Einführung in die Statistik-Umgebung R

für den Weiterbildungslehrgang in Angewandter Statistik

Seminar für Statistik
ETH Zürich

16. April 2015

1997	Christian Keller (‚S-Hinweise‘)
1998	Roberto Invernizzi (Anpassungen)
2000	Werner Stahel, René Locher (Anpassungen)
2001-03	Werner Stahel, Monika Ferster, Ruth Meili, Roberto Frisullo (Neue Einführung, Anpassung an R, neuer Titel)
2003	Werner Stahel, Beat Jaggi (Anpassungen, neue Kapitel)
2010	Simon Rentzmann (Anpassungen, neue Kapitel)
2015	Nina Anderegg, Lukas Meier (Anpassungen)

Inhaltsverzeichnis

1 Einleitung

1.1 Was ist R?

R ist ein Programm-System, das sich für statistische Analysen in einem breiten Spektrum bestens eignet. Es beruht auf einer Sprache, die ursprünglich S heisst und die auch dem Paket S-Plus zugrunde liegt. R ist frei verfügbar – im Gegensatz zu S-Plus, für welches man Lizenzgebühr zahlt.

R enthält eine sehr grosse Bibliothek von Funktionen, die für statistische Analysen und grafische Darstellungen benützt werden können. Diese Bibliothek wird von angewandten Statistiker/innen rund um die Welt ständig erweitert, und viele benützen die Sprache, um neue Methoden zu entwickeln und zu verbreiten.

Weshalb kein Menu-System? Die soeben erwähnten Eigenschaften von R tönen zu hoch gezielt für einfachere Analysen. Es gibt auch etliche gute menü-geführte Statistikpakete, z.B. SPSS, SAS und Systat. S-Plus umfasst ebenfalls ein Menü-System, das mit der erwähnten Sprache S arbeitet. Menü-gesteuerte Systeme würden uns jedoch nicht bis zum Ende des Kurses führen. Sie kommen also nicht darum herum, eine Kommandosprache, wie sie R zugrunde liegt, zu lernen.

Einfache Analysen sind auch mit dem System R leicht durchzuführen, obwohl man nur mit Kommandos arbeitet. Deshalb wollen wir darauf verzichten, zuerst ein Menü-System zu benützen.

1.2 Ziel dieser Einführung

Es gibt viele gute Einführungen und Handbücher für die **S-Sprache**, also der gemeinsamen Sprache von R und S-Plus.

Das Ziel dieser Einführung ist es, ab Kapitel 2 den Einstieg möglichst sanft zu gestalten, so dass Sie neben dem möglicherweise neuen Jargon in Wahrscheinlichkeit und Statistik nicht gleich noch eine volle Programmiersprache verkraften müssen.

In den späteren Kapiteln werden die S-Funktionen besprochen, die zu den einzelnen Blöcken des Kurses gehören. Sie erhalten den Text jeweils mit den Vorlesungsunterlagen.

1.3 Dokumentation

Wenn Ihnen diese Einführung zu langsam vorwärts geht, können Sie eine der folgenden Unterlagen studieren:

- **An Introduction to R.** Dies ist die „offizielle“ Einführung in R. Sie wird vom „R Core Team“ permanent aktualisiert und kann als pdf-File vom Internet heruntergeladen werden (ca. 100 Seiten). Zu einem späteren Zeitpunkt (im Verlauf des Grundsemesters) wird es sich lohnen, diese Einführung systematisch durchzusehen. Sie finden sie unter der Web-Adresse `http://stat.ethz.ch/CRAN/`, links unter „Documentation“: `Manuals`. Dieses Dokument existiert auch in HTML-Form.
- Es gibt eine Reihe weiterer Einstiegshilfen für R-Novizen. Zum Beispiel **„R for Beginners“**, zu finden unter `http://stat.ethz.ch/CRAN/`, links unter „Documentation“: `Contributed`. Diese englische Einführung ist ursprünglich in französisch geschrieben worden und nun in beiden Sprachen erhältlich. Sie wird allerdings nicht permanent aktualisiert.
- Auch unter „Documentation“ findet man die **„Frequently Asked Questions“**. Es lohnt sich bisweilen, bei spontan auftauchenden Fragen dort nachzusehen.

1.4 Betriebs-Plattformen

Die erste Version von R ist für Macintosh konzipiert worden. Längst steht das Programm jedoch für alle Varianten von Unix, inkl. Linux, und die Windows-Plattformen zur Verfügung. Die Sprache selber ist auf allen Plattformen dieselbe, und entsprechend sind die Dokumentationen zu R sowohl für Unix- als auch für Windows-Benützer brauchbar. Der Unterschied liegt im Wesentlichen in der Umgebung, in welcher man arbeitet.

Viele Befehle, die man eingetippt hat, möchte man später leicht verändern und wieder verwenden. Man speichert sie deshalb mit Vorteil in einer so genannten „Skript“-Datei, die mit einem Editor bearbeitet wird. Je nach Plattform hat sich ein anderer Editor durchgesetzt. Mittlerweile am verbreitesten ist **R-Studio**, welches man unter `http://www.rstudio.com` herunterladen kann. R-Studio ist eigentlich nicht ein Editor, sondern eine eigentliche Programmierumgebung für R.

R herunterladen und installieren. Das Programm ist zu finden auf der Internetseite `http://stat.ethz.ch/CRAN/`.

R starten, benützen und beenden. Klicken Sie auf das beim Installieren entstandene Desktop-Icon oder wählen Sie `Start/Programme/R/R Gui`. R wird dann gestartet und auf dem Bildschirm erscheint die `R-Console`. Hier können Sie Ihre Befehle direkt eintippen. Dass R bereit ist, einen Befehl entgegenzunehmen, sehen Sie daran, dass am linken Rand der `R-Console` ein „>“, der so genannte „prompt“ steht. Tippen Sie versuchshalber nach diesem „prompt“ den Befehl `2+4`, und drücken Sie die Enter-Taste. Wenn Sie als Antwort die Zeile

```
[1] 6
```

erhalten, dann funktioniert's. Über weitere Befehle und erste Schritte im Programmieren erfahren Sie mehr in den folgenden Kapiteln. Beenden Sie nun R via Menü `File/Exit` bzw. `Datei/Beenden` oder indem Sie nach dem „prompt" den Befehl `q()` eingeben (q für „quit"). Es erscheint ein Dialogfenster mit der Frage „Save Workspace Image?". Klicken Sie „y", so können Sie beim nächsten Aufruf von R in dem Zustand weiterfahren, in dem Sie jetzt aufhören; sonst beginnen Sie mit einem leeren Workspace (Genaueres später). Wir werden ab jetzt R nicht mehr direkt starten, sondern immer direkt via R-Studio verwenden. Schliessen Sie also die R-Console nun.

Arbeiten mit dem Editor und Verwalten der erstellten R-Dateien. Wie oben gesagt, ist es im Allgemeinen sinnvoll, mit einem Editor zu arbeiten, damit jederzeit auf die bereits durchgeführte Programmierarbeit zurückgegriffen werden kann.

Die folgenden Hinweise sind auf Windows-Umgebungen ausgerichtet. Erstellen Sie zunächst einen Ordner mit dem Namen *RFiles*. An der ETH sollte dieser Ordner in Ihrem Home directory *AFS(T:)* liegen. Kreieren Sie ihn mit Hilfe des Windows Explorers (`Datei/Neu/Ordner`). Öffnen sie dann R-Studio über `Start/Programme/RStudio`. R-Studio öffnet von sich aus selber schon eine (interne) R-Konsole . Erstellen Sie eine neue R-Skript Datei unter dem Menü `File/New File/R Script`. Sie haben nun ein viergeteiltes Fenster auf Ihrer Bildschirmoberfläche: Die `R-Console` (links unten) und einen `Editor` (links oben), alles innerhalb R-Studio. Schreiben Sie einen Befehl in den Editor, zum Beispiel `z <- c(8,13,21)` (c für „concatenate") und in einer weiteren Zeile `2*z`. Markieren Sie die beiden Zeilen mit der linken Maustaste und drücken Sie `Control-Enter`. In der `R-Console` wird nun der Wert von `2*z` angezeigt:

```
[1] 16 26 42.
```

Speichern Sie die editierte Datei unter dem Namen *ersterSchritt.R* im Ordner *RFiles*. Dann können Sie sie bei einer späteren R-Session wieder öffnen und verwenden.

2 Grundlagen

2.1 Einführende Beispiele

Ziel: In diesem Kapitel lernen Sie anhand weniger Elemente der S-Sprache, einfache beschreibende Darstellungen von Zahlen- bzw. Datenmengen zu erzeugen.

Daten, data.frame. Statistische Auswertungen gehen in den meisten Fällen von einer „Datentabelle“ oder **Datenmatrix** aus. In einer Zeile dieser Tabelle stehen die Werte aller **Variablen** (oder Merkmale), die zu einer **Beobachtung** gehören. In S sind sie in einem so genannten `data.frame` gespeichert. Die Variablen können numerisch (quantitativ) oder nominal (kategorial) sein.

Daten einlesen. Um den ersten Datensatz für das System verfügbar zu machen, rufen wir die S-Funktion `read.table` auf. Der Befehl

```
> d.sport  <- read.table(
    "http://stat.ethz.ch/Teaching/Datasets/WBL/sport.dat", header=TRUE)
```

liest die Daten der Textdatei „sport.dat“ von der angegebenen Internetseite und speichert sie unter dem Namen `d.sport` ab.

Bemerkung:
Mit diesem Befehl greifen Sie über das Internet direkt auf den vom Seminar für Statistik für den Weiterbildungslehrgang zur Verfügung gestellten Datensatz „sport.dat“ zu. Sie könnten den Datensatz auch zuerst lokal speichern – indem Sie ihn auf der Webseite anklicken und dann mittels `Datei/Speichern` an einem geeigneten Ort, z.B. in einem Ordner `C:/Eigene Dateien/Datasets` ablegen. Der Befehl zum Einlesen des Datensatzes lautet dann:
`d.sport  <- read.table("C:/Eigene Dateien/Datasets/sport.dat", header=TRUE)`.

Wenn wir nun

```
> d.sport
```

in die R-Console schreiben, erscheint auf dem Bildschirm die Datenmatrix

```
           weit kugel hoch  disc stab speer punkte
OBRIEN     7.57 15.66  207 48.78  500 66.90   8824
BUSEMANN   8.07 13.60  204 45.04  480 66.86   8706
DVORAK     7.60 15.82  198 46.28  470 70.16   8664
:          :    :      :   :      :   :       :
:          :    :      :   :      :   :       :
:          :    :      :   :      :   :       :
CHMARA     7.75 14.51  210 42.60  490 54.84   8249
```

Damit haben wir bereits ein paar wichtige Grundelemente der S-Sprache benützt: Wir haben die Funktion `read.table` des Systems benützt und ihr Ergebnis mit dem Zuweisungspfeil $<-$ (ein Kleiner-Zeichen und ein Minus) dem von uns erfundenen Namen `d.sport` zugeordnet. Im zweiten Befehl haben wir das unter dem Namen `d.sport` gespeicherte „**Objekt**“ aufgerufen. Da wir auf dieser Zeile keine Zuweisung vorgenommen haben, wird das Ergebnis auf dem Bildschirm gezeigt.

