Reihenherausgeber:
Prof. Dr. Holger Dette · Prof. Dr. Wolfgang Härdle

Statistik und ihre Anwendungen

Weitere Bände dieser Reihe finden Sie unter http://www.springer.com/series/5100

Uwe Ligges

Programmieren mit R

Dritte, überarbeitete und erweiterte Auflage

Springer

Professor Dr. Uwe Ligges
Technische Universität Dortmund
Fakultät Statistik
Vogelpothsweg 87
44221 Dortmund
ligges@statistik.tu-dortmund.de

ISBN 978-3-540-79997-9 e-ISBN 978-3-540-79998-6

DOI 10.1007/978-3-540-79998-6

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Mathematics Subject Classification (2000): 68-01, 68N15

Herstellung: le-tex publishing services oHG, Leipzig
Einbandgestaltung: WMXDesign GmbH, Heidelberg

Gedruckt auf säurefreiem Papier

9 8 7 6 5 4 3 2 1

springer.de

Vorwort zur dritten Auflage

R ist eine objektorientierte und interpretierte Sprache und Programmierumgebung für Datenanalyse und Grafik — unter der GPL[1] und für alle gängigen Betriebssysteme.

R basiert auf den Definitionen von S, das an den Bell Labs von John Chambers und anderen über drei Jahrzehnte hinweg speziell für Probleme der Datenanalyse entwickelt wurde. Die „Association for Computing Machinery" (ACM) vergab 1998 ihren „Software System Award" für S:

> *[...] the* S *system, which has forever altered the way people analyze, visualize, and manipulate data [...]* S *is an elegant, widely accepted, and enduring software system, with conceptual integrity, thanks to the insight, taste, and effort of John Chambers.*

R bietet vieles, was von einem modernen Werkzeug zur Datenanalyse erwartet wird. Der einfache Umgang mit Daten aus verschiedenen Quellen und Zugang zu Datenbanken ist eine Grundvoraussetzung. Für statistische Methodik aller Art gibt es Funktionen, angefangen von deskriptiven Verfahren über das lineare Modell bis hin zu modernen Verfahren wie etwa *Support Vector Machines*.

Selbstentwickelte Verfahren können einfach implementiert werden, wobei auch Schnittstellen zu anderen Sprachen (z.B. C, C++, Fortran) eine hohe Flexibilität bieten. R lässt den Anwender sehr schnell und einfach zum Programmierer werden, weil Ideen meist direkt in Programmcode umgesetzt werden können. Nicht zuletzt sollen Grafiken einfach zu erzeugen sein. Hier ist nicht nur die direkte Visualisierung der Daten während der Analyse möglich, sondern auch das Erstellen hochwertiger Grafiken für Publikationen und Präsentationen.

[1] *GNU General Public License*, `http://www.gnu.org/copyleft/gpl.html`

Ziel dieses Buches ist es, die Leser nicht nur ausführlich in die Grundlagen der Sprache R einzuführen, sondern auch ein Verständnis der Struktur der Sprache zu vermitteln und in deren Tiefen einzudringen. So können leicht eigene Methoden umgesetzt, Objektklassen definiert und, z.B. für größere Projekte, ganze Pakete aus Funktionen und zugehöriger Dokumentation zusammengestellt werden. Das Buch ist damit für Anfänger und Fortgeschrittene gleichermaßen geeignet.

Noch ein Hinweis eher rechtlicher Natur. Der Name der folgenden im Buch erwähnten Software und Hardware ist jeweils durch eingetragenes Warenzeichen der jeweiligen Eigentümer geschützt: Access, CART, DCOM, Debian, Excel, Linux, Macintosh, MacOS X, Minitab, MySQL, New S, ODBC, PostgreSQL, S, S-PLUS, SAS, SPSS, TIBCO, Trellis, UNIX, Windows, Word. Es wird hiermit nicht ausgeschlossen, dass es sich auch bei weiteren in diesem Buch verwendeten Namen um eingetragene Warenzeichen handelt.

In die zweite und dritte Auflage des Buchs sind jeweils Fehlerkorrekturen, Anpassungen an neue R Versionen und Ergänzungsvorschläge von Lesern eingeflossen.

Falls bisher unentdeckte Fehler in diesem Buch auffallen sollten oder Verbesserungsvorschläge vorhanden sind, so bitte ich darum, mir diese unter folgender Adresse mitzuteilen: `ligges@statistik.tu-dortmund.de`.

Danksagung

Zunächst möchte ich Claus Weihs für all seine Unterstützung danken. Axel Scheffner brachte mir als erster etwas über S bei, und Detlef Steuer erzählte mir, dass es R gibt und darum alles gut wird.

Für viel Hilfe und fruchtbare Diskussionen danke ich allen Mitgliedern des *R Development Core Teams*, insbesondere Kurt Hornik, Friedrich Leisch, Martin Mächler und Brian Ripley. Ohne das R Development Core Team und die Entwickler von S gäbe es weder R noch dieses Buch. Viele weitere Freiwillige haben zu R mit ihrer Arbeit auf verschiedenste Art und Weise beigesteuert.

Jürgen Groß, Guido Knapp und Sibylle Sturtz haben mich darin bestärkt, dieses Buch zu schreiben. Für sehr gute kollegiale Zusammenarbeit danke ich außerdem Daniel Enache, Martina Erdbrügge, Ursula Garczarek, Karsten Lübke, Nils Raabe, Olaf Schoffer, Gero Szepannek, Winfried Theis und Lars Tschiersch.

Ich möchte allen Lesern der ersten Auflage danken, die sich die Mühe gemacht haben, mich auf Fehler und Ungenauigkeiten hinzuweisen. Zu sehr vielen Verbesserungen hat Motohiro Ishida, der Übersetzer der japanischen Ausgabe des Buches, beigetragen.

Besonderer Dank, nicht nur für das Korrekturlesen der ersten Auflage dieses Werkes, gilt meiner lieben, hilfsbereiten Kollegin Anja Busse, die bei jeglicher Zusammenarbeit so viel Frohsinn verbreitet hat.

Die meisten Korrektur- und Verbesserungsvorschläge für die zweite Auflage brachte Sandra Leissen ein. Danke, dass Du stets für mich da bist!

Meine Eltern haben mich zeitlebens in allen Situationen unterstützt. Danke!

Dortmund, *Uwe Ligges*
Juli 2008

Inhaltsverzeichnis

1

Einleitung

„R: A Language for Data Analysis and Graphics“, so lautet der Titel des Artikels von Ihaka und Gentleman (1996). Bei R handelt es sich um eine *Open Source* Software[1] und eine hochflexible Programmiersprache und -umgebung für (statistische) Datenanalyse und Grafik, die auch Mittel zum Technologie- und Methodentransfer, etwa mit Hilfe von Zusatzpaketen, bereitstellt. Es gibt Datenzugriffsmechanismen für Textdateien, Binärdateien, R Workspaces, Datensätze anderer Statistiksoftware und Datenbanken. Abbildung 1.1 zeigt eine laufende R Sitzung unter Windows

Ausprobieren

Es empfiehlt sich, Beispiele direkt am Rechner in R auszuprobieren und nach eigenem Ermessen zu verändern, denn nur durch eigenes Ausprobieren, Arbeiten und „Spielen“ mit einer Programmiersprache kann man sie schnell und sicher erlernen.

CRAN

Bei *CRAN* (*C*omprehensive *R* *A*rchive *N*etwork) handelt es sich um ein Netz von Servern, das weltweit den Quellcode und die Binärdateien für diverse Betriebssysteme bereitstellt. Der zentrale Server ist unter `http://CRAN.R-project.org` zu erreichen. Dort gibt es u.a. auch weit mehr als 800 Zusatzpakete zu diversen (statistischen) Verfahren sowie Publikationen, Dokumentation, FAQs, Zugriff auf Mailinglisten, Links zu anderer relevanter Software und Projekten. Die R Homepage findet man unter `http://www.R-project.org`.

Mehr Details zum Herunterladen, zur Installation und zum Aufruf von R gibt es in Anhang A.

[1] Open Source Software: Software, deren Quellcode frei erhältlich ist.

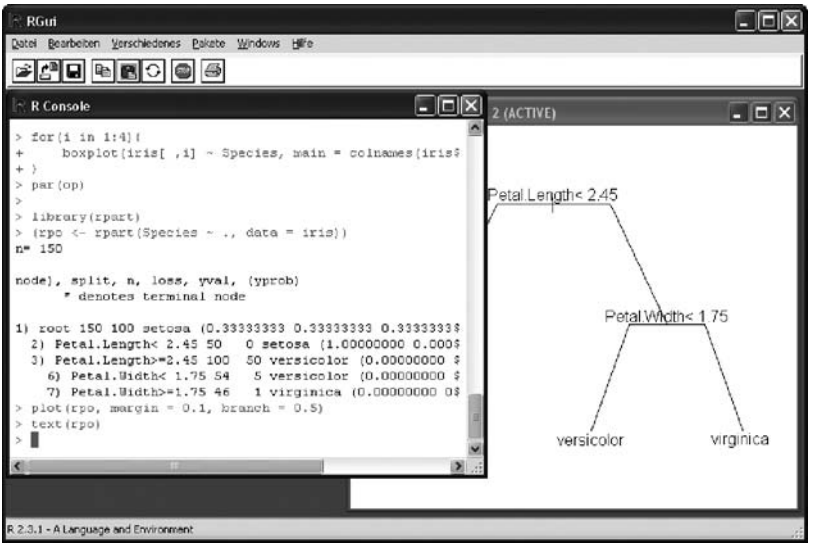

Abb. 1.1. Laufende R Sitzung unter Windows

Notation

Die Notation in diesem Buch beschränkt sich im Wesentlichen auf folgende Unterscheidungen vom Fließtext. Neu eingeführte Begriffe und Eigennamen werden *kursiv* gesetzt und die Namen von zusätzlichen Paketen (*packages*) für R sind **fett** gedruckt. Jeglicher Programmcode in R und Namen von Funktionen sowie Befehle in der Kommandozeile des Betriebssystems (durch einleitendes „$“ gekennzeichnet) erkennt man an `dieser Schrift gleicher Zeichenbreite`.

Längerer Programmcode wird eingerückt und vom Text abgesetzt gedruckt. Sollte auch eine Ausgabe von R abgedruckt sein, so ist der vorangehende, diese Ausgabe erzeugende Code mit einem anführenden Zeichen „>“ versehen, das auch in der R Konsole die Eingabeaufforderung anzeigt. Sollte ein Ausdruck am Ende einer Zeile syntaktisch nicht vollständig sein, wird die nächste Zeile stattdessen mit „+“ markiert, z.B.:

```
> cat("Hallo",          # R Code
+     "Welt\n")         # Fortsetzung des letzten Ausdrucks
Hallo Welt              # Ausgabe
```

Einige Ausgaben von R sind nachträglich stillschweigend editiert worden, um Platz zu sparen. Daher ist auch die angezeigte Genauigkeit häufig von standardmäßig 7 auf 4 Stellen und die Breite der Ausgabe auf 67 Zeichen reduziert worden.

Technisches

Sämtliche Berechnungen, Ausgaben und Grafiken in diesem Buch wurden mit R-2.3.1 unter dem Betriebssystem Windows XP erzeugt. Für die Geschwindigkeitsangaben und deren Vergleich ist festzuhalten, dass diese auf einem Rechner mit AMD Athlon Prozessor mit 1.67 GHz und 512 MB DDR-RAM entstanden sind.

WWW

Neuigkeiten und Änderungen von R, die erst nach dem Erscheinen dieses Buches bekannt werden, sowie die in diesem Buch benutzten Datensätze und einige Skripts, sind im WWW zu finden unter:
`http://www.statistik.uni-dortmund.de/~ligges/PmitR/`.

1.1 Die Geschichte

Am Anfang gab es S, eine seit 1976 in den *Bell Laboratories* bei *AT&T* (heute bei *Lucent Technologies*) für Statistik, stochastische Simulation und Grafik entwickelte Programmiersprache, zu deren zweiter Version 1984 von Becker und Chambers das erste (*braune*) Buch veröffentlicht wurde. *The New S* wurde 1988 von Becker et al. (das *blaue* Buch) eingeführt und enthält mehr oder weniger die heute bekannte S Funktionalität. Objektorientierte Ansätze von S (auch unter S3 bekannt) wurden 1992 in dem *weißen* Buch von Chambers und Hastie beschrieben, die auch Möglichkeiten hinzufügten, statistische Modelle in einer Art Formelnotation einfach zu spezifizieren. Die neueste Version 4 von S ist in Chambers (1998, das *grüne* Buch) beschrieben. Für weitere Details zur Geschichte von S siehe Becker (1994). Die Bücher, welche S definieren, werden unter S Anwendern und Programmierern häufig mit der Farbe ihres Einbands bezeichnet.

Eine kommerzielle Implementation von S gibt es seit 1988 mit S-PLUS (Insightful Corporation, 2006), das zunächst von *Statistical Sciences Inc.*, dann von *Mathsoft* und heute von der *Insightful Corp.*[2] vertrieben wird. Die aktuelle Version, S-PLUS 7, ist für Windows und einige UNIX Systeme (darunter Linux) erhältlich und enthält sehr große Teile der S Definition nach Chambers (1998).

