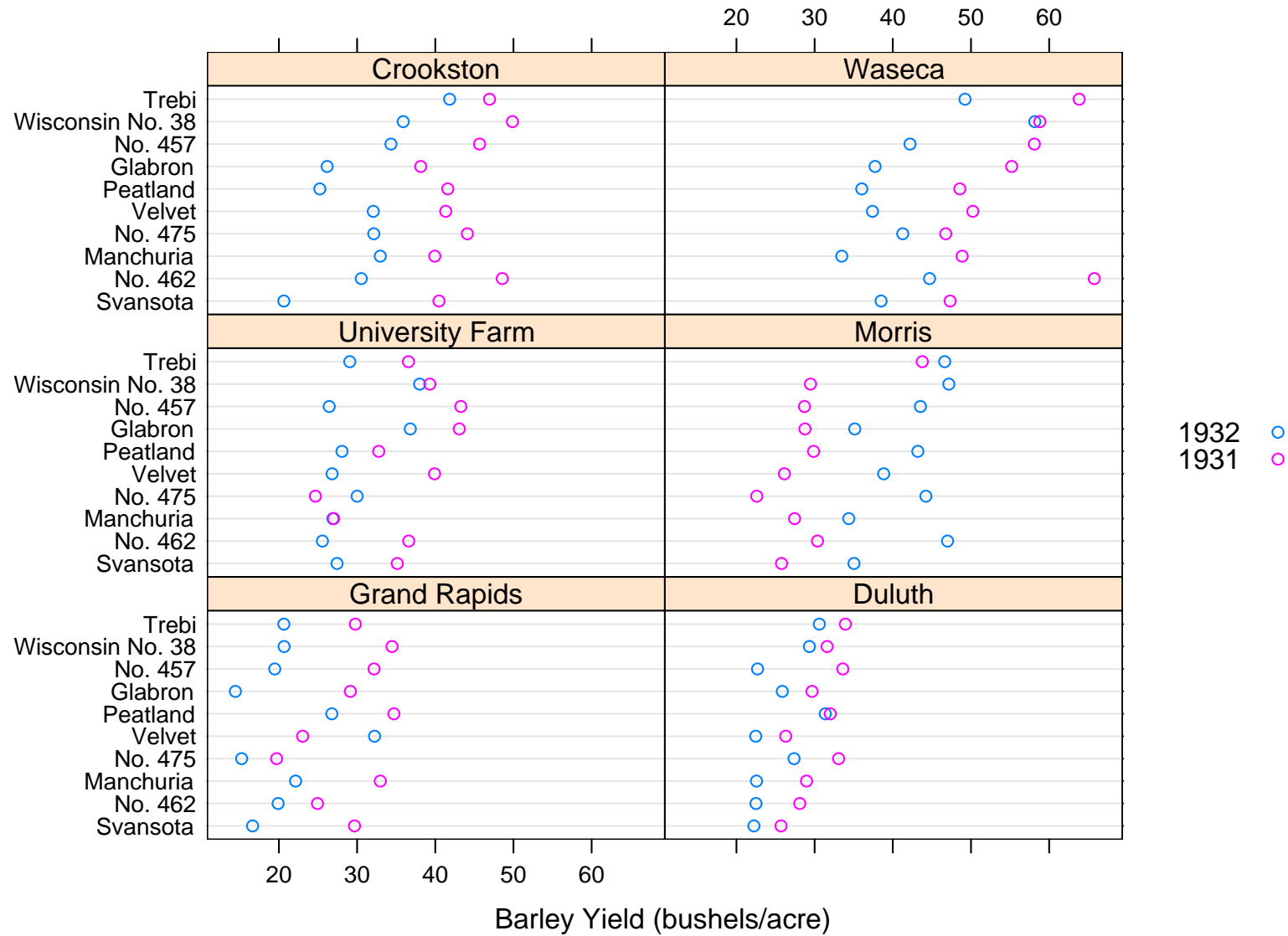
Syntax notes

Trellis graphics are specified by a formula:

```
response~predictor|conditioning*variables
```

The conditioning variables must be discrete or be shingles produced with a function such as `equal.count`.

Some Trellis functions (eg `histogram`) do not have a response variable, and some (eg `cloud`, `levelplot`) have more than one predictor variable.

The panel function that draws each subplot can be customised either by specifying options or by writing a new panel function.

Read Cleveland's Visualizing Data for examples (and because everyone producing statistical graphics should read it).

R FAQ 7.22

Why do lattice/trellis graphics not work?

The most likely reason is that you forgot to tell R to display the graph. Lattice functions such as `xyplot()` create a graph object, but do not display it (the same is true of Trellis graphics in S-Plus). The `print()` method for the graph object produces the actual display. When you use these functions interactively at the command line, the result is automatically printed, but in `source()` or inside your own functions you will need an explicit `print()` statement.

Coplot: NY ozone

We can produce a conditioning plot of NY summer ozone with the lattice package

```
xyplot(Ozone~Solar.R|equal.count(Temp,4)*equal.count(Wind,4),
    data=airquality,
    strip=strip.custom(var.name=c("Temperature","Wind speed")))
```

The labelling is slightly different from the coplot version

Coplot: NY ozone
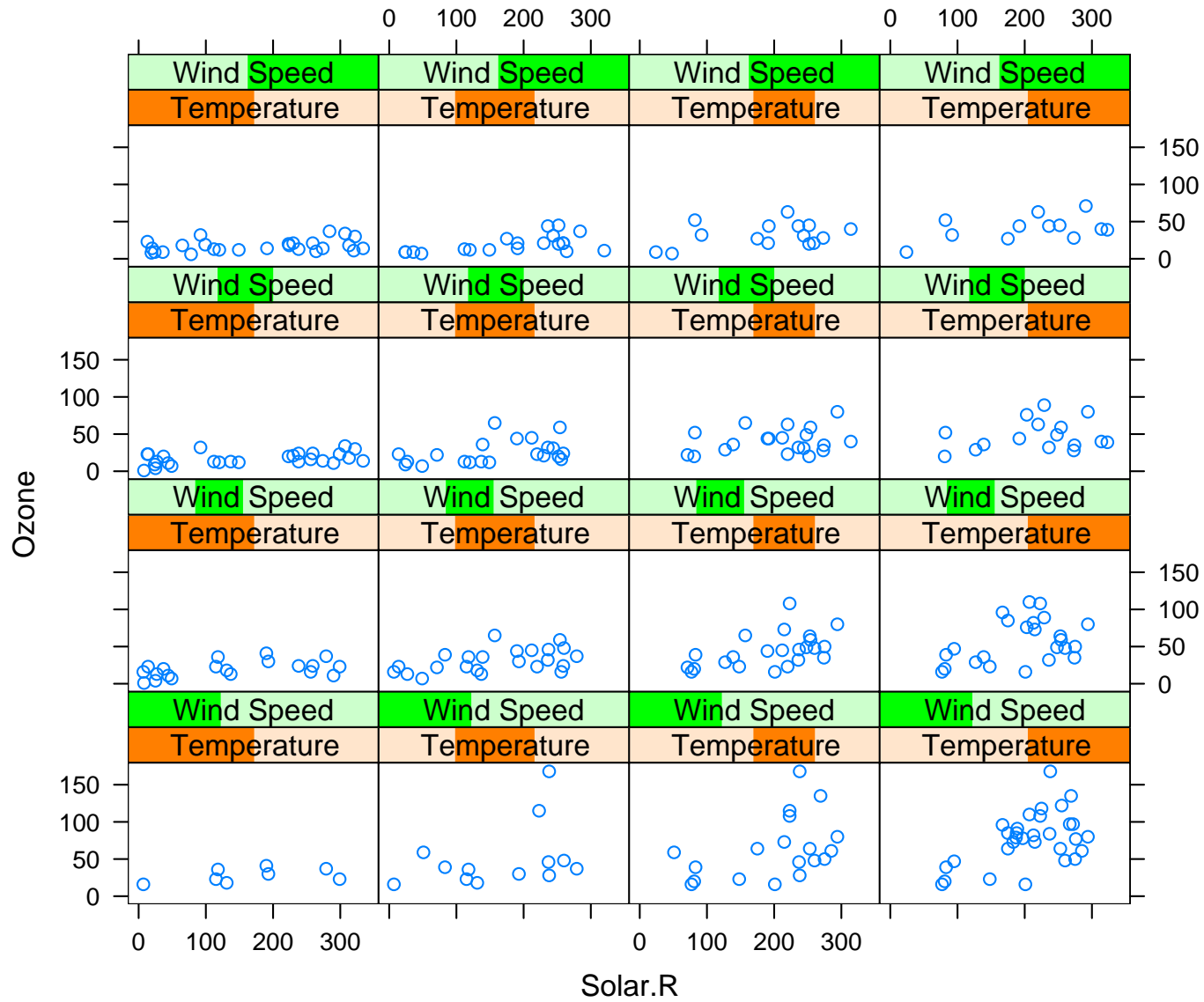
Stereo pairs

As an example of more advanced things that can be done, here is a pointless example of three-d graphics

```
par.set <-list(axis.line = list(col = "transparent"),
            clip = list(panel = FALSE))
print(cloud(Sepal.Length ~ Petal.Length * Petal.Width,
            data = iris, cex = .8,
            groups = Species,
            main = "Stereo",
            screen = list(z = 20, x = -70, y = 3),
            par.settings = par.set),
        split = c(1,1,2,1), more = TRUE)
print(cloud(Sepal.Length ~ Petal.Length * Petal.Width,
            data = iris, cex = .8,
            groups = Species,
            main = "Stereo",
            screen = list(z = 20, x = -70, y = 0),
            par.settings = par.set),
        split = c(2,1,2,1))
```

