

---

# Introduction to R

September 4, 2019

Søren Højsgaard  
Department of Mathematical Sciences  
Aalborg University, Denmark  
<http://people.math.aau.dk/~sorenh/>

© 2019

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>My first R session</b>                               | <b>4</b>  |
| 1.1      | My first R commands . . . . .                           | 4         |
| 1.2      | R as a calculator . . . . .                             | 5         |
| 1.3      | Vectors . . . . .                                       | 6         |
| 1.4      | Using add-on packages . . . . .                         | 6         |
| 1.5      | Indexing . . . . .                                      | 7         |
| 1.6      | Dataframes . . . . .                                    | 9         |
| 1.7      | Simple (linear) regression . . . . .                    | 11        |
| 1.8      | Getting help . . . . .                                  | 12        |
| 1.9      | R as a programming language . . . . .                   | 13        |
| <b>2</b> | <b>A more nerdy section</b>                             | <b>14</b> |
| 2.1      | Generating random and systematic data . . . . .         | 14        |
| 2.2      | Matrices . . . . .                                      | 15        |
| 2.3      | Vectors, lists and dataframes . . . . .                 | 15        |
| 2.3.1    | Vectors . . . . .                                       | 15        |
| 2.3.2    | Lists . . . . .   | 16        |
| 2.3.3    | Matrices . . . . .                                      | 18        |
| 2.3.4    | Dataframes . . . . .                                    | 18        |
| 2.4      | Getting data into functions . . . . .                   | 19        |
| <b>3</b> | <b>Programming - R functions</b>                        | <b>20</b> |
| 3.1      | Hello World . . . . .                                   | 20        |
| 3.2      | More on input and returning values* . . . . .           | 20        |
| 3.3      | Conditional and repetetive execution . . . . .          | 21        |
| 3.4      | Example: Calculating the factorial . . . . .            | 22        |
| 3.5      | Example: Calculating the square root . . . . .          | 22        |
| 3.6      | Example: Recursion and iteration . . . . .              | 24        |
| 3.7      | Exercises – programming . . . . .                       | 24        |
| 3.7.1    | Calculating the (natural) logarithm . . . . .           | 24        |
| 3.7.2    | Calculating the $p$ th root . . . . .                   | 25        |
| <b>4</b> | <b>Computer arithmetic</b>                              | <b>25</b> |
| 4.1      | Computer arithmetic is not exact . . . . .              | 25        |
| 4.2      | Addition and subtraction – Significant digits . . . . . | 26        |
| 4.3      | Floating point arithmetic . . . . .                     | 27        |
| 4.4      | The Relative Machine Precision . . . . .                | 27        |
| 4.5      | Floating-Point Precision . . . . .                      | 28        |
| 4.6      | Error in Floating-Point Computations . . . . .          | 28        |
| 4.7      | Example: A polynomial . . . . .                         | 28        |
| 4.8      | Example: Numerical derivatives . . . . .                | 30        |
| 4.9      | Example: Solving a quadratic equation* . . . . .        | 32        |
| 4.10     | Example: Computing the exponential . . . . .            | 33        |
| 4.11     | EXERCISES: Programming . . . . .                        | 35        |
| 4.12     | Debugging R code . . . . .                              | 35        |
| <b>5</b> | <b>Graphics in R</b>                                    | <b>35</b> |
| 5.1      | Scatterplot using <code>plot()</code> . . . . .         | 35        |
| 5.2      | The Puromycin data . . . . .                            | 36        |
| 5.3      | Simple scatterplots – side by side . . . . .            | 37        |
| 5.4      | Annotating the plot . . . . .                           | 37        |
| 5.5      | Adding a title and some text . . . . .                  | 38        |
| 5.6      | Overlaying two scatterplots . . . . .                   | 39        |

|           |  |           |
|-----------|--|-----------|
| 5.7       | Adding legends . . . . .   | 40        |
| 5.8       | Conditional annotation . . . . .                                   | 41        |
| 5.9       | Add a reference line – <u>abline()</u> . . . . .                   | 42        |
| 5.10      | Connecting points and adding a smooth line . . . . .               | 43        |
| 5.11      | A small detour: Different ways of accessing data . . . . .         | 43        |
| 5.12      | Some special plots . . . . .                                       | 44        |
| 5.12.1    | Boxplot – <u>boxplot()</u> . . . . .                               | 44        |
| 5.12.2    | Dotplot – <u>stripchart()</u> . . . . .                            | 45        |
| 5.12.3    | Histogram and density – <u>hist()</u> , <u>density()</u> . . . . . | 45        |
| 5.12.4    | Q-Q plot – <u>qqnorm()</u> , <u>qqline()</u> . . . . .             | 47        |
| 5.12.5    | Grouped data . . . . .   | 47        |
| 5.12.6    | Scatterplotmatrix – <u>pairs()</u> . . . . .                       | 48        |
| <b>6</b>  | <b>Introduction – ggplot2</b>                                      | <b>50</b> |
| 6.1       | Scatterplots using <u>qplot()</u> . . . . .                        | 50        |
| 6.2       | Scatterplots using <u>ggplot()</u> . . . . .                       | 52        |
| 6.3       | Some special plots using <u>qplot()</u> . . . . .                  | 53        |
| 6.3.1     | Boxplot . . . . .  | 53        |
| 6.3.2     | Dotplot . . . . .  | 54        |
| 6.3.3     | Histogram . . . . .  | 55        |
| <b>7</b>  | <b>Summarizing data</b>  | <b>59</b> |
| 7.1       | Mesures of location . . . . .                                      | 60        |
| 7.2       | Measures of spread . . . . .                                       | 60        |
| 7.3       | Standardizing variables . . . . .                                  | 61        |
| 7.4       | An empirical rule . . . . .  | 62        |
| 7.5       | Covariance and correlation . . . . .                               | 62        |
| 7.6       | Computations . . . . .   | 63        |
| <b>8</b>  | <b>Linear models</b>   | <b>64</b> |
| 8.1       | Running example: The cars data . . . . .                           | 64        |
| 8.2       | Running example: The Puromycin data . . . . .                      | 66        |
| 8.3       | Linear models (informally) . . . . .                               | 67        |
| 8.4       | Polynomial regression . . . . .                                    | 68        |
| 8.5       | Analysis of covariance (ANCOVA) . . . . .                          | 69        |
| <b>9</b>  | <b>Linear models</b>   | <b>70</b> |
| 9.1       | Matrix representation of a linear model . . . . .                  | 70        |
| 9.2       | Least squares estimation . . . . .                                 | 71        |
| <b>10</b> | <b>Least squares estimation – in theory an practice</b>            | <b>72</b> |
| 10.1      | General remark on solving $Ax = b$ . . . . .                       | 72        |
| 10.2      | The general remark and normal equations . . . . .                  | 73        |
| 10.3      | Example: $X$ is nearly rank deficient . . . . .                    | 74        |
| 10.3.1    | The "textbook formula" fails . . . . .                             | 74        |
| 10.3.2    | $QR$ -factorization and the normal equations . . . . .             | 74        |
| 10.3.3    | $QR$ -factorization works . . . . .                                | 74        |

# 1 My first R session

## 1.1 My first R commands

After starting R we see a prompt (`>`) where commands are typed followed by hitting “enter”. R will evaluate the commands and print the result.

Anything that exists in R is an OBJECT (sometimes also called a VARIABLE); anything we do in R involves calling functions. A function takes zero, one or more objects as input and returns an object as output.

We create two objects (or variables), `m` and `n`, holding the values 10 and 1.56. This can be done in different ways:

```
m <- 10
n = 1.56
```

The ASSIGNMENT operators “<=” and “=” can be used interchangeably, and the effect above is to create variables `m` and `n` and store some values in them. Simply typing the names at the command prompt causes R to print the values:

```
m
## [1] 10
n
## [1] 1.56
```

Objects can be modified. For example

```
m <- m * 10 + 7
n = n + m
n
## [1] 108.56
```

The “<=”-assignment operator is perhaps the most intuitive: It reads: “Take whatever is on the right hand side and store in the object on the left hand side”.

For example, the `sqrt()` function takes a single (numerical) argument as input and computes the square root. We can have multiple computations on one line but then they must be separated by semi colon (“;”). Anything after a hashmark (#) is regarded as comments in R. For example:

```
## multiple computations on one line are separated by ";"
sqrt.n <- sqrt( n ); sqrt.n # Another comment
## [1] 10.419
```

The assignment operator `<=` and `+` are also functions: We can write<sup>1</sup>

```
"<=" ( n, "+" ( 2, 7 ) )
n
## [1] 9
```

Another example of a function in R is `q()` which is used for exiting (or quitting) R. If we simply type `q` then we see how the function is defined (which is not of interest to us at this stage). To actually invoke the function we must do

---

<sup>1</sup>Move to a nerdy section

| Symbol  | Meaning                           |
|---------|-----------------------------------|
| +       | Addition                          |
| -       | Subtraction                       |
| *       | Multiplication                    |
| /       | Division                          |
| ** or ^ | Exponentiation                    |
| <       | Less than                         |
| >       | Greater than                      |
| <=      | Less or equal                     |
| >=      | Greater or equal                  |
| ==      | Logical equal                     |
| %%      | Remainder after division (modulo) |
| %/%     | Integer part                      |

Table 1: R as a calculator

q()

## 1.2 R as a calculator

R functions nicely as a simple calculator. Tables 1 and 2 show examples of simple mathematical operations and functions, respectively. Below is an example on how to do some of the calculations.

```

-3^2          ## power
## [1] -9
sqrt(15)      ## square root
## [1] 3.873
sin(2)        ## sine function
## [1] 0.9093
log(5)        ## natural log with base e
## [1] 1.6094
log(5, base=10) ## log with base 10 (alternative: log10(5))
## [1] 0.69897
exp(5)        ## exponential function
## [1] 148.41
a <- 1/0; a    ## Infinity
## [1] Inf
b <- 0/0; b    ## Not a number
## [1] NaN
7 %/% 2       ## integer division
## [1] 3
7 %% 2        ## modulo; remainder
## [1] 1
pi            ## pi
## [1] 3.1416
1e-6         ## 10^(-6)
## [1] 1e-06

```

| Symbol               | Meaning |
|----------------------|---------|
| <code>exp(x)</code>  |         |
| <code>log(x)</code>  |         |
| <code>sqrt(x)</code> |         |
| <code>abs(x)</code>  |         |

Table 2: R as a calculator (part 2)

### 1.3 Vectors

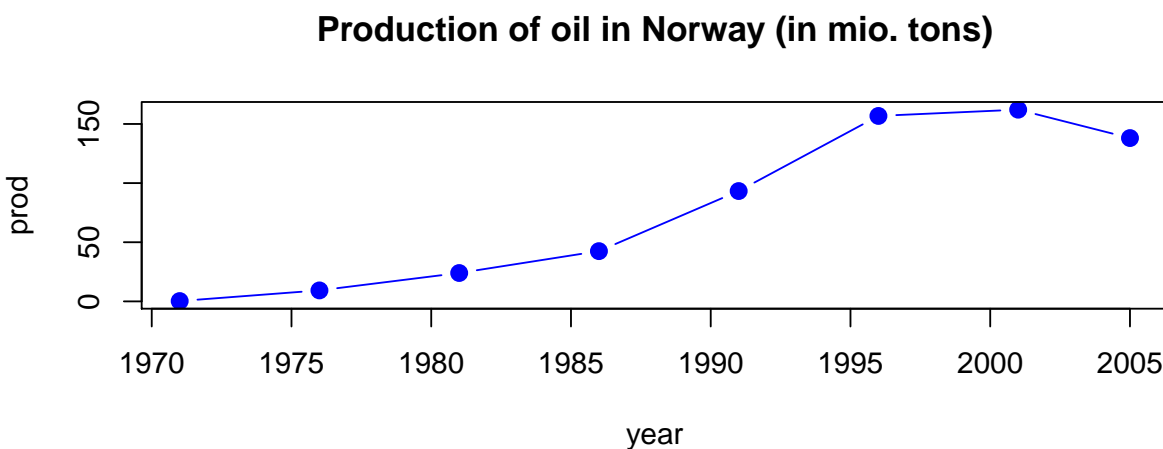
R has several data structures and vectors is the most fundamental one. (The variables `m` and `n` created in Section 1.1 are actually vectors of length 1). The function `c()` will concatenate its input into a vector.

Example: Production of oil in Norway (in mio. tons):

```
year <- c(1971, 1976, 1981, 1986, 1991, 1996, 2001, 2005)
year
## [1] 1971 1976 1981 1986 1991 1996 2001 2005
prod <- c(.3, 9.3, 24.0, 42.5, 93.3, 156.8, 162.1, 138.1)
prod
## [1] 0.3 9.3 24.0 42.5 93.3 156.8 162.1 138.1
```

Using R's own standard built-in graphics one can create a plot of the data for example as:

```
plot(year, prod, col="blue", pch=19, cex=1.2, type="b",
     main="Production of oil in Norway (in mio. tons)")
```



### 1.4 Using add-on packages

Much of R's versatility comes from the many add-on packages available from CRAN (the Comprehensive R Archive Network), see [www.r-project.org](http://www.r-project.org) (at the time of writing this, there are some 9.000 packages on CRAN). One package which we shall use extensively is the **ggplot2** package. This package does not come with R, so it must be installed separately on the computer which can be done as

```
install.packages("ggplot2")
```

This installation must be done only once.

To make the package available in an R session the package must be loaded which can be done with

```
library(ggplot2)
```

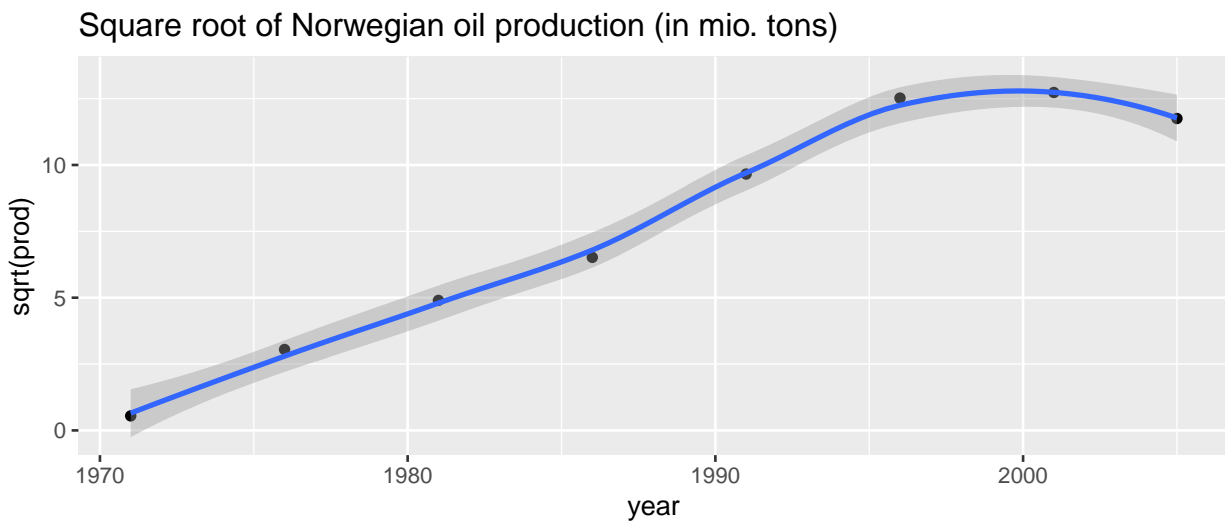
The package must be loaded in any R session where it is to be used.

Computations in R are VECTORIZED which has as a consequence that we can calculate the square root of all elements in `prod` as

```
sqrt( prod )  
## [1]  0.54772  3.04959  4.89898  6.51920  9.65919 12.52198  
## [7] 12.73185 11.75160
```

This can be used in connection with plotting the square root transformed production data. To plot these we choose to use the `qplot()` function:

```
qplot(year, sqrt(prod), geom=c("point", "smooth")) +  
  labs(title="Square root of Norwegian oil production (in mio. tons)")
```



Notice that the `geom` arguments specifies that we want to see the (transformed) data points and a smoothed curve.

## 1.5 Indexing

It is easy work with data using a simple indexing mechanism: On the square root scale, oil production grows approximately linearly until the end of the 20th century. Suppose that we want to create new vectors only with the data from the 20th century by extracting sub-vectors of `year` and `prod`.

A very simple approach is to notice that the relevant data are in the first 6 entries in the two data vectors. We can extract these data by creating a vector with the entries we want and put this into square brackets:

```
entries <- 1:6 # same as c(1, 2, 3, 4, 5, 6) but much shorter.
entries
## [1] 1 2 3 4 5 6
prod2 <- prod[ entries ]
prod2
## [1] 0.3 9.3 24.0 42.5 93.3 156.8
```

This works fine in this small example but pinpointing specific entries in a longer vector becomes tedious and error prone. We can therefore do the following:

```
b <- year < 2000
b
## [1] TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE
```

The statement `year < 2000` means that for each element in `year` it is checked whether the year is larger than 2000 or no. Hence this is another example of a vectorized computation. The result is a logical vector which is stored in the variable `b`.

To extract the year and production corresponding to the condition that year is smaller than 2000 we can do

```
year[ b ]
## [1] 1971 1976 1981 1986 1991 1996
prod[ b ]
## [1] 0.3 9.3 24.0 42.5 93.3 156.8
```

Notice that we have seen that we can index a vector by specifying entries or via a logical vector (of the same length as the vector we want to index). We just notice that we can easily get from the latter to the former using `which()`

```
which( b )
## [1] 1 2 3 4 5 6
```

The steps above do not change `year` and `prod`, the steps only extract sub vectors of these vectors. Let us create new variables containing these vectors:

```
year2 <- year[ b ]
prod2 <- prod[ b ]
```

To continue with the indexing, consider this:

```
prod
## [1] 0.3 9.3 24.0 42.5 93.3 156.8 162.1 138.1
b <- prod > 150; b
## [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE
```

The statement `prod > 150` implies that for each element in `prod` it is checked whether it is larger than 150 or not.

```
which( b ) ## where in 'prod' are these values ?
## [1] 6 7
prod[ b ] ## what are these values ?
## [1] 156.8 162.1
year[ b ] ## in which year was this production ?
## [1] 1996 2001
```



We can find the productions and years for the cases where the production is smaller than 150 using LOGICAL NEGATION ! (which turns TRUE into FALSE and vice versa):

```
prod[ !b ]
## [1] 0.3 9.3 24.0 42.5 93.3 138.1
year[ !b ]
## [1] 1971 1976 1981 1986 1991 2005
```

We can find data for production over 50 before the turn of the 20th century by combining logical expressions:

```
## '&' is logical AND
b <- (prod > 50) & (year < 2000); b
## [1] FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE
prod[ b ]; year[ b ]
## [1] 93.3 156.8
## [1] 1991 1996
```

Similarly, we can find production data before 1980 and after 2000:

```
## '|' is logical OR
b <- (year < 1980) | (year > 2000); b
## [1] TRUE TRUE FALSE FALSE FALSE FALSE TRUE TRUE
prod[ b ]; year[ b ]
## [1] 0.3 9.3 162.1 138.1
## [1] 1971 1976 2001 2005
```

Suppose we discover that all years before the end of the 20th century is off by one year. This is easy to repair

```
b <- year < 2000
year[ b ] <- year[ b ] + 1; year
## [1] 1972 1977 1982 1987 1992 1997 2001 2005
```

## 1.6 Dataframes

A dataframe is the “spreadsheet” of R. Data frames are usually formed by importing data from a text file into R, but a data frame can also be created from vectors using the data.frame() function; for example:

```
oil <- data.frame(Year=year, Prod=prod)
```

Hence, a dataframe provides a handle on many vectors at one time. The first rows of oil are displayed with

```
head( oil, 4 )
## Year Prod
## 1 1972 0.3
## 2 1977 9.3
## 3 1982 24.0
## 4 1987 42.5
```

We can extract a column in a dataframe in different ways; for example

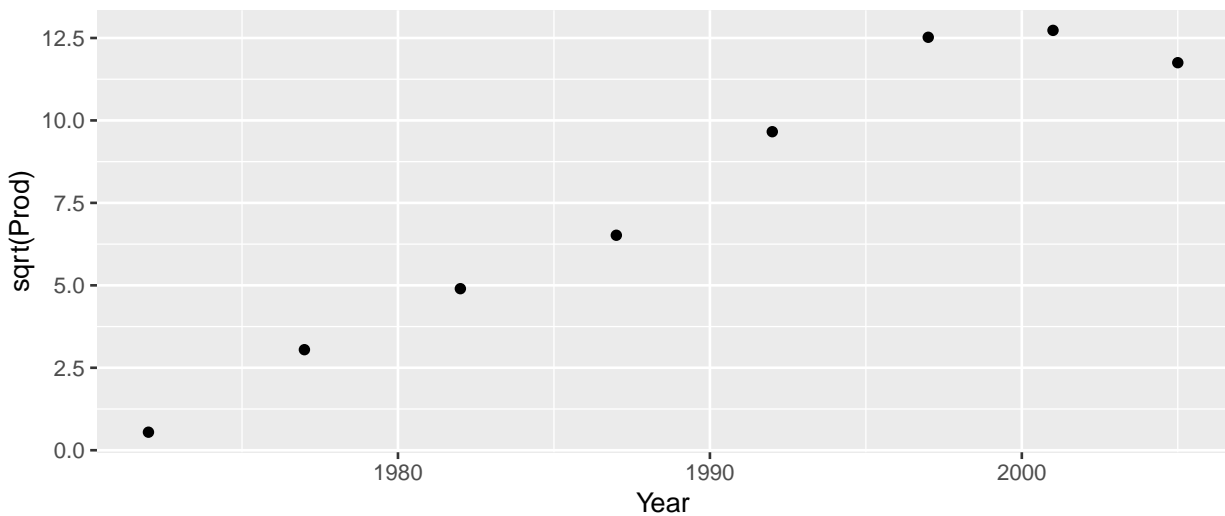
```
oil$Prod
## [1] 0.3 9.3 24.0 42.5 93.3 156.8 162.1 138.1
oil[["Prod"]]
## [1] 0.3 9.3 24.0 42.5 93.3 156.8 162.1 138.1
oil[,2]
## [1] 0.3 9.3 24.0 42.5 93.3 156.8 162.1 138.1
```

The indexing mechanism applies to dataframes as well; but here we use two indices. The first index refers to rows and the second to columns.

```
oil[2:4, c(2,1)]
##   Prod Year
## 2  9.3 1977
## 3 24.0 1982
## 4 42.5 1987
```

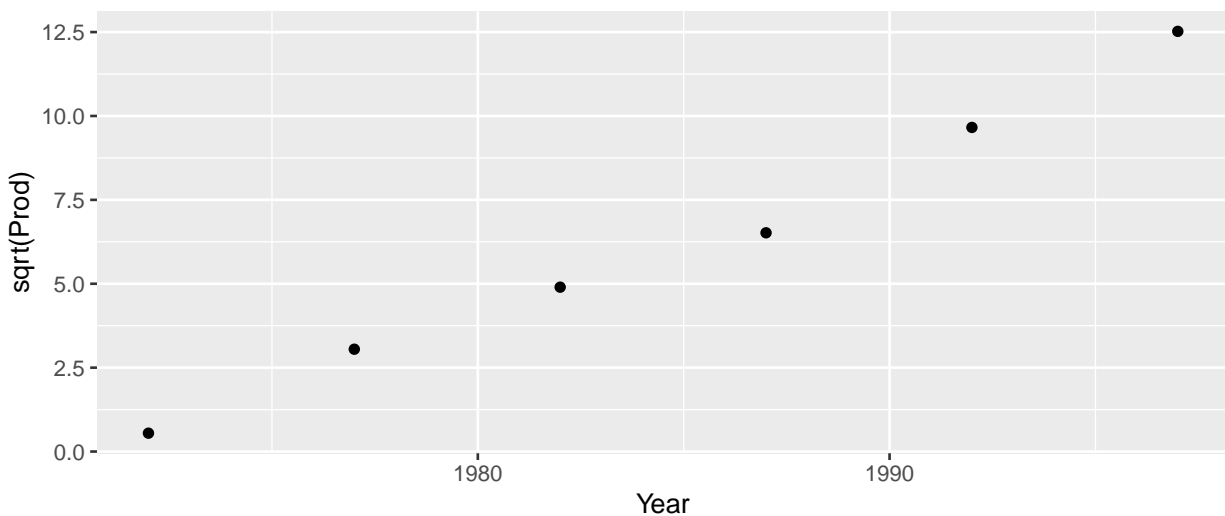
If we do not write anything for the first index then all rows are selected and likewise for columns. Many functions can take a dataframe as input via a `data=` argument; for example

```
qplot(Year, sqrt(Prod), data=oil)
```