R (R Development Core Team, 2006a) ist eine weitere, freie Implementation, die unter der GNU GENERAL PUBLIC LICENSE[3] zunächst für Lehrzwecke entwickelt wurde. Sie basiert auf den Ideen (nicht aber dem Code)

[2] `http://www.insightful.com`
[3] `http://www.gnu.org/copyleft/gpl.html`

von S. Die folgende Zeittafel (s. auch Hornik und Leisch, 2002, dieser Artikel ist auch für mehr Details zur Geschichte von R lesenswert) zeigt die rasante Entwicklung von R:

1992: Ross Ihaka and Robert Gentleman (1996) starten ihr Projekt.
1993: Die erste Binärversion erscheint auf Statlib.
1995: R erscheint unter der GPL.
1997: Das *R Development Core Team* (kurz: R Core Team) vereinigt sich. Heute sind daran 17 Personen aus Forschung und Wirtschaft beteiligt, die rund um den Globus verteilt sind, darunter auch S „Erfinder“ John Chambers.
1998: Das Comprehensive R Archive Network (CRAN, siehe S. 1) wird gegründet.
1999: Die erste DSC (Distributed Statistical Computing) Konferenz[4] findet in Wien statt (später im Rhythmus von 2 Jahren) – damit auch das erste gemeinsame Treffen aller Mitglieder des R Core Teams.
2000: Die erste zur Sprache S in Version 3 vollständig kompatible Version, R-1.0.0, erscheint passend zum Schaltjahr am 29.02.2000.
2001: Die Zeitschrift R News (s. S. 18) wird erstmalig herausgegeben.
2002: Die R Foundation[5] wird gegründet.
2004: Die erste R Anwenderkonferenz, *useR!*[6], findet in Wien statt.
Am 04.10.2004 erscheint R-2.0.0. Der Versionssprung zeigt, dass sich die Sprache R in den Jahren seit erscheinen der Version 1.0.0 in besonderem Umfang weiterentwickelt hat, u.a. mit neuen Errungenschaften wie S4 Methoden, Namespaces, verbessertem Paketmanagement und vielen neuen Funktionen. S4 Methoden nach Chambers (1998) sind von John Chambers in dem R Paket **methods** implementiert worden, das in aktuellen Versionen standardmäßig mitinstalliert und -geladen wird.
2006: Die aktuelle Version ist R-2.3.1 (Stand: Juli 2006).

Wie es zum Namen R kam ist nicht mehr genau bekannt (klar ist, dass R nahe an S liegt). Die beiden plausibelsten „Legenden“ besagen, dass er entweder entstand, da Ross Ihakas und Robert Gentlemans Vornamen jeweils mit R beginnen, oder weil man eine „*r*educed version of S“ wollte.

[4] `http://www.ci.tuwien.ac.at/Conferences/DSC.html`
[5] Ziel des Vereins „The R Foundation for Statistical Computing“ ist die Förderung des „R Project for Statistical Computing“. Details findet man unter der URL `http://www.R-project.org/foundation/`
[6] `http://www.ci.tuwien.ac.at/Conferences/useR-2004/`

1.2 Warum R?

Wichtige Anforderungen an eine Sprache für Datenanalyse und Statistik sind sicherlich die Möglichkeiten zur Automatisierung sich wiederholender Abläufe und Analysen, zur Anpassung vorhandener Verfahren an neue Problemstellungen, zur Implementation neuer Verfahren und nicht zuletzt das Programmieren von Simulationen. Solche Anforderungen können nur von einer Programmiersprache erfüllt werden, die sehr flexibel ist, Schnittstellen zu anderen Sprachen bildet und eine möglichst große Anzahl an bereits implementierten Verfahren mitbringt.

Chambers (1998) schreibt, dass die Ziele von S u.a. darin bestehen, *einfach* interaktiv mit Daten rechnen zu können, *einfach* den Benutzer (häufig Statistiker) zum Programmierer werden zu lassen, *einfach* Grafiken für explorative Datenanalyse und Präsentationen zu erstellen und *einfach* bereits entwickelte Funktionen wieder verwenden zu können. Selbiges gilt auch für R als eine Implementation der Sprache S (s. Abschn. 1.1).

Vorteile von R

Zu den herausragenden Vorteilen von R gehört, dass es sich um Open Source Software handelt und unter der *GPL* lizensiert ist. Es ist möglich, allen Quellcode einzusehen, der frei zum Herunterladen bereit liegt, so dass es sich bei R an keiner Stelle um eine „Black Box“ handelt, denn es kann *überall* nachvollzogen werden, wie und was gerechnet wird. Außerdem ist R auf einer Vielzahl von Betriebssystemen und Plattformen lauffähig[7], darunter Macintosh, UNIX (AIX, FreeBSD, Linux, Solaris, ...) und Windows.

Wegen dieser Offenheit ist R „am Puls der Forschung“. Als Beispiel für diese Aktualität von R sei das *BioConductor* Projekt[8] (Gentleman et al., 2004) genannt, das sich mit Datenanalyse im Bereich der Genetik beschäftigt.

Neue Methoden werden häufig von den Methodenentwicklern selbst in R programmiert und als Paket für die Allgemeinheit zur Verfügung gestellt. Sollte eine Methode noch nicht implementiert sein, so wird sich sicherlich ein R Benutzer finden, der diese Methode wegen Eigenbedarfs bald implementiert und dann auch wieder der Allgemeinheit als Paket zugänglich macht.

Ein weiterer Vorteil ist der großartige Support, der für R von vielen Freiwilligen, darunter die Mitglieder des R Core Teams, geleistet wird. Dazu gehört die meist sehr schnelle (innerhalb weniger Stunden oder gar Minuten) und kompetente Beantwortung von Fragen auf der Mailingliste *R-help* (s. Abschn. 2.4) sowie die Unterstützung von Entwicklern auf der Mailingliste

[7] für Details s. `http://buildd.debian.org/build.php?&pkg=r-base`
[8] `http://www.Bioconductor.org/`

R-devel. Als ganz besonderer Pluspunkt kann die häufig extrem schnelle Beseitigung von Fehlern angesehen werden. Eine fehlerbereinigte Version steht oft schon wenige Tage nach Eingang eines Fehlerberichts bereit.

Nachteile von R

Ein oftmals genannter Nachteil von R ist das Fehlen einer vollständigen grafischen Benutzeroberfläche (Graphical User Interface, kurz GUI), wie sie etwa in S-PLUS vorhanden ist. So möchten einige Benutzer gerne Verfahren über Menüs und Dialogboxen erreichen können, um ihre Analysen „zu klicken“. Nicht nur wegen der Portabilität von R ist die Entwicklung einer GUI auf den verschiedenen Betriebssystemen sehr aufwendig. Werkzeuge zur GUI Entwicklung stehen hingegen zur Verfügung, etwa das Paket **tcltk**. Wer sich mit der Sprache beschäftigt, wird feststellen, dass er sehr bald anstelle des Klickens Aufgaben schneller und lieber per kurzem Befehl in der Kommandozeile absetzt. Eine Beschreibung verschiedener Ansätze zu grafischen Benutzeroberflächen gibt es in Anhang C.

Leider ist direkt in R weder dynamische noch interaktive Grafik enthalten, d.h. Funktionen zum Rotieren von Punktwolken oder auch für verknüpfte Grafik, um markierte Datenpunkte gleichzeitig in verschiedenen Grafiken hervorheben zu können. In Abschn. 8.3 werden Pakete vorgestellt, die solche Art von Grafiken zur Verfügung stellen oder Schnittstellen zu Programmen bieten, die entsprechende Visualisierungsformen beherrschen.

Bei R handelt es sich um eine Interpretersprache, d.h. Programmcode wird nicht kompiliert, sondern zur Laufzeit interpretiert. Daher scheint R in Extremsituationen (u.a. bei Benutzung unnötiger Schleifen) langsam zu sein. Meist kann durch effizienteres Programmieren unter Ausnutzung der Eigenschaften der Sprache (s. Kap. 5) eine deutliche Geschwindigkeitssteigerung erreicht werden. Wenn Geschwindigkeit von besonderer Relevanz ist, können in dieser Hinsicht problematische Teile eines Programms auch z.B. durch eine in C programmierte DLL ersetzt werden (s. Abschn. 9.1).

1.3 Überblick

Zunächst werden in Kap. 2 wesentliche Grundlagen vermittelt, deren Kenntnis unverzichtbar für R Benutzer ist, insbesondere die Handhabung von Datenstrukturen und die Benutzung von Konstrukten. Ebenso grundlegend ist die in Kap. 3 beschriebene Ein- und Ausgabe von Daten verschiedenster Formate und der Zugriff auf Datenbanken.

Wer eigene Programme schreiben will, die über die Benutzung vorhandener Funktionen hinausgehen, etwa zur Automatisierung von Abläufen oder

zur Implementation neuer Verfahren, wird mehr Details zur Sprache wissen wollen. Kapitel 4 ist u.a. Regeln zum Auffinden von Objekten und zur Auswertung von Ausdrücken gewidmet. Ebenso wird gezeigt, wie sich Fehler finden und beseitigen lassen. Die effiziente Programmierung (s. Kap. 5) ist auch heutzutage (so schnell moderne Rechner auch sein mögen) von hohem Interesse, gerade wenn rechenintensive Verfahren auf sehr großen Datenbeständen angewandt werden oder Methoden online – mitten im Datenstrom – mitrechnen müssen. Für objektorientiertes Programmieren in R werden zwei Varianten in Kap. 6 vorgestellt.

Da R eine Sprache für Datenanalyse ist, darf die Übersicht über Funktionen für statistische Verfahren in Kap. 7 nicht fehlen, die aber wegen der enormen Fülle der in R implementierten statistischen Verfahren keineswegs vollständig sein kann. Den Anwendern wird eine gute R bezogene Literaturübersicht zu den verschiedenen Gebieten der Statistik gegeben. Die in allen Kapiteln vorhandenen Beispiele sind, sofern es sinnvoll ist, auf statistische Anwendungen bezogen.

Die enorme Fähigkeit zur flexiblen und hochwertigen Grafikproduktion wird in Kap. 8 ausführlich beschrieben. Es wird nicht nur auf die Anwendung vordefinierter Grafikfunktionen eingegangen, sondern auch auf deren Ergänzung und das Entwickeln eigener Visualisierungsformen und deren Umsetzung in Funktionen. Neben den „konventionellen“ Grafikfunktionen wird auch der Einsatz von Trellis-Grafiken und deren Formelinterface erläutert.

Das Einbinden von Quellcode (C, C++, Fortran), die Integration von anderen Programmen in R, der Batch Betrieb und Interaktionen mit dem Betriebssystem werden in Kap. 9 erläutert. In Kap. 10 ist nachzulesen, wie man seine Funktionen (und Daten) dokumentieren und sie mit anderen – sei es innerhalb der Abteilung oder mit der ganzen R Gemeinde – mit Hilfe von Paketen teilen kann. Auch Vorgehen zur Installation und Verwaltung von Paketen in Libraries wird dort beschrieben.

Im Anhang des Buches werden Hilfestellungen zur Installation von R und zur Konfiguration eines geeigneten Editors für die Programmierung geleistet. R bezogene englische und deutsche Begriffe werden gegenübergestellt.

2

Grundlagen

Dieses Kapitel widmet sich den Grundlagen, wobei der Einstieg über die intuitive Benutzung von R als „Taschenrechner“ erfolgt (Abschn. 2.1). Zuweisungen und Objekte (Abschn. 2.2–2.3) sind von fundamentaler Bedeutung in jeder Sprache. Die Benutzung von Hilfe (Hilfeseiten, Handbücher, weiterführende Literatur usw., s. Abschn. 2.4) ist nicht nur zum Erlernen von R, sondern auch in der täglichen Anwendung von zentraler Bedeutung. Die in Abschn. 2.5 gezeigte Beispielsitzung soll durch implizites Kennenlernen von Funktionalität Interesse an den im Anschluss eingeführten weiteren Grundlagen der Sprache wecken.

2.1 R als Taschenrechner

Im Büro des Autors wird man keinen Taschenrechner finden, denn R ist nicht nur mindestens genauso einfach, schnell und intuitiv zu bedienen, sondern auch wesentlich mächtiger. Nach dem Start von R ist es möglich, direkt einfache Ausdrücke (*expressions*) in der Konsole einzugeben:

```
> 1 + 2 * 3
[1] 7
> 2 * 5^2 - 10 * 5  #  ein Kommentar
[1] 0
> 4 * sin(pi / 2)
[1] 4
> 0 / 0             #  nicht definiert (Not a Number)
[1] NaN
```

Es fällt auf, dass die Punkt- vor Strichrechnung beachtet wird. Details zur Reihenfolge, in der Rechenoperatoren angewandt werden, können auf einer entsprechenden Hilfeseite nachgelesen werden, die durch Eingabe von `?Syntax`

aufgerufen wird. Bei gleichberechtigten Operatoren erfolgt die Auswertung eines Ausdrucks immer von links nach rechts. Ausnahmen bilden der Potenzoperator (`^`) und Zuweisungen (s. Abschn. 2.2). Es ist oft sinnvoll Klammern zu setzen, um die Lesbarkeit eines Programms zu verbessern und Irrtümern vorzubeugen.

Das Zeichen `#` zeigt den Beginn eines Kommentars an. Alle darauf folgenden Zeichen innerhalb derselben Zeile werden nicht ausgewertet. Eine andere Möglichkeit zum Auskommentieren *mehrerer* Zeilen wird in Abschn. 2.10.1 beschrieben.

Grundlegende arithmetische Operatoren, Funktionen und Werte

Schon bei der Benutzung von R als „Taschenrechner“ werden neben grundlegenden arithmetischen Operatoren bald einige elementare Funktionen benötigt. Eine Auswahl derjenigen, die auch beim Programmieren von fundamentaler Bedeutung sind, werden zusammen mit einigen Konstanten und Werten, die dem Benutzer in R häufig begegnen, in Tabelle 2.1 aufgeführt.

Tabelle 2.1. Grundlegende arithmetische Operatoren, Funktionen und Werte

Operator, Funktion, Wert	Beschreibung
`^` oder `**`	Potenz
`*, /`	Multiplikation, Division
`+, -`	Addition, Subtraktion
`%/%`	Ganzzahlige Division
`%%`	Modulo Division
`max(), min()`	Extremwerte
`abs()`	Betrag
`sqrt()`	Quadratwurzel
`round(), floor(), ceiling()`	Runden (Ab-, Auf-)
`sum(), prod()`	Summe, Produkt
`log(), log10(), log2()`	Logarithmen
`exp()`	Exponentialfunktion
`sin(), cos(), tan(),` `asin(), acos(), atan()`	trigonometrische Funktionen
`pi`	Die Zahl π
`Inf, -Inf` (infinity)	Unendlichkeit
`NaN` (Not a Number)	nicht definiert
`NA` (Not Available)	fehlende Werte
`NULL`	leere Menge

Kurzeinführung zu Funktionen

Generell kann in diesem Buch eine Auflistung von Funktionen wegen der Fülle an verfügbaren Funktionen selten vollständig sein. Eine Auflistung aller in R enthaltenen Funktionen (nicht die in selbst nachinstallierten Paketen, s. Kap. 10) wird vom R Development Core Team (2006a) angegeben. Für die genaue Syntax zum Aufruf von Funktionen sei auf Abschn. 4.1 verwiesen, da der Aufruf einer (einfachen) Funktion, wie etwa im vorigen Beispiel von `sin(pi/2)`, schon intuitiv klar sein sollte.