Eine Variable auswählen.

```
> d.sport[,"kugel"]
```

wählt die Variable „kugel“ (also die Spalte mit dem Namen „kugel“) aus. Die eckigen Klammern wählen Teile von data.frames und anderen Objekten aus.

Eine Beobachtung auswählen.

```
> d.sport[5,]
```

zeigt die Werte aller Variablen für die 5. Beobachtung (also 5. Zeile). Genaueres in Abschnitt 2.5.

Ein Histogramm. Die Funktion `hist` zeichnet ein Histogramm:

```
> hist(d.sport[,"kugel"])
```

Funktions-Aufruf. Funktions-Aufrufe sind das zentrale Geschäft der Datenanalyse mit S. Funktionen haben **obligatorische Argumente**, die das Programm braucht, um etwas Sinnvolles zu tun. Zusätzlich gibt es meistens **freiwillige Argumente**. Werden diese weggelassen, so rechnet das Programm mit festgelegten, sinnvollen „Weglasswerten“, so genannten Defaults. Durch die Angabe von freiwilligen Argumenten können die festgelegten Defaults variiert werden. Beispielsweise kennt `hist` ein freiwilliges Argument `nclass`.

```
> hist(d.sport[,"kugel"], nclass=10)
```

produziert ein Histogramm mit ungefähr 10 Klassen.

Einige Funktionen haben keine obligatorischen Argumente. Dann muss man sie **mit leeren Klammern** aufrufen. Ein nützliches Beispiel ist

```
> objects()
```

(oder `ls()`), das Ihnen alle Objekte zeigt, die Sie selber (durch Zuweisung $<-$) erzeugt haben. Ein weiteres Beispiel: Wenn Sie die R-Session beenden wollen, rufen Sie die Funktion `q()` mit Klammern auf. **Wenn Sie die Klammern weglassen,** wird die Definition der Funktion auf dem Bildschirm gezeigt – was am Anfang verwirrend statt informativ sein kann.

Die Argumente einer Funktion haben eine bestimmte Reihenfolge und jedes hat auch einen Namen. Das erste Argument von `hist` besteht aus den Daten, die dargestellt werden sollen. Seinen Namen `x` haben wir weggelassen. Wir hätten auch schreiben können

```
> hist(x=d.sport[,"kugel"], nclass=10)
```

und hätten das Gleiche bewirkt. Für `nclass` haben wir den Namen angegeben, da wir nicht auswendig lernen wollen, dass `nclass` das 17. Argument ist. Es lohnt sich übrigens aus „Leserlichkeitsgründen“, die Namen der Argumente immer anzugeben.

Help. Wir wollen sowieso die Argumente der Funktionen nicht auswendig lernen. Wenn wir

```
> help(hist)
```

oder

```
> ?hist
```

eintippen, erscheint in einem Fenster eine Beschreibung der Funktion `hist` und all ihrer Argumente – allerdings mit viel Jargon, den Sie noch nicht verstehen müssen.

Oft hilft es, sich das Beispiel anzusehen, das auf der Help-Seite aufgeführt wird. Man kann das in R automatisch ausführen mit

```
> example(hist)
```

Fehlermeldungen. Fehlermeldungen sind leider nicht immer so verständlich, wie man sie gerne hätte. Vorläufig werden Sie ab und zu Hilfe brauchen, um sie zu interpretieren. Die S-Sprache ist so flexibel, dass Sie einiges schreiben können, was das System so halb versteht und deshalb vielleicht den Fehler am falschen Ort sucht.

Streudiagramme. Eine der grundlegendsten grafischen Darstellungen ist sicher das Streudiagramm.

```
> plot(d.sport[,"kugel"], d.sport[,"speer"])
```

trägt die Werte der Variablen `kugel` in x-Richtung und `speer` in y-Richtung auf. Es gibt natürlich viele freiwillige Argumente für die Funktion `plot`, mit denen das Streudiagramm verändert werden kann.

Streudiagramm-Matrix.

```
> pairs(d.sport)
```

erzeugt eine ganze Matrix von Streudiagrammen nach dem Prinzip „ jede Variable der Datenmatrix `d.sport` gegen jede“.

Graphik drucken. Im R gibt es eine Funktion

```
> dev.print()
```

die die Graphik, die gerade im aktiven Graphikfenster gezeigt wird, auf den Drucker schickt (sofern das richtig installiert wurde).

Session beenden. Tippen Sie

```
> q()
```

Das System fragt: Save workspace image? Wenn Sie „y“ antworten, werden Ihnen in der nächsten Session alle „Objekte“, die Sie in dieser Session erzeugt haben, wieder zur Verfügung stehen – und das ist gut so. Antworten Sie also `y`. Wenn man mit R-Studio arbeitet, schliesst man einfach R-Studio und nicht die (interne) R-Console einzeln.

2.2 Namensgebung

Die Namen der Objekte, die Sie selbst durch Zuweisung erzeugen, müssen mit einem Buchstaben beginnen und dürfen ausser dem Punkt (.) keine Spezialzeichen enthalten. (Wenn ein Name mit . beginnt, wird er mit `objects()` oder `ls()` nicht angezeigt.) Da auch alle Objekte, die das System zur Verfügung stellt (vor allem die Funktionen) solche Namen haben, verliert man leicht den Überblick. Besonders beliebt ist es, dem Buchstaben `c` etwas zuzuweisen und damit die grundlegende Funktion `c` mit einer anderen Funktion zu „überschreiben“. Man wird dann merkwürdige Meldungen erhalten.

In diesem Dokument verwenden wir folgende Konvention, um ein Durcheinander zu vermeiden:

> **Die Namen aller Objekte, beginnen mit einem Buchstaben, gefolgt von einem Punkt, gefolgt von weiteren Buchstaben und Ziffern.**

Die Buchstaben vor dem Punkt werden dazu benützt, die Art der Objekte zu unterscheiden. Beispielsweise sollen Daten mit `d.` beginnen – wie das für `d.sport` bereits angewandt worden ist.

Alles andere soll im Moment mit `t.` beginnen, da alle Objekte nur von *t*emporärer Bedeutung sind. Später werden wir dauerhaftere Resultate mit `r.` bezeichnen, selbst erzeugte Funktionen mit `f.` usw.

2.3 Vektoren

Vektoren erzeugen. Ein Vektor ist eine Zusammenfassung von Zahlen zu einem Objekt. Wir haben oben `d.sport[,"kugel"]` benützt.

```
> t.v  <- d.sport[,"kugel"]
> t.v
[1] 15.66 13.60 15.82 15.31 16.32 14.01 13.53 14.71 16.91 15.57 14.85 15.52
[13] 16.97 14.69 14.51
```

zeigt die Werte der Variablen `kugel` für alle Beobachtungen. Die Zahlen in eckigen Klammern am Anfang der Zeilen geben an, dem wievielten Element des Vektors die erste Zahl auf der Zeile entspricht; 16.97 ist also der Wert der 13. Beobachtung.

Man kann einem Vektor auch direkt Werte zuweisen, und zwar mit der Funktion `c` (concatenate):

```
> t.a  <- c(3.1, 5, -0.7, 0.9, 1.7)
```

(Die Funktion `c` folgt nicht dem üblichen Schema der Argumente: Man kann beliebig viele Argumente eingeben; sie werden alle zusammengehängt zum Resultat.)

Die Funktion `seq` erzeugt Zahlenfolgen mit gleicher Differenz,

```
> seq(0,3,by=0.5)
[1]  0.0  0.5  1.0  1.5  2.0  2.5  3.0
```

Für die wichtigsten Folgen dieser Art – aufeinanderfolgende ganze Zahlen – gibt es das spezielle Zeichen „:“

```
> 1:9
[1]  1  2  3  4  5  6  7  8  9
```

Das gleiche Ergebnis liefert `seq(1,9,by=1)` oder `seq(1,9)`.

Die Funktion `rep` erzeugt Vektoren mit immer wieder gleichen Zahlen. Die einfachste Version ist

```
> rep(0.7,5)
[1]  0.7  0.7  0.7  0.7  0.7
```

Es geht aber auch flexibler,

```
> rep(c(1,3,5),length=8)
[1]  1  3  5  1  3  5  1  3
```

Nun soll für solche Vektoren auch etwas ausgewertet werden. Tabelle 2.1 zeigt einige wichtige Funktionen, die auf numerische Vektoren anwendbar sind.

Aufruf, Beispiel	Bedeutung
`length(t.a)`	Länge, Anzahl Elemente
`sum(t.a)`	Summe aller Elemente
`mean(t.v)`	arithmetisches Mittel der Elemente
`var(t.v)`	empirische Varianz
`range(t.v)`	Wertebereich

Tabelle 2.1: Wichtige Funktionen für numerische Vektoren

Alphanumerische Vektoren. Wie jede Programmiersprache kann auch S mit „Wörtern“ oder „character strings“ umgehen. Man erzeugt sie zum Beispiel mit `c`,

```
> t.b  <- c("Andi", "Bettina", "Christian")
```

Eine nützliche Funktion ist `paste`, die ihre Argumente nötigenfalls in solche Strings verwandelt und dann zusammenhängt,

```
> paste("ABC","XYZ",17)
[1]   "ABC XYZ 17"
```

Was zwischen den strings steht, lässt sich mit dem Argument `sep` verändern,

```
> paste("ABC","IJK","XYZ",sep=":")
[1]   "ABC:IJK:XYZ"
```

Wenn die Argumente Vektoren sind, entsteht wieder ein Vektor,

```
> paste(c("a","b","c"),1:3)
[1]   "a 1"   "b 2"   "c 3"
```

Wenn man alle Elemente eines Vektors zusammenhängen will, muss man das Argument `collapse` brauchen:

```
> paste(letters, collapse="; ")
[1] "a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q; r; s; t; u; v; w; x;
```

```
y; z"
```

hängt das (im System unter `letters` gespeicherte) Alphabet zusammen.

Die Funktion `paste` ist also sehr flexibel. Wie `c` hat sie beliebig viele Argumente. Deshalb müssen die speziellen Argumente `sep` und `collapse` – von denen man jeweils nur eines benützen kann – mit ihren Namen angesprochen werden.