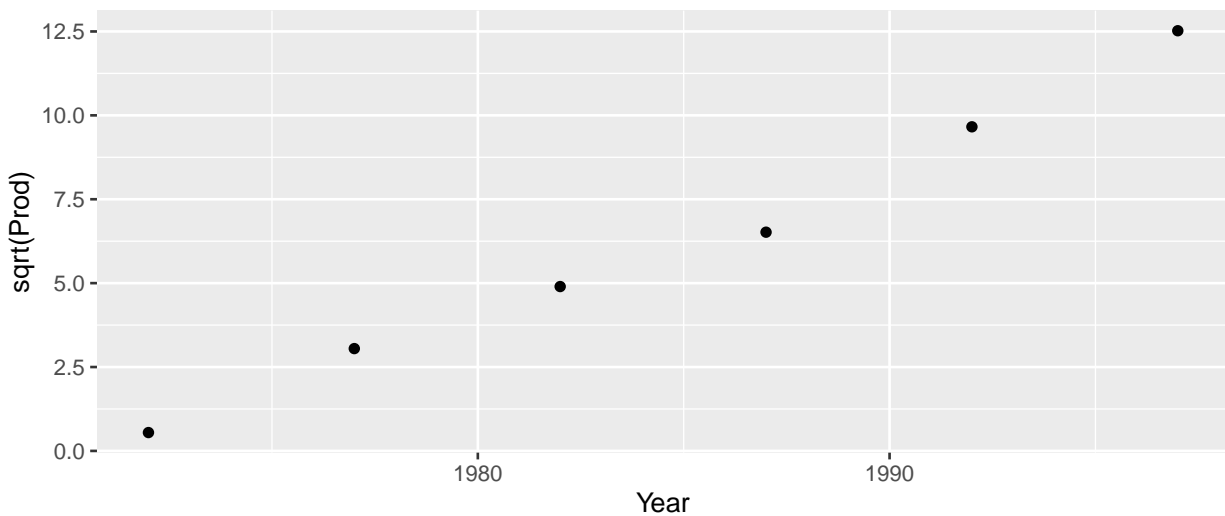
If we want to plot only oil data from the 20th century we can do

```
qplot(Year, sqrt(Prod), data=oil[ oil$Year < 2000, ])
```



However, this can be obtained easier using the `subset()` function

```
qplot(Year, sqrt(Prod), data=subset(oil, subset=Year<2000))
```



## 1.7 Simple (linear) regression

Next we look at a simple dataset: Age and fat percentage in 9 adults:

```
age <- c(23, 28, 38, 44, 50, 53, 57, 59, 60)
fatpct <- c(19.2, 16.6, 32.5, 29.1, 32.8, 42, 32, 34.6, 40.5)
```

Data is shown as dots in Figure ???. By linear regression we find the best approximating straight line, using the method of ORDINARY LEAST SQUARES (OLS). The function in R is `lm()` (short for linear model).

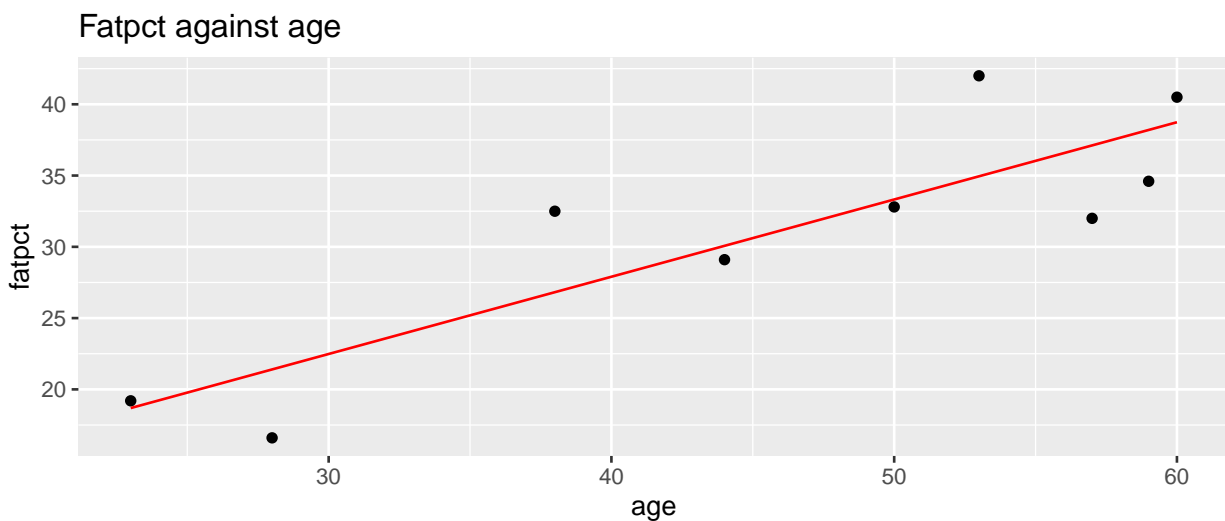
```
reg <- lm(fatpct ~ age)
summary(reg)
##
```

```
## Call:
## lm(formula = fatpct ~ age)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -5.115  -3.599  -0.521   1.759   7.053
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)    6.225     5.711     1.09  0.3118
## age            0.542     0.120     4.51  0.0028 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.61 on 7 degrees of freedom
## Multiple R-squared:  0.744, Adjusted R-squared:  0.707
## F-statistic: 20.3 on 1 and 7 DF, p-value: 0.00277
```

The column Estimate shows the estimated regression coefficients. The interpretation is that when age increases by one year then the fat percentage increases by 0.54.

The plot in Figure ?? shows estimated regression line added on top of the data points. The plot is created as:

```
qplot(age, fatpct) +
  geom_line(aes(age, predict( reg )), color="red" ) +
  labs( title = "Fatpct against age" )
```



## 1.8 Getting help

- Commands in R are really function calls. Example: R can be terminated by the function `q()`.
- Help about a command is obtained with `help()` or `?`. Example: `rnorm()` simulates data from a normal distribution and help is obtained with

```
help( rnorm )
```

- The possible arguments to a function are shown with

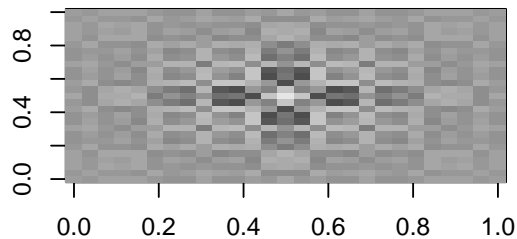
```
args( rnorm )  
## function (n, mean = 0, sd = 1)  
## NULL
```

- The command `help( help )` will give you an overview over the help facilities and `help.start()` will open a web-browser with the help facilities.

Most help pages have an examples-section at the end and these examples are often worthwhile to study for inspiration.

An example is the `image()` function. The examples can be executed directly as follows (where `par(mfrow=c(m,n))` will set up the plot window in  $m$  rows and  $n$  columns.

```
par( mfrow=c(2,2) ) # set up plot window with 2 rows and 2 cols  
example( image )
```

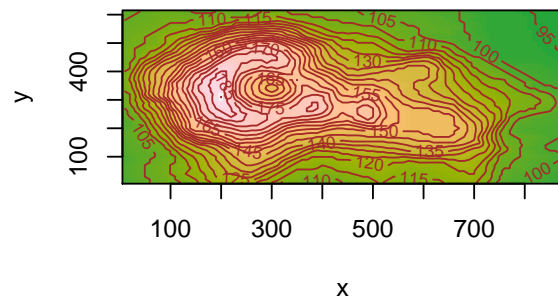
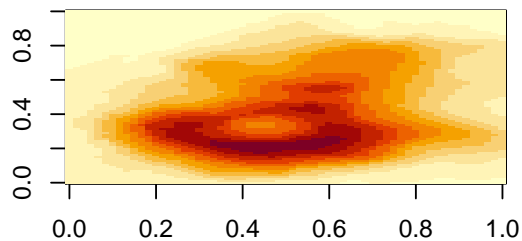


**Math can be beautiful ...**



$$\cos(r^2)e^{-r/6}$$

**Maunga Whau Volcano**



## 1.9 R as a programming language

R is also a programming language, and we shall from time to time write small functions “on the fly” as we need them.

Example: Calculate the sum of the numbers  $1, 2, 3, \dots, N$ . This sum can be computed as  $N(N+1)/2$ , but if we did not know that we could simply do:

```
N <- 100
sum <- 0
for (i in 1:N)
  sum <- sum + i
cat("The sum of the numbers from 1 to", N, "is:", sum, "\n")
## The sum of the numbers from 1 to 100 is: 5050
```

We can create a general purpose function that does the trick:

```
sum.1toN <- function( N ){
  sum <- 0
  for (i in 1:N)
    sum <- sum + i
  cat("The sum of the numbers from 1 to", N, "is:", sum, "\n")
  sum
}
sum.1toN( 1000 )
## The sum of the numbers from 1 to 1000 is: 500500
## [1] 500500
sum.1toN( 10 )
## The sum of the numbers from 1 to 10 is: 55
## [1] 55
```

## 2 A more nerdy section

### 2.1 Generating random and systematic data

In addition to `c()`, vectors can be created in other ways, for example using `:`, `rep()` and `seq()`.

```
1:4
## [1] 1 2 3 4
3:-3
## [1] 3 2 1 0 -1 -2 -3
rep(1:3, times = 2)
## [1] 1 2 3 1 2 3
rep(1:3, each = 2)
## [1] 1 1 2 2 3 3
seq(1, 3, by = 0.5)
## [1] 1.0 1.5 2.0 2.5 3.0
seq(1, 3, length = 4)
## [1] 1.0000 1.6667 2.3333 3.0000
```

Random data can be generated as follows: Samples from 5 independent uniform random variables on  $[0, 10]$  are generated by `runif()`:

```
runif( n = 5, min = 0, max = 10 )
## [1] 0.72685 2.69839 8.48730 4.09890 3.88035
```

Samples from 5 independent normal random variables with mean 1 and standard deviation 2 are generated by `rnorm()`:

```
rnorm( n = 5, mean = 1, sd = 2 )
## [1] 0.81861 3.64313 3.54232 2.99917 0.18640
```

## 2.2 Matrices

Next define a  $3 \times 3$  matrix with the integers from one to nine: @

```
A <- matrix( 1:9, nrow = 3, ncol = 3 ); A;
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
dim( A )
## [1] 3 3
```

The numbers are read column-wise; if this is not what you want, use the optional argument `byrow=TRUE` to `matrix()`. Indexing matrices is similar to indexing vectors except that an index vector defining rows and an index vector defining columns are needed. @

```
A[1:2, c(1,3)]  ## extract submatrix by some rows and cols
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
A[1:2, ]         ## extract submatrix by some rows and all cols
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
A[2, ]          ## no longer a matrix, but a vector
## [1] 2 5 8
A[2, , drop=FALSE]  ## still a matrix
##      [,1] [,2] [,3]
## [1,]    2    5    8
```

## 2.3 Vectors, lists and dataframes

### 2.3.1 Vectors

The basic data structure in R is a vector. Think of a vector as a train wagon in which all passengers are of the same type.

Vectors can be created using the `c()` function (short for concatenate)

```
v1 <- c("here", "comes", "the", "sun") # Character vector
v2 <- c(7, 9, 13)                      # Numeric vector
v3 <- c(T, F, T)                       # Logical vector
v1; v2; v3
## [1] "here" "comes" "the"  "sun"
## [1]  7  9 13
## [1] TRUE FALSE TRUE
```

Elements of vectors can be named (and used for indexing)

```
x <- c(a=123, b=234, c=345); x
##   a    b    c
## 123 234 345
x[ c("a","c") ]
##   a    c
## 123 345
```

Elements are coerced to the least restrictive type:

```
x <- c(1,2,3, "hello world"); x
## [1] "1"          "2"          "3"          "hello world"
```

### 2.3.2 Lists

If a vector is a train wagon in which all passengers have the same type, then a list is a train consisting of a sequence of train wagons. Wagons can be named.

```
x <- list(a=c(2,3), b="hello world", c(T,F)); x
## $a
## [1] 2 3
##
## $b
## [1] "hello world"
##
## [[3]]
## [1] TRUE FALSE
```

Indexing:

```
x[ c(1,2) ] # The train consisting of wagons 1 and 2
## $a
## [1] 2 3
##
## $b
## [1] "hello world"
x[ c("a","b") ]
## $a
## [1] 2 3
##
## $b
## [1] "hello world"
x[ 1 ]      # The train consisting of wagon 1 only
## $a
## [1] 2 3
x[ "a" ]
## $a
## [1] 2 3
x[[ 1 ]]    # Wagon 1
## [1] 2 3
x$a
## [1] 2 3
```



Remove and add wagons:

```
x$a <- NULL; x
## $b
## [1] "hello world"
##
## [[2]]
## [1] TRUE FALSE
x$h <- 2:4; x    # Wagons are added at the end
## $b
## [1] "hello world"
##
## [[2]]
## [1] TRUE FALSE
##
## $h
## [1] 2 3 4
x[5] <- 1000; x  # Empty wagons are added too
## $b
## [1] "hello world"
##
## [[2]]
## [1] TRUE FALSE
##
## $h
## [1] 2 3 4
##
## [[4]]
## NULL
##
## [[5]]
## [1] 1000
```

The train analogy does not carry through all the way: An element of a list can be a list itself (i.e. a wagon can be a train itself):

```
x <- list(a=c(2,3), b="hello world", c=c(T,F), d=list(g=23, 45)); x
## $a
## [1] 2 3
##
## $b
## [1] "hello world"
##
## $c
## [1] TRUE FALSE
##
## $d
## $d$g
## [1] 23
##
## $d[[2]]
## [1] 45
str( x )
## List of 4
```

```
## $ a: num [1:2] 2 3
## $ b: chr "hello world"
## $ c: logi [1:2] TRUE FALSE
## $ d:List of 2
## ..$ g: num 23
## ..$ : num 45
dput( x )
## list(a = c(2, 3), b = "hello world", c = c(TRUE, FALSE), d = list(
##      g = 23, 45))
```

### 2.3.3 Matrices

A matrix can be created with the `matrix()` function.

```
A <- matrix(1:9, nrow=3, ncol=3); A
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
dim(A)
## [1] 3 3
dput(A)
## structure(1:9, .Dim = c(3L, 3L))
```

Extract submatrices by indexing rows and/or columns:

```
A <- matrix(1:9, nrow=3, ncol=3); A
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
A[1:2, c(1,3)]
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
A[1:2,]
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
A[c(T, T, F),]
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
A[2,]      # no longer a matrix, but a vector
## [1] 2 5 8
A[2,,drop=F] # still a matrix
##      [,1] [,2] [,3]
## [1,]    2    5    8
```

### 2.3.4 Dataframes

A dataframe is basically a list in which each element has the same length.

```
d <- data.frame(x=5:8, y=c("here", "comes", "the", "sun"), z=c(T,F,T,T)); d
##   x    y    z
## 1 5  here TRUE
## 2 6 comes FALSE
## 3 7   the  TRUE
## 4 8   sun  TRUE
```

Operate on dataframes as on lists. In addition, subscript as for matrices:

```
d[ c(1,4), c(1,3) ]
##   x    z
## 1 5 TRUE
## 4 8 TRUE
```

## 2.4 Getting data into functions

If we want to plot production against year we can do

```
plot(Prod ~ Year, data=oil)
```

This works because there is a version of the data

A function like `smooth.spline()`, however, does not take data as input:

```
args( smooth.spline )
## function (x, y = NULL, w = NULL, df, spar = NULL, lambda = NULL,
##          cv = FALSE, all.knots = FALSE, nknots = .nknots.smspl, keep.data = TRUE,
##          df.offset = 0, penalty = 1, control.spar = list(), tol = 1e-06 *
##          IQR(x), keep.stuff = FALSE)
## NULL
```

Hence we must provide data in another way. Can be done as

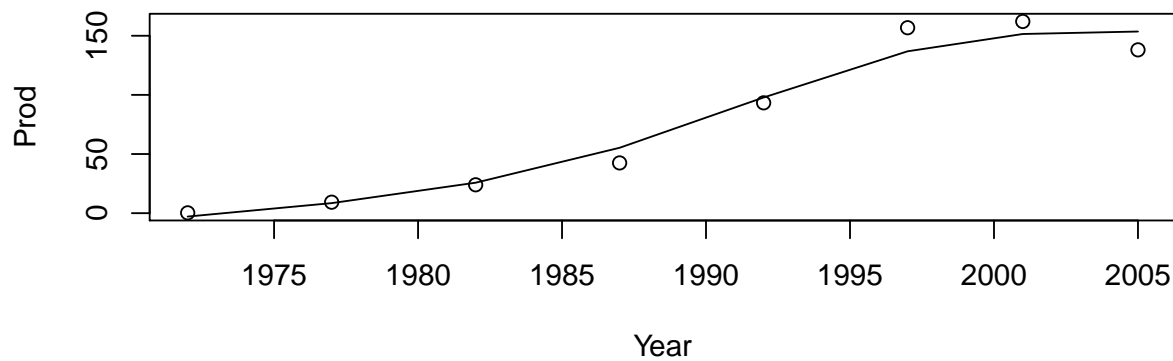
```
sm <- smooth.spline(oil$Year, oil$Prod, df=4)
```

or using `with()`

```
sm <- with( oil, smooth.spline( Year, Prod, df=4 ) )
```

The spline fit looks like:

```
plot(Prod ~ Year, data=oil)
lines(sm)
```



## 3 Programming - R functions

### 3.1 Hello World

Functions in R are like mathematical functions: Rules that maps some input arguments (possibly none, possibly many) to some output (possibly none represented by NULL in R).

```
hello <- function(){ ## define function
  cat("Hello world\n")
}
hello() ## call function
## Hello world
```

```
add <- function(x, y){
  x + y
}
add(2,5)
## [1] 7
```

The syntax for R functions is

```
f <- function(arg1, arg2, arg3, ...){
  ## expr1
  ## expr2
  ## ...
  ## exprn (The last expression is returned)
}
```

### 3.2 More on input and returning values\*

The last expression is returned:

```
add <- function(x, y){
  x + y
}
add(2,5)
## [1] 7
```

Arguments can be given a default value:

```
add <- function(x, y=0){
  x + y
}
add(2)
## [1] 2
## or
add <- function(x, y=x){
  x + y
}
add(7)
## [1] 14
```

To return not just the sum but also  $x$  and  $y$  we can use a list:

```
add <- function(x,y){
  list(x + y, arg=c(x=x, y=y))
}
(z <- add(2, 5))
## [[1]]
## [1] 7
##
## $arg
## x y
## 2 5
```

Attributes can also be used for returning multiple values from function. Attributes can be set with `attr()`:

```
add <- function(x, y){
  ans <- x + y
  attr(ans, "arg") <- c(x, y)
  ans
}
(z <- add(2,5))
## [1] 7
## attr("arg")
## [1] 2 5
```

### 3.3 Conditional and repetitive execution

Conditional execution:

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

Repetitive execution:

```
for (name in expr1) {expr2}
repeat {expr}
while (condition) {expr}
```

Here condition must evaluate to a single logical value.

The break statement can be used to terminate any loop (and it is the only way to terminate a repeat loop).

A collection of expressions can be grouped by curly braces.

### 3.4 Example: Calculating the factorial

The factorial of a positive integer  $n$  is  $n! = n(n-1)(n-2)\dots 2 \cdot 1$ . Since  $n! = n(n-1)!$  a recursive implementation is straight ahead:

@

```
fact <- function( n ){
  if ( n < 0 )
    stop("Error; n must be positive")
  else
    if ( n == 1 ) return( 1 )
  else
    n * fact( n - 1 )
}
fact(5)
## [1] 120
factorial( 5 )
## [1] 120
```

### 3.5 Example: Calculating the square root

Finding the square root of  $x > 0$  is the same as solving

$$f(z) = z^2 - x = 0$$

for  $z$ , which in turn is the same as solving  $z - x/z = 0$ .

The “Babylonian method”:

1. Start with initial guess for  $z$  (any starting value will do; for example  $x$  itself).
2. Iteratively replace  $z$  by the average of  $z$  and  $x/z$ , i.e. set  $z \leftarrow \frac{1}{2}(z + x/z)$ .
3. Stop when  $z$  and  $x/z$  are as close as desired.

This method is really the same as a Newton–Raphson method:

$$z_{n+1} \leftarrow z_n - \frac{f(z_n)}{f'(z_n)} = \frac{1}{2} \left( z_n + \frac{x}{z_n} \right)$$

```
## Using while(){}
mysqrt1 <- function(x, z = x){
  while( abs( z^2 - x ) > 1e-7 ){
```

```

    z <- ( z + x / z ) / 2
  }
  z
}

```

```

c( mysqrt1(.01), sqrt(.01) )
## [1] 0.1 0.1
c( mysqrt1(10), sqrt(10) )
## [1] 3.1623 3.1623
c( mysqrt1(1000), sqrt(1000) )
## [1] 31.623 31.623
c( mysqrt1(10000), sqrt(10000) )
## [1] 100 100

```

```

## Using repeat{}
mysqrt2 <- function(x, z=x){
  repeat{
    z.old <- z
    z <- (z + x/z)/2
    if (abs(z-z.old) < 1e-7)
      break
  }
  z
}

```

```

c(mysqrt2(.01), sqrt(.01))
## [1] 0.1 0.1
c(mysqrt2(10), sqrt(10))
## [1] 3.1623 3.1623
c(mysqrt2(1000), sqrt(1000))
## [1] 31.623 31.623
c(mysqrt2(10000), sqrt(10000))
## [1] 100 100

```

```

## Using for(){}
mysqrt3 <- function(x, z=x){
  for( i in 1:10 ){
    z <- (z + x/z)/2
  }
  z
}

```

```

c(mysqrt3(.01), sqrt(.01))
## [1] 0.1 0.1
c(mysqrt3(10), sqrt(10))
## [1] 3.1623 3.1623
c(mysqrt3(1000), sqrt(1000))
## [1] 31.623 31.623
c(mysqrt3(10000), sqrt(10000))
## [1] 100 100

```

### 3.6 Example: Recursion and iteration

The factorial function  $n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1$  can be defined recursively as  $n! = f(n) = n f(n-1)$  with  $f(1) = 1$ .

```
## Recursive implementation:
fact.rec <- function(n){
  if (n==1) {
    1
  } else {
    n * fact.rec(n-1)
  }
}

## Iterative implementation:
fact.it <- function(n){
  out <- 1
  for (i in 2:n){
    out <- out * i
  }
  out
}
```

The iterative version is much faster than the recursive version.

Benchmarking:

```
library(microbenchmark)
microbenchmark( fact.rec(100), fact.it(100) )
## Unit: microseconds
##      expr      min       lq     mean  median      uq      max neval
## fact.rec(100) 27.241  27.594  52.585  27.7825  28.101 2475.9   100
## fact.it(100)  2.271   2.321  18.743   2.4545   2.491 1633.9   100
## cld
##      a
##      a
```

Comments:

- Recursive version is “conceptually attractive”; iterative version less so.
- Recursive version computationally more expensive because many function calls must be kept “alive” in computers memory.