Für den Augenblick gehen wir von einer leicht vereinfachten Sichtweise aus, denn für das vollständige Verstehen von Funktionen werden noch weitere Grundlagen benötigt. Sei ein Objekt `foo` eine Funktion. Dann kann diese Funktion mit folgender Syntax aufgerufen werden:

```
foo(Argument1 = Wert1, Argument2 = Wert2, usw.)
```

Eine Funktion muss keine Argumente haben (Aufruf: `foo()`). Wenn eine Funktion ein oder mehrere Argumente hat, so werden diese benannt oder unbenannt beim Aufruf angegeben, also etwa z.B. `sin(pi/2)` oder `sin(x = pi/2)`.

2.2 Zuweisungen

Ergebnisse von Berechnungen sollen beim Programmieren häufig als Objekte gespeichert werden, damit sie später weiter verwendet werden können. Bei solchen Ergebnissen kann es sich um einfache Skalare handeln, aber auch um komplexere Objekte. In R ist prinzipiell alles ein Objekt (s. auch Chambers, 1998). Details zu Objekten, insbesondere auch zulässige Namen für Objekte, werden in Abschn. 2.3 besprochen.

Für dieses „Speichern“ benutzt man eine Zuweisung (*assignment*). Es gibt, z.T. historisch bedingt, mehrere Möglichkeiten für solche Zuweisungen, die hier zunächst beispielhaft gezeigt werden:

```
> x1 <- 3.25    # dem Objekt x1 wird die Zahl 3.25 zugewiesen
> x1            # schreibt den Wert von x1 auf die Konsole
 [1] 3.25

> x2 = 3.25     # neu, aber nur eingeschränkt zu empfehlen
> 6 -> x3       # schlecht lesbar und verwirrend
```

Der Zuweisungspfeil (`<-`), bestehend aus einem „kleiner als“(`<`) und einem Minuszeichen (`-`), ist die in R gebräuchlichste und in diesem Buch verwendete Möglichkeit. Der Ausdruck rechts von diesem Pfeil wird dabei ausgewertet (wie bekannt von links nach rechts) und dann dem auf der linken Seite genannten Objekt zugewiesen. Bei mehreren aufeinander folgenden Zuweisungen

(`x1 <- x2 <- 3.25`) erfolgen die Zuweisungen selbst jedoch von rechts nach links, also zuerst zu `x2` und dann zu `x1`.

Das Gleichheitszeichen (`=`) ist erst kürzlich nach der Definition von Chambers (1998) eingeführt worden und wird z.Zt. noch recht selten verwendet. Es kann aber auch zur Verwirrung beitragen, da es beispielsweise bereits bei benannten Argumenten in Funktionen (Abschn. 4.1) und bei logischen Vergleichen (Abschn. 2.7) als syntaktisches Element verwendet wird.

Von der Benutzung des umgekehrten Zuweisungspfeils (`->`) wird hier abgeraten. Längere Programme werden nämlich sehr unübersichtlich, wenn Objekte nicht linksbündig definiert sind.

Der Unterstrich (`_`) ist ein veraltetes Zuweisungssymbol, das in aktuellen Versionen von `R` nicht mehr akzeptiert wird. Alte Programme können noch dieses Zuweisungssymbol enthalten, so dass der Unterstrich hier durch einen Pfeil (`<-`, oder das Gleichheitszeichen) ersetzt werden muss.

Weitere Möglichkeiten für Zuweisungen bieten der Doppelpfeil (`<<-`) und die Funktion `assign()`. Beide werden mit ihren speziellen Eigenschaften in Abschn. 4.3 beschrieben. Wer nicht sicher weiß, dass sie im konkreten Fall wirklich benötigt werden, sollte diese Zuweisungen auch nicht einsetzen.

Es fällt auf, dass ein Objekt nur dann auf der Konsole ausgegeben wird, wenn es aus einer Berechnung entsteht oder direkt angegeben wird, aber nicht im Falle von Zuweisungen. Das liegt daran, dass bei Angabe eines Objekts (ohne Zuweisung) dieses implizit mit der Funktion `print()` auf der Konsole ausgegeben wird. Im Falle einer Zuweisung kann eine solche Ausgabe explizit erreicht werden durch direkten Aufruf von `print()` oder wieder implizit durch Eingabe des Objektnamens in der folgenden Zeile. Die Umklammerung der gesamten Zuweisung bewirkt, dass der darin enthaltene Ausdruck als Objekt aufgefasst wird, so dass dann die Ausgabe erfolgt:

```
> x1 <- 7
> x1
 [1] 7
> print(x1)
 [1] 7
> (x1 <- 3.25)
 [1] 3.25
```

Übersichtlichkeit

Weil Code meist aus Zuweisungen besteht, gibt es hier ein entsprechendes Beispiel für schlecht und gut lesbaren Programmcode:

```
> neueVariable1 <- x1
> neueVariable2=neueVariable1+2#kaum lesbar!
> neueVariable2 <- neueVariable1 + 2        # jetzt besser lesbar
```

In der ersten Zeile wird der Wert von `x1` dem Objekt `neueVariable1` zugewiesen. Sinnvoll eingesetzte Groß- und Kleinschreibung eignet sich gut zur Strukturierung von Objektnamen.

Bei Betrachtung der beiden letzten Zeilen fällt auf, dass man sinnvoll Leerzeichen um Zuweisungssymbole und Operatoren einfügen sollte, um die Lesbarkeit zu erhöhen. Auch nach Kommata eingefügte Leerzeichen tragen oft zur besseren Übersichtlichkeit bei. Mehr Details zur Übersichtlichkeit und Lesbarkeit von Code findet man in Abschn. 5.1 auf S. 101.

2.3 Objekte

Alles ist ein Objekt, jegliche Art Daten und Funktionen! So ist die Matrix `X` genauso ein Objekt wie die Funktion `foo()`.

In Abschn. 2.2 wurden durch Zuweisungen bereits erste Objekte selbst erzeugt. Wurde vergessen, ein Ergebnis einem Objekt zuzuweisen, so kann auf das Ergebnis des zuletzt ausgewerteten nicht trivialen Ausdrucks mit `.Last.value` zugegriffen werden.

R ist eine objektorientierte Sprache. D.h. es gibt Klassen von Objekten, für die es spezielle Methoden bestimmter (generischer) Funktionen geben kann (für Details s. Kap. 6). Durch eine Klasse werden damit bestimmte Eigenschaften eines Objekts festgelegt. Jedes Objekt hat (prinzipiell seit R-1.7.0) eine Klasse, die mit `class()` abgefragt wird. Ebenso hat jedes Objekt eine Länge (`length()`).

Des Weiteren ist R eine vektorbasierte Sprache. Jedes Objekt wird intern durch einen Vektor repräsentiert (Abschn. 2.9.1).

Objektnamen

Objektnamen sollten mit einem Buchstaben beginnen, dürfen aber auch Zahlen und einige Sonderzeichen (aber z.B. keine Operatoren) enthalten.

Wenn ein Objektname irregulär ist in dem Sinne, dass er nicht mit einem Buchstaben beginnt oder unerlaubte Sonderzeichen enthält, so ist der Zugriff darauf schwierig und in keinem Fall ist die Benutzung eines solchen Namens zu empfehlen. Der Zugriff kann aber in der Regel trotzdem mit in einfachen rückwärts gerichteten Anführungszeichen (Backticks, s. auch Abschn. 2.11, S. 56) gesetzten Objektnamen erfolgen, z.B. `` `1x` ``.

Groß- und Kleinschreibung wird beachtet, die Objekte `a` und `A` sind also verschieden. Geschickte Groß- und Kleinschreibung (z.B. in `neueVariable`) eignet sich dazu, Objektnamen zu strukturieren. Ein Objekt, dessen Name mit einem Punkt beginnt, ist versteckt.

Um Verwirrung und Fehler zu vermeiden, sollte man keine Namen doppelt vergeben. Darunter fallen auch solche Namen von bereits vorhandenen Funktionen und Konstanten, insbesondere häufig verwendete wie z.B.: `c()`, `t()`, `F`, `FALSE`, `T`, `TRUE`, `pi` usw.

Datentyp und Attribute

Jedes Objekt hat einen Datentyp oder auch Modus (*mode*), der mit `mode()` abgefragt werden kann. Das ist für Daten meist *logical*, *numeric*, *complex* oder *character* (s. Abschn. 2.8), während Sprachobjekte meist vom Modus *function* oder *expression* sind.

Ein Objekt kann weitere *Attribute* besitzen, die mit den Funktionen `attributes()` und `attr()` abgefragt und gesetzt werden können. Objekte können auch gleich mit ihren Attributen mit Hilfe der Funktion `structure()` erzeugt werden. Auch wenn die Syntax im folgenden Beispiel noch nicht vollständig bekannt ist, beleuchtet es die Nützlichkeit von Attributen im Fall einer Matrix:

```
> (X <- matrix(1:6, 2))     # Erzeugen einer Matrix X
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> mode(X)
[1] "numeric"
> attributes(X)             # Eine Matrix hat Dimensionsattribute
$dim
[1] 2 3
```

Die sehr nützliche Funktion `str()` zeigt die Struktur eines Objekts inklusive seines Modus, seiner Attribute und seiner geeignet gekürzten Daten an.

2.4 Hilfe

Sowohl zum täglichen Arbeiten mit R als auch zum Erlernen der Sprache gehört die häufige Benutzung von Hilfe. Das sind nicht nur Hilfeseiten zu einzelnen Funktionen oder zur Syntax, sondern auch Erläuterungen in einem größeren Kontext, wie man sie in Handbüchern oder Büchern wie diesem findet. Unterschiedliche Vorkenntnisse erfordern dabei die jeweils angemessene Dokumentation und Erklärung.

Hier wird beschrieben, welche Handbücher, Bücher, Hilfefunktionen, Suchmöglichkeiten und auch Mailinglisten dem Anwender zur Verfügung stehen, und wie man diese Möglichkeiten angemessen nutzt (s. dazu auch Ligges, 2003a).

2.4.1 Das integrierte Hilfesystem

Zugriff auf die Dokumentation zum Thema *help*, in diesem Fall der Name einer Funktion, bietet `?help`. Das Fragezeichen (`?`) ist sicherlich der am meisten benutzte Weg, Hilfe zu erlangen, während die Funktion `help()` zwar flexibler, dafür aber ein wenig umständlicher zu benutzen ist.

Formate

Als Voreinstellung wird eine Hilfeseite als formatierter Text in R angezeigt. Andere Formate kann man wählen, indem man ein entsprechendes Argument zur Funktion `help()` angibt (einmalig für den konkreten Hilfeaufruf) oder mit Hilfe der Funktion `options()` eine entsprechende Option setzt (für die gesamte R Sitzung). Zum Beispiel kann man das HTML Format durch Setzen des Arguments `htmlhelp=TRUE` anzeigen lassen. Das HTML Format bietet u.a. den Vorzug, dass Verknüpfungen (*links*) zu anderen Hilfeseiten durch einfaches Anklicken verfolgt werden können. Details zu den verschiedenen Formaten (kann sich je nach Betriebssystem unterscheiden) erhält man mit `?help`.

Unter Windows eignet sich das Compiled HTML Format. Dazu wird `options(chmhelp = TRUE)` gesetzt. Um den Befehl nicht in jeder Sitzung erneut eingeben zu müssen, kann ein entsprechender Eintrag in der Datei '.Rprofile' erfolgen (s. Anhang A.2).

Suchen

Häufig ist der exakte Name einer interessierenden Funktion, eines Datensatzes oder sonstigen Themas jedoch unbekannt. Es besteht also der Bedarf, nach Dokumentation zu suchen. Eine solche Suche kann mit Hilfe der Funktion `help.search()` ausgeführt werden, die eine gegebene Zeichenfolge in Namen, Titeln und Schlagworten (*keywords*) von Hilfeseiten sucht (s. `?help.search` für Details).

Falls nötig kann auch mit dem Teil eines kompletten Namens nach Hilfethemen gesucht werden. Hier ist `apropos()` die richtige Wahl.

Einen breiten Überblick über Dokumentation und Hilfe im HTML Format erhält man mit `help.start()`. Diese Funktion ruft einen Webbrowser mit einer Indexseite auf, die Links zu weiteren Seiten bietet. Darunter sind HTML Versionen der Handbücher und FAQs (s. Abschn. 2.4.2), eine Liste von installierten Paketen, eine Menge weiterer Informationen und eine Seite, die eine Suchmaschine und einen Schlagwortindex enthält. Auf der letztgenannten Seite kann nach Schlagworten, Funktionen, Datensätzen und Text in den Titeln von Hilfeseiten gesucht werden. Der Schlagwortindex ist sehr nützlich, wenn andere Suchmöglichkeiten versagt haben, denn der Benutzer erhält zu einem

gewählten Schlagwort eine komplette Liste aller Funktionen und Datensätze, die zu diesem Schlagwort passen.

Natürlich ist es mit den o.g. Hilfsmitteln leider nicht möglich, nach Funktionalität, Daten und Dokumentation in (evtl. unbekannten) Paketen (s. Kap. 10) zu suchen, die nicht auf dem lokalen System installiert sind. Eine Liste von Paketen und deren Titel sowie ein Index der darin enthaltenen Funktionen und das zugehörige Referenzhandbuch sind vom CRAN abrufbar. Trotzdem kann es schwierig sein, die interessierende Funktion bzw. das interessierende Paket zu finden. Genau zu diesem Zweck gibt es eine von Jonathan Baron eingerichtete Suchmaschine[1], die auch über die Suchseite von CRAN[2] und die Funktion `RSiteSearch()` innerhalb von R zugänglich ist. Hier können neben Hilfeseiten und Handbüchern auch Archive von Mailinglisten durchsucht werden.

2.4.2 Handbücher und weiterführende Literatur

Die im Literaturverzeichnis dieses Buches angegebene Literatur, die zum Teil auch in diesem Abschnitt zitiert wird, ist bis auf sehr wenige Ausnahmen in englischer Sprache verfasst, da Literatur in deutscher Sprache zu vielen der behandelten Themen fehlt.

Die Handbücher von R

Zu R gibt es eine ganze Reihe von Handbüchern, die in der Regel mitinstalliert werden. Sie sind dann im Verzeichnis '.../doc/manuals/' zu finden und auch mittels `help.start()` zugänglich.