Logische Vektoren. Neben numerischen und alphanumerischen Vektoren gibt es logische, deren Elemente nur die Werte `TRUE` oder `FALSE` haben können. Sie werden sich im nächsten Abschnitt als sehr nützlich erweisen. Sie entstehen meistens durch Vergleichsoperationen,

```
> (1:5)>=3
[1]    FALSE    FALSE    TRUE    TRUE    TRUE
```

Für das erste und zweite *Element* von `(1:5)` ist die Ungleichung *nicht* erfüllt (`FALSE`), für die letzten drei ist sie erfüllt (`TRUE`). Die Vergleichsoperationen werden geschrieben als `<, <=, >, >=, ==, !=`. Beachten Sie, dass das „vergleichende Gleich“ mit zwei Gleichheitszeichen geschrieben werden muss, da das einfache = zur Identifikation der Argumente von Funktionen gebraucht wird.

(Siehe auch `help("Comparison")`.)

Die logischen Operationen heissen `&` (und), | (oder), `!` (nicht).

```
> t.i  <- (t.v>2)&(t.v<5)
```

ergibt `TRUE` an den *Stellen der Elemente* von `t.v`, deren Werte zwischen 2 und 5 liegen.

2.4 Arithmetik

Selbstverständlich kann man mit S auch rechnen,

```
> 2+5
[1]  7
```

Die Grundoperationen heissen `+ , - , * , /` . Das Zeichen `^` bedeutet „hoch“.

Auf Vektoren werden die Operationen **elementweise** angewandt,

```
> (2:5) ^ c(2,3,1,0)
[1]   4  27   4   1
```

Die Prioritäten der Operationen sind die üblichen. Klammern setzen ist im Zweifelsfall sehr nützlich.

Der eine der beiden Operatoren kann nur eine Zahl sein,

```
> (2:5) ^ 2
[1]   4   9  16   25
```

Sind beide Operatoren Vektoren, aber von unterschiedlicher Länge, so wird der kürzere auf die Länge des längeren gebracht, indem er zyklisch wiederverwendet wird,

```
> (1:5)-(0:1)
[1] 1 1 3 3 5
```

Weil es hier nicht aufgeht, produziert S die Warnung

```
Warning message:
longer object length is not a multiple of shorter object length in:
(1:5) - (0:1)
```

(Das kann nützlich sein – ungeschickt, wenn es zufälligerweise aufgeht und die Warnung nützlich gewesen wäre!)

Siehe auch `help("Arithmetic")`.

2.5 Elemente auswählen

In der Statistik will man oft nur Teile von gesammelten Daten bearbeiten. Wir haben oben schon eine Spalte oder eine Zeile eines data.frames ausgewählt (Abschnitt 2.1). Die Auswahl erfolgt mit den eckigen Klammern `[ ]`. Diese werden auch gebraucht, um Teile von Vektoren zu erhalten.

Mit Hilfe von Vektoren können grössere Teile ausgewählt werden. Es gibt 3 Varianten:

- Indices (ganze Zahlen):

```
> t.v[c(1,3,5)]
[1] 15.66 15.82 16.32
> d.sport[c(1,3,5),1:3]
          weit kugel hoch
OBRIEN     7.57 15.66  207
DVORAK     7.60 15.82  198
HAMALAINEN 7.48 16.32  198
```

- Logische Vektoren:

```
> t.a[c(TRUE,FALSE,TRUE,TRUE,FALSE)]
[1] 3.1 -0.7 0.9
> d.sport[t.v > 16,c(2,7)]
          kugel punkte
HAMALAINEN 16.32   8613
PENALVER   16.91   8307
SMITH      16.97   8271
```

  Der logische Vektor muss gleich viele Elemente haben wie der Vektor, aus dem ausgewählt wird, oder wie das data.frame Zeilen resp. Spalten hat.

- Bei data.frames: Namen der Zeilen oder Spalten

```
> d.sport[c("OBRIEN","DVORAK"),c("kugel","speer","punkte")]
       kugel speer punkte
OBRIEN 15.66 66.90   8824
DVORAK 15.82 70.16   8664
```

Bemerkung: Wenn man eine einzige Variable eines data.frames bearbeiten oder benützen will, kann man auch mit Hilfe des Dollarzeichens auf sie zugreifen:

```
> d.sport$kugel
```

ruft die Variable `kugel` des data.frames `d.sport` auf.

2.6 Verteilungen und Zufallszahlen

Für alle gebräuchlichen Wahrscheinlichkeits-Verteilungen enthält S die entsprechenden Funktionen:

- Die Wahrscheinlichkeiten $P(X = 0), ..., P(X = 5)$ für die Binomialverteilung Bin($n = 5, \pi = 0.7$) erhält man durch

  ```
  > dbinom(0:5, size=5, prob=0.7)
  [1] 0.00243 0.02835 0.13230 0.30870 0.36015 0.16807
  ```

- Die Funktion `pbinom(0:5, size=5, prob=0.7)` liefert die kumulative Verteilungsfunktion,
- `qbinom(seq(0.1,0.9,0.1), size=5, prob=0.7)` gibt die Quantile an. (`qbinom` ist also die Umkehrfunktion von `pbinom`.)
- Binomialverteilte Zufallszahlen (mit $n = 5$ und $\pi = 0.7$) erhält man mit

  ```
  > rbinom(20, size=5, prob=0.7)
  [1] 2 2 4 5 5 4 3 4 2 3 5 5 4 4 3 5 3 4 3 3
  ```

  Mit jedem erneuten Aufruf erhalten Sie ein anderes Resultat – eben ein (pseudo-) zufälliges.

Für die Poissonverteilung geht das genau gleich: `dpois, ppois, qpois, rpois`. Zum Beispiel liefert

```
> rpois(20, lambda=3.5)
```

20 Zufallszahlen, die (pseudo-) poissonverteilt sind mit Parameter $\lambda = 3.5$.

Die Namensgebung ist nicht gerade ein Glücksfall:

- Die Autoren des ursprünglichen S, die nicht an eine so grosse Verbreitung dachten, waren wohl etwas zu schreibfaul. Man könnte sich wohl pbinomial besser merken als pbinom, und rpoisson besser als rpois.
- Wieso steht d für einzelne Wahrscheinlichkeiten und p für kumulative? Für Mathematiker sind Einzel-Wahrscheinlichkeiten „*D*ichten bezüglich des Zählmasses“. Für stetige Verteilungen liefert die „d-Funktion“ die Dichte $f(x)$ im üblichen Sinn. Die kumulative Verteilungsfunktion dient zur Berechnung der Wahrscheinlichkeit von Ereignissen, was den Buchstaben p einigermassen erklären kann.

Tabelle 2.2 listet gebräuchliche Verteilungen und ihre Namen in S auf.

Im Zusammenhang mit der Binomialverteilung treten die Zahlen „n tief k“ auf – auf wie viele Arten lassen sich k Elemente aus n auswählen. Sie werden von der Funktion `choose` geliefert.

Diskrete Verteilungen	
`binom`	Binomial-V.
`pois`	Poisson-V.
`hyper`	hypergeometrische V.
Stetige Verteilungen	
`unif`	uniforme V.
`norm`	Normalverteilung
`exp`	Exponentialverteilung
`lnorm`	Log-Normalverteilung
`t, F, chisq`	t-, F-, χ^2- (Chiquadrat-)Verteilung
`weibull, gamma`	Weibull-, Gamma-Verteilung

Tabelle 2.2: Verteilungen

Zufallszahlen. Die Funktion `runif` liefert uniform verteilte Zufallszahlen.

```
> runif(4)
[1] 0.81341 0.04849 0.17556 0.02286
```

Bei nochmaligem Aufruf werden neue Zufallszahlen produziert

```
> runif(2)
[1] 0.1241 0.2918
```

Manchmal möchte man Ergebnisse exakt reproduzieren können. Mit der Funktion `set.seed` kann man den „Startwert“ des Zufallszahlengenerators festlegen, so dass bei einem neuen Aufruf von z.B. `runif` wieder die gleichen Pseudo-Zufallszahlen produziert werden,

```
> set.seed(27)
> runif(1)
[1] 0.8573463
> set.seed(27)
> runif(1)
[1] 0.8573463
```

Darstellung von Verteilungen. Diskrete Verteilungen stellt man am besten mit `plot(...,type="h")` dar,

```
> plot(0:15,dpois(0:15,lambda=3.5), type="h", lwd=3)
```

(Das Argument `lwd` legt die Dicke (line width) von Linien fest.)

Dichtefunktion einer Normal- und einer Log-Normal-Verteilung:

```
> t.x  <- seq(0,10,length=100)[-1]
> plot(t.x,dnorm(t.x, 5, 2), type="l", xlab="x", ylab="Dichte",
  main="Normalverteilung")
> plot(t.x,dlnorm(t.x, log(3), log(1.7)), type="l", xlab="x",
  ylab="Dichte", main="Log-Normalverteilung")
```

Alternativ kann man auch den Befehl `curve` verwenden.

2.7 Einfache Statistik-Funktionen

Hier wollen wir einige Statistik-Funktionen von S vorstellen, die für grundlegende Problemstellungen wie Ein- und Zwei-Stichproben-Test gebraucht werden.

Tests und Vertrauensintervalle. Für Tests und Vertrauensintervalle für den Parameter der Binomial-Verteilung ist die Funktion `binom.test` da,

```
> binom.test(3,20, p=0.4)
```

liefert ein ausführliches Resultat mit P-Wert des Tests auf $\pi = 0.4$ und einem Vertrauensintervall.

Für die Poisson-Verteilung gibt es keine solche Funktion. Einen einseitigen P-Wert erhält man über die kumulative Verteilungsfunktion, die, wie bereits besprochen, von der Funktion `ppois` berechnet wird:

```
> t.x  <- 5
> 1-ppois(t.x-1,lambda=2.7)
[1] 0.1370921
```

Für zwei unabhängige Stichproben liefert `wilcox.test` die Resultate des Rangsummentests und `t.test` diejenigen des t-Tests,

```
> t.hh  <- d.sport[,"hoch"]>200
> t.kugel  <- d.sport[,"kugel"]
> wilcox.test(t.kugel[t.hh],t.kugel[!t.hh])
        Wilcoxon rank sum test
data:  t.kugel[t.hh] and t.kugel[!t.hh]
W = 20, p-value = 0.4559
alternative hypothesis: true mu is not equal to 0

> t.test(t.kugel[t.hh],t.kugel[!t.hh], var.equal = TRUE)

        Two Sample t-test

data:  t.kugel[t.hh] and t.kugel[!t.hh]
t = -0.8066, df = 13, p-value = 0.4344
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -1.694  0.773
sample estimates:
mean of x mean of y
    15.01     15.47
```

Für gepaarte Stichproben oder eine einzelne Stichprobe liefern die gleichen Funktionen mit dem Argument `paired=TRUE` die Resultate.