Stereo pairs
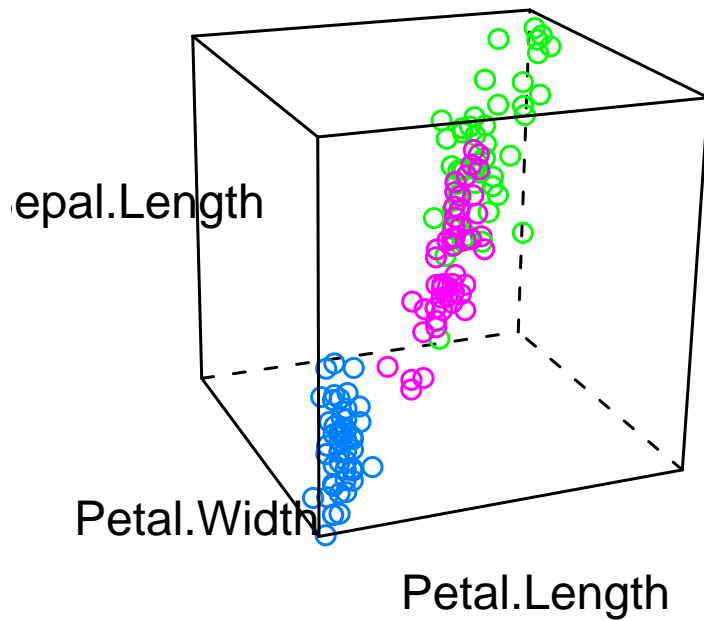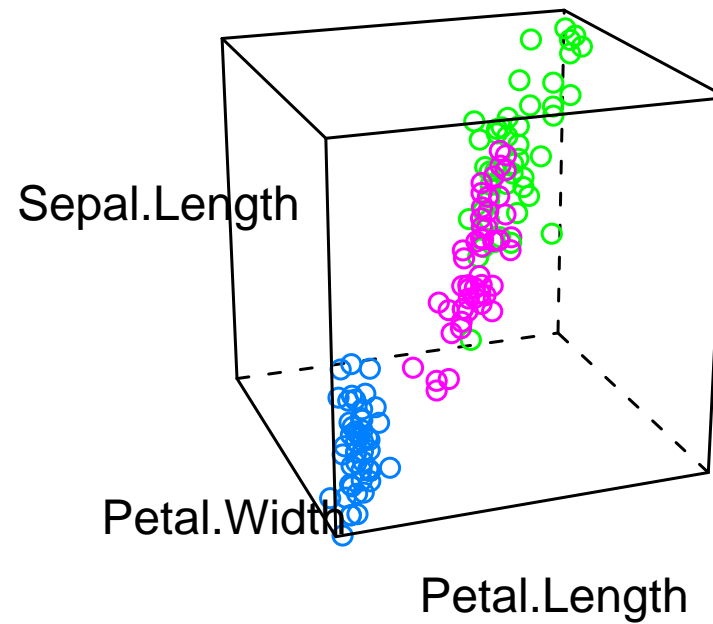
Mathematical annotation

An expression can be specified in R for any text in a graph (`help(plotmath)` for details). Here we annotate a graph drawn with `polygon`.

```
x<-seq(-10,10,length=400)
y1<-dnorm(x)
y2<-dnorm(x,m=3)
par(mar=c(5,4,2,1))
plot(x,y2,xlim=c(-3,8),type="n",
    xlab=quote(Z==frac(mu[1]-mu[2],sigma/sqrt(n))),
    ylab="Density")
polygon(c(1.96,1.96,x[240:400],10),
        c(0,dnorm(1.96,m=3),y2[240:400],0),
        col="grey80",lty=0)
lines(x,y2)
lines(x,y1)
polygon(c(-1.96,-1.96,x[161:1],-10),
        c(0,dnorm(-1.96,m=0),y1[161:1],0),
        col="grey30",lty=0)
polygon(c(1.96,1.96,x[240:400],10),
        c(0,dnorm(1.96,m=0),y1[240:400],0),
        col="grey30")
```

Mathematical annotation

```
legend(4.2,.4,fill=c("grey80","grey30"),
    legend=expression(P(abs(Z)>1.96,H[1])==0.85,
        P(abs(Z)>1.96,H[0])==0.05),bty="n")
text(0,.2,quote(H[0]:~~mu[1]==mu[2]))
text(3,.2,quote(H[1]:~~mu[1]==mu[2]+delta))
```

Mathematical annotation
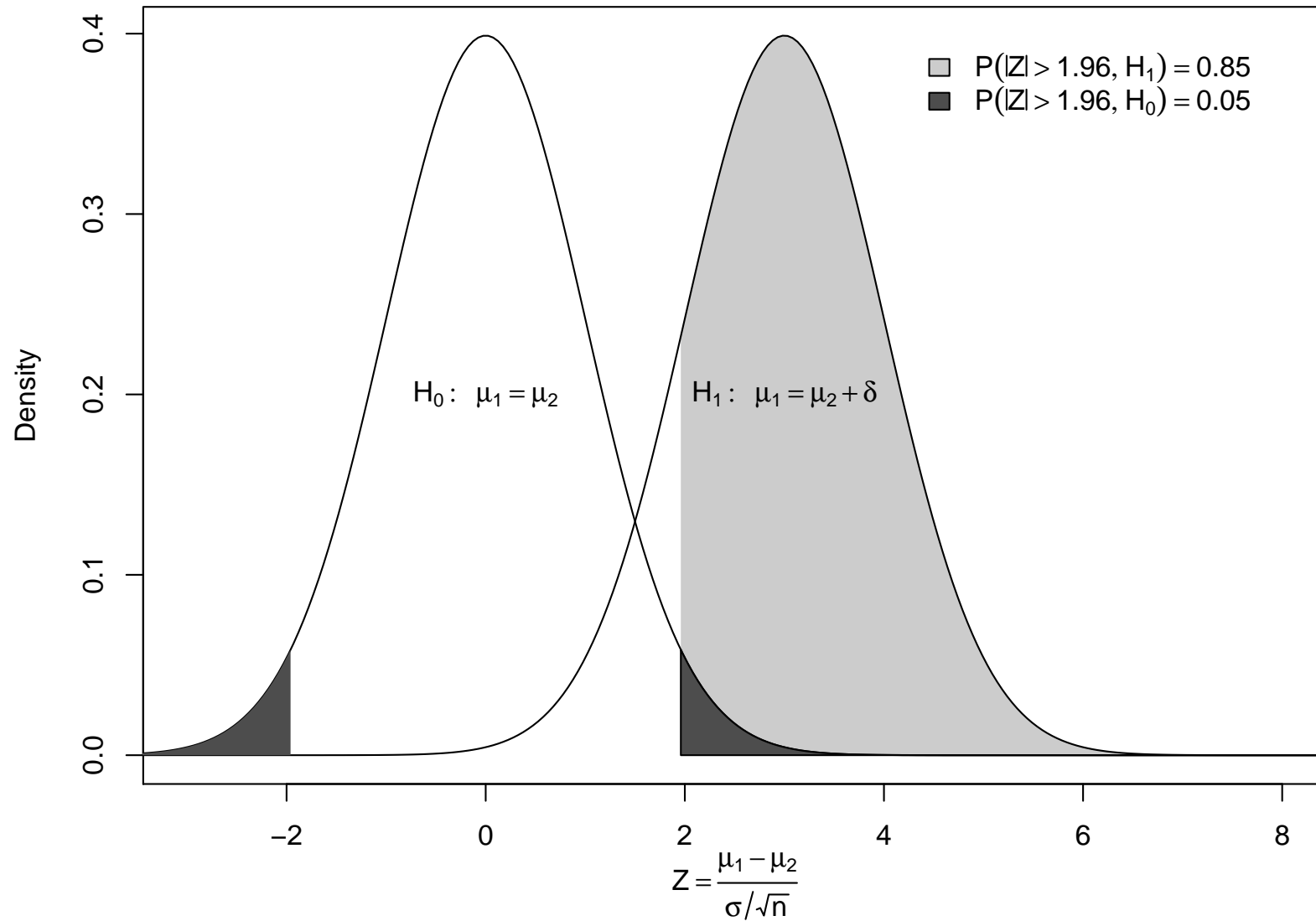
Managing code and data

Options for storage

- Workspace. When R starts it will read in the file `.RData`, and when it exits you are given the chance to save the workspace to that file (you can save it at any time with `save.image()`). This saves everything except loaded packages and is the equivalent of the `.Data` directory in S-PLUS.

- Binary files. The `save()` command puts specified functions and data in a binary file. This file can be `attach()`ed (like a directory in S-PLUS) or `load()`ed.

- Source code. Instead of saving results and data they can be recreated as needed from source code.

Multiple projects

There are two extreme ways to handle multiple projects in R

- Store each project in a separate directory and use the `.RData` file to hold everything. If you want to share functions or objects with another project then explicitly export and import them. The `.RData` file is primary; any transcripts or source files are documentation.
- Store everything as source code. For every piece of analysis have a file that reads in data, processes it, and possibly `save`s a modified version to a new file. The source is primary, any saved files are just a labour-saving device.

The same choices apply to programming as well as data analysis, and to other dialects of S.

Workspace is primary

The first method is common among new users of S-PLUS, partly because S-PLUS automatically saves your workspace. Many of us subsequently find that we aren't sufficiently organised to be sure that we keep records of how every analysis was done.

This approach is (even) riskier in R than in S-PLUS.

- In R the workspace is only saved when you explicitly ask for it; in S-PLUS it is saved frequently
- In R a corrupted `.RData` is likely to be completely unreadable, in S-PLUS many objects will still be recoverable.

It makes sense for short-term projects, especially if data loss is not critical, or for very careful people.

Source is primary

Managing projects is easy when everything can be reconstructed from source files. These files and intermediate data files can be stored in a project directory where they are easy to find and are automatically time-stamped by the operating system.

Emacs Speaks Statistics (ESS) is particularly useful for this style of R use. With R running in an Emacs window you can run sections of code with a few keystrokes. The S-PLUS script window also offers many of the features of ESS.

Logging

One problem with interactive data analysis is keeping a good log of your analysis. This problem can be avoided by using interactive analysis only to construct a script that is then run to provide the final results.

Other options:

- ESS will capture a complete transcript, as will the Windows or Mac GUIs and the JGR GUI

- in R, `sink(filename, split=TRUE)` will send all output to both `filename` and the screen. Use `sink()` before quitting to close the file.

Note that none of these will log graphics output.

Merging and matching

The data for an analysis often do not come in a single file. Combining multiple files is necessary.

If two data sets have the same individuals in the same order, they can simply be pasted together side by side.

```
## CHS baseline data
baseline <- read.spss("I:/DISTRIB/BASEBOTH.SAV", to.data.frame=TRUE)
## Events data (eg death, heart attack, ...)
events <- read.spss("I:/SAVEFILES/EVSUM04.SAV", to.data.frame=TRUE)

if (!all(baseline$IDNO==events$IDNO)) {
    stop("PANIC: They don't match!")
} else {
    alldata <- cbind(baseline, events[,c("TTODTH","DEATH",
                     "TTOMI","INCMI")])
}
```

Merging: order

The data might need to be sorted first

```
index1 <- order(baseline$IDNO)
baseline <- baseline[index1,]
index2 <- order(events$IDNO)
events <- events[index2,]
if (!all(baseline$IDNO==events$IDNO)) {
   stop("PANIC: They still don't match!")
} else {
   alldata <- cbind(baseline, events[,c("TTODTH","DEATH",
                      "TTOMI","INCMI")])
}
```

Note that `order(baseline$IDNO)` gives a subset of row numbers containing all the rows but in a different (increasing) order.

Merging: merge

Or there might be different rows in the two data sets

- Some people are missing from one or other data set (eg baseline and year 5 visits)

- Some people have multiple records in one data set (eg baseline data and all hospitalisations

The merge function can do an database outer join, giving a data set that has all the possible matches between a row in one and a row in the other

Merging: merge

```
combined <- merge(baseline, hospvisits, by="IDNO", all=TRUE)
```

- `by=IDNO` says that the IDNO variable indicates individuals who should be matched.