Die aktuellen Handbücher sind auf CRAN einzeln zugänglich. Sie können von Interesse sein, wenn die lokal installierte R Version, und damit auch die zugehörigen Handbücher, veraltet ist oder die Installation von R nicht gelingen will.

Für die Installation und Wartung von R und den zugehörigen Paketen auf den verschiedensten Betriebssystemen ist das „R Installation and Administration“ Handbuch (R Development Core Team, 2006c) geeignet.

Das Handbuch „An Introduction to R“ (Venables et al., 2006) führt in die Grundlagen von R ein. Das Einlesen und Ausgeben von Daten (Kap. 3) ist von fundamentaler Bedeutung, wenn Datenanalyse betrieben werden soll. Dementsprechend gibt es dazu ein passendes Handbuch: „R Data Import/Export“ (R Development Core Team, 2006b).

[1] `http://finzi.psych.upenn.edu/search.html`
[2] `http://CRAN.R-project.org/search.html`

Häufig auf den Mailinglisten gestellte Fragen (*FAQ, Frequently Asked Questions*) und deren Antworten werden von Hornik (2006) in „R FAQ“ zusammengefasst. Es ist ratsam, zunächst in dieser nicht nur für Anfänger wichtigen Quelle nachzuschlagen, wenn eine aufkommende Frage nicht durch Lesen der anderen Handbücher geklärt werden kann. Außerdem gibt es auf CRAN spezielle FAQ Sammlungen[3] für die R Portierungen für den Macintosh (Iacus et al., 2006) und Windows (Ripley und Murdoch, 2006).

Für diejenigen Anwender, die bereits erste Erfahrungen mit der Programmierung gesammelt haben und etwas mehr Details zur Sprache erfahren möchten, etwa um die Sprache effizienter nutzen zu können, ist die „R Language Definition“ (R Development Core Team, 2006d, z.Zt. noch als Entwurf gekennzeichnet) eine geeignete Quelle. Dieses schon recht ausführliche Handbuch erläutert eine Reihe wichtiger Aspekte der Sprache, wie z.B. Objekte, das Arbeiten mit *expressions*, direktes Arbeiten mit der Sprache und Objekten, eine Beschreibung des Parsers und das Auffinden von Fehlern (*debugging*).

Erfahrenere Benutzer wollen u.U. eigene Sammlungen von Funktionen, Datensätzen und zugehöriger Dokumentation in Paketen zusammenstellen, sei es für die private Nutzung oder die Weitergabe eines solchen Pakets an Dritte (z.B. Veröffentlichung auf CRAN). In „Writing R Extensions“ (R Development Core Team, 2006e) findet man zur Erstellung von Paketen, Schreiben von Dokumentation usw. die relevanten Informationen. Dort ist auch beschrieben, wie Programme optimiert und die Schnittstellen für C, C++ und Fortran Code benutzt werden, und wie solcher Code kompiliert und eingebunden wird.

R Development Core Team (2006a) wird allgemein als Referenz für R angegeben. In diesem Handbuch sind die Hilfeseiten aller Funktionen von R und den relevanten Paketen enthalten.

Gelegentlich existiert (angezeigt durch `library(help = "Paketname")`) zu einem Paket ein eigenes Handbuch oder andere weiterführende Dokumentation, eine so genannte Vignette, im Verzeichnis '.../library/Paketname/doc/'.

Literatur – Online

Es gibt inzwischen eine ganze Reihe von Büchern, die online betrachtet bzw. aus dem WWW heruntergeladen werden können. Unter anderem ist einige Literatur im PDF Format frei auf CRAN[4] erhältlich oder gelinkt (alle über 100 Seiten).

Eine sehr schöne Einführung in R für Datenanalyse und Grafik gibt Maindonald (2004). Auf der WWW-Seite zu diesem Werk sind auch die darin benutzten Datensätze und Skripts vorhanden.

[3] `http://CRAN.R-project.org/faqs.html`
[4] `http://CRAN.R-project.org/other-docs.html`

Seit einigen Jahren gibt es schon das sehr umfassende Buch von Burns (1998) zur Sprache S, das nicht direkt auf R eingeht und leider in weiten Teilen nicht mehr ganz aktuell ist (z.Zt. noch in der ersten Auflage von 1998).

Als deutschsprachiges Werk ist Sawitzki (2005) zu nennen, das für eine Einführung in die Sprache S im Rahmen eines Programmierkurses für Studenten mit Grundkenntnissen in Wahrscheinlichkeitstheorie entwickelt wurde.

Eine weitere Quelle für Informationen rund um R ist die online erhältliche Zeitschrift *R News*, der Newsletter des R Projekts[5]. Hier werden referierte Artikel publiziert, Neuigkeiten und Änderungen in neuen Versionen von R besprochen, neue Pakete und Literatur angekündigt, und es gibt eine „Programmer's Niche" für Tricks bei der Programmierung in R. Des Weiteren gibt es Artikel und Beispiele zur Datenanalyse mit R und auch Tipps für Anfänger, wie etwa in der Kolumne „R Help Desk". Es ist eine Umbenennung von *R News* in *The R Journal* im Laufe des Jahres 2006 geplant.

Bücher – Offline

Neben den oben genannten Literaturstellen gibt es eine Reihe von Büchern, die auf die Sprache S im Allgemeinen und auch speziell auf R eingehen. Diese Übersicht erhebt nicht den Anspruch auf Vollständigkeit.

Ein sehr gutes und tief greifendes Buch zur Programmierung in S haben Venables und Ripley (2000) geschrieben. Darin wird sowohl auf R als auch auf S-PLUS mit deren jeweiligen Eigenarten eingegangen. Von denselben Autoren ist das unter der Kurzform **MASS** (so auch der Name eines zum Buch gehörenden Pakets der Autoren) bekannte und noch berühmtere Buch „Modern Applied Statistics with S" (Venables und Ripley, 2002). Letzteres gibt, neben einer Einführung, einen Überblick über die Anwendung einer Vielzahl moderner statistischer Verfahren mit R und S-PLUS. Die Anwendung einer Vielzahl von statistischen Standardverfahren mit R steht auch in dem Handbuch von Everitt und Hothorn (2006) im Vordergrund.

Peter Dalgaard (2002b) hat ein einführendes Buch geschrieben, das sich sehr gut als Begleitung für Studenten in den ersten Semestern eines Statistikstudiums eignet. Auf Maindonald (2004; s. auch S. 17) basiert das Buch von Maindonald und Braun (2003). Ein speziell auf die Anwendung der Regressionsanalyse (Schwerpunkt in den Sozialwissenschaften) mit R und S-PLUS abgestimmtes Buch ist Fox (2002), welches als Begleiter zu Fox (1997) gedacht ist. Für Modelle mit gemischten Effekten eignet sich das Buch von Pinheiro und Bates (2000). Die Autoren haben selbst die entsprechenden Funktionen in R und S-PLUS implementiert. Von Murrell (2005) ist das Standardwerk zu Grafiken (s. Kap. 8) mit R.

[5] *R News* ist auf CRAN zu finden: `http://CRAN.R-project.org/doc/Rnews/`

2.4.3 Mailinglisten

Manchmal helfen zur Beantwortung einer Frage oder zur Lösung eines Problems weder Handbücher noch Hilfefunktionen noch sonstige Literatur. Dann kann man andere R Anwender um Rat fragen und eine der auf CRAN erwähnten Mailinglisten[6] benutzen. Es existieren auch Listen zur Entwicklung von R und zum Mitteilen von Fehlern.

R-help

R-help ist die geeignete Mailingliste, um Fragen zu stellen, die nicht durch Handbücher, FAQ, Hilfeseiten oder sonstige Literatur beantwortet werden. Fragen werden auf dieser Liste meist innerhalb weniger Stunden, manchmal sogar innerhalb weniger Minuten, beantwortet.

Doch bevor man eine Frage stellt, sollte man die Archive[7] der Mailingliste nach einer Antwort durchsuchen. Diese Archive enthalten eine riesige Menge an Informationen und können wesentlich hilfreicher sein, als eine direkte Antwort auf eine Frage. Die meisten Fragen sind im Laufe der Jahre schon mehrmals, zumindest in ähnlicher Form, gestellt und beantwortet worden. Dadurch sind die Antworten auf leicht unterschiedliche Aspekte fokussiert und erklären in ihrer Summe einen größeren Zusammenhang.

Es ist wirklich einfach, eine Frage auf *R-help* zu stellen, es ist aber sehr sinnvoll, die Handbücher *vorher* zu lesen. Natürlich dauert es länger, ein Handbuch bzw. einen Abschnitt eines Handbuches zu lesen, als eine Frage zu stellen. Andererseits wird man bald auf ähnliche Probleme stoßen, zu deren Lösung die einmalige Lektüre eines Handbuches schon geholfen hätte. Denn bei einer solchen Lektüre lernt man über das beschränkte Problem hinaus weitere Dinge kennen, wie z.B. benutzerfreundliche, mächtige Funktionen, Programmiertricks usw. In anderen Worten: Auf Dauer zahlt sich das Lesen von Handbüchern aus. Zudem wird es auch nicht gerne gesehen, wenn Fragen gestellt werden, die in Handbüchern beantwortet sind. Man bedenke, dass die Hilfeleistenden auf *R-help* freiwillig Antworten geben und nicht dafür bezahlt werden. Wer sich an den „Posting-Guide[8]“ hält, kann beim Fragen stellen allerdings kaum etwas falsch machen.

Wer *R-help* abonnieren möchte, möge bedenken, dass die Anzahl der dort verschickten Nachrichten beträchtlich ist. Es besteht die Möglichkeit, diese zunächst in *digest* Form (einmal täglich als Sammlung aller Nachrichten) zu abonnieren.

[6] erreichbar über `http://www.R-project.org/mail.html`

[7] Die Archive können durchsucht werden, `http://CRAN.R-project.org/search.html`

[8] `http://www.R-project.org/posting-guide.html`

Andere Mailinglisten

Die Liste *R-announce* ist ausschließlich für Ankündigungen wichtiger Informationen und Neuerungen zu R gedacht. Es gibt eine sehr geringe Anzahl von Nachrichten auf dieser Liste, so dass man sich ruhigen Gewissens dort eintragen kann. Nachrichten, die auf dieser Liste erscheinen, werden automatisch auch an die Liste *R-help* geschickt. Da die Liste moderiert ist, ist auch kein Ärger mit unerwünschten Werbemitteilungen o.Ä. zu erwarten.

R-packages ist eine Liste, die insbesondere für Informationen über neue oder stark verbesserte Zusatzpakete gedacht ist. Wie auch *R-announce* ist diese Liste moderiert und Nachrichten werden an *R-help* weitergeleitet.

Für Diskussionen zur Entwicklung von R und Paketen ist die *R-devel* Mailingliste gedacht.

Weiterer *R-SIG-xyz* Mailinglisten (SIG für *S*pecial *I*nterest *G*roup), z.B. *R-SIG-Mac* zu R auf dem Mac, sind ebenfalls unter `http://www.R-project.org/mail.html` erreichbar.

Fehlerberichte

Bevor man einen Fehlerbericht einreicht, sollte man sich sicher sein, dass der Fehler wirklich in R zu finden ist. Wer sich unsicher ist, sollte lieber vorher eine Frage an *R-help* oder *R-devel* (wenn man sich etwas sicherer ist) richten. Zum Auffinden vorhandener und Einreichen neuer Fehlerberichte steht das *R Bug Tracking System*[9] zur Verfügung. Alle eingereichten Fehlerberichte (*bug reports*) zu R werden automatisch an die *R-devel* Mailingliste weitergeleitet.

2.5 Eine Beispielsitzung

Eine Beispielsitzung soll durch implizites Kennenlernen von Funktionalität Appetit auf mehr machen. Es werden in dieser Sitzung die berühmten von Anderson (1935) gesammelten *iris* Daten, die zu drei Schwertlilienarten Längen und Breiten von Blüten- und Kelchblättern enthalten, verwendet (s. auch Fisher, 1936).

Wer die Beispielsitzung am Rechner mitverfolgt, wird auch bald schon eigene Probleme intuitiv lösen können. Hierzu ist es sinnvoll, die Befehle einzeln in R einzugeben und die Ausgabe genau zu betrachten. Die Hilfeseiten zu den verwendeten Funktionen kann man sich schon anschauen. Die Leser mögen sich ermutigt fühlen, die Beispiele nach eigenem Ermessen abzuändern und Neues auszuprobieren (das gilt für *alle* Beispiele in diesem Buch), denn durch eigenes Ausprobieren lernt man die Sprache (und Fehlermeldungen) besser kennen.

[9] `http://bugs.R-project.org`

`$ R`	Je nach Betriebssystem R starten, s. Anhang A.1.
`?iris`	Ein paar Informationen zu den Daten.
`iris`	Kein großer Datensatz, aber doch schon unübersichtlich.
`summary(iris)`	Eine Zusammenfassung der Daten durch elementare Kenngrößen.
`attach(iris)`	Variablen des Datensatzes werden in den Suchpfad gehängt, s. Abschn. 2.9.5 und 4.3.
`species.n <-` `  as.numeric(Species)`	Die Variable *Species* enthält Faktoren, die Kodierung soll aber zur Farbbestimmung als Zahlen kodiert sein.
`plot(iris, col = species.n)`	Erzeugt eine Streudiagrammmatrix aller Variablen des Datensatzes.
`hist(Petal.Length)`	Ein Histogramm der Blütenblattlängen.
`op <- par(mfrow = c(2, 2),` `  lend = 1)`	Teilt das Grafik *Device* in 2 Zeilen / 2 Spalten, setzt den Typ der Linienenden auf „butt“ (1) für die folgenden Grafiken und speichert die alten Einstellungen in `op`.

`for(i in 1:4){`
`  boxplot(iris[ ,i] ~ Species,`
`  main = colnames(iris)[i])`
`}`
	In einer Schleife wird für die ersten 4 Variablen des Datensatzes jeweils eine Grafik mit parallelen Boxplots für alle Gruppen des Faktors *Species* erzeugt (Abb. 2.1).