Grafische Darstellungen für zwei Stichproben. Will man zwei Stichproben grafisch vergleichen, dann kann man das recht summarisch mit Boxplots tun,

```
> boxplot(t.kugel[t.hh],t.kugel[!t.hh],notch=TRUE)
```

Eleganter geht es mit der Funktion `split`:

```
> boxplot(split(t.kugel,t.hh), notch=TRUE)
```

Zwei Histogramme erhält man – eher kompliziert – mit

```
> t.br <- 13:17
> t.h1 <- hist(t.kugel[t.hh],breaks=t.br, plot=FALSE)
> t.h2 <- hist(t.kugel[!t.hh],breaks=t.br, plot=FALSE)
> barplot(rbind(t.h1$density,t.h2$density), beside=TRUE,
          names.arg=t.h1$mid)
```

Viele gleiche Rechnungen. Oft will man für mehrere Teile des Datensatzes immer wieder das Gleiche tun. Beispielsweise soll für alle Variablen das Mittel gerechnet werden. Solches erledigt die Funktion `apply`

```
> apply(d.sport,2,mean)
   weit    kugel     hoch     disc     stab    speer   punkte
  7.597   15.199  202.000   46.376  498.000   61.995 8444.667
```

Das erste Argument von `apply` bestimmt den Datensatz. Das dritte sagt, welche Funktion jeweils ausgewertet werden soll. Das zweite Argument gibt an, ob diese Funktion auf jede Zeile (1) oder jede Spalte (2) angewandt werden soll.

Man kann auch eine Funktion auf mehrere Untergruppen einer Variablen anwenden. Dies besorgen die Funktionen `by` und `aggregate`.

Wenn Sie eine „konventionelle“ Programmiersprache kennen, würden Sie solche Aufgaben wohl über eine Schleife (loop) lösen. Später werden wir solche Programmierelemente einführen – und gleichzeitig „predigen“, dass man sie vermeiden soll.

2.8 Fehlende Daten

In vielen Anwendungsgebieten der Statistik bilden vollständige Datensätze die grosse Ausnahme. Wenn eine einzelne Variable beobachtet wird, ist der Fall einfach: Die „missglückten“ Beobachtungen werden gar nicht aufgeschrieben oder nicht in S eingelesen (was zu Verfälschungen führen kann!). Wenn mehrere Variable gemessen oder beobachtet werden sollen, dann fehlt immer wieder einmal ein Wert. Ein solcher fehlender Wert wird innerhalb von S mit `NA` (not available) bezeichnet.

Beim Importieren von Daten mit `read.table` kann man die in der externen Datei verwendeten Bezeichnungen für fehlende Daten als Argument `na.strings` angeben. Man kann auch NAs setzen,

```
> t.kugel[2]  <- NA
> t.kugel[t.kugel<14]  <- NA
```

```
> t.kugel
[1] 15.66 NA 15.82 NA 16.32 14.01 NA 14.71 16.91 ...
```

Die Funktion `is.na` stellt fest, welche Elemente eines Vektors „fehlen“,

```
> is.na(t.kugel)
[1] FALSE TRUE FALSE TRUE FALSE FALSE TRUE FALSE FALSE ...
```

Mit dem Vergleichsperator „`==`“ funktioniert es nicht! Wenn wir z.B. die `NA`'s durch 0 ersetzen wollen, so können wir dies mit dem Befehl

```
> t.kugel[is.na(t.kugel)]  <- 0
```

erledigen.

Die S-Funktionen reagieren auf fehlende Daten unterschiedlich. Viele Grafik-Funktionen lassen die Daten einfach weg. Arithmetische Operationen liefern NA, wenn ein Argument NA ist. Bei rechnerischen Funktionen, die einen Vektor zusammenfassen, gibt es meistens ein Argument `na.rm` (remove NAs?), das man `TRUE` setzen kann. Wenn wir mit dem ursprünglichen `t.kugel` (inkl. NAs) arbeiten, so erhalten wir

```
> sum(t.kugel)
[1] NA
> sum(t.kugel, na.rm=TRUE)
[1] 214.4
```

Ebenso funktionieren: `mean, median, min, max, range`. Die Funktion `var` liefert nicht NA, sondern eine Fehlermeldung, wenn die Daten fehlende Werte enthalten. „Höhere“ Funktionen, wie beispielsweise `wilcox.test` und `t.test`, haben ein Argument `na.action`, für das eine Funktion eingesetzt wird. Der Default ist die Funtion `na.omit`, die (wie `na.rm=TRUE` bei `sum`) die fehlenden Werte einfach weglässt.

3 Grundelemente der Sprache S

In diesem Kapitel werden einige grundlegende Bemerkungen zum Aufbau und zu Eigenheiten von S angeführt. Sie sollen das Verständnis erleichtern. Es reicht aber, das Kapitel diagonal zu lesen, da es nichts enthält, was Sie am Anfang unbedingt wissen müssen.

3.1 Funktionen schreiben

Funktionen können Sie nicht nur aufrufen, Sie können auch selber welche schreiben. Als ganz einfaches Beispiel schreiben wir eine Funktion, die das Maximum eines Vektors (einer Variablen) bestimmt und gleichzeitig angibt, das wie vielte Element dieses Maximum liefert. (Bei gleichen Werten soll der erste gelten.) Wir brauchen neben `max` die Funktion `match`, die dieses Element bestimmt, und setzen also eigentlich nur diese beiden Funktionen zusammen.

```
> f.maxi  <- function(data, na.remove=TRUE) {
>   l.max  <- max(data, na.rm=na.remove)
>   l.i  <- match(l.max, data)
>   c(max=l.max, i=l.i)
> }
```

Eine Funktions-Definition beginnt mit dem Wort `function`, gefolgt von den Argumenten in Klammern, wie wir sie von den Aufrufen her kennen. Das zweite Argument soll angeben, wie mit den fehlenden Werten umgegangen werden soll. Da ein Default-Wert (`TRUE`) gegeben ist, wird man dieses Argument bei einem Aufruf nicht angeben müssen.

Dann folgen in geschweiften Klammern die beiden Befehle, die die Aufgabe der Funktion erledigen, und als letzte Zeile wird das Ergebnis zusammengestellt zu einem Vektor mit zwei Elementen. Wir haben etwas vorgegriffen und den Vektor mit Namen (max, i) für seine Elemente versehen.

Nun können wir die Funktion aufrufen,

```
> f.maxi(t.kugel)
  max     i
16.97 13.00
```

Nun, die Funktion ist grundlegend genug, dass es sie im R eigentlich schon gibt: `which.max` liefert das zweite Element des Resultats.

Eine Funktion zu schreiben, nur um zwei Befehle zusammen zu fassen, scheint nicht gerade nützlich zu sein. Oder doch? Wir können diese Funktion nun zum Beispiel in `apply` brauchen,

```
> apply(d.sport,2,f.maxi)
    weit kugel hoch  disc stab speer punkte
max 8.07 16.97  213 49.84  540 70.16   8824
i   2.00 13.00    8  4.00    6  3.00      1
```

3.2 Schlaufen und Bedingungen

Schlaufe. Schlaufen werden nötig, wenn man iterative Verfahren oder rekursive Berechnungen ausführen will. Die Fibonacci-Zahlen sind gegeben durch die Rekursionsformel

$$n_{k+1} = n_{k-1} + n_k$$

mit den Anfangswerten $n_1 = 1$ und $n_2 = 1$. Eine Schlaufe, die `m` weitere Zahlen berechnet, kann so aussehen:

```
> n  <- c(1,1)
> for (i in 1:m){
>   n  <- c(n, n[i] + n[i+1])
> }
```

Die `for`-Schlaufe beginnt immer mit `for(... in ...)`. Nach dem Wort `in` steht ein Vektor, der die Werte enthält, die die Laufvariable durchläuft. Das sind oft die Zahlen 1, 2, 3, ... m, also der Vektor `1:m`, es kann aber auch ein beliebiger Vektor sein, wie `c("Mo","Mi","Fr")`.

Es gibt auch Schlaufen ohne Laufvariable, mit den Schlüsselwörtern `while` und `repeat`. Wir diskutieren sie hier nicht.

Bedingungen. Fallunterscheidungen werden üblicherweise mit `if (...)` vorgenommen. Sie sind beispielsweise nützlich, wenn man eine Schlaufe unter gewissen Bedingungen abbrechen will. Man kann beispielsweise in der Fibonacci-Reihe abbrechen – mit dem Schlüsselwort `break` –, wenn die Zahl grösser als eine Million wird:

```
> n  <- c(1,1)
> for (i in 1:m) {
>   n  <- c(n, n[i]+n[i+1])
>   if (max(n) > 1E6) break
> }
```

Verwendung in Funktionen.

```
if (any(x<0)) stop ("Negative numbers not allowed")

if (length(names)==0) names  <- paste("X",1:length(data),sep="")
```

Schlaufen vermeiden! Die meisten Schlaufen, die man mit wenig Erfahrung in einer Vektor- und Matrix-orientierten Programmiersprache benützen will, sind unnötig und sollten vermieden werden. Zunächst wirken viele Funktionen ja elementweise auf Vektoren, beispielsweise liefert `pmax(t.v,0)` für jedes negative Element von `t.v` eine 0 und lässt die positiven unverändert. Zudem sind die in 2.7 erwähnten Funktionen `apply`, `aggregate` und `by` besonders nützlich zur Vermeidung von expliziten Schlaufen. Generell sind Schlaufen eigentlich nur nötig, wenn das Ergebnis eines Durchlaufs im nächsten Durchlauf benützt wird, also, wie erwähnt, für rekursive Berechnungen und iterative Verfahren.

3.3 Fehlermeldungen

Fehlermeldungen sind hin und wieder sehr einfach zu verstehen. Andernfalls ist Ihre Phantasie gefragt.

Häufig rufen S-Funktionen (oder diejenigen, die Sie selber schreiben) wieder andere Funktionnen auf, und die Fehlermeldung entsteht in einer Funktion, die man nie selber direkt aufgeruft hat. Die Funktion

```
> traceback()
```

liefert Ihnen die Angaben, welche Funktionen welche weiteren aufgerufen hatten, bevor der Fehler geschah.

Wenn Sie selber Funktionen mit bleibendem Wert schreiben, ist es wichtig, dass Sie auch Fehlermeldungen liefern. Beispielsweise führt

```
if (!is.vector(data)) stop("argument data must be a vector")
```

dazu, dass das Programm mit der angegebenen Fehlermeldung abbricht, wenn beim Aufruf der Funktion ein Argument `data` gegeben wird, das kein Vektor ist. (Genaueres zu `if` und `is.vector` folgt.)

