

Christoph Scherber

# An introduction to statistical data analysis using R

Basic operations, graphics and modelling using R

Christoph Scherber  
09.10.2007

# 1 What is R?

"The history of R begins at AT&T Bell laboratories, when they decided to develop a programming language designed to do statistical analysis - the result was the **S language**. S proved very popular with statisticians and led to a proposal to market it as a commercial product - after the addition of an extensive graphical user interface (GUI) to make it more user-friendly - as **S-Plus**. Ross Ihaka and Robert Gentleman from the University of Auckland in New Zealand decided to write a pared down version of S for use in teaching. Following in the line of a series of minimalist computer program names (like C), the two R's named their teaching version of S "**R**". The R source code was released in 1995 under a **General Public License** (GPL). The development of R is now guided by an international development team and R is now easily downloaded from the internet from a network of CRAN (Comprehensive R Archive Network) mirror sites." (citation: Andy Hector, Zurich)

If you haven't downloaded the R software package yet, you can easily get it from

<http://www.r-project.org>

Where you can also find lots of other interesting background information.

The screenshot shows the R Project for Statistical Computing website. The browser window title is "The R Project for Statistical Computing - Mozilla Firefox". The address bar shows "http://www.r-project.org/index.html". The website content includes:

- The R Project for Statistical Computing** logo and title.
- PCA 5 vars**: A plot showing the first three principal components of five variables: Fertility, Catholic, Examination, Education, and Agriculture. The plot shows a strong negative correlation between Fertility and Examination/Education. A bar chart below shows the variance explained by the first three components: (1-3) 60%.
- Clustering 4 groups**: A dendrogram showing the hierarchical clustering of data points into four groups. A bar chart below shows the size of each group: 28, 16, 1, and 2.
- Factor 1 [41%]** and **Factor 3 [19%]**: Two histograms showing the distribution of the first and third principal components, with normal distribution curves overlaid.
- Important News**: A section containing the news item: "R version 2.0.1 has been released on 2004-11-15."
- Related Projects**: A list of links to other projects: Bioconductor, Omega, and eRarchical models.

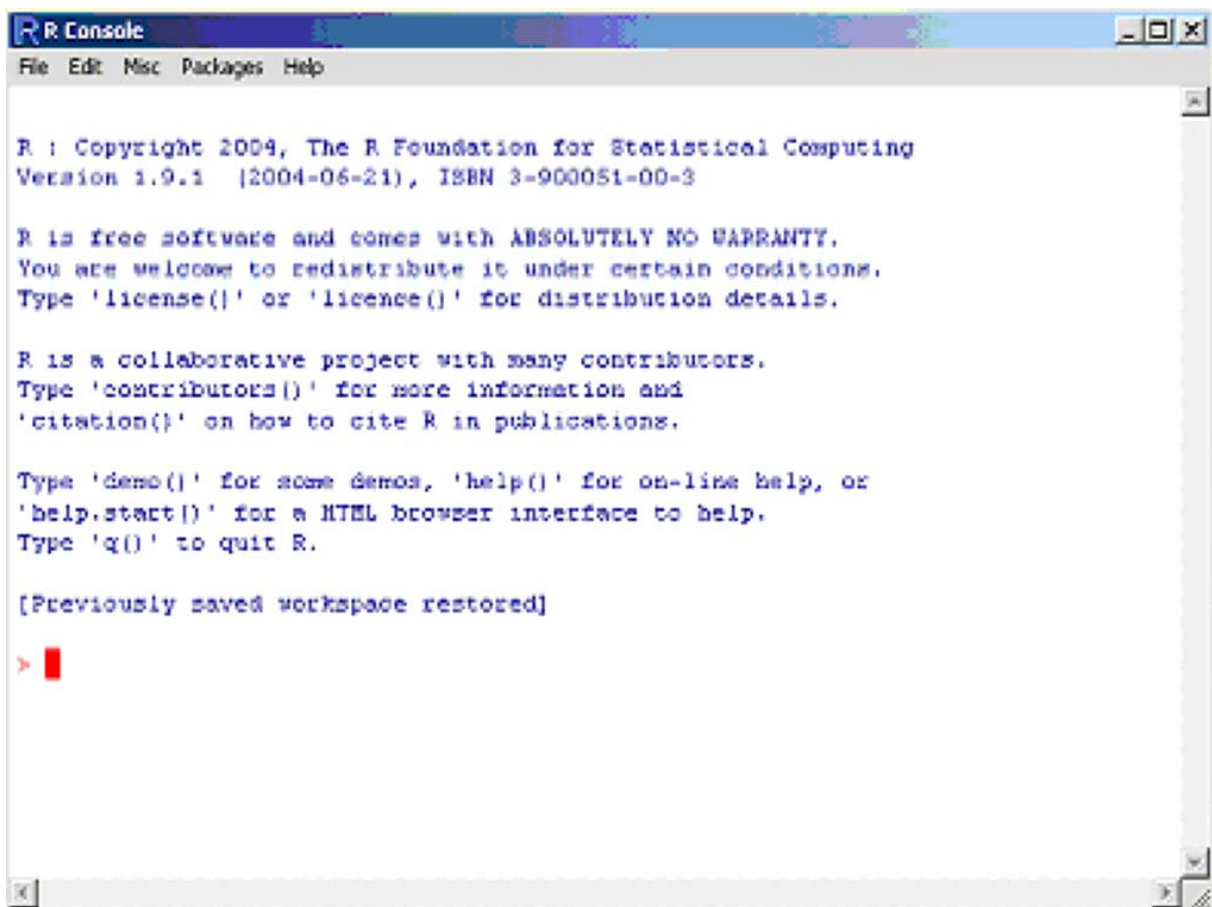
The footer of the website states: "This server is hosted by the Center for Computational Intelligence of the TU Wien." The browser status bar shows "http://www.r-project.org/misc/acplust.R" and "Adblock".

The corresponding download page is at

<http://cran.uk.r-project.org/>

Simply click on "Precompiled Binary Distributions" and then on the version supporting your Operating System (e.g. Microsoft Windows, Linux or Mac OS). Then download the "base" package.

Unpack the file and install R to a desired path (e.g. "C:"). Load the R Console, and you're ready to experience R!

A screenshot of the R Console window. The title bar reads "R Console". The menu bar contains "File", "Edit", "Misc", "Packages", and "Help". The main text area displays the following information: "R : Copyright 2004, The R Foundation for Statistical Computing", "Version 1.9.1 (2004-06-21), ISBN 3-900051-00-3", "R is free software and comes with ABSOLUTELY NO WARRANTY. You are welcome to redistribute it under certain conditions. Type 'license()' or 'licence()' for distribution details.", "R is a collaborative project with many contributors. Type 'contributors()' for more information and 'citation()' on how to cite R in publications.", "Type 'demo()' for some demos, 'help()' for on-line help, or 'help.start()' for a HTML browser interface to help. Type 'q()' to quit R.", and "[Previously saved workspace restored]". A red prompt character ">" is visible at the bottom left of the text area.

```
R : Copyright 2004, The R Foundation for Statistical Computing
Version 1.9.1 (2004-06-21), ISBN 3-900051-00-3

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for a HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

> █
```

## 2 Contents

|       |                                                       |    |
|-------|-------------------------------------------------------|----|
| 1     | What is R? .....                                      | 2  |
| 2     | Contents.....                                         | 4  |
| 3     | Why use R? .....                                      | 5  |
| 4     | Need Help?.....                                       | 6  |
| 5     | Contributed Packages .....                            | 7  |
| 6     | The commands and the scripts window .....             | 7  |
| 6.1   | The commands window .....                             | 7  |
| 6.2   | The Script Window .....                               | 8  |
| 6.3   | Saving your work .....                                | 9  |
| 6.4   | The working directory .....                           | 10 |
| 7     | Importing and exporting data .....                    | 10 |
| 7.1   | Importing from Microsoft Excel .....                  | 10 |
| 7.2   | Importing from a text file .....                      | 11 |
| 7.3   | Exporting data to a text file .....                   | 12 |
| 8     | Typing in Data.....                                   | 13 |
| 9     | An introductory session.....                          | 13 |
| 10    | Working with large datasets .....                     | 17 |
| 11    | Sorting and summarizing data.....                     | 19 |
| 12    | Creating high-level plots in R .....                  | 21 |
| 12.1  | Exploratory Data Analysis .....                       | 21 |
| 12.2  | Plotting a Histogram.....                             | 21 |
| 12.3  | A simple Scatterplot .....                            | 22 |
| 12.4  | Plotting multivariate data .....                      | 23 |
| 12.5  | Line plot .....                                       | 25 |
| 12.6  | Boxplot.....                                          | 26 |
| 12.7  | Three-dimensional plots.....                          | 26 |
| 12.8  | Piecharts .....                                       | 30 |
| 12.9  | Trellis Scatterplots.....                             | 31 |
| 13    | Creating pdf's and postscript files.....              | 34 |
| 14    | Creating publication-quality graphs .....             | 37 |
| 15    | Statistical Modelling .....                           | 38 |
| 15.1  | Simple tests: The t test .....                        | 38 |
| 15.2  | Model Formulae in R .....                             | 39 |
| 15.3  | Regression .....                                      | 40 |
| 15.4  | Non-Linear Regression .....                           | 41 |
| 15.5  | Analysis of variance .....                            | 45 |
| 15.6  | Time Series.....                                      | 49 |
| 15.7  | A Generalized Linear Model (from Bill Venables) ..... | 50 |
| 16    | Generating Experimental Designs.....                  | 52 |
| INDEX | .....                                                 | 53 |
| 17    | Author's Address.....                                 | 56 |

### 3 Why use R?

If you like point-and-click adventures, you're going to be fully satisfied with "big shots", such as SPSS or Statistica. But there may come a time where you will notice that you're going to have to start programming - even in SPSS.

Have you ever tried a split-plot analysis of variance in SPSS? Well, you should, because eventually you'll find out that it is not possible without switching to the syntax window.

Or, have you tried installing the most recent releases of S-Plus or SPSS? They are all server-based, watching with eagle's eyes on how many different computers you try to install it.

R is different. Not only that it's free, it also offers state-of-the-art statistical data analysis, high-level graphics, and the greatest flexibility you could ever dream of. What's more, it will not only work with Windows-based systems, but also on Apple Mac OS and Linux.

Of course, you are going to miss the menus. However, the problem with menus is that they are inefficient for repetitive tasks. The good thing about self-written syntax is: Once you've finished one project and moved onto a new one, you will soon find that you will already have a text file of an old analysis that can be edited and adapted for the new dataset.

Now let's move on to see what this all means.

#### 4 Need Help?

For help at any time, simply type `?` or `help.search("subject")` where "subject" is the area you're interested in; for example, if you want to know how to do an analysis of variance, simply type

```
help.search("analysis of variance")
```

Or, if you already know the command you're interested in, just use the "?" command. Let's say you want to know how to use "help.search":

```
?help.search
```

This gives you almost everything you need to know about how to use R help.

## 5 Contributed Packages

If you install R for the first time, most things you might want to do will be possible - such as basic statistical operations, graphics and so on.

Yet, you may come to a point where you wish to do **more than just basic things**. You may want to do something like

- Import Excel files
- Use tree models
- Use mixed effects models
- Do conditioned trellis graphics
- Analyze spatial data
- Create interactive plots,

and so on. Whenever you want to do things like those mentioned above, you are lucky - because one of the main strengths of R is that there are loads of so-called "**contributed packages**" allowing you to install "extra components" for special purposes. Nothing is easier than installing those contributed packages - and in case you want to learn more, just try the following:

If you want to know what packages are available in principle, use

```
library()
```

For more detailed information on an installed package, just type (for example)

```
library(help="nlme")
```

(gives an overview of the functions contained in the "nlme" package, with which you can do non-linear and linear mixed-effects models)

```
packageDescription("nlme")
```

(shows the general package description and authorship)

## 6 The commands and the scripts window

When you start R for the first time, you will find that there are two basic possibilities to tell the program what to do:

### 6.1 The commands window

This is where you usually type in what you want R to do, using your keyboard.

For those of you who haven't tried out a programming language (such as Pascal or Basic) yet: The command line basically works **like a text editor**, but with the big difference that

what you type is "**translated**" to the computer, and eventually it will result in the computer "doing something" for you.

For example, if you type

```
Hello
```

The computer's "answer" will be:

```
Error: Object "Hello" not found
```

So you should try to type in something more meaningful, such as

```
print("Hello")
```

Now the computer knows what you want it to do. You tell it to print the text called "Hello" to the screen:

```
[1] "Hello"
```

Using the commands window is made even more convenient by a special feature: When you use your „up“ and „down“ **arrow keys**, you can access whatever you've typed before. Just try it out!