`par(op)`	Zurücksetzen auf die alten Grafikeinstellungen.
`library("rpart")`	Mit dem Paket **rpart**, das hier geladen wird, soll ein Klassifikationsbaum bestimmt werden.
`(rpo <- rpart(Species ~ .,` `  data = iris))`	*Species* soll aus allen anderen *iris* Variablen (`.`) vorhergesagt werden.
`plot(rpo, margin = 0.1,` `  branch = 0.5)`	Das *rpart* Objekt kann man graphisch darstellen ...
`text(rpo)`	... und beschriften (Abb. 1.1, S. 2).

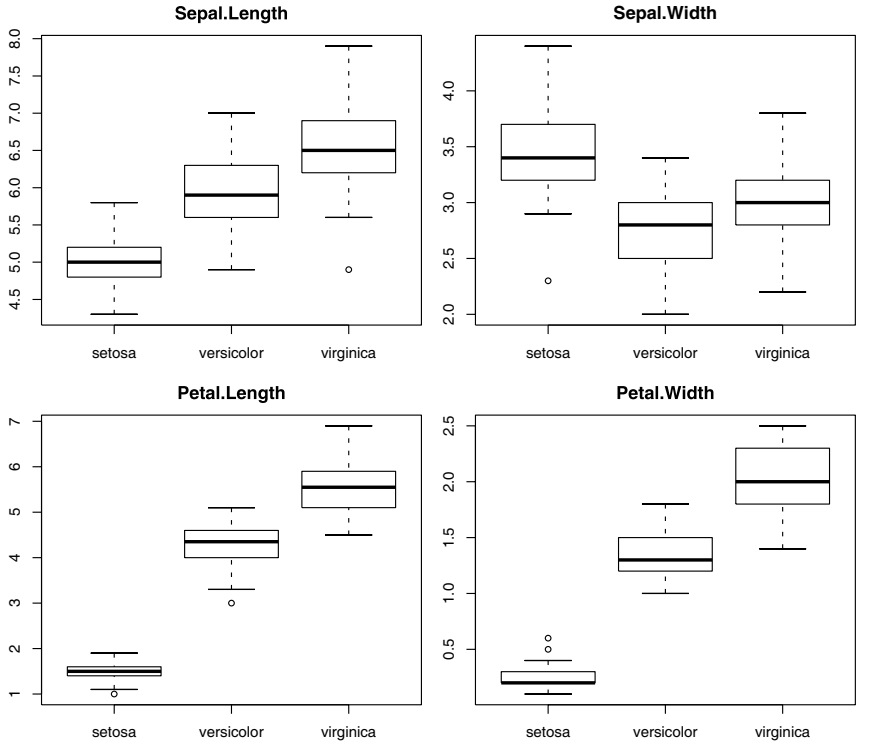

Abb. 2.1. In der Beispielsitzung erzeugte Grafik mit parallelen Boxplots für jeweils alle Gruppen des Faktors *Species* im *iris* Datensatz

`library("MASS")`	Die Funktion für (lineare) Diskriminanzanalyse (LDA) wird vom Paket **MASS** bereitgestellt.
`(ldao <- lda(Species ~ ., data = iris))`	Syntax bei `lda()` analog zu `rpart()`.
`plot(ldao, abbrev = TRUE, col = species.n)`	Zeichnen der ersten beiden Diskriminanzkomponenten.
`detach(iris)`	Datensatz wieder aus dem Suchpfad entfernen.
`ls()`	Anschauen, was sich bereits alles im *Workspace* angehäuft hat.

Mit einem zweiten Beispiel wird die Beispielsitzung fortgesetzt:

`set.seed(123)`	Setzen eines Startwerts (zur Reproduzierbarkeit) für die Zufallszahlenerzeugung.
`x <- rnorm(1000)`	Erzeugt 1000 standardnormalverteilte (Pseudo-)Zufallszahlen.
`y <- x + rnorm(x, sd = 0.5)`	
`plot(x, y)`	Hier wird y gegen x abgetragen.
`(lmo <- lm(y ~ x))`	Berechnet eine lineare Regression, dabei ist x die erklärende und y die abhängige Variable.
`summary(lmo)`	Mehr Details zur Regression, deren Resultate zuvor dem Objekt `lmo` zugewiesen wurden.
`abline(lmo, lwd = 2)`	Eine Regressionsgerade wird der bestehenden Grafik hinzugefügt.
`plot(lmo)`	Eine Reihe von Grafiken zur Modelldiagnose.
`hist(x, freq = FALSE)`	Histogramm von x.
`lines(density(x), lwd = 2, col = "red")`	Einzeichnen eines Dichteschätzers.
`qqnorm(x)`	QQ-Plot: Ist x wohl normalverteilt?
`qqline(x, lwd = 2, col = "blue")`	Die Gerade für einfacheren Vergleich fehlt noch.
`q()`	R beenden. Da man erzeugte oder geänderte Objekte sicherlich nicht speichern möchte, kann man die Frage danach verneinen.

2.6 *Workspace* – der Arbeitsplatz

Während einer R Sitzung erzeugt der Benutzer die verschiedensten Objekte, darunter vermutlich einige wichtige, die er später weiterverwenden möchte. Die meisten Objekte sind vermutlich unwichtig, denn sie wurden beim Ausprobieren verschiedener Methoden und Programmiertricks oder zur Speicherung temporärer Ergebnisse verwendet.

Wenn Objekte direkt in der R Konsole erzeugt werden, so landen sie im *Workspace*. Damit Objekte, die man in der Konsole erzeugt, nicht mit anderen Objekten durcheinander geraten, etwa Funktionen oder Datensätzen in Paketen, gibt es verschiedene *environments* (Umgebungen). Zum Beispiel erzeugt ein Funktionsaufruf eine eigene Umgebung, in der die aufgerufene Funktion eigene Objekte erzeugen kann, ohne u.U. gleichnamige Objekte im Workspace zu überschreiben. Der Workspace ist eine besondere Umgebung, die „`.GlobalEnv`“. Mehr Details zu Umgebungen findet man im Abschn. 4.3.

Unter dem Workspace kann man sich einen Büroarbeitsplatz vorstellen. Man erzeugt den ganzen Tag über Objekte, etwa Schmierzettel mit Telefonnummern, man macht sich Notizen von kurzen Besprechungen, die hinterher in einem Bericht zusammengefasst werden, usw. Ab und zu sollte man auch wieder aufräumen. Schmierzettel wird man wegwerfen wollen, während Berichte abgeheftet werden sollen. Ähnlich sollte man auch mit dem Workspace in R verfahren.

Die Funktion `ls()` zeigt Objekte im aktuellen Workspace an. Möchte man ein Objekt aus dem Workspace löschen, hilft `rm(Objektname)`. Den Workspace kann man in seinem aktuellen Zustand mit `save.image()` speichern und einen gespeicherten Workspace mit `load()` wieder laden. Dabei vereinigt `load()` den aktuellen Workspace mit dem zu ladenden Workspace so, dass gleichnamige Objekte von dem zu ladenden Workspace diejenigen des aktuellen Workspace überschreiben. Zum Speichern und Laden einzelner Objekte s. auch Kap. 3.

Das Beenden von R geschieht mit der Funktion `quit()` oder kurz `q()`. Beim Beenden wird, falls `q()` ohne Argumente ausgeführt wurde, die Frage gestellt, ob man den aktuellen Workspace speichern möchte. Bei Beantwortung der Frage mit `JA` (oder yes) wird die *history* (eine Liste der zuletzt in die Konsole eingegebenen Befehle) in der Datei '.Rhistory' und der Workspace in der Datei '.Rdata' gespeichert. Als Voreinstellung für den Speicherort wird das aktuelle Arbeitsverzeichnis gewählt (s.u.). Beim Starten von R wird der im aktuellen Arbeitsverzeichnis gespeicherte Workspace automatisch geladen. Mehr Details zur Konfigurierbarkeit gibt es in Anhang A.2.

Arbeitsverzeichnis

Das aktuelle Arbeitsverzeichnis von R ist das Verzeichnis (Ordner, Pfad), aus dem R aufgerufen wurde[10]. Es kann mit der Funktion `getwd()` abgefragt und mit `setwd()` geändert werden. Auf dem Rechner des Autors (unter Windows) z.B.:

```
> getwd()
 [1] "d:\\uwe\\R"
> setwd("c:/temp")
> getwd()
 [1] "c:\\temp"
```

Da das Zeichen „\“ ein Sonderzeichen in R ist, kann der Pfad zu einer Datei unter dem Betriebssystem Windows nicht wie dort üblich angegeben werden. Stattdessen kann entweder eine Verdoppelung (\\) oder das Zeichen „/“ in Pfadangaben verwendet werden.

Je nach Betriebssystem und eigener Arbeitsumgebung unterscheiden sich geeignete Angaben. Zwei typische Beispiele für vollständig spezifizierte Pfade sind:

```
Windows: "c:/eigenes/meinRVerzeichnis"
Linux:   "/home/meinName/meinRVerzeichnis"
```

Diese Prinzipien gelten immer, wenn in R Pfadangaben oder Angaben zu Dateien gemacht werden müssen.

2.7 Logik und fehlende Werte

Logische Operationen sind in nahezu allen Programmiersprachen bekannt und gehören zum täglichen Handwerkszeug. In R werden sie nicht nur für die Steuerung des Programmablaufs (s. Abschn. 2.10) verwendet, sondern z.B. auch zur Indizierung vieler Datenstrukturen (Abschn. 2.9).

Einen logischen Vektor (s. auch Abschn. 2.9.1) erzeugt man mit `logical()`, während man mit `is.logical(x)` fragt, ob x ein logisches Objekt ist. Man kann ein Objekt mit Hilfe von `as.logical()` zum Typ „logical“ zwingen (engl.: *coerce*). Meist sind diese Funktionen erst bei trickreichen Programmieraufgaben notwendig.

Tabelle 2.2 zeigt gebräuchliche Operatoren und Funktionen für logische Vergleiche und Verknüpfungen.

[10] Unter Windows ist es der in der auf R zeigenden Verknüpfung als Arbeitsverzeichnis eingetragene Pfad (s. auch Anhang A.1, A.2 und Abb. A.1 auf S. 214).

Tabelle 2.2. Logische Operatoren, Funktionen und Verknüpfungen

Funktion, Operator, Wert	Beschreibung
`==, !=`	gleich, ungleich
`>, >=`	größer als, größer gleich
`<, <=`	kleiner als, kleiner gleich
`!`	nicht (Negation)
`&, &&`	und
`|, ||`	oder
`xor()`	entweder oder (ausschließend)
`TRUE, FALSE` (`T, F`)	wahr, falsch

Zunächst einige Beispiele:

```
> 4 < 3
 [1] FALSE
> (3 + 1) != 3
 [1] TRUE
> -3<-2           # Fehler, "<-" ist eine Zuweisung!
 Error: Target of assignment expands to non-language object
> -3 < -2         # das Leerzeichen fehlte zuvor!
 [1] TRUE
```

Man beachte, dass die beiden Kombinationen von Minuszeichen und „kleiner als“-Zeichen bzw. „größer als“-Zeichen, `<-` und `->`, Zuweisungszeichen sind und nur mit einem trennenden Leerzeichen als Vergleiche arbeiten – ein weiteres Argument für das zur Übersichtlichkeit beitragende Einfügen von Leerzeichen.

Entweder „und“ oder „oder“ – logische Verknüpfungen

Die Operatoren `&&` (und) und `||` (oder) für logische Verknüpfungen arbeiten *nicht* vektorwertig, sondern liefern immer einen einzelnen Wahrheitswert, wie es z.B. manchmal zur Programmablaufkontrolle (`if`) nötig ist. Ihr großer Vorteil besteht darin, dass nur so viel der Logikverknüpfung ausgewertet wird, wie zu einer korrekten Aussage benötigt wird. Daher sind sie für nicht vektorwertige Berechnungen sehr effizient. Hingegen arbeiten die Operatoren `&` und `|` vektorwertig, es wird aber auch immer der vollständige Ausdruck ausgewertet.

Die Zeile

```
> (3 >= 2) && (4 == (3 + 1))
 [1] TRUE
```

verknüpft die beiden Ausdrücke `(3 >= 2)` und `(4 == (3 + 1))` mit einem logischen *und*. Das Ergebnis ist also nur wahr, wenn beide Teilausdrücke wahr sind. Im Falle eines logischen *oder* reicht es aus, wenn einer der beiden Teilausdrücke wahr ist.

Für die weiteren Beispiele reduzieren wir die Teilausdrücke jetzt auf ihre möglichen Wahrheitswerte `TRUE` und `FALSE`.

```
> FALSE && TRUE
 [1] FALSE
> TRUE && FALSE
 [1] FALSE
> TRUE || FALSE
 [1] TRUE
> x <- 0
> TRUE || (x <- 3)
 [1] TRUE
> x
 [1] 0
> FALSE || TRUE
 [1] TRUE
> FALSE || (x <- 3)
 [1] TRUE
> x
 [1] 3
```

Da in der ersten Zeile der erste Ausdruck `FALSE` ist, braucht der zweite Ausdruck einer logischen *und*-Verknüpfung nicht mehr ausgewertet zu werden, denn das Ergebnis ist bereits vorher klar. In der zweiten Zeile hingegen muss der zweite Ausdruck ausgewertet werden, denn das Ergebnis ist zunächst noch unklar. In der dritten Zeile ist das Ergebnis schon nach Auswertung des ersten Ausdrucks klar. Die darauf folgenden Zeilen zeigen, dass tatsächlich eine unnötige Auswertung nicht stattfindet: Das x bleibt zunächst (`TRUE || (x <- 3)`) unverändert, und wird erst durch `FALSE || (x <- 3)` verändert, wobei eine Auswertung des zweiten Ausdrucks erfolgen muss.