Im Moment lassen wir es bei diesen wenigen Bemerkungen zum Stichwort Fehlermeldungen und drücken Ihnen die Daumen.

3.4 Objekte

Bisher haben wir die verschiedenen Arten von „Objekten“, die die S-Sprache kennt, nur so weit eingeführt, als sie gerade gebraucht wurden. Sie kennen bereits Vektoren, Dataframes und Funktionen. Nun wollen wir Ihnen die wichtigen weiteren Strukturen vorstellen. Viele von ihnen werden Sie nicht aktiv brauchen, aber es wird Ihnen das Verständnis von vorhandenen Funktionen, Skript-Files oder von Fehlermeldugen erleichtern, wenn Sie die Strukturen besser kennen.

Listen. Eine Liste (`list`) ist die Zusammenfassung irgendwelcher Objekte zu einem „Superobjekt“, das dann unter einem einzigen Namen angesprochen werden kann. Viele nützliche Funktionen liefern nicht nur eine Zahl oder einen Vektor als Resultat, sondern eine ganze Liste von Resultaten.

Ein Beispiel bildet die Funktion `hist`, die nicht nur zum Zeichnen eines Histogramms, sondern auch zur Bestimmung der entsprechenden Häufigkeiten verwendet werden kann, indem man das Argument `plot=FALSE` setzt,

```
> t.l  <- hist(t.kugel,plot=FALSE)
> t.l
$breaks
[1] 13 14 15 16 17
$counts
[1] 2 5 5 3
...
```

```
$equidist
[1] TRUE
attr(,"class")
[1] "histogram"
```

Die letzten beiden Zeilen werden später erklärt (Attribute, Klassen). Die vorhergehenden Zeilenpaare geben jeweils den Namen nach dem $-Zeichen (vgl. Spalten von Dataframes) und den Inhalt der Elemente der Liste wieder.

Einen Teil einer Liste wählt man aus wie einen Teil eines Vektors: `t.l[2:3]` zeigt die Elemente `$counts` und `$intensities`, `t.l[c("breaks","intensities")]` benützt die Namen der Elemente.

Wählt man nur ein Element der Liste aus (`t.l["counts"]`, so entsteht eine Liste mit nur einem Element. Das ist ab und zu sinnvoll. Meistens aber möchte man dann das Element ohne „Superstruktur“, hier also direkt den Vektor, der die Anzahlen enthält. Das erreicht man auf zwei Arten: mit Doppelklammer oder mit dem $-Zeichen und dem Namen,

```
> t.l[[2]]    oder  t.l[["counts"]]  oder  t.l$counts
[1] 2 5 5 3
```

(Das $-Zeichen tritt deshalb auch bei der Ausgabe auf.)

Matrizen. Matrizen sind eine vereinfachte Version von Dataframes. Sie können nur Daten vom gleichen Typ (mode, siehe unten) enthalten, also entweder nur numerische, nur logische oder nur character Daten. Matrizen werden mit der Funktion `matrix` erzeugt,

```
> matrix(1:15, nrow=3)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15
```

Man sieht, dass die Matrix spaltenweise mit den als erstes Argument gegebenen Daten gefüllt wird. Will man zeilenweise füllen, dann setzt man das Argument `byrow=TRUE`. Man kann statt der Zeilenzahl `nrow` auch die Spaltenzahl `ncol` angeben – oder beides:

```
> matrix(NA, nrow=2, ncol=3)
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
```

Ein Dataframe kann man in eine Matrix verwandeln mit `as.matrix`, beispielsweise

```
> t.sportmat  <- as.matrix(d.sport)
```

Wenn allerdings nur eine Spalte kategoriell (ein „Faktor“) ist, wird die ganze Matrix eine character-Matrix. Die Funktion `data.matrix` dagegen codiert solche Variablen durch ganze Zahlen und liefert immer eine numerische Matrix.

Matrizen entstehen auch, wenn man Spalten oder Zeilen mit den Funktionen `cbind` respektive `rbind` „zusammenklebt“.

```
> cbind(4:6,13:15)
     [,1] [,2]
[1,]    4   13
[2,]    5   14
[3,]    6   15
```

Mit den Funktionen `nrow(t.m), ncol(t.m), dim(t.m)` erhält man die Anzahl der Zeilen, der Spalten bzw. beides zusammen.

```
> dim(t.m)
[1] 3 5
```

Die Auswahl von Elementen geht genau wie bei Dataframes:

```
> t.m  <- matrix(1:15, nrow=3)
> t.m[2,1:3]
[1] 2 5 8
```

Damit die Auswahl auch hier mit Namen von Spalten oder Zeilen erfolgen kann, müssen solche Namen zuerst zugeordnet werden. Wie das geht, wird weiter unten gesagt.

Arrays. Arrays sind „höherdimensionale Matrizen“. Man erzeugt sie mit der Funktion `array`. Wir wollen hier nicht näher auf diese Objekte eingehen – sie werden selten gebraucht.

Andere Objekte. Zwei weitere Typen, `formula` und `expression`, werden wir später kennen lernen. Es gibt noch weitere Typen von Objekten, die man als Benützer kaum je braucht. Sie werden innerhalb der S-Funktionen verwendet.

Namen für Elemente, Zeilen und Spalten. Den Elementen eines Vektors oder einer Liste kann man Namen geben,

```
> t.note  <- c(4.5, 6, 3.9)
> names(t.note)  <- c("Anna","Berta","Christa")
> t.note["Berta"]
Berta
    6
```

Das kennen wir ja schon von den Spalten eines Dataframes. Wenn man `names(t.d)` für ein Dataframe `t.d` abfragt, erhält man die Namen der Variablen.

Die Syntax von S ist hier erstaunlich flexibel. In kaum einer anderen Sprache kann links vom Zuweisungszeichen (`<-`) etwas in Klammern erscheinen. Es kommt noch besser:

```
> names(t.note)[2]  <- "Berti"
```

korrigiert den zweiten Namen.

In Dataframes haben auch die Zeilen Namen. Man kann sie mit der Funktion `row.names` abfragen oder ändern. Wenn man nichts angibt, werden die Zeilen bei der Erzeugung des Dataframes durchnummeriert. Wenn man Zeilen weglässt, bleiben die row.names erhalten.

Man erhält dann Merkwürdiges:

```
> t.d  <- data.frame(fac=c("a","a","b"),note=c(4.5, 6, 3.9))
> t.dd  <- t.d[-1,]
> t.dd[2,"note"]
[1] 3.9
> t.dd["2","note"]
[1] 6
```

Bei genauerem Überlegen ist das für Datenanalysen sehr nützlich: Wenn gewisse Beobachtungen aus einem Dataframe gestrichen werden, kann man nachher immer noch die ursprünglichen Beobachtungsnummern ansprechen. Die Verwirrung kann man vermeiden, wenn man die `row.names` selber festlegt;

```
> row.names(t.d)  <- LETTERS[1:nrow(t.d)]
```

benützt die Grossbuchstaben der Reihe nach zu diesem Zweck (falls `t.d` höchstens 26 Zeilen hat).

Für Matrizen läuft die Namensgebung etwas anders: Man muss die Namen für Zeilen und Spalten zu einer Liste zusammenfassen und schreiben

```
> dimnames(t.m)  <-
  list(c("Anna","Berta","Christa"),paste("Var.",1:5,sep=""))
```

Jetzt lassen sich Elemente wie für Dataframes mit Namen ansprechen,

```
> t.m["Berta",]
Var.1 Var.2 Var.3 Var.4 Var.5
    2     5     8    11    14
```

In R können die Namen von Zeilen und Spalten von Dataframes und Matrizen einheitlich mit `rownames, colnames` und `dimnames` festgelegt und abgefragt werden.

Attribute. Nützliche Informationen, die zu einem Objekt gehören, können als so genannte Attribute „an die Objekte angehängt“ werden. Die oben vorgestellten Namen sind ein Beispiel dafür. Wenn Sie

```
> attributes(d.sport)
```

tippen, sehen Sie, dass die Attribute eines Dataframes eine Liste bilden mit den Namen `names`, `class` und `row.names`. Auf das zweite kommen wir unter 3.5 zurück, die anderen beiden kennen Sie.

Man kann auch selber Attribute einführen. Beispielsweise erzeugt

```
> attr(d.sport,"doc")  <- "Leichtathletik-Resultate"
```

ein Attribut mit dem Namen `doc`, das dem Datensatz `d.sport` angehängt bleibt und mit `attr(d.sport,"doc")` wieder abgerufen werden kann (und auch bei `attributes(d.sport)` erscheint).

Solche Informationen könnten auch mit gespeichert werden, indem alles in eine Liste „versorgt“ wird. Der Vorteil von Attributen besteht darin, dass die „Hauptinformation“ nicht als Teil einer Liste, sondern direkt anzusprechen ist.

Drei Attribute sind bei allen Objekten vorhanden. Die ersten beiden sind `length` und `mode`. `length` gibt die Anzahl Elemente des Objekts an, und `mode` sagt, von welcher Art die Elemente sind,

```
> mode(t.kugel)
[1] "numeric"
```

Es gibt die folgenden Modes: character, numeric, logical, complex, list, function, expression, call. Diese beiden Attribute werden von `attributes` nicht angezeigt. Das dritte Attribut, das immer festgelegt ist, heisst `class`. Für einfache Objekte ist `class` oft mit dem `mode` identisch. Wir kommen auf `class` zurück.

Ein Objekt von bestimmtem Typ erzeugen. Wie die Funktionen `matrix` eine Matrix, `list` eine Liste und `function` eine Funktion erzeugen, so gibt es auch die Funktionen `array, data.frame, vector`. Einen Vektor mit einem bestimmten Mode kann man ähnlich erzeugen,

```
> t.c  <- character(3)
> t.c
[1] ""  ""  ""
```

Das Argument von `character` gibt die gewünschte Länge an; der Vektor wird mit „leeren Buchstaben“ gefüllt.

```
> numeric(0)
```

erzeugt einen „leeren Vektor“ mit Mode `numeric`. Solche merkwürdigen Gebilde werden meist nur gebraucht, wenn man Funktionen schreibt und darin Schlaufen programmiert.

Was ist was? Oft fragt man sich, welche Art von Objekt sich hinter einem Namen versteckt.

```
> is.matrix(d.sport)
[1] FALSE
> is.data.frame(d.sport)
[1] TRUE
```

zeigt, dass `d.sport` keine Matrix ist, sondern ein Dataframe.

