

Pedro J. Aphalo

Learn R

As a Language

Contents

List of Figures	vii
1 The R language: “Verbs” and “nouns” for statistics	1
1.1 Aims of this chapter	1
1.2 Statistical summaries	2
1.3 Distributions	3
1.3.1 Density from parameters	4
1.3.2 Probabilities from parameters and quantiles	4
1.3.3 Quantiles from parameters and probabilities	5
1.3.4 “Random” draws from a distribution	6
1.4 “Random” sampling	7
1.5 Correlation	8
1.5.1 Pearson’s r	8
1.5.2 Kendall’s τ and Spearman’s ρ	9
1.6 Model fitting in R	10
1.7 Fitting linear models	11
1.7.1 Regression	11
1.7.2 Analysis of variance, ANOVA	20
1.7.3 Analysis of covariance, ANCOVA	24
1.8 Generalized linear models	24
1.9 Non-linear regression	27
1.10 Model formulas	29
1.11 Time series	38
1.12 Multivariate statistics	41
1.12.1 Multivariate analysis of variance	41
1.12.2 Principal components analysis	42
1.12.3 Multidimensional scaling	44
1.12.4 Cluster analysis	46
1.13 Further reading	48
Bibliography	49
General index	51
Index of R names by category	53
Alphabetic index of R names	55



List of Figures



1

The R language: “Verbs” and “nouns” for statistics

The purpose of computing is insight, not numbers.

Richard W. Hamming
Numerical Methods for Scientists and Engineers, 1987

1.1 Aims of this chapter

This chapter aims to give the reader an introduction to the approach used in base R for the computation of statistical summaries, the fitting of models to observations and tests of hypothesis. This chapter does *not* explain data analysis methods, statistical principles or experimental designs. There are many good books on the use of R for different kinds of statistical analyses (see further reading on page 48) but most of them tend to focus on specific statistical methods rather than on the commonalities among them. Although base R's model fitting functions target specific statistical procedures, they use a common approach to model specification and for returning the computed estimates and test outcomes. This approach, also followed by many contributed extension packages, can be considered as part of the philosophy behind the R language. In this chapter you will become familiar with the approaches used in R for calculating statistical summaries, generating (pseudo-)random numbers, sampling, fitting models and carrying out tests of significance. We will use linear correlation, *t*-test, linear models, generalized linear models, non-linear models and some simple multivariate methods as examples. The focus is on how to specify statistical models, contrasts and observations, how to access different components of the objects returned by the corresponding fit and summary functions, and how to use these extracted components in further computations or for customized printing and formatting.

1.2 Statistical summaries

Being the main focus of the R language in data analysis and statistics, R provides functions for both simple and complex calculations, going from means and variances to fitting very complex models. Below are examples of functions implementing the calculation of the frequently used data summaries mean or average (`mean()`), variance (`var()`), standard deviation (`sd()`), median (`median()`), mean absolute deviation (`mad()`), mode (`mode()`), maximum (`max()`), minimum (`min()`), range (`range()`), quantiles (`quantile()`), length (`length()`), and all-encompassing summaries (`summary()`). All these methods accept numeric vectors and matrices as an argument. Some of them also have definitions for other classes such as data frames in the case of `summary()`. (The R language does not define a function for calculation of the standard error of the mean. Please, see section ?? on page ?? for how to define your own.)

```
x <- 1:20
mean(x)
var(x)
sd(x)
median(x)
mad(x)
mode(x)
max(x)
min(x)
range(x)
quantile(x)
length(x)
summary(x)
```



In contrast to many other examples in this book, the summaries computed with the code in the previous chunk are not shown. You should *run* them, using vector `x` as defined above, and then play with other real or artificial data that you may find interesting.

By default, if the argument contains `NA`s these functions return `NA`. The logic behind this is that if one value exists but is unknown, the true result of the computation is unknown (see page ?? for details on the role of `NA` in R). However, an additional parameter called `na.rm` allows us to override this default behavior by requesting any `NA` in the input to be removed (or discarded) before calculation,

```
x <- c(1:20, NA)
mean(x)
## [1] NA

mean(x, na.rm = TRUE)
## [1] 10.5
```

Other more advanced functions are also available, such as `boxplot.stats()` that computes the values needed to draw a boxplot.

i In many cases you will want to compute statistical summaries by group or treatment in addition or instead of for a whole data set or vector. See section ?? on page ?? for details on how to compute summaries of data stored in data frames.

1.3 Distributions

Density, distribution functions, quantile functions and generation of pseudo-random values for several different distributions are part of the R language. Entering `help(Distributions)` at the R prompt will open a help page describing all the distributions available in base R. For each distribution the different functions contain the same “root” in their names: `norm` for the normal distribution, `unif` for the uniform distribution, and so on. The “head” of the name indicates the type of values returned: “d” for density, “q” for quantile, “r” (pseudo-)random draws, and “p” for probabilities (Table 1.1).

TABLE 1.1

Theoretical probability distributions in R. Partial list of base R functions related to probability distributions. The full list can be obtained by executing the command `help(Distributions)`.

Distribution	symbol	density	P	quantiles	draws
normal	N	<code>dnorm()</code>	<code>pnorm()</code>	<code>qnorm()</code>	<code>rnorm()</code>
Student's	t	<code>dt()</code>	<code>pt()</code>	<code>qt()</code>	<code>rt()</code>
F	F	<code>df()</code>	<code>pf()</code>	<code>qf()</code>	<code>rf()</code>
binomial	B	<code>dbinom()</code>	<code>pbinom()</code>	<code>qbinom()</code>	<code>rbinom()</code>
multinomial	M	<code>dmultinom()</code>	<code>pmultinom()</code>	<code>qmultinom()</code>	<code>rmultinom()</code>
Poisson		<code>dpois()</code>	<code>ppois()</code>	<code>qpois()</code>	<code>rpois()</code>
X-squared	X^2	<code>dchisq()</code>	<code>pchisq()</code>	<code>qchisq()</code>	<code>rchisq()</code>
log-normal		<code>dlnorm()</code>	<code>plnorm()</code>	<code>qlnorm()</code>	<code>rlnorm()</code>
uniform		<code>dunif()</code>	<code>punif()</code>	<code>qunif()</code>	<code>runif()</code>

Theoretical distributions are defined by mathematical functions that accept parameters that control the exact shape and location. In the case of the Normal distribution, these parameters are the *mean* controlling location and (standard deviation) (or its square, the *variance*) controlling the spread around the center of the distribution. The four different functions differ in which values are calculated (the unknowns) and which values are supplied as arguments (the known inputs).

In what follows we use the normal distribution as an example, but with differences in their parameters, the functions for other theoretical distributions follow a similar naming pattern.

1.3.1 Density from parameters

To obtain a single point from the distribution curve we pass a vector of length one as an argument for x .

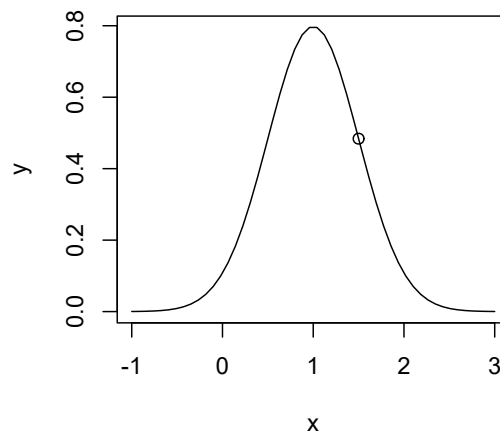
```
dnorm(x = 1.5, mean = 1, sd = 0.5)
## [1] 0.4839414
```

To obtain multiple values we can pass a longer vector as an argument.

```
dnorm(x = seq(from = -1, to = 1, length.out = 5), mean = 1, sd = 0.5)
## [1] 0.0002676605 0.0088636968 0.1079819330 0.4839414490 0.7978845608
```

With 50 equally spaced values for x we can plot a line (`type = "l"`) that shows that the 50 generated data points give the illusion of a continuous curve. We also add a point showing the value for $x = 1.5$ calculated above.

```
my.x <- seq(from = -1, to = 3, length.out = 50)
my.data <- data.frame(x = my.x,
                      y = dnorm(x = my.x, mean = 1, sd = 0.5))
plot(y~x, data = my.data, type = "l")
points(x = 1.5, y = dnorm(x = 1.5, mean = 1, sd = 0.5))
```



1.3.2 Probabilities from parameters and quantiles


If we have a known quantile value we can look up the corresponding p -value from the Normal distribution, i.e., the area under the curve, either to the right or to the left of a given value of x . When working with observations, the quantile, mean and standard deviation are in most cases computed from the same observations under the null hypothesis. In the example below, we use invented values for all parameters q , the quantile, $mean$, and sd , the standard deviation.

```
pnorm(q = 4, mean = 0, sd = 1)
## [1] 0.9999683

pnorm(q = 4, mean = 0, sd = 1, lower.tail = FALSE)
## [1] 3.167124e-05

pnorm(q = 4, mean = 0, sd = 4, lower.tail = FALSE)
## [1] 0.1586553

pnorm(q = c(2, 4), mean = 0, sd = 1, lower.tail = FALSE)
## [1] 2.275013e-02 3.167124e-05
```

 In tests of significance, empirical z -values and t -values are computed by subtracting from the observed mean for one group or raw quantile, the “expected” mean (possibly a hypothesized theoretical value, the mean of a control condition used as reference, or the mean computed over all treatments under the assumption of no effect of treatments) and then dividing by the standard deviation. Consequently, the p -values corresponding to these empirical z -values and t -values need to be looked up using `mean = 0` and `sd = 1` when calling `pnorm()` or `pt()` respectively. These frequently used values are the defaults.


1.3.3 Quantiles from parameters and probabilities

The reverse computation from that in the previous section is to obtain the quantile corresponding to a known p -value or area under one of the tails of the distribution curve. These quantiles are equivalent to the values in the tables of precalculated quantiles used in earlier times to assess significance with statistical tests.

```
qnorm(p = 0.01, mean = 0, sd = 1)
## [1] -2.326348

qnorm(p = 0.05, mean = 0, sd = 1)
## [1] -1.644854

qnorm(p = 0.05, mean = 0, sd = 1, lower.tail = FALSE)
## [1] 1.644854
```

 Quantile functions like `qnorm()` and probability functions like `pnorm()` always do computations based on a single tail of the distribution, even though it is possible to specify which tail we are interested in. If we are interested in obtaining simultaneous quantiles for both tails, we need to do this manually. If we are aiming at quantiles for $P = 0.05$, we need to find the quantile for each tail based on $P/2 = 0.025$.

```
qnorm(p = 0.025, mean = 0, sd = 1)
## [1] -1.959964

qnorm(p = 0.025, mean = 0, sd = 1, lower.tail = FALSE)
## [1] 1.959964
```

We see above that in the case of a symmetric distribution like the Normal, the quantiles in the two tails differ only in sign. This is not the case for asymmetric distributions.

When calculating a p -value from a quantile in a test of significance, we need to first decide whether a two-sided or single-sided test is relevant, and in the case of a single sided test, which tail is of interest. For a two-sided test we need to multiply the returned value by 2.

```
pnorm(q = 4, mean = 0, sd = 1) * 2
## [1] 1.999937
```


1.3.4 “Random” draws from a distribution


True random sequences can only be generated by physical processes. All “pseudo-random” sequences of numbers generated by computation are really deterministic although they share some properties with true random sequences (e.g., in relation to autocorrelation).

It is possible to compute not only pseudo-random draws from a uniform distribution but also from the Normal, t , F and other distributions. In each case, the probability with which different values are “drawn” approximates the probabilities set by the corresponding theoretical distribution. Parameter `n` indicates the number of values to be drawn, or its equivalent, the length of the vector returned.

```
rnorm(5)
## [1] -0.8248801  0.1201213 -0.4787266 -0.7134216  1.1264443

rnorm(n = 10, mean = 10, sd = 2)
## [1] 12.394190  9.697729  9.212345 11.624844 12.194317 10.257707 10.082981
## [8] 10.268540 10.792963  7.772915
```

 Edit the examples in sections 1.3.2, 1.3.3 and 1.3.4 to do computations based on different distributions, such as Student’s t , F or uniform.

 It is impossible to generate truly random sequences of numbers by means of a deterministic process such as a mathematical computation. “Random numbers” as generated by R and other computer programs are *pseudo random numbers*, long deterministic series of numbers that resemble random draws. Random number generation uses a *seed* value that determines where in the series we start. The usual way of automatically setting the value of the seed is to take the milliseconds or similar rapidly changing set of digits from the real time clock of the computer. However, in cases when we wish to repeat a calculation using the same series of pseudo-random values, we can use `set.seed()` with an arbitrary integer as an argument to reset the generator to the same point in the underlying (deterministic) sequence.



Execute the statement `rnorm(3)` by itself several times, paying attention to the values obtained. Repeat the exercise, but now executing `set.seed(98765)` immediately before each call to `rnorm(3)`, again paying attention to the values obtained. Next execute `set.seed(98765)`, followed by `c(rnorm(3), rnorm(3))`, and then execute `set.seed(98765)`, followed by `rnorm(6)` and compare the output. Repeat the exercise using a different argument in the call to `set.seed()`. analyze the results and explain how `setseed()` affects the generation of pseudo-random numbers in R.