## 6.2 The Script Window

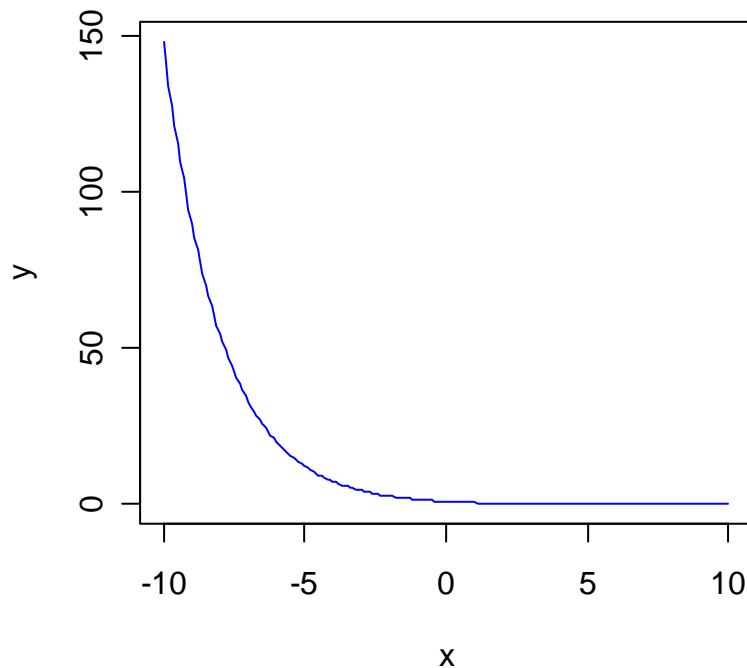
Go to the "**File**" Toolbar (or press "Ctrl"+"N") to create a new **script window**. As you will see, a script window opens. Here, you can basically do the same as described for the commands window, but with the additional feature of having more flexibility.

Well, you won't find out unless you try it out, so here is some R code to start with:

```
x<-seq(-10,10,0.1)  
y<-exp(-0.5*x)  
plot(x,y,type="l",col="blue")
```

After typing the code to the script window,  
- mark the text you want to run using your mouse or keyboard  
- press „Ctrl"+"R" to run the script  
- and just see what happens





Don't worry too much about the code behind it (you'll learn this later). What we have done is plotting a negative exponential function using blue line colours.

Note, too, that the whole code you've run **also** appears in the „**Commands Window**“. Thus, you can now **switch between both windows**, depending on what you prefer to do.

### 6.3 Saving your work

Whenever you want to save some text you've typed, some graph you've produced, and so on, just use your mouse cursor and open the "**File**" scroll-down menu. You should be able to save your file(s) to text or graphics format depending on what kind of file you have produced.

The **Graphical User Interface** (GUI) also offers you to save any graph you produced by right-clicking with your mouse.

But be prepared that even much more "magical" things are possible with R. For example, you can directly open Windows explorer from within R using the following command:

```
system("explorer C:\\")
```

Or the MS-DOS commands window using

```
system("cmd")
```

If you wish to save data to a text file (for import into Excel), just use the `write.table()` command.

**Example:**

Create a dataframe consisting of two variables A and B:

```
A<-seq(1,10,1) #creates two vectors A and B
B<-rep("B",10)
df<-data.frame(A,B) #creates a data frame
```

Now export these data to a text file:

```
write.table(df,"C:\\File1.txt") #saves the frame to a file
```

## 6.4 The working directory

Another important thing that you may wish to change is the **working directory**. This is the default directory into which R files, history files etc. are saved.

First, it is useful to assess the current state of the working directory. This is done using

```
getwd()
```

For example, on my personal computer, this command gives

```
[1] "E:/Programs/R/R-2.5.1"
```

You can set the working directory by typing

```
setwd("C:\\data\\examples")
```

## 7 Importing and exporting data

### 7.1 Importing from Microsoft Excel

There are several possibilities for importing Excel files. All methods require the separate installation of specific data import-export libraries.

The at current most convenient way is to use the new `xlsReadWrite` package. The syntax to be used is:

```
read.xls( file,
          colNames = TRUE,
          sheet = 1,
```

```
type = "data.frame",
from = 1,
rowNames = NA, colClasses = NA, checkNames = TRUE,
dateTimeAs = "numeric",
stringsAsFactors = default.stringsAsFactors() )
```

For example, a straightforward way of importing an Excel file called "sample.xls" might be:

```
sample.data<-read.xls("C:\\sample.xls",sheet=1,colNames=T)
```

A more old-fashioned way of importing Excel files is to establish a connection using a package called "RODBC".

Loading the package is most convenient using the command "library()"

```
library(RODBC)
```

Now the corresponding excel file is opened as follows:

```
z<-odbcConnectExcel("C:\\sample.xls")
frame<-sqlFetch(z,"Sheet1") #or any other name of the sheet
close(z)
```

The attach() command connects the data frame to the column titles given in the first row (if header=T)

```
attach(frame)
```

|                                                                                                                          |
|--------------------------------------------------------------------------------------------------------------------------|
| Note that all import-export filters at current to <b>not</b> support <b>XLSX</b> files of <b>Microsoft Office 2007</b> . |
|--------------------------------------------------------------------------------------------------------------------------|

|                                                                                                                                                                                                                                                                                                                                                           |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Note also that it can be <b>dangerous</b> to trust too much in simple <b>Excel</b> import-export filters. The <b>proper way</b> to deal with data in R is to work with <b>tab-delimited text files</b> . They use up far less space, and they will be readable even in 150 years from now (while Excel versions will always change every couple of years) |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## 7.2 Importing from a text file

Rather than using Excel files, you should try to **always import your datasets from text files**.

Text files have several **advantages** over Excel files, including:

- small size
- every computer program can read them
- they can be edited using the Windows notepad, or any other text editor
- they force the user to have a very clear idea of the structure of a dataframe

The standard textbook method is:

```
frame<-read.table("C:\\sample.txt",header=T)
attach(frame)
```

The attach() command connects the data frame to the column titles given in the first row (if header=T)

You will be likely to fail if you only use this basic command. If your dataset contains, say, **empty cells** or **missing values**, you should always use something like the command written below.

```
frame<-read.table("C:\\sample.txt",header=T,
na.strings="NA",sep=" ",dec=".",fill=T)
attach(frame)
```

**sep** gives the column separator; " " is for blank space  
na.strings indicates how missing values are labelled in the file  
dec indicates the sign used for decimal point ( "." or ",")  
fill=T columns with unequal length are filled with blank cells

### 7.3 Exporting data to a text file

This is as easy as using read.table.

```
write.table(x, file = "C:\\sample.txt",sep = " ", na = "NA",
dec = ".")
```

see the specifications for read.table.

Remember that you should always adjust the commands to your own needs! Simply copying what I have written here will of course not always work - depending on the type of dataset you are dealing with.

For example, if your Excel version works with **decimal commas** instead of decimal dots, you should use sep="\t" and dec=","

## 8 Typing in Data

The "gets" assignment "`<-`" is probably the most important thing you will need to know when using R.

```
"x <- 1" reads "1 is assigned to x"
          or, even easier to say, "x gets 1"
```

Note that in new versions of R, you can also write

```
x = 1
```

but be careful not to confuse such statements with logical operators. For example,

```
x[x=1] gives those values of x that have the value 1.
```

Use the **scan()** command to read in data from the keyboard; terminate input by pressing "return" two times

```
var<-scan()
```

The **c()** command produces a vector (**concatenate**) consisting of the values given in brackets and separated by commas.

```
var<-c(1,2,3,....)
```

The **rep()** command repeats a value (or string) n times

```
var<-rep(value,n)
```

The **seq()** command produces a sequence from the start value to the end value with steps of size **step**.

```
var<-seq(start,end,step)
```

## 9 An introductory session

To get a first impression on what you can do with R, let's create an **artificial dataset** consisting of just two variables, x and y. While the x values are fixed, we want the y values to be dependent on x, but with some "random component" of variation ("scatter").

Make x = (1,2,3,....,20):

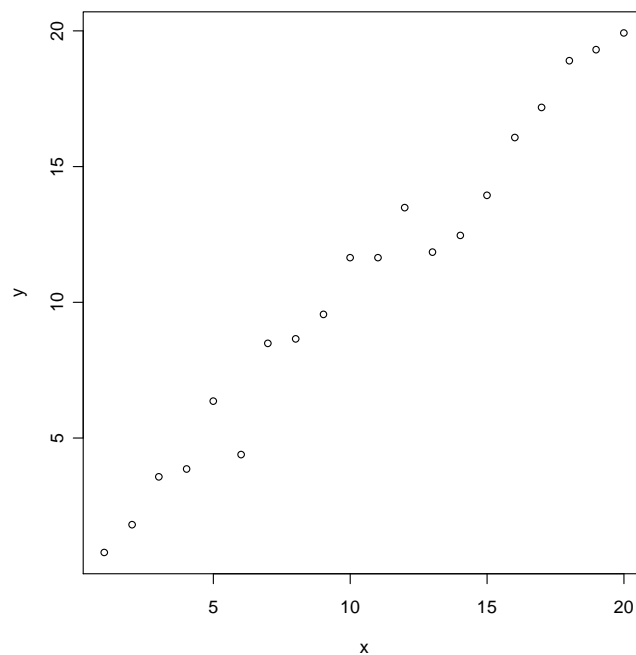
```
x <- 1:20
```

Make  $y$  a linear function of  $x$  plus normally distributed deviations:

```
y <- x+rnorm(x)
```

Now create a plot of  $y$  against  $x$ :

```
plot(x,y)
```



**This is a good time to get a feeling for R's Graphical User Interface (GUI).**

Be sure to go to the „**History**“ Scroll-Down menu and check the „**Recording**“ item. This makes handling and saving graph sheets much more convenient!

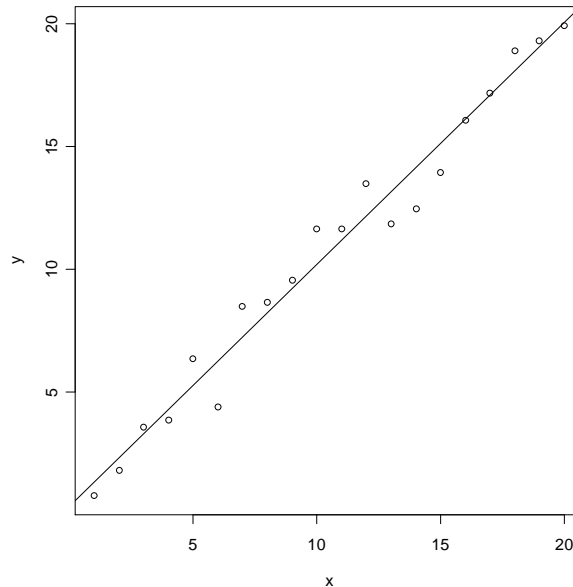
You should try out the „Page Up“ and „Page Down“ keys and see how you can change between different graph sheets you have produced in your latest R session.

Alternatively, you can set the options from the command line by typing

```
options(graphics.record=TRUE)
```

Now, returning to our graph: All data points seem to lie roughly along a straight line, so it is sensible to try to fit a linear regression through the data:

```
modell1<-lm(y~x)
abline(modell1)
```



summary(modell1) gives the following output:

**Call:**

```
lm(formula = y ~ x)
```

**Residuals:**

| Min     | 1Q      | Median | 3Q     | Max    |
|---------|---------|--------|--------|--------|
| -1.8794 | -0.4967 | 0.1542 | 0.5370 | 1.4382 |

**Coefficients:**

|             | Estimate | Std. Error | t value | Pr(> t )     |
|-------------|----------|------------|---------|--------------|
| (Intercept) | 0.33633  | 0.46571    | 0.722   | 0.479        |
| x           | 0.98716  | 0.03888    | 25.392  | 1.51e-15 *** |

---

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

**Residual standard error: 1.003 on 18 degrees of freedom**

**Multiple R-Squared: 0.9728, Adjusted R-squared: 0.9713**

**F-statistic: 644.8 on 1 and 18 DF, p-value: 1.510e-15**

This shows the values for the coefficients (the intercept and the slope) plus their standard errors. We see, that the intercept is about 0.3 (compare this to the graph!) and the slope is roughly 1 (as expected). The multiple  $R^2$  value is 0.97, which means that our regression line explains about 97% of the data points.





## 10 Working with large datasets

R has some major advantages over many other statistical software packages (and spreadsheet-based applications such as MS Excel).