Die folgenden drei Zeilen zeigen Unterschiede zwischen vektorwertigen und nicht vektorwertigen logischen Verknüpfungen:

```
> c(TRUE, TRUE)   & c(FALSE, TRUE)      # vektorwertig
 [1] FALSE  TRUE
> c(TRUE, TRUE)  && c(FALSE, TRUE)      # nicht vektorwertig
 [1] FALSE
> c(FALSE, FALSE) | c(FALSE, TRUE)      # vektorwertig
 [1] FALSE  TRUE
```

Zu den Wahrheitswerten `TRUE` und `FALSE` gibt es die zugehörigen Kurzformen `T` und `F`, die dringend vermieden werden sollten, da sie (versehentlich)

durch den Benutzer umdefiniert werden könnten. Die Kurzformen sind weder in R selbst noch in Paketen auf CRAN erlaubt. Der Grund wird durch folgendes Beispiel sofort klar:

```
> T & F
 [1] FALSE
> T & T
 [1] TRUE
> T <- 0
> T & T     # Achtung: T wurde auf 0 gesetzt und damit FALSE
 [1] FALSE
```

Rechnen mit logischen Werten

Wie man im direkt vorangegangenen Beispiel sieht, wirkt die Zahl 0, als wäre der Wert von `T` auf `FALSE` gesetzt worden. Tatsächlich kann mit logischen Werten gerechnet werden. Ein wahrer Wert (`TRUE`) wird dabei als 1, `FALSE` hingegen als 0 interpretiert:

```
> FALSE + TRUE + FALSE
 [1] 1
> TRUE + TRUE + FALSE
 [1] 2
> TRUE - FALSE - TRUE
 [1] 0
```

Die Nützlichkeit dieser Eigenschaft wird schnell in folgendem Beispiel klar:

```
> x <- c(-3, 5, 2, 0, -2)
> x < 0
 [1]  TRUE FALSE FALSE FALSE  TRUE
> sum(x < 0)
 [1] 2
```

Hier wird ein Vektor x aus 5 Elementen definiert. Wenn man nun wissen möchte, wie viele Elemente kleiner als 0 sind, so erzeugt man einen logischen Vektor durch `x < 0`. Mit `sum()` kann dann darüber summiert werden. Das interessierende Ergebnis ist die Anzahl der `TRUE` Werte des logischen Vektors.

Weitere Operationen mit logischen Werten

Es gibt eine Reihe nützlicher Funktionen, die Zusammenfassungen von logischen Vektoren liefern. Die Funktion `any()` gibt aus, ob mindestens irgendein Element eines Vektors `TRUE` ist, während `all()` die Frage beantwortet, ob alle Elemente `TRUE` sind. Um den Index aller Elemente zu erfahren, die `TRUE` sind, wird `which()` verwendet. Die stärker spezialisierten Funktionen `which.max()` und `which.min()` geben direkt den Index des Minimums bzw. des Maximums eines Vektors aus. Die folgenden Beispiele verdeutlichen die Arbeitsweise dieser Funktionen:

```
> x <- c(2, 9, 3, 1, 6) # ein einfacher Vektor
> einige <- x > 4
> alle <- x > 0
> any(einige)           # Ist "einige" min. einmal TRUE?
 [1] TRUE
> any(alle)             # Ist "alle" min. einmal TRUE?
 [1] TRUE
> any(!alle)            # Ist die Negation von "alle" TRUE?
 [1] FALSE
> all(einige)           # Ist "einige" immer TRUE?
 [1] FALSE
> all(alle)             # Ist "alle" immer TRUE?
 [1] TRUE
> which(einige)         # An welchen Stellen ist "einige" TRUE?
 [1] 2 5
> which.max(x)          # An welcher Stelle ist das Max. von "x"?
 [1] 2
```

Unter anderem wird in diesem Beispiel auch von der Negation (`!`) Gebrauch gemacht, was sich in vielen Fällen als nützlich erweisen kann.

Fehlende Werte – NAs

`NA` (*Not Available*) steht für fehlende Werte. Bei `NA` handelt es sich zunächst um eine logische Konstante, die aber beliebig in jeden anderen Datentyp umgewandelt werden kann, also z.B. auch in „character“ oder „numeric“(s. Abschn. 2.8):

```
> x <- NA
> str(x)
 logi NA
> y <- c(3, x)
> str(y)
 num [1:2] 3 NA
```

Per Definition ist das Ergebnis einer Rechnung mit einem `NA` immer wieder `NA` selbst, solange das Ergebnis nicht eindeutig ist. Insbesondere für logische Operationen hat das Konsequenzen. So ist das Ergebnis von `TRUE & NA` definitionsgemäß `NA`, wie auch der Vergleich `TRUE == NA`, aber `NA & FALSE` ist `FALSE`, da das Ergebnis für „unbekanntes“ `NA` trotzdem eindeutig ist.

Auch `x == NA` ist `NA`! Dieser Vergleich wird häufig fälschlicherweise von Anfängern benutzt, um zu überprüfen, ob es fehlende Werte gibt. Stattdessen sollte auf fehlende Werte mit der Funktion `is.na()` getestet werden. Die Ersetzungsfunktion `is.na<-()` ist für das Setzen fehlender Werte gedacht:

```
> x <- c(5, 7, NA, 22)
> is.na(x)
 [1] FALSE FALSE  TRUE FALSE
> is.na(x[1]) <- TRUE       # Setze 1. Element von x auf NA
> x
 [1] NA  7 NA 22
```

Einige Funktionen erlauben das Ausschließen fehlender Beobachtungen mit Hilfe des Arguments `na.rm`, z.B. `mean(x, na.rm = TRUE)`. Die Funktionen `na.fail()`, `na.exclude()`, `na.pass()` und `na.omit()` bieten einfache Behandlungsmöglichkeiten für fehlende Werte. Am häufigsten wird davon sicherlich `na.omit()` benutzt. Diese Funktion lässt Fälle mit fehlenden Werten weg. Die Fortsetzung des letzten Beispiels liefert:

```
> mean(x)
 [1] NA
> mean(x, na.rm = TRUE)
 [1] 14.5
> na.omit(x)
 [1]  7 22
 attr(,"na.action")
 [1] 1 3
 attr(,"class")
 [1] "omit"
```

Eine allgemeinere Einstellmöglichkeit zur Behandlung fehlender Werte bietet die Funktion `options()` mit ihrem Argument `na.action` (siehe `?options`).

Im Allgemeinen werden undefinierte Werte (`NaN`, *Not A Number*) wie `NA` behandelt. Zur Unterscheidung kann die Funktion `is.nan()` dienen, die ausschließlich auf `NaN` testet.

2.8 Datentypen

Bei der Betrachtung von Objekten in Abschn. 2.3 ist aufgefallen, dass zwischen verschiedenen Datentypen unterschieden wird. Atomare Datentypen sind grundlegend und können nicht weiter unterteilt werden.

Bei den in Tabelle 2.3 aufgelisteten Typen ist die leere Menge ein Sonderfall. Unter den anderen Typen kann eine Reihenfolge ausgemacht werden. So kann man durch eine Zeichenfolge sowohl komplexe, reelle und ganze Zahlen als auch logische Werte darstellen. Jeder andere Datentyp hingegen kann nicht beliebige Zeichenfolgen darstellen. Die komplexen Zahlen können wiederum alle in Tabelle 2.3 weiter oben aufgeführten Typen repräsentieren (z.B. kann die reelle Zahl 3.14 durch $3.14 + 0i$ repräsentiert werden), nicht aber umgekehrt.

Tabelle 2.3. Atomare Datentypen

Beschreibung	Beispiel	Datentyp
die leere Menge	`NULL`	*NULL*
logische Werte	`FALSE`	*logical*
ganze und reelle Zahlen	`3.14`	*numeric*
komplexe Zahlen	`2.13+1i`	*complex*
Buchstaben und Zeichenfolgen	`"Hallo"`	*character*

Letztendlich können logische Werte durch alle anderen (außer *NULL*) Typen repräsentiert werden, sie selbst können diese aber nicht repräsentieren.

Diese Eigenschaften sind entscheidend für die Regeln zum Zusammenführen mehrerer Objekte. Dabei wird im Allgemeinen einem Informationsverlust vorgebeugt, indem bei der Zusammenführung mehrerer Objekte als gemeinsamer Datentyp derjenige unter den Datentypen dieser Objekte gewählt wird, der in Tabelle 2.3 am weitesten unten steht. Beispiele dazu findet man in Abschn. 2.9.

Im Unterschied zu diesen Klassen von Datentypen gibt es noch den R internen Speichermodus, der mit `typeof()` abgefragt werden kann. Hier gibt es zwei Möglichkeiten für den Datentyp *numeric*, nämlich *integer* für Ganzzahlen und *double* für die Darstellung reeller Zahlen in doppelter Maschinengenauigkeit. Der Speichermodus *single* existiert intern nicht wirklich, kann aber wegen S-PLUS Kompatibilität spezifiziert werden. Die Speicherung erfolgt dann als *double*.

Vektoren eines Datentyps können mit Funktionen erzeugt werden, die meist den Namen des korrespondierenden Datentyps tragen. Das ist z.B. die Funktion `numeric()` für den Datentyp *numeric*. Ob ein Objekt einen bestimmten Datentyp hat, testet man mit Funktionen, die diesem Namen ein „*is.*“ vorangestellt haben; `is.character()` testet also auf den Datentyp *character*. Durch vorangestelltes „*as.*“ kann ein anderer Datentyp erzwungen werden. Hier einige Beispiele:

```
> (x <- pi)
 [1] 3.141593
> mode(x)
 [1] "numeric"
> typeof(x)
 [1] "double"
> (y <- as.integer(x))      # Informationsverlust!
 [1] 3
> typeof(y)
 [1] "integer"
```

```
> is.character(y)
 [1] FALSE
> x <- -1
> sqrt(x)                    # Für eine reelle Zahl x=-1 ist die
 [1] NaN                    #  Quadratwurzel von x nicht definiert.
 Warning message:
 NaNs produced in: sqrt(x)
> sqrt(as.complex(x))        # Für eine komplexe Zahl x=-1 ist das
 [1] 0+1i                   #  Ergebnis hingegen definiert.
```

Der Umgang mit Zeichenketten (Datentyp *character*) wird detailliert in Abschn. 2.11 behandelt.

Faktoren

Zur Darstellung qualitativer (diskreter) Merkmale (z.B. eine kategorielle Variable „Obst“, die die Ausprägungen „Apfel“, „Birne“ und „Banane“ annehmen kann) bieten sich Faktoren an, denen wir später immer wieder begegnen werden. Dieser Datentyp ist kein atomarer Typ und wird z.B. mit Hilfe der generierenden Funktion `factor()` erzeugt. Intern wird eine Nummer vergeben, nach außen wird ein Faktor aber durch seinen Namen dargestellt.

Einige der in folgendem Beispiel verwendeten Funktionen werden erst im Abschn. 2.9.1 formal eingeführt:

```
> (trt <- factor(rep(c("Control", "Treated"), c(3, 4))))
 [1] Control Control Control Treated Treated Treated Treated
 Levels: Control Treated
> str(trt)
 Factor w/ 2 levels "Control","Treated": 1 1 1 2 2 2 2

> mode(trt)
 [1] "numeric"
> length(trt)
 [1] 7
```

Man sieht an der durch `str()` angezeigten Struktur des Objekts, dass zwei Faktoreinstellungen mit den Namen `"Control"` und `"Treated"` vorhanden sind, die mit den Zahlen 1 und 2 kodiert sind. Entsprechend ist der zugrunde liegende atomare Datentyp *numeric*.

2.9 Datenstrukturen und deren Behandlung

Unter Datenstrukturen versteht man eine Beschreibung dessen, wie die Daten dargestellt werden und angeordnet sind. Je nach Problemstellung und Art der

Daten gibt es verschiedene geeignete Darstellungsarten. Aus der Informatik sind Datenstrukturen meist daher bekannt, dass Daten so angeordnet werden, dass im Sinne der Rechengeschwindigkeit effiziente Algorithmen entstehen können, z.B. Bäume für Sortierverfahren.

In der Statistik werden Datenstrukturen eher dazu benutzt, die Natur der Daten nachzubilden, so dass sie angemessen repräsentiert werden und in Modellen spezifiziert werden können. Wir werden u.a. folgende Datenstrukturen kennen lernen: Vektoren, Matrizen, Arrays, Datensätze (*Data Frames*) und Listen. In R gibt es keine Skalare im eigentlichen Sinne. Sie werden durch Vektoren der Länge 1 dargestellt.

Immer wieder wird uns in den folgenden Abschnitten die Funktion `str()` die Struktur eines Objekts anzeigen. Sie ist immer dann von besonderer Nützlichkeit, wenn Unklarheit über die Struktur eines Objekts herrscht, weil sie eben diese Struktur in komprimierter Form darstellt.

2.9.1 Vektoren

Vektoren sind sicherlich jedem Anwender von R aus elementarer Mathematik ein Begriff. In R sind sie ähnlich elementar, denn fast alle Objekte in R werden intern durch Vektoren repräsentiert. Anwender sollten das zwar im Hinterkopf behalten, zunächst einmal aber die von R auf Benutzerebene angebotenen Datenstrukturen verwenden, zu denen natürlich auch Vektoren gehören.

Mit der Funktion `c()` (für *combine* oder *concatenate*) kann man auf die einfachste Art Vektoren erzeugen, indem man andere Vektoren miteinander verknüpft. Das sind häufig auch vom Benutzer eingegebene Skalare, also Vektoren der Länge 1.

```
> (x <- c(4.1, 5.0, 6.35))
 [1] 4.10 5.00 6.35
> (x <- c(7.9, x, 2))
 [1] 7.90 4.10 5.00 6.35 2.00
> (y <- c("Hallo", "Leser"))
 [1] "Hallo" "Leser"
> (y <- c("Hallo", TRUE, x))
 [1] "Hallo" "TRUE"  "7.9"   "4.1"   "5"     "6.35"  "2"
> length(y)
 [1] 7
```

Vektoren können von einem beliebigen Datentyp (Abschn. 2.8) sein, dieser muss jedoch für alle Elemente innerhalb eines Vektors derselbe sein. In obigem Beispiel wird zunächst ein Vektor `x` aus numerischen Werten erzeugt, der in einem zweiten Schritt vorne und hinten erweitert wird. Genauer gesagt verknüpft man die Vektoren `7.9`, `x` und `2` und überschreibt das alte x. In

einem dritten Schritt wird ein zweielementiger Vektor `y` des Typs *character* erzeugt.

Im vorletzten Schritt wird `y` überschrieben mit einer Verknüpfung aus dem *character* Vektor `"Hallo"`, `TRUE` vom Typ *logical* und dem numerischen Vektor `x`. Weil alle Elemente denselben Datentyp haben müssen, treten dabei die in Abschn. 2.8 beschriebenen Regeln in Kraft, d.h. es wird automatisch der niedrigste Datentyp erzwungen, der gerade noch alle in den einzelnen Elementen enthaltenen Informationen verlustfrei darstellt. In diesem Fall ist das *character*, denn `"Hallo"` ist weder als numerischer noch als Wahrheitswert darstellbar.