1.4 “Random” sampling

In addition to drawing values from a theoretical distribution, we can draw values from an existing set or collection of values. We call this operation (pseudo-)random sampling. The draws can be done either with replacement or without replacement. In the second case, all draws are taken from the whole set of values, making it possible for a given value to be drawn more than once. In the default case of not using replacement, subsequent draws are taken from the values remaining after removing the values chosen in earlier draws.

```
sample(x = LETTERS)
## [1] "Z" "N" "Y" "R" "M" "E" "W" "J" "H" "G" "U" "O" "S" "T" "L" "F" "X" "P" "K"
## [20] "V" "D" "A" "B" "C" "I" "Q"

sample(x = LETTERS, size = 12)
## [1] "M" "S" "L" "R" "B" "D" "Q" "W" "V" "N" "J" "P"

sample(x = LETTERS, size = 12, replace = TRUE)
## [1] "K" "E" "V" "N" "A" "Q" "L" "C" "T" "L" "H" "U"
```

In practice, pseudo-random sampling is useful when we need to select subsets of observations. One such case is assigning treatments to experimental units in an experiment or selecting persons to interview in a survey. Another use is in bootstrapping to estimate variation in parameter estimates using empirical distributions.

As described in section ?? on page ??, data frames are commonly used to store one observation per row. To sample a subset of rows we need to generate a random set of indices to use with the extraction operator (`[]`). In the final example we sample four rows from data frame `cars` included in R. These data consist of stopping distances for cars moving at different speeds as described in the documentation available by entering `help(cars)`.

```
cars[sample(x = 1:nrow(cars), size = 4), ]
##   speed dist
## 33    18   56
## 31    17   50
## 50    25   85
## 36    19   36
```



Consult the documentation of `sample()` and explain why the code below is equivalent to that in the example immediately above.

```
cars[sample(x = nrow(cars), size = 4), ]
```

1.5 Correlation

Both parametric (Pearson’s) and non-parametric robust (Spearman’s and Kendall’s) methods for the estimation of the (linear) correlation between pairs of variables are available in base R. The different methods are selected by passing arguments to a single function. While Pearson’s method is based on the actual values of the observations, non-parametric methods are based on the ordering or rank of the observations, and consequently less affected by observations with extreme values.

1.5.1 Pearson’s r

Function `cor()` can be called with two vectors of the same length as arguments. In the case of the parametric Pearson method, we do not need to provide further arguments as this method is the default one. We use data set `cars`.

```
cor(x = cars$speed, y = cars$dist)
## [1] 0.8068949
```

It is also possible to pass a data frame (or a matrix) as the only argument. When the data frame (or matrix) contains only two columns, the returned value is equivalent to that of passing the two columns individually as vectors.

```
cor(cars)
##           speed      dist
## speed 1.0000000 0.8068949
## dist  0.8068949 1.0000000
```

When the data frame or matrix contains more than two numeric vectors, the returned value is a matrix of estimates of pairwise correlations between columns. We here use `rnorm()` described above to create a long vector of pseudo-random values drawn from the Normal distribution and `matrix()` to convert it into a matrix with three columns (see page ?? for details about R matrices).

```
my.mat <- matrix(rnorm(54), ncol = 3,
                 dimnames = list(rows = 1:18, cols = c("A", "B", "C")))
cor(my.mat)
##           A           B           C
## A 1.0000000 0.1899797 0.07591003
## B 0.18997966 1.0000000 0.36800323
## C 0.07591003 0.3680032 1.00000000
```


 Modify the code in the chunk immediately above constructing a matrix with six columns and then computing the correlations.

While `cor()` returns an estimate for r the correlation coefficient, `cor.test()` also computes the t -value, p -value, and confidence interval for the estimate.

```
cor.test(x = cars$speed, y = cars$dist)
##
## Pearson's product-moment correlation
##
## data: cars$speed and cars$dist
## t = 9.464, df = 48, p-value = 1.49e-12
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
## 0.6816422 0.8862036
## sample estimates:
## cor
## 0.8068949
```

As described below for model fitting, `cor.test()` also accepts a formula plus data as arguments. The code below is equivalent to that above. See section 1.10 on page 29 for more information on the use of model formulas in R.

```
cor.test(formula = ~ speed + dist, data = cars)
```

 Functions `cor()` and `cor.test()` return R objects, that when using R interactively get automatically “printed” on the screen. One should be aware that `print()` methods do not necessarily display all the information contained in an R object. This is almost always the case for complex objects like those returned by R functions implementing statistical tests. As with any R object we can save the result of an analysis into a variable. As described in section ?? on page ?? for lists, we can peek into the structure of an object with method `str()`. We can use `class()` and `attributes()` to extract further information. Run the code in the chunk below to discover what is actually returned by `cor()`.

```
a <- cor(cars)
class(a)
attributes(a)
str(a)
```

Methods `class()`, `attributes()` and `str()` are very powerful tools that can be used when we are in doubt about the data contained in an object and/or how it is structured. Knowing the structure allows us to retrieve the data members directly from the object when predefined extractor methods are not available.

1.5.2 Kendall's τ and Spearman's ρ

We use the same functions as for Pearson's r but explicitly request the use of one of these methods by passing an argument.

```
cor(x = cars$speed, y = cars$dist, method = "kendall")
## [1] 0.6689901

cor(x = cars$speed, y = cars$dist, method = "spearman")
## [1] 0.8303568
```

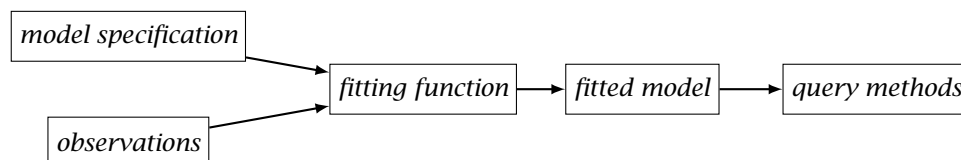
Function `cor.test()`, described above, also allows the choice of method with the same syntax as shown for `cor()`.



Repeat the exercise in the playground immediately above, but now using non-parametric methods. How does the information stored in the returned `matrix` differ depending on the method, and how can we extract information about the method used for calculation of the correlation from the returned object.

1.6 Model fitting in R


The general approach to model fitting in R is to separate the actual fitting of a model from the inspection of the fitted model. A model fitting function minimally requires a description of the model to fit, as a model `formula` and a data frame or vectors with the data or observations to which to fit the model. These functions in R return a model fit object. This object contains the data, the model formula, call and the result of fitting the model. To inspect this model several methods are available. In the diagram we show the overall approach used fit models to data.



Models are described using model formulas such as $y \sim x$ which we read as y is explained by x . We use lhs (left-hand-side) and rhs (right-hand-side) to signify all terms to the left and right of the tilde (\sim), respectively (`<lhs> ~ <rhs>`). Model formulas are used in different contexts: fitting of models, plotting, and tests like t -test. The syntax of model formulas is consistent throughout base R and numerous independently developed packages. However, their use is not universal, and several packages extend the basic syntax to allow the description of specific types of models. As most things in R, model formulas are objects and can be stored in variables. See section 1.10 on page 29 for a detailed discussion of model formulas.


Although there is some variation, especially for fitted model classes defined in extension packages, in most cases the *query functions* bulked together in the rightmost box in the diagram include methods `summary()`, `anova()` and `plot()`, with several other methods such as `coef()`, `residuals()`, `fitted()`, `predict()`, `AIC()`, `BIC()` usually also available. Additional methods may be available. However, as model fit objects are derived from class `list`, these and other components can be

extract or computed programmatically when needed. Consequently, the examples in this chapter can be adapted to the fitting of types of models not described here.

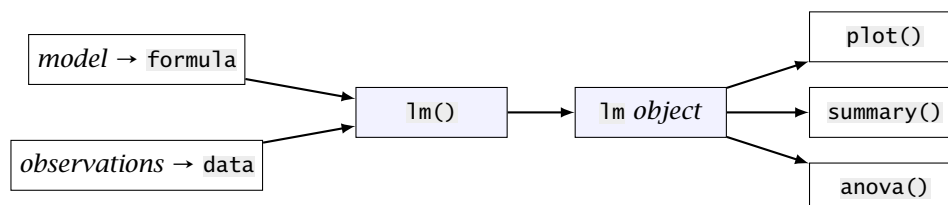
 Fitted model objects in R are self contained and include a copy of the data to which the model was fit, as well as residuals and possibly even intermediate results of computations. Although this can make the size of these objects large, it allows querying and even updating them in the absence of the data in the current R workspace.

1.7 Fitting linear models

Function `lm()` is used to fit linear models. If the explanatory variable is continuous, the fit is a regression. If the explanatory variable is a factor, the fit is an analysis of variance (ANOVA) in broad terms. However, there is another meaning of ANOVA, referring only to the tests of significance rather to an approach to model fitting. Consequently, rather confusingly, results for tests of significance for fitted parameter estimates can both in the case of regression and ANOVA, be presented in an ANOVA table. In this second, stricter meaning, ANOVA means a test of significance based on the ratios between pairs of variances.

 If you do not clearly remember the difference between numeric vectors and factors, or how they can be created, please, revisit chapter ?? on page ??.

The generic diagram from the previous section redrawn to show a linear model fit, done with function `lm()` where the non-filled boxes represent what is in common with the fitting of other types of models, and the filled ones what is specific to `lm()`. The diagram includes only the three most frequently used query methods and both response variables and explanatory variables are included under *observations*.



1.7.1 Regression

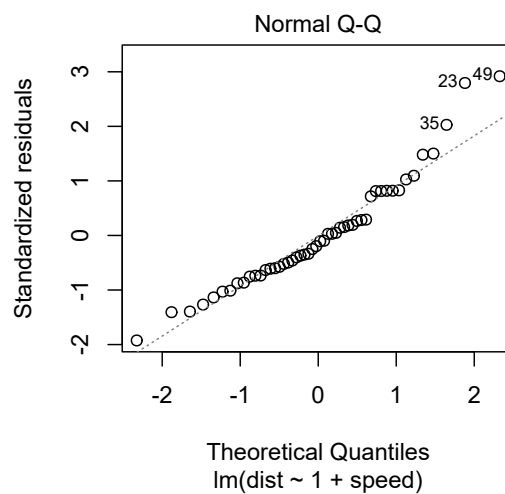
In this section we continue using the `cars` data set, which contains to numeric variables.

We fit a simple linear model $y = \alpha \cdot 1 + \beta \cdot x$ where y corresponds to stopping distance (`dist`) and x to initial speed (`speed`). Such a model is formulated in R as `dist ~ 1 + speed`. We save the fitted model as `fm1` (a mnemonic for fitted-model one).


```
fm1 <- lm(dist ~ 1 + speed, data=cars)
class(fm1)
## [1] "lm"
```

The next step is diagnosis of the fit. Are assumptions of the linear model procedure used reasonably close to being fulfilled? In R it is most common to use plots to this end. We show here only one of the four plots normally produced. This quantile vs. quantile plot allows us to assess how much the residuals deviate from being normally distributed.

```
plot(fm1, which = 2)
```



In the case of a regression, calling `summary()` with the fitted model object as argument is most useful as it provides a table of coefficient estimates and their errors. Remember that as is the case for most R functions, the value returned by `summary()` is printed when we call this method at the R prompt.

```
summary(fm1)
##
## Call:
## lm(formula = dist ~ 1 + speed, data = cars)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601  0.0123 *
## speed         3.9324     0.4155   9.464 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF, p-value: 1.49e-12
```

Let's look at the printout of the summary, section by section. Under "Call:" we find, `dist ~ 1 + speed` or the specification of the model fitted, plus the data used. Under "Residuals:" we find the extremes, quartiles and median of the residuals, or deviations between observations and the fitted line. Under "Coefficients:" we find the estimates of the model parameters and their variation plus corresponding t -tests. At the end of the summary there is information on degrees of freedom and overall coefficient of determination (R^2).

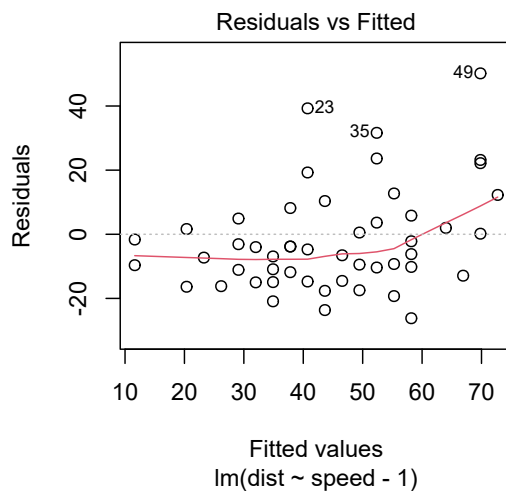
If we return to the model formulation, we can now replace α and β by the estimates obtaining $y = -17.6 + 3.93x$. Given the nature of the problem, we *know based on first principles* that stopping distance must be zero when speed is zero. This suggests that we should not estimate the value of α but instead set $\alpha = 0$, or in other words, fit the model $y = \beta \cdot x$.