### 3.7 Exercises – programming

#### 3.7.1 Calculating the (natural) logarithm

Calculating the (natural) logarithm of  $x > 0$  can be done by solving  $f(z) = \exp(z) - x = 0$ . The Newton step becomes

$$z_{n+1} \leftarrow z_n - \frac{f(z_n)}{f'(z_n)} = z_n - 1 + \frac{x}{\exp(z_n)}$$

Implement this in the function `mylog()`. Think about what a good starting value for the iteration could be.

Benchmark your function along these lines:



```
d <- (mylog(1000)-log(1000))/log(1000); d; abs(d) < 1e-8
## [1] 0
## [1] TRUE
d <- (mylog(10)-log(10))/log(10); d; abs(d) < 1e-8
## [1] 0
## [1] TRUE
```

### 3.7.2 Calculating the $p$ th root

Finding the  $p$ th root of  $x > 0$  where  $p$  a positive integer is the same as solving

$$f(z) = z^p - x = 0$$

for  $z$ .

Derive the Newton–Rapson step for solving this equation iteratively and implement a function called `myrootp(x, p)`.

Benchmark your function along these lines:

```
x <- 8; p <- 2
d <- (myrootp(x,p) - x^(1/p))/x^(1/p); d; abs(d) < 1e-8
## [1] -1.5701e-16
## [1] TRUE
x <- 8; p <- 3
d <- (myrootp(x,p) - x^(1/p))/x^(1/p); d; abs(d) < 1e-8
## [1] 0
## [1] TRUE
```

## 4 Computer arithmetic

### 4.1 Computer arithmetic is not exact

The following R statement appears to give the correct answer.

```
0.3 - 0.2
## [1] 0.1
```

However,

```
0.3 - 0.2 == 0.1
## [1] FALSE
```

The difference between the values being compared is small, but important.

```
(0.3 - 0.2) - 0.1
## [1] -2.7756e-17
```

There is no such thing as “real numbers” on a computer. A number on a computer is always given as its decimal representation, and since the amount of memory is finite, there must be finitely many decimals.

A proxy for real numbers is floating point numbers which are represented as

$$x = s \times b^E$$

where  $s$  is the significand (or mantissa),  $b$  is the base and  $E$  is the exponent.

R has “numeric” (floating points) and “integer” numbers

```
class(1)
## [1] "numeric"
class(1L)
## [1] "integer"
```

Mathematical operations on integers are “exact”; mathematical operations on floating point numbers are only approximate.

```
z <- 1.23456789000000000000e-10
print(z, digits = 20)
## [1] 1.23456789000000000696e-10
x <- 1 + z
print(x, digits = 20)
## [1] 1.0000000001234568003
y = x - 1
print(y, digits = 20)
## [1] 1.234568003383174073e-10
```

Mathematically  $z$  and  $y$  should be identical, but they are not.

```
z - y
## [1] -1.1338e-17
```

Numbers in R are accurate to about 16 significant digits.

There are 16 correct digits in  $x$ , but only 6 correct digits in  $y$ .

Subtraction of nearly equal quantities (known as near cancellation or catastrophic cancellation) is a major source of inaccuracy in numerical calculations.

Subtraction of positive values is one place where the finite precision of floating-point arithmetic is a potential problem.

```
x = 1+1.234567890e-10
print(x, digits = 20)
## [1] 1.0000000001234568003
```

```
y = x - 1
print(y, digits = 20)
## [1] 1.234568003383174073e-10
```

Subtraction of nearly equal quantities (known as near cancellation or catastrophic cancellation) is a major source of inaccuracy in numerical calculations.

## 4.2 Addition and subtraction – Significant digits

In  $x = s \times b^E$ , let  $s$  be a 7-digit number

A simple method to add floating-point numbers is to first represent them with the same exponent.

```
123456.7 = 1.234567 * 10^5
101.7654 = 1.017654 * 10^2 = 0.001017654 * 10^5
```

So the true result is

```
(1.234567 + 0.001017654) * 10^5 = 1.235584654 * 10^5
```

But if the computer can only work with 7 significant digits, the approximate result the computer would give is (the last digits (654) are lost)

```
1.235585 * 10^5      (final sum: 123558.5)
```

In extreme cases, the sum of two non-zero numbers may be equal to one of them

Quiz: In which order should one sum a sequence of positive numbers  $x_1, \dots, x_n$  to obtain large accuracy?

### 4.3 Floating point arithmetic

Real numbers “do not exist” in computers. Numbers in computers are represented in floating point form

$$x = s \times b^E$$

where  $s$  is the significand (or mantissa),  $b$  is the base and  $E$  is the exponent.

If  $1 \leq |s| < b$  then  $x$  is said to be normalized.

R has “numeric” (floating points) and “integer” numbers

```
class(1)
## [1] "numeric"
class(1L)
## [1] "integer"
```

Mathematical operations on integers are “exact”; mathematical operations on floating point numbers are only approximate.

### 4.4 The Relative Machine Precision

The accuracy of a floating-point system is measured by the relative machine precision or machine epsilon which is the smallest positive value which can be added to 1 to produce a value different from 1.

A machine epsilon of  $10^{-7}$  indicates that there are roughly 7 decimal digits of precision in the numeric values stored and manipulated by the computer.

It is easy to write a program to determine the relative machine precision

```
.Machine$double.eps
## [1] 2.2204e-16
```

```
machine.eps <- function(){
  eps <- 1
  while(1+eps/2 != 1)
    eps <- eps / 2
  eps
}
machine.eps()
## [1] 2.2204e-16
```

Numbers are accurate to about 15 significant digits.

## 4.5 Floating-Point Precision

The preceding program shows that there are roughly 16 decimal digits of precision to R arithmetic.

It is possible to see the effects of this limited precision directly.

```
a <-  
  12345678901234567890  
print(a, digits=20)  
## [1] 12345678901234567168
```

The effects of finite precision show up in the results of calculations.

## 4.6 Error in Floating-Point Computations

Numbers are accurate to about 15 significant digits.

Subtraction of positive values is one place where the finite precision of floating-point arithmetic is a potential problem.

```
x = 1+1.234567890e-10  
print(x, digits = 20)  
## [1] 1.0000000001234568003
```

```
y = x - 1  
print(y, digits = 20)  
## [1] 1.234568003383174073e-10
```

There are 16 correct digits in  $x$ , but only 6 correct digits in  $y$ .

Subtraction of nearly equal quantities (known as near cancellation or catastrophic cancellation) is a major source of inaccuracy in numerical calculations.

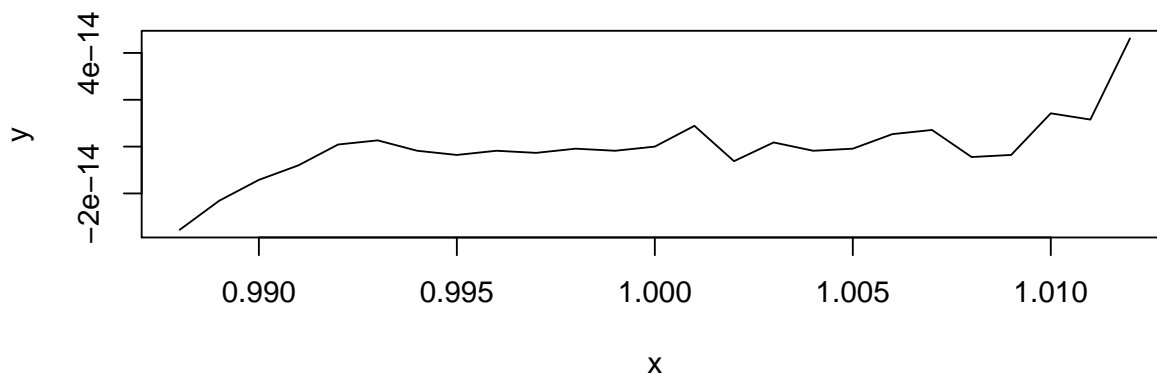
## 4.7 Example: A polynomial

A 7th degree polynomial; its graph should appear very smooth:

$$f(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$$

but: @

```
x = seq(.988, 1.012, by = 0.001)  
y = x^7 - 7*x^6 + 21*x^5 - 35*x^4 + 35*x^3 - 21*x^2 + 7*x - 1  
plot(x, y, type = "l")
```



To see where the cancellation error comes split the polynomial into individual terms and see what happens when we sum them.

@

```
options("digits"=7)
x = .99
y = c(x^7, -7*x^6, 21*x^5, -35*x^4, 35*x^3, -21*x^2, 7*x, -1)
sum(y)
## [1] -1.521006e-14
y
## [1] 0.9320653 -6.5903610 19.9707910 -33.6208604 33.9604650
## [6] -20.5821000 6.9300000 -1.0000000
cumsum(y)
## [1] 9.320653e-01 -5.658296e+00 1.431250e+01 -1.930837e+01
## [5] 1.465210e+01 -5.930000e+00 1.000000e+00 -1.521006e-14
```

It is the last subtraction (of 1) which causes the catastrophic cancellation and loss of accuracy.

We can reformulate the problem by noticing that

$$f(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 = (x-1)^7$$

Notice that although we are still getting cancellation, when 1 is subtracted from values close to 1, we are only losing a few digits of accuracy:

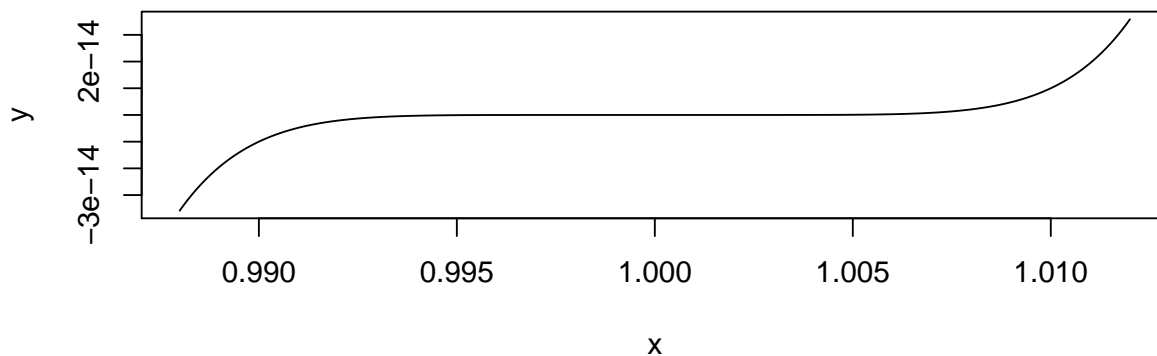
```
(x-1)^7
## [1] -1e-14
```

The relative error is remarkable:

```
(sum(y) - (x - 1)^7) / (x - 1)^7
## [1] 0.5210055
```

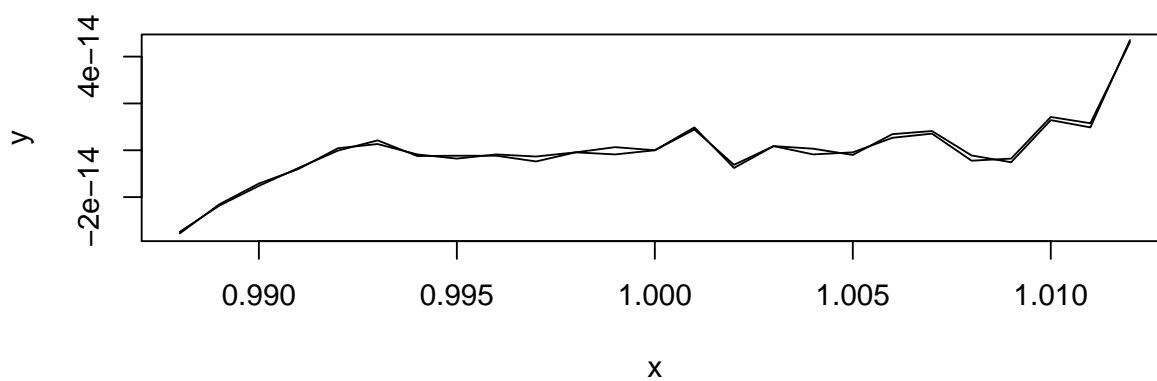
The difference is apparent in the plot.

```
x = seq(.988, 1.012, by = 0.0001)
y = (x - 1)^7
plot(x, y, type = "l")
```



Another detail: Because of numerical cancellations, the order of addition matters:

```
x = seq(.988, 1.012, by = 0.0001)
y = x^7 - 7*x^6 + 21*x^5 - 35*x^4 + 35*x^3 - 21*x^2 + 7*x - 1
y2 = -1 + 7*x - 21*x^2 + 35*x^3 - 35*x^4 + 21*x^5 - 7*x^6 + x^7
plot(x, y, type = "l")
lines(x, y2)
```



## 4.8 Example: Numerical derivatives

The derivative  $f'(x)$  may be approximated by

$$f'(x) \approx \frac{f(x + h/2) - f(x - h/2)}{h}, \quad h \text{ small}$$

A generic R function is (notice that we can pass functions as arguments to functions) @

```
numDeriv <- function(f, x, h=1e-8){  
  (f(x + h / 2) - f(x - h / 2)) / h  
}
```

For small  $h$  we get near cancellation errors.

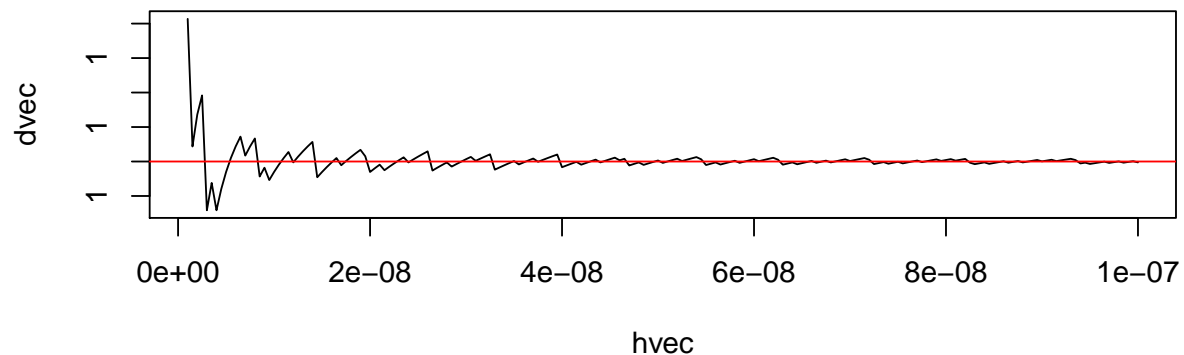
Find derivative of exponential at  $x = 0$ :

```
print(numDeriv(exp, 0), digits=20)  
## [1] 0.99999999392252902908  
print(exp(0), digits=20)  
## [1] 1
```

Try range of  $h$  values. Too small values of  $h$  gives numerical imprecisions.

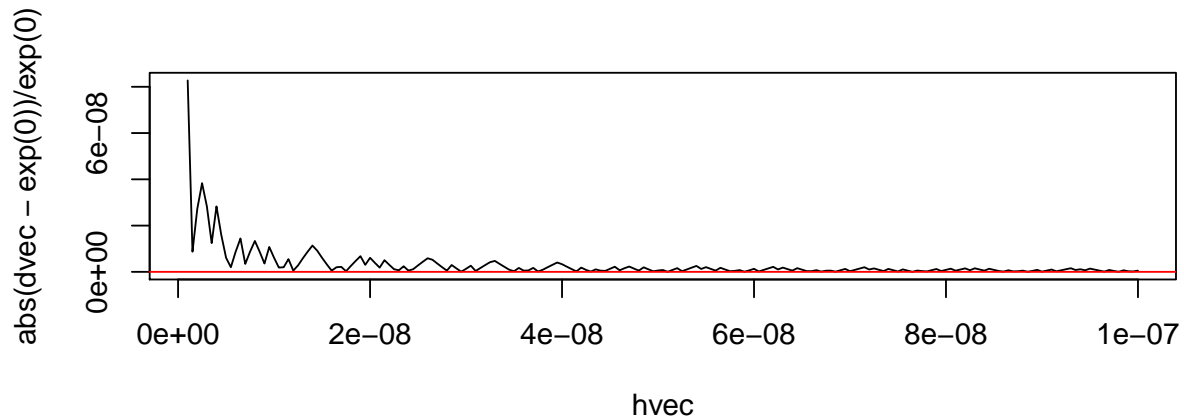
```
hvec <- seq(1e-9, 1e-7, 5e-10)  
dvec <- numDeriv(exp, 0, hvec)
```

```
plot(hvec, dvec, type="l");  
abline(h=1, col='red')
```



@

```
plot(hvec, abs(dvec-exp(0))/exp(0), type='l')  
abline(h=0, col='red')
```



#### 4.9 Example: Solving a quadratic equation\*

Consider solving

$$ax^2 + bx + c = 0$$

Letting  $D = b^2 - 4ac$ , the roots are

$$r_1 = \frac{-b + \sqrt{D}}{2a}, \quad r_2 = \frac{-b - \sqrt{D}}{2a}$$

If  $b^2 \gg ac$  then  $\sqrt{D} = \sqrt{b^2 - 4ac} \approx |b|$ .

In this case, if  $b > 0$  then  $-b + \sqrt{D}$  involves a near cancellation (same for  $-b - \sqrt{D}$  if  $b < 0$ ).

Solution: Rewrite the problem. Multiply numerator and denominator of  $r_1$  by  $-b + \sqrt{D}$  (and of  $r_2$  by  $-b + \sqrt{D}$ ) by to obtain

$$r_1 = \frac{2c}{-b - \sqrt{D}}, \quad r_2 = \frac{2c}{-b + \sqrt{D}}$$

Example: Consider  $f(x) = K(x - r_1)(x - r_2) = ax^2 + bx + c = 0$ . Multiplying out gives: @

```
KK <- 10000
r1 <- 1717; r2 <- 1.234e-12 ## the roots
aa <- KK
bb <- -KK*(r1+r2)
cc <- KK*r1*r2
c(a=aa, b=bb, c=cc)
##           a           b           c
## 1.000000e+04 -1.717000e+07 2.118778e-05
DD <- bb^2 - 4*aa*cc
## Near cancellation errors are likely
sqrt(DD)
## [1] 17170000
-bb+sqrt(DD) ## OK
## [1] 34340000
-bb-sqrt(DD) ## Problematic
## [1] 2.607703e-08
```



@

```
## First solution
##
(x1 <- (-bb+sqrt(DD))/(2*aa)) ## OK
## [1] 1717
(x2 <- (-bb-sqrt(DD))/(2*aa)) ## Problematic
## [1] 1.303852e-12
c( (x1-r1)/r1, c(x2-r2)/x2 )
## [1] 0.00000000 0.05357328
## Second solution
##
(x1 <- 2*cc/(-bb-sqrt(DD))) ## Problematic
## [1] 1625.015
(x2 <- 2*cc/(-bb+sqrt(DD))) ## OK
## [1] 1.234e-12
c( (x1-r1)/r1, c(x2-r2)/r2 )
## [1] -0.05357328 0.00000000
```

## 4.10 Example: Computing the exponential

Recall the exponential function

$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{3 \cdot 2} + \frac{x^4}{4 \cdot 3 \cdot 2} + \dots$$

Letting  $t_n = \frac{x^n}{n!}$  we have  $t_0 = 1$ , and  $t_{n+1} = t_n \frac{x}{n+1}$  so these terms must eventually become small. Hence we may stop the computations when nothing significant is added to the result.

```
myexp <- function(x){
  n <- 0; t <- 1; ans <- 1
  while(abs( t ) > .Machine$double.eps) {
    n = n + 1
    t = t * x / n
    ans <- ans + t
  }
  ans
}
```

Compare myexp() with R's built in function. For positive values, the results are good:

```
(myexp(1) - exp(1))/exp(1)
## [1] 1.633713e-16
(myexp(20) - exp(20))/exp(20)
## [1] -1.228543e-16
```

For negative values less so:

```
(myexp(-1) - exp(-1))/exp(-1)
## [1] 3.017899e-16
(myexp(-20) - exp(-20))/exp(-20)
## [1] 1.727543
```

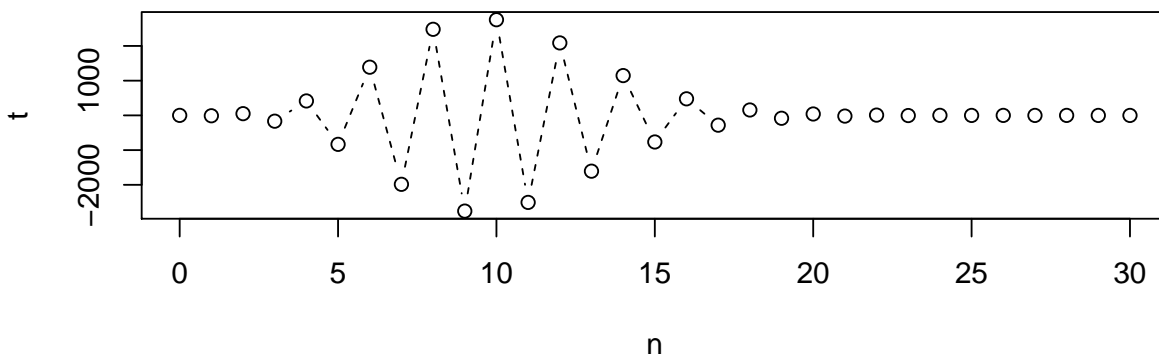
Why?

Look at the terms  $t_n = \frac{x^n}{n!}$  in

$$\exp(x) = \sum_{n=0}^n \frac{x^n}{n!}$$

for  $x < 0$ .

```
x <- -10
n <- 0:30
t <- x^n/factorial(n)
plot(n, t, type='b', lty=2)
```



For numerical large but negative  $x$  values,  $\exp(x)$  is small while at least some of the terms are large.

Hence, at some point, there has to be near-cancellation of the accumulated sum and the next term of the series. This is the source of the large relative error.

Notice: When the argument to `myexp()` is positive, all the terms of the series are positive and there is no cancellation.

There is an easy remedy:

Since  $\exp(-x) = 1/\exp(x)$  it is for negative  $x$  better to compute the result as  $\exp(x) = 1/\exp(-x)$

```
myexp2 <- function(x){
  if (x<0) {
    1/myexp(-x)
  } else {
    myexp(x)
  }
}
```

```
(myexp2(-1) - exp(-1))/exp(-1)
## [1] -1.50895e-16
(myexp2(-20) - exp(-20))/exp(-20)
## [1] 2.006596e-16
```

## 4.11 EXERCISES: Programming

If you are a newbie to R-programming:

1. Write functions calculating the factorial and exponential using the `for`, `repeat` and `while` constructs.
2. Using the `rbenchmark` package, compare your own functions with R's built-in functions.

## 4.12 Debugging R code

R has various facilities for debugging programs.

Some functions to investigate are

- `debug()`,
- `browser()`,
- `trace()`,
- `traceback()`
- `mtrace()` in the `debug` package.

# 5 Graphics in R

Several graphics facilities in R:

- R's own built-in graphics functions
- The **ggplot2** package

Very elegant graphics can be produced very easily using **ggplot2**.