- `all=TRUE` says that even people with no records in the `hospvisits` data set should be kept in the merged version.

How does it work: match

You could imagine a dumb algorithm for merging

```
for(row in firstdataset){
    for(otherrow in seconddataset){
      if (row$IDNO==otherrow$IDNO)
        ##add the row to the result
 }
}
```

More efficiently, the match function gives indices to match one variable to another

```
> match(c("B","I","O","S","T","A","T"),LETTERS)
[1]  2  9 15 19 20  1 20
> letters[match(c("B","I","O","S","T","A","T"),LETTERS)]
[1] "b" "i" "o" "s" "t" "a" "t"
```

Reshaping

Sometimes data sets are the wrong shape. Data with multiple observations of similar quantities can be in long form (multiple records per person) or wide form (multiple variables per person).

Example: The SeattleSNPs genetic variation discovery resource supplies data in a format

```
 SNP    sample al1 al2
000095 D001 C T
000095 D002 T T
000095 D003 T T
```

so that data for a single person is broken across many lines. To convert this to one line per person

```
> data<-read.table("http://pga.gs.washington.edu/data/il6
                   /ilkn6.prettybase.txt",
                   col.names=c("SNP","sample","allele1","allele2"))
> dim(data)
[1] 2303    4
> wideData<-reshape(data, direction="wide", idvar="sample",
                    timevar="SNP")
> dim(wideData)
[1] 47 99
> names(wideData)
 [1] "sample"        "allele1.95"    "allele2.95"    "allele1.205"
 [5] "allele2.205"   "allele1.276"   "allele2.276"   "allele1.321"
 [9] "allele2.321"   "allele1.657"   "allele2.657"   "allele1.1086"
...
```

- `direction="wide"` says we are going from long to wide format

- `idvar="sample"` says that sample identifies the rows in wide format

- `timevar="SNP"` says that SNP identifies which rows go into the same column in wide form (for repeated measurements over time it would be the time variable)

Broken down by age and sex

A common request for Table 1 or Table 2 in a medical paper is to compute means and standard deviations, percentages, or frequency tables of many variables broken down by groups (eg case/control status, age and sex, exposure,...).

That is, we need to apply a simple computation to subsets of the data, and apply it to many variables. One useful function is by(), another is tapply(), which is very similar (but harder to remember).

```
> by(airquality$Ozone, list(month=airquality$Month),
        mean, na.rm=TRUE)
month: 5
[1] 23.61538
-----------------------------------------------------------------
month: 6
[1] 29.44444
-----------------------------------------------------------------
month: 7
[1] 59.11538
-----------------------------------------------------------------
month: 8
[1] 59.96154
-----------------------------------------------------------------
month: 9
[1] 31.44828
```

Notes

- The first argument is the variable to be analyzed.

- The second argument is a list of variable defining subsets. In this case, a single variable, but we could do `list(month=airquality$Month, toohot=airquality$Temp>85)` to get a breakdown by month and temperature

- The third argument is the analysis function to use on each subset

- Any other arguments (na.rm=TRUE) are also given to the analysis function

- The result is really a vector (with a single grouping variable) or array (with multiple grouping variables). It prints differently.

Confusing digression: str()

How do I know it is an array? Because str() summarises the internal structure of a variable.

```
> a<- by(airquality$Ozone, list(month=airquality$Month,
                              toohot=airquality$Temp>85),
                              mean, na.rm=TRUE)
> str(a)
 by [1:5, 1:2] 23.6 22.1 49.3 40.9 22.0 ...
 - attr(*, "dimnames")=List of 2
  ..$ month : chr [1:5] "5" "6" "7" "8" ...
  ..$ toohot: chr [1:2] "FALSE" "TRUE"
 - attr(*, "call")= language by.data.frame(data =
     as.data.frame(data), INDICES = INDICES,
     FUN = FUN, na.rm = TRUE)
 - attr(*, "class")= chr "by"
```

One function, many variables

There is a general function, apply() for doing something to rows or columns of a matrix (or slices of a higher-dimensional array).

```
> apply(psa[,1:8],2,mean,na.rm=TRUE)
          id       nadir       pretx           ps         bss       grade
   25.500000   16.360000  670.751163    80.833333    2.520833    2.146341
       grade         age     obstime
    2.146341   67.440000   28.460000
```

In this case there is a special, faster, function `colMeans`, but the `apply` can be used with other functions such as `sd`, `IQR`, `min`,…

apply

- the first argument is an array or matrix or dataframe

- the third argument is the analysis function

- the second argument says which margins to keep (1=rows, 2=columns, ...), so 2 means that the result should keep the columns: apply the function to each column.

- any other arguments are given to the analysis function

There is a widespread belief that apply() is faster than a for() loop over the columns. This is a useful belief, since it encourages people to use apply(), but it is not true.

New functions

Suppose you want the mean and standard deviation for each variable. One solution is to apply a new function. Watch carefully,...

```
> apply(psa[,1:8], 2, function(x) c(mean=mean(x,na.rm=TRUE),
                            stddev=sd(x,na.rm=TRUE)))
              id    nadir     pretx        ps         bss      grade
mean    25.50000  16.3600  670.7512  80.83333  2.5208333  2.1463415
stddev  14.57738  39.2462 1287.6384  11.07678  0.6838434  0.7924953
              age   obstime
mean    67.440000  28.46000
stddev   5.771711  18.39056
```

New function

```
function(x) c(mean=mean(x,na.rm=TRUE),
              stddev=sd(x,na.rm=TRUE))
```

translates as: "If you give me a vector, which I will call `x`, I will mean it and sd it and give you the results"

We could give this function a name and then refer to it by name

```
mean.and.sd <- function(x) c(mean=mean(x,na.rm=TRUE),
                             stddev=sd(x,na.rm=TRUE))
apply(psa[,1:8], 2, mean.and.sd)
```

which would save typing if we used the function many times. Note that giving the function a name is not necessary, any more than giving 2 a name.

by() revisited

Now we know how to write simple functions we can use by() more generally

```
> by(psa[,1:8], list(remission=psa$inrem),
      function(subset) round(apply(subset, 2, mean.and.sd), 2))
remission: no
          id nadir   pretx    ps  bss grade   age obstime
mean    31.03 22.52  725.99 79.71 2.71  2.11 67.17   21.75
stddev  11.34 44.91 1362.34 10.29 0.52  0.83  5.62   15.45
-------------------------------------------------------------
remission: yes
          id nadir   pretx    ps  bss grade   age obstime
mean    11.29  0.53  488.45 83.57 2.07  2.23 68.14   45.71
stddev  12.36  0.74 1044.14 12.77 0.83  0.73  6.30   13.67
```

Notes

```
function(subset) round(apply(subset, 2, mean.and.sd), 2)
```

translates as "If you give me a data frame, which I will call `subset`, I will apply the `mean.and.sd` function to each variable, round to 2 decimal places, and give you the results"

Capturing output

To send text output to a file

```
sink("filename")
```

and to cancel it

```
sink()
```

- Error messages are not diverted.

- Use `sink("filename",split=TRUE)` to send output to the file and to the screen

To capture output in a variable, use `capture.output()`

```
> output <- capture.output(example(by))
> length(output)
[1] 107
> output[1]
[1] ""
> output[2]
[1] "by> require(stats)"
> output[3]
[1] "[1] TRUE"
```

Capturing pretty output

Having `chunks of output in typewriter font in the middle of the document` is convenient but you may want something prettier.

The `xtable()` function in the `xtable` package will produce LaTeX or HTML tables from matrices or from statistical model output. The HTML can be saved to a file and read into eg Word or Powerpoint.

Sweave

Sweave is a literate programming tool for data analysis. That is, it is a tool that translates a mixture of text and code into text and (numerical and graphical) output. Sweave is based on LATEX

Uses:

- Reproducible data analysis reports

- Teaching: it ensures the output matches the input

- HOWTO explanations: the code can be automatically rerun so it remains up to date.

The `R2HTML` package provides some similar features with HTML output rather than LaTeX.

Example: forest plots

An epidemiologist in Australia wanted to know how to do meta-analyses and draw forest plots for non-binary data in R (her other software could only handle binary outcomes).

You have the `.Rnw`, `.tex`, and `.pdf` files. In the `.Rnw` input file the code chunks are delimited by <<>>= and @. In the `.tex` file the input code is in an `Sinput` environment and the output in `Soutput`.

Functions

We saw simple functions earlier.

```
function(x) c(mean = mean(x), stddev = sd(x))
```

Functions are more important in R than in other statistical packages and more important than in many programming languages.

S, and now R, are deliberately designed to blur the distinction between users and programmers. R is a good language for rapid development of tools: whether the tool is a customized barplot or a package of survey functions.

This comes at the expense of speed and memory efficiency, but it doesn't take many hours of programming time to pay for a gigabyte of memory.

Example: ROC curve

Plotting the sensitivity and specificity of a continuous variable as a predictor of a binary variable in an ROC curve.

```
ROC <- function(test, disease){
  cutpoints <- c(-Inf, sort(unique(test)), Inf)
  sensitivity<-sapply(cutpoints,
      function(result) mean(test>result & disease)/mean(disease))
  specificity<-sapply(cutpoints,
      function(result) mean(test<=result & !disease)/mean(!disease))
  plot(sensitivity, 1-specificity, type="l")
  abline(0,1,lty=2)
  return(list(sens=sensitivity, spec=specificity))
}
```

Example: ROC curve

```
> x<-rnorm(100,mean=0)
> y<-rnorm(100, mean=1)
> isx<-rep(c(TRUE,FALSE),each=100)
> ROC(c(x,y), isx)
$sens
  [1] 1.00 0.99 0.98 0.97 0.96 0.95 0.94 0.93 0.93 0.93 0.92 0.91 0.9
 [21] 0.85 0.84 0.83 0.82 0.81 0.80 0.79 0.78 0.77 0.76 0.75 0.74 0.7
...
```