So let us create a sample dataset with hundred columns and thousand rows to see the 'power' of R!

```
x<-rnorm(100*1000)  
dim(x)<-c(1000,100)  
fix(x)
```

- "rnorm" samples from a standard normal distribution 100\*1000 times
- "dim" tells R that x shall be divided into 1000 rows and 100 columns
- "fix" lets us inspect the newly created dataset in a spreadsheet-like manner

|       | col1        | col2       | col3        | col4         |
|-------|-------------|------------|-------------|--------------|
| [1,]  | -1.24650940 | -1.1467091 | -0.74042771 | -0.005724874 |
| [2,]  | -0.01967889 | -0.2151414 | 0.09073095  | 0.425347667  |
| [3,]  | 0.89854828  | -0.1263240 | -0.77334287 | 0.807388085  |
| [4,]  | -0.92055640 | -2.4847047 | -0.04134650 | -0.309679654 |
| [5,]  | 0.06183694  | 0.7479552  | 0.85152309  | -2.000672747 |
| [6,]  | -0.34801913 | -0.7275596 | -0.96744469 | -1.198995246 |
| [7,]  | 0.23341553  | 0.3177949  | -0.23907289 | -1.804130980 |
| [8,]  | -1.38771819 | 0.7156175  | 0.28950161  | 0.549642668  |
| [9,]  | 0.91197232  | 0.7816294  | -0.68355235 | 0.215647213  |
| [10,] | -0.21823882 | 0.3123124  | -0.19459990 | -0.672982445 |

This shows the principal arrangements of "spreadsheets" (data frames, matrices) in R:

The rows and columns are addressed using pairs of x and y values of the form [row,column]. Thus, the cell in the first row of column 1 would be addressed as x[1,1]:

```
> x[1,1]  
[1] -1.246509
```

The cell in the 7<sup>th</sup> row of column 4 would be addressed as x[7,4]:

```
> x[7,4]  
[1] -1.804131
```

So this looks O.K.; Let's now calculate the sum of every row (i.e., thousand row sums):

```
rowSums(x)
```

Similarly, we can do this for all the 100 columns:

```
colSums(x)
```

Now for the tricky part: Why don't we create a hundred graphs from this dataset? Well, in any point-and-click software package, this would mean at least one day of work (selecting any one column and plotting it against any other!) - in R it's just that easy:

```
par(ask=T)
```

This tells the graphics device to let you press "return" to see the next graph. And then:

```
for (i in 1:99) plot(x[,i],x[,i+1])
```

That's all we need! Well, to make the graphs look a bit better, we would ideally want to have each x and y axis labelled, so the full call to R could look like:

```
for (i in 1:99) plot(x[,i],x[,i+1],  
xlab=c("x",i),ylab=c("x",i+1),  
main=c("The ",i,"th plot"))
```

These two lines tell R to plot each column (i) in the dataframe against each next column (i+1), so that we end up with 99 plots in total. The code reads as follows:

```
for (i in 1:99).....do the following thing 99 times:  
plot.....create a new plot with  
x[,i].....the i'th column against  
x[,i+1].....the (i+1)th column  
xlab.....label the x axis with:  
c("x",i)....."x" and the column number (i) and  
ylab.....label the y axis with:  
c("x",i+1)....."x" and the next column number (i+1)  
main.....give the graph a main title which is  
c("The ",i,"th plot").numbered from 1 to i
```

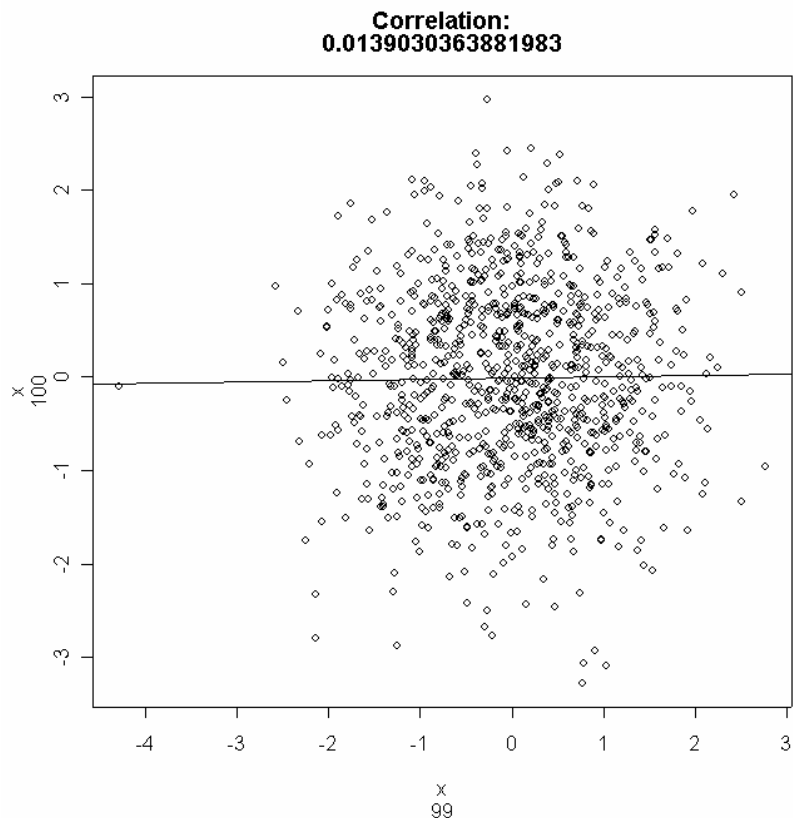
Now let's take this a step further and add regression lines and correlation coefficients to each of the 99 plots:

```
for (i in 1:99)  
{ correl<-cor(x[,i],x[,i+1])  
plot(x[,i],x[,i+1],xlab=c("x",i),ylab=c("x",i+1),  
main=c("Correlation:",correl))  
abline(lsfite(x[,i],x[,i+1]))
```

```
}
```

The code now reads as follows:

```
for (i in 1:99).....do the following thing 99 times:  
correl<-.....create a variable called "correl"  
cor(x[,i],x[,i+1])....calculate the correlation coefficient  
plot.....plot the same thing as before  
abline.....create a line based on the formula:  
lsfit(x[,i],x[,i+1])..a regression of x[,i] against x[,i+1]
```



## 11 Sorting and summarizing data

Let's now come back to an easier dataset; we make it very small so that everything is quick and easy to see:

```
x<-c("c","a","b")  
y<-c(10,13,20)  
z<-c(5,10,1)  
w<-cbind(x,y,z)
```

w

```
   x    y    z  
[1,] "c" "10" "5"  
[2,] "a" "13" "10"
```

```
[3,] "b" "20" "1"
```

There are three columns (x,y and z) and three rows. Column x contains a categorical variable with the levels "a", "b" and "c". Y and z are numerical. We now want to sort the data in this small data frame. We can do this individually (i.e. column-wise) using sort:

```
sort(w[,1])  
[1] "a" "b" "c"
```

But ideally we want the whole dataframe w to be sorted, e.g. after column 1:

```
w[order(w[,1]),]  
  x  y  z  
[1,] "a" "13" "10"  
[2,] "b" "20" "1"  
[3,] "c" "10" "5"
```

Additionally, we might want to have some quantitative summaries, such as

```
table(w)  
w  
  1 10 13 20  5  a  b  c  
  1  2  1  1  1  1  1  1
```

which shows how often elements occur in the dataframe; or

```
summary(w)  
  x      y      z  
a:1 10:1  1 :1  
b:1 13:1 10:1  
c:1 20:1  5 :1
```

which gives a similar output, but column-wise.

We can also calculate mean values for every level of x:

```
tapply(y,x,mean)
```

```
a  b  c  
13 20 10
```

Which reads: "Apply the function 'mean' to y for every value of x."

## 12 Creating high-level plots in R

You can find out about the full capabilities of R's graphics system by typing

```
demo(graphics)
```

which will create several built-in demo graphs.

### 12.1 Exploratory Data Analysis

So let's try out the full capabilities of R, using datasets provided by the Software Developers.

All available datasets can be accessed by typing

```
data()
```

Note: This function is only available in R, in S-Plus you'd have to type `?example`

### 12.2 Plotting a Histogram

We start with a sample dataset on tree growth.

```
attach(trees)  
names(trees)
```

```
[1] "Girth"  "Height"  
[3] "Volume"
```

```
hist(Height)
```

We could also improve the output of our histogram, using

```
hist(Height)
```

```
hist(Height, breaks=5,ylim=c(0, 12),col="grey")
```

Let's further assume we want to see if our data are normally distributed:

First, we need to find out the minimum and the maximum of Height:

```
min(Height)  
[1] 63  
max(Height)
```

```
[1] 87
```

```
hist(Height,breaks=63:87,col="grey")
```

Now we can draw the corresponding normal curve using **lines()** and a call to the routine **dnorm()** to plot the curve. This involves several calculations:

First, we have to create a vector of "artificial" x values for our normal curve. We do this by typing

```
x.values<- seq(63,87,length=31)
```

Now we create a corresponding vector of y values, which is going to be the height of our normal curve. The parameters we supply to the **dnorm()** function ("d" stands for "density") are:

- the number of trees that were measured: **length(Height)**
- the probability density for each tree, which is going to be a normal distribution with mean **mean(Height)** and standard deviation **sd(Height)**

We stick this all together using

```
y.values<-  
length(Height)*  
dnorm( seq(63,87,length=31), mean(Height), sd(Height))
```

Finally, we say:

```
lines(x.values,y.values)
```

And it's done!

### 12.3 A simple Scatterplot

Let's use our trees dataset again. There are several different ways of producing scatterplots, which will be shown step by step.

First, let's set up a graphics window that is split into four parts, using the **par()** command.

```
par(mfrow=c(2, 2))
```

This says: Set up a 2x2 panel screen (using **mfrow**); if you'd like, you could also increase the font size, using **cex()**:

```
par(mfrow=c(2, 2), cex=0.6)
```

So let's look at the data now:

```
plot(Height, type="p") #gives only single points
plot(Height, type="l") #gives invisible points joint by lines
plot(Height, type="b") #gives both points and lines
plot(Height, type="h") #gives the height of the y values

plot(Height, type="l", col="blue") #changes the color
plot(Height, type="l", col="red", lwd=3) #changes color&width
plot(Height, log="x", type="l") #changes the scale of x axis
plot(Height, type="l", lty="dotted") #line type is dotted now
```

## 12.4 Plotting multivariate data

Let's use the Iris dataset. Here it is:

```
fix(iris)
```

```
Sepal.Length Sepal.Width      (...) and so on
1             5.1          3.5
2             4.9          3.0
3             4.7          3.2
4             4.6          3.1
5             5.0          3.6
6             5.4          3.9
```

We want to see how the following variables:

- sepal length,
- sepal width,
- petal length and
- petal width

are related to one another. We first increase font size using `par()`:

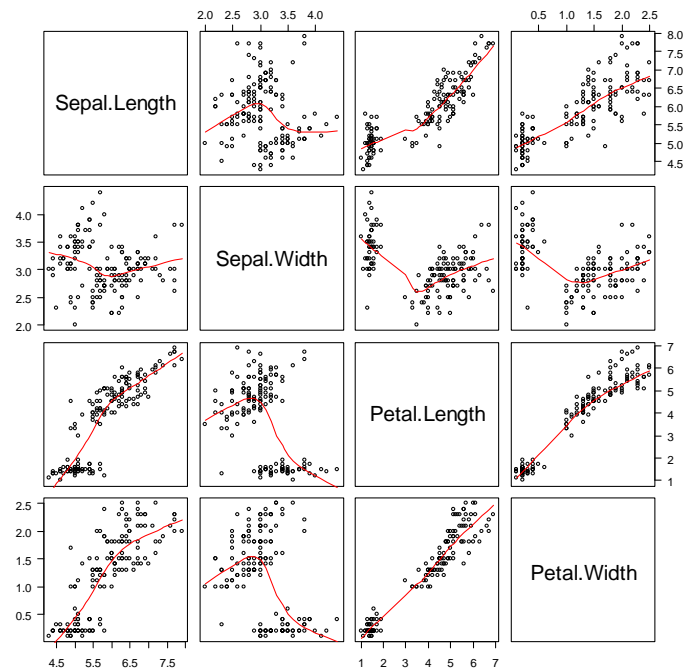
```
par(cex=0.6)
```

Now the `pairs()` command does the trick for us. We are only going to look at the first four columns in the dataset, using an index from 1 to 4:

```
pairs(iris[1:4])
```

It is also possible to add scatterplot smoothers quickly to each of the panels:

```
pairs(iris[1:4],panel=panel.smooth)
```



Another possibility to visualize multivariate data is using the `coplot()` command:

```
#Data:
```

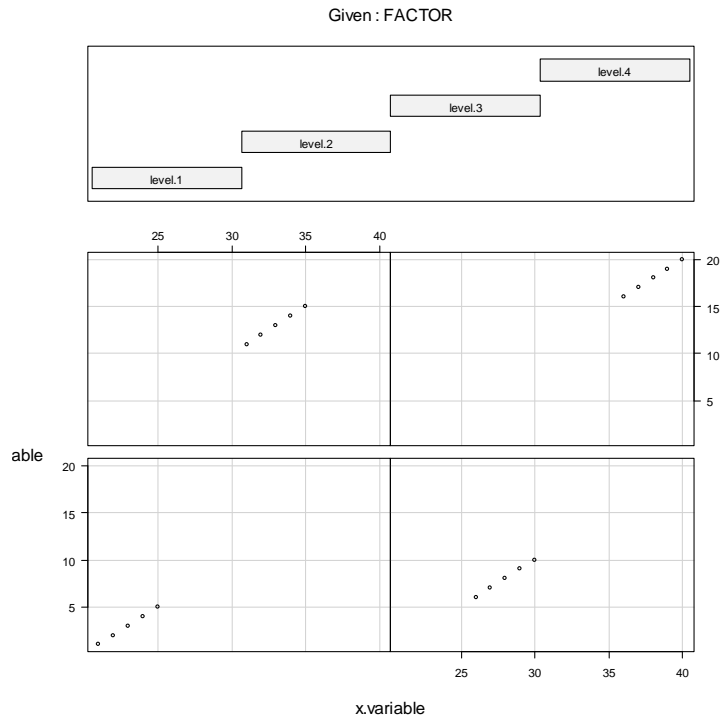
```
y.variable <- c(1:20)
x.variable <- c(21:40)
f <- rep(c("level.1", "level.2", "level.3", "level.4"), c(5,
5, 5, 5))
FACTOR <- factor(f)
```

```
data.4.coplot <- data.frame(x.variable, y.variable, FACTOR) ;
data.4.coplot
```

Now we can plot `y.variable` as a function of `x.variable` conditioned by our categorical variable called `FACTOR`:

```
coplot(y.variable ~ x.variable | FACTOR)
```





The panels in `coplot()` are read from bottom left via bottom right to top left and top right.