Es gibt viele `is.`-Funktionen, beispielsweise `is.numeric, is.logical, is.list, ...`

Eine sehr nützliche Funktion, die versucht, die Struktur eines Objektes vollständig anzugeben, ist `str`,

```
> str(d.sport)
 'data.frame':      15 obs. of  7 variables:
$ weit  : num  7.57 8.07 7.6 7.77 7.48 7.88 7.64 7.61 7.27 7.49 ...
$ kugel : num  15.7 13.6 15.8 15.3 16.3 ...
$ hoch  : num  207 204 198 204 198 201 195 213 207 204 ...
$ disc  : num  48.8 45.0 46.3 49.8 49.6 ...
$ stab  : num  500 480 470 510 500 540 540 520 470 470 ...
$ speer : num  66.9 66.9 70.2 65.7 57.7 ...
$ punkte: num  8824 8706 8664 8644 8613 ...
- attr(*, "doc")= chr "Leichtathletik-Resultate"
```

Wieso so kompliziert? Man würde doch auch sehen, welche Art von Objekt `d.sport` ist, indem man einfach `d.sport` eingibt! – Abgesehen davon, dass Objekte so gross sein können, dass man nur noch viele Bildschrime voll von Zahlen und Buchstaben vorbeiziehen sieht, gibt eine solche Eingabe oft ein falsches oder unvollständiges Bild dessen, welche Informationen in einem Objekt wirklich enthalten sind. Der Grund liegt darin, dass die Funktion `print`, die beim Eintippen des Objektnamens (oder einem anderen Befehl ohne Zuweisung) vom System aufgerufen wird, versucht, die Objekte in einer benützerfreundlichen Form zu zeigen. Wir kommen auf diesen Punkt gleich zurück.

Umwandlungen. Das Dataframe `d.sport` könnte auch als Matrix gespeichert werden – und zwar als numerische, da es nur Zahlen enthält. Das geschieht mit einer „as-Funktion“,

```
> t.sport.mat  <- as.matrix(d.sport)
```

(oder `data.matrix(d.sport)`, siehe oben). In diesem Beispiel macht das wenig Sinn. Ein Vorteil kann darin bestehen, dass man dann Matrix-Operationen anwenden kann – was in der Form eines Dataframes nicht geht. Analoge Funktionen, die mit `as.` beginnen, gibt es für alle Objekttypen und Modes. Sie tun ihr Bestes, ein Objekt an die gewünschte Form anzupassen, und geben bei Misserfolg eine Fehlermeldung.

Eine nützliche Funktion, die nicht diesem Schema folgt, ist `unlist`. Sie hängt alle Elemente einer Liste zusammen, so dass sie einen einzigen Vektor bilden,

```
> unlist(list(a=1:2, b=5:7))
a1 a2 b1 b2 b3
 1  2  5  6  7
```

3.5 Objektorientierte Programm-Strukturen

Klassen. S erlaubt eine bestimmte Art von so genannt „objekt-orientiertem Programmieren“, unter Beibehaltung des so genannten „functional paradigm“. Das Grundelement des objekt-orientierten Programmierens ist die Definition von Objekt-Klassen. Eine solche Definition legt fest, welche „Teile“ in einem Objekt, das zur Klasse gehört, vorhanden sein müssen oder können. In S zeigt das Attribut `class`, zu welcher Klasse ein Objekt gehört.

Das Beispiel einer Klasse, das Sie schon kennen, sind die Dataframes. Die Abfrage

```
> class(t.d)
[1] "data.frame"
```

zeigt die Klasse eines Objektes. Ein Dataframe besteht aus einer Liste von gleich langen Vektoren, die die Variablen enthalten, und deren Namen (`names(t.d)`). Zusätzlich ist das Attribut `row.names` vorhanden. Das ist auch schon alles, was zur Definition gehört.

Weitere Beispiele von Klassen, die bereits erwähnt wurden, sind `hist`, erzeugt von der gleichnamigen Funktion, und `htest`, zu der die Resultate von `wilcox.test` und `t.test` gehören.

Generische Funktionen. Der Witz am objekt-orientierten Programmieren besteht darin, dass Funktionen und Objektklassen aufeinander abgestimmt sind. In der in S implementierten Variante reagieren viele „höhere“ Funktionen auf die Klasse von Argumenten – meistens auf die Klasse des ersten Arguments. Sie heissen dann „generische Funktionen“ und bilden eigentlich eine ganze Funktionen-Familie.

Der Prototyp einer solchen Funktion ist `print` – die Funktion, die jeweils aufgerufen wird, wenn in einem Befehl keine Zuweisung erfolgt. Je nach Klasse des anzuzeigenden Objekts wird dieses anders dargestellt. Das Dataframe `t.d`, „angeschrieben“ mit dem `class`-Attribut `data.frame`, wird anders dargestellt als eine Liste ohne dieses Attribut. Die generische Funktion `print` stellt zunächst fest, welches `class`-Attribut das erste Argument hat, und wählt dementsprechend das „Mitglied der print-Funktionen-Familie“, das wirklich die Arbeit verrichten soll. Es wird in unserem Fall die „Methode“ ausgewählt, die Dataframes sinnvoll darstellt. Entspricht der Klasse des ersten Arguments keine bestimmte Methode, dann wird die „default“-Methode benützt.

Einen ganz speziellen Stil von Darstellung liefert beispielsweise die Methode von `print`, die zur Klasse `htest` gehört und die deshalb das Resultat eines Wilcoxon- oder t-Tests anzeigt. Wenn man ein solches Testresultat nämlich abspeichert, erhält man ein Objekt der Klasse `htest`, das eigentlich eine lange Liste von Einzelinformationen ist.

```
> t.tt  <- t.test(t.kugel[t.hh],t.kugel[!t.hh])
> class(t.tt)
[1] "htest"
> names(t.tt)
[1] "statistic" "parameter" "p.value" "conf.int" "estimate"
[6] "null.value" "alternative" "method" "data.name"
```

Die „`htest`-Methode“ weiss, wie man all diese Informationen einigermassen verständlich anzeigt; das Resultat steht weiter vorne (2.7).

Zwei weitere grundlegende generische Funktionen sind `plot` und `summary`. Wir werden ihre Flexibilität in den kommenden Kapiteln kennen und schätzen lernen.

3.6 Ihr Workspace und die Packages

Die Objekte, die Sie erzeugen, sind in einem „Workspace“ abgelegt, der in R den Namen .Globalenv trägt. Die Objekte, die vom System zur Verfügung gestellt werden – vor allem die vielen Funktionen – sind in so genannten „Packages“ gespeichert, die von allen gelesen, aber natürlich nicht verändert werden können. Einige dieser Packages werden beim Aufstarten von R verfügbar gemacht – allen voran das `"package:base"`. Daneben gibt es viele Packages, die weitere Funktionen (und andere Objekte) enthalten. Sie werden nicht bei jedem Einstieg ins R automatisch „geladen“, da sie in vielen Sessions nicht benötigt werden. Das verkürzt die Antwortzeiten und verkleinert den Platzbedarf. Die Packages bilden die Grundlage der ständigen Erweiterbarkeit des Systems; unzählige Erweiterungs-Packages wurden von verschiedenen Autoren geschrieben. Wenn man ein bestimmtes Package braucht, zum Beispiel `MImput` (multivariate Analyse), dann verlangt man dies mit dem Befehl

```
> library("MImput")
```

Das Package `MImput` bleibt nur während der laufenden Session aktiv. In der nächsten Session muss man es wieder aktivieren. Der Befehl `library()` mit leerer Klammer zeigt die Namen aller Libraries, die auf dem benützten Computer installiert sind.

Die Liste der verfügbaren packages erhält man durch

```
> search()
[1] ".GlobalEnv"   "package:MImput"   ...   "package:base"
```

Wenn ein Objekt mittels seines Namens verlangt wird, dann sucht das System die aktivierten Packages der Reihe nach ab, bis es den Namen findet. Diese Art des Suchens nach einem Objekt hat zur Folge, dass Sie Systemfunktionen oder -objekte durch Ihre eigene Version „zudecken“. So können Sie für π plötzlich 3.111 erhalten, nachdem Sie
`pi <- 28/9` eingetippt haben. (Da hilft nur `remove(pi)`.) Sie sollten NIE eine Systemfunktion durch Ihre eigene Version ersetzen. Wenn Sie es besser können – und das ist immer wieder möglich – dann wählen Sie einen neuen (ähnlichen) Namen, beispielsweise `g.plot`.

Anmerkung. Das System tut sein Bestes, unabsichtliche Verwirrungen zu vermeiden. Die Funktionen des Systems haben ihren eigenen „name space“, in dem sie die benötigten Objekte suchen.

Wenn Sie wissen wollen, wo ein Objekt vom System gefunden wird, schreiben Sie

```
> find(pi)
[1] "package:base"
> pi  <- 0.78
> find(pi)
[1] ".GlobalEnv"
```

Ihr eigener Bereich. Ihre eigenen Objekte sind, wie erwähnt, im .GlobalEnv gespeichert. Wenn Sie wissen wollen, was sich da alles angesammelt hat, schreiben Sie

```
> objects()
 [1] "d.sport"      "f.maxi"   "last.warning"   "t.kugel"   "t.speer"
 [6] "t.x"          "t.y"
```

(`ls()` ist synonym mit `objects()`). Wahrscheinlich ist die Liste, die Sie erhalten, länger! Der Befehl

```
> objects(pattern="^t\\.")
```

liefert alle Objekte, die mit `t.` beginnen – also oben die letzten vier. Die merkwürdige Angabe im Argument `pattern` folgt der Syntax der „regular expressions“, die aus dem Unix stammt und inzwischen auch in anderen Software-Stücken verwendet wird. Wenn Ihnen die Liste zu lang wird, entfernen Sie Objekte mit `rm` oder

```
> remove(t.x)
```

Eine ganze Liste von Namen können Sie zum Rausschmiss freigeben durch

```
> remove(list=objects(pattern="^t\\."))
```

Hier zeigt sich der Vorteil einer konsequenten Namensgebung!

Den Inhalt von anderen packages liefert `objects("package:MImput")`.

Variablen eines data.frames direkt ansprechen. Es gibt die Möglichkeit, die Variablen eines data.frames direkt verfügbar zu machen, indem man

```
> attach(d.sport)
```

eintippt. Nun kann man beispielsweise `plot(hoch,kugel)` statt `plot(d.sport[,"hoch"],d.sport[,"kugel"])` verwenden. Sobald man aber Daten verändern will, entsteht ein Durcheinander. Deshalb ist mit dieser Möglichkeit Vorsicht am Platz. Vorläufig gehen Sie wohl besser auf Nummer Sicher und verzichten auf den Komfort der abgekürzten Schreibweise.

Absichern. Es kann nützlich sein, gewisse selbst erzeugte Objekte getrennt abzusichern, um sie vor versehentlichem Löschen zu bewahren, siehe `save`, `load`. Gründliches Aufräumen besteht aus dem Absichern der Objekte von bleibendem Wert, gefolgt von dem Befehl `remove(list=objects())`.