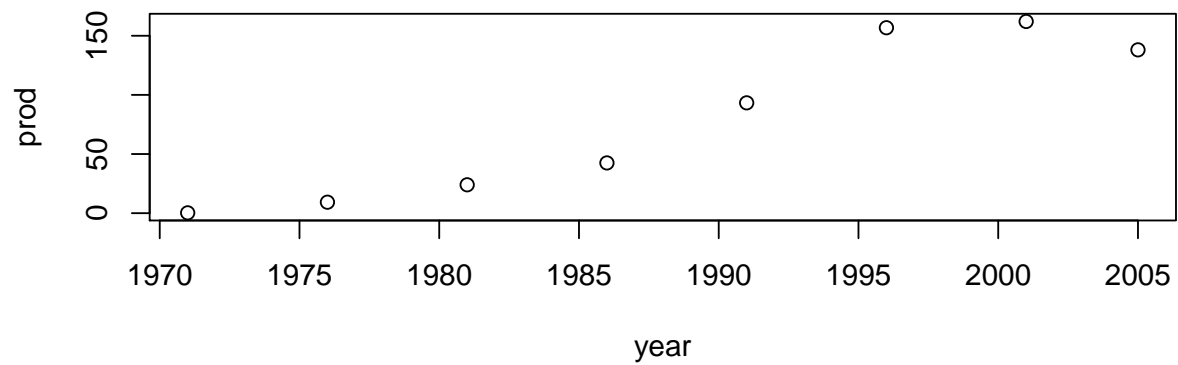
Nonetheless it is very helpful to know the basics of creating graphics using R's own built-in graphics functions.

## 5.1 Scatterplot using `plot()`

Example: Production of oil in Norway (in mio. tons):

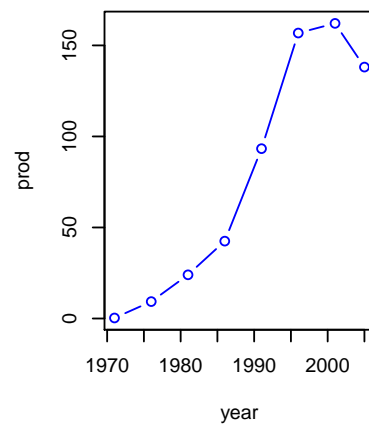
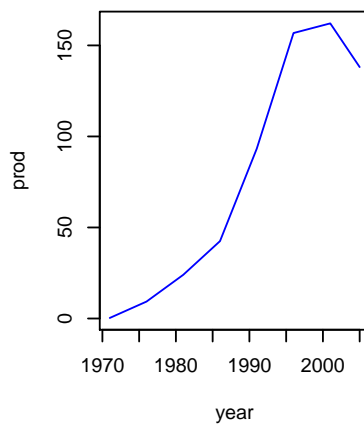
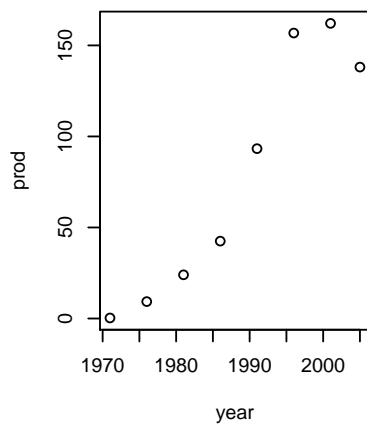
```
year <- c(1971, 1976, 1981, 1986, 1991, 1996, 2001, 2005)
prod <- c(.3, 9.3, 24.0, 42.5, 93.3, 156.8, 162.1, 138.1)
```

```
plot( year, prod )
```



```
## plot( prod ~ year )           ## alternative
## plot( list(x=year, y=prod) ) ## alternative
```

```
par(mfrow=c(1,3))
plot( year, prod, type="p" ) # type="p" (for points) is default
plot( year, prod, type="l", col="blue" )
plot( year, prod, type="b", col=4 )
```



## 5.2 The Puromycin data

The Puromycin data give the reaction velocity versus substrate concentration in an enzymatic reaction involving untreated cells or cells treated with Puromycin.

```
head( Puromycin )
##   conc rate  state
## 1 0.02  76 treated
```

```
## 2 0.02 47 treated
## 3 0.06 97 treated
## 4 0.06 107 treated
## 5 0.11 123 treated
## 6 0.11 139 treated
dim( Puromycin )
## [1] 23 3
```

```
summary( Puromycin )
##      conc      rate      state
##  Min.   :0.0200   Min.   : 47.0   treated  :12
## 1st Qu.:0.0600   1st Qu.: 91.5   untreated:11
##  Median:0.1100   Median :124.0
##   Mean  :0.3122   Mean   :126.8
## 3rd Qu.:0.5600   3rd Qu.:158.5
##   Max.  :1.1000   Max.   :207.0
```

For later use, generate two subsets of data.

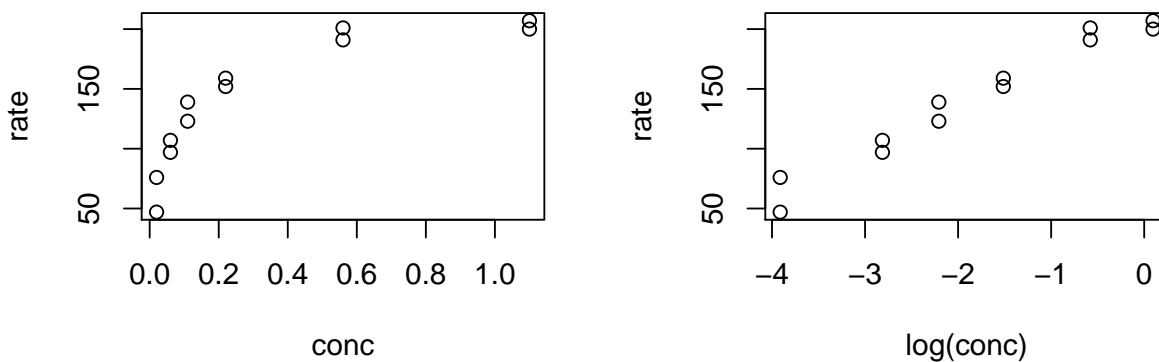
```
puroT <- subset( Puromycin, state=='treated' )
puroU <- subset( Puromycin, state=='untreated' )
```

### 5.3 Simple scatterplots – side by side

We start with a simple scatterplot:

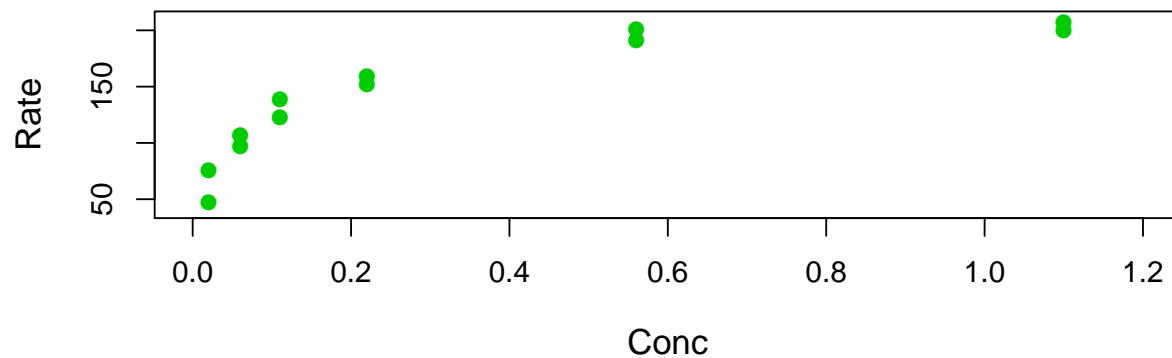
Using the `par()`-function it is possible to place several plots on one page.

```
par( mfrow = c(1,2) ) # one row and two cols of plots
plot( rate ~ conc, data = puroT )
plot( rate ~ log( conc ), data = puroT )
```



### 5.4 Annotating the plot

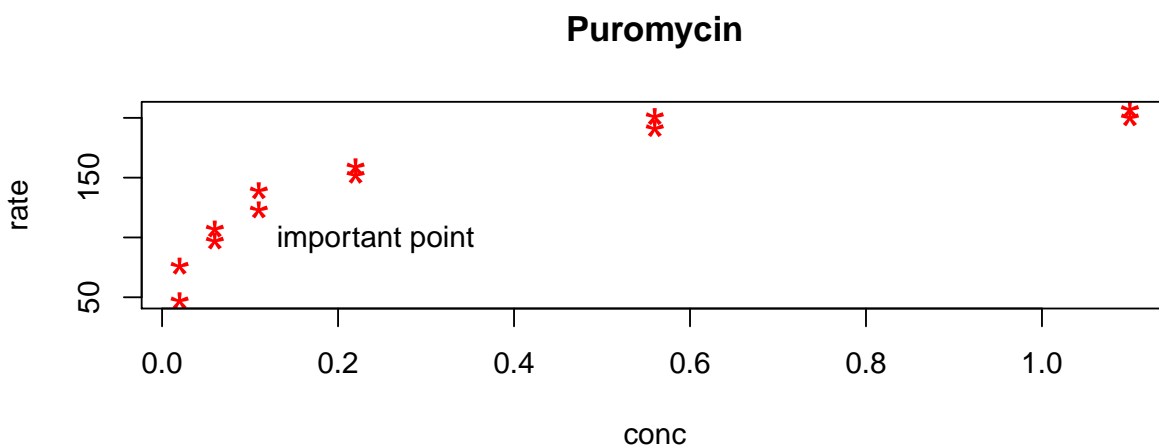
```
plot(rate ~ conc, data = puroT,
     xlim = c(0,1.2), ylim = c(40,210), ## Axis ranges
     xlab = 'Conc', ylab = 'Rate',      ## Axis labels
     cex.lab=1.2,                       ## Axis label scaling
     pch = 16,                          ## Plotting character
     col = 3,                           ## Colour
     cex = 1.2                           ## Plot character scaling
)
```



## 5.5 Adding a title and some text

A title for the plot is added using the `title()` function.

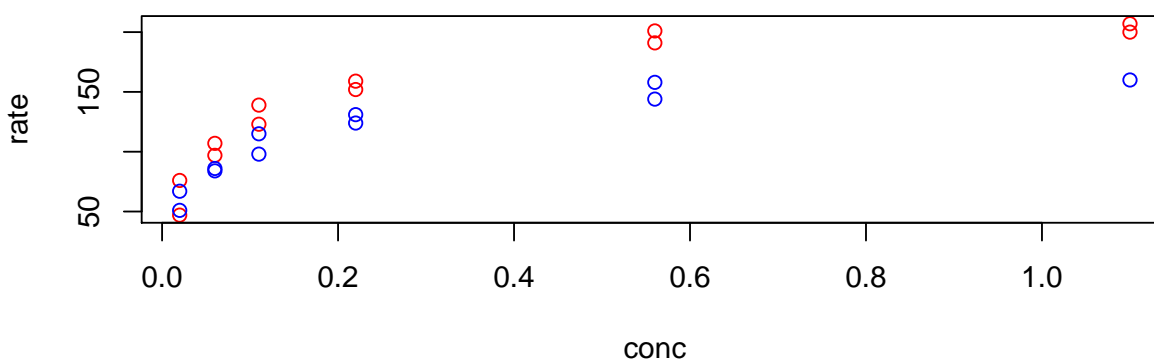
```
plot(rate ~ conc, data=puroT, pch='*', col='red', cex=2)
title(main='Puromycin', cex.main=1.2)
# annotate the observation with conc=0.11 and rate=98.
text(0.11, 98, label = 'important point',
     pos = 4) ## Put the text to the right of the coordinates
```



## 5.6 Overlaying two scatterplots

Plot observations of both levels of state in one plot with different colours for the different state variables. There are 12 rows with `state=treated` followed by 11 rows with `state=untreated`.

```
table( Puromycin$state )
##
##   treated untreated
##      12      11
col.vec <- c( rep(2, 12), rep(4, 11) ); col.vec
## [1] 2 2 2 2 2 2 2 2 2 2 2 2 4 4 4 4 4 4 4 4 4 4
plot(rate ~ conc, data = Puromycin, col = col.vec)
```



A more general (and elegant approach) is as follows:

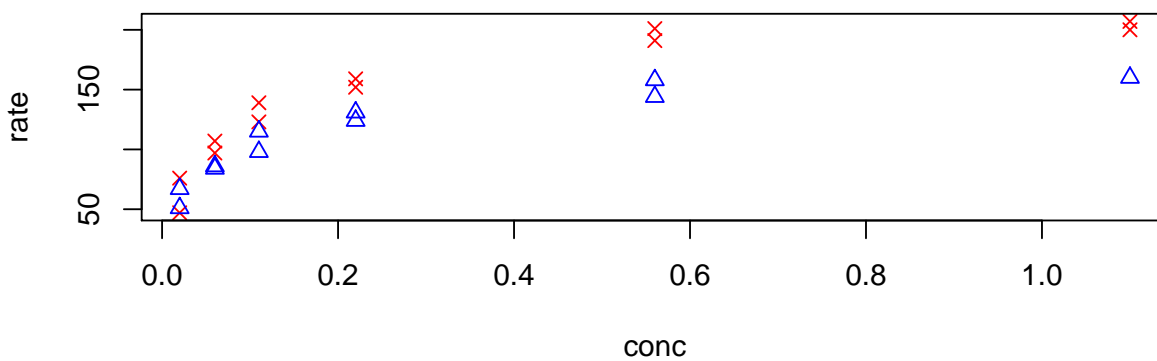
Notice: `state` is a factor (essentially a vector of integers):

```
head(Puromycin$state)
## [1] treated treated treated treated treated treated
## Levels: treated untreated
str(Puromycin$state)
## Factor w/ 2 levels "treated","untreated": 1 1 1 1 1 1 1 1 1 1 1 ...
```

We can create vectors defining colours etc. as:

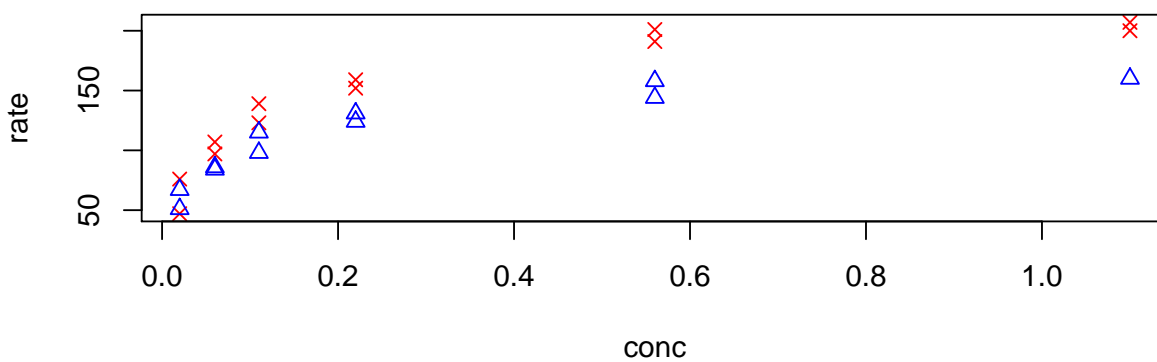
```
pch.vec <- c(4, 2)[ Puromycin$state ]
col.vec <- c(2, 4)[ Puromycin$state ]
```

```
plot(rate ~ conc, data = Puromycin,
     pch = pch.vec,
     col = col.vec)
```



Alternatively we can create these vectors controlling colour, plotting character etc. on the fly:

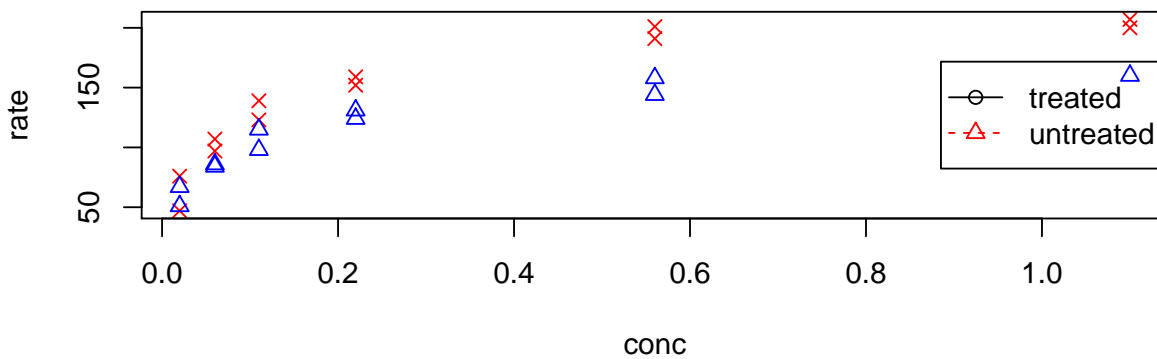
```
plot(rate ~ conc, data = Puromycin,
     pch = c(4,2)[state],
     col = c(2,4)[state])
```



## 5.7 Adding legends

```
plot(rate ~ conc, data = Puromycin,
     pch = pch.vec,
     col = col.vec)
## add legends
legend('right', 'bottom',
      legend=c("treated", "untreated"),
      col=1:2, pch=1:2, lty=1:2)
```

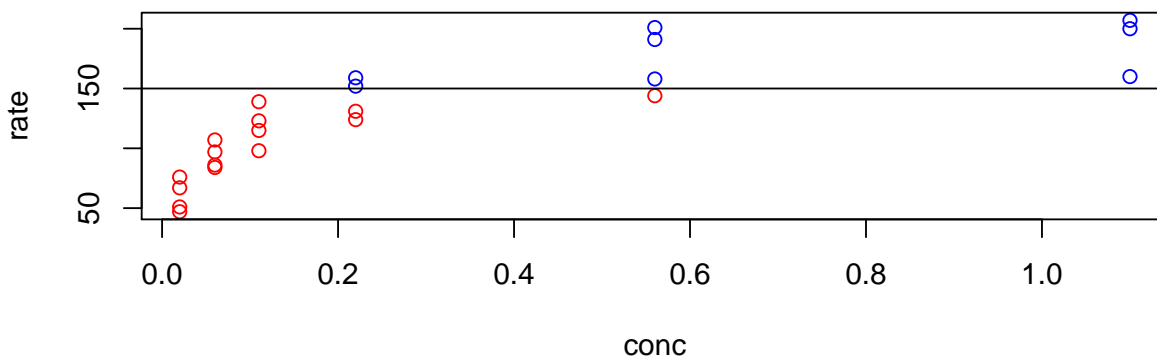




## 5.8 Conditional annotation

A common situation is that e.g. the colour of the points must depend on the value of some variable. For example, the colour must be blue when the rate is larger than 150 and red otherwise.

```
col.vec <- c("red", "blue")[1+(Puromycin$rate>150)]
plot(rate ~ conc, data = Puromycin, col=col.vec)
abline(h=150)
```



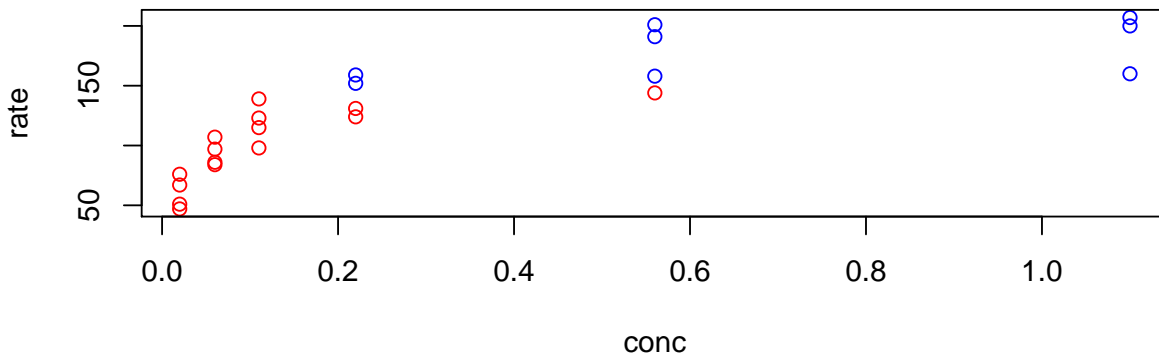
This works because

```
Puromycin$rate > 150
## [1] FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE
## [11] TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [21] FALSE TRUE TRUE
as.numeric( Puromycin$rate > 150 )
## [1] 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1
c("red", "blue")[1+(Puromycin$rate>150)]
```

```
## [1] "red" "red" "red" "red" "red" "red" "blue" "blue"
## [9] "blue" "blue" "blue" "blue" "red" "red" "red" "red"
## [17] "red" "red" "red" "red" "red" "blue" "blue"
```

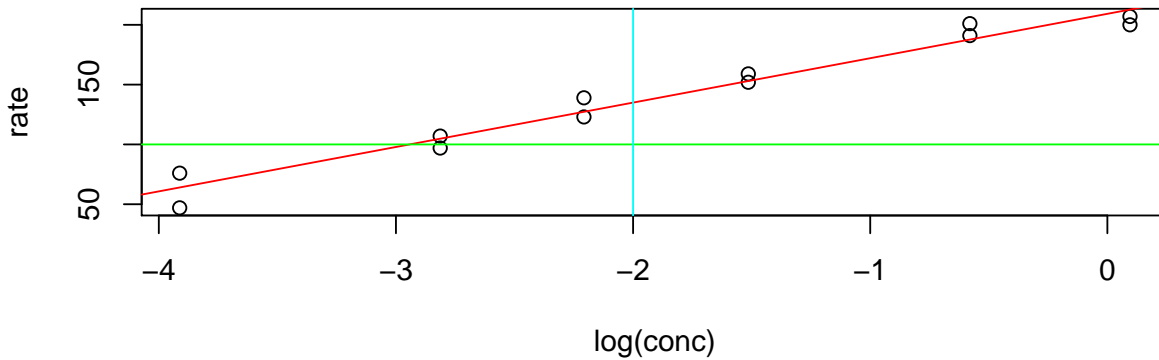
We can do the same as a one-liner:

```
plot(rate ~ conc, data = Puromycin, col=c(2,4)[1+(rate>150)])
```



## 5.9 Add a reference line – `abline()`

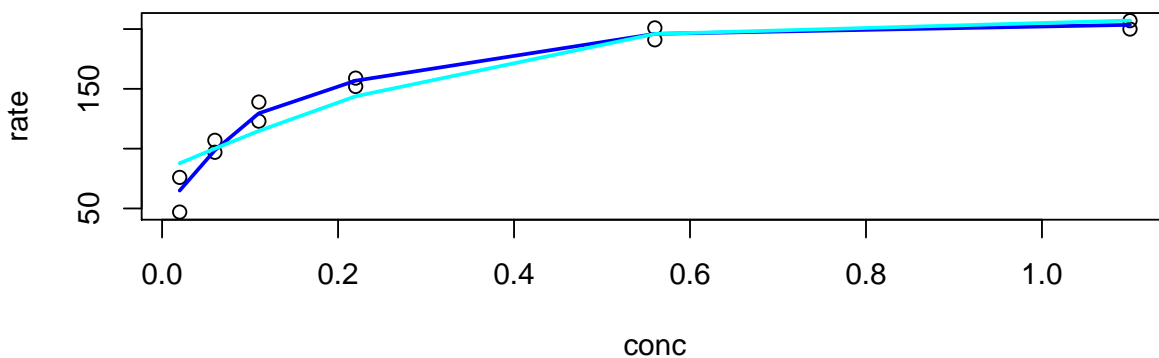
```
m <- lm( rate ~ log(conc), data = puroT )
coef(m)
## (Intercept) log(conc)
## 209.19449 37.11044
plot( rate ~ log(conc), data = puroT )
abline( m, col="red" )
# abline(a=209.19, b=37.11) # equivalent
abline( h=100, col="green" )
abline( v=-2, col="cyan" )
```



## 5.10 Connecting points and adding a smooth line

R provides several functions to compute smooth lines. Here we use smoothing splines available in `smooth.spline()`