However, in R models, the intercept is always implicitly included, so the model fitted above can be formulated as `dist ~ speed`—i.e., a missing `+ 1` does not change the model. To exclude the intercept from the previous model, we need to specify it as `dist ~ speed - 1` (or its equivalent `dist ~ speed + 0`), resulting in the fitting of a straight line passing through the origin ($x = 0, y = 0$).

```
fm2 <- lm(dist ~ speed - 1, data = cars)
summary(fm2)
##
## Call:
## lm(formula = dist ~ speed - 1, data = cars)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -26.183 -12.637  -5.455   4.590  50.181
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## speed      2.9091      0.1414   20.58  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 16.26 on 49 degrees of freedom
## Multiple R-squared:  0.8963, Adjusted R-squared:  0.8942
## F-statistic: 423.5 on 1 and 49 DF,  p-value: < 2.2e-16
```

Now there is no estimate for the intercept in the summary, only an estimate for the slope.

```
plot(fm2, which = 1)
```



The equation of the second fitted model is $y = 2.91x$, and from the residuals, it can be seen that it is inadequate, as the straight line does not follow the curvature of the relationship between `dist` and `speed`.



You will now fit a second-degree polynomial, a different linear model: $y = \alpha \cdot 1 + \beta_1 \cdot x + \beta_2 \cdot x^2$. The function used is the same as for linear regression, `lm()`. We only need to alter the formulation of the model. The identity function `I()` is used to protect its argument from being interpreted as part of the model formula. Instead, its argument is evaluated beforehand and the result is used as the, in this case second, explanatory variable.

```
fm3 <- lm(dist ~ speed + I(speed^2), data = cars)
plot(fm3, which = 3)
summary(fm3)
anova(fm3)
```

The “same” fit using an orthogonal polynomial can be specified using function `poly()`. Polynomials of different degrees can be obtained by supplying as the second argument to `poly()` the corresponding positive integer value. In this case, the different terms of the polynomial are bulked together in the summary.

```
fm3a <- lm(dist ~ poly(speed, 2), data = cars)
summary(fm3a)
anova(fm3a)
```

We can also compare two model fits using `anova()`, to test whether one of the models describes the data better than the other. It is important in this case to take into consideration the nature of the difference between the model formulas, most importantly if they can be interpreted as nested—i.e., interpreted as a base model vs. the same model with additional terms.

```
anova(fm2, fm1)
```

Three or more models can also be compared in a single call to `anova()`. However, be careful, as the order of the arguments matters.

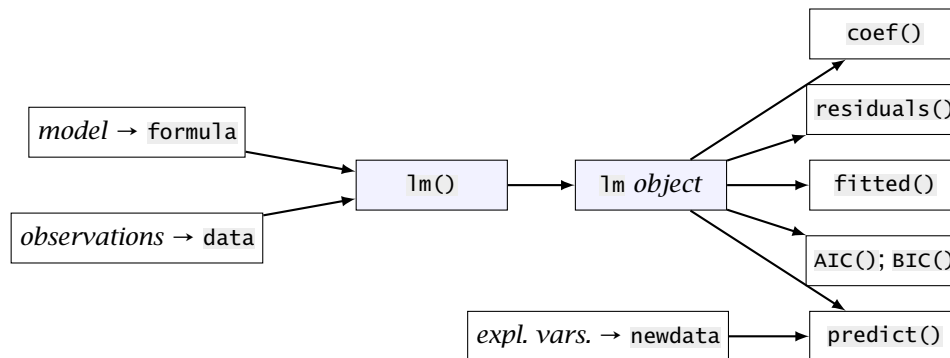
```
anova(fm2, fm3, fm3a)
anova(fm2, fm3a, fm3)
```


We can use different criteria to choose the “best” model: significance based on p -values or information criteria (AIC, BIC). AIC (Akaike’s “An Information Criterion”) and BIC (“Bayesian Information Criterion” = SBC, “Schwarz’s Bayesian criterion”) that penalize the resulting “goodness” based on the number of parameters in the fitted model. In the case of AIC and BIC, a smaller value is better, and values returned can be either positive or negative, in which case more negative is better. Estimates for both BIC and AIC are returned by `anova()`, and on their own by `BIC()` and `AIC()`


```
BIC(fm2, fm1, fm3, fm3a)
AIC(fm2, fm1, fm3, fm3a)
```

Once you have run the code in the chunks above, you will be able to see that these three criteria do not necessarily agree on which is the “best” model. Find in the output p -value, BIC and AIC estimates, for the different models and conclude which model is favored by each of the three criteria. In addition you will notice that the two different formulations of the quadratic polynomial are equivalent.

Additional methods give easy access to different components of fitted models: `vcov()` returns the variance-covariance matrix, `coef()` and its alias `coefficients()` return the estimates for the fitted model coefficients, `fitted()` and its alias `fitted.values()` extract the fitted values, and `resid()` and its alias `residuals()` the corresponding residuals (or deviations). Less frequently used accessors are `effects()`, `terms()`, `model.frame()` and `model.matrix()`. The diagram below shows how some of these methods fit in the model fitting workflow.



 Familiarize yourself with these extraction and summary methods by reading their documentation and use them to explore `fm1` fitted above or model fits to other data of your interest.

 The objects returned by model fitting functions are rather complex and contain the full information, including the data to which the model was fit to. The different functions described above, either extract parts of the object or do additional calculations and formatting based on them. There are different specializations of these methods which are called depending on the class of the model-fit object. (See section ?? on page ??.)

```
class(fm1)
## [1] "lm"

names(fm1)
## [1] "coefficients" "residuals" "effects" "rank"
## [5] "fitted.values" "assign" "qr" "df.residual"
## [9] "xlevels" "call" "terms" "model"
```

We rarely need to manually explore the structure of these model-fit objects when using R interactively. In contrast, when including model fitting in scripts or package code, the need to efficiently extract specific members from them happens more frequently. As with any other R object we can use `str()` to explore them. As this prints as a long text, we call `str()` as an example, only with one component of the `fm1` object and leave to the reader the task of exploring the remaining ones.

```
str(fm1$call)
## language lm(formula = dist ~ 1 + speed, data = cars)
```

We frequently only look at the output of `anova()` and `summary()` as implicitly displayed by `print()`. However, both `anova()` and `summary()` return complex objects, derived from `list`, containing additional component members not displayed by the matching `print()` methods.

The class of these objects depends on the class of the model fit object.

```
class(summary(fm1))
## [1] "summary.lm"
```

Knowing that these objects contain additional information can be very useful, for example, when we want to display the results from the fit in a different format or to implement additional tests or computations. Once again we use `str()` to look at the structure.

Once we know the structure of the object and the names of members, we can simply extract them using the usual R rules for member extraction. This is useful when we need to do further calculations or when we want to present these values in a different format than that provided by the `print()` method, for example to add a textual annotation to a plot.

The coefficients estimates in the summary are accompanied by estimates for the corresponding standard errors, *t*-value and *P*-value estimates, while in the model object `fm1` the additional estimates are not included.

```

coef(fm1)
## (Intercept)      speed
## -17.579095      3.932409

str(fm1$coefficients)
## Named num [1:2] -17.58 3.93
## - attr(*, "names")= chr [1:2] "(Intercept)" "speed"

print(summary(fm1)$coefficients)
##              Estimate Std. Error   t value    Pr(>|t|)
## (Intercept) -17.579095  6.7584402 -2.601058 1.231882e-02
## speed        3.932409  0.4155128  9.463990 1.489836e-12

str(summary(fm1)$coefficients)
## num [1:2, 1:4] -17.579 3.932 6.758 0.416 -2.601 ...
## - attr(*, "dimnames")=List of 2
## ..$ : chr [1:2] "(Intercept)" "speed"
## ..$ : chr [1:4] "Estimate" "Std. Error" "t value" "Pr(>|t|)"

```

As an example of the use of values extracted from the `summary.lm` object, we test if the slope from a linear regression fit deviates significantly from a constant value different from the usual zero. A null hypothesis of zero for the slope tests for the presence of an “effect” of an explanatory variable, which is usually of interest in an experiment. In contrast, when testing for deviations from a calibration by comparing two instruments or an instrument and a reference, a null hypothesis of one for the slope will test for deviations from the true readings. In some cases, we may want to test if the estimate for a parameter exceeds some other value, such as acceptable product tolerances. In other cases, when comparing the effectiveness of interventions we may be interested to test if a new approach surpasses that in current use by at least a specific margin. There exist many situations where the question of interest is not that an effect deviates from zero. Furthermore, when dealing with big data, very small deviations from zero can be statistically significant but biologically or practically irrelevant. In such case we can set the smallest response that is of interest, instead of zero, as the null hypothesis in the test.

The examples above, using `anova()` and `summary()` are for a null hypothesis of slope = 0. Here we do the equivalent test with a null hypothesis of slope = 1. The procedure is applicable to any constant value as a null hypothesis for any of the fitted parameter estimates. However, for the *P*-value estimates to be valid, the hypotheses should be set in advance of the study, i.e., independent of the observations used for the test. The examples use a two-sided test. In some cases, a single-sided test should be used (e.g., if its known a priori because of physical reasons that deviation is possible only in one direction away from the null hypothesis, or because only one direction of response is of interest).

To estimate the *t*-value we need an estimate for the parameter value and an estimate of the standard error for this estimate, and the degrees of freedom. We can extract all these values from the summary of a fitted model object.

```


est.slope.value <- summary(fm1)$coefficients["speed", "Estimate"]
est.slope.se <- summary(fm1)$coefficients["speed", "Std. Error"]
degrees.of.freedom <- summary(fm1)$df[2]

```

A new t -value is computed based on the difference between the value of the null hypothesis and the value for the parameter estimated from the observations. A new probability estimate is computed based on computed t -value, or quantile, and the t distribution with matching degrees of freedom with a call to `pt()` (see section 1.3 on page 3.) For a two-tails test we multiply by two the one-tail P estimate.

```
hyp.null <- 1
t.value <- (est.slope.value - hyp.null) / est.slope.se
p.value <- 2 * pt(q = t.value, df = degrees.of.freedom, lower.tail = FALSE)
cat("slope =", signif(est.slope.value, 3), "with s.e. =", signif(est.slope.se, 3),
    "\nt.value =", signif(t.value, 3), "and P-value =", signif(p.value, 3))
## slope = 3.93 with s.e. = 0.416
## t.value = 7.06 and P-value = 6.01e-09
```


This example is for a linear model fitted with function `lm()` but the same approach can be applied to other model fit procedures for which parameter estimates and their corresponding standard error estimates can be extracted or computed.


 Check that the procedure above agrees with the output of `summary()` when we set `hyp.null <- 0` instead of `hyp.null <- 1`.

Modify the example above so as to test whether the intercept is significantly larger than 5 feet, doing a one-sided test.

Use `class(anova(fm1))` and `str(anova(fm1))` to explore the R object returned by the call `anova(fm1)`.

Method `predict()` uses the fitted model together with new data for the independent variables to compute predictions. As `predict()` accepts new data as input, it allows interpolation and extrapolation to values of the independent variables not present in the original data. In the case of fits of linear- and some other models, method `predict()` returns, in addition to the prediction, estimates of the confidence and/or prediction intervals. The new data must be stored in a data frame with columns using the same names for the explanatory variables as in the data used for the fit, a response variable is not needed and additional columns are ignored. (The explanatory variables in the new data can be either continuous or factors, but they must match in this respect those in the original data.)

 Method `predict()` is behind most plotting of lines corresponding to fitted models. For some types of models plotting is automated by ready available methods that both generate the predicted values and plot them. In other cases it is necessary to generate the predicted values with `predict()` and use these values as data input for a line-plotting method.

 Predict using both `fm1` and `fm2` the distance required to stop cars moving at 0, 5, 10, 20, 30, and 40 mph. Study the help page for the `predict` method for linear models (using `help(predict.lm)`). Explore the difference between "prediction" and "confidence" bands: why are they so different?

1.7.2 Analysis of variance, ANOVA

We use here the `InsectSprays` data set, giving insect counts in plots sprayed with different insecticides. In these data, `spray` is a factor with six levels.