Selbst geschriebene *Funktionen* schreibt man aber besser von Anfang an in einer eigenen Datei, beispielsweise mit dem Namen `"r-functions.R"` und „aktiviert“ sie, indem man die Datei mit `source("r-functions.R")` ausführen lässt.

3.7 Die Funktionen `options` und `.First`

Mit der Funktion `options` kann man einige Feinheiten festlegen, die das Verhalten des Systems beeinflussen. Ein oft nützlicher Befehl ist

```
> options(digits=3)
```

Er bestimmt, dass die Zahlen mit 3 signifikanten Stellen auf dem Bildschirm angegeben werden sollen. Er setzt also eine Grösse, die die Funktion `print` benützt, um das Resultat wie gewünscht anzuzeigen.

`options` ist also eine recht spezielle Funktion. Im Moment bringt sie kein Resultat. Sie speichert lediglich eine Information ab, die sich auf das Verhalten des Systems in Zukunft auswirkt. Das gewählte Verhalten verschwindet mit dem Abschluss der Session (`q()`), analog zu den Packages, die verschwinden (3.6).

Will man das Eintippen von `options`, `library` oder Ähnlichem am Anfang jeder Session vermeiden, so hilft dabei die Funktion `.First`. Wenn man eine solche Funktion erzeugt, beispielsweise

```
> .First  <- function() {
+  options(digits=3)
+  library("MImput") }
```

(und auch über die Session hinaus speichert, indem man dies nach dem Eintippen von `q()` verlangt), dann werden die beiden Befehle am Anfang jeder Session, die man vom gleichen Ordner (directory) aus aufruft, ausgeführt. In der Funktion `.First` können auch beliebige weitere Befehle eingefügt werden, beispielsweise `print("Salue! Wie geht's Dir heute?")`. Die Funktion besitzt keine Argumente, da sie ja normalerweise direkt vom System aufgerufen wird und deshalb nicht auf veränderliche Werte von Argumenten reagieren kann.

4 Grafische Darstellungen

4.1 Grafische Hauptfunktionen

Wir haben schon einige grafische Funktionen kennen gelernt: `hist, plot, pairs`. Diese Funktionen erzeugen eine ganze grafische Darstellung. Sie kennen eine ansehnliche Anzahl Argumente, die es erlauben, Beschriftungen anzupassen und Varianten der Darstellung zu verlangen.

```
> plot(x, y, type="b", pch=letters[1:length(x)],
  main="Es geht aufwärts", xlab="Zeit", ylab="Kosten")
```

passt die Achsenbeschriftung an, setzt einen Titel (`main`), beschriftet die Punkte mit Kleinbuchstaben (`pch`) und verbindet sie zusätzlich mit geraden Linien (`type="b"`). Tabelle 4.3 zeigt die wichtigen Argumente von `plot`, die auch in vielen anderen grafischen Funktionen vorkommen. Weitere Argumente sind in Tabelle 4.5 aufgeführt. Eine vollständige Beschreibung liefert `?plot`, zusammen mit `?par`. Die Funktion `par` dient dazu, allgemeine „grafische Parameter“ festzulegen, die für weitere Grafiken anwendbar sind, bis sie wieder mit `par` geändert werden – analog zur Funktion `options` (vergleiche Kapitel 3.7). Sie wird in Kapitel 4.4 besprochen.

Was die Funktion `plot` überhaupt darstellt, hängt von den ersten Argumenten ab. Die einfachste Form, die bisher jeweils benützt wurde, erhält als erste Argumente die Koordinaten `x` und `y` von Punkten, die dargestellt werden sollen. Wir werden aber noch etliche andere Formen von Argumenten von `plot` und andere Resultate kennen lernen. `plot` ist eine „generische“ Funktion, siehe Kapitel 3.5.

Weitere nützliche grafische Hauptfunktionen (high-level graphics) sind: `barplot, pairs, matplot` und `coplot` Es gibt noch viele nützliche grafische Funktionen in S. Wir wollen sie nicht hier aufzählen, sondern sie im Zusammenhang mit den entsprechenden statistischen Problemstellungen in den späteren Kapiteln einführen.

Weitere Informationen findet man auf den HTML-help-Seiten, die mit `help.start()` aufgerufen werden. Im erscheinenden Browser wählen Sie „Search Engine & Keywords“. Unter „Keywords by topic“ finden Sie ”Graphics” und dort weitere Informationen zu grafischen Darstellungen.

4.2 Einteilung der Bildfläche

Es ist nützlich, die übliche Einteilung der Bildfläche mit ein paar Begriffen zu charakterisieren, die im Folgenden benützt werden:

- In einem inneren Rechteck, dem **Plotbereich**, werden durch Punkte, Linien und andere Elemente die Daten dargestellt. Die Punkte im Plotbereich werden mit den **Benützer-**

Argument	Bedeutung
	Inhalt
`type`	In welcher Form sollen die Punkte gezeichet werden? `="p"`: Punkte zeichnen (default). `="l"`: Linien zwischen den Punkten zeichnen. `="b"`: Punkte und Linien zeichnen. `="n"`: Punkte nicht zeichnen (vgl. 4.3)
`log`	logarithmische Darstellung der Achsen: `="x"`: x-Achse wird logarithmisch dargestellt. Ebenso: `="y"` `="xy"`: beide Achsen werden logarithmisch dargestellt. (Andere Transformationen sind nicht so bequem zu haben.)
`xlim`	Vektor `c(a,b)`. Wertebereich der x-Koordinate (von `a` bis `b`). Ebenso: `ylim`. Punkte und Linien, die ausserhalb dieser Bereiche liegen, werden ohne Warnung ignoriert.
`pch`	plotting character. Kann ein Buchstabe sein oder eine Zahl (1 bis 25; Bedeutung: `plot(1:25,rep(1,25),pch=1:25,cex=2)`). Von einem string wird der erste Buchstabe verwendet. In R kann ein Vektor der Länge `length(x)` angegeben werden, der dann für jeden Punkt ein eigenes Symbol festlegt. Ebenso:
`col`	Farbe für die Darstellung der Punkte.
`lty`	Linienmuster (line type). 1 heisst ausgezogen, 2 gestrichelt, 3 gepunktet, (siehe `?par`)
	Beschriftung
`main`	Titel. Erscheint am oberen Rand in vergrösserter Schrift.
`sub`	Untertitel erscheint am unteren Rand (flexibler: `mtext`, siehe Tabelle 4.4)
`xlab`	Beschriftung der x-Achse. Ebenso: `ylab`.
`axes`	`=FALSE` unterdrückt das Zeichnen der Achsen (vgl. unten).

Tabelle 4.3: Argumente von grafischen high-level Funktionen

koordinaten (user coordinates) angesprochen. Andere Koordinatensysteme folgen in `library(grid)`.

- Um diese Region wird oft ein **Rahmen** (bounding box) gezeichnet.
- Die **Ränder** (margins) dienen der Beschriftung der **Achsen** durch **Marken** (tick marks) und entsprechende **Marken-Beschriftungen** (labels). Die Achsen tragen zudem **Achsenbezeichnungen** (axis labels).
- Zur Bestimmung eines gewünschten Randes sind die **Seiten** durchnummeriert: 1=unten, 2=links, 3=oben, 4=rechts. Die Koordinaten in den Rändern sind primär die **Linien**, die vom Plotbereich nach aussen gezählt werden. Falls nötig, werden für die Position entlang der Linie die Koordinaten des Plotbereiches übernommen.

- Der Plotbereich und die Ränder zusammen füllen den **Figurenbereich** (figure region).

In Abschnitt 4.5 wird angegeben, wie mehrere Figuren in einem Bild gezeichnet werden können.

4.3 Ergänzende Funktionen

Neben diesen grafischen Hauptfunktionen gibt es andere, die zu einer Darstellung etwas hinzufügen. Beispielsweise zeichnet `abline(a,b)` eine Gerade mit Achsenabschnitt `a` und Steigung `b` in die gerade gezeichnete Darstellung ein. Tabelle 4.4 stellt einige solche ergänzende Funktionen zusammen. Beachten Sie, dass die meisten Argumente Vektoren sein können – und es typischerweise auch sind. Für genauere Erklärungen verweisen wir Sie auf `help`.

Will man die Darstellung der Punkte in einem Streudiagramm mit Hilfe von `points` oder `text` selber kontrollieren, dann muss man die Funktion `plot` auffordern, dies nicht auf ihre Art zu tun, indem man `type="n"` schreibt,

```
> plot(x,y, type="n")
> text(x,y, paste("X",1:length(x),sep=""))
```

Ebenso unterdrückt man die übliche Achsendarstellung, um eine eigene anzuhängen,

```
> plot(x,y, yaxt="n")
> axis(2, at=c(exp(1),5), labels=c("e","5"))
```

Entlang der y-Achse werden nur die Zahlen e und 5 markiert. Beide Achsen und der Rahmen (box) werden unterdrückt durch `plot(x,y, axes=FALSE)`.

4.4 Die Funktion `par`

Viele Darstellungselemente – beispielsweise die Schriftgrösse – haben für alle möglichen grafischen Darstellungen eine gleiche oder ähnliche Bedeutung. Man kann sie daher nach dem eigenen Geschmack einrichten und braucht sie dann nicht bei jeder einzelnen Darstellung wieder anzugeben. Diese „grafischen Optionen“ werden mit der Funktion `par` festgelegt und abgefragt. Die wichtigsten Argumente von `par` – die Namen der festzulegenden grafischen Elemente – zeigt Tabelle 4.5, siehe auch `?par`. Schreibt man also

```
> par(mar=c(3,3,1,1), mgp=c(2,0.8,0))
```

dann werden bei den folgenden grafischen Darstellungen die Ränder, die für Beschriftungen rund um den Plotbereich freigelassen werden, auf je drei Linien unten und links und je eine Linie oben und rechts festgelegt. Damit man mit so schmalen Rändern auskommt, wird mit dem Argument `mgp` festgelegt, dass der Achsenname auf Linie 2, die Zahlen-Beschriftung auf „Linie 0.8“ und die Achse selbst auf Linie 0 zu liegen kommt.

Will man die gültigen Werte abfragen, dann wird der Name des Argumentes in Anführungszeichen gesetzt:

```
> par("usr")
```

zeigt die gültigen Grenzen des Plotbereiches an.

Zusammen ermöglichen die drei Ebenen – Haupt- und Ergänzungsfunktionen und `par` – fast alles, was man sich an Möglichkeiten für *statische* Grafiken denken kann. Am Schluss des Kapitels werden wir das an Hand von Beispielen zeigen. Für *dynamische* Grafiken ist nicht gerade viel vorhanden. Wir kommen darauf in der Multivariaten Statistik zurück.