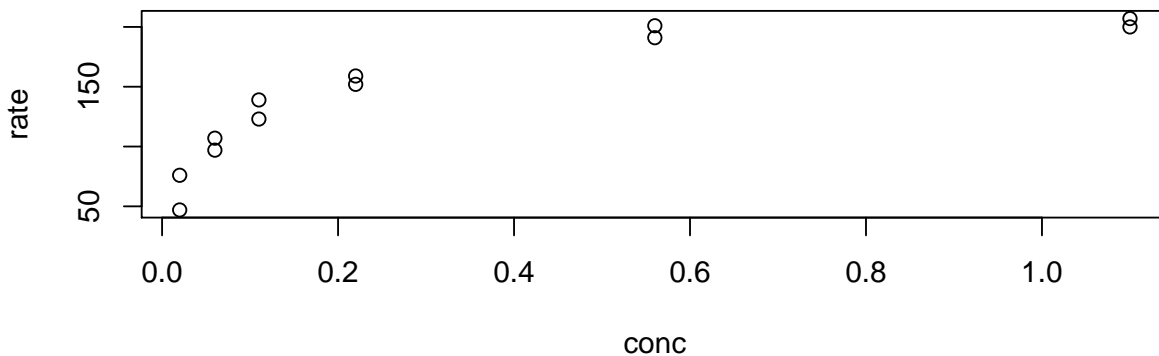
```
plot( rate ~ conc, data = puroT)
# df controls smoothness
sm5 <- smooth.spline( puroT$rate~puroT$conc, df=5 )
sm3 <- smooth.spline( puroT$rate~puroT$conc, df=3 )
lines( sm5, col='blue', lwd=2 )
lines( sm3, col='cyan', lwd=2 )
```



## 5.11 A small detour: Different ways of accessing data

We have seen that the `plot()` function can be given a dataframe containing the data to be plotted as input:

```
plot( rate ~ conc, data = puroT)
```



A function like `smooth.spline()`, however, does not take data as input:

```
args( smooth.spline )
## function (x, y = NULL, w = NULL, df, spar = NULL, lambda = NULL,
##      cv = FALSE, all.knots = FALSE, nknots = .nknots.smspl, keep.data = TRUE,
##      df.offset = 0, penalty = 1, control.spar = list(), tol = 1e-06 *
##      IQR(x), keep.stuff = FALSE)
## NULL
```

Hence we must provide data in another way. Can be done as

```
sm5 <- smooth.spline( puroT$rate~puroT$conc, df=5 )
```

or using `with()`

```
sm5 <- with( puroT, smooth.spline( rate~conc, df=5 ) )
```

Yet another way is via `attach()` and `detach()`, but here you must be careful:

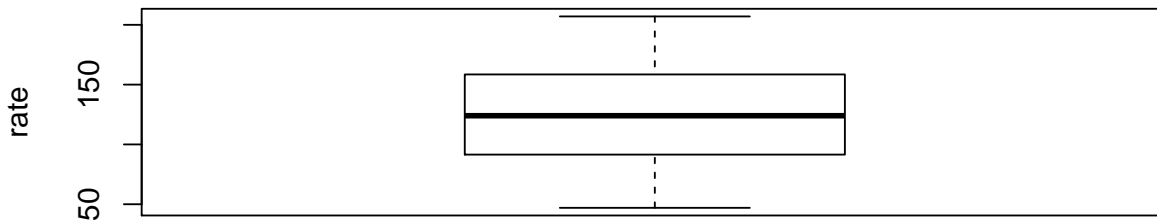
```
attach(Puromycin)
sm5 <- smooth.spline( rate~conc, df=5 )
detach(Puromycin)
```

## 5.12 Some special plots

### 5.12.1 Boxplot – `boxplot()`

Box and whisker plots showing certain quantiles of a variable can be created using the `boxplot`-function.

```
with(Puromycin, boxplot(rate, ylab='rate'))
```

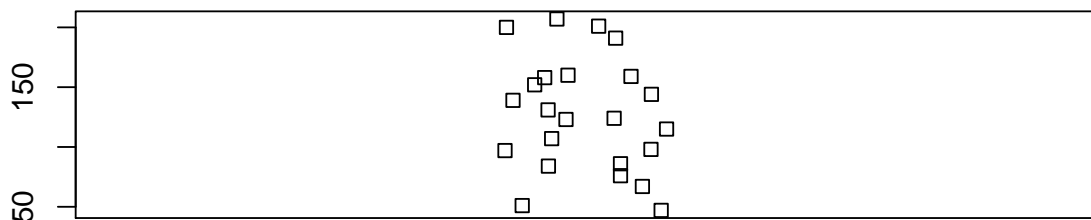


### 5.12.2 Dotplot – `stripchart()`

If there are only few observations in each group, the boxplot is not reasonable. One should plot each point.

To avoid overplotting of points use the `jitter`-method. The value in the argument `jitter` determines how far the points are spread apart.

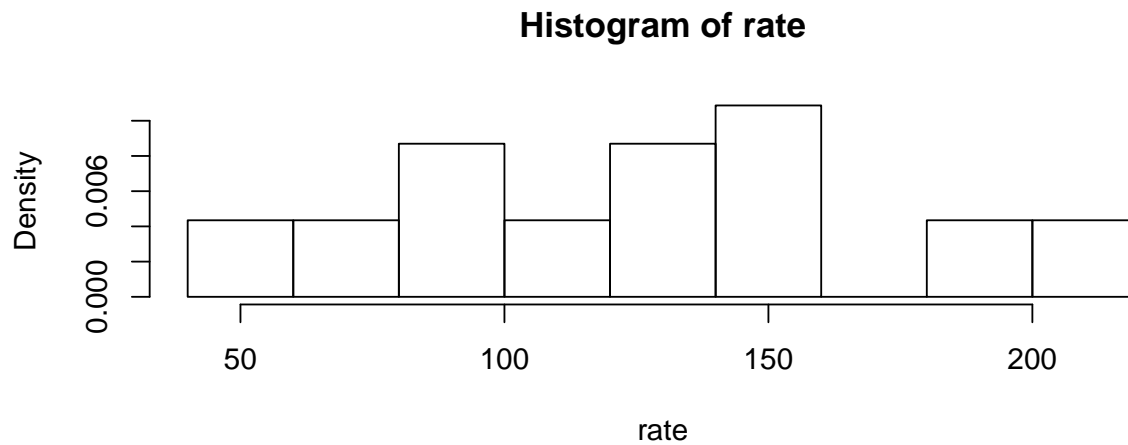
```
with(Puromycin, stripchart(rate, vertical=T,
                           method="jitter", jitter=0.05))
```



### 5.12.3 Histogram and density – `hist()`, `density()`

Graphical representation of the frequency distribution of data.

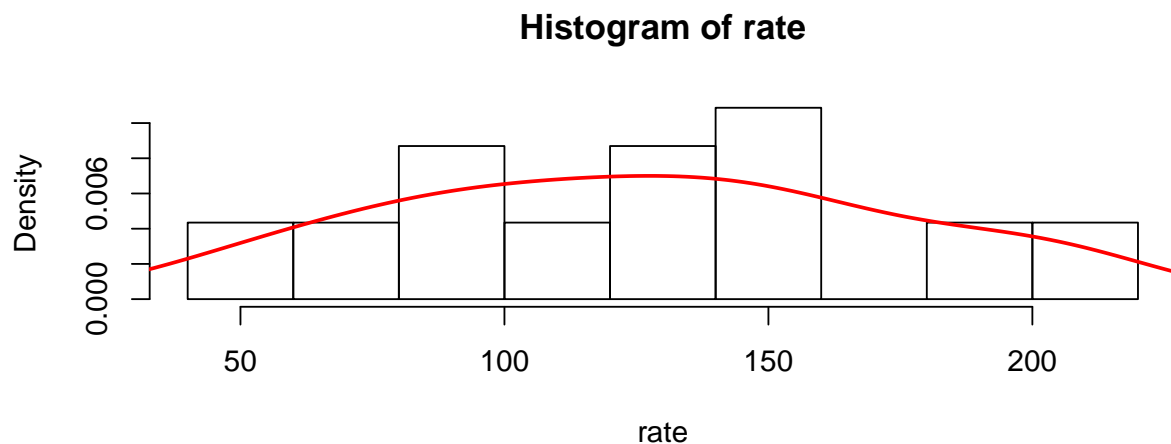
```
with(Puromycin, hist(rate, probability=TRUE))
```



Using `probability=TRUE` the y-axis will denote density instead of number of counts.

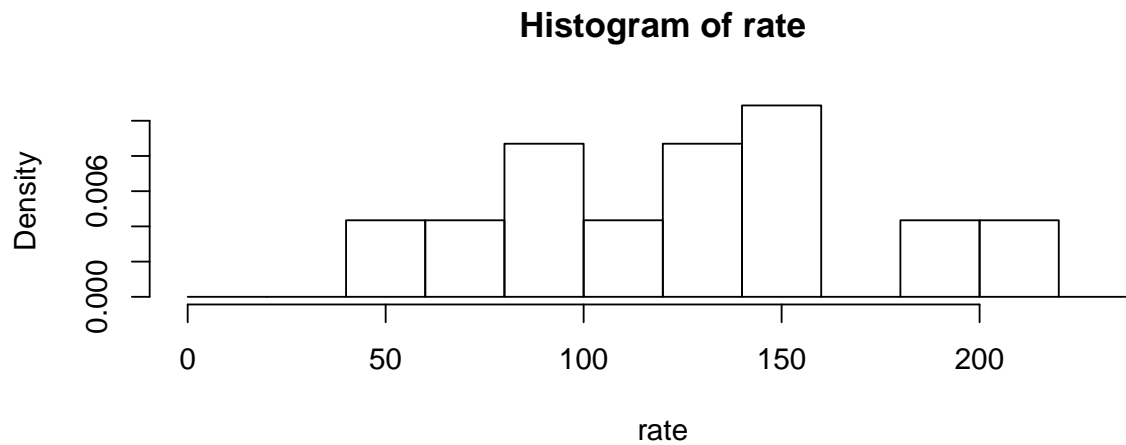
You can add to the histogram a smooth density estimate with the `density()` function. The density-based estimate is not dependent on the choice of the breaks as the histogram

```
with( Puromycin, hist( rate, probability=TRUE ) )
with( Puromycin, lines( density( rate ), col=2, lwd=2 ) )
```



Using the `breaks`-argument it is possible to control the intervals used for counting observations.

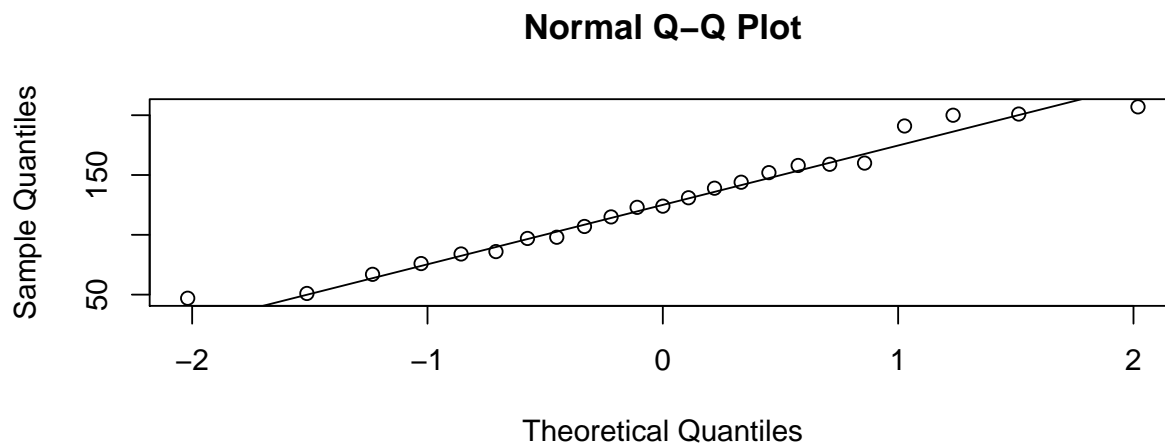
```
with( Puromycin, hist( rate, breaks=seq(0, 250, by=20),
                        probability=TRUE) )
```



#### 5.12.4 Q-Q plot – `qqnorm()`, `qqline()`

A Q-Q-plot compares the quantiles of the data to those of a standard normal distribution. The Q-Q-plot and a reference line are produced by `qqnorm()`-function and `qqline()` function.

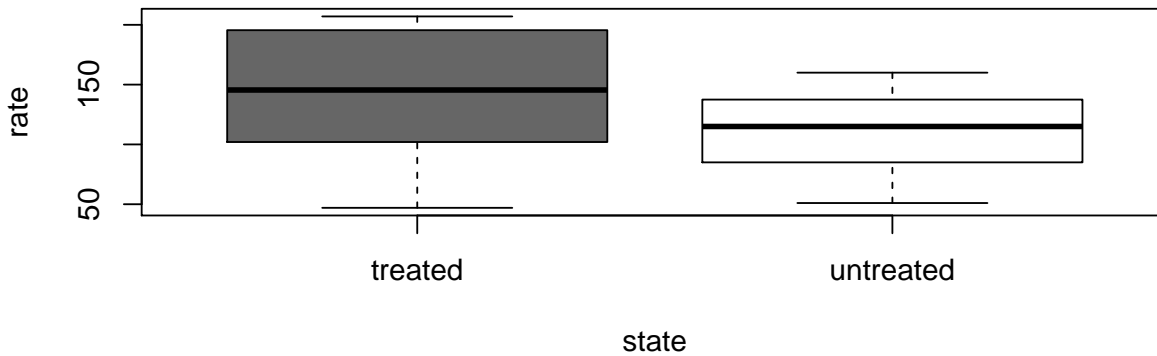
```
with( Puromycin, qqnorm( rate ))
with( Puromycin, qqline( rate ))
```



#### 5.12.5 Grouped data

It is possible to get more than one boxplot in one figure. To get a boxplot for each level of state one uses the formula-based version of `boxplot`

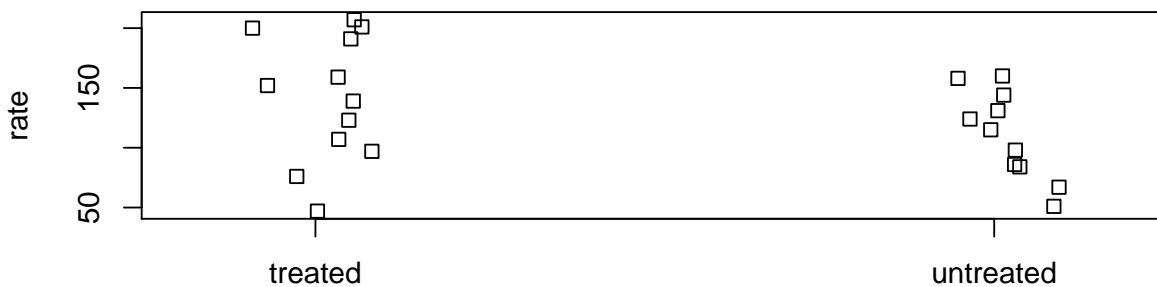
```
boxplot( rate ~ state, data=Puromycin,
        col=grey(c(0.4, 1)))
```



If there are only few observations in each group, the boxplot is not reasonable. One should plot each point.

To avoid overplotting of points use the `jitter`-method. The value in the argument `jitter` determines how far the points are spread apart.

```
stripchart(rate~state, data=Puromycin,
           method="jitter", jitter=0.1, vertical=T)
```

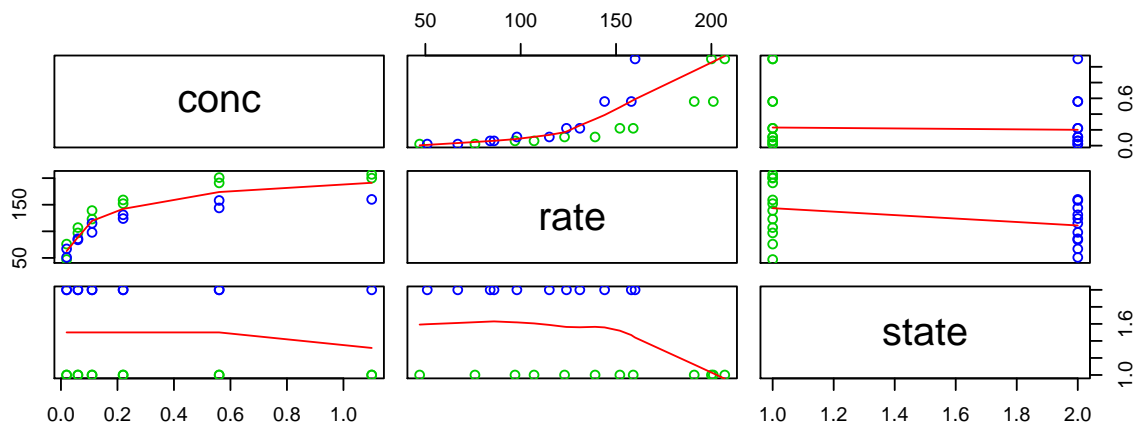


### 5.12.6 Scatterplotmatrix – `pairs()`

A quick overview over all pairwise relations between the variables is obtained with:

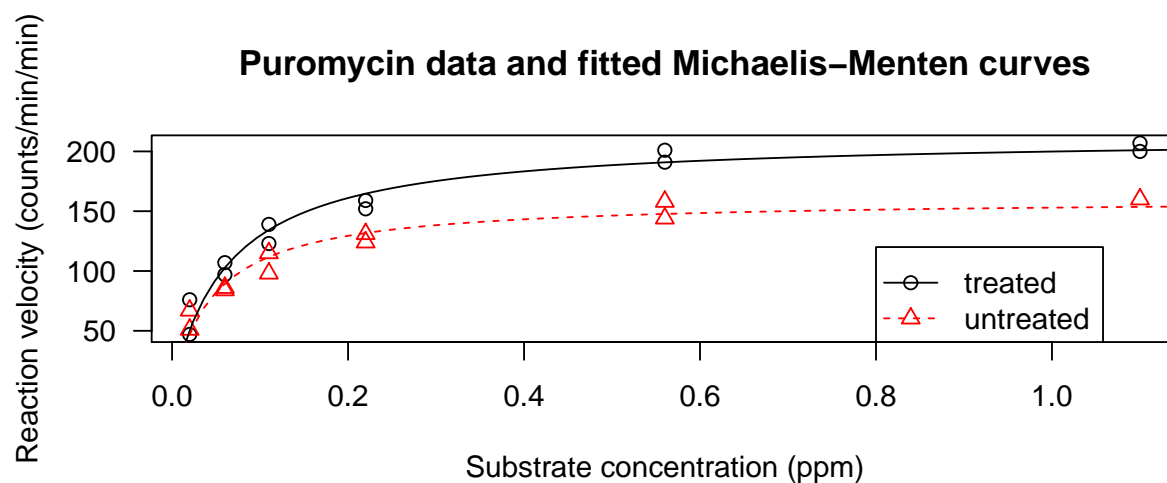
```
pairs(Puromycin, panel=panel.smooth, col=c(3,4)[Puromycin$state])
```





```
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")
## simplest form of fitting the 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))
fm2 <- nls(rate ~ Vm * conc/(K + conc), data = Puromycin,
           subset = state == "untreated",
           start = c(Vm = 160, K = 0.05))
summary(fm1)
##
## Formula: rate ~ Vm * conc/(K + conc)
##
## Parameters:
##      Estimate Std. Error t value Pr(>|t|)
## Vm 2.127e+02  6.947e+00  30.615 3.24e-11 ***
## K  6.412e-02  8.281e-03   7.743 1.57e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 10.93 on 10 degrees of freedom
##
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 8.824e-06
summary(fm2)
##
## Formula: rate ~ Vm * conc/(K + conc)
##
## Parameters:
##      Estimate Std. Error t value Pr(>|t|)
## Vm 1.603e+02  6.480e+00  24.734 1.38e-09 ***
## K  4.771e-02  7.782e-03   6.131 0.000173 ***
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 9.773 on 9 degrees of freedom
##
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 4.446e-06
## add fitted lines to the plot
conc <- seq(0, 1.2, length.out = 101)
lines(conc, predict(fm1, list(conc = conc)), lty = 1, col = 1)
lines(conc, predict(fm2, list(conc = conc)), lty = 2, col = 2)
legend(0.8, 120, levels(Puromycin$state), col = 1:2, lty = 1:2, pch = 1:2)
```



```
## using partial linearity
fm3 <- nls(rate ~ conc/(K + conc), data = Puromycin,
  subset = state == "treated", start = c(K = 0.05),
  algorithm = "plinear")
```

## 6 Introduction – ggplot2

The **ggplot2** package has the capabilities to produce very high quality graphics in a relatively easy way.

There are two main functions to know: `qplot()` and `ggplot()`.

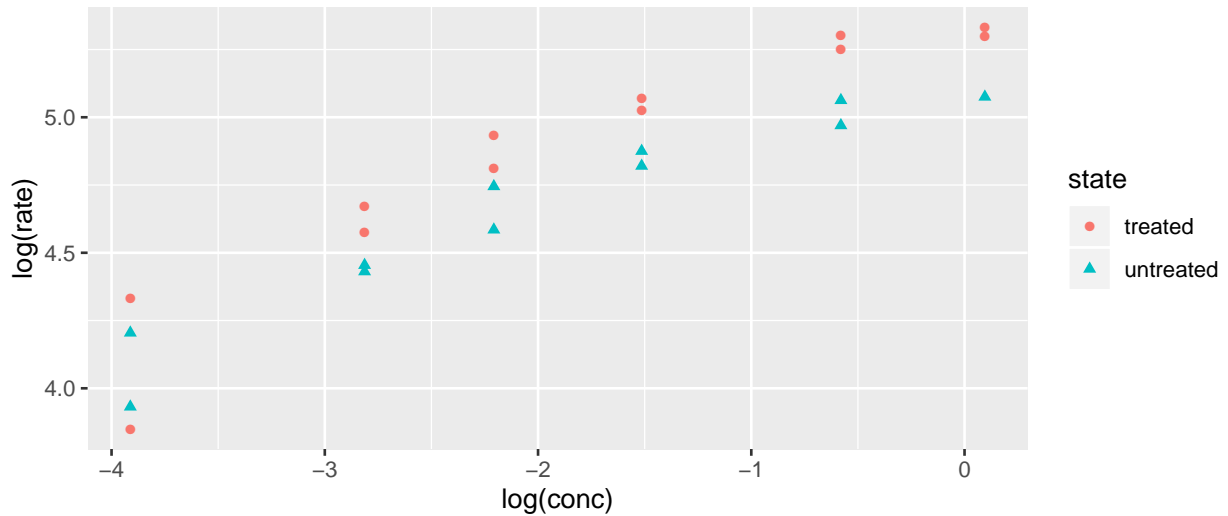
`qplot()` (short for “quick plot”) meets many requirements for producing very sophisticated graphics in a single line.

`ggplot()` offers additional capabilities and constructs graphics in a slightly different (but more logical) way.

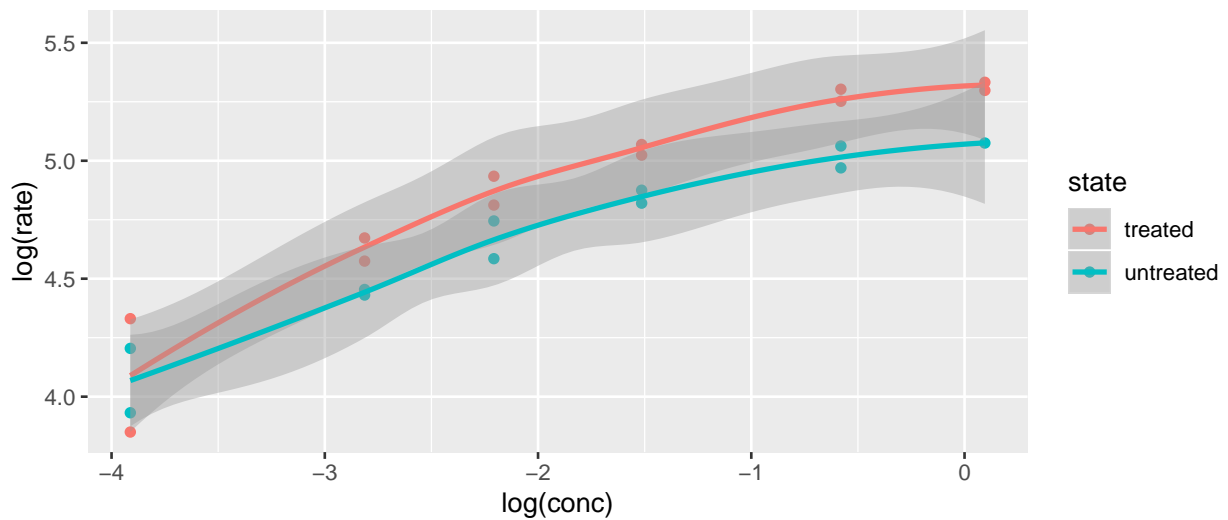
See <http://ggplot2.org/> for more information.