Example: ROC curve
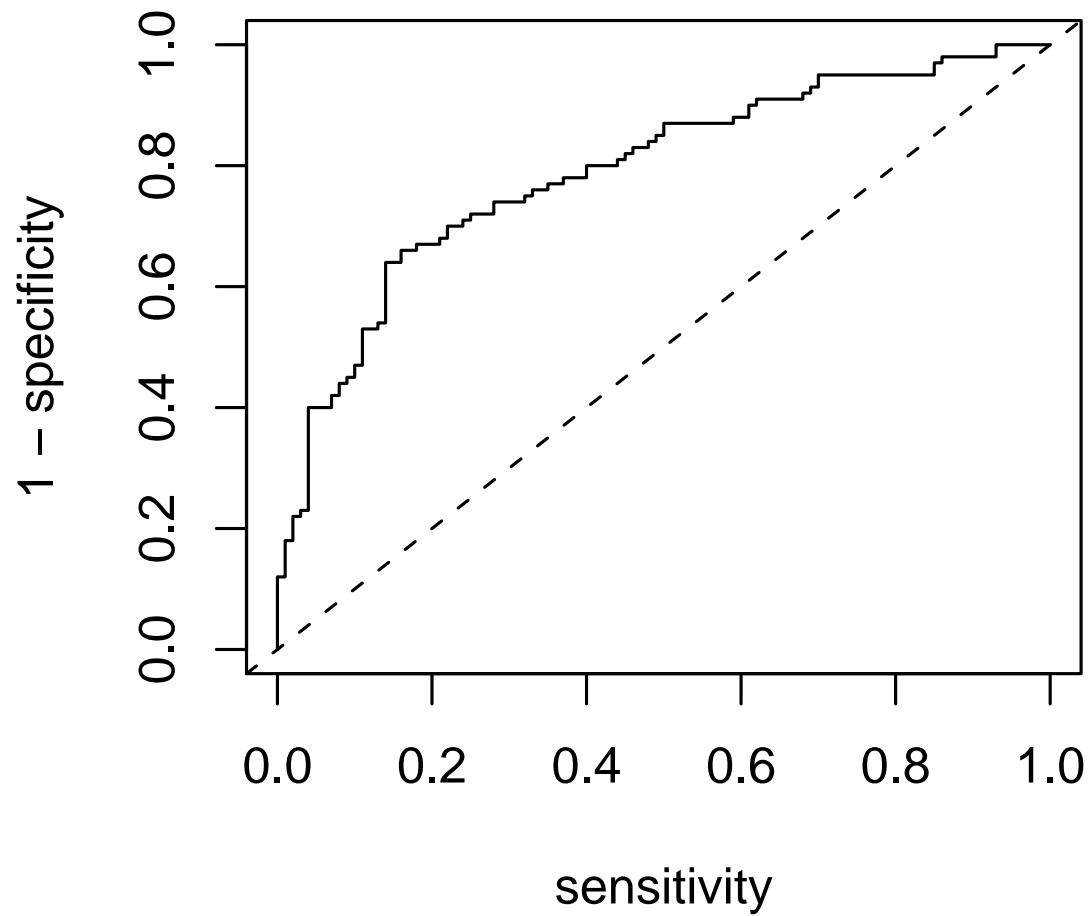
Notes

- `sort` sorts a vector, so `sort(unique(test))` are the ordered observed values. `-Inf` and `Inf` are added to ensure that the curve gets to (0,0) and (1,1).

- `disease` is a logical variable (or treated as one). `!disease` means "not disease"

- Variables created inside the function are local

- In R, variables that are visible where a function is defined (eg `test` and `disease`) will be visible inside the function. This isn't true in S-PLUS, where this ROC function won't work. Read 3.3.1 and 7.12 in the R FAQ if you are curious.

  In S-PLUS we would have to write

Notes

```
sensitivity<-sapply(cutpoints,
    function(result,test, disease)
            mean(test>result & disease)/mean(disease),
    test=test,
    disease=disease)
```

making this a less attractive approach.

- `return()` is optional. Recall that every expression in R has some value: the value of the last expression will be returned.

- `rep()` repeats things. Two most common versions are `rep(something, times)` and `rep(somethings, each=times)`, but there are more complex versions.

Theoretical note

In principle, the use of user-written functions and second-order functions such as `apply()` and `by()` makes it possible never to change the value of a variable.

Variables can then be thought of as names for values, as in math; rather than storage for values, as in C or Fortran.

The extremist form of this position is called "functional programming". It is a useful idea in moderation: code is easier to understand when a variable doesn't change values.

Historical and cultural note

There have always been multiple versions of the assignment operator available in R and S, not always the same ones.

- In the Old Days, R and S-PLUS allowed <- and _. The underscore actually printed as a left arrow on some Bell Labs terminals.

- In S-PLUS since 5.0 and R since 1.4.0 = has been allowed as an alternative to <-.

- In R since 1.8.0 the _ has been removed as an assignment operator and is now an ordinary character that can be used in variable names.

In R, $=$ can't be used in some places (where you probably wouldn't have meant to do an assignment), so that

```
a = 4
if(a = 5) b = 4
print(a)
```

gives 5 on S-PLUS and a syntax error in R.

I use <-, but there's nothing wrong with using = if you prefer. Do get used to leaving spaces around it.

Don't use _, even in S-PLUS where it is legal. You can't imagine how much some people hate it.

Example: computing the median

Suppose we wanted to write a function to compute the median. A simple algorithm is to sort the data and take the middle element.

```
ourmedian <- function(x){
        n<-length(x)
        return(sort(x)[(n+1)/2])
      }
```

Notes

- `sort()` sorts a vector

- `return()` is optional. Remember that everything is an expression and produces a value. If there is no `return()` statement the value of the function is the value of the last expression evaluated.

For even sample sizes we might prefer the average of the two middle values

```
ourmedian <- function(x){
        n<-length(x)
        if (n %% 2==1) ## odd
           sort(x)[(n+1)/2]
        else {           ## even
           middletwo <- sort(x)[(n/2)+0:1]
           mean(middletwo)
        }
       }
```

We need to handle missing values

```r
ourmedian <- function(x, na.rm=FALSE){
        if(any(is.na(x))) {
            if(na.rm)
                x<-x[!is.na(x)]
            else
                return(NA)
        }
        n<-length(x)
        if (n %% 2==1) ## odd
            sort(x)[(n+1)/2]
        else {            ## even
            middletwo <- sort(x)[(n/2)+0:1]
            mean(middletwo)
        }
    }
```

We might also want to

- Check that `x` is numeric, so that a median makes sense

- Check that `n` is not 0

The built-in function also takes advantage of an option to `sort()` that stops sorting when specific indices (eg `(n+1)/2`) are correct. This is faster for large vectors (by 1sec=50% for $n = 10^6$).

Simulating Data

S has a wide range of functions to handle mathematical probability distributions

- `pnorm` gives the Normal cumulative distribution function
- `qnorm` is the inverse: the Normal quantile function
- `dnorm` is the Normal density
- `rnorm` simulates data from Normal distributions

Similar sets of `p,q,d,r` functions for Poisson, binomial, t, F, hypergeometric, χ^2, Beta,...

Also `sample` for sampling from a vector, `replicate` for repeating a computation.

Bootstrap

The basic problem of probability is: Given a distribution F what is the distribution of a statistic T

Statisticians have a harder problem: Given data that come from an unknown distribution F, what is the distribution of a statistic T?

We do have an estimate of the true data distribution. It should look like the sample data distribution. (we write $\mathbb{F}_n$ for the sample data distribution and $\mathbb{F}$ for the true data distribution). We can work out the sampling distribution of $T_n(\mathbb{F}_n)$ by simulation, and hope that this is close to that of $T_n(F)$.

Simulating from $\mathbb{F}_n$ just involves taking a sample, with replacement, from the observed data. This is called the bootstrap. We write $\mathbb{F}_n^*$ for the data distribution of a resample.

Too good to be true?

There are obviously some limits to this

- It requires large enough samples for $\mathbb{F}_n$ to be close to F.

- It works better for some statistics (eg mean, variance) than others (eg median, quantiles)

- It doesn't work at all for some statistics (eg min, max, number of unique values)

The reason for the difference between statistics is that $\mathbb{F}_n$ needs to be "close to" F in an appropriate sense of "close" for the statistic. Precise discussions of this take a lot of math.

Uses of bootstrap

There are two main uses

- When you know the distribution of T_n is normal with mean θ, you just don't know how to compute the variance

- With a well-behaved statistic where the sample size is a little small for the Normal approximation.

It can also be used when you don't know what the asymptotic distribution is, but then you do need quite a bit of analysis to be sure that the bootstrap works for this statistic.

There are many ways of actually doing the bootstrap computations. In most cases they all work, but in difficult cases it matters which one you use. Read a good book (eg Davison & Hinkley Bootstrap methods and their application)

Example

Median bilirubin in PBC data

```
data(pbc, package="survival")

resample.a.median<-function(x){
xstar<- sample(x, size=length(x), replace=TRUE)
median(xstar)
}

lots.of.medians<-replicate(1000, resample.a.median(pbc$bili))

hist(lots.of.medians, col="peachpuff",prob=TRUE)
```

Example

Histogram of lots.of.medians
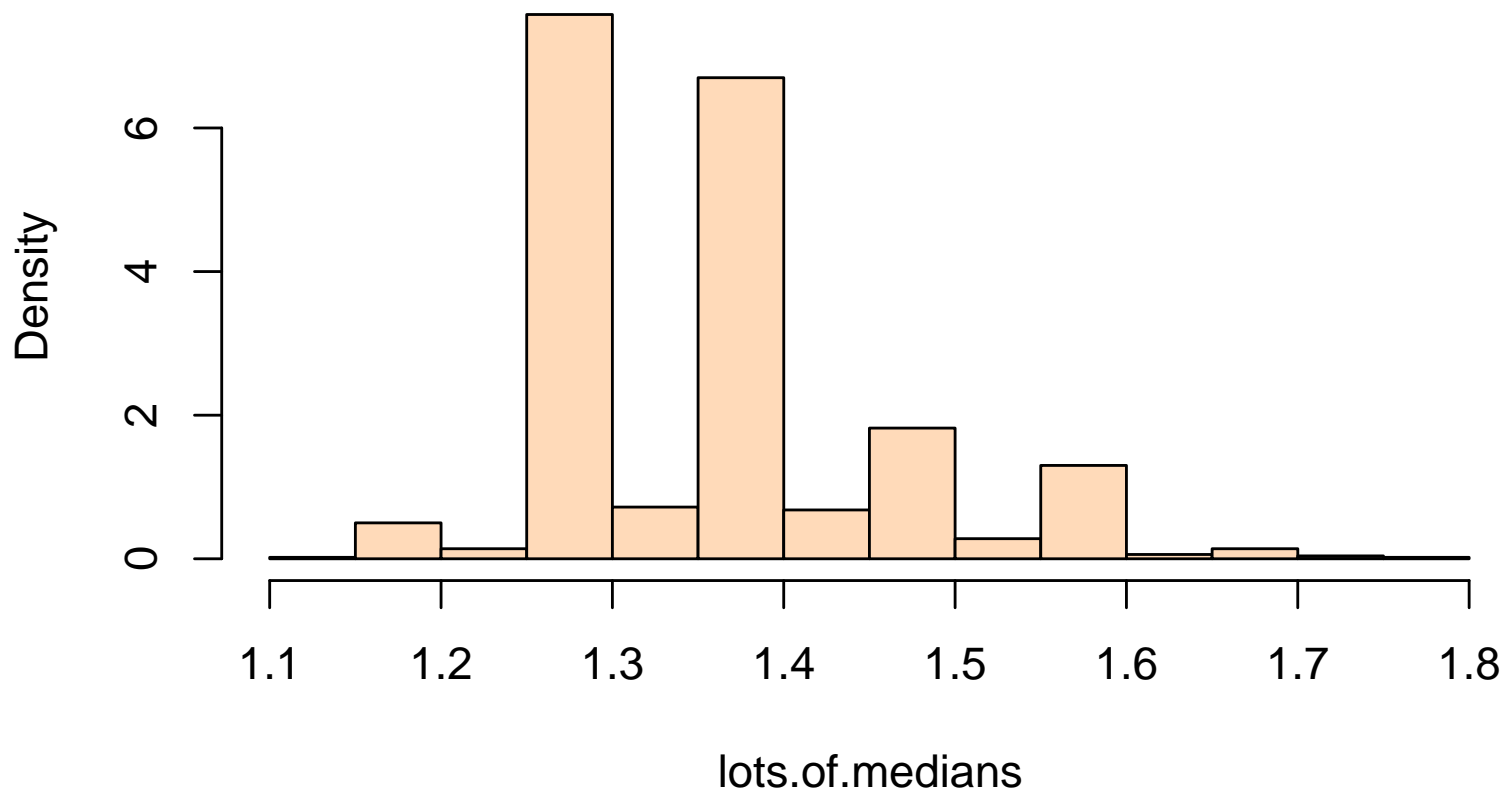
Notes

- sample() takes a sample from a given vector. This can be with or without replacement and with equal or unequal probabilities.