Analog dazu wird in `c(5, 2+1i)` die reelle Zahl als komplexe darstellbar sein, aber nicht umgekehrt. Das Ergebnis ist folgerichtig `[1] 5+0i 2+1i`.

Die Länge eines Vektors, d.h. die Anzahl seiner Elemente, zeigt die Funktion `length()` im letzten Schritt des Beispiels an. Ein Vektor kann mit Hilfe der Funktion `t()` transponiert werden.

Es ist auch möglich, die Elemente eines Vektors zu benennen. Zur geeigneten Darstellung wird die Ausgabe dabei ein wenig angepasst:

```
> c(WertA = 5, WertB = 99)
 WertA WertB
     5    99
```

Folgen und Wiederholungen

Häufig benötigt man in Vektoren einfache Zahlenfolgen oder eine gewisse Anzahl von Wiederholungen desselben Objekts. Ganzzahlige Zahlenfolgen mit Abstand 1 können mit Hilfe des Doppelpunkts (`:`) erzeugt werden:

```
> 3:10
 [1]  3  4  5  6  7  8  9 10
> 6:-2
 [1]  6  5  4  3  2  1  0 -1 -2
```

Die allgemeinere Funktion `seq()` kann beliebige Zahlenfolgen gleichen Abstands erzeugen. Alternativ kann auch per Argument `along` ein zu einem angegebenen Vektor gehöriger Indexvektor passender Länge erzeugt werden:

```
> seq(3, -2, by = -0.5)
 [1]  3.0  2.5  2.0  1.5  1.0  0.5  0.0 -0.5 -1.0 -1.5 -2.0
> x <- c(5, 7)
> seq(along = x)
 [1] 1 2
```

Diese Konstruktion wird häufig in Schleifen (s. Abschn. 2.10.2) verwendet.

Für Wiederholungen desselben Objekts kann die Funktion `rep()` verwendet werden, deren Funktionsweise im folgenden Beispiel direkt ersichtlich ist.

```
> rep(3, 12)
 [1] 3 3 3 3 3 3 3 3 3 3 3 3
> rep(c(5, 7), 2)
 [1] 5 7 5 7
> rep(3:5, 1:3)                    # 1x3, 2x4, 3x5
 [1] 3 4 4 5 5 5
> rep(TRUE, 3)
 [1] TRUE TRUE TRUE
```

Bemerkenswert ist, dass ein zu wiederholender Vektor mit einem Vektor aus Wiederholungsanzahlen kombiniert werden kann. Die Vektorwertigkeit von R tritt hier zu Tage.

Rechnen mit Vektoren

Das Rechnen mit Vektoren geschieht komponentenweise:

```
> (x <- c(4.1, 5.0, 6.35) * 2)     # 4.1*2, 5.0*2, 6.35*2
 [1]  8.2 10.0 12.7
> x + 5:7                          # 8.2+5, 10.0+6, 12.7+7
 [1] 13.2 16.0 19.7
> 3:5 - 1:6                        # 3-1, 4-2, 5-3, 3-4, 4-5, 5-6
 [1]  2  2  2 -1 -1 -1
```

Im ersten Schritt werden alle Elemente eines mit `c()` erzeugten Vektors mit 2 multipliziert und dem Objekt `x` zugewiesen. Dieser Vektor wird daraufhin zu einem anderen Vektor gleicher Länge elementweise addiert.

Im dritten Schritt scheint das Ergebnis zunächst eigenartig zu sein. Wenn die Längen (`length()`) zweier Vektoren nicht übereinstimmen ($m < n$), so wird der kürzere Vektor so oft wiederholt wie nötig. Also wird `3:5` wiederholt, weil `1:6` gerade doppelt so lang ist. Voll ausgeschrieben bedeutet das:

```
3:5 == c(3, 4, 5)                  # Länge m = 3
1:6 == c(1, 2, 3, 4, 5, 6)         # Länge n = 2*m = 6
3:5 - 1:6 == c(3, 4, 5,   3, 4, 5) - c(1, 2, 3, 4, 5, 6)
```

Wenn die Länge eines Vektors nicht das ganzzahlige Vielfache der Länge des anderen ist, wird durch Wiederholung der ersten $n - m$ Elemente verlängert bis alles passt, zusätzlich wird jedoch eine Warnung ausgegeben:

```
> 3:5 - 2:3
 [1] 1 1 3
 Warning message:
 longer object length
    is not a multiple of shorter object length in: 3:5 - 2:3
```

Sollte man mit Vektoren eine Matrixmultiplikation durchführen wollen, so hilft der Operator `%*%`. Das Skalarprodukt zweier Vektoren ist dann z.B. wie folgt zu berechnen[11]:

```
> t(2:4) %*% 1:3
     [,1]
[1,]   20
```

Die Eingabe `2:4 %*% 1:3` hat dieselbe Wirkung. Wenn die Dimensionen nämlich nicht passen, aber durch Transponieren eines Vektors passend gemacht werden können, berechnet R das Skalarprodukt. Man sieht an der Ausgabe, dass tatsächlich eine 1×1 Matrix vorliegt. Analog ist das Vektorprodukt:

```
> 2:4 %*% t(1:3)
     [,1] [,2] [,3]
[1,]    2    4    6
[2,]    3    6    9
[3,]    4    8   12
```

Indizierung von Vektoren

Einzelne Elemente eines Vektors müssen auch einzeln angesprochen werden können, etwa mit Hilfe des zugehörigen Index. Dabei muss man sie nicht nur einzeln abfragen, sondern auch ersetzen können. Dazu wird dem Namen des Vektors der entsprechende Index in eckigen Klammern nachgestellt, z.B. greift man mit `x[3]` auf das dritte Element des Vektors `x` zu.

Als Index müssen nicht unbedingt einzelne Ganzzahlen verwendet werden:

- Mehrere Indizes können gleichzeitig als Vektor angegeben werden, es wird dann also ein Vektor durch einen Vektor von Indizes indiziert.
- Ein vorangestelltes Minuszeichen zeigt den *Ausschluss* eines oder mehrerer Elemente an (inverse Indizierung).
- Auch logische Indizierung ist möglich, wobei `TRUE` für „ausgewählte“ und `FALSE` für „abgewählte“ Elemente verwendet wird.
- Benannte Elemente können über ihren Namen angesprochen werden.
- Eine auf der linken Seite einer Zuweisung stehende Indizierung ist für Ersetzung von Elementen gedacht.
- Eine leere eckige Klammer (`[]`) dient zum Ersetzen aller Elemente eines Vektors, anstatt ihn zu überschreiben.

Diese Punkte sollen ausschließlich anhand des folgenden, ausführlich kommentierten Beispiels verdeutlicht werden, das die Sachverhalte sicherlich kürzer und doch verständlicher als viele erklärende Worte wiedergibt. Eine aufmerksame Lektüre dieses Beispiels wird daher dringend empfohlen.

[11] Tatsächlich ist hier jedoch `crossprod(2:4, 1:3)` für schnellere und stabilere Ergebnisse vorzuziehen (s. Abschn. 2.9.2).

```
> x <- c(3, 6, 9, 8, 4, 1, 2)
> length(x)
 [1] 7
> x[3]                      # das 3. Element
 [1] 9
> x[c(4, 2)]                # das 4. und 2. Element (Reihenfolge!)
 [1] 8 6
> x[-c(2, 3, 7)]            # die Elemente 2, 3, 7 ausschließen
 [1] 3 8 4 1
> (logik.vektor <- x < 4)   # TRUE, wenn x < 4, sonst FALSE
 [1]  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE
> x[logik.vektor]           # alle x, die kleiner als 4 sind
 [1] 3 1 2
> x[x < 4]                  # das Gleiche direkt
 [1] 3 1 2
> y <- c(Wasser = 3, Limo = 5, Cola = 2)
> y["Cola"]
 Cola
    2
> y["Cola"] <- 99           # das Element "Cola" ersetzen
> y
 Wasser   Limo   Cola
      3      5     99
> x[9:10] <- 10:9           # das 9. und 10. Element zuweisen
> x                         # Element 8 existierte noch nicht: NA
 [1]  3  6  9  8  4  1  2 NA 10  9
> x[] <- 2                  # alle 10 Elemente ersetzen
> x
 [1] 2 2 2 2 2 2 2 2 2 2
> (x <- 2)                  # x überschreiben (kein "[]")
 [1] 2
```

2.9.2 Matrizen

Am einfachsten können Matrizen mit der Funktion `matrix()` erzeugt werden.

Diese Funktion akzeptiert neben dem ersten Argument (`data`), das die nach Spalten oder Zeilen sortierten Elemente der Matrix als Vektor enthält, die Argumente `nrow` für die Zeilenanzahl und `ncol` für die Spaltenanzahl der Matrix. Von den drei genannten Argumenten müssen nur jeweils zwei angegeben werden, da das Dritte sich eindeutig aus den anderen beiden ergibt; z.B. ist bei einem Datenvektor der Länge 12 und 3 Zeilen klar, dass die Matrix 4 Spalten hat. Wenn das Argument `byrow` auf `TRUE` gesetzt ist, wird die Matrix aus den Daten zeilenweise aufgebaut, sonst spaltenweise.

```
> (Z <- matrix(c(4, 7, 3, 8, 9, 2), ncol = 3))
     [,1] [,2] [,3]
[1,]    4    3    9
[2,]    7    8    2
> (Y <- matrix(c(4, 7, 3, 8, 9, 2), nrow = 3, byrow = TRUE))
     [,1] [,2]
[1,]    4    7
[2,]    3    8
[3,]    9    2
> all(t(Z) == Y)        # Sind alle Elemente von t(Z) und Y gleich?
[1] TRUE
> (A <- matrix(c(4, 7, 0, 5), nrow = 2))
     [,1] [,2]
[1,]    4    0
[2,]    7    5
> (Ainv <- solve(A))    # Invertierung von A
      [,1] [,2]
[1,]  0.25  0.0
[2,] -0.35  0.2
> Ainv %*% A
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Im obigen Beispiel fragt man mit Hilfe von `all()`, ob *alle* Elemente im elementweisen (logischen) Vergleich `t(Z) == Y` wahr sind. Wie auch bei Vektoren dient `t()` zum Transponieren.

Die Funktion `solve()` löst das Gleichungssystem $A \times X = B$ für die Argumente A und B nach X auf. Im Beispiel wird nur das Argument A (hier auch Objekt `A`) angegeben. In diesem Fall wird B als passende Einheitsmatrix automatisch gesetzt, man erhält also die Inverse von A als Ergebnis.

Eine Zusammenstellung wichtiger und häufig benutzter Funktionen zum Umgang mit Matrizen ist in Tabelle 2.4 zu finden. Ebenso wie bei Vektoren müssen auch alle Elemente einer Matrix von demselben Datentyp sein.

Indizierung von Matrizen

Die Indizierung von Matrizen geschieht analog zur Indizierung von Vektoren (Abschn. 2.9.1), also auch per Index, negativem Index, Namen oder logischen Werten.

Um den Wert mit Index (i, j) (Zeile i, Spalte j) einer Matrix X anzusprechen, verwendet man die Form `X[i, j]`. Das Weglassen einer Spaltenangabe, also etwa `X[i, ]`, ermöglicht das Ansprechen des i-ten Zeilenvektors; analog liefert `X[, j]` den j-ten Spaltenvektor:

Tabelle 2.4. Wichtige Funktionen zum Umgang mit Matrizen

Funktion	Beschreibung
`%*%`	Matrixmultiplikation
`cbind()`, `rbind()`	Matrizen oder Vektoren spalten- bzw. zeilenweise zusammenfügen
`crossprod()`	$X'Y$ (oder $X'X$) sehr schnell berechnen
`diag()`	Abfragen und Setzen der Hauptdiagonalen
`dim()`, `ncol()`, `nrow()`	Anzahl von Zeilen und Spalten
`dimnames()`	Zeilen- und Spaltennamen
`eigen()`	Eigenwerte und -vektoren
`kappa()`	Konditionszahl einer Matrix
`qr()`	QR-Zerlegung
`solve()`	Invertierung einer Matrix (u.a.)
`svd()`	Singulärwertzerlegung
`t()`	Transponieren einer Matrix

```
> (X <- matrix(c(4, 7, 3, 8, 9, 2), nrow = 2))
     [,1] [,2] [,3]
 [1,]    4    3    9
 [2,]    7    8    2
> X[1, 2]
 [1] 3
> X[2, ]                # 2. Zeile
 [1] 7 8 2
> X[, 3]                # 3. Spalte
 [1] 9 2
```

Wenn ein Objekt indiziert wird, kann ein niedriger dimensionales Objekt entstehen, ohne dass es beabsichtigt ist. Das kann zu Problemen in komplexen Programmen führen. Als Lösung bietet es sich an, das Argument `drop = FALSE` zu `[]` wie im folgenden Beispiel *immer* dann zu spezifizieren, wenn Matrizen zurückgegeben werden sollen:

```
> (X <- matrix(1:4, 2))     # eine 2x2 Matrix
     [,1] [,2]
 [1,]    1    3
 [2,]    2    4
> X[1, 1]                   # nur noch ein Vektor (1D)
 [1] 1
> X[1, 1, drop = FALSE]     # ohne Verlust von Dimensionen (2D)
     [,1]
 [1,]    1
```

Die Hauptdiagonale einer Matrix kann mit `diag()` nicht nur abgefragt, sondern auch ersetzt werden. Auch eine Einheitsmatrix kann so direkt erzeugt werden.