The call is exactly the same as the one for linear regression, only the names of the variables and data frame are different. What determines that this is an ANOVA is that `spray`, the explanatory variable, is a **factor**.

```
data(InsectSprays)
is.numeric(InsectSprays$spray)
## [1] FALSE

is.factor(InsectSprays$spray)
## [1] TRUE

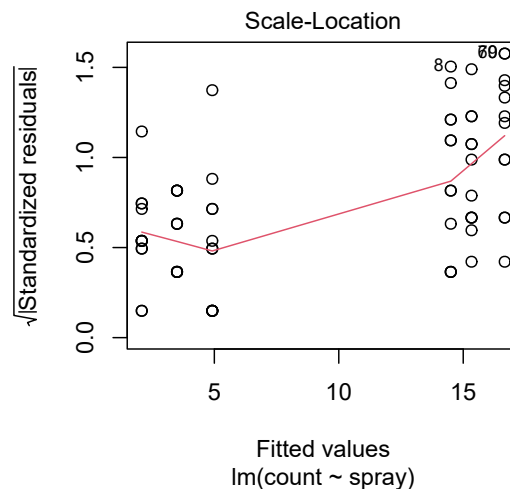
levels(InsectSprays$spray)
## [1] "A" "B" "C" "D" "E" "F"
```

We fit the model in exactly the same way as for linear regression; the difference is that we use a factor as the explanatory variable. By using a factor instead of a numeric vector, a different model matrix is built from an equivalent formula.

```
fm4 <- lm(count ~ spray, data = InsectSprays)
```


Diagnostic plots are obtained in the same way as for linear regression.


```
plot(fm4, which = 3)
```



In ANOVA we are mainly interested in testing hypotheses, and `anova()` provides the most interesting output. Function `summary()` can be used to extract parameter estimates. The default contrasts and corresponding p -values returned by `summary()` test hypotheses that have little or no direct interest in an analysis of variance. Function `aov()` is a wrapper on `lm()` that returns an object that by default when printed displays the output of `anova()`.

```
anova(fm4)
## Analysis of Variance Table
##
## Response: count
##          Df Sum Sq Mean Sq F value    Pr(>F)
## spray      5 2668.8   533.77   34.702 < 2.2e-16 ***
## Residuals 66 1015.2    15.38
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

 The defaults used for model fits and ANOVA calculations vary among programs. There exist different so-called “types” of sums of squares, usually called I, II, and III. In orthogonal designs the choice has no consequences, but differences can be important for unbalanced designs, even leading to different conclusions. R’s default, type I, is usually considered to suffer milder problems than type III, the default used by SPSS and SAS.

 The contrasts used affect the estimates returned by `coef()` and `summary()` applied to an ANOVA model fit. The default used in R is different to that used in some other programs (even different than in S). The default, `contr.treatment` uses the first level of the factor (assumed to be a control) as reference for estimation of coefficients and testing of their significance. Instead, `contr.sum` uses as reference the mean of all levels, i.e., using as condition that the sum of the coefficient estimates is equal to zero. Obviously this changes what the coefficients describe, and consequently also the estimated *p*-values, and most importantly how the result of the tests should be interpreted.

The most straightforward way of setting a different default for a whole series of model fits is by setting R option `contrasts`, which we here only print.

```
options("contrasts")
## $contrasts
##          unordered          ordered
## "contr.treatment"    "contr.poly"
```

The option is set to a named character vector of length two, with the first value, named `unordered` giving the name of the function used when the explanatory variable is an unordered factor (created with `factor()`) and the second value, named `ordered`, giving the name of the function used when the explanatory variable is an ordered factor (created with `ordered()`).


It is also possible to select the contrast to be used in the call to `aov()` or `lm()`.

```
fm4trea <- lm(count ~ spray, data = InsectSprays,
              contrasts = list(spray = contr.treatment))
fm4sum  <- lm(count ~ spray, data = InsectSprays,
              contrasts = list(spray = contr.sum))
```

Interpretation of any analysis has to take into account these differences and users should not be surprised if ANOVA yields different results in base R and SPSS or SAS given the different types of sums of squares used. The interpretation of

ANOVA on designs that are not orthogonal will depend on which type is used, so the different results are not necessarily contradictory even when different.

In `fm4trea` we used `contr.treatment()`, thus contrasts for individual treatments are done against `spray1` taking it as the control or reference, and can be inferred from the generated contrasts matrix. For this reason, there is no row for `spray1` in the summary table. Each of the rows `spray2` to `spray6` is a test comparing these treatments individually against `spray1`.

 Contrast are specified as matrices that are constructed by functions based on the number of levels in a factor. Constructor function `contr.treatment()` is the default in R for unordered factors, constructor `contr.SAS()` mimics the contrasts used in many SAS procedures, and `contr.helmert()` matches the default in S. Contrasts depend on the order of factor levels so it is crucial to ensure that the ordering in use yields the intended tests of significance for individual parameter estimates. (How to change the order of factor levels is explained in section ?? starting on page ??.)

```
contr.treatment(length(levels(InsectSprays$spray)))
##   2 3 4 5 6
## 1 0 0 0 0 0
## 2 1 0 0 0 0
## 3 0 1 0 0 0
## 4 0 0 1 0 0
## 5 0 0 0 1 0
## 6 0 0 0 0 1
```

```
summary(fm4trea)
##
## Call:
## lm(formula = count ~ spray, data = InsectSprays, contrasts = list(spray = contr.treatment))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -8.333 -1.958 -0.500  1.667  9.333
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  14.5000     1.1322  12.807  < 2e-16 ***
## sprayB       0.8333     1.6011   0.520   0.604
## sprayC     -12.4167     1.6011 -7.755 7.27e-11 ***
## sprayD     -9.5833     1.6011 -5.985 9.82e-08 ***
## sprayE    -11.0000     1.6011 -6.870 2.75e-09 ***
## sprayF       2.1667     1.6011  1.353  0.181
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.922 on 66 degrees of freedom
## Multiple R-squared:  0.7244, Adjusted R-squared:  0.7036
## F-statistic: 34.7 on 5 and 66 DF, p-value: < 2.2e-16
```


In `fm4sum` we used `contr.sum()`, thus contrasts for individual treatments are done differently, as can be inferred from the contrasts matrix. The sum is constrained to be zero, thus estimates for the last treatment level are determined by the sum of the previous ones, and not tested for significance.

```

contr.sum(length(levels(InsectSprays$spray)))
##      [,1] [,2] [,3] [,4] [,5]
## 1      1      0      0      0      0
## 2      0      1      0      0      0
## 3      0      0      1      0      0
## 4      0      0      0      1      0
## 5      0      0      0      0      1
## 6     -1     -1     -1     -1     -1

summary(fm4sum)
##
## Call:
## lm(formula = count ~ spray, data = InsectSprays, contrasts = list(spray = contr.sum))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -8.333 -1.958 -0.500  1.667  9.333
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   9.5000     0.4622  20.554 < 2e-16 ***
## spray1        5.0000     1.0335   4.838 8.22e-06 ***
## spray2        5.8333     1.0335   5.644 3.78e-07 ***
## spray3       -7.4167     1.0335  -7.176 7.87e-10 ***
## spray4       -4.5833     1.0335  -4.435 3.57e-05 ***
## spray5       -6.0000     1.0335  -5.805 2.00e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.922 on 66 degrees of freedom
## Multiple R-squared:  0.7244, Adjusted R-squared:  0.7036
## F-statistic: 34.7 on 5 and 66 DF, p-value: < 2.2e-16

```

 Explore how taking the last level as reference in `contr.SAS()` instead of the first one as in `contr.treatment()` affects the estimates. Reorder the levels of factor `spray` so that the test using `contr.SAS()` becomes equivalent to that obtained above with `contr.treatment()`. Consider why `contr.poly()` is the default for ordered factors and when `contr.helmert()` could be most useful.

In the case of contrasts, they always affect the parameter estimates independently of whether the experiment design is orthogonal or not. A different set of contrasts simply tests a different set of possible treatment effects. Contrasts, on the other hand, do not affect the table returned by `anova()` as this table does not deal with the effects of individual factor levels. It can also be seen that the overall estimates shown at the bottom of the summary table remain unchanged. In other words, what changes is how the total variation explained by the fitted model is partitioned into components to be tested for specific contributions to the overall model fit.

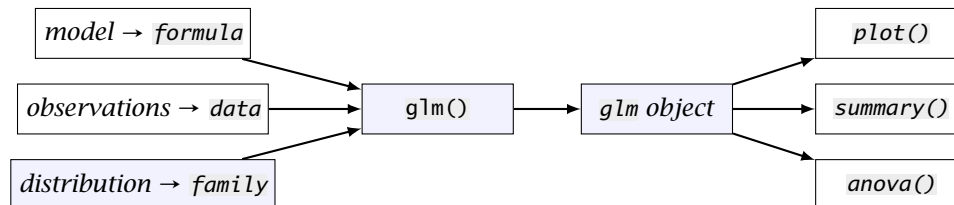
Contrasts and their interpretation are discussed in detail by Venables and Ripley (2002) and Crawley (2012).

1.7.3 Analysis of covariance, ANCOVA

When a linear model includes both explanatory factors and continuous explanatory variables, we may call it *analysis of covariance* (ANCOVA). The formula syntax is the same for all linear models and, as mentioned in previous sections, what determines the type of analysis is the nature of the explanatory variable(s). As the formulation remains the same, no specific example is given. The main difficulty of ANCOVA is in the selection of the covariate and the interpretation of the results of the analysis (e.g. Smith 1957).

1.8 Generalized linear models

Linear models make the assumption of normally distributed residuals. Generalized linear models, fitted with function `glm()` are more flexible, and allow the assumed distribution to be selected as well as the link function.



For the analysis of the `InsectSprays` data set above (section 1.7.2 on page 20), the Normal distribution is not a good approximation as count data deviates from it. This was visible in the quantile-quantile plot above.

For count data, GLMs provide a better alternative. In the example below we fit the same model as above, but we assume a quasi-Poisson distribution instead of the Normal. In addition to the model formula we need to pass an argument through `family` giving the error distribution to be assumed—the default for `family` is `gaussian` or Normal distribution.

```
fm10 <- glm(count ~ spray, data = InsectSprays, family = quasipoisson)
anova(fm10)
## Analysis of Deviance Table
##
## Model: quasipoisson, link: log
##
## Response: count
##
## Terms added sequentially (first to last)
##
##
##      Df Deviance Resid. Df Resid. Dev
## NULL              71      409.04
## spray   5       310.71      66       98.33
```

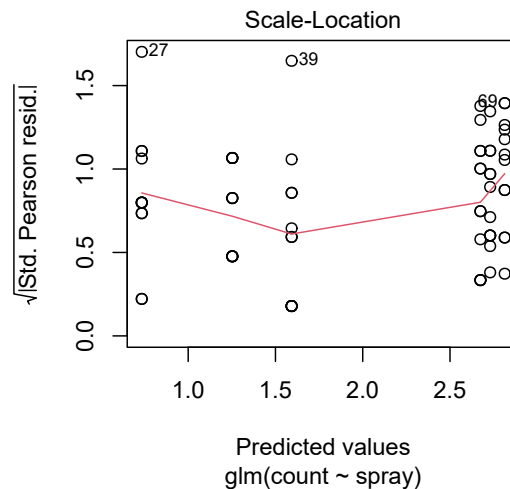
The printout from the `anova()` method for GLM fits has some differences to that for LM fits. By default, no significance test is computed, as a knowledgeable choice

is required depending on the characteristics of the model and data. We here use "F" as an argument to request an *F*-test.

```
anova(fm10, test = "F")
## Analysis of Deviance Table
##
## Model: quasipoisson, link: log
##
## Response: count
##
## Terms added sequentially (first to last)
##
##
##          Df Deviance Resid. Df Resid. Dev      F    Pr(>F)
## NULL                71      409.04
## spray    5      310.71      66      98.33 41.216 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Method `plot()` as for linear-model fits, produces diagnosis plots. We show as above the q-q-plot of residuals.

```
plot(fm10, which = 3)
```



We can extract different components similarly as described for linear models (see section 1.7 on page 11).


```
class(fm10)
## [1] "glm" "lm"

summary(fm10)
##
## Call:
## glm(formula = count ~ spray, family = quasipoisson, data = InsectSprays)
##
## Deviance Residuals:
```

```
##      Min      1Q   Median      3Q      Max
## -2.3852 -0.8876 -0.1482  0.6063  2.6922
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2.67415    0.09309  28.728 < 2e-16 ***
## sprayB       0.05588    0.12984   0.430  0.668
## sprayC      -1.94018    0.26263  -7.388 3.30e-10 ***
## sprayD      -1.08152    0.18499  -5.847 1.70e-07 ***
## sprayE      -1.42139    0.21110  -6.733 4.82e-09 ***
## sprayF       0.13926    0.12729   1.094  0.278
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for quasipoisson family taken to be 1.507713)
##
##      Null deviance: 409.041  on 71  degrees of freedom
## Residual deviance:  98.329  on 66  degrees of freedom
## AIC: NA
##
## Number of Fisher Scoring iterations: 5

head(residuals(fm10))
##           1           2           3           4           5           6
## -1.2524891 -2.1919537  1.3650439 -0.1320721 -0.1320721 -0.6768988

head(fitted(fm10))
##      1      2      3      4      5      6
## 14.5 14.5 14.5 14.5 14.5 14.5
```

 If we use `str()` or `names()` we can see that there are some differences with respect to linear model fits. The returned object is of a different class and contains some members not present in linear models. Two of these have to do with the iterative approximation method used, `iter` contains the number of iterations used and `converged` the success or not in finding a solution.

```
names(fm10)
## [1] "coefficients"      "residuals"        "fitted.values"
## [4] "effects"           "R"                 "rank"
## [7] "qr"                "family"            "linear.predictors"
## [10] "deviance"          "aic"               "null.deviance"
## [13] "iter"              "weights"           "prior.weights"
## [16] "df.residual"       "df.null"           "y"
## [19] "converged"         "boundary"          "model"
## [22] "call"              "formula"           "terms"
## [25] "data"              "offset"            "control"
## [28] "method"            "contrasts"         "xlevels"

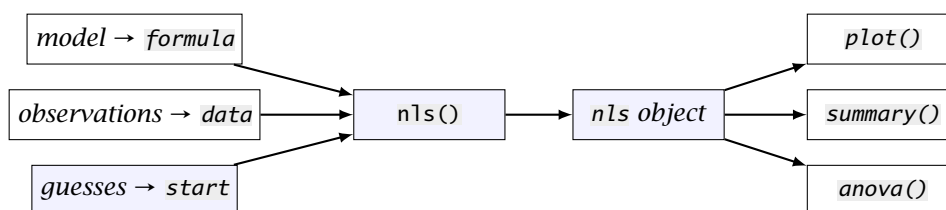
fm10$converged
## [1] TRUE

fm10$iter
## [1] 5
```

1.9 Non-linear regression

Function `nls()` is R's workhorse for fitting non-linear models. By *non-linear* it is meant non-linear *in the parameters* whose values are being estimated through fitting the model to data. This is different from the shape of the function when plotted—i.e., polynomials of any degree are linear models. In contrast, the Michaelis-Menten equation used in chemistry and the Gompertz equation used to describe growth are non-linear models in their parameters.