- replicate executes an expression many times and returns the results. It is tidier than a loop or apply.

- data() has a package argument for when you want the dataset but not the whole package.

- The histogram is fairly discrete, because the data are rounded to 2 decimal places: the true sampling distribution of the median is discrete. The true distribution of serum bilirubin isn't, but we have no data from that distribution.

How well does it work?

These graphs show the 5% and 95% points of the estimated sampling distribution. 90% of these should cover the true value. We need to use known distributions for this.

```
library(MASS)  ## Modern Applied Statistic in S (V&R)

resample.a.corr<-function(xy){
    index <- sample(nrow(xy),size=nrow(xy),replace=TRUE)
    cor(xy[index,1],xy[index,2])
}

lots.of.corr<-replicate(30, {
        dat<-mvrnorm(50,c(0,0), Sigma=matrix(c(1,.5,.5,1),2))
        replicate(400, resample.a.corr(dat))
        })
```

How well does it work?

```
qq<-apply(lots.of.corr,2,quantile, probs=c(0.05,0.95))
plot(1,1,xlim=c(1,30),ylim=range(c(0.5,qq)),ylab="Correlation",xlab="
abline(h=0.5,lty=2)
in.interval<-qq[1,]<0.5 & qq[2,]>0.5
segments(1:30,qq[1,],1:30,
    qq[2,],col=ifelse(in.interval,"grey50","purple"),lwd=2)
```

How well does it work?
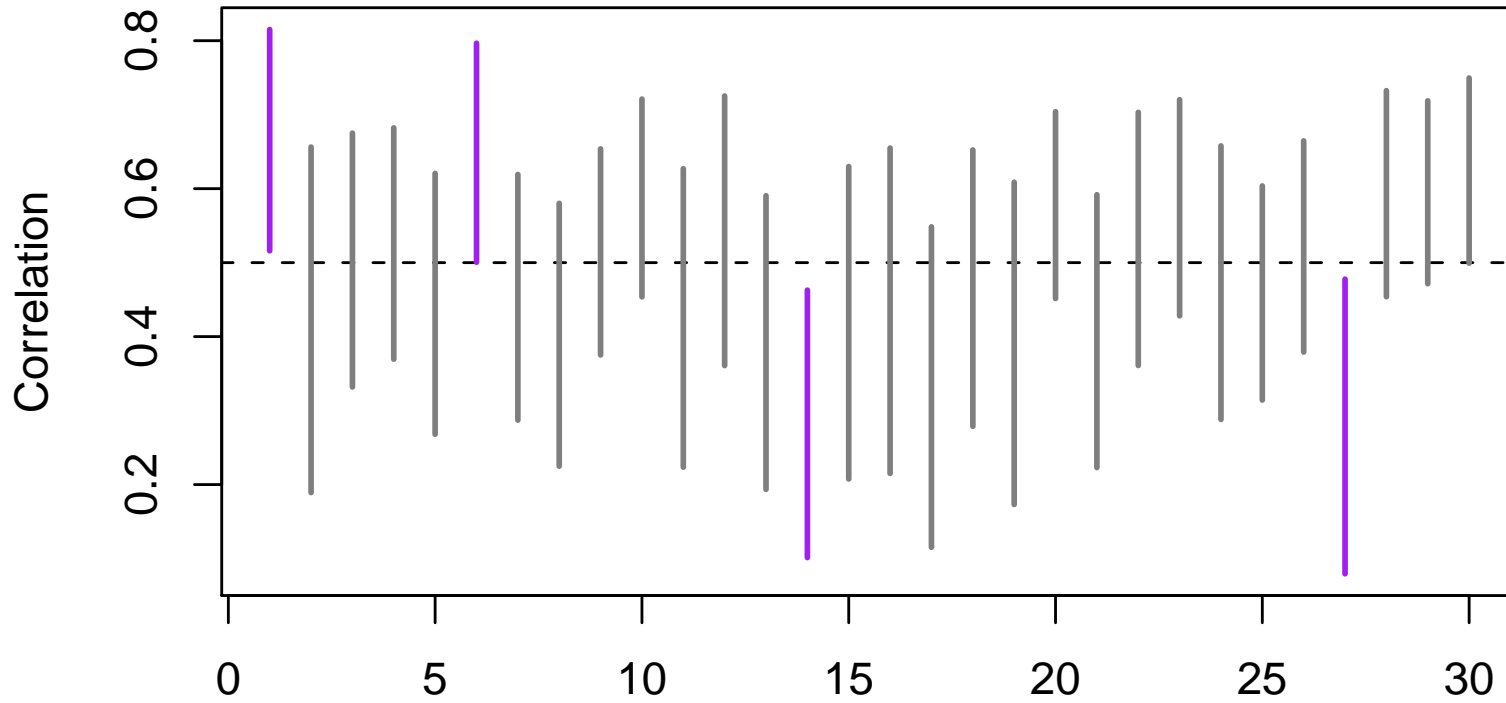
Notes

- We need to simulate the entire bootstrap process — draw a real sample, take 400 resamples from it — thirty times
- We resample rows, by sampling from numbers 1...`nrow(xy)` and then apply this as a subset index.
- 400 is a minimal reasonable number for boostraps and most simulations. The uncertainty in the 90% range is about 1.5%, in a 95% range would be about 3.5%. Usually between 1000 and 10,000 is a good number.
- The percentile bootstrap will always give estimates between -1 and 1 for correlation (unlike the t-bootstrap)
- The percentile bootstrap isn't improved by transforming the statistic, the t may be, eg, for correlation, bootstrapping $z = \tanh^{-1} r$

Lower quartile

```r
resample.a.q25<-function(x){
    x <- sample(x,length(x),replace=TRUE)
quantile(x, prob=0.25)
}


lots.of.q25<-replicate(30, {
        dat<-rnorm(20)
        replicate(400, resample.a.q25(dat))
      })


qq<-apply(lots.of.q25,2,quantile, probs=c(0.05,0.95))
plot(1,1,xlim=c(1,30),ylim=range(qq),ylab="Lower quartile",xlab="")
abline(h=qnorm(0.25),lty=2)
in.interval<-qq[1,]<qnorm(0.25) & qq[2,]>qnorm(0.25)
segments(1:30,qq[1,],1:30,
    qq[2,],col=ifelse(in.interval,"grey50","purple"),lwd=2)
```

Lower quartile
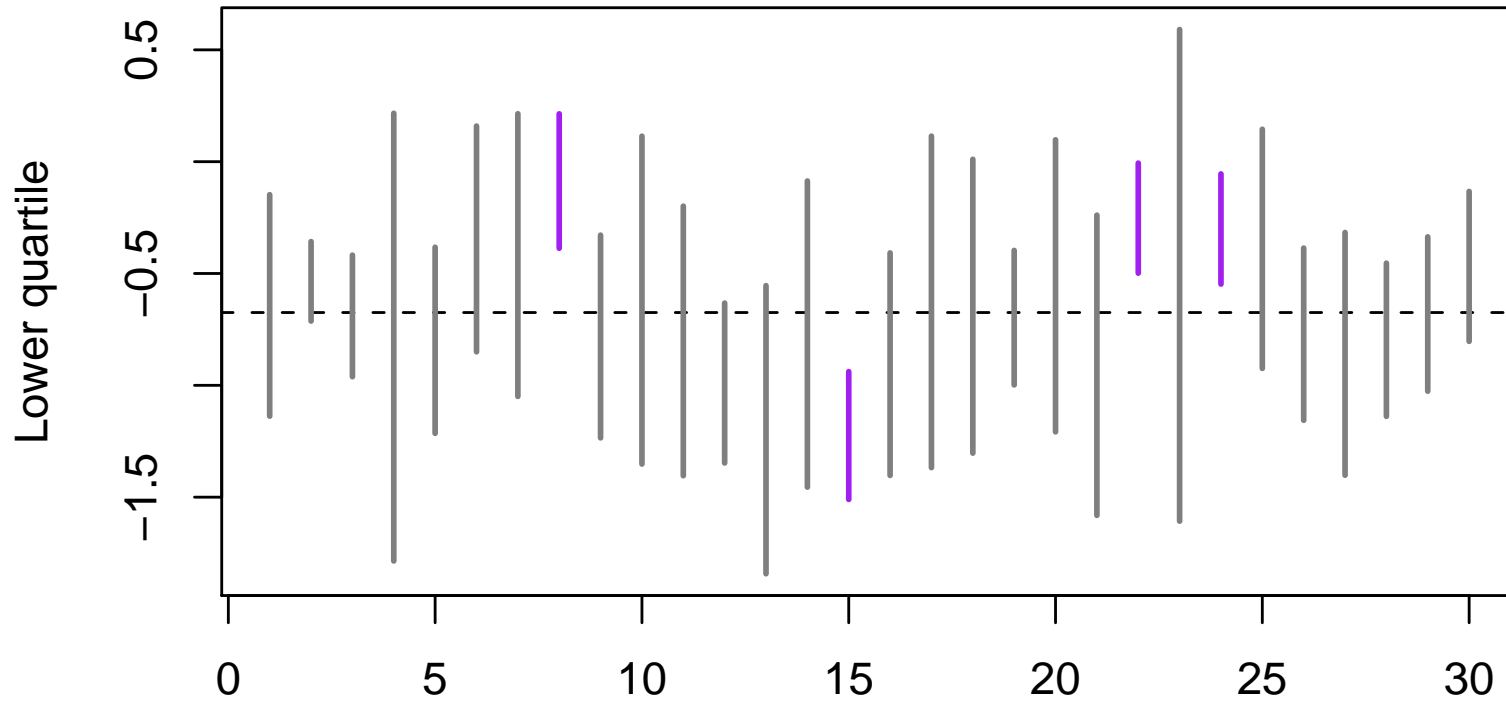
Minimum

```
resample.a.min<-function(x){
    x <- sample(x,length(x),replace=TRUE)
    min(x)
}


lots.of.min<-replicate(30, {
        dat<-rgamma(20,2,2)
        replicate(400, resample.a.min(dat))
        })


qq<-apply(lots.of.min,2,quantile, probs=c(0.05,0.95))
plot(1,1,xlim=c(1,30),ylim=range(c(-0.5,qq)),ylab="Minimum",xlab="")
abline(h=0,lty=2)
in.interval <- qq[1,] < 0 & qq[2,]> 0
segments(1:30,qq[1,],1:30,
    qq[2,],col=ifelse(in.interval,"grey50","purple"),lwd=2)
```