Die vorgestellte Konzeption der Grafik in S trägt die Spuren einer langen Geschichte. In der Tat geht sie auf die Uranfänge von S zurück – das Bedürfnis nach flexibler Gestaltung von grafischen Darstellungen gab den Anstoss zur Programmierung eines entsprechenden Programmpakets, das dann zu einer „Statistik-Sprache“ ausgebaut wurde. Die Sprache wurde später in drei Schritten grundlegend klarer konzipiert und erweitert, während die Konzeption der Grafik noch allzu sehr an die Ursprünge erinnert.

Funktion	Bedeutung
	Zeichnen
`points(x,y,pch=1)`	Punkte zeichnen mit Symbol `pch`, Koordinaten `[x,y]`
`text(x,y,text)`	Punkte mit `text` anschreiben, Koordinaten `[x,y]`
`lines(x,y,lty=1)`	Linien ziehen (Strichmuster `lty`) zwischen Punkten mit Koordinaten `[x,y]`
`polygon(x,y,col)`	zeichnet ebenfalls Linien, zusätzlich eine Verbindung vom letzten zum ersten Punkt, und füllt die umschlossene Fläche mit Farbe `col` oder allenfalls mit einer Schraffur, siehe `?polygon`.
`abline(a,b)`	Eine Gerade zeichnen mit Achsenabschnitt `a` und Steigung `b`.
`abline(h=y,v=x)`	horizontale (auf Höhe `y`) oder vertikale Geraden zeichnen (bei `x`)
`segments(x1,y1,x2,y2,lty=1)`	Linien ziehen von den Punkten mit Koordinaten `[x1,y1]` zu `[x2,y2]`
`matpoints(x,y)`	zu jedem x-Wert mehrere y-Werte zeichnen, vgl. `matplot`. Ebenso: `matlines`
	Beschriftungen
`axis(side,at,labels)`	Achse zeichnen. Marken bei `at` mit `labels` beschriften. `side`: Seite der Darstellung: 1=unten, 2=links, 3=oben, 4=rechts
`mtext(text,side,line=0)`	Text im Rand `side` schreiben auf Linie `line`.
`title("Titel")`	Titel `"Titel"` oben (in grösseren Buchstaben) schreiben.
`box(bty="o")`	Plotbereich umrahmen (allenfalls nicht alle Seiten, siehe Tabelle 4.5)
`legend(x,y,text,lty,pch)`	Eine Legende an der Stelle `(x,y)` schreiben.

Tabelle 4.4: Ergänzende grafische Funktionen

Argument	Bedeutung
`mar`	Vektor `c(u,l,o,r)`. Breite der Ränder (unten, links, oben, rechts) in Anzahl „Linien“
	Achsen
`mgp`	Linien für den Achsen-Titel, die Beschriftung der Marken und die Marken, siehe Beispiel. Default `c(3,1,0)`.
`lab`	Vektor `c(x, y, len)`. Gibt die ungefähre Zahl der Marken in x- und y-Richtung und die Anzahl Stellen der Beschriftung an.
`xaxp`	Vektor `c(x1, x2, n)`. Kleinster und grösster markierter Wert und Anzahl Intervalle dazwischen für die x-Achse. Ebenso `yaxp`
`xaxt`	`="n"` unterdrückt das Zeichnen der x-Achse. Ebenso `yaxt`
`xaxs`	Wie lange soll die x-Achse insgesamt werden? Es sei $[a, b]$=`range(x)` oder der Wert von `xlim`. `"r"`: $[a, b]$, zu beiden Seiten um 4% erweitert. `"i"`: $[a, b]$ Ebenso `yaxs`
`usr`	Vektor `c(x1,x2,y1,y2)`. Plotbereich: Bereich der Koordinaten in x- und y-Richtung
`bty`	Box type: Welcher Rahmen soll um die Darstellung gezeichnet werden? Die Werte `"o", "l", "7", "c", "u", or "]"` zeichnen jeweils eine Linie auf 4, 2 oder 3 Seiten, die dem Zeichen entsprechen. `"n"` unterdrückt die Box
	Zeichen und Linien
`cex`	Character expansion: Wie gross sollen Zeichen werden, im Verhältnis zum default-Wert (der vom device u.a. abhängt)? `cex.axis, cex.lab, cex.main, cex.sub` setzen dies spezifisch für die entsprechenden Elemente.
`font`	Font-Auswahl, siehe `?par`.
`srt`	string rotation in Grad. Ebenso `crt`: character rotation
`lwd`	Line width, Strichbreite für Linien.
`col`	Color: Farbe der gezeichneten Striche und Symbole.
	Gesamte Darstellung
`ask`	`=TRUE` führt dazu, dass jeweils gefragt wird, bevor der Bildschirm für die nächste Figur gelöscht wird.
`new`	`=TRUE` verhindert, dass die nächste Plot-Funktion eine neue Figur beginnt. („Tu so, als ob die Figur noch nicht da wäre.“)
`pin`	Breite und Höhe des aktiven Plotbereiches.
`pty`	`="s"` erzeugt einen quadratischen Plotbereich.
`xpd`	`=TRUE`: Punkte und Linien werden auch im Rand der Figur gezeichnet, nicht nur im Plotbereich.

Tabelle 4.5: Argumente der Funktion `par`

4.5 Mehrere Figuren auf dem gleichen Bild

Oft möchte man mehrere grafische Darstellungen auf einem Blatt oder in einem Bildschirm-Fenster sehen. Es gibt zwei Methoden, das Blatt in mehrere „Frames“ aufzuteilen.

Die alte Methode benützt die Funktion `par`.

```
> par(mfrow=c(2,3))
```

teilt das Blatt in 2 Zeilen und 3 Spalten auf. Die erste Grafik wird links oben platziert, die nächste rechts daneben usw. Tabelle 4.6 zählt Argumente von `par` auf, die mit dieser Möglichkeit verbunden sind. Mit diesem Mechanismus arbeitet auch die Funktion `mult.fig`, die gleichzeitig die Ränder (`mar`) und weitere ähnliche Grössen sinnvoll verändert und damit das Arbeiten mit mehreren Figuren erleichtert.

Argument	Bedeutung
`mfrow`	Vektor `c(r,c)`. Aufteilung in eine „Figuren-Matrix“ mit `r` Zeilen und `c` Spalten; wird zeilenweise aufgefüllt.
`mfcol`	ditto, spaltenweise aufgefüllt.
`mfg`	Vektor `c(i, j)`. Welche Figur in der Matrix soll gezeichnet werden resp. wird gerade gezeichnet?
`oma`	Es werden äussere Ränder um die Figuren-Matrix herum eingeführt mit der Breite der Anzahl gewählter „Linien“. In diese äusseren Ränder schreibt man mit `mtext(..., outer=TRUE)`.

Tabelle 4.6: Argumente der Funktion `par` zur Aufteilung des Bildes in mehrere Figuren

4.6 Beschriftungen

Beschriftungen, also Titel, Achsennamen, Legenden und Ähnliches müssen als `character`-Grössen angegeben werden. Um sie an die gegebene Situation anzupassen, ist die Funktion `paste` häufig sehr nützlich, siehe 2.3.

```
> text(x,y,paste("[", round(x,3), ",", round(y,3), "]", sep=""))
```

beschriftet den Punkt $[x, y]$ mit seinen (auf 3 Dezimalstellen gerundeten) Koordinaten. Um Zahlen in lesbare `character`-Grössen zu verwandeln, ist auch die Funktion `format` hilfreich.

In R können in Beschriftungen auch Formeln verwendet werden. Siehe `?plotmath`.

4.7 Grafiken für Tex(t)-Dokumente

Normalerweise werden Grafiken auf dem Bildschirm gezeigt. Wenn man sie in Berichte einbinden will, erzeugt man stattdessen ein Postscript- oder PDF-File und importiert es (allenfalls nach geeigneter Umwandlung) anschliessend in die Text- (Word- oder LaTeX-) Datei. Nach dem Aufruf

```
> postscript("bild.eps", width=7, height=5, horizontal=TRUE)
```

werden die Grafiken nicht mehr auf dem Bildschirm angezeigt, sondern in die Datei `bild.eps` gezeichnet. Um die Datei abzuschliessen, schreibt man

```
> dev.off()
```

Für ein PDF verwendet man entsprechend die Funktion `pdf()`, für eine JPG-Datei `jpeg()` etc.

Man kann mehrere grafische Ausgabe-Orte gleichzeitig offen halten – vor allem unbeabsichtigt – indem man weitere Aufrufe von Grafik-Treiber-Funktionen eingibt.

```
> dev.list()
```

sagt, welche Ausgabe-Orte (devices) im Moment offen sind. Der letzte in der Liste wird benützt, bis er geschlossen wird – ausser man ändere dies mit `dev.set`. Wenn Sie mehrere Fenster auf dem Bildschirm erzeugen wollen, rufen Sie `windows()` auf (oder `X11()`, wenn Sie unter Xwindows arbeiten).

4.8 Beispiele für selbst gestaltete grafische Funktionen

Die folgenden Befehle erzeugen eine Figur, die Sie im Skript zur Regression wieder finden werden:

```
postscript("p_spreinfach.eps")
l.d <- data.frame(dist = c(49, 46, 44, 74, 36, 39, 40, 39, 117,
   110, 106, 54, 66), ersch = c(5.99, 5.39, 4.88, 3.14, 9.83, 8.03,
   4.79, 4.55, 0.77, 0.92, 0.77, 3.29, 2.99))
plot(l.d$dist,l.d$ersch, log="xy", xlab="Distanz", ylab="Erschütterung")
l.r <- lm(log10(ersch)~log10(dist),data=l.d) # Regression
abline(l.r) # abline benützt die geschätzten Koeffizienten
l.fit <- fitted(l.r) # fitted values (Modellwerte)
lines(rep(l.d$dist[1],2), c(l.d$ersch[1],10^(l.fit[1])),
      lty=3) # zeichne eine
text(52, 6.8, "[x_i,y_i]")
text(51.5, 5.4, expression(r[i]))
dev.off()
```

4.9 Trellis, lattice, ggplot2

Trellis ist eine Sammlung von Grafik-Funktionen mit hoher Aussagekraft und sehr durchdachter Gestaltung, die von W. Cleveland geschaffen wurden (siehe William S. Cleveland, The Elements of Graphing Data, Hobart Press (1994); ETH-Bib.: 761 026). Leider sind die Funktionen schlecht veränderbar. In R findet man eine Sammlung im Package `lattice`.

Ein weiteres neues System mit eigener „Grafik-Grammatik“ findet man im Package `ggplot2`.