While analytical algorithms exist for finding estimates for the parameters of linear models, in the case of non-linear models, the estimates are obtained by approximation. For analytical solutions, estimates can always be obtained, except in infrequent pathological cases where reliance on floating point numbers with limited resolution introduces rounding errors that “break” mathematical algorithms that are valid for real numbers. For approximations obtained through iteration, cases when the algorithm fails to *converge* onto an answer are relatively common. Iterative algorithms attempt to improve an initial guess for the values of the parameters to be estimated, a guess frequently supplied by the user. In each iteration the estimate obtained in the previous iteration is used as the starting value, and this process is repeated one time after another. The expectation is that after a finite number of iterations the algorithm will converge into a solution that “cannot” be improved further. In real life we stop iteration when the improvement in the fit is smaller than a certain threshold, or when no convergence has been achieved after a certain maximum number of iterations. In the first case, we usually obtain good estimates; in the second case, we do not obtain usable estimates and need to look for different ways of obtaining them. When convergence fails, the first thing to do is to try different starting values and if this also fails, switch to a different computational algorithm. These steps usually help, but not always. Good starting values are in many cases crucial and in some cases “guesses” can be obtained using either graphical or analytical approximations.



For functions for which computational algorithms exist for “guessing” suitable starting values, R provides a mechanism for packaging the function to be fitted together with the function generating the starting values. These functions go by the name of *self-starting functions* and relieve the user from the burden of guessing and supplying suitable starting values. The self-starting functions available in R are `SSasymp()`, `SSasympOff()`, `SSasympOrig()`, `SSbiexp()`, `SSfol()`, `SSfp1()`, `SSgomptz()`, `SSlogis()`, `SSmicmen()`, and `SSweibull()`. Function `selfstart()` can be used to define new ones. All these functions can be used when fitting models with `nls` or `nlsme`. Please, check the respective help pages for details.

In the case of `nls()` the specification of the model to be fitted differs from that

used for linear models. We will use as an example fitting the Michaelis-Menten equation describing reaction kinetics in biochemistry and chemistry. The mathematical formulation is given by:

$$v = \frac{d[P]}{dt} = \frac{V_{\max}[S]}{K_M + [S]} \quad (1.1)$$

The function takes its name from Michaelis and Menten’s paper from 1913 (Johnson and Goody 2011). A self-starting function implementing the Michaelis-Menten equation is available in R under the name `SSmicmen()`. We will use the `Puromycin` data set.

```
data(Puromycin)
names(Puromycin)
## [1] "conc" "rate" "state"
```

```
fm21 <- nls(rate ~ SSmicmen(conc, Vm, K), data = Puromycin,
            subset = state == "treated")
```


We can extract different components similarly as described for linear models (see section 1.7 on page 11).

```
class(fm21)
## [1] "nls"

summary(fm21)
##
## Formula: rate ~ SSmicmen(conc, Vm, K)
##
## 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: 0
## Achieved convergence tolerance: 1.929e-06

residuals(fm21)
## [1] 25.4339971 -3.5660029 -5.8109605  4.1890395 -11.3616075  4.6383925
## [7] -5.6846886 -12.6846886  0.1670798 10.1670798  6.0311723 -0.9688277
## attr("label")
## [1] "Residuals"

fitted(fm21)
## [1] 50.5660 50.5660 102.8110 102.8110 134.3616 134.3616 164.6847 164.6847
## [9] 190.8329 190.8329 200.9688 200.9688
## attr("label")
## [1] "Fitted values"
```

 If we use `str()` or `names()` we can see that there are differences with respect to linear model and generalized model fits. The returned object is of class `nls` and contains some new members and lacks others. Two members are related to the iterative approximation method used, `control` containing nested members holding iteration settings, and `convInfo` (convergence information) with nested members with information on the outcome of the iterative algorithm.

```
str(fm21, max.level = 1)
## List of 6
## $ m          :List of 16
## $ .. attr(*, "class")= chr "nlsModel"
## $ convInfo    :List of 5
## $ data        : symbol Puromycin
## $ call        : language nls(formula = rate ~ SSmicmen(conc, Vm, K), data = Puromycin, subset = state == "tr
## $ dataClasses: Named chr "numeric"
## $ .. attr(*, "names")= chr "conc"
## $ control     :List of 7
## $ - attr(*, "class")= chr "nls"

fm21$convInfo
## $isConv
## [1] TRUE
##
## $finIter
## [1] 0
##
## $finTol
## [1] 1.928554e-06
##
## $stopCode
## [1] 0
##
## $stopMessage
## [1] "converged"
```

1.10 Model formulas

R is consistent in how it treats various objects, to a extent that can be surprising to those familiar with other computer languages. Model formulas are objects of class `formula` and `model call` and can be manipulated and stored similarly to objects of other classes.

```
class(y ~ x)
## [1] "formula"

mode(y ~ x)
## [1] "call"
```

Like any other R object formulas can be assigned to variables and be members

of lists and vectors. Consequently, the first linear model fit example from page 11 can be rewritten as follows.


```
my.formula <- dist ~ 1 + speed
fm1 <- lm(my.formula, data=cars)
```

In some situations, e.g., calculation of correlations, models lacking a lhs term (a term on the left hand side of `~`) are used but at least one term on the rhs must be present in the rhs.

```
class(~ x + y)
## [1] "formula"

mode(~ x + y)
## [1] "call"

is.empty.model(~ x + y)
## [1] FALSE
```

 In this box we highlight, for completeness, some idiosyncracies of R formulas that are seldom important in everyday use but can be important in advanced scripts. As with other classes, empty objects or vectors of length zero are valid. In the case of formulas there is an additional kind of emptiness, a formula describing a model with no explanatory terms on its rhs.

An empty object of class `formula` can be created with `formula()`. The last statement triggers an error as there is no model formula.

```
class(formula())
## [1] "formula"

mode(formula())
## [1] "list"

# is.empty.model(formula())
```

An object of class `formula` containing a formula object describing an empty model. While `y ~ 1` describes a model with only an intercept (estimating $a = \bar{x}$), `y ~ 0` or its equivalent `y ~ -1`, describes an empty model that cannot be fitted to data.

```
class(y ~ 0)
## [1] "formula"

mode(y ~ 0)
## [1] "call"

is.empty.model(y ~ 0)
## [1] TRUE

is.empty.model(y ~ 1)
## [1] FALSE

is.empty.model(y ~ x)
## [1] FALSE
```

The value returned by `length()` on a single formula is not always 1, the number of formulas in the vector of formulas, but instead the number of components in the formula. For longer and shorter vectors, it does return the number of member formulae. Because of this, it is better to store model formulas in objects of class `list` than in vectors, as `length()` consistently returns the expected value on lists.

```
length(formula())  
## [1] 0  
  
length(y ~ 0)  
## [1] 3  
  
length(y ~ 1)  
## [1] 3  
  
length(y ~ x)  
## [1] 3  
  
length(c(y ~ 1, y ~ x))  
## [1] 2  
  
length(list(y ~ 1))  
## [1] 1  
  
length(list(y ~ 1, y ~ x))  
## [1] 2
```

In the examples in previous sections we fitted simple models. More complex ones can be easily formulated using the same syntax. First of all, one can avoid use of operator `*` and explicitly define all individual main effects and interactions using operators `+` and `:`. The syntax implemented in base R allows grouping by means of parentheses, so it is also possible to exclude some interactions by combining the use of `*` and parentheses.

The same symbols as for arithmetic operators are used for model formulas. Within a formula, symbols are interpreted according to formula syntax. When we mean an arithmetic operation that could be interpreted as being part of the model formula we need to “protect” it by means of the identity function `I()`. The next two examples define formulas for models with only one explanatory variable. With formulas like these, the explanatory variable will be computed on the fly when fitting the model to data. In the first case below we need to explicitly protect the addition of the two variables into their sum, because otherwise they would be interpreted as two separate explanatory variables in the model. In the second case, `log()` cannot be interpreted as part of the model formula, and consequently does not require additional protection, neither does the expression passed as its argument.

```
y ~ I(x1 + x2)  
y ~ log(x1 + x2)
```

R formula syntax allows alternative ways for specifying interaction terms. They allow “abbreviated” ways of entering formulas, which for complex experimental designs saves typing and can improve clarity. As seen above, operator `*` saves us from having to explicitly indicate all the interaction terms in a full factorial model.

```
y ~ x1 + x2 + x3 + x1:x2 + x1:x3 + x2:x3 + x1:x2:x3
```

Can be replaced by a concise equivalent.

```
y ~ x1 * x2 * x3
```

When the model to be specified does not include all possible interaction terms, we can combine the concise notation with parentheses.

```
y ~ x1 + (x2 * x3)
y ~ x1 + x2 + x3 + x2:x3
```

That the two model formulas above are equivalent, can be seen using `terms()`

```
terms(y ~ x1 + (x2 * x3))
## y ~ x1 + (x2 * x3)
## attr("variables")
## list(y, x1, x2, x3)
## attr("factors")
##      x1 x2 x3 x2:x3
## y    0  0  0      0
## x1    1  0  0      0
## x2    0  1  0      1
## x3    0  0  1      1
## attr("term.labels")
## [1] "x1"      "x2"      "x3"      "x2:x3"
## attr("order")
## [1] 1 1 1 2
## attr("intercept")
## [1] 1
## attr("response")
## [1] 1
## attr(".Environment")
## <environment: R_GlobalEnv>
```

```
y ~ x1 * (x2 + x3)
y ~ x1 + x2 + x3 + x1:x2 + x1:x3
```

```
terms(y ~ x1 * (x2 + x3))
## y ~ x1 * (x2 + x3)
## attr("variables")
## list(y, x1, x2, x3)
## attr("factors")
##      x1 x2 x3 x1:x2 x1:x3
## y    0  0  0      0      0
## x1    1  0  0      1      1
## x2    0  1  0      1      0
## x3    0  0  1      0      1
## attr("term.labels")
## [1] "x1"      "x2"      "x3"      "x1:x2" "x1:x3"
## attr("order")
## [1] 1 1 1 2 2
## attr("intercept")
## [1] 1
## attr("response")
## [1] 1
## attr(".Environment")
## <environment: R_GlobalEnv>
```

The `^` operator provides a concise notation to limit the order of the interaction terms included in a formula.

```
y ~ (x1 + x2 + x3)^2
y ~ x1 + x2 + x3 + x1:x2 + x1:x3 + x2:x3
```

```
terms(y ~ (x1 + x2 + x3)^2)
## y ~ (x1 + x2 + x3)^2
## attr("variables")
## list(y, x1, x2, x3)
## attr("factors")
##      x1 x2 x3 x1:x2 x1:x3 x2:x3
## y    0  0  0      0      0      0
## x1    1  0  0      1      1      0
## x2    0  1  0      1      0      1
## x3    0  0  1      0      1      1
## attr("term.labels")
## [1] "x1"      "x2"      "x3"      "x1:x2" "x1:x3" "x2:x3"
## attr("order")
## [1] 1 1 1 2 2 2
## attr("intercept")
## [1] 1
## attr("response")
## [1] 1
## attr(".Environment")
## <environment: R_GlobalEnv>
```



For operator `^` to behave as expected, its first operand should be a formula with no interactions! Compare the result of expanding these two formulas with `terms()`.

```
y ~ (x1 + x2 + x3)^2
y ~ (x1 * x2 * x3)^2
```