Minimum
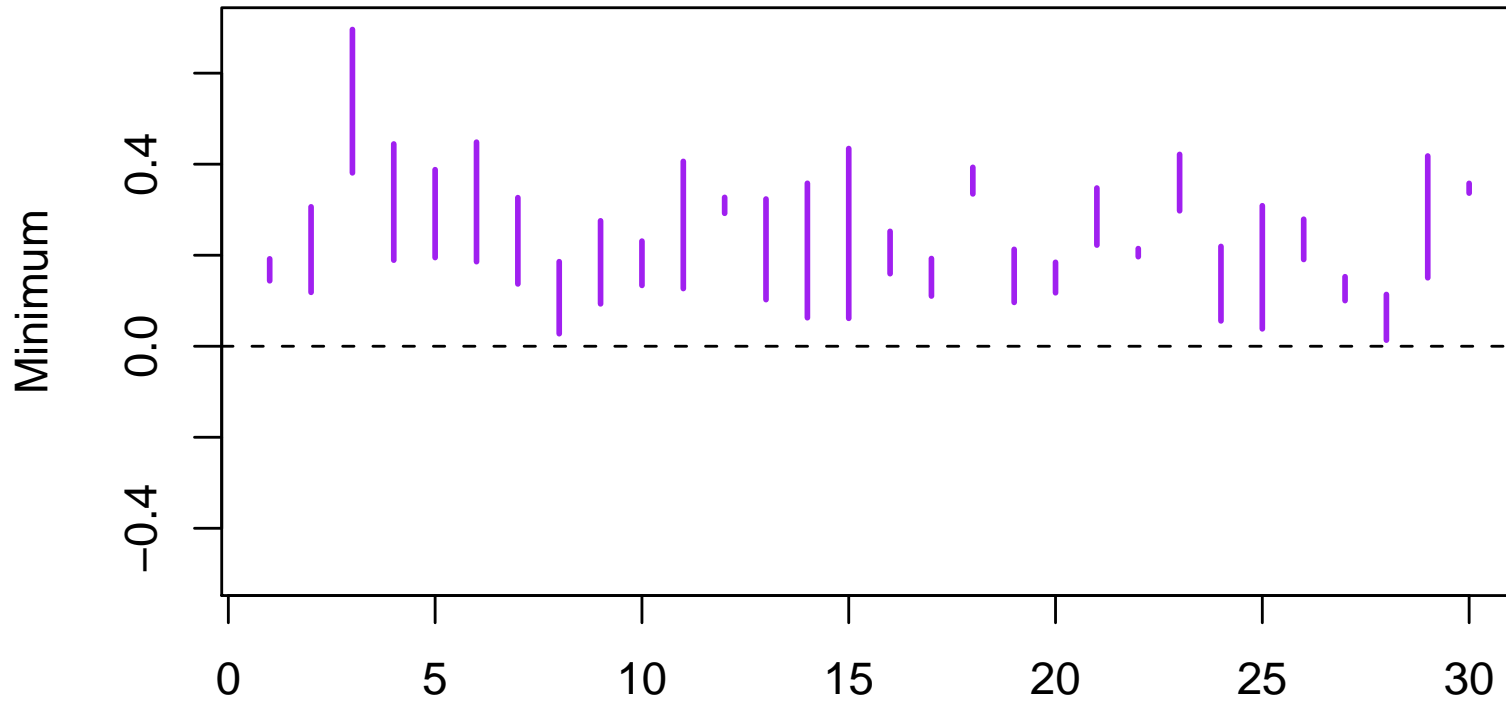
Bootstrap packages

You don't have to write your own bootstrap functions: there are two packages

- boot, associated with a book by Davison and Hinkley, and written by Angelo Canty

- bootstrap, associated with book by Efron and Tibshirani

The boot package comes with R and is more comprehensive. S-PLUS also has nice bootstrap functions written by Tim Hesterberg (at Insightful).

The boot package

As we did with resampling correlations, the boot function resamples row indices rather than data. You have to provide a function that takes a data set as its first argument and a set of row indices as the second argument.

We could redo the correlation example changing just a few lines

```
library(MASS)  ## Modern Applied Statistics in S (V&R)


resample.a.corr<-function(xy, index){
    cor(xy[index,1],xy[index,2])
}


lots.of.corr<-replicate(30, {
        dat<-mvrnorm(50,c(0,0), Sigma=matrix(c(1,.5,.5,1),2))
        boot(dat, resample.a.corr, R=400)$t        })
```

The boot package

```
qq<-apply(lots.of.corr,2,quantile, probs=c(0.05,0.95))
plot(1,1,xlim=c(1,30),ylim=range(c(0.5,qq)),ylab="Correlation",xlab="
abline(h=0.5,lty=2)
in.interval<-qq[1,]<0.5 & qq[2,]>0.5
segments(1:30,qq[1,],1:30,
    qq[2,],col=ifelse(in.interval,"grey50","purple"),lwd=2)
```

More usefully, the package provides a variety of bootstrap estimates. For one sample we get

```
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 400 bootstrap replicates

CALL :
boot.ci(boot.out = b)

Intervals :
Level       Normal              Basic
95%    ( 0.3008,  0.6914 )   ( 0.3069,  0.7266 )

Level      Percentile            BCa
95%    ( 0.2692,  0.6889 )   ( 0.2446,  0.6665 )
Calculations and Intervals on Original Scale
```

Some BCa intervals may be unstable

Warning message:

Bootstrap variances needed for studentized intervals in: boot.ci(b)

Debugging and optimization

Premature optimization is the root of all evil

<div align="right">Donald Knuth</div>

The first task when programming is to get the program correct, which is easier if it is written more simply and clearly.

Some clarity optimizations make the code faster, eg operating on whole vectors rather than elements. Some have no real impact, eg using `*apply` functions. Some make the code slower, like adding names to vectors.

When the code works correctly, the next step is to find out which parts, if any, are too slow, and then speed them up. This requires measurement, rather than guessing.

Timing

- R and S-PLUS both have `proc.time()`, which returns the current time. Save it before a task and subtract from the value after a task.

- S-PLUS has `sys.time(expression)`, R has `system.time()` to time the evaluation of `expression`

- In R, `Rprof(filename)` turns on the profiler, and `Rprof(NULL)` turns it off. The profiler writes a list of the current functions being run to `filename` many times per second. `summaryRprof(filename)` summarizes this to report how much time is spent in each function.

Remember that a 1000-fold speedup in a function that uses 10% of the time is less helpful than a 30% speedup in a function that uses 50% of the time.

Memory

- S-PLUS has `mem.tally.report`, which gives the maximum simultaneous allocation of memory, cache and evaluation memory, since the last call to `mem.tally.reset`

- R for windows has `memory.size`, which can report either current allocation or maximum ever allocated; on all platforms `gc()` will report maximum allocation since the last call to `gc(reset=TRUE)`.

- In R, `gcinfo(TRUE)` asks for a report on memory use every time the garbage collector is run.

Debugging

- `traceback()` shows where S was at the last error: what function it was in, where this was called from, and so on back to your top-level command.
- `options(error=dump.frames)` saves the entire state of your program when an error occurs. `debugger()` then lets you start the debugger to inspect any function that was being run. `options(error=recover)` starts the debugger as soon as an error occurs.
- `browser()` starts the debugger at this point in your code.
- `options(warn=2)` turns warnings into errors.
- In R `debug(fname)` starts the debugger when function `fname()` is called.

The debugger in both R and S-PLUS gives you an interactive command prompt inside your function. They differ in minor details.

Debugging: trace

`trace()` is a more powerful and flexible interface to the debugger. For example, we can set the debugger to start on statement 4 of `ourmedian` if the number of observations is zero.

```
> trace(ourmedian, tracer=quote(if(n==0) browser()), at=4)
> x<-rnorm(10)
> ourmedian(x)
Tracing ourmedian(x) step 4
[1] -0.7614219
> ourmedian(x[x>0])
Tracing ourmedian(x[x > 0]) step 4
[1] 0.5219192
> ourmedian(x[x>2])
Tracing ourmedian(x[x > 2]) step 4
Called from: ourmedian(x[x > 2])
Browse[1]>
```

Faster code

- Operations on whole vectors are fast.
- Matrix operations may be faster even than naive C code
- Functions that have few options and little error checking are faster: eg `sum(x)/length(x)` is faster than `mean(x)`
- Allocating memory all at once is faster than incremental allocation: `x<-numeric(10000); x[i]<-f(i)` rather than `x<-c(x, f(i))`
- Data frames are much slower than matrices (especially large ones).
- Running out of memory makes code much slower, especially under Windows.

If none of this works, coding a small part of the program in C may make it hundreds of times faster.

A very little on objects

Many functions in R return objects, which are collections of information that can be operated on by other functions.

In more extreme object-oriented languages objects have no user-serviceable parts. In R you can always get direct access to the internals of an object. You shouldn't use this access if there is another way to get the information: the developer may change the internal structure and break your code.

Use `str` and `names` to guess the internal structure.

Generics and methods

Many functions in R are generic. This means that the function itself (eg `plot`, `summary`, `mean`) doesn't do anything. The work is done by methods that know how to plot, summarize or average particular types of information. Earlier I said this was done by magic. Here is the magic.

If you call `summary` on a `data.frame`, R works out that the correct function to do the work is `summary.data.frame` and calls that instead. If there is no specialized method to summarize the information, R will call `summary.default`

You can find out all the types of data that R knows how to summarize with two functions

```
> methods("summary")
 [1]  summary.Date              summary.POSIXct        summary.POSIXlt
 [4]  summary.aov               summary.aovlist        summary.connection
 [7]  summary.data.frame        summary.default        summary.ecdf*
[10]  summary.factor            summary.glm            summary.infl
[13]  summary.lm                summary.loess*         summary.manova
[16]  summary.matrix            summary.mlm            summary.nls*
[19]  summary.packageStatus*    summary.ppr*           summary.prcomp*
[22]  summary.princomp*         summary.stepfun        summary.stl*
[25]  summary.table             summary.tukeysmooth*

    Non-visible functions are asterisked
> getMethods("summary")
NULL
```

There are two functions because S has two object systems, for historical reasons.