```
> diag(X)               # Fortsetzung des letzten Beispiels
 [1] 1 4
> diag(X) <- 0          # Hauptdiagonale Null setzen
> X
      [,1] [,2]
 [1,]    0    3
 [2,]    2    0
> diag(2)               # 2x2 Einheitsmatrix (quadratisch!)
      [,1] [,2]
 [1,]    1    0
 [2,]    0    1
```

Häufig wird auf den R Mailinglisten gefragt, wie man eine obere bzw. untere Dreiecksmatrix extrahiert. Hier helfen die Funktionen `upper.tri()` und `lower.tri()`.

Diesen Funktionen liegt die Überlegung zu Grunde, dass für die obere Dreiecksmatrix einer $n \times n$ Matrix zu einer Zeile $1 \leq i \leq n$ alle Spalteneinträge $j \geq i$ mit $1 \leq j \leq n$ benötigt werden. Man erhält also die Werte auch per Indizierung mit `X[col(X) >= row(X)]`.

Struktur einer Matrix

Betrachten wir die Aussage „Jedes Objekt wird intern durch einen Vektor repräsentiert.“ (Abschn. 2.3) näher:

```
> (X <- matrix(c(4, 7, 3, 8, 9, 2), nrow = 2))
      [,1] [,2] [,3]
 [1,]    4    3    9
 [2,]    7    8    2
> str(X)
 num [1:2, 1:3] 4 7 3 8 9 2
```

Offensichtlich ist die Aussage für Matrizen richtig, denn es handelt sich dabei (genauso wie bei Arrays im folgenden Abschn.) um Vektoren mit Dimensionsattributen. Man braucht also nicht unbedingt `matrix()` zur Erzeugung von Matrizen, denn eine Zuweisung von Dimensionsattributen für einen Vektor reicht aus. Die folgenden Zuweisungen erzeugen eine Matrix Y, die identisch zu oben erzeugtem X ist:

```
> Y <- c(4, 7, 3, 8, 9, 2)
> dim(Y) <- c(2, 3)
> str(Y)
 num [1:2, 1:3] 4 7 3 8 9 2
```

Im Gegensatz zur mathematischen Sichtweise ist aus Sicht von R die Matrix damit eher ein Spezialfall eines Vektors – und nicht umgekehrt.

2.9.3 Arrays

Bisher wurden die Datenstrukturen vom in R eigentlich nicht existenten Skalar über den Vektor bis zur (zweidimensionalen) Matrix erweitert. Eine weitere Verallgemeinerung ist das beliebig dimensionale *Array*.

Arrays können beliebig viele Dimensionen besitzen und mit `array()` erzeugt werden. Bei Aufruf dieser Funktion muss, abgesehen von einem Datenvektor, das Argument `dim` als Vektor der Anzahl an Elementen je Dimension angegeben werden. Das folgende 3-dimensionale Array hat in der ersten Dimension (Zeilen) 2, in der zweiten (Spalten) 3 und der dritten Dimension wieder 2 Elemente:

```
> (A <- array(1:12, dim = c(2, 3, 2)))
 , , 1
      [,1] [,2] [,3]
 [1,]    1    3    5
 [2,]    2    4    6
 , , 2
      [,1] [,2] [,3]
 [1,]    7    9   11
 [2,]    8   10   12
> A[2, 2, 2]
 [1] 10
```

Offensichtlich erfolgt die Indizierung analog zu Matrizen (Abschn. 2.9.2) und Vektoren, wobei eine der Dimensionalität des Arrays (hier 3) entsprechende Anzahl an durch Kommata getrennten Indizes angegeben werden muss.

Es macht selten Sinn Arrays mit mehr als 3 Dimensionen zu benutzen, da die Struktur schnell unübersichtlich wird.

2.9.4 Listen

Eine sehr flexible Datenstruktur ist die *Liste*. Listen sind rekursiv definiert und können als Elemente Objekte unterschiedlicher Datenstruktur enthalten. Jedes Element kann dann natürlich auch einen anderen Datentyp haben, es muss nur innerhalb seiner eigenen Datenstruktur konsistent sein.

Eine Liste kann damit z.B. verschieden lange Vektoren oder Matrizen unterschiedlichen Typs enthalten, aber auch selbst wieder Element einer Liste sein.

Listen werden mit `list()` erzeugt. Der Zugriff auf Elemente einer Liste erfolgt mittels des `[[]]`-Operators, wobei numerische Indizes und Namen (character) benutzt werden können. Bei Eingabe eines Indexvektors wird rekursiv indiziert, d.h. `Liste[[i]]` für einen Vektor `i` ist analog zu `Liste[[i[1]]][[i[2]]]...[[i[n]]]`.

```
> (L1 <- list(c(1, 2, 5, 4), matrix(1:4, 2), c("Hallo", "Welt")))
 [[1]]
 [1] 1 2 5 4

 [[2]]
      [,1] [,2]
 [1,]    1    3
 [2,]    2    4

 [[3]]
 [1] "Hallo" "Welt"

> L1[[1]]               # 1. Element von L1
 [1] 1 2 5 4
> L1[[2]][2, 1]         # Element [2, 1] des 2. Elements von L1
 [1] 2
> L1[[c(3,2)]]          # Rekursiv: zunächst 3. Element von L1,
 [1] "Welt"            #   dann davon das 2.
```

Analog zu Vektoren können die Elemente einer Liste benannt sein, was bei den oft komplexen, durch Listen repräsentierten Objekten einen besonders großen Nutzen hat. Namen können im Index verwendet werden. Zur Vereinfachung steht der `$`-Operator bereit, mit dessen Hilfe man direkt (und ohne Anführungszeichen) auf ein benanntes Element zugreifen kann.

```
> (L2 <- list(Info = "R Buch", Liste1 = L1))
 $Info
 [1] "R Buch"

 $Liste1
 $Liste1[[1]]
 [1] 1 2 5 4

 $Liste1[[2]]
    . . . .            # usw. wie L1.

> L2[["Info"]]
 [1] "R Buch"
> L2$Info
 [1] "R Buch"
```

Der Weg, mehrere Listenelemente gleichzeitig auszuwählen, scheint wegen der Möglichkeit der rekursiven Indizierung versperrt zu sein. Da aber auch eine Liste intern durch einen Vektor repräsentiert wird, kann man auf mehrere Listenelemente gleichzeitig wie auf einen Vektor zugreifen, nämlich mit `[]`.

```
> L2[[2]][c(1, 3)]      # L2[[2]] ist identisch zu L1, man erhält
 [[1]]                  #  also die Elemente 1 und 3 der Liste L1.
 [1] 1 2 5 4

 [[2]]
 [1] "Hallo" "Welt"
```

Achtung: Da ein Vektor in allen Elementen (der Länge 1!) den gleichen Datentyp haben muss, enthält dieser Vektor entsprechend Elemente des Typs *list*. Also wird in jedem Fall eine Liste zurückgeliefert – bei einelementiger Indizierung auch eine einelementige Liste.

2.9.5 Datensätze – *data frames*

Bei *data frames* (Datensätze oder auch Datentabellen), die z.B. mit der Funktion `data.frame()` erzeugt werden können, handelt es sich um spezielle *Listen* (Abschn. 2.9.4). Diese Listen haben die Einschränkung, dass die einzelnen Elemente nur Vektoren gleicher Länge sein dürfen.

Bei *data frames* handelt es sich um *die* typische Datenstruktur für Datensätze in R! Sehr viele Funktionen erwarten diese Datenstruktur als Argument.

Als Beispiel konstruieren wir eine Einkaufsliste, auf der neben dem Produkt und der Menge auch die Abteilung eingetragen wird, in der wir das Produkt erwarten, z.B. damit wir nicht allzu wirr durch das Geschäft laufen:

```
> Einkaufen <- data.frame(Produkt = c("Apfelsaft", "Quark",
+         "Joghurt", "Schinken", "Wasser", "Wurst", "Bier"),
+     Abteilung = c("Getränke", "Milchprod.", "Milchprod.",
+         "Fleischw.", "Getränke", "Fleischw.", "Getränke"),
+     Menge = c(4, 2, 2, 1, 3, 1, 2))

> Einkaufen                     # so wird es übersichtlicher:
     Produkt  Abteilung Menge
 1 Apfelsaft   Getränke     4
 2     Quark Milchprod.     2
 3   Joghurt Milchprod.     2
 . . . .                       # usw.
```

```
> str(Einkaufen)      # Struktur des data frame:
 'data.frame':   7 obs. of  3 variables:
  $ Produkt  : Factor w/ 7 levels "Apfelsaft","Bier",..: 1 4 3 ..
  $ Abteilung: Factor w/ 3 levels "Fleischw.","Get..",..: 2 3 3 ..
  $ Menge    : num  4 2 2 1 3 1 2
```

Die Struktur (mit `str()` erzeugt) unserer Einkaufsliste zeigt, dass die eingegebenen Zeichenketten durch `data.frame()` als Faktoren (Abschn. 2.8) interpretiert werden. Für die Variable „Abteilung“ ist das sicherlich in unserem Fall auch wünschenswert. Wenn man dieses Verhalten jedoch nicht wünscht, muss es explizit angegeben werden.

Die Indizierung kann sowohl wie bei *Listen* als auch wie bei *Matrizen* (Abschn. 2.9.2) erfolgen:

```
> Einkaufen$Menge[2]  # ... ist das Gleiche wie Einkaufen[[3]][2]
 [1] 2
> Einkaufen[2,3]      # und nochmal dasselbe Element
 [1] 2
```

Einhängen eines Datensatzes in den Suchpfad

Die Funktion `attach()` erlaubt es, einen Datensatz in den Suchpfad (s. Abschn. 4.3) einzuhängen. Damit kann man dann auf alle Variablen des Datensatzes direkt zugreifen, ohne den Namen des Datensatzes angeben zu müssen. Anstelle von

```
> Einkaufen$Menge[2]
```

kann man also schreiben:

```
> attach(Einkaufen)
> Menge[2]
 [1] 2
```

Wenn man jedoch jetzt ein Objekt ändert, so muss es im Datensatz direkt geändert werden, sonst ändert man nämlich nur eine Kopie im Workspace (s. auch Abschn. 4.3) unter demselben Namen:

```
> Menge[2] <- 7
> Menge[2]
 [1] 7
> Einkaufen$Menge[2]    # Achtung, es hat sich nicht geändert
 [1] 2
> detach(Einkaufen)
```

Mit `detach()` wird der Datensatz wieder aus dem Suchpfad entfernt.

Wird komfortabler Zugriff auf mehrere Objekte eines Datensatzes oder einer Liste in einer oder wenigen Zeilen benötigt, so empfiehlt es sich, die

Funktion `with()` zu benutzen. Sie wertet einen als zweites Argument angegebenen Ausdruck in einer eigens aus den angegebenen Daten konstruierten Umgebung (s. Abschn. 4.3) aus:

```
> with(Einkaufen, rep(Produkt, Menge))
  [1] Apfelsaft Apfelsaft Apfelsaft Apfelsaft Quark     Quark
  [7] Joghurt   Joghurt   Schinken  Wasser    Wasser    Wasser
 [13] Wurst     Bier      Bier
Levels: Apfelsaft Bier Joghurt Quark Schinken Wasser Wurst
```

Teilmengen mit subset(), %in% und split()

Um mit Teilmengen von Daten eines *data frame* zu arbeiten, möchte man oft bestimmte Zeilen extrahieren, meist abhängig von Werten gewisser Variablen. Eine einfache und bereits prinzipiell bekannte Möglichkeit dazu ist die Indizierung. Die Abfrage

```
> Einkaufen[Einkaufen[["Abteilung"]] == "Fleischw.", ]
    Produkt Abteilung Menge
4 Schinken Fleischw.      1
6    Wurst Fleischw.      1
```

zeigt alle Zeilen des Datensatzes an, in denen die Variable „Abteilung“ den Wert „Fleischw.“ hat.

Eine mächtigere, und gerade bei mehreren Abhängigkeiten übersichtlichere, Methode bietet die Funktion `subset()`, die mit folgendem Aufruf zu demselben Ergebnis kommt:

```
> subset(Einkaufen, Abteilung == "Fleischw.")
```

Da wir bereits wissen, dass nur „Fleischwaren“ ausgegeben werden, kann beispielsweise die zweite Spalte des Datensatzes auch durch das `select` Argument ausgeschlossen werden:

```
> subset(Einkaufen, Abteilung == "Fleischw.", select = -2)
    Produkt Menge
4 Schinken      1
6    Wurst      1
```

Wenn man dazu noch den Operator `%in%` verwendet, hat man sehr komplexe Auswahlmöglichkeiten in *data frames*. Dieser Operator liefert für `A %in% B` für jedes Element von `A` den Wahrheitswert für die Behauptung, dass das jeweilige Element in der Menge `B` enthalten sei.

Alle zu erledigenden Einkäufe in der Menge der Abteilungen „Getränke“ und „Milchprodukte“ werden dann ausgegeben mit

```
subset(Einkaufen, Abteilung %in% c("Getränke", "Milchprod."))
```

Auch die SQL (s. auch Abschn. 3.5) Anhänger werden von der Flexibilität von `subset()` begeistert sein. Eine Frage auf der Mailingliste *R-help* lautete, ob es für einen *data frame* `d` etwas wie die folgende in Pseudo-R-Syntax übertragene SQL Anfrage gebe

```
maleOver40 <- select.data.frame(d, "(sex=m or sex=M) and age > 40"),
```

ohne dass über die Zeilen des Datensatzes iteriert werden müsse. Die Antwort von Peter Dalgaard bestand aus der Zeile:

```
maleOver40 <- subset(d, sex %in% c("m", "M") & age > 40)
```

und verdeutlicht die Einfachheit der Umsetzung.

Wenn aus einem Datensatz mehrere Datensätze entstehen sollen, und zwar für jede mögliche Ausprägung einer bestimmten Variablen ein eigener Datensatz, wird die Funktion `split()` verwendet. Die Variable, nach der der Datensatz aufgeteilt werden soll, ist idealerweise ein Faktor, wie in unserem Beispiel die „Abteilung“:

```
> split(Einkaufen, Einkaufen$Abteilung)
 $Fleischw.
    Produkt Abteilung Menge
 4 Schinken Fleischw.     1
 6    Wurst Fleischw.     1

 $"Getränke"
 . . . .            # usw.

 $Milchprod.
 . . . .            # usw.
```

Zusammenfügen mit merge()

Auch das Zusammenfügen mehrerer Datensätze zu einem einzigen kann benötigt werden. Zum einen ist es möglich, Datensätze mit `rbind()` (für zeilenweises Zusammenfügen) „untereinander zu hängen“, zum anderen ist es möglich, bei zusätzlichen Informationen zu einzelnen Beobachtungen die *data frames* beobachtungsweise mit `merge()` zusammenzufügen.

Das Beispiel zum Einkauf von Lebensmitteln lässt sich wie folgt fortsetzen. Nehmen wir an, dass wir einige Lebensmittel nur von einer bestimmten Marke kaufen möchten. Diese Information