Operator `%in%` can also be used as a shortcut for including only some of all the possible interaction terms in a formula.

```
y ~ x1 + x2 + x1 %in% x2
```

```
terms(y ~ x1 + x2 + x1 %in% x2)
## y ~ x1 + x2 + x1 %in% x2
## attr("variables")
## list(y, x1, x2)
## attr("factors")
##      x1 x2 x1:x2
## y    0  0      0
## x1    1  0      1
## x2    0  1      1
## attr("term.labels")
## [1] "x1"      "x2"      "x1:x2"
## attr("order")
## [1] 1 1 2
## attr("intercept")
```

```
## [1] 1
## attr("response")
## [1] 1
## attr(".Environment")
## <environment: R_GlobalEnv>
```



Execute the examples below using the `npk` data set from R. They demonstrate the use of different model formulas in ANOVA. Use these examples plus your own variations on the same theme to build your understanding of the syntax of model formulas. Based on the terms displayed in the ANOVA tables, first work out what models are being fitted in each case. In a second step, write each of the models using a mathematical formulation. Finally, think how model choice may affect the conclusions from an analysis of variance.

```
data(npk)
anova(lm(yield ~ N * P * K, data = npk))
anova(lm(yield ~ (N + P + K)^2, data = npk))
anova(lm(yield ~ N + P + K + P %in% N + K %in% N, data = npk))
anova(lm(yield ~ N + P + K + N %in% P + K %in% P, data = npk))
```

Nesting of factors in experiments using hierarchical designs such as split-plots or repeated measures, results in the need to compute additional error terms, differing in their degrees of freedom. In such a design, different effects are tested based on different error terms. Whether nesting exists or not is a property of an experiment. It is decided as part of the design of the experiment based on the mechanics of treatment assignment to experimental units. In base-R model-formulas, nesting needs to be described by explicit definition of error terms by means of `Error()` within the formula. Nowadays, linear mixed-effects (LME) models are most frequently used with data from experiments and surveys using hierarchical designs, as implemented in packages ‘nlme’ and ‘lme4’. These two packages use their own extensions to the model formula syntax to describe nesting and distinguishing fixed and random effects. Additive models have required other extensions, most of them specific to individual packages. These extensions fall outside the scope of this book.



R will accept any syntactically correct model formula, even when the results of the fit are not interpretable. It is *the responsibility of the user to ensure that models are meaningful*. The most common, and dangerous, mistake is specifying for factorial experiments, models that are missing lower-order interactions.

Fitting models like those below to data from an experiment based on a three-way factorial design should be avoided. In both cases simpler terms are missing, while higher-order interaction(s) that include the missing term are included in the model. Such models are not interpretable, as the variation from the missing term(s) ends being “disguised” within the remaining terms, distorting their apparent significance and parameter estimates.

```
y ~ A + B + A:B + A:C + B:C
y ~ A + B + C + A:B + A:C + A:B:C
```

In contrast to those above, the models below are interpretable, even if not “full” models (not including all possible interactions).

```
y ~ A + B + C + A:B + A:C + B:C
y ~ (A + B + C)^2
y ~ A + B + C + B:C
y ~ A + B * C
```


As seen in chapter ??, almost everything in the R language is an object that can be stored and manipulated. Model formulas are also objects, objects of class “formula”.

```
class(y ~ x)
## [1] "formula"
```

```
a <- y ~ x
class(a)
## [1] "formula"
```

There is no method `is.formula()` in base R, but we can easily test the class of an object with `inherits()`.

```
inherits(a, "formula")
## [1] TRUE
```

 **Manipulation of model formulas.** Because this is a book about the R language, it is pertinent to describe how formulas can be manipulated. Formulas, as any other R objects, can be saved in variables including lists. Why is this useful? For example, if we want to fit several different models to the same data, we can write a `for` loop that walks through a list of model formulas. Or we can write a function that accepts one or more formulas as arguments.

The use of `for` loops for iteration over a list of model formulas is described in section ?? on page ??.

```
my.data <- data.frame(x = 1:10, y = (1:10) / 2 + rnorm(10))
anovas <- list()
formulas <- list(a = y ~ x - 1, b = y ~ x, c = y ~ x + x^2)
for (formula in formulas) {
  anovas <- c(anovas, list(lm(formula, data = my.data)))
}
str(anovas, max.level = 1)
## List of 3
## $ :List of 12
## .. attr(*, "class")= chr "lm"
## $ :List of 12
## .. attr(*, "class")= chr "lm"
## $ :List of 12
## .. attr(*, "class")= chr "lm"
```

As could be expected, a conversion constructor is available with name `as.formula()`. It is useful when formulas are input interactively by the user or

read from text files. With `as.formula()` we can convert a character string into a formula.

```
my.string <- "y ~ x"
lm(as.formula(my.string), data = my.data)
##
## Call:
## lm(formula = as.formula(my.string), data = my.data)
##
## Coefficients:
## (Intercept)          x
##      1.4419      0.2677
```

As there are many functions for the manipulation of character strings available in base R and through extension packages, it is straightforward to build model formulas programmatically as strings. We can use functions like `paste()` to assemble a formula as text, and then use `as.formula()` to convert it to an object of class `formula`, usable for fitting a model.

```
my.string <- paste("y", "x", sep = "~")
lm(as.formula(my.string), data = my.data)
##
## Call:
## lm(formula = as.formula(my.string), data = my.data)
##
## Coefficients:
## (Intercept)          x
##      1.4419      0.2677
```

For the reverse operation of converting a formula into a string, we have available methods `as.character()` and `format()`. The first of these methods returns a character vector containing the components of the formula as individual strings, while `format()` returns a single character string with the formula formatted for printing.

```
formatted.string <- format(y ~ x)
formatted.string
## [1] "y ~ x"

as.formula(formatted.string)
## y ~ x
```

It is also possible to *edit* formula objects with method `update()`. In the replacement formula, a dot can replace either the left-hand side (lhs) or the right-hand side (rhs) of the existing formula in the replacement formula. We can also remove terms as can be seen below. In some cases the dot corresponding to the lhs can be omitted, but including it makes the syntax clearer.

```
my.formula <- y ~ x1 + x2
update(my.formula, . ~ . + x3)
## y ~ x1 + x2 + x3

update(my.formula, . ~ . - x1)
## y ~ x2

update(my.formula, . ~ x3)
## y ~ x3

update(my.formula, z ~ .)
## z ~ x1 + x2

update(my.formula, . + z ~ .)
## y + z ~ x1 + x2
```

R provides high-level functions for model selection. Consequently many R users will rarely need to edit model formulas in their scripts. For example, step-wise model selection is possible with R method `step()`.

A matrix of dummy coefficients can be derived from a model formula, a type of contrast, and the data for the explanatory variables.

```
treats.df <- data.frame(A = rep(c("yes", "no"), c(4, 4)),
                        B = rep(c("white", "black"), 4))
treats.df
##      A      B
## 1 yes white
## 2 yes black
## 3 yes white
## 4 yes black
## 5 no  white
## 6 no  black
## 7 no  white
## 8 no  black
```

The default contrasts types currently in use.

```
options("contrasts")
## $contrasts
##      unordered      ordered
## "contr.treatment" "contr.poly"
```

A model matrix for a model for a two-way factorial design with no interaction term:

```

model.matrix(~ A + B, treats.df)
##      (Intercept) Ayes Bwhite
## 1             1    1      1
## 2             1    1      0
## 3             1    1      1
## 4             1    1      0
## 5             1    0      1
## 6             1    0      0
## 7             1    0      1
## 8             1    0      0
## attr(,"assign")
## [1] 0 1 2
## attr(,"contrasts")
## attr(,"contrasts")$A
## [1] "contr.treatment"
##
## attr(,"contrasts")$B
## [1] "contr.treatment"

```

A model matrix for a model for a two-way factorial design with interaction term:

```

model.matrix(~ A * B, treats.df)
##      (Intercept) Ayes Bwhite Ayes:Bwhite
## 1             1    1      1           1
## 2             1    1      0           0
## 3             1    1      1           1
## 4             1    1      0           0
## 5             1    0      1           0
## 6             1    0      0           0
## 7             1    0      1           0
## 8             1    0      0           0
## attr(,"assign")
## [1] 0 1 2 3
## attr(,"contrasts")
## attr(,"contrasts")$A
## [1] "contr.treatment"
##
## attr(,"contrasts")$B
## [1] "contr.treatment"

```

1.11 Time series

Longitudinal data consist of repeated measurements, usually done over time, on the same experimental units. Longitudinal data, when replicated on several experimental units at each time point, are called repeated measurements, while when not replicated, they are called time series. Base R provides special support for the analysis of time series data, while repeated measurements can be analyzed with nested linear models, mixed-effects models, and additive models.

Time series data are data collected in such a way that there is only one observa-

tion, possibly of multiple variables, available at each point in time. This brief section introduces only the most basic aspects of time-series analysis. In most cases time steps are of uniform duration and occur regularly, which simplifies data handling and storage. R not only provides methods for the analysis and manipulation of time-series, but also a specialized class for their storage, "ts". Regular time steps allow more compact storage—e.g., a ts object does not need to store time values for each observation but instead a combination of two of start time, step size and end time.

We start by creating a time series from a numeric vector. By now, you surely guessed that you need to use a constructor called `ts()` or a conversion constructor called `as.ts()` and that you can look up the arguments they accept by reading the corresponding help pages.

For example for a time series of monthly values we could use:

```
my.ts <- ts(1:10, start = 2019, deltat = 1/12)
class(my.ts)
## [1] "ts"

str(my.ts)
## Time-Series [1:10] from 2019 to 2020: 1 2 3 4 5 6 7 8 9 10
```

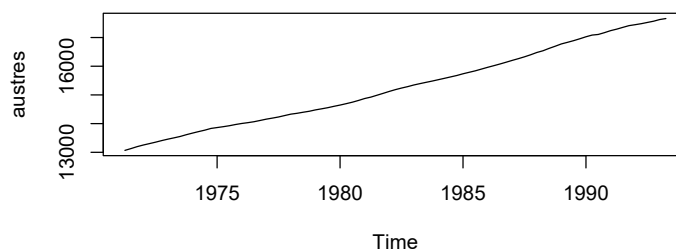
We next use the data set `austres` with data on the number of Australian residents and included in R.

```
class(austres)
## [1] "ts"

is.ts(austres)
## [1] TRUE
```

Time series `austres` is dominated by the increasing trend.

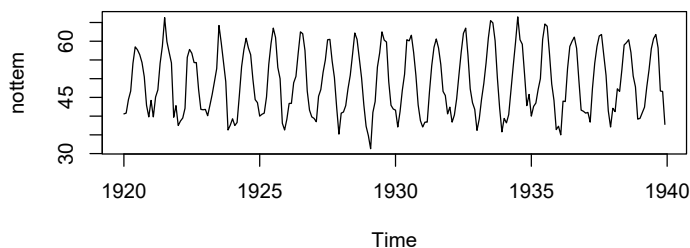
```
plot(austres)
```



A different example, using data set `nottem` containing meteorological data for Nottingham, shows a clear cyclic component. The annual cycle of mean air temperatures (in degrees Fahrenheit) is clear when data are plotted.

```
data(nottem)
is.ts(nottem)
## [1] TRUE

plot(nottem)
```

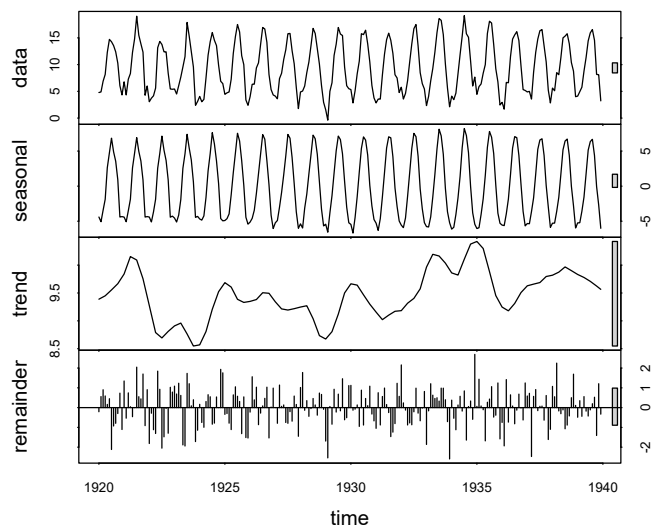


In the next two code chunks, two different approaches to time series decomposition are used. In the first one we use a moving average to capture the trend, while in the second approach we use Loess (a smooth curve fitted by local weighted regression) for the decomposition, a method for which the acronym STL (Seasonal and Trend decomposition using Loess) is used. Before decomposing the time-series we reexpress the temperatures in degrees Celsius.

```
nottem.celcius <- (nottem - 32) * 5/9
```

We set the seasonal window to 7 months, the minimum accepted.

```
nottem.stl <- stl(nottem.celcius, s.window = 7)
plot(nottem.stl)
```



It is interesting to explore the class and structure of the object returned by `stl()`, as we may want to extract components. Run the statements below to find out, and then plot individual components from the time series decomposition.

```
class(nottem.stl)
str(nottem.stl)
```

1.12 Multivariate statistics

1.12.1 Multivariate analysis of variance

Multivariate methods take into account several response variables simultaneously, as part of a single analysis. In practice it is usual to use contributed packages for multivariate data analysis in R, except for simple cases. We will look first at *multivariate* ANOVA or MANOVA. In the same way as `aov()` is a wrapper that uses internally `lm()`, `manova()` is a wrapper that uses internally `aov()`.