Methods

The class and method system makes it easy to add new types of information (eg survey designs) and have them work just like the built-in ones.

Some standard methods are

- `print`, `summary`: everything should have these
- `plot` or `image`: if you can work out an obvious way to plot the thing, one of these functions should do it.
- `coef`, `vcov`: Anything that has parameters and variance matrices for them should have these.
- `anova`, `logLik`, `AIC`: a model fitted by maximum likelihood should have these.
- `residuals`: anything that has residuals should have this.

New classes

Creating a new class is easy

```
class(x) <- "UW.class.of.2005"
```

(in S-PLUS, use `oldClass` rather than `class`.)

S will now automatically look for the `print.UW.class.of.2005` method, the `summary.UW.class.of.2005` method, and so on.

ROC curves (again)

A slightly more efficient version of the ROC function, and one that handles ties in the test variable:

```
drawROC<-function(T,D){
  DD <- table(-T,D)
  sens <- cumsum(DD[,2])/sum(DD[,2])
  mspec <- cumsum(DD[,1])/sum(DD[,1])
  plot(mspec, sens, type="l")
}
```

Note that we use the vectorized `cumsum` rather than the implied loop of `sapply`.

We want to make this return an ROC object that can be plotted and operated on in other ways

ROC curve object

```
ROC<-function(T,D){
  DD <- table(-T,D)
  sens <- cumsum(DD[,2])/sum(DD[,2])
  mspec <- cumsum(DD[,1])/sum(DD[,1])
  rval <- list(tpr=sens, fpr=mspec,
                cutpoints=rev(sort(unique(T))),
                call=sys.call())
  class(rval)<-"ROC"
  rval
}
```

Instead of plotting the curve we return the data needed for the plot, plus some things that might be useful later. `sys.call()` is a copy of the call.

Methods

We need a `print` method to stop the whole contents of the object being printed

```
print.ROC<-function(x,...){
    cat("ROC curve: ")
    print(x$call)
}
```

Methods

A plot method

```
plot.ROC <- function(x, xlab="1-Specificity",
                            ylab="Sensitivity", type="l",...){
      plot(x$fpr, x$tpr, xlab=xlab, ylab=ylab, type=type, ...)
}
```

We specify some graphical parameters in order to set defaults for them. Others are automatically included in ....

Methods

We want to be able to add lines to an existing plot

```
lines.ROC <- function(x, ...){
lines(x$fpr, x$tpr, ...)
}
```

and also be able to identify cutpoints

```
identify.ROC<-function(x, labels=NULL, ...,digits=1)
{
  if (is.null(labels))
    labels<-round(x$cutpoints,digits)
  identify(x$fpr, x$tpr, labels=labels,...)
}
```

Statistical Models in S

- An overview of statistical models in R
  - model formulas
  - model objects
- Examples of statistical modelling
  - linear and logistic regression
  - survival analysis
  - other

Statistical Modelling in S

The systematic part of a model is specified a model formula with basic structure

```
outcome~exposure*modifier+confounder
```

- The left-hand side is the outcome (response, independent) variable, the right-hand side describes the predictors.
- The * specifies an interaction and the corresponding main effects.
- Factors (eg race, subtype of disease) are coded by default with indicator variables for all except the first category.
- terms can be variables, simple expressions, or composite objects

Examples

- `depress~rural*agegp+partner+parity+income` Does the risk of postnatal depression vary between urban and rural areas, separately for each age group, adjusted for having a domestic partner, previous numbr of pregnancies, income?
- `asthma~pm25+temp+I(temp^2)+month` How does the number of hospital admissions for asthma vary with fine particulate air pollution, adjusted for temperature and month of the year?
- `log(pm25)~temp+stag+month+lag(temp,1)` Predict (log-transformed) fine particulate air pollution from temperature, air stagnation, month, and yesterday's temperature
- `Surv(ttoMI,MI)~LDL+age+sex+hibp+diabetes` How does LDL cholesterol predict (time to) myocardial infarction after adjusting for age, sex, hypertension, and diabetes?

Generalised linear models

Generalised linear models (linear regression, logistic regression, poisson regression) are handled by the `glm()` function. This requires

- A model formula
- A dataframe containing the variables [optional]
- A model family:
  binomial() logistic regression
  gaussian() linear regression
  poisson() Poisson regression
  and others less commonly used

```
glm(asthma~pm25+temp+I(temp^2)+month,
                data=pmdat,family=poisson())
```

Model objects

Typical statistics packages fit a model and output the results. In S a model object is created that stores all the information about the fitted model. Coefficients, diagnostics, and other model summaries are produced by methods for this object.

We saw some of these methods earlier.

Classes of model

R has a wide range of regression models

- `lm()` Linear regression
- `glm()` generalised linear models
- `coxph()` Cox model (in survival package)
- `clogit()` Conditional logistic regression (in survival package)
- `gee()` Generalised Estimating Equations (in gee and geepack packages)
- `lme()` Linear mixed models (in nlme package)
- `polr()` Proportional odds model (in MASS package)
- Two implementations of gam, in the mgcv and gam packages.

It does not have generalised linear mixed models (SAS PROC NLMIXED, BUGS), although there are functions for the PQL approximation.

Example of linear regression

`data(birthwt)` in the MASS package describes birthweight of 189 infants born in a Springfield,MA, hospital. Low birthweight is an important risk factor for many problems in infancy and childhood.

```
## A few of the exploratory graphs we might draw
pairs(birthwt[,c("age","lwt","ptl","bwt")])
boxplot(bwt~ftv,data=birthwt)
boxplot(bwt~race,data=birthwt)
## A linear model
model1<-lm(bwt~age+lwt+factor(race)+ptl+ftv+ht,data=birthwt)
anova(model1)
par(mfrow=c(2,2)) ##2x2 table of plots
plot(model1)
```

There is no suggestion that age is important, or preterm visits. Two points (labelled "188" and "226") seem influential.

Linear regression (cont)

Examining $\Delta\beta$s computed by `lm.influence()` shows that removing one of these points would have a fairly important effect on the coefficient for `ptl`. This is a large infant born to a small woman who smoked and had had three previous preterm deliveries.

```
> dbeta<-lm.influence(model1)$coef
> strange<-which(row.names(birthwt) %in% c("188","226"))
> round(dbeta[strange,],2)
          [,1]   [,2]  [,3]    [,4]    [,5]    [,6]  [,7]  [,8]
[1,] -135.81   9.30 -0.44    9.37 -10.34 -19.99 -7.09  0.54
[2,]    18.67 -0.11 -0.11 -10.63 -19.33  92.13 -4.13 -2.99
> birthwt[strange,]
     low age lwt race smoke ptl ht ui ftv   bwt
226    0  45 123    1     0   0  0  0   1 4990
188    0  25  95    1     1   3  0  1   0 3637
```

Example of logistic regression

We may be more interested in identifying risk factors for low birth weight (defined as < 2500g) than in predicting actual weights. It may be more productive to fit a logistic regression model to the binary variable bwt< 2500g.

```
model2<-glm(I(bwt<2500)~age+lwt+factor(race)+ptl+ftv+ht,
            data=birthwt,family=binomial())
anova(model2)
plot(model2) ## pretty useless plots
```

The same infant is still influential: normal birth weight despite being very high risk.

Survival Analysis

S-PLUS has had unusually good facilities for survival analysis for years. The survival package in R uses the same underlying code base, written by Terry Therneau at the Mayo Clinic.

- descriptive statistics
- comparing groups
- person-year calculations
- Simple Cox models
- Left truncation
- Time-dependent covariates
- Recurrent events
- Parametric survival models
- Interval censoring

Survival objects

The outcome in a survival analysis involves both an observed time and a censoring status. These are packaged together in a `Surv` object.

- `Surv(time,status)` for simple right-censoring
- `Surv(entry,exit,status)` for left truncation (late entry) and for time-dependent covariates.
- `Surv(entry,exit,status,origin=zero)` for when time=0 is different for different people
- `Surv(a,b,type="interval2")` Interval censoring

Descriptive statistics

`survfit()` computes Kaplan–Meier survival functions (with confidence intervals). These can be printed or plotted.

```
library(survival) ## survival analysis package
data(ovarian)     ## a Mayo ovarian cancer study
km<- survfit(Surv(futime,fustat)~rx,data=ovarian)
km                ## some summaries
summary(km)       ## more summaries
plot(km)          ## plot
plot(km[1])       ## plot one group
```

Cox regression

The majority of the "survival" package implements the proportional hazards model, which says that the relative risk of an event between subjects with different covariates is constant over time, so

$$\lambda(t; z) = \lambda_0(t)e^{\beta' z}.$$

The simplest case is right-censored data starting at time 0

```
data(ovarian)
model1<-coxph(Surv(futime,fustat)~rx+ecog+age,data=ovarian)
## summary of model
model1
plot(survfit(model1)) ##survival at mean covariates
## fitted survival at new covariates
plot(survfit(model1,data=data.frame(rx=c(0,1),
                ecog=c(3,3),age=c(50,50)))))
```

Data mining

Data mining involves, among other things

- Clustering for feature selection or market segmentation
- Adaptive regression models for prediction

It is typically done on large collections of data.

R has a number of clustering and adaptive regression procedures, but few are adapted for gigabyte datasets. R may be useful for detailed modelling of a random or systematic sample from a data warehouse, or for predictive modelling of smaller datasets. The database connection packages could be used to facilitate this.

Clustering

The cluster package contains clustering code by Rousseuw et al. for a variety of methods.

Most of these require $O(n^2)$ storage, except `clara` which requires only $O(n)$, and so are feasible only for small data sets (a few thousand observations).

Classification

Automated regression techniques include

- rpart: The CART algorithm for building and pruning classification trees. Includes surrogate splits and handles different types of response variable. rpart is part of the standard R distribution and is also available for S-PLUS as an improvement over the old tree functions. The RandomForest package implements random forest classifiers, an extension of trees.
- MARS: The polymars package implements the MARS method for binary and categorical response variables.
- k-NN: k-nearest neighbour classification is implemented in the class package in the VR bundle.
- neural nets: in the nnet package in the VR bundle.
- SVM. The e1071 package implements support vector machines.

Classification

The R classification functions are all designed for noisy problems, unlike some machine learning techniques.

They all keep the entire dataset in memory, so can be used on a few tens or hundreds of thousands of reasonably complicated observations. Subsampling or partitioning is needed for larger datasets.

R is not particularly well suited for data mining in the sense of completely automated searches in very large databases. It is quite good for predictive modelling by humans with a good understanding of the substantive area. As in many other areas R is optimised for knowledgable, interactive use, not for automated production.