## 12.5 Line plot

```
x <- c(0.5, 2, 4, 8, 12, 16)
y1 <- c(1, 1.3, 1.9, 3.4, 3.9, 4.8)
y2 <- c(4, .8, .5, .45, .4, .3)
par(las=1, mar=c(4, 4, 2, 4))
plot.new()

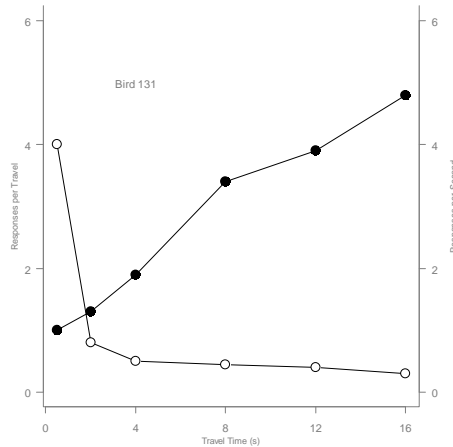
plot.window(range(x), c(0, 6))

lines(x, y1)
lines(x, y2)
points(x, y1, pch=16, cex=2)
points(x, y2, pch=21, bg="white", cex=2)

par(col="grey50", fg="grey50", col.axis="grey50")
axis(1, at=seq(0, 16, 4))
axis(2, at=seq(0, 6, 2))
axis(4, at=seq(0, 6, 2))

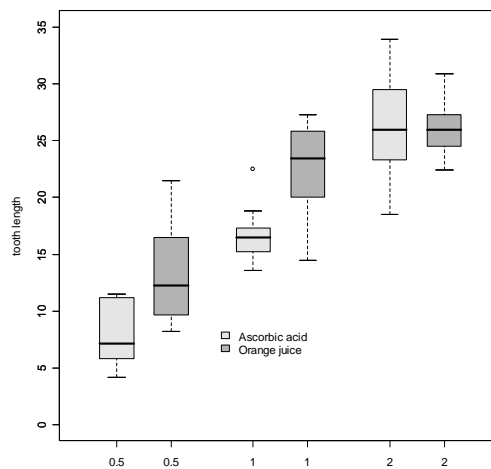
box(bty="u")
mtext("Travel Time (s)", side=1, line=2, cex=0.8)
mtext("Responses per Travel", side=2, line=2, las=0, cex=0.8)
mtext("Responses per Second", side=4, line=2, las=0, cex=0.8)
text(4, 5, "Bird 131")
```

```
par(mar=c(5.1, 4.1, 4.1, 2.1), col="black", fg="black",
col.axis="black")
```



## 12.6 Boxplot

```
par(mar=c(3, 4.1, 2, 0))
  boxplot(len ~ dose, data = ToothGrowth,
  boxwex = 0.25, at = 1:3 - 0.2,
  subset= supp == "VC", col="grey90",
  xlab="",
  ylab="tooth length", ylim=c(0,35))
  mtext("Vitamin C dose mg", side=1, line=2.5, cex=0.8)
  boxplot(len ~ dose, data = ToothGrowth, add = TRUE,
  boxwex = 0.25, at = 1:3 + 0.2,
  subset= supp == "OJ", col="grey70")
  legend(1.5, 9, c("Ascorbic acid", "Orange juice"),
  bty="n",
  fill = c("grey90", "grey70"))
par(mar=c(5.1, 4.1, 4.1, 2.1))
```



## 12.7 Three-dimensional plots

Here is an example from the Agricultural University of Copenhagen (Denmark):

```
mydata <- data.frame(x = runif(20, 0, 10), y = runif(20, 10, 20), zinc = rnorm(20, 4, 2))
```

```
mydata[1:15, ]
```

```
library(lattice)
```

```
cloud(zinc~x*y,data=mydata,scales=list(arrows=F))
```

More complicated surfaces can be plotted using the **wireframe** function, which gives complex three-dimensional representations of data. The example below comes from Deepayan Sarkar, the programmer of the lattice library:

First, we use **expand.grid ()** to create the source dataset.

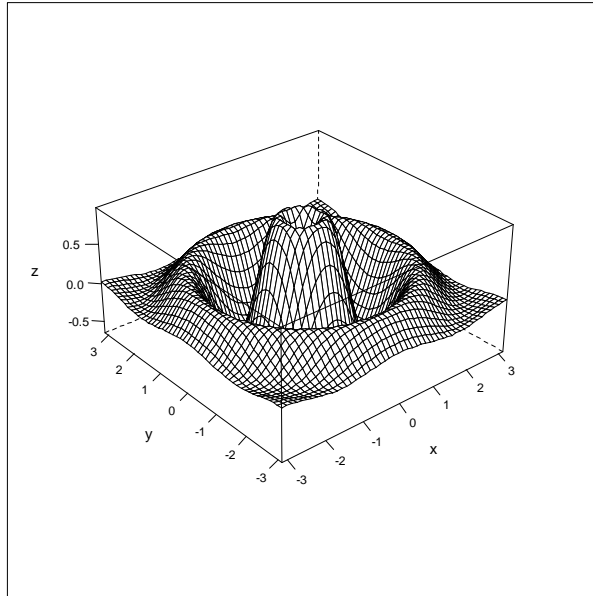
```
surf <-expand.grid(x = seq(-pi, pi, length = 50),  
                  y = seq(-pi, pi, length = 50))
```

Now the z variable (plotted perpendicularly to the x and y plane) shall be a complex sine function of x and y:

```
surf$z <-  
  with(surf, {  
    d <- 3 * sqrt(x^2 + y^2)  
    exp(-0.02 * d^2) * sin(d)  
  })
```

```
g=surf
```

```
wireframe(z ~ x * y, g, aspect = c(1, .5),  
          scales = list(arrows = FALSE))
```



Now let us assume you would wish to add scatter points to this plot.

To modify this plot, write an own **panel function** using `panel.3d.wireframe`:

```
wireframe(z ~ x * y, g, aspect = c(1, .5),
          scales = list(arrows = FALSE),
          panel.3d.wireframe = function(...) {
            panel.3dwire(...)
          })
```

...and add points using `3dscatter`; The trick is to make liberal use of the `...` argument, only naming arguments that you need to work with or override. So our first try might be to write an explicit but minimal **panel.3d.wireframe** function that does nothing new:

```
wireframe(z ~ x * y, g, aspect = c(1, .5),
          scales = list(arrows = FALSE),
          panel.3d.wireframe = function(x, y, z, ...) {
            panel.3dwire(x = x, y = y, z = z, ...)
          })
```

Now let's add a few points using **panel.3dscatter**:

```
wireframe(z ~ x * y, g, aspect = c(1, .5),
          scales = list(arrows = FALSE),
          panel.3d.wireframe = function(x, y, z, ...) {
            panel.3dwire(x = x, y = y, z = z, ...)
            panel.3dscatter(x = runif(10, -0.5, 0.5),
                           y = runif(10, -0.5, 0.5),
                           z = runif(10, -0.25, 0.25),
```

```

        ...))
    })

```

Now usually the points to add to such a 3D plot will not be on the transformed 3D scale (they almost always will be on some original scale), whereas `panel.3dwire` wants data in a different (linearly shifted and scaled) scale suitable for 3-D transformations.

In practice, you would have to make the conversion from data scale to transformed scale yourself, and that's where the `*lim` and `*lim.scaled` arguments come in. They contain the range of the data cube in the original and transformed scales respectively.

So let's say the points you want to add (in the original scale) are:

```

pts <-
  data.frame(x = runif(10, -pi, pi),
            y = runif(10, -pi, pi),
            z = runif(10, -1, 1))

```

Then the suitable transformation can be done as follows:

```

wireframe(z ~ x * y, g, aspect = c(1, .5),
          scales = list(arrows = FALSE),
          pts = pts,
          panel.3d.wireframe =
            function(x, y, z,
                    xlim, ylim, zlim,
                    xlim.scaled, ylim.scaled, zlim.scaled,
                    pts,
                    ...) {
              panel.3dwire(x = x, y = y, z = z,
                          xlim = xlim,
                          ylim = ylim,
                          zlim = zlim,
                          xlim.scaled = xlim.scaled,
                          ylim.scaled = ylim.scaled,
                          zlim.scaled = zlim.scaled,
                          ...)
            }
xx <-
  xlim.scaled[1] + diff(xlim.scaled) *
    (pts$x - xlim[1]) / diff(xlim)
yy <-
  ylim.scaled[1] + diff(ylim.scaled) *
    (pts$y - ylim[1]) / diff(ylim)
zz <-
  zlim.scaled[1] + diff(zlim.scaled) *
    (pts$z - zlim[1]) / diff(zlim)

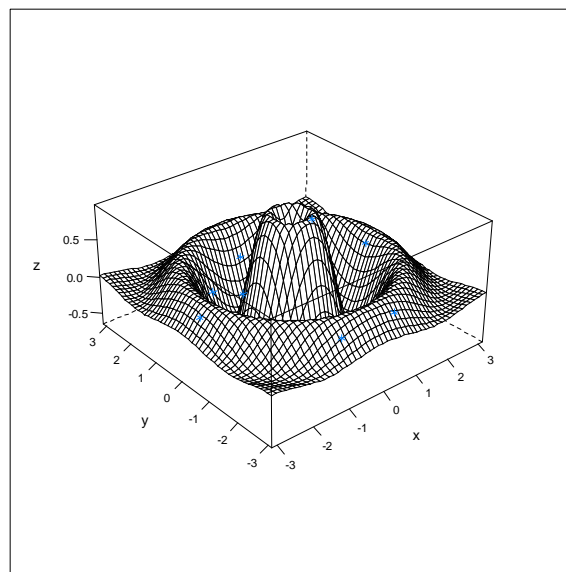
```

```

panel.3dscatter(x = xx,
               y = yy,
               z = zz,
               xlim = xlim,
               ylim = ylim,
               zlim = zlim,
               xlim.scaled = xlim.scaled,
               ylim.scaled = ylim.scaled,
               zlim.scaled = zlim.scaled,
               ...)
})

```

The resulting plot now contains the original wireframe, plus some added datapoints floating around in 3D space:



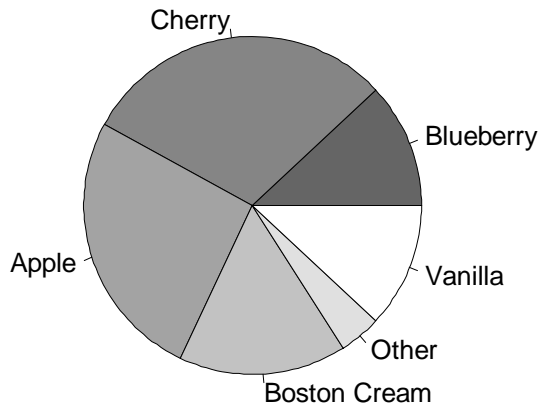
## 12.8 Piecharts

Here is an example of a piechart from the `pie()` help page:

```

par(mar=c(0, 2, 1, 2), xpd=FALSE, cex=2)
pie.sales <- c(0.12, 0.3, 0.26, 0.16, 0.04, 0.12)
names(pie.sales) <- c("Blueberry", "Cherry",
                    "Apple", "Boston Cream", "Other", "Vanilla")
pie(pie.sales, col = gray(seq(0.4,1.0,length=6)))

```



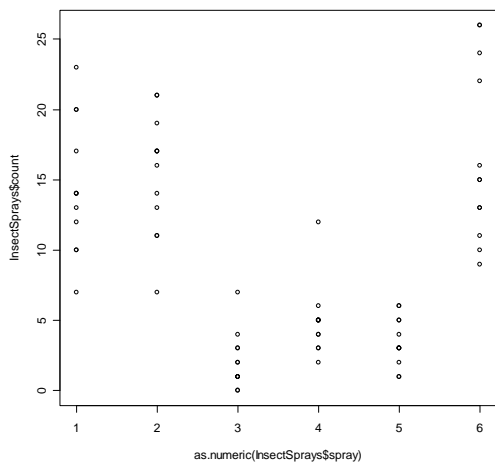
## 12.9 Trellis Scatterplots

Trellis plots offer several possibilities to visualize multidimensional data. Specifically, you can plot the response variable against several numerical and categorical explanatory variables at the same time.