Multivariate model formulas in base R require the use of column binding (`cbind()`) on the left-hand side (lhs) of the model formula. For the next examples we use the well-known `iris` data set, containing size measurements for flowers of two species of *Iris*.

```
data(iris)
mmf1 <- lm(cbind(Petal.Length, Petal.Width) ~ Species, data = iris)
anova(mmf1)
## Analysis of Variance Table
##
##              Df  Pillai approx F num Df den Df    Pr(>F)
## (Intercept)   1 0.98786   5939.2     2    146 < 2.2e-16 ***
## Species       2 1.04645    80.7     4    294 < 2.2e-16 ***
## Residuals    147
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

summary(mmf1)
## Response Petal.Length :
##
## Call:
## lm(formula = Petal.Length ~ Species, data = iris)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.260 -0.258  0.038  0.240  1.348
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)    1.46200    0.06086   24.02  <2e-16 ***
## Speciesversicolor 2.79800    0.08607   32.51  <2e-16 ***
## Speciesvirginica  4.09000    0.08607   47.52  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4303 on 147 degrees of freedom
## Multiple R-squared:  0.9414, Adjusted R-squared:  0.9406
## F-statistic: 1180 on 2 and 147 DF, p-value: < 2.2e-16
##
##
## Response Petal.Width :
##
## Call:
## lm(formula = Petal.Width ~ Species, data = iris)
##
## Residuals:
```

```
##      Min      1Q  Median      3Q      Max
## -0.626 -0.126 -0.026  0.154  0.474
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)    0.24600    0.02894   8.50 1.96e-14 ***
## Speciesversicolor 1.08000    0.04093  26.39 < 2e-16 ***
## Speciesvirginica  1.78000    0.04093  43.49 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2047 on 147 degrees of freedom
## Multiple R-squared:  0.9289, Adjusted R-squared:  0.9279
## F-statistic: 960 on 2 and 147 DF, p-value: < 2.2e-16
```

```
mmf2 <- manova(cbind(Petal.Length, Petal.Width) ~ Species, data = iris)
anova(mmf2)
## Analysis of Variance Table
##
##              Df Pillai approx F num Df den Df    Pr(>F)
## (Intercept)   1 0.98786   5939.2      2   146 < 2.2e-16 ***
## Species       2 1.04645    80.7      4   294 < 2.2e-16 ***
## Residuals    147
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

summary(mmf2)
##              Df Pillai approx F num Df den Df    Pr(>F)
## Species       2 1.0465   80.661      4   294 < 2.2e-16 ***
## Residuals    147
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```



Modify the example above to use `aov()` instead of `manova()` and save the result to a variable named `mmf3`. Use `class()`, `attributes()`, `names()`, `str()` and extraction of members to explore objects `mmf1`, `mmf2` and `mmf3`. Are they different?

1.12.2 Principal components analysis

Principal components analysis (PCA) is used to simplify a data set by combining variables with similar and “mirror” behavior into principal components. At a later stage, we frequently try to interpret these components in relation to known and/or assumed independent variables. Base R’s function `prcomp()` computes the principal components and accepts additional arguments for centering and scaling.

```
pc <- prcomp(iris[c("Sepal.Length", "Sepal.Width",
                    "Petal.Length", "Petal.Width")],
             center = TRUE,
             scale = TRUE)
```

By printing the returned object we can see the loadings of each variable in the principal components `P1` to `P4`.

```
class(pc)
## [1] "prcomp"

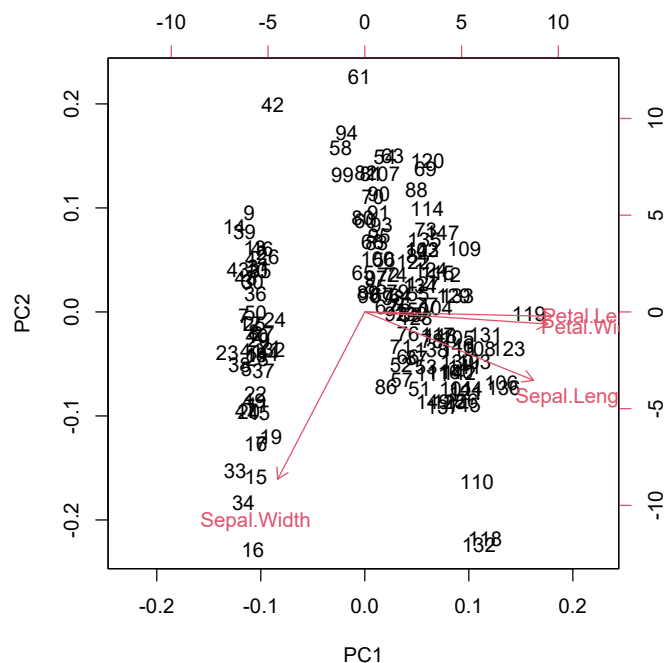
pc
## Standard deviations (1, ..., p=4):
## [1] 1.7083611 0.9560494 0.3830886 0.1439265
##
## Rotation (n x k) = (4 x 4):
##           PC1      PC2      PC3      PC4
## Sepal.Length 0.5210659 -0.37741762 0.7195664 0.2612863
## Sepal.Width  -0.2693474 -0.92329566 -0.2443818 -0.1235096
## Petal.Length 0.5804131 -0.02449161 -0.1421264 -0.8014492
## Petal.Width  0.5648565 -0.06694199 -0.6342727 0.5235971
```

In the summary, the rows “Proportion of Variance” and “Cumulative Proportion” are most informative of the contribution of each principal component (PC) to explaining the variation among observations.

```
summary(pc)
## Importance of components:
##           PC1      PC2      PC3      PC4
## Standard deviation 1.7084 0.9560 0.38309 0.14393
## Proportion of Variance 0.7296 0.2285 0.03669 0.00518
## Cumulative Proportion 0.7296 0.9581 0.99482 1.00000
```

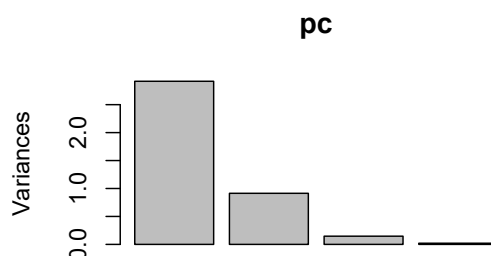
Method `biplot()` produces a plot with one principal component (PC) on each axis, plus arrows for the loadings.

```
biplot(pc)
```





Method `plot()` generates a bar plot of variances corresponding to the different components.

```
plot(pc)
```



Visually more elaborate plots of the principal components and their loadings can be obtained using package ‘ggplot’ described in chapter ?? starting on page ??. Package ‘ggfortify’ extends ‘ggplot’ so as to make it easy to plot principal components and their loadings.

 For growth and morphological data, a log-transformation can be suitable given that variance is frequently proportional to the magnitude of the values measured. We leave as an exercise to repeat the above analysis using transformed values for the dimensions of petals and sepals. How much does the use of transformations change the outcome of the analysis?

 As for other fitted models, the object returned by function `prcomp()` is a list with multiple components.

```
str(pc, max.level = 1)
```

1.12.3 Multidimensional scaling

The aim of multidimensional scaling (MDS) is to visualize in 2D space the similarity between pairs of observations. The values for the observed variable(s) are used to compute a measure of distance among pairs of observations. The nature of the data will influence what distance metric is most informative. For MDS we start with a matrix of distances among observations. We will use, for the example, distances in kilometers between geographic locations in Europe from data set `eurodist`.

```
loc <- cmdscale(eurodist)
```

We can see that the returned object `loc` is a `matrix`, with names for one of the dimensions.

```

class(loc)
## [1] "matrix" "array"

dim(loc)
## [1] 21  2

dimnames(loc)
## [[1]]
## [1] "Athens"          "Barcelona"      "Brussels"      "Calais"
## [5] "Cherbourg"        "Cologne"        "Copenhagen"     "Geneva"
## [9] "Gibraltar"        "Hamburg"        "Hook of Holland" "Lisbon"
## [13] "Lyons"            "Madrid"         "Marseilles"     "Milan"
## [17] "Munich"           "Paris"          "Rome"           "Stockholm"
## [21] "Vienna"
##
## [[2]]
## NULL

head(loc)
##              [,1]      [,2]
## Athens      2290.27468 1798.8029
## Barcelona  -825.38279  546.8115
## Brussels     59.18334 -367.0814
## Calais      -82.84597 -429.9147
## Cherbourg  -352.49943 -290.9084
## Cologne     293.68963 -405.3119

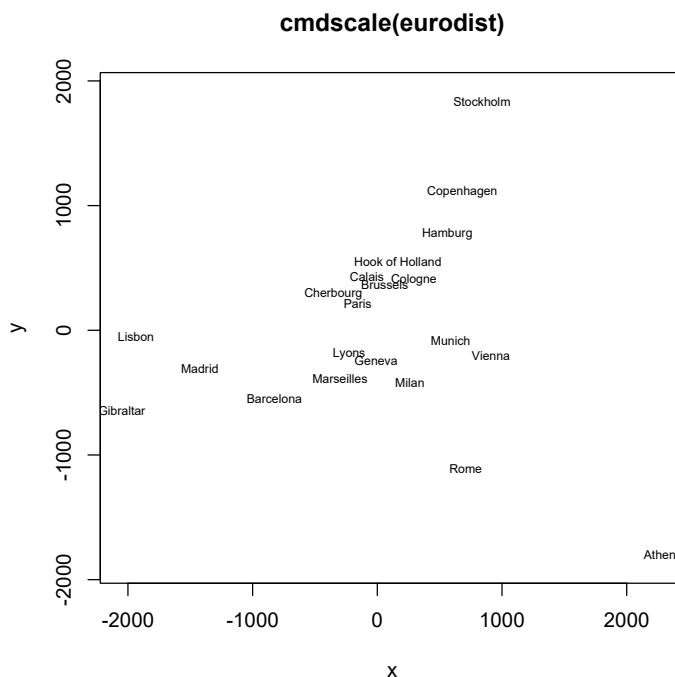
```


To make the code easier to read, two vectors are first extracted from the matrix and named *x* and *y*. We force aspect to equality so that distances on both axes are comparable.

```

x <- loc[, 1]
y <- -loc[, 2] # change sign so North is at the top
plot(x, y, type = "n", asp = 1,
     main = "cmdscale(eurodist)")
text(x, y, rownames(loc), cex = 0.6)

```



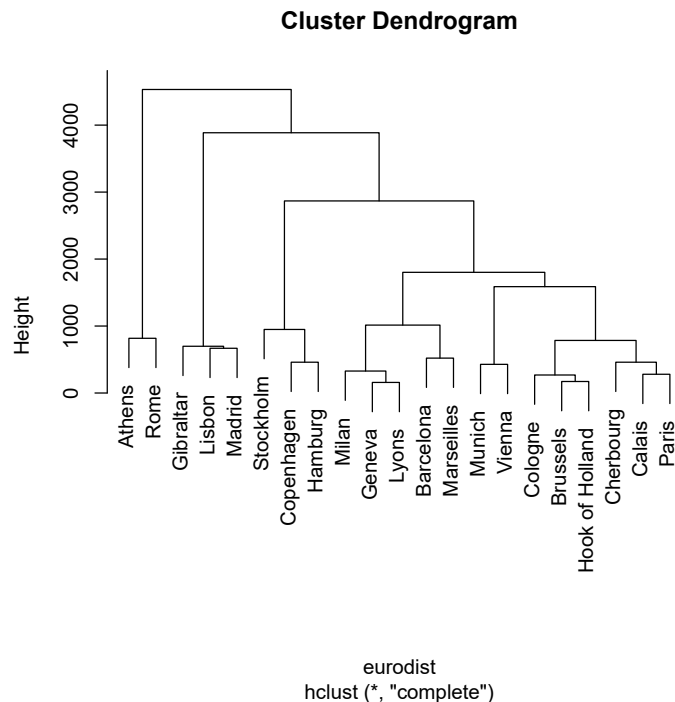
 Find data on the mean annual temperature, mean annual rainfall and mean number of sunny days at each of the locations in the `eurodist` data set. Next, compute suitable distance metrics, for example, using function `dist`. Finally, use MDS to visualize how similar the locations are with respect to each of the three variables. Devise a measure of distance that takes into account the three climate variables and use MDS to find how distant the different locations are.

1.12.4 Cluster analysis

In cluster analysis, the aim is to group observations into discrete groups with maximal internal homogeneity and maximum group-to-group differences. In the next example we use function `hclust()` from the base-R package ‘stats’. We use, as above, the `eurodist` data which directly provides distances. In other cases a matrix of distances between pairs of observations needs to be first calculated with function `dist` which supports several methods.