### 6.1 Scatterplots using `qplot()`

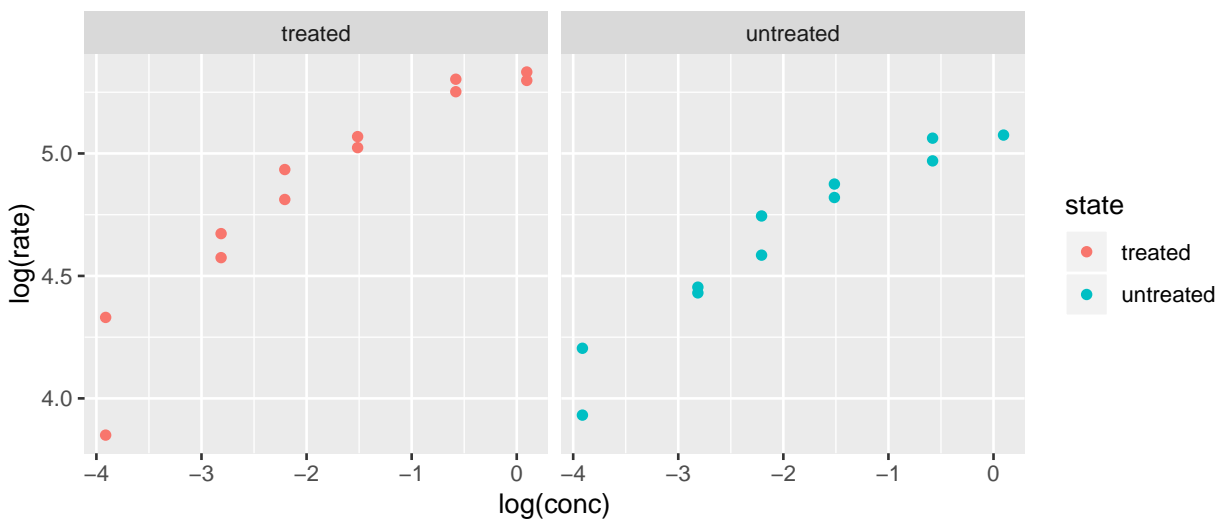
```
library( ggplot2 )
qplot(log(conc), log(rate), data=Puromycin,
      colour=state, # colour by state variable
      shape=state) # plot symbol by state variable
```



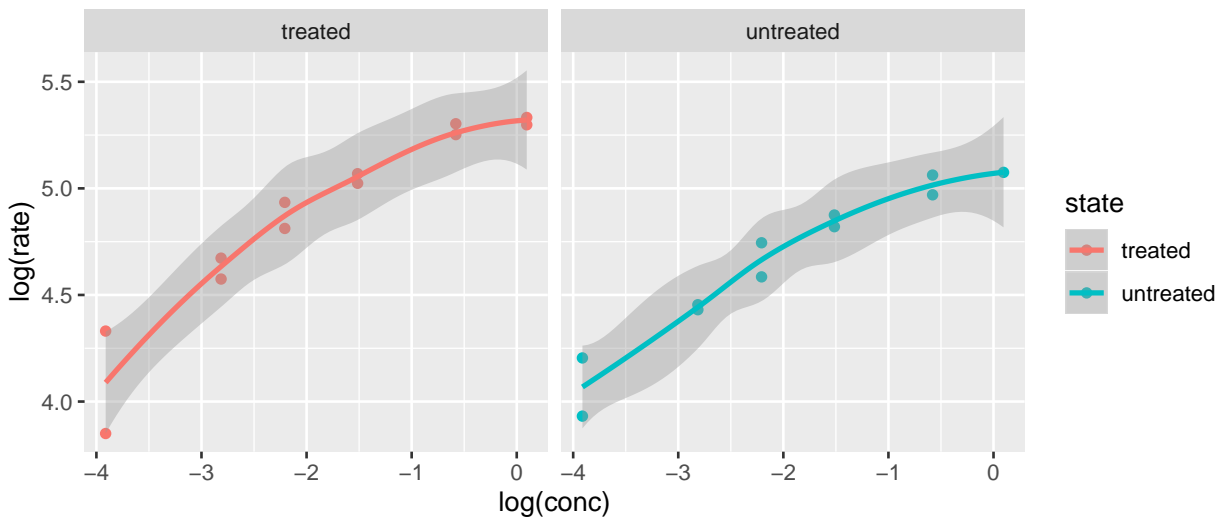
```
qplot(log(conc), log(rate), data=Puromycin, colour=state,
      geom=c("point", "smooth"))
```



```
qplot(log(conc), log(rate), data=Puromycin, colour=state,
      facets= ~state) # create groupwise plots
```



```
qplot(log(conc), log(rate), data=Puromycin, colour=state,
      facets= ~state,
      geom=c("point", "smooth"))
```



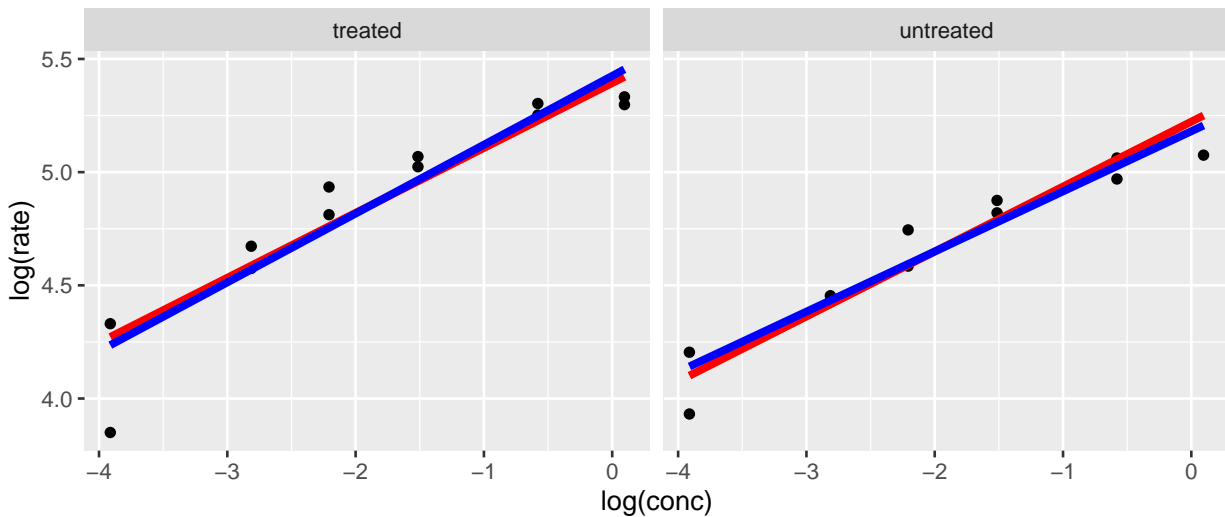
## 6.2 Scatterplots using `ggplot()`

Plot data and add curves with fitted values

```
mod1 <- lm(log(rate)~state+log(conc), data=Puromycin)
coef( mod1 ) # different intercepts; common slope
##      (Intercept) stateuntreated      log(conc)
##      5.3936426   -0.1708056    0.2867549
mod2 <- lm(log(rate)~state+state*log(conc), data=Puromycin)
coef( mod2 ) # different intercepts; different slopes
##              (Intercept)      stateuntreated
##              5.42504519      -0.24444621
##              log(conc) stateuntreated:log(conc)
##              0.30399115      -0.03839565
```

```
Puromycin$fit1 <- fitted( mod1 )
Puromycin$fit2 <- fitted( mod2 )
```

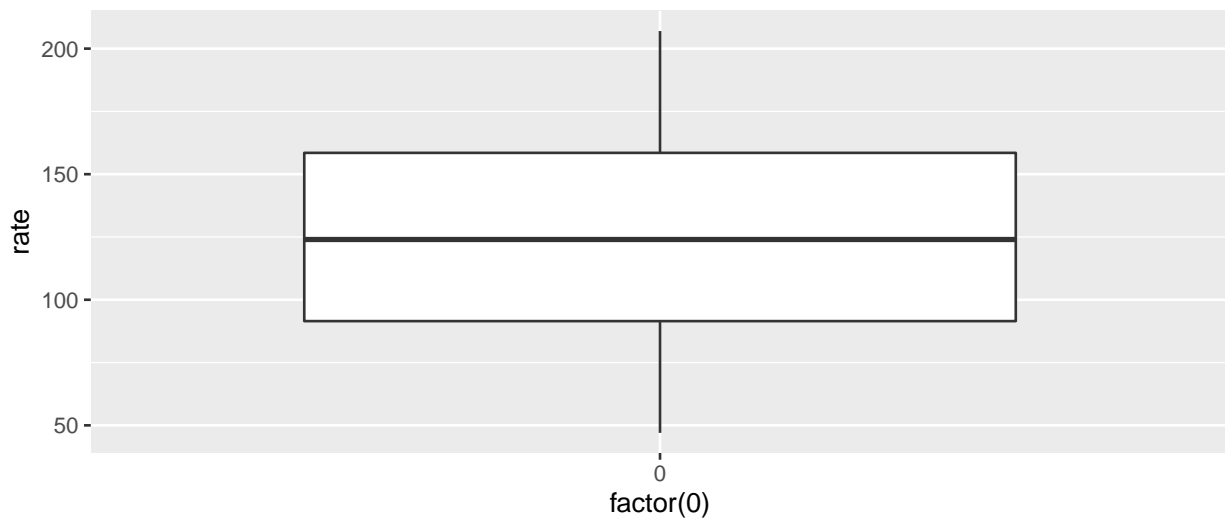
```
ggplot(Puromycin) + geom_point(aes(x=log(conc), y=log(rate))) +
  facet_grid(. ~ state ) +
  geom_line(aes(x=log(conc), y=fit1), col='red',lwd=1.5) +
  geom_line(aes(x=log(conc), y=fit2), col='blue',lwd=1.5)
```



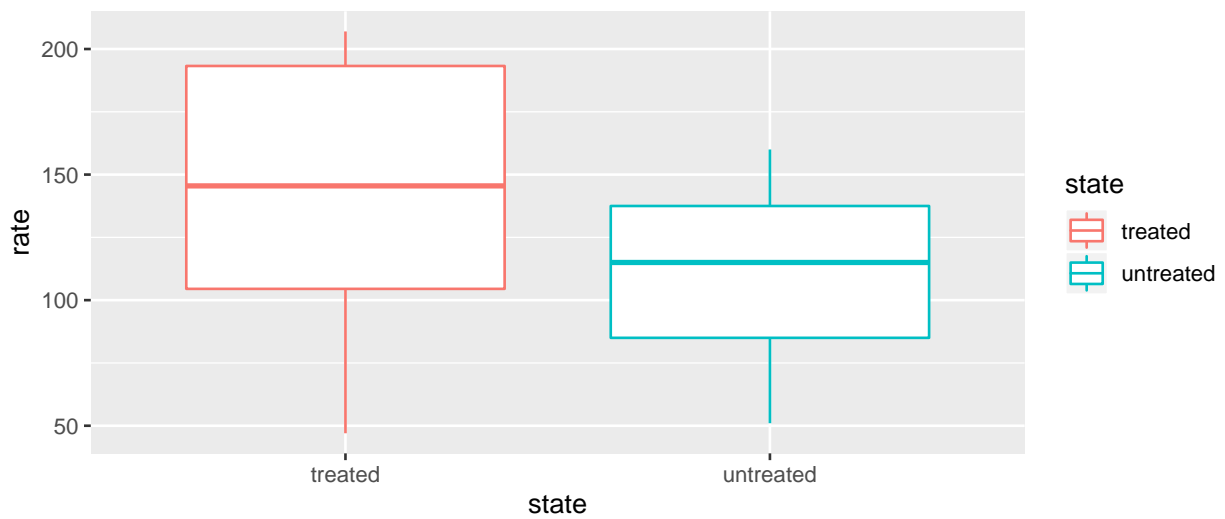
## 6.3 Some special plots using `qplot()`

### 6.3.1 Boxplot

```
qplot(factor(0), rate, data=Puromycin, geom="boxplot")
```

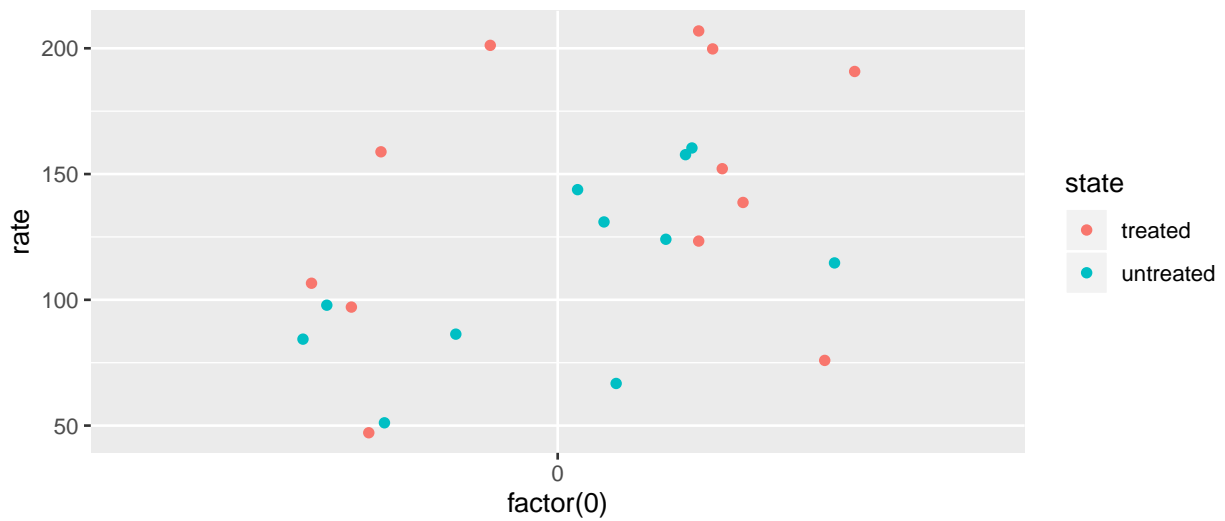


```
qplot(state, rate, data=Puromycin, geom="boxplot", color=state)
```

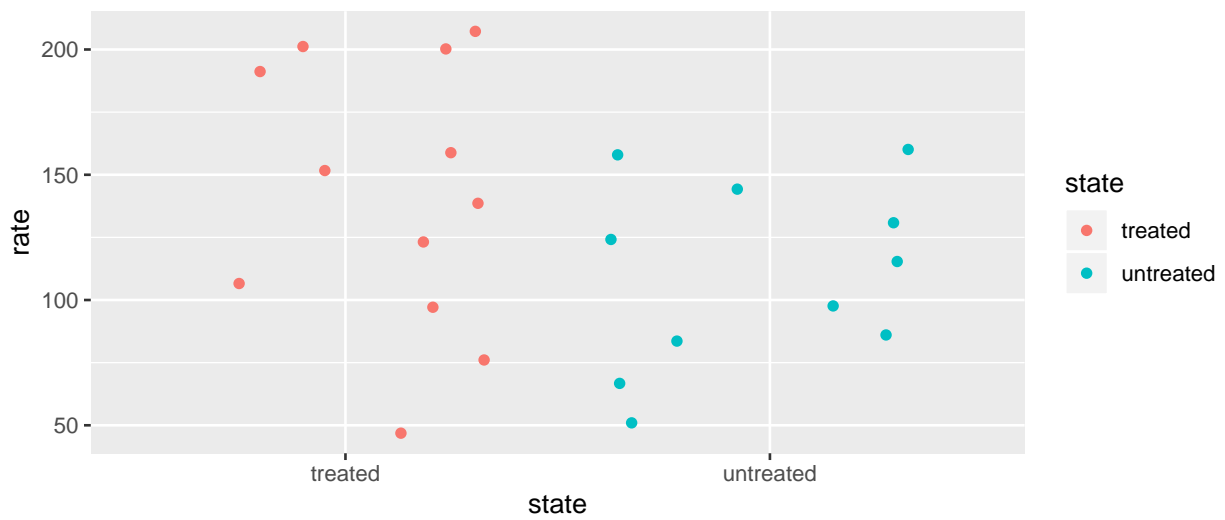


### 6.3.2 Dotplot

```
qplot(factor(0), rate, data=Puromycin, geom="jitter", color=state)
```

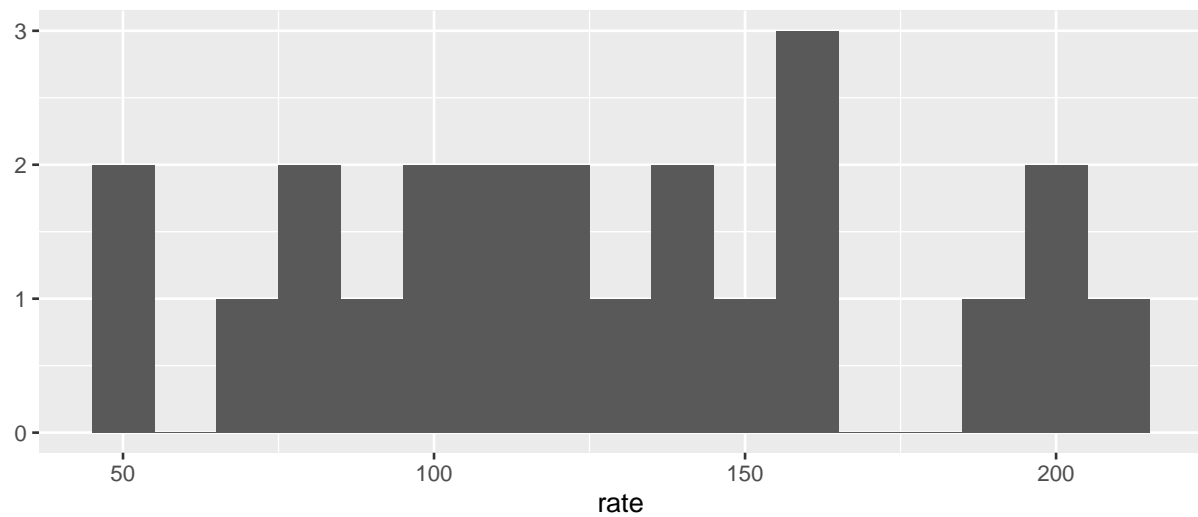


```
qplot(state, rate, data=Puromycin, geom="jitter", color=state)
```

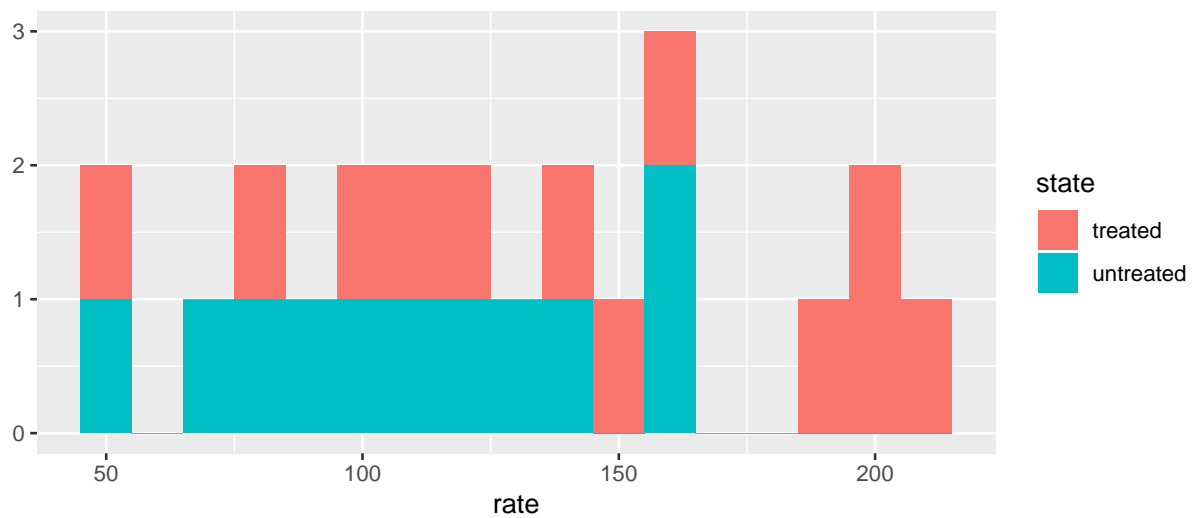


### 6.3.3 Histogram

```
qplot(rate, data=Puromycin, geom="histogram", binwidth=10)
```



```
qplot(rate, data=Puromycin, geom="histogram", fill=state, binwidth=10)
```



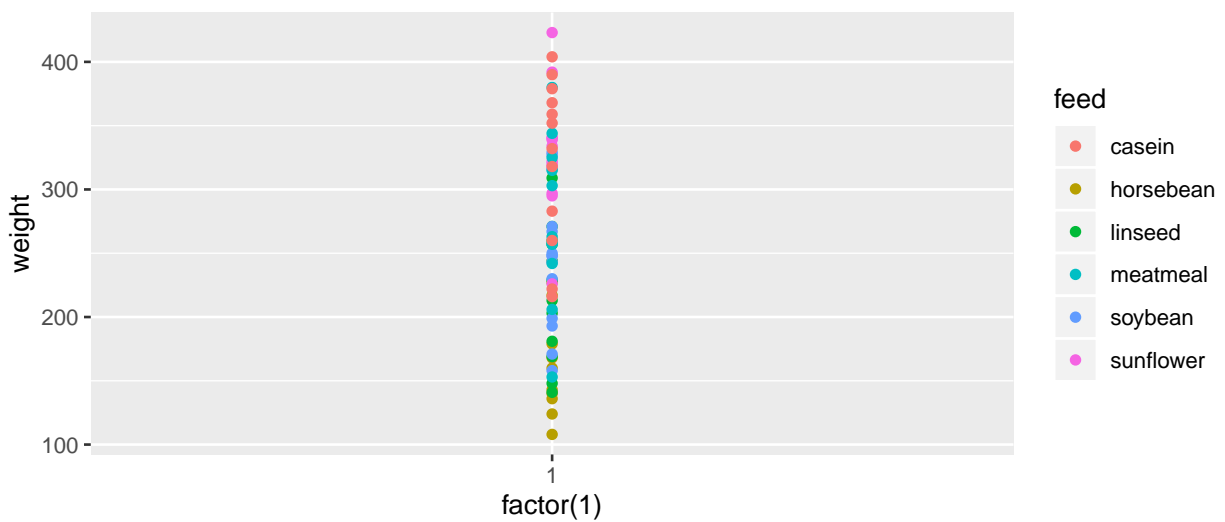
```
head( chickwts, 15 )
##    weight    feed
## 1    179 horsebean
## 2    160 horsebean
## 3    136 horsebean
## 4    227 horsebean
## 5    217 horsebean
## 6    168 horsebean
## 7    108 horsebean
## 8    124 horsebean
## 9    143 horsebean
## 10   140 horsebean
## 11   309  linseed
## 12   229  linseed
## 13   181  linseed
## 14   141  linseed
## 15   260  linseed
```

```
qplot(feed, weight, data=chickwts, color=feed)
```