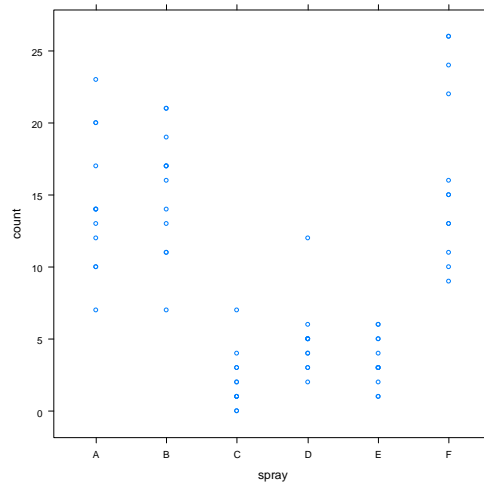
Trellis plots are highly effective, but they involve different sets of commands compared with standard graphics.

To get a first feeling for how trellis graphics work, here is a comparison between `xyplot()` and `plot()`:

```
plot(as.numeric(InsectSprays$spray), InsectSprays$count)
xyplot(count~spray, data=InsectSprays)
```



Conventional plot()



xyplot() version

So, in general, the most important difference between both plot types is the specification of (x,y) vs. (y~x).

However, xyplots can become quite complex once you want to change things such as font sizes etc.; the command "trellis.par.get" lists all the components of a trellis plot that you might want to change:

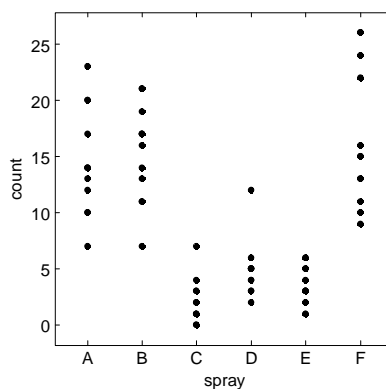
```
trellis.par.get()
```

The list of parameters is huge; to change just a few of them, we type

```
trellis.par.set(  
list(fontsize=list(text=14),  
par.xlab.text=list(cex=1.5),  
par.ylab.text=list(cex=1.5),  
par.sub.text=list(cex=1.5),  
add.text=list(cex=1.5),  
plot.symbol=list(pch=16,cex=1.3)))
```

And then re-run the xyplot command:

```
xyplot(count~spray,InsectSprays,scales=list(tck=-1,cex=1.5))
```



The barley dataset, included in the lattice library, serves as a further example:

```
library(lattice)  
trellis.par.set(theme = canonical.theme("postscript",  
col=FALSE))  
trellis.par.set(list(fontsize=list(text=6),  
par.xlab.text=list(cex=1.5),  
add.text=list(cex=1.5),  
plot.symbol=list(cex=.5)))  
key <- simpleKey(levels(barley$year), space = "right")  
key$text$cex <- 1.5  
print(  

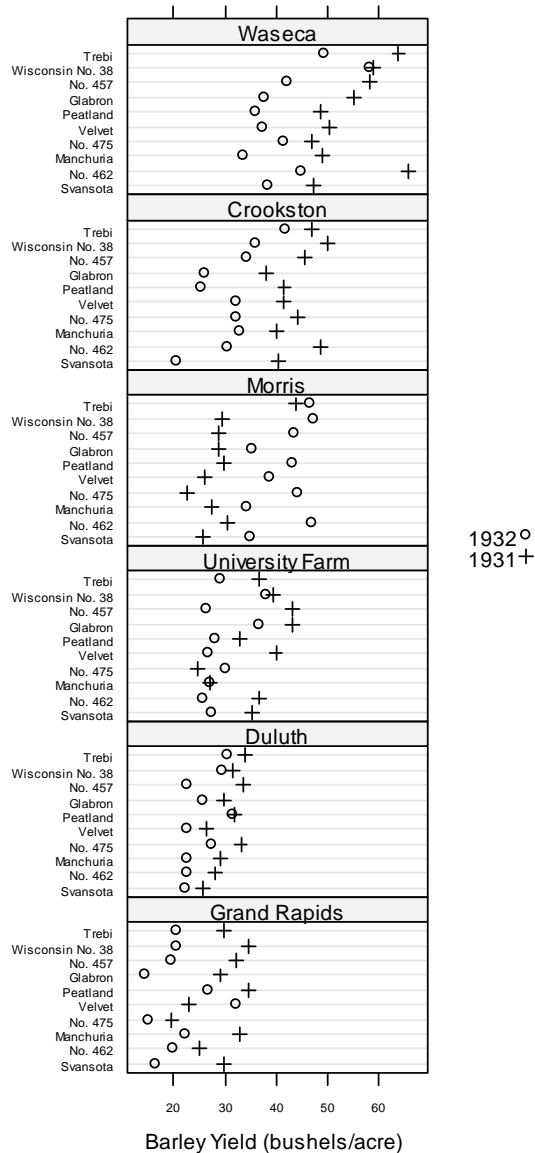
```



```

dotplot(variety ~ yield | site, data = barley, groups =
year,
        key = key,
        xlab = "Barley Yield (bushels/acre) ",
        aspect=0.5, layout = c(1,6), ylab=NULL)
)

```



However, trellis plots show their strengths especially with multiple explanatory variables. Here is a more complex example using the built-in Iris dataset:

```

trellis.device(theme="col.whitebg")
library(lattice)

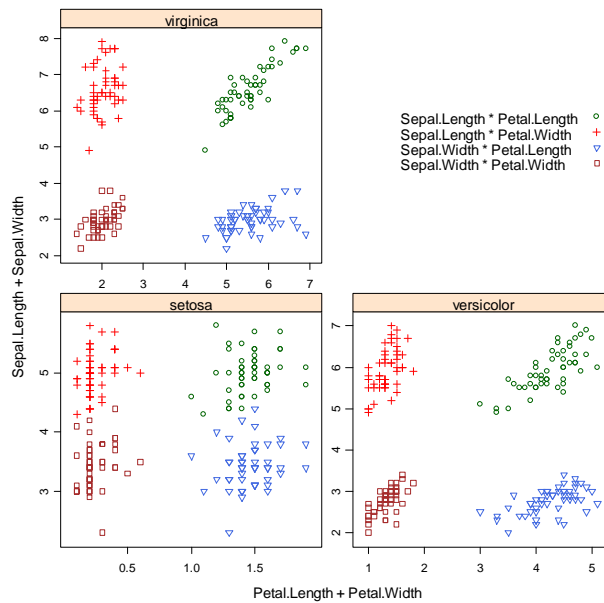
```

```

xyplot(Sepal.Length + Sepal.Width ~ Petal.Length + Petal.Width
| Species,
data = iris, scales = "free",
layout = c(2, 2),

```

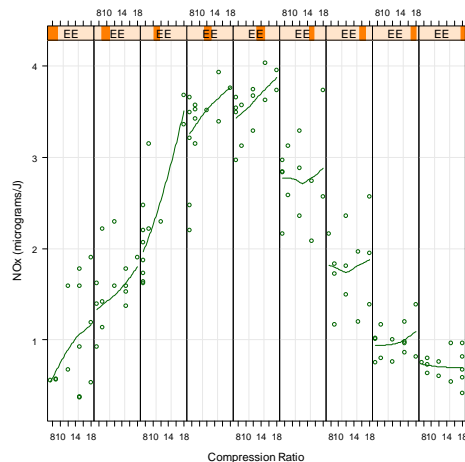
```
auto.key = list(x = .6, y = .7, corner = c(0, 0))
```



A nice example, also from the lattice library, is the Ethanol dataframe:

```
EE <- equal.count(ethanol$E,
number=9, overlap=1/4)
```

```
xyplot(NOx ~ C | EE,
data = ethanol, prepanel =
function(x, y)
prepanel.loess(x, y, span =
1),
xlab = "Compression Ratio",
ylab = "NOx (micrograms/J)",
panel = function(x, y) {
panel.grid(h=-1, v= 2)
panel.xyplot(x, y)
panel.loess(x,y, span=1) },
aspect = "xy")
```



### 13 Creating pdf's and postscript files

There are options that enable you to directly create pdf or postscript documents from within R. For example, there's a special graphics device, the **pdf device**, which can be called using the **pdf()** command like this:

```
pdf("C:\\File1.pdf",horizontal=FALSE,onfile=FALSE,pointsize=1
6,family="Times",height=7.5,width=10,pagecentre=FALSE)
```

The `pdf()` command is just the first step in creating a pdf document; it just "tells" R that all further operations will not produce a conventional graphics output, but instead create a pdf file. Thus, you will need to tell R when to finish, using the `dev.off()` command.

Here comes an example:

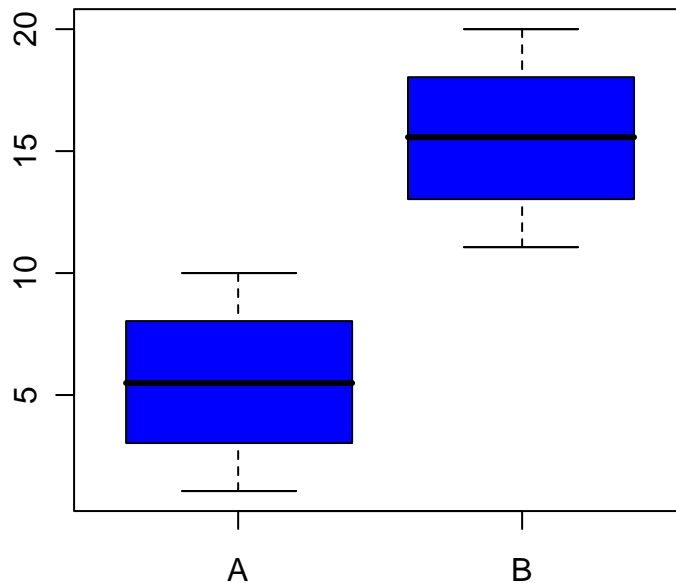
```
A<-seq(1,20,1) #creates two vectors A and B
B<-c(rep("A",10),rep("B",10))

pdf("C:\\File1.pdf",horizontal=FALSE,onefile=FALSE,pointsize=16,
family="Times",height=7.5,width=10,pagecentre=FALSE)

par(lwd=1,las=1,mgp=c(2.3,1,0))

plot(as.factor(B),A,col="blue")

dev.off()
```



If you want to do **more advanced pdf graphics**, you can install the so-called **lattice** package. With this package installed, you can create graphs and save them as pdf using a special graphics device, the so-called **trellis device**.

First, create three vectors of values A, B and C:

```
A<-seq(1,20,1) #creates two vectors A and B
B<-c(rep("A",10),rep("B",10))
```

```
C<-c(rep("C",5),rep("D",5),rep("E",5),rep("F",5))
```

Now load the lattice package,

```
library(lattice)
```

and start the trellis device like this:

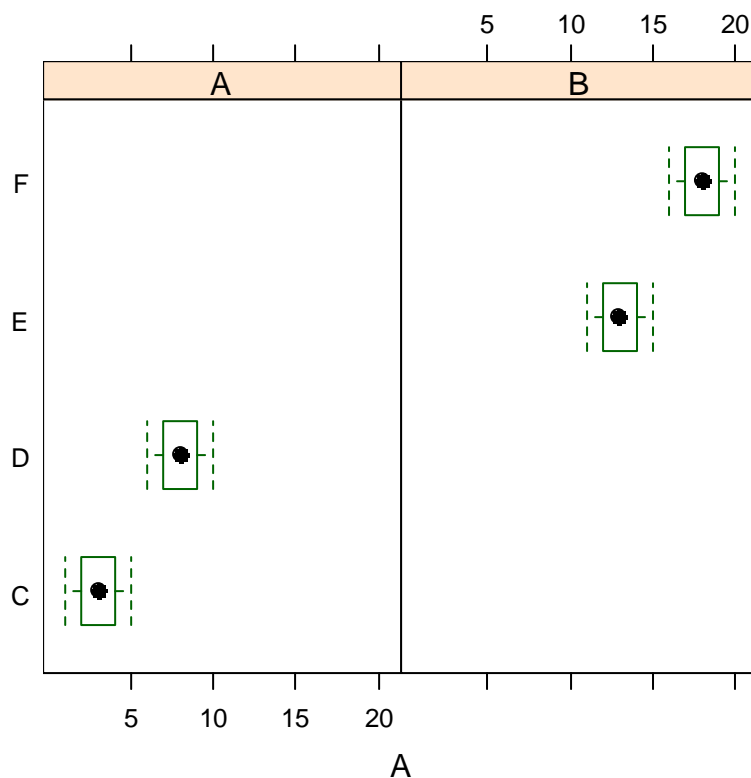
```
trellis.device(pdf,file="C:\\File2.pdf",  
encoding="WinAnsi",paper="special",pointsize=12,onefile=T)
```

Now comes the plotting command:

```
bwplot(as.factor(C)~A|as.factor(B))
```

In the end, the trellis device needs to be closed again, using the `dev.off()` command:

```
dev.off()
```



Alternatively, you may wish to create a **postscript** file using the postscript driver inside the trellis device:

```
library(lattice)  
trellis.device(postscript,file="C:\\File1.ps",color=F)  
bwplot(as.factor(C)~A|as.factor(B))  
dev.off()
```

## 14 Creating publication-quality graphs

This is an example kindly provided by Andy Hector (Zurich, Switzerland):

First of all, we start with creating two linearly correlated variables:

```
x.var.1 <- c(1:10)
sequence.1 <- c(1:10)
noise <- rnorm(10, 1, 0.1)
y.var.1 <- sequence.1 + noise
```

Now we edit some of the graphics parameters;

```
par(mfrow=c(1,2),
    mar=c(5,4,4,2)+0.1,
    bty="l", pty="s", cex.lab=0.9,
    tck=0.02, mgp=c(2, 0.3, 0))
```

```
y.label <- expression(paste("Insect density (m-2," )"))
x.label <- expression(paste("plant mass (g m-2," )"))
y.lim <- c(0, 10) ; x.lim <- c(0,10)
```

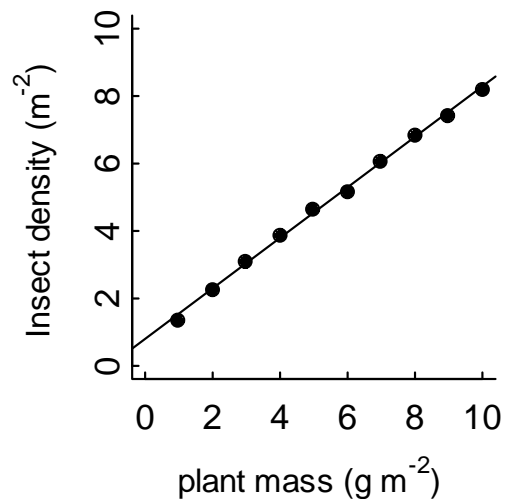
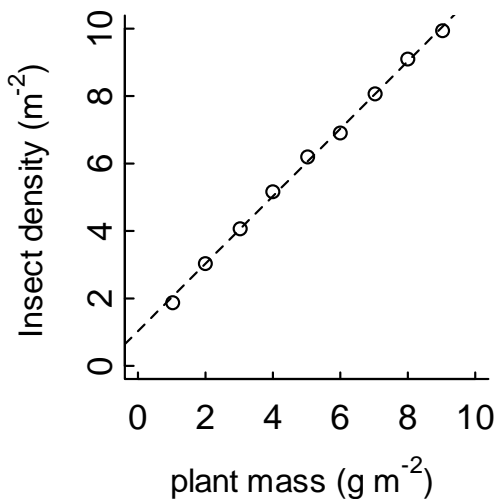
and finally start plotting:

```
plot(y.var.1 ~ x.var.1,
     ylab=y.label, xlab=x.label,
     ylim=y.lim, xlim=x.lim)
```

```
abline(lm(y.var.1 ~ x.var.1), lty = 2)
main="a) Insect A" ;
```

```
plot(y.var.2 ~ x.var.1, pch = 16,
     ylab=y.label, xlab=x.label,
     ylim=y.lim, xlim=x.lim)
```

```
abline(lm(y.var.2 ~ x.var.1))
main="b) Insect B" ;
```



## 15 Statistical Modelling

### 15.1 Simple tests: The t test

The following example (from Altman 1991, cited in Dalgaard 2002) tests whether the daily energy intake of 11 women differs significantly from a recommended value of 7725 kJ. We can type this small dataset in directly, inspect some of its properties and test it versus the expected value with the `t.test` function (if we do not specify `mu` the default value is taken as zero):

```
intake <- c(5260, 5470, 5640, 6180, 6390, 6515, 6805, 7515,
7515, 8230, 8770)
```

```
summary(intake)
```

```
t.test(intake, mu=7725)
```

```
Min. 1st Qu. Median Mean 3rd Qu. Max.
5260 5910 6515 6754 7515 8770
```

```
t.test(intake, mu=7725)
```

```
One Sample t-test
```

```
data: intake
t = -2.8208, df = 10, p-value = 0.01814
alternative hypothesis: true mean is not equal to 7725
95 percent confidence interval:
 5986.348 7520.925
sample estimates:
mean of x
 6753.636
```

The output gives the value of  $t$ , its probability and degrees of freedom (which we could report in the text of a report or paper).

## 15.2 Model Formulae in R

All model formulae in R have a similar form:

General form:

|                                             |
|---------------------------------------------|
| response variable ~ explanatory variable(s) |
|---------------------------------------------|

### Linear Models

|                                                             |                        |
|-------------------------------------------------------------|------------------------|
| $y \sim x$                                                  | - Simple regression    |
| $y \sim 1 + x$                                              | - Explicit intercept   |
| $y \sim -1 + x$                                             | - Through the origin   |
| $y \sim x + x^2$                                            | - Quadratic regression |
| $y \sim x_1 + x_2 + x_3$                                    | - Multiple regression  |
| $y \sim G + x_1 + x_2$                                      | - Parallel regressions |
| $y \sim G / (x_1 + x_2)$                                    | - Separate regressions |
| $\text{sqrt}(\text{Hard}) \sim \text{Dens} + \text{Dens}^2$ | - Transformed          |

### ANOVA

|                                                  |                    |
|--------------------------------------------------|--------------------|
| $y \sim A + B$                                   | - two-way ANOVA    |
| $y \sim A * B$                                   | - factorial ANOVA  |
| $y \sim A * B + \text{Error}(\text{Block/plot})$ | - split-plot ANOVA |
| $y \sim A / B / C$                               | - nested ANOVA     |

### More Model formulae

|            |                         |
|------------|-------------------------|
| $y \sim G$ | - Single classification |
|------------|-------------------------|

|                              |                            |
|------------------------------|----------------------------|
| <code>y ~ A + B</code>       | - Randomized block         |
| <code>y ~ B + N*P</code>     | - Factorial in blocks      |
| <code>y ~ x + B + N*P</code> | - with covariate           |
| <code>y ~ . - X1</code>      | - All variables except X1  |
| <code>. ~ . + A:B</code>     | - Add interaction (update) |

### 15.3 Regression

First, we create some artificial data to illustrate a very simple linear regression :

```
x.var.1 <- c(1:10)
sequence.1 <- c(1:10)
```

Now we add some random noise to `sequence.1` :

```
noise <- rnorm(10, 1, 0.1)
y.var.1 <- sequence.1 + noise
```

Finally, we create a second `y` variable, with all values being 25% lower than `y.var.1`:

```
y.var.2 <- y.var.1 - (0.25 * y.var.1)
data.1 <- data.frame(x.var.1, y.var.1)
```

Plotting these variables yields:

```
plot(x.var.1, y.var.1,col="blue")
points(x.var.1,y.var.2,col="red")
```

Our linear regression analysis involves the `lm()` command, and we can plot the results using

```
abline(lm(y.var.1 ~ x.var.1), lty = 2,col="blue")
abline(lm(y.var.2 ~ x.var.1), lty = 2,col="red")
```

Now let's come back to our initial dataset:

```
x <- 1:20
```

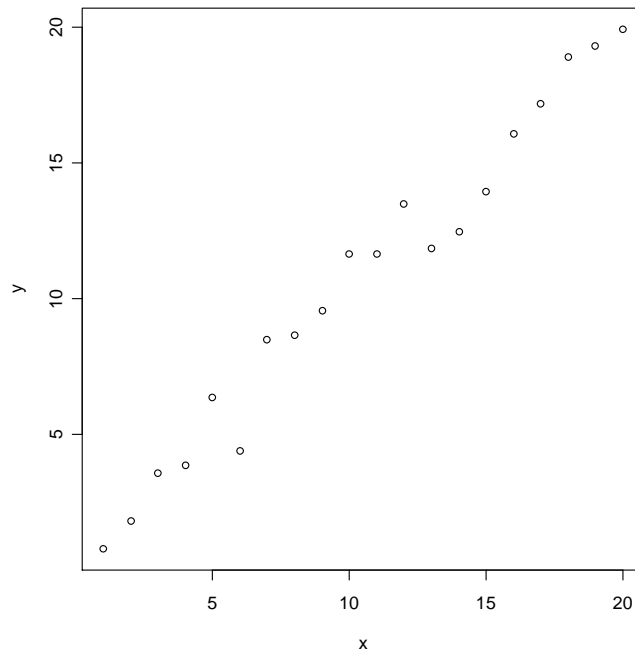
Make `y` a linear function of `x` plus normally distributed deviations:

```
y <- x+rnorm(x)
```

Now create a plot of `y` against `x`:

```
plot(x,y)
```





All data points seem to lie roughly along a straight line, so it is sensible to try to fit a linear regression through the data:

```
modell<-lm(y~x)
abline(modell)
```

And we inspect the parameter estimates using

```
summary(modell)
```

#### 15.4 Non-Linear Regression

Let's come back to our example from the course (the "Puromycin" dataset):

Data on the "velocity" of an enzymatic reaction were obtained by Treloar (1974). The number of counts per minute of radioactive product from the reaction was measured as a function of substrate concentration in parts per million (ppm) and from these counts the initial rate, or "velocity," of the reaction was calculated (counts/min/min). The experiment was conducted once with the enzyme treated with Puromycin, and once with the enzyme untreated.

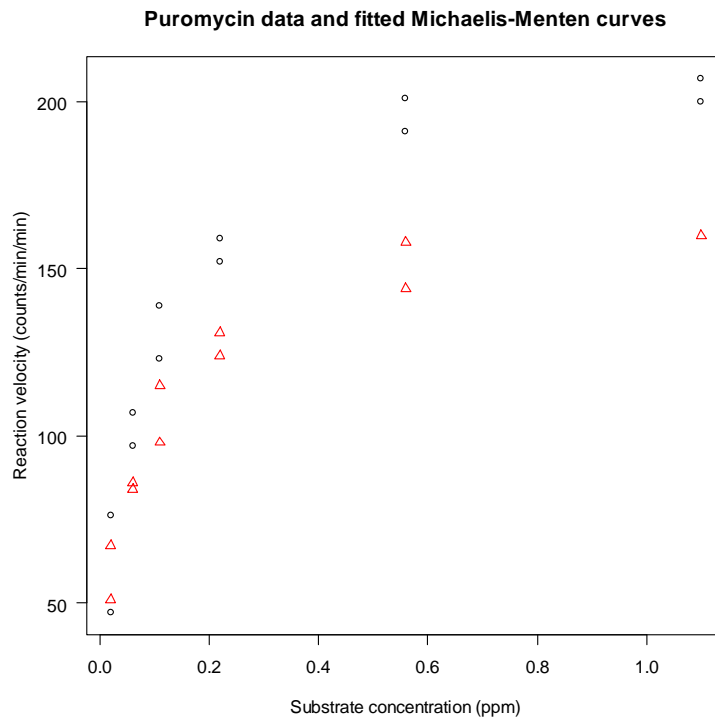
First, let's plot the data:

```
plot(rate ~ conc, data = Puromycin, las = 1,
      xlab = "Substrate concentration (ppm)",
      ylab = "Reaction velocity (counts/min/min)",
```

```

pch = as.integer(Puromycin$state),
col = as.integer(Puromycin$state),
main = "Puromycin data and fitted Michaelis-Menten
curves")

```



Now, we can fit a Michaelis-Menten model to these data:

```

fm1 <- nls(rate ~ Vm * conc/(K + conc), data = Puromycin,
           subset = state == "treated",
           start = c(Vm = 200, K = 0.05), trace = TRUE)
fm2 <- nls(rate ~ Vm * conc/(K + conc), data = Puromycin,
           subset = state == "untreated",
           start = c(Vm = 160, K = 0.05), trace = TRUE)
summary(fm1)
summary(fm2)

```

And now we can add the fitted lines to the plot:

```

conc <- seq(0, 1.2, len = 101)
lines(conc, predict(fm1, list(conc = conc)), lty = 1, col = 1)
lines(conc, predict(fm2, list(conc = conc)), lty = 2, col = 2)

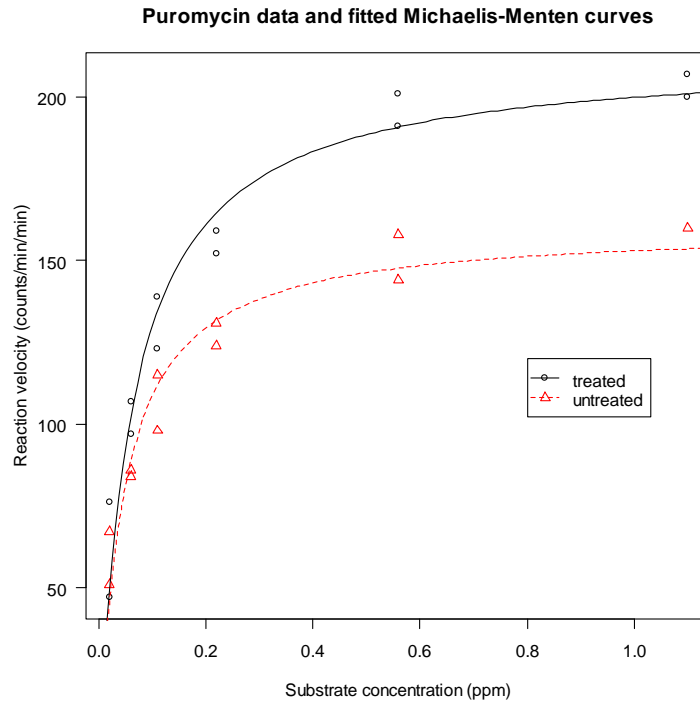
```

And, finally, a legend:

```

legend(0.8, 120, levels(Puromycin$state),
      col = 1:2, lty = 1:2, pch = 1:2)

```



Let's try another example: The **"trees"** data set provides measurements of the girth, height and volume of timber in 31 felled black cherry trees. Note that girth is the diameter of the tree (in inches) measured at 4 ft 6 in above the ground.

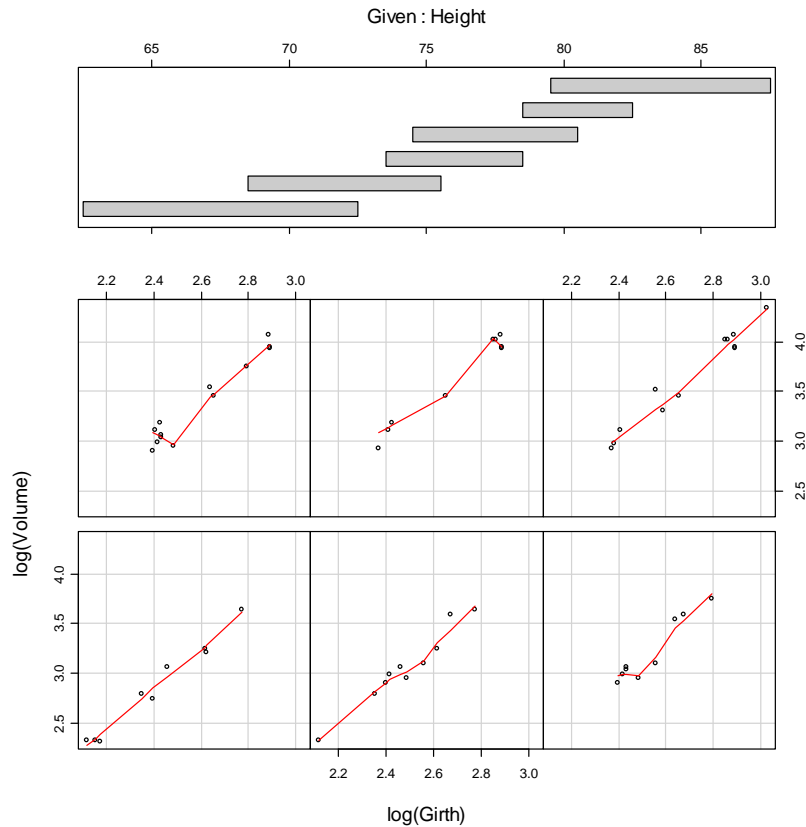
First, let's look at different ways to plot these data:

```
pairs(trees, panel = panel.smooth, main = "trees data")
```

```
plot(Volume ~ Girth, data = trees, log = "xy")
```

```
par(mai=c(10,5,5,4))
```

```
coplot(log(Volume) ~ log(Girth) | Height, data = trees,
       panel = panel.smooth)
```



Now it seems that taking logs at both sides has linearized the data:

```
modell1<-lm(log(Volume) ~ log(Girth), data = trees)
summary(modell1)
```

```
model2 <- update(modell1, ~ . + log(Height), data = trees)
summary(model2)
```

```
Call:
lm(formula = log(Volume) ~ log(Girth) + log(Height), data = trees)
```

```
Residuals:
    Min       1Q   Median       3Q      Max
-0.168561 -0.048488  0.002431  0.063637  0.129223
```

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -6.63162    0.79979  -8.292 5.06e-09 ***
log(Girth)   1.98265    0.07501  26.432 < 2e-16 ***
log(Height)  1.11712    0.20444   5.464 7.81e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 0.08139 on 28 degrees of freedom
Multiple R-Squared:  0.9777,    Adjusted R-squared:  0.9761
F-statistic: 613.2 on 2 and 28 DF,  p-value: < 2.2e-16
```

## 15.5 Analysis of variance

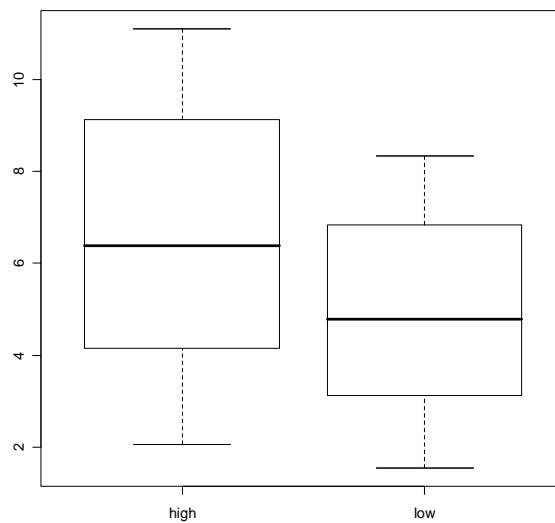
We can use our data from section "Regression" to work on:

```
x.var.1 <- c(1:10)
sequence.1 <- c(1:10)
noise <- rnorm(10, 1, 0.1)
y.var.1 <- sequence.1 + noise
y.var.2 <- y.var.1 - (0.25 * y.var.1)
data.1 <- data.frame(x.var.1, y.var.1)

y.long <- c(y.var.1, y.var.2) ; y.long
X <- rep(c("high" , "low"), c(10, 10)) ; X
X <- factor(X)
data.long <- data.frame(X, y.long) ; data.long ;
```

What do these data look like if we plot them?

```
plot(X, y.long)
```



```
par(mfrow=c(1,2))
```

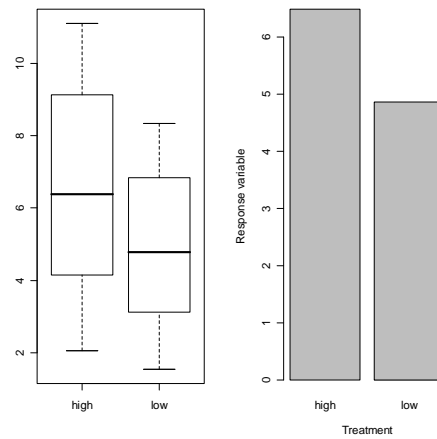
```
#Boxplot
```

```
boxplot(y.long ~ X)
```

```
#Barplot
```

```
y.mean <- tapply(y.long, X, mean)
```

```
barplot(y.mean, ylab="Response variable", xlab="Treatment",
names=levels(X))
```

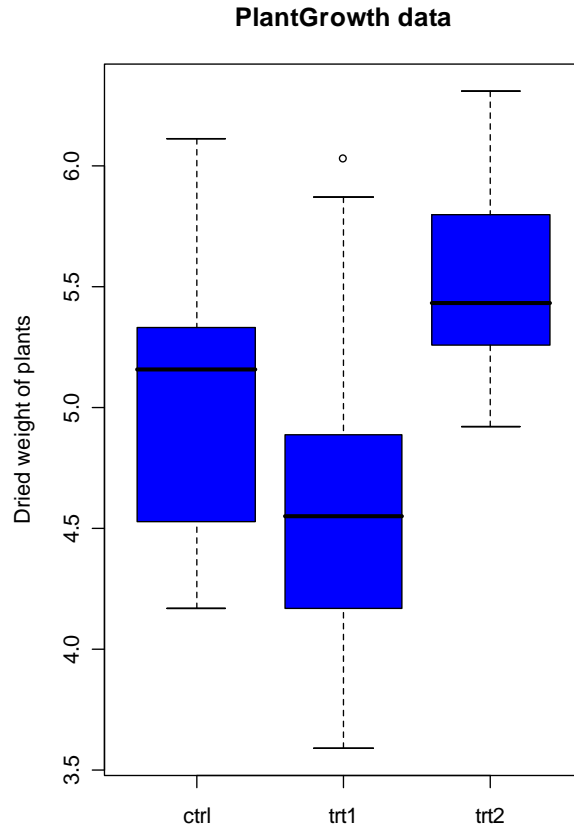


So far for the graphs of this dataset.

Now let's try an easy one-way ANOVA using a **new dataset**. The **response variable** is continuous (growth of plants). The results are from an experiment to compare yields (as measured by dried weight of plants) obtained under a control and two different treatment conditions.

Let's inspect our data first:

```
require(stats)
boxplot(weight ~ group, data = PlantGrowth, main =
"PlantGrowth data",ylab = "Dried weight of plants", col =
"blue")
```



We could have also used boxplots with notches, as in:

```
boxplot(weight ~ group, data = PlantGrowth, main =
"PlantGrowth data",ylab = "Dried weight of plants", col =
"lightgray", notch = TRUE, varwidth = TRUE)
```

Now, here comes the ANOVA model:

```
modell1<-aov(weight ~ group, data = PlantGrowth)
summary(modell1)
```

```
          Df Sum Sq Mean Sq F value Pr(>F)
group      2  3.7663  1.8832   4.8461 0.01591 *
Residuals 27 10.4921  0.3886
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

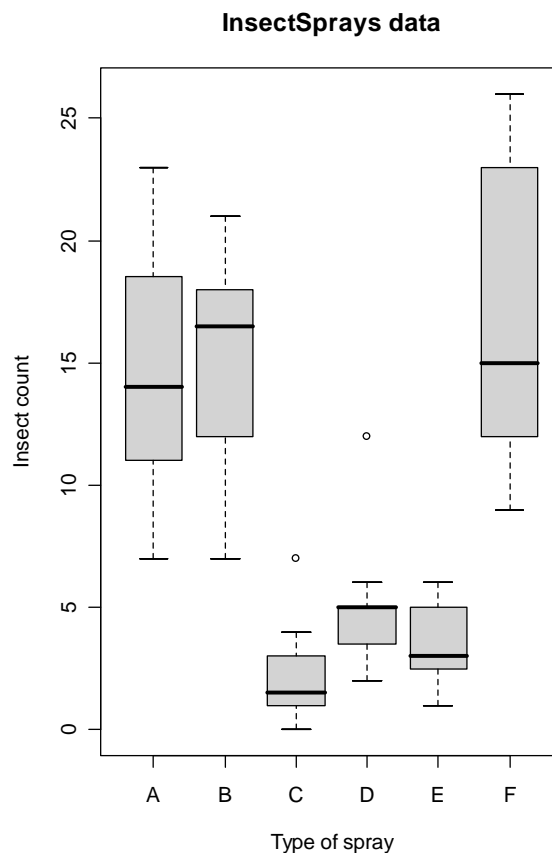
The next example we want to try out comes from an Agricultural experiment:

```
data(InsectSprays)
names(InsectSprays)
InsectSprays
```

The response variable is counts of insects in agricultural experimental units treated with different insecticides.

As usual, we start by plotting and inspecting the data:

```
require(stats)
boxplot(count ~ spray, data = InsectSprays,
        xlab = "Type of spray", ylab = "Insect count",
        main = "InsectSprays data", varwidth = TRUE,
        col = "lightgray")
```



Now, let's construct our first ANOVA model:

```
fm1 <- aov(count ~ spray, data = InsectSprays)
summary(fm1)
par(mfrow = c(2,2))
plot(fm1)
```

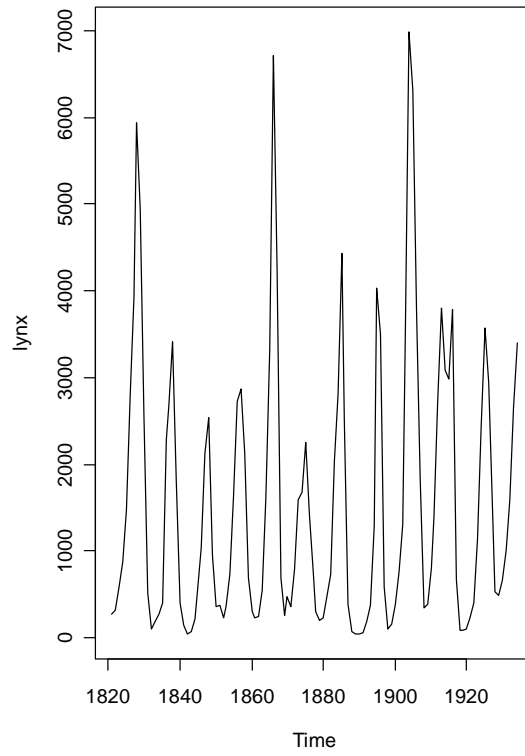
Let's try the same analysis using a transformation of the response:

```
fm2 <- aov(sqrt(count) ~ spray, data = InsectSprays)
summary(fm2)
plot(fm2)
```

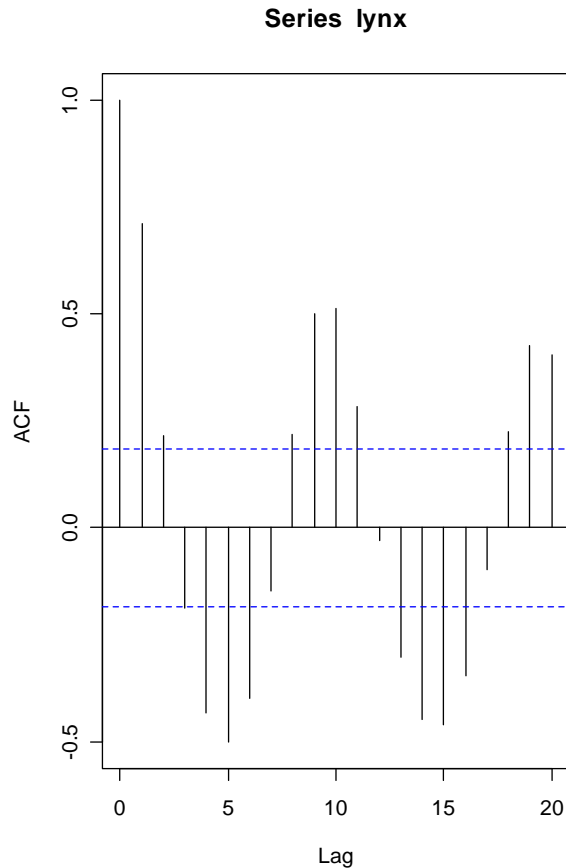


## 15.6 Time Series

```
data(lynx)  
ts.plot(lynx)
```



```
plot(acf(lynx))
```



### 15.7 A Generalized Linear Model (from Bill Venables)

Generalized linear models are an own class of models where you can specify what error structure your data have. Generalized linear models don't have to be linear; they have to be linear **in their parameters**. In general, such a model consists of three parts:

- the linear predictor
- the link function and
- the Error Structure

The **linear predictor** is just another form of a regression equation ( $y=a+bx$ ), namely

$$y = \sum x\beta, \text{ where}$$

- y is the vector of individual data points (observations)
- the xs are the covariates in the model
- the  $\beta$ s are the parameters or the model (whose values are to be estimated)

The **link function** links the linear predictor to the observations:

$\eta = \sum x\beta$  is the linear predictor;  $\eta = \sum x\beta$

The link function is then just the reciprocal of  $y=f(\eta)$ .

The Error structure refers to the kind of errors associated with our data, e.g.

- Poisson Errors for count data
- Binomial Errors for proportion data
- Gamma Errors for data on time-to-death

Specific Error Structures are often associated with so-called canonical link functions:

- Normal Errors: Identity link
- Poisson Errors: Log link
- Gamma Errors: Reciprocal link
- Binomial Errors: Logit Link

So, let's create again a dataset:

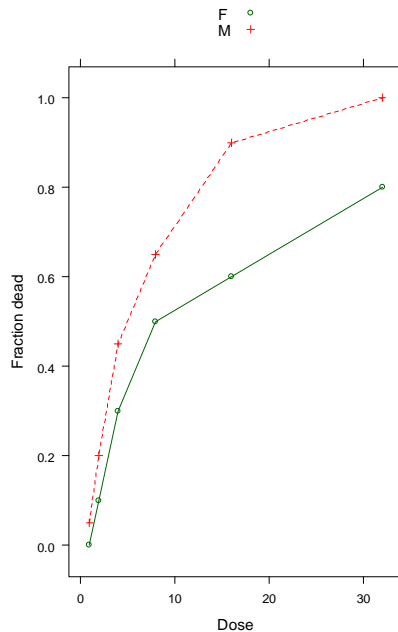
```
Budworms <- data.frame(Logdose = rep(0:5, 2),  
  Sex = factor(rep(c("M", "F"), each = 6)),  
  Dead = c(1, 4, 9, 13, 18, 20, 0, 2, 6, 10, 12, 16))
```

Obviously, these are going to be proportion data (as budowrms are either "dead" or "alive").

```
Budworms$Alive <- 20 - Budworms$Dead
```

To plot these data, we need the Lattice package:  
`library(lattice)`

```
xyplot(Dead/20 ~ I(2^Logdose), Budworms, groups = Sex, panel =  
panel.superpose, xlab = "Dose", ylab = "Fraction  
dead", auto.key=T , type="b")
```



And here comes the generalized linear model, using `glm()`:

```
bud.1 <- glm(cbind(Dead, Alive) ~ Sex*Logdose, binomial,
Budworms, trace=T, eps=1.0e-9)
```

```
summary(bud.1)
```

"binomial" indicates we're using binomial errors with a logit link function. Let's see if we can simplify this model:

```
bud.0 <- update(bud.1, .~-Sex:Logdose)
```

Finally, we compare both models using "`anova()`":

```
anova(bud.0, bud.1, test="Chisq")
```

## 16 Generating Experimental Designs

There are some very useful built-in functions in R that allow you to generate layouts for experimental designs.

The `gl()` command generates levels of a factor (which is very useful for factorial designs):

```
gl(2, 8, label = c("Control", "Treatment"))
```

```
[1] Control Control Control Control Control Control Control
[8] Control Treatment Treatment Treatment Treatment Treatment Treatment
[15] Treatment Treatment
Levels: Control Treatment
```

## INDEX

### A

**abline** 15, 18, 19, 37, 40, 41  
**analysis of variance** 6  
Analysis of variance 45  
**ANOVA** 39, 46, 47, 48  
**aov** 47, 48  
**artificial dataset** 13  
**aspect** 27, 28, 29, 33, 34  
assignment 13  
**attach** 11, 12, 21  
**auto.key** 34  
axis 18, 23, 25, 26

### B

barley, dataset 32  
base 3  
Binary Distributions 3  
Binomial Errors 51  
blank cells 12  
blank space 12  
**box** 25  
**boxplot** 26, 46, 47, 48  
Boxplots 4, 26, 45  
**boxwex** 26  
**bwplot** 36

### C

**c** 13, 17, 18, 19, 20, 21, 22, 23, 25, 26, 30, 33, 35, 36, 38, 42, 48, 51  
categorical variable 20  
**cbind** 19, 52  
**close** 11  
coefficients 15, 18  
**Coefficients** 15  
**col** 8, 21, 22, 23, 25, 26, 30, 32, 35, 42, 46, 47, 48  
**colSums** 18  
command line 7  
commands 7, 8, 9  
Contributed Packages 7  
coplot 24, 43  
**cor** 18, 19  
correlation coefficient 19

### D

**data** 10, 11, 12, 13, 14, 15, 17, 19, 20, 21, 23, 26, 33, 39, 41, 42, 43, 46, 47, 48, 49, 50, 51  
**data.frame** 10, 51  
dataframe 10, 18, 20  
dec 12  
**Decimal comma** 12  
decimal point 12  
**degrees of freedom** 15, 39  
**dev.off** 35, 36  
**dim** 17  
**dnorm** 22  
**dotplot** 33  
download 2

### E

Error Structure 50  
ethanol, dataset 34  
Excel files, importing 11  
**exp** 8  
**expand.grid** 27  
Experimental Designs 52  
Exploratory Data Analysis 21  
**explorer** 9  
exporting 10  
**expression, add mathematical annotations** 37

### F

**File** 8, 9  
fill 12, 26  
**fix** 17, 23  
**F-statistic** 15

### G

Gamma Errors 51  
Generalized Linear Model 50  
**getwd()** 10  
**gl** 52  
graph 9, 14, 15, 18  
**Graphical User Interface** 9, 14  
Graphics, recording  
  graphics.record 14

graphs, creating publication  
quality 4, 37

## H

**header** 11, 12  
**help** 6, 7  
**help.search** 6  
**hist** 21, 22

## I

Import Excel 7  
Importing 10, 11  
**InsectSprays, dataset** 31,  
32, 47, 48  
install 3, 7, 35  
interactive plots 7  
intercept 15, 39  
**Intercept** 15  
**iris, dataset** 23, 33

## L

**lattice** 32, 35, 36, 51  
**layout** 33  
**legend** 26, 42  
**length** 12, 22, 23, 26, 30  
**library** 7, 11, 32, 36, 51  
**Linear Models** 39  
linear predictor 50, 51  
linear regression 14, 41  
**lines** 18, 22, 23, 25, 42  
link function 50, 51, 52  
**lm** 15, 41  
**lm, linear models** 37, 40  
**log** 23, 43  
**lsfit** 18, 19  
**lty** 23, 42

## M

**main** 7, 18, 37, 42, 43, 46,  
47, 48  
**mar** 25, 26, 30, 37  
**max** 21  
**mean** 18, 20, 22, 39  
**mfrow** 22, 23, 37, 45, 48  
**mgp** 35, 37  
Microsoft Excel 10  
**Microsoft Office 2007** 11  
**min** 21, 41  
missing values 12  
mixed effects models 7

mixed-effects models 7  
Model Formulae 39  
mouse cursor 9  
MS-DOS 9  
**mtext** 25, 26

## N

**na.strings** 12  
**names** 2, 21, 30, 47  
nlme 7  
normal curve 22  
numerical 20

## O

**odbcConnectExcel** 11  
Operating System 3  
**options** 14  
**order** 20

## P

**packageDescription** 7  
**pairs** 17, 23, 43  
panel 22, 43, 51  
**panel.3dscatter** 28, 30  
**panel.3dwire** 28, 29  
**panel.grid** 34  
**panel.loess** 34  
**par** 18, 22, 23, 25, 26, 30,  
32, 35, 48  
pdf 34, 35, 36  
**plot** 8, 14, 18, 19, 22, 23,  
25, 32, 35, 39, 40, 41, 42,  
43, 48, 49, 51  
points 14, 15, 23, 25, 41,  
50  
Poisson Errors 51  
**postscript** 32, 34, 36  
**prepanel** 34  
**print** 8, 32  
programming language 7

## R

R Console 3  
**read.table** 12  
**read.xls** 11  
Regression 40, 41  
regression lines 18  
**rep** 10, 13, 35, 36, 51  
**Residual standard error** 15  
**rnorm** 14, 17, 27, 37, 40

RODBC 11  
rowSums 18  
R-Squared 15  
runif 27, 28, 29

## S

save 9, 10, 35  
**scales** 27, 28, 29, 32, 33  
**scan** 13  
Scatterplot 22, 31  
Script Window 8  
scripts 7  
**sd** 22  
**sep** 12  
**seq** 8, 10, 13, 22, 25, 30,  
35, 42  
**setwd** 10  
**simpleKey** 32  
slope 15  
**sort** 20  
Sorting 19  
spatial data 7  
spreadsheet 17  
**sqlFetch** 11  
**step** 13, 18, 22, 35  
summarizing 19  
**summary** 20, 38, 41, 42, 47,  
48, 52  
Summary 15  
**system** 9, 10

## T

t test 38  
**t.test** 38, 39  
**table** 20  
**tapply** 20  
text 7, 8, 9, 10, 11, 12,  
25, 32, 39

text editor 7  
**Text files, advantages over  
Excel files** 12  
**Text files, tab-delimited**  
11  
three-dimensional plots 26  
tree models 7  
Trellis 31  
**trellis device** 35, 36  
trellis graphics 7  
Trellis plots 31  
**trellis.device** 33, 36  
**trellis.par.get** 32  
**trellis.par.set** 32  
**type** 6, 7, 8, 21, 23, 38,  
51

## V

vector 13, 22, 50

## W

**wireframe** 27, 28, 29, 30  
**Working directory, setting  
the** 10  
**write.table** 10, 12

## X

**xlab** 18, 26, 32, 33, 41,  
48, 51  
**xlsReadWrite** 10  
**xyplot** 31, 32, 33, 34, 51

## Y

**ylab** 18, 26, 33, 41, 46,  
47, 48, 51

## 17 Author's Address

Christoph Scherber  
DNPW  
Agroecology  
Waldweg 26  
37073 Goettingen  
**phone** 0551-39 8807  
**e-mail** cscherbl@gwdg.de