```
hc <- hclust(eurodist)
print(hc)
##
## Call:
## hclust(d = eurodist)
##
## Cluster method : complete
## Number of objects: 21
```

```
plot(hc)
```



We can use `cutree()` to limit the number of clusters by directly passing as an argument the desired number of clusters or the height at which to cut the tree.

```
cutree(hc, k = 5)
```

##	Athens	Barcelona	Brussels	Calais	Cherbourg
##	1	2	3	3	3
##	Cologne	Copenhagen	Geneva	Gibraltar	Hamburg
##	3	4	2	5	4
##	Hook of Holland	Lisbon	Lyons	Madrid	Marseilles
##	3	5	2	5	2
##	Milan	Munich	Paris	Rome	Stockholm
##	2	3	3	1	4
##	Vienna				
##	3				

The object returned by `hclust()` contains details of the result of the clustering, which allows further manipulation and plotting.

```
str(hc)
```

```
## List of 7
## $ merge      : int [1:20, 1:2] -8 -3 -6 -4 -16 -17 -5 -7 -2 -12 ...
## $ height     : num [1:20] 158 172 269 280 328 428 460 460 521 668 ...
## $ order      : int [1:21] 1 19 9 12 14 20 7 10 16 8 ...
## $ labels     : chr [1:21] "Athens" "Barcelona" "Brussels" "Calais" ...
## $ method     : chr "complete"
## $ call       : language hclust(d = eurodist)
## $ dist.method: NULL
## - attr(*, "class")= chr "hclust"
```

1.13 Further reading

Two recent text books on statistics, following a modern approach, and using R for examples, are *OpenIntro Statistics* (Diez et al. 2019) and *Modern Statistics for Modern Biology* (Holmes and Huber 2019). Three examples of books introducing statistical computations in R are *Introductory Statistics with R* (Dalgaard 2008), *A Handbook of Statistical Analyses Using R* (B. S. Everitt and Hothorn 2010) and *A Beginner’s Guide to R* (Zuur et al. 2009). More advanced books are available with detailed descriptions of various types of analyses in R, including thorough descriptions of the methods briefly presented in this chapter. Good examples of books with broad scope are *The R Book* (Crawley 2012) and the classic reference *Modern Applied Statistics with S* (Venables and Ripley 2002). More specific books are also available from which a few suggestions for further reading are *An Introduction to Applied Multivariate Analysis with R* (B. Everitt and Hothorn 2011), *Linear Models with R* (Faraway 2004), *Extending the linear model with R: generalized linear, mixed effects and nonparametric regression models* (Faraway 2006), *Mixed-Effects Models in S and S-Plus* (Pinheiro and Bates 2000) and *Generalized Additive Models* (Wood 2017).

Bibliography

- Crawley, M. J. (2012). *The R Book*. Wiley, p. 1076. ISBN: 0470973927 (cit. on pp. 23, 48).
- Dalgaard, P. (2008). *Introductory Statistics with R*. Springer, p. 380. ISBN: 0387790543 (cit. on p. 48).
- Diez, D., M. Cetinkaya-Rundel, and C. D. Barr (2019). *OpenIntro Statistics*. 4th ed. 422 pp. URL: <https://www.openintro.org/stat/os4.php> (visited on 11/20/2022) (cit. on p. 48).
- Everitt, B. and T. Hothorn (2011). *An Introduction to Applied Multivariate Analysis with R*. Springer, p. 288. ISBN: 1441996494 (cit. on p. 48).
- Everitt, B. S. and T. Hothorn (2010). *A Handbook of Statistical Analyses Using R*. 2nd ed. Chapman & Hall/CRC, p. 376. ISBN: 1420079336 (cit. on p. 48).
- Faraway, J. J. (2004). *Linear Models with R*. Boca Raton, FL: Chapman & Hall/CRC, p. 240 (cit. on p. 48).
- (2006). *Extending the linear model with R: generalized linear, mixed effects and nonparametric regression models*. Chapman & Hall/CRC, p. 345. ISBN: 158488424X (cit. on p. 48).
- Hamming, R. W. (1987). *Numerical Methods for Scientists and Engineers*. Dover Publications Inc. 752 pp. ISBN: 0486652416.
- Holmes, S. and W. Huber (2019). *Modern Statistics for Modern Biology*. Cambridge University Press. 382 pp. ISBN: 1108705294 (cit. on p. 48).
- Johnson, K. A. and R. S. Goody (2011). “The Original Michaelis Constant: Translation of the 1913 Michaelis–Menten Paper”. In: *Biochemistry* 50, pp. 8264–8269. DOI: 10.1021/bi201284u (cit. on p. 28).
- Pinheiro, J. C. and D. M. Bates (2000). *Mixed-Effects Models in S and S-Plus*. New York: Springer (cit. on p. 48).
- Smith, H. F. (1957). “Interpretation of adjusted treatment means and regressions in analysis of covariance”. In: *Biometrics* 13, pp. 281–308 (cit. on p. 24).
- Venables, W. N. and B. D. Ripley (2002). *Modern Applied Statistics with S*. 4th. New York: Springer. ISBN: 0-387-95457-0 (cit. on pp. 23, 48).
- Wood, S. N. (2017). *Generalized Additive Models*. Chapman and Hall/CRC. 476 pp. ISBN: 1498728332 (cit. on p. 48).
- Zuur, A. F., E. N. Ieno, and E. Meesters (2009). *A Beginner’s Guide to R*. 1st ed. Springer, p. 236. ISBN: 0387938362 (cit. on p. 48).



General index

- analysis of variance
 - model formula, 34
- ANCOVA, *see* analysis of covariance
- ANOVA, *see* analysis of variance
- chemical reaction kinetics, 28
- cluster analysis, 46–47
- correlation, 8–10
 - Kendall, 9
 - non-parametric, 9
 - parametric, 8
 - Pearson, 8
 - Spearman, 9
- distributions, 3–8
 - density from parameters, 4
 - probabilities from quantiles, 4
 - pseudo-random draws, 6
 - quantiles from probabilities, 5
- functions
 - base R, 2
- further reading
 - statistics in R, 48
- generalized linear models, 24–26
- GLM, *see* generalized linear models
- languages
 - S, 21
- linear models, 11–24
 - analysis of covariance, 24
 - analysis of variance, 20
 - contrasts, 21–23
 - linear regression, 11
 - polynomial regression, 14
 - summary table, 13
- LM, *see* linear models
- 'lme4', 34
- MANOVA, *see* multivariate analysis of variance
- MDS, *see* multidimensional scaling
- Michaelis-Menten equation, 28
- model formulas, 29–38
 - manipulation, 35
- models
 - generalized linear, *see*
 - generalized linear models
 - linear, *see* linear models
 - non-linear, *see* non-linear models
 - selfstart, 28
- models fitting, 10–11
- multidimensional scaling, 44–46
- multivariate analysis of variance, 41–42
- multivariate methods, 41–47
- 'nlme', 34
- NLS, *see* non-linear models
- non-linear models, 27–29
- Normal distribution, 3
- packages
 - 'lme4', 34
 - 'nlme', 34
 - 'stats', 46
- PCA, *see* principal components analysis
- polynomial regression, 14
- principal components analysis, 42–44
- programmes
 - S, 22
 - SAS, 21, 22
 - SPSS, 21
- pseudo-random numbers, 6
- pseudo-random sampling, 7

random numbers, *see*
 pseudo-random numbers
random sampling, *see*
 pseudo-random sampling

S, 21, 22

SAS, 21, 22

self-starting functions, 27

SPSS, 21

‘stats’, 46

summaries

 statistical, 2

time series, 38–41

 decomposition, 40

Index of R names by category

classes and modes
 call, 29
 formula, 29, 30
 list, 31

data objects
 austres, 39
 cars, 11
 eurodist, 44, 46
 InsectSprays, 20, 24
 iris, 41
 nottem, 39
 npk, 34
 Puromycin, 28

functions and methods
 AIC(), 10, 15
 anova(), 10, 14-16, 18, 20, 23, 24
 aov(), 20, 41
 as.formula(), 35, 36
 as.ts(), 39
 attributes(), 9
 BIC(), 10, 15
 biplot(), 43
 boxplot.stats(), 2
 class(), 9
 coef(), 10, 15, 21
 coefficients(), 15
 contr.helmert(), 22, 23
 contr.poly(), 23
 contr.SAS(), 22, 23
 contr.sum(), 22
 contr.treatment(), 22, 23
 cor(), 8-10
 cor.test(), 9, 10
 cutree(), 47
 dbinom(), 3
 dchisq(), 3
 decompose(), 40
 df(), 3
 dist, 46
 dlnorm(), 3
 dmultinom(), 3
 dnorm(), 3
 dpois(), 3
 dt(), 3
 dunif(), 3
 effects(), 15
 factor(), 21
 fitted(), 10, 15
 fitted.values(), 15
 glm(), 24
 hclust(), 46, 47
 I(), 14, 31
 inherits(), 35
 length(), 2, 31
 lm(), 11, 14, 19, 20, 41
 log(), 31
 mad(), 2
 manova(), 41
 matrix(), 8
 max(), 2
 mean(), 2
 median(), 2
 min(), 2
 mode(), 2
 model.frame(), 15
 model.matrix(), 15
 nlme, 27
 nls, 27
 nls(), 27
 ordered(), 21
 pbinom(), 3
 pchisq(), 3
 pf(), 3
 plnorm(), 3

plot(), 10, 25
pmultinom(), 3
pnorm(), 3, 5
poly(), 14
ppois(), 3
prcomp(), 42, 44
predict(), 10, 19
print(), 9
pt(), 3, 5, 19
punif(), 3
qbinom(), 3
qchisq(), 3
qf(), 3
qlnorm(), 3
qmultinom(), 3
qnorm(), 3, 5
qpois(), 3
qt(), 3
quantile(), 2
qunif(), 3
range(), 2
rbinom(), 3
rchisq(), 3
resid(), 15

residuals(), 10, 15
rf(), 3
rlnorm(), 3
rmultinom(), 3
rnorm(), 3, 6–8
rpois(), 3
rt(), 3
runif(), 3, 6
sample(), 8
sd(), 2
set.seed(), 6, 7
SSmicmen(), 28
stl(), 40
str(), 9, 16
summary(), 2, 10, 12, 16, 18, 20,
 21
terms(), 15, 33
ts(), 39
update(), 36
var(), 2
vcov(), 15

operators

[], 7

Alphabetic index of R names

[], 7

AIC(), 10, 15

anova(), 10, 14–16, 18, 20, 23, 24

aov(), 20, 41

as.formula(), 35, 36

as.ts(), 39

attributes(), 9

austres, 39

BIC(), 10, 15

biplot(), 43

boxplot.stats(), 2

call, 29

cars, 11

class(), 9

coef(), 10, 15, 21

coefficients(), 15

contr.helmert(), 22, 23

contr.poly(), 23

contr.SAS(), 22, 23

contr.sum(), 22

contr.treatment(), 22, 23

cor(), 8–10

cor.test(), 9, 10

cutree(), 47

dbinom(), 3

dchisq(), 3

decompose(), 40

df(), 3

dist, 46

dlnorm(), 3

dmultinom(), 3

dnorm(), 3

dpois(), 3

dt(), 3

dunif(), 3

effects(), 15

eurodist, 44, 46

factor(), 21

fitted(), 10, 15

fitted.values(), 15

formula, 29, 30

glm(), 24

hclust(), 46, 47

I(), 14, 31

inherits(), 35

InsectSprays, 20, 24

iris, 41

length(), 2, 31

list, 31

lm(), 11, 14, 19, 20, 41

log(), 31

mad(), 2

manova(), 41

matrix(), 8

max(), 2

mean(), 2

median(), 2

min(), 2

mode(), 2

model.frame(), 15

model.matrix(), 15

nlme, 27

nls, 27

nls(), 27

nottem, 39

npk, 34

ordered(), 21

pbinom(), 3

pchisq(), 3

pf(), 3
plnorm(), 3
plot(), 10, 25
pmultinom(), 3
pnorm(), 3, 5
poly(), 14
ppois(), 3
prcomp(), 42, 44
predict(), 10, 19
print(), 9
pt(), 3, 5, 19
punif(), 3
Puromycin, 28

qbinom(), 3
qchisq(), 3
qf(), 3
qlnorm(), 3
qmultinom(), 3
qnorm(), 3, 5
qpois(), 3
qt(), 3
quantile(), 2
qunif(), 3

range(), 2

rbinom(), 3
rchisq(), 3
resid(), 15
residuals(), 10, 15
rf(), 3
rlnorm(), 3
rmultinom(), 3
rnorm(), 3, 6–8
rpois(), 3
rt(), 3
runif(), 3, 6

sample(), 8
sd(), 2
set.seed(), 6, 7
SSmicmen(), 28
stl(), 40
str(), 9, 16
summary(), 2, 10, 12, 16, 18, 20, 21

terms(), 15, 33
ts(), 39

update(), 36

var(), 2
vcov(), 15