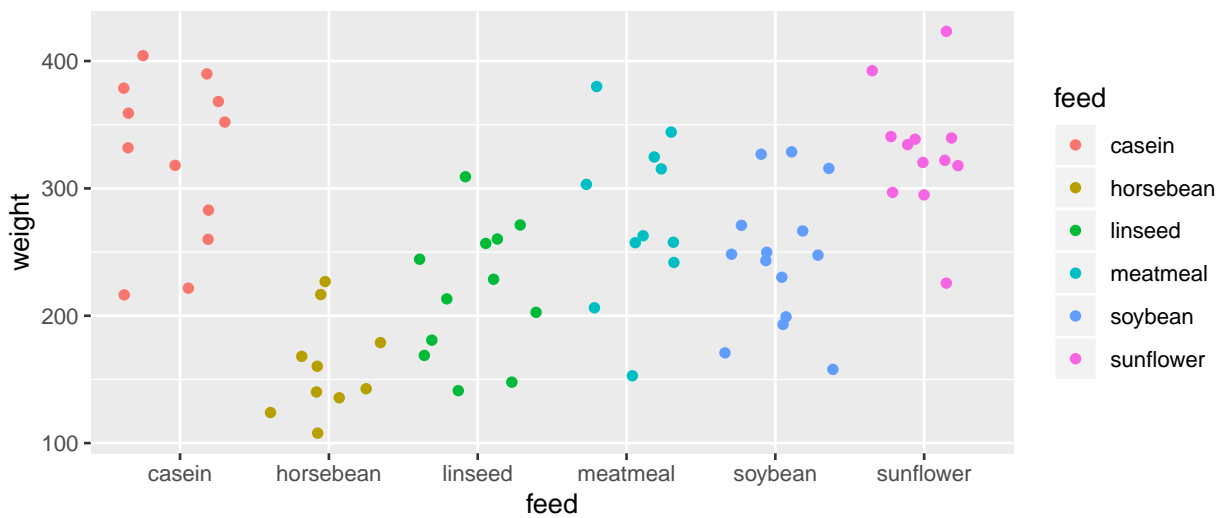




```
qplot(factor(1), weight, data=chickwts, color=feed)
```



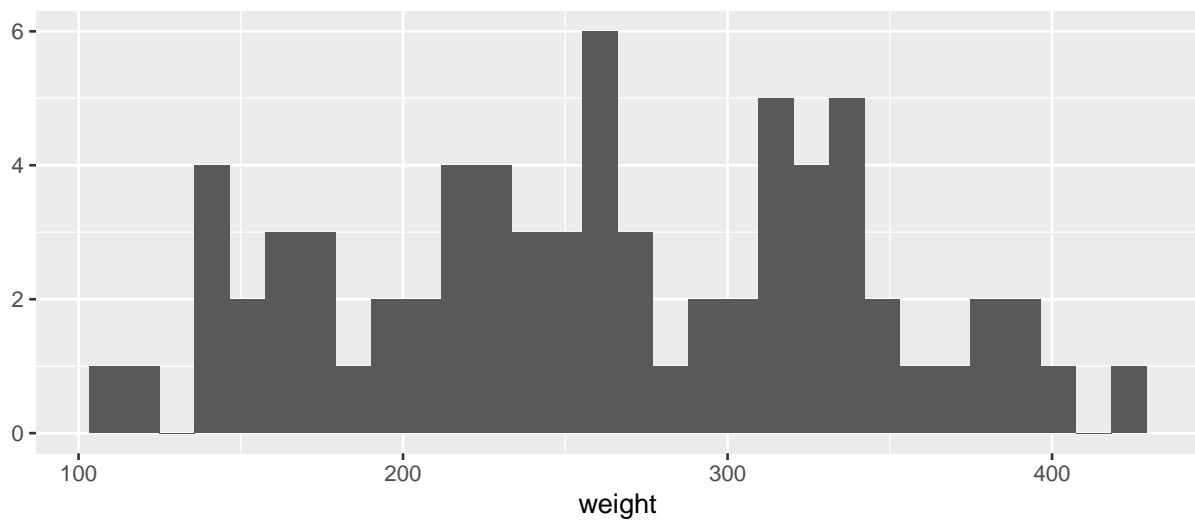
```
qplot(feed, weight, data=chickwts, geom="jitter", color=feed)
```



```
qplot(feed, weight, data=chickwts, geom="boxplot", color=feed)
```



```
qplot(weight, data=chickwts, geom="histogram")
```



## 7 Summarizing data

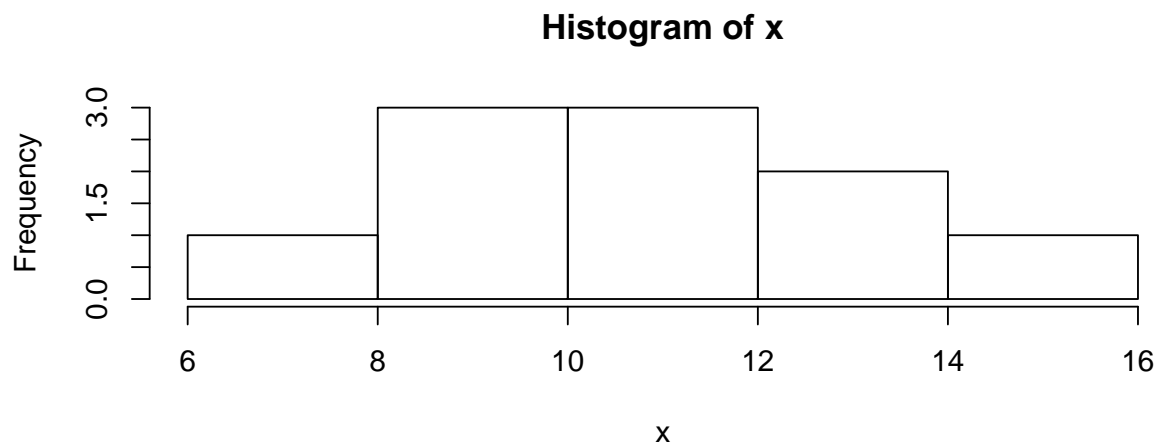
Data is a list of two vectors, giving the wear of shoes of materials x and y for one foot each of ten boys.

```
data(shoes, package="MASS")
names(shoes) <- c("x", "y")
shoes.df <- as.data.frame(shoes)
```

First focus on data for material x;

```
x <- shoes.df$x; x
## [1] 13.2  8.2 10.9 14.3 10.7  6.6  9.5 10.8  8.8 13.3
```

```
hist( x )
```



We shall look at measures of where is the "location" or "center" of the data and what is the "spread" of data.

## 7.1 Measures of location

We have  $n = 10$  observations in the vector  $x$ . Denote these symbolically by

$$x_1, x_2, x_3, \dots, x_9, x_{10}$$

and they read “x one”, “x two” etc.

For the sum  $x_1 + x_2 + x_3 + \dots + x_9 + x_{10}$  we write

$$x_{\cdot} = \sum_{i=1}^{10} x_i = x_1 + x_2 + x_3 + \dots + x_9 + x_{10}$$

and the left hand side reads “x dot” and  $\sum_{i=1}^{10} x_i$  reads “the sum of  $x_i$  as  $i$  goes from 1 to  $n$ ”. The symbol  $x_{\cdot}$  is of course an arbitrary name for the sum.

If we divide  $x_i$  by the number of observations  $n$  (here  $n = 10$ ) we get the mean (or average). It is common to write  $\bar{x}$  (pronounced “x bar”) for the average:

$$\bar{x} = \frac{1}{10} x_{\cdot} = \frac{1}{10} \sum_{i=1}^{10} x_i = \frac{1}{10} (x_1 + x_2 + x_3 + \dots + x_9 + x_{10})$$

The mean (or average) is

```
sum(x) / length(x)
## [1] 10.63
mean(x)
## [1] 10.63
```

The median is a related measure: We sort the data:

```
x <- sort( x ); x
## [1] 6.6 8.2 8.8 9.5 10.7 10.8 10.9 13.2 13.3 14.3
```

If the number of data points is odd, the median is the middle data point.

If the number is even, the median is the average around the two points in the middle:

```
(10.7 + 10.8) / 2
## [1] 10.75
median( x )
## [1] 10.75
```

## 7.2 Measures of spread

There are many different measures of spread. The most common measure of spread is the standard deviation.

The squared distance of  $x_i$  to  $\bar{x}$  is  $(x_i - \bar{x})^2$ . The average squared distance

$$\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$$

is a measure of spread of data. For technical reasons we divide by  $n - 1$  rather than  $n$  and obtain the sample variance

$$s_x^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$$

If the units of the  $x_i$ s are, say “meter”, then so is the unit of  $\bar{x}$  but the units of  $s_x^2$  is “meter squared”. This leads to the sample standard deviation which also has unit “meter”:

$$s_x = \sqrt{s_x^2}$$

```
var( x )
## [1] 6.009
sd( x )
## [1] 2.451326
```

An alternative measure of spread is the interquartile range (IQR). Consider this

```
quant <- quantile(x)
quant
##      0%      25%      50%      75%     100%
##  6.600   8.975  10.750  12.625  14.300
```

The 50% quantile is the median. The 75% quantile is the median of the highs and the 25% quantile is the median of the lows. The difference between these two quantiles is the IQR:

```
quant[4] - quant[2]
##      75%
##  3.65
```

## 7.3 Standardizing variables

The Z-SCORE is a measure of how many standard deviations measurements are from the sample mean:

$$z_i = \frac{x_i - \bar{x}}{s_x}$$

Notice that  $z_i$  has no unit.

```
( x - mean( x ) ) / sd( x )
## [1] -1.64400807 -0.99130015 -0.74653468 -0.46097497  0.02855597
## [6]  0.06935022  0.11014446  1.04841209  1.08920634  1.49714879
```

One important point about standardized variables is that it does not change under linear transformations:

```
x2 <- 100 * x + 1000
mean(x)
## [1] 10.63
mean(x2)
## [1] 2063
sd(x)
## [1] 2.451326
sd(x2)
## [1] 245.1326
( x2 - mean( x2 ) ) / sd( x2 )
## [1] -1.64400807 -0.99130015 -0.74653468 -0.46097497  0.02855597
## [6]  0.06935022  0.11014446  1.04841209  1.08920634  1.49714879
```

## 7.4 An empirical rule

A practical interpretation of  $\bar{x}$  and  $s_x$  is that for nearly symmetrical mound shaped datasets, about 68% of data is within one standard deviation of the mean and 95% is within two standard deviations of the mean.

Often we can get reasonable estimates of the sample mean and standard deviation by simply looking at data:

If data is not highly skewed then the mean and median are roughly the same. An estimate of the median is obtained by looking at the center of the (sorted) data vector:

```
x
## [1] 6.6 8.2 8.8 9.5 10.7 10.8 10.9 13.2 13.3 14.3
```

Hence, the median is somewhere between 10 and 11:

```
median( x )
## [1] 10.75
mean( x )
## [1] 10.63
```

About 95% of the observations will fall in the interval  $\bar{x} \pm 2s_x$ .

If we say that 95% of the observations are “practically all observation” then practically all observations fall in this interval

Hence the largest minus the smallest value is about 4 standard deviations (we call this the 4SD-RULE and hence that a crude estimate of the standard deviation is

```
(max( x ) - min( x )) / 4
## [1] 1.925
sd( x )
## [1] 2.451326
```

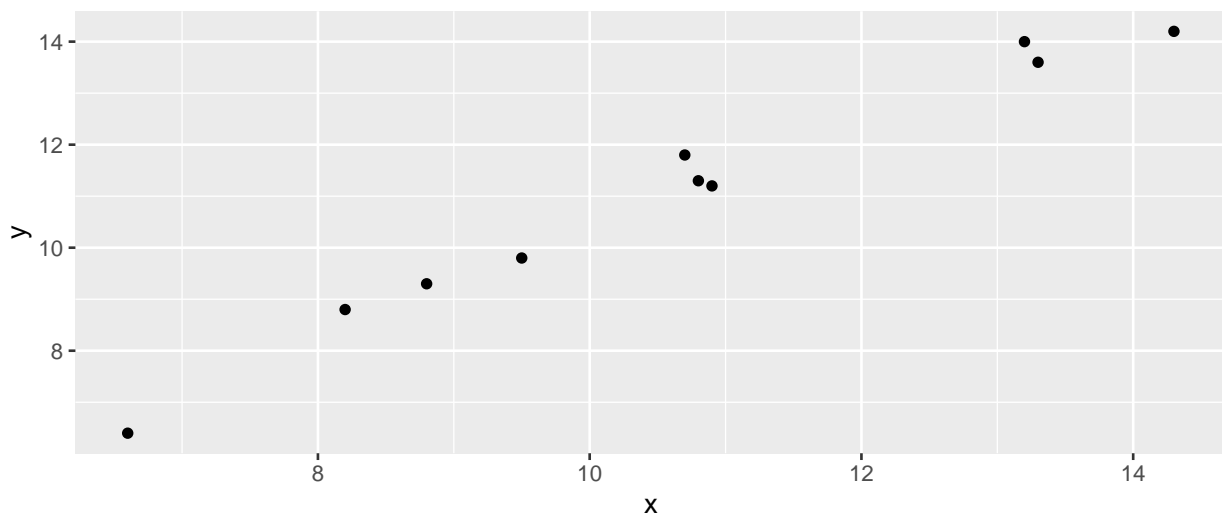
## 7.5 Covariance and correlation

In the shoes.df data there are two vectors

```
x <- shoes.df$x
y <- shoes.df$y
```

Clearly these measurements “co-vary”

```
library(ggplot2)
qplot(x,y)
```



A measure of how they co-vary is the (sample) covariance between  $x$  and  $y$

$$s_{xy} = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$$

Notice: If we replace  $y_i$  by  $x_i$  and  $\bar{y}$  by  $\bar{x}$  above then we get the sample covariance between  $x$  and  $x$  which is the (sample) variance of  $x$ .

Closely related to the covariance is the correlation coefficient:

$$\rho_{xy} = \frac{s_{xy}}{s_x s_y}$$

The correlation is always in the interval  $\pm 1$ . If the correlation is 1 or  $-1$  there is a perfect linear association between  $x$  and  $y$ . If the correlation is 0 there is no linear association at all.

```
cov( x, y )
## [1] 6.100889
cor( x, y )
## [1] 0.9882255
```

Just like the  $z$ -score is unaffected by change in scale and location, so is the correlation:

```
cov( x, y )
## [1] 6.100889
cov( 100 * x + 1000, 10 * y + 100 )
## [1] 6100.889
cor( x, y )
## [1] 0.9882255
cor( 100 * x + 1000, 10 * y + 100 )
## [1] 0.9882255
```

## 7.6 Computations

```

cov( shoes.df )
##           x           y
## x 6.009000 6.100889
## y 6.100889 6.342667
cor( shoes.df )
##           x           y
## x 1.0000000 0.9882255
## y 0.9882255 1.0000000
cov.wt( shoes.df )
## $cov
##           x           y
## x 6.009000 6.100889
## y 6.100889 6.342667
##
## $center
##           x           y
## 10.63 11.04
##
## $n.obs
## [1] 10

```

## 8 Linear models

### 8.1 Running example: The cars data

The cars dataset (that comes with R) contains pairs of measurements of stopping distance (dist in ft) and speed (speed in mph). The data were recorded in the 1920s.

```

head(cars, 4)
##   speed dist
## 1     4     2
## 2     4    10
## 3     7     4
## 4     7    22

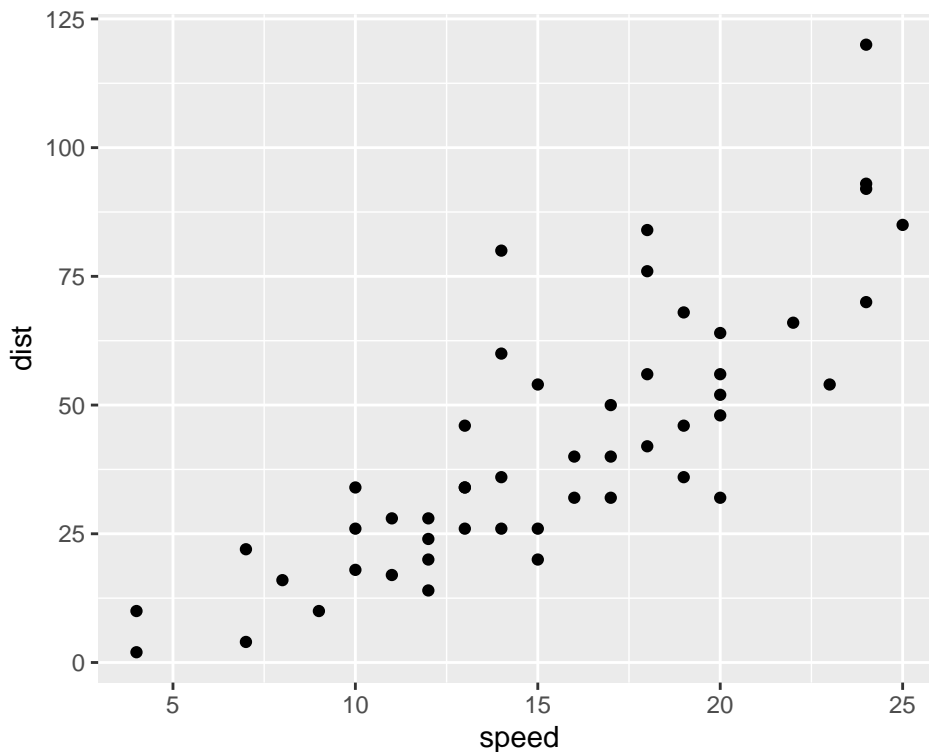
```

```

qplot(speed, dist, data=cars)

```



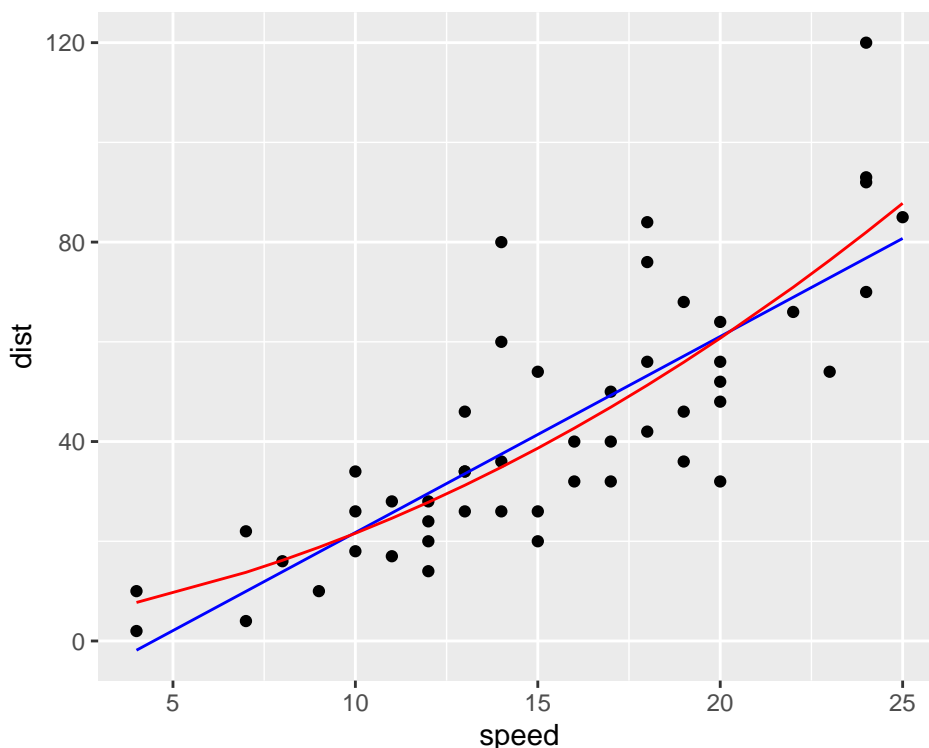


If we want to predict stopping distance from speed, one option could be a linear relationship between  $y=\text{dist}$  and  $x=\text{speed}$ . There is an indication of upwards curvature in the plot above which is supported by physical theory: Stopping distance increases with the square of speed. Perhaps an approximately quadratic relation between  $y=\text{dist}$  and  $x=\text{speed}$ .

The PARAMETERS describing a straight line and a 2nd degree polynomial can be estimated using the lm() function:

```
cars.01 <- lm( dist ~ speed, data=cars )
coef( cars.01 )
## (Intercept)      speed
##   -17.5791     3.9324
cars.012 <- lm( dist ~ speed + I(speed^2), data=cars )
coef( cars.012 )
## (Intercept)      speed  I(speed^2)
##    2.470138    0.913288    0.099959
```

```
qplot(speed, dist, data=cars) +
  geom_line(aes(speed, predict( cars.01 )), color="blue" ) +
  geom_line(aes(speed, predict( cars.012 )), color="red" )
```



## 8.2 Running example: The Puromycin data

The Puromycin data frame has 23 rows and 3 columns of the reaction velocity versus substrate concentration in an enzymatic reaction involving untreated cells or cells treated with Puromycin. 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.

```
head(Puromycin, 4)
##   conc rate  state
## 1 0.02   76 treated
## 2 0.02   47 treated
## 3 0.06   97 treated
## 4 0.06  107 treated
```

```
p1 <- ggplot(Puromycin, aes(x=conc, y=rate, colour=state)) + geom_point()
p2 <- ggplot(Puromycin, aes(x=log2(conc), y=rate, colour=state)) + geom_point()
gridExtra::grid.arrange(p1, p2, ncol=1)
```

If we ignore the state variable, Fig. 1 suggests an almost linear relationship between rate and  $\log(\text{conc})$ . If we include state it seems that there are two parallel lines; one for each level of state. The [PARAMETERS](#) describing two parallel regression lines can be estimated using the `lm()` function:

```
puro1 <- lm(rate ~ log2(conc) + state, data=Puromycin)
coef( puro1 )
##   (Intercept)      log2(conc) stateuntreated
##      200.911         22.571         -25.181
```

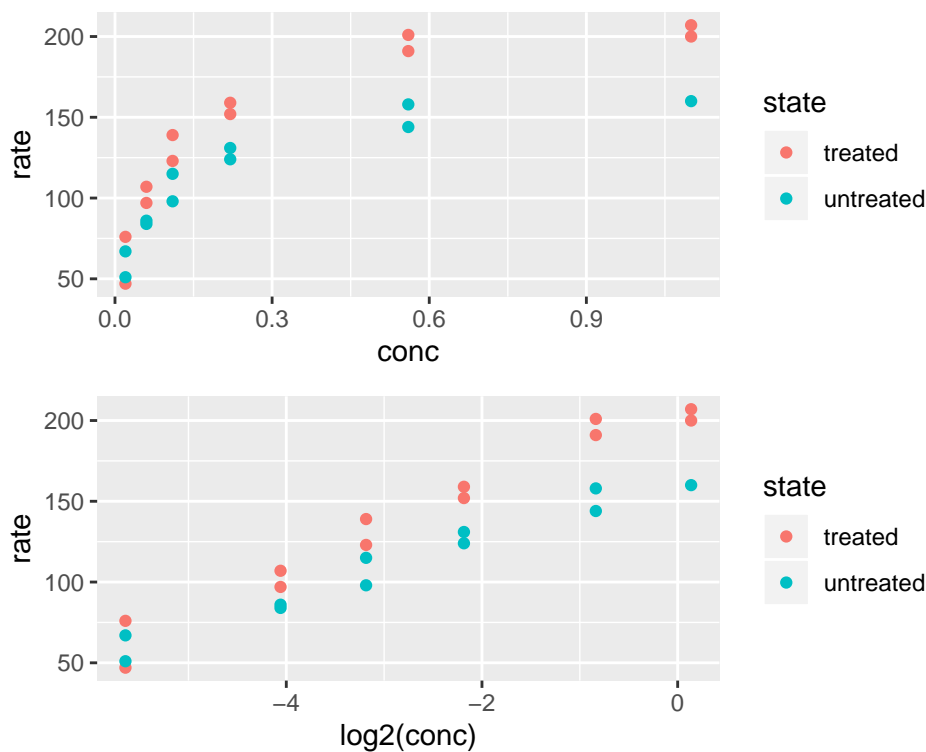


Figure 1: Puromycin...

```
p <- ggplot(Puromycin, aes(x=log2(conc), y=rate, colour=state))
p1 <- p + geom_point() + geom_line(aes(x=log2(conc), y=predict(puro1)))
p <- ggplot(Puromycin, aes(x=conc, y=rate, colour=state))
p2 <- p + geom_point() + geom_line(aes(x=conc, y=predict(puro1)))
gridExtra::grid.arrange(p1, p2, ncol=1)
```

### 8.3 Linear models (informally)

We want to model a RESPONSE  $y$  from, say, three PREDICTORS or EXPLANATORY VARIABLES or COVARIATES  $x_1, x_2, x_3$ .