Objects again

Now we know logistic regression we might want to construct an ROC curve as a summary of a logistic regression. We need the ROC function to be generic, so it can work for regression models or vectors

```
ROC <- function(T,...)  UseMethod("ROC")
```

The previous ROC function now becomes the default method

```
ROC<-function(T,D){
  DD <- table(-T,D)
  sens <- cumsum(DD[,2])/sum(DD[,2])
  mspec <- cumsum(DD[,1])/sum(DD[,1])
  rval <- list(tpr=sens, fpr=mspec,
            cutpoints=rev(sort(unique(T))),
            call=sys.call())
  class(rval)<-"ROC"
  rval
}
```

Objects again

and we need a new `ROC.glm` function that computes T and D from a fitted logistic regression model.

```
ROC.glm<-function(T,...){
  if (!(T$family$family %in%
            c("binomial", "quasibinomial")))
    stop("ROC curves for binomial glms only")

  test<-fitted(T)
  disease<-(test+resid(T, type="response"))
  disease<-disease*weights(T)

  TT<-rev(sort(unique(test)))
  DD<-table(-test,disease)

  sens<-cumsum(DD[,2])/sum(DD[,2])
  mspec<-cumsum(DD[,1])/sum(DD[,1])

  rval<-list(tpr=sens, fpr=mspec,
            cutpoints=TT,call=sys.call())
  class(rval)<-"ROC"
  rval
}
```

Connecting to other software

- Connections

- ODBC

- Tcl/Tk

- Rweb, Rcgi

Connections

Connections follow the Unix tradition of making everything a file, but go further by making network socket connections into files as well.

Connections can be used instead of files in eg

- `read.table` to read from a pipe or URL

- `cat()` to send text to a pipe or a network connection

- `sink()` to send R output to a string for further manipulation

Connections are described in the first issue of the R newsletter and in the Data Import/Export Manual.

Types of connections

- File connections

- Pipe connections.

- gzipped file connections

- URL connections

- Socket connections

- Text connections

Files

A file connection differs from a file in that it can be read a piece at a time

```
f<-file("temp","w")
data(trees)
write.table(trees,file=f)
close(f)
f<-file("temp","r")
done<-FALSE
i<-0
while(!done){
  i<-i+1
  tt<-readLines(f,1)
  if (length(tt)==0) done<-TRUE else cat(i,tt,"\n")
}
```

Used in `read.table` to determine file format by reading a few lines. Could be used to write eg linear model code for very large data sets.

Socket connections

An interface to the BSD-style sockets on Unix and Windows.
These allow computers to talk to each other

A simple HTTP transaction:

```
con<-socketConnection(port=80,host="www.washington.edu")

writeLines("GET / HTTP/1.0",con)
writeLines("Connection: Keep-Alive",con)
writeLines("Accept: text/html",con)
writeLines("",con)
a<-readLines(con)
while(length(a)==0){
  Sys.sleep(1)
  a<-readLines(con)
}
```

Socket connections

```
close(con)
```

We have already seen URL connections, which automate simple HTTP or FTP reads. The RCurl package has more sophisticated HTTP/FTP, including eg `https`.

Text Connections

Text connections allow R output to be captured with `sink()` and
stored in a string

```
zzz<-textConnection("ex.lm.out","w")
sink(zzz)
example(lm)
sink()
close(zzz)
ex.lm.out[1:10]
```

A prepackaged function to do this is `capture.output`

ODBC

There are interfaces of various stages of sophistication to ODBC, MySQL, PostgreSQL, Oracle and others. The aim is to produce a unified abstract interface, and there is a database SIG working on this. Again, the Data Import/Export Manual has details.

A brief example of the ODBC interface, which is the easiest under Windows. Data base at `http://faculty.washington.edu/tlumley/spring.mdb` (USGS database of historically documented springs and spring flow measurements in Texas).

ODBC

Use Control Panel to set up data source name

```
library(RODBC)
channel<-odbcConnect("spring_data")
sqlTables(channel)
odbcClose(channel)
```

GUIs

The other major use of linking R to other software is for interfacing with people rather than databases.

- Tcl/Tk for writing GUIs

- Rweb — using R as a web service

Tcl/Tk interface

Tcl/Tk is a popular graphical scripting language. It's not very powerful but is quite good at simple GUIs. It has been quite popular under Unix for adding GUIs to older software.

The syntax is very different from that of R, but the `tcltk` package provides a convenient set of wrappers.

- Tcl/Tk variables are accessed as `tclvar$varname`.
- Widget commands are generally translated to functions operating on the widget

```
## $ wish
## % button .a
## % pack .a
## % .a configure -text "hello"
root<-tktoplevel()
a<-tkbutton(root,text="hello")
tkpack(a)
```

Interacting with Tcl/Tk

- R functions can be passed as Tcl/Tk callbacks, but data have to be passed to R in Tcl variables
- R functions can call Tcl/Tk functions but can only pass limited types of variables (strings, numbers).

From `tkdensity`

```
regen <- function(...) {
  if (tclvar$dist==1)
  y<<-rnorm(as.numeric(tclvar$size))
  else
  y<<-rexp(as.numeric(tclvar$size))
  xlim <<- range(y) + c(-2,2)
  replot() }

base <- tktoplevel()
tkwm.title(base, "Density")

spec.frm <- tkframe(base,borderwidth=2)
left.frm <- tkframe(spec.frm)
right.frm <- tkframe(spec.frm)

## Two left frames:
frame1 <- tkframe(left.frm, relief="groove", borderwidth=2)
tkpack(tklabel(frame1, text="Distribution"))
tkpack(tkradiobutton(frame1, command=regen,
  text="Normal", value=1,
  variable="dist"), anchor="w")
```

Example: Sudoku

Under Windows, uses `getGraphicsEvent` to pick up keyboard and mouse events

Under Unix, uses `tkrplot` package to embed an R graphics window in a Tk widget and uses Tk/Tcl keyboard and mouse event notification.

Also uses a URL connection to download Sudoku puzzles from `www.sudoku.org.uk`

Rpad

There are a number of packages available to run R as a web service As R can produce text, GIF, PNG and PDF output it can easily send output to a browser.

`Rweb`, developed at the University of Montana, can be used on their web server. Both `Rweb` and `Rcgi` run using CGI scripts that call R. Other approaches include `Rzope`, which uses the Zope webservice platform; `Rserve`, which uses sockets; `Rpad`, which has its own webserver built in; and `RApache`, which embeds R in the Apache web server.

At the moment R cannot be made secure — you must assume that everything the R process can touch is untrusted.

Example:`http://www.rpad.org/Rpad`

R Packages

The most important single innovation in R is the package system, which provides a cross-platform system for distributing and testing code and data.

The Comprehensive R Archive Network (`http://cran.r-project.org`) distributes public packages, but packages are also useful for internal distribution.

A package consists of a directory with a `DESCRIPTION` file and subdirectories with code, data, documentation, etc. The Writing R Extensions manual documents the package system, and `package.skeleton()` simplifies package creation.

Packaging commands

- `R CMD INSTALL packagename` installs a package.
- `R CMD check packagename` runs the QA tools on the package.
- `R CMD build packagename` creates a package file.

The DESCRIPTION file

From the survey package

```
Package: survey
Title: analysis of complex survey samples
Description: Summary statistics, generalised linear models, and general maximum
pseudolikelihood estimation for stratified, cluster-sampled, unequally weighted
survey samples. Variances by Taylor series linearisation or replicate weights. P
ost-stratification and raking. Graphics.
Version: 2.9
Author: Thomas Lumley
Maintainer: Thomas Lumley <tlumley@u.washington.edu>
License: LGPL
Depends:
Requires: R (>=2.0.1)
Suggests: survival
Packaged: Tue Mar  8 16:30:43 2005; thomas
```

`Depends:` lists R packages needed to build this one. `Requires:` is used mostly for requiring a version of R. `Suggests:` lists packages needed eg to run examples. `Packaged:` is added automatically by the system.

The INDEX file

This also goes in the package directory and contains information about every sufficiently interesting function in the package.

If an `INDEX` file is not present it will be created from the titles of all the help pages. The `INDEX` file is displayed by

```
library(help=packagename)
```

Interpreted code

R code goes in the `R` subdirectory, in files with extension `.s`, `.S`, `.r`, `.R` or `.q`.

The filenames are sorted in ASCII order and then concatenated (one of the few places that R doesn't observe locale sorting conventions).

`R CMD check` detects a number of common errors such as using `T` instead of `TRUE`.

Documentation

Documentation in `.Rd` format (which looks rather like LaTeX) is the the `man` subdirectory.

`R CMD Sd2Rd` will convert S-PLUS documentation (either the old troff format or the new SGML) and `R CMD Rdconv` will do the reverse.

The QA tools check that every object is documented and that the arguments a function is documented to have are the same as the ones it actually has, and that all the examples run.

Data

Data go in the `data` subdirectory and are read with the `data()` function.

- ASCII tables with `.tab`, `.txt` or `.TXT`, read using `read.table( ,header=TRUE)`
- R source code with `.R` or `.r` extensions, read using `source`
- R binary format with `.Rdata` or `.rda` extensions, read using `load`.

The directory has an index file (`00INDEX`) to provide descriptions of the data files.

Compiled code

C or Fortran code (or other code together with a `Makefile`) goes in the `src` subdirectory.

It is compiled and linked to a DLL, which can be loaded with the `library.dynam` function.

Obviously this requires suitable compilers. The nice people at CRAN compile Windows and Macintosh versions of packages for you, but only if it can be done without actual human intervention.

inst/ and Vignettes

The contents of the `inst` subdirectory are copied on installation. A `CITATION` file can be supplied in `inst` to give information on how to cite the package. These are read by the `citation()` function. Please cite R and packages that you use.

Vignettes, Sweave documents that describe how to carry out particular tasks, go in the `inst/doc/` subdirectory. The Bioconductor project in bioinformatics is requiring vignettes for its packages.

You can put anything else in `inst/` as well.

Tests

Additional validation tests go in the `tests` subdirectory. Any `.R` file will be run, with output going to a corresponding `.Rout` file. Errors will cause `R CMD check` to fail.

If there is a `.Rout.save` file it will be compared to the `.Rout` file, with differences listed to the screen.

Distributing packages

If you have a package that does something useful and is well-tested and documented, you might want other people to use it too. Contributed packages have been very important to the success of R (and before that of S-PLUS).

Packages can be submitted to CRAN by ftp.

- The CRAN maintainers will make sure that the package passes `CMD check` (and will keep improving `CMD check` to find more things for you to fix in future versions).
- Other users will complain if it doesn't work on more esoteric systems and no-one will tell you how helpful it has been.
- But it will be appreciated. Really.

ROC curves (yet again)

We can now put the ROC curve functions into a simple package.

- Use `package.skeleton` to start

- Edit `DESCRIPTION`

- Edit help files