In a linear model we assume that  $y$  is related linearly to the predictors as

$$y = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + e$$

where  $e$  is an error term. The terms  $\beta_1, \dots, \beta_3$  are unknown PARAMETERS which are to be estimated from data.

Sometimes we want to indicate that we have  $n$  observations  $y_1, y_2, \dots, y_n$ . Similarly, there are  $n$  instances of the predictor  $x_j$  which we write  $x_{1j}, x_{2j}, \dots, x_{nj}$ . In this case we write

$$y_i = \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + e_i \quad (1)$$

However, this extra index ( $i$ ) is often just cumbersome to keep track of and therefore we often omit it.

In a linear model the parameters enter linearly, but the predictors need not do so. For example

$$y = \beta_1 + \beta_2 \log x_2 + \beta_3 x_1^2 x_3 + e$$

We can just think of  $1$ ,  $\log x_2$ , and  $x_1^2 x_3$  as predictors and we are back in the original form. On the other hand  $y = \beta_1 x_1^{\beta_2} + e$  is not linear.

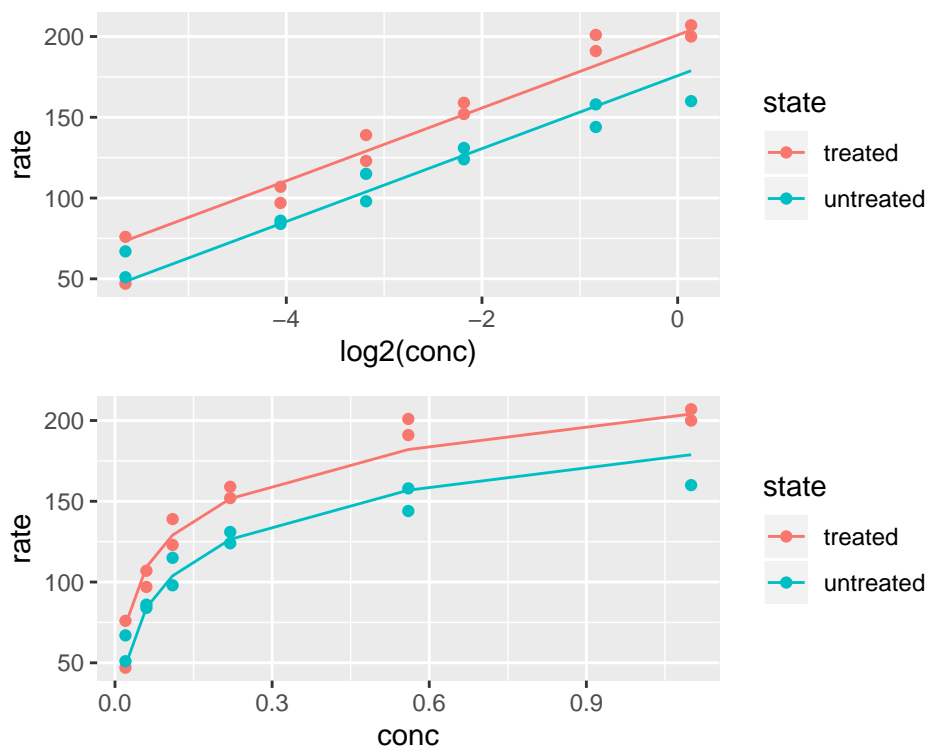


Figure 2: Parallel regression lines seem to fit well to the Puromycin data.

## 8.4 Polynomial regression

For the cars data we considered a POLYNOMIAL REGRESSION model:

$$y_i = \beta_1 + \beta_2 x_i + \beta_3 x_i^2 + \epsilon_i$$

Above,  $\epsilon_i$  denotes an “random error term” or a “deviation from the mean”  $\mu_i = \beta_1 + \beta_2 x_i + \beta_3 x_i^2$ . We shall assume that  $E(\epsilon_i) = 0$ , the  $\text{Var}(\epsilon_i) = \sigma^2$  and that the  $\epsilon_i$ s are uncorrelated. We make no assumptions about the distribution of  $\epsilon_i$ .

Return again to the linear model

$$y = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + e$$

If we have  $n$  cases in a dataset we can organize the  $y$ 's in the vector  $y = (y_1, y_2, \dots, y_n)$  and the predictors in the matrix  $X$ :

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \vdots & \vdots & \vdots \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix}$$

If we stack all  $n = 50$  cases in the cars data on top of each other we get the matrix representation<sup>2</sup>

$$y = X\beta + \epsilon$$

---

<sup>2</sup>Only if we have col of 1s

## 8.5 Analysis of covariance (ANCOVA)

Consider the Puromycin data of Section 8.2, page 66.

```
head(Puromycin, 4)
##   conc rate  state
## 1 0.02   76 treated
## 2 0.02   47 treated
## 3 0.06   97 treated
## 4 0.06  107 treated
```

Fig. 1 suggests an almost linear relationship between rate and  $\log(\text{conc})$ . If we ignore the state variable, then we are in a linear regression setting, see Section 8.1, where the response  $y$  is rate and the explanatory variable, say  $x_1$ , is  $\log(\text{conc})$ . An extension of a linear regression model (based on looking at data) is to assume that there are two parallel regression lines; one for level of state. This is encompassed by introducing another explanatory variable  $x_2$  where  $x_2 = 0$  when state=treated and  $x_2 = 1$  when state=untreated. The model then becomes

$$y = \beta_1 + \beta_2 x_1 + \beta_3 x_2 + \epsilon$$

Hence the role of  $x_2$  is to act as a switch: When  $x_2 = 1$ , the effect of  $\beta_3$  comes into play in the sense that the intercept becomes  $\beta_1 + \beta_3$  whereas when  $x_2 = 0$  the intercept is simply  $\beta_1$ . The slope is  $\beta_2$  in both cases. In that way we can obtain two parallel regression lines

```
puro1 <- lm(rate ~ log2(conc) + state, data=Puromycin)
coef( puro1 )
##   (Intercept)      log2(conc) stateuntreated
##      200.911         22.571         -25.181
```

Hence the effect of changing from treated to untreated is a change in rate by -25.2.

The interpretation of this model is that increasing  $\log_2(\text{conc})$  by one unit from some value  $x$  to  $x + 1$  means that the rate on average will increase with about 23. Changing  $\log_2(\text{conc})$  with one unit is the same as doubling conc.

Parallel regression lines seem to fit well to data as shown in Fig 2. But suppose we want to consider a more general model, namely that not only intercept but also slope depends on state. This can be accomplished by this model

$$y = \beta_1 + \beta_2 x_1 + \beta_3 x_2 + \beta_4 x_1 x_2 + \epsilon$$

Above, when state=treated then  $x_2 = 0$  and several terms disappear (or are switched off), and we can write the model as  $y = \beta_1 + \beta_2 x_1 + \epsilon$ . When state=untreated then  $x_2 = 1$  then several terms are switched on, and we have  $y = (\beta_1 + \beta_3) + (\beta_2 + \beta_4)x_1 + \epsilon$ .

```
puro2 <- lm(rate ~ log2(conc) + state + log2(conc):state, data=Puromycin)
puro2
##
## Call:
## lm(formula = rate ~ log2(conc) + state + log2(conc):state, data = Puromycin)
##
## Coefficients:
##           (Intercept)           log2(conc)
##           209.19           25.72
##      stateuntreated log2(conc):stateuntreated
##          -44.61           -7.02
```

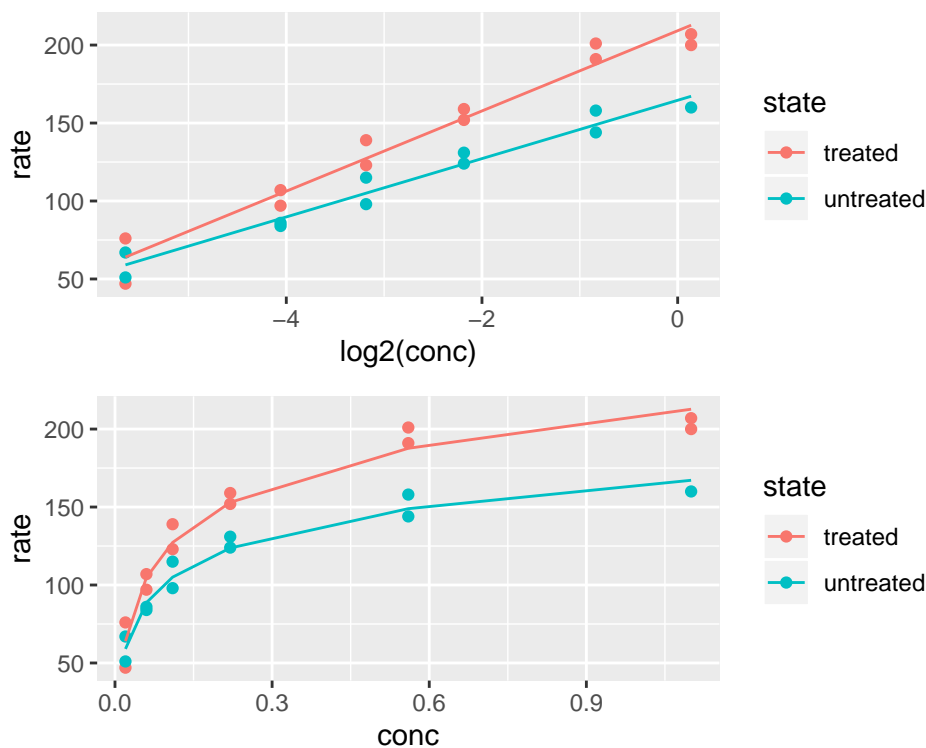


Figure 3: Non-parallel regression lines seem to fit well to the Puromycin data.

```
p <- ggplot(Puromycin, aes(x=log2(conc), y=rate, colour=state))
p1 <- p + geom_point() + geom_line(aes(x=log2(conc), y=predict(puro2)))
p <- ggplot(Puromycin, aes(x=conc, y=rate, colour=state))
p2 <- p + geom_point() + geom_line(aes(x=conc, y=predict(puro2)))
gridExtra::grid.arrange(p1, p2, ncol=1)
```

## 9 Linear models

### 9.1 Matrix representation of a linear model

Return again to the linear model

$$y = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + e$$

We organize the predictors in a matrix (often called MODEL MATRIX)

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \vdots & \vdots & \vdots \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix}$$

Likewise we can organize the  $y$ 's in the vector  $y = (y_1, y_2, \dots, y_n)$ . Let  $\beta = (\beta_1, \beta_2, \beta_3)$  and  $e = (e_1, e_2, \dots, e_n)$  then we can write

$$y = X\beta + e$$

By a LINEAR MODEL we informally mean

$$y = X\beta + e, \quad \mathbb{E}(e) = 0, \quad \text{Cov}(e) = \sigma^2 I$$

Above,  $y$  is an  $n$  vector of random variables,  $X$  is a known  $n \times p$  matrix (called a model matrix),  $\beta$  is a  $p$ -vector of unknown coefficients and  $e$  is an  $n$ -vector of random errors.

For now we make no assumption about the distribution of  $e$  except that each  $e_i$  in  $e$  must have mean zero and variance  $\sigma^2$  and that different components  $e_i$  and  $e_j$  must be uncorrelated.

Notice that what is observed is  $y$  and  $X$ .

For example, for the quadratic for the cars data we have

```
head( model.matrix( cars.012 ), 4 )
##      (Intercept) speed I(speed^2)
## 1             1     4      16
## 2             1     4      16
## 3             1     7      49
## 4             1     7      49
```

## 9.2 Least squares estimation

For any value of  $\beta$  the RESIDUALS are  $r = y - X\beta$ . The parameter  $\beta$  is estimated by minimizing the residual sum of squares

$$RSS = \sum r_i^2 = r^\top r = (y - X\beta)^\top (y - X\beta).$$

Differentiation of  $RSS$  with respect to  $\beta$  leads to a set of equations, the so-called NORMAL EQUATIONS to be solved:

$$X^\top X\beta = X^\top y$$

If  $X$  has full rank, then  $X^\top X$  is invertible and the least squares estimate for  $\beta$  is

$$\hat{\beta} = (X^\top X)^{-1} X^\top y$$

Internally, the `lm()` function creates the MODEL MATRIX  $X$  but `lm()` uses (for numerical reasons) a different method for calculating  $\hat{\beta}$ . Still we can do it here. Consider the polynomial for the cars data.

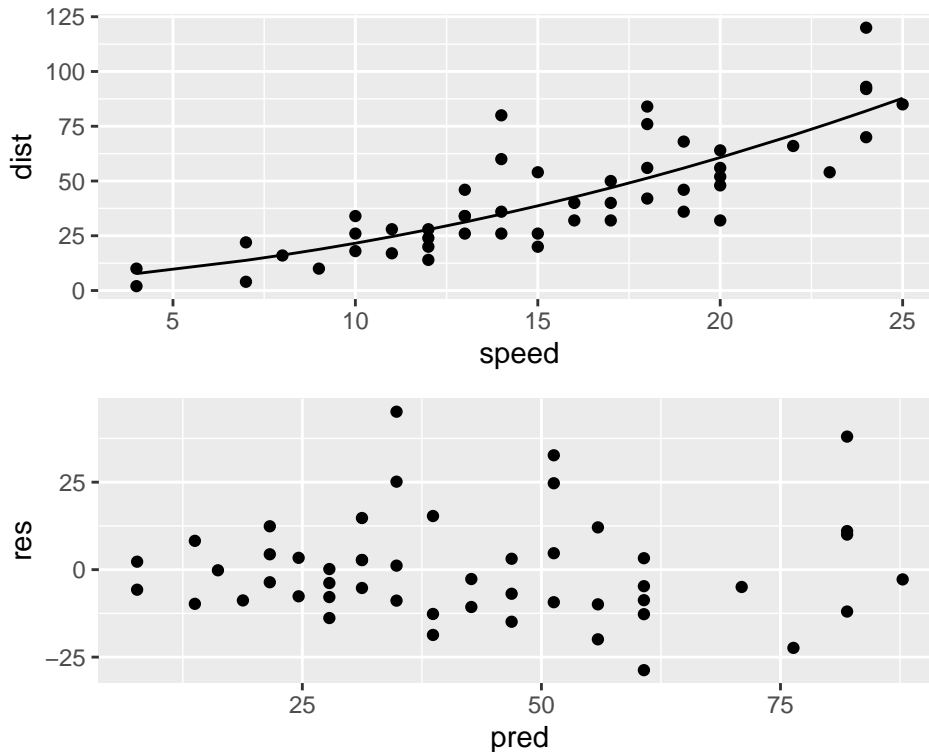
Define  $X$  and  $y$

```
X <- cbind(1, cars$speed, cars$speed^2)
## X <- model.matrix( cars.012 ) ## Alternative
y <- cars$dist
head( X, 4 )
##      [,1] [,2] [,3]
## [1,]    1    4   16
## [2,]    1    4   16
## [3,]    1    7   49
## [4,]    1    7   49
head( y )
## [1]  2 10  4 22 16 10
beta.hat <- solve( t(X) %*% X, t(X) %*% y )
beta.hat
##      [,1]
## [1,] 2.470138
## [2,] 0.913288
## [3,] 0.099959
```

The PREDICTED VALUES and RESIDUALS are:

```
pred <- X %*% beta.hat
res <- y - pred
```

```
p1 <- qplot(speed, dist, data=cars) + geom_line(aes(speed, pred))
p2 <- qplot(pred, res)
gridExtra::grid.arrange(p1, p2, ncol=1)
```



## 10 Least squares estimation – in theory an practice

### 10.1 General remark on solving $Ax = b$

Consider general system of  $n$  linear equations with  $n$  unknowns

$$Ax = b$$

The “mathematical solution” to finding  $x$  is  $x = A^{-1}b$ , but this is often not what we do in practice. That is, we do not first find  $A^{-1}$  and then multiply  $A^{-1}$  and  $b$ .

A numerically better approach is to solve the system of equations directly. For example, form the augmented matrix  $[A : b]$ . Do Gauss elimination on this matrix until  $A$  is replaced by  $I$ . We then have  $[I : x]$ .

In R:

```
x <- solve(A) %*% b ## NO!
x <- solve(A, b)    ## YES!
```



$$\begin{bmatrix} 1 & -1 \\ -1 & 1 + 10^{-10} \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 10^{-10} \end{bmatrix}$$

Example: A HILBERT MATRIX  $A$  is a square matrix whose entries are  $A_{ij} = \frac{1}{i+j-1}$ . A Hilbert matrix is symmetric, positive definite and close to being singular.

```
options("digits" = 2)
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
p <- 8
A <- hilbert( p )
A
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,] 1.00 0.50 0.33 0.25 0.20 0.167 0.143 0.125
## [2,] 0.50 0.33 0.25 0.20 0.167 0.143 0.125 0.111
## [3,] 0.33 0.25 0.20 0.167 0.143 0.125 0.111 0.100
## [4,] 0.25 0.20 0.17 0.143 0.125 0.111 0.100 0.091
## [5,] 0.20 0.17 0.14 0.125 0.111 0.100 0.091 0.083
## [6,] 0.17 0.14 0.12 0.111 0.100 0.091 0.083 0.077
## [7,] 0.14 0.12 0.11 0.100 0.091 0.083 0.077 0.071
## [8,] 0.12 0.11 0.10 0.091 0.083 0.077 0.071 0.067
```

Construct system of equations and solve this:

```
x.true <- rep(1, p)
b <- A %*% x.true
x1 <- solve( A ) %*% b ## NO!
x2 <- solve( A, b )    ## YES!

options("digits"=8)
as.numeric( x1 )
## [1] 0.99999977 1.00000104 0.99999914 0.99999952 0.99999952
## [6] 1.00000095 0.99999928 1.00000012
as.numeric( x2 )
## [1] 1.00000000 1.00000000 1.00000001 0.99999997 1.00000008
## [6] 0.99999988 1.00000008 0.99999998
sum( (x1 - x.true)^2 )
## [1] 3.7610794e-12
## Error is 100 times smaller than above
sum( (x2 - x.true)^2 )
## [1] 3.0003578e-14
options("digits"=5)
```

## 10.2 The general remark and normal equations

In least squares estimation we therefore find  $\hat{\beta}$  by solving the NORMAL EQUATIONS  $X^T X \beta = X^T y$  instead of copying the mathematical expression  $\hat{\beta} = (X^T X)^{-1} X^T y$ :

```
beta.hat <- solve( t(X) %*% X, t(X) %*% y ) ## YES
beta.hat <- solve( t(X) %*% X ) %*% t(X) %*% y ## NO
```

### 10.3 Example: $X$ is nearly rank deficient

Example: Minimize  $(y - X\beta)^\top (y - X\beta)$  where

$$X = \begin{bmatrix} 1 & -1 \\ 0 & 10^{-5} \\ 0 & 0 \end{bmatrix} \quad y = \begin{bmatrix} 0 \\ 10^{-5} \\ 1 \end{bmatrix}$$

Notice:  $X$  has rank 2 but only just. Normal equations  $X^\top X\beta = X^\top y$  are:

$$\begin{bmatrix} 1 & -1 \\ -1 & 1 + 10^{-10} \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 10^{-10} \end{bmatrix}$$

Solution:  $\beta = (1, 1)$ .

#### 10.3.1 The "textbook formula" fails

```
options("digits"=15)
X <- matrix(c(1,0,0, -1, 1e-5, 0), nr=3)
y <- c(0, 1e-5, 1);
```

```
XtX <- t(X) %*% X; XtX
##      [,1]      [,2]
## [1,]      1 -1.0000000000
## [2,]     -1  1.0000000001
Xty <- t(X) %*% y; Xty
##      [,1]
## [1,] 0e+00
## [2,] 1e-10
b.wrong <- solve( XtX, Xty )
b.wrong
##      [,1]
## [1,] 0.999999917259636
## [2,] 0.999999917259636
```

#### 10.3.2 QR-factorization and the normal equations

We can factor  $X$  ( $n \times p$ ) as  $X = QR$  where  $Q$  is  $n \times p$  matrix with orthonormal columns and  $R$  is  $p \times p$  upper triangular, invertible matrix; this is called the QR-factorization.

The normal equations  $X^\top X\beta = X^\top y$  become

$$R^\top Q^\top QR\beta = R^\top R\beta = R^\top Q^\top y$$

Since  $R^\top$  is invertible we have

$$R\beta = Q^\top y$$

This is a triangular system of equations which can be solved by backward substitution.

Notice, symbolically (but we do not do so in practice) we have:

$$\beta = R^{-1}Q^\top y$$

#### 10.3.3 QR-factorization works

```

options("digits"=15)
QR <- qr(X) # only intermediate quantity
Q  <- qr.Q( QR ); Q
##      [,1] [,2]
## [1,]  -1   0
## [2,]   0  -1
## [3,]   0   0
R  <- qr.R( QR ); R
##      [,1] [,2]
## [1,]  -1  1e+00
## [2,]   0 -1e-05
Qty <- t(Q) %*% y; Qty
##      [,1]
## [1,]  0e+00
## [2,] -1e-05
solve(R, Qty)
##      [,1]
## [1,]    1
## [2,]    1

```

Solution to  $R\beta = Q^\top y$  must be (1,1).

## Index

, 9  
:, 14  
?, 12  
abline(), 42  
attach(), 44  
attr(), 21  
boxplot(), 44  
c(), 6, 14, 15  
data.frame(), 9  
density(), 45  
detach(), 44  
ggplot(), 50, 52  
help( help ), 13  
help(), 12  
help.start(), 13  
hist(), 45  
image(), 13  
lm(), 11, 65, 66, 71  
matrix(), 15, 18  
pairs(), 48  
par(mfrow=c(m,n)), 13  
plot(), 35, 43  
q(), 4, 12  
qplot()[ggplot2], 7  
qplot(), 50, 53  
qqline(), 47  
qqnorm(), 47  
rep(), 14  
rnorm(), 12, 15  
runif(), 14  
seq(), 14  
smooth.spline(), 19, 43, 44  
sqrt(), 4  
stripchart(), 45  
title(), 38  
which(), 8  
with(), 19, 44  
4sd-rule, 62  
  
assignment, 4  
  
covariates, 67  
  
explanatory variables, 67  
  
Hilbert matrix, 73  
  
linear model, 71  
logical negation, 9  
  
model matrix, 70, 71  
  
normal equations, 71, 73  
  
object, 4  
OLS, 11  
ordinary least squares, 11  
  
parameters, 65--67  
polynomial regression, 68  
predicted values, 72  
predictors, 67  
  
residuals, 71, 72  
response, 67  
  
subset(), 11  
  
variable, 4  
vectorized, 7  
  
z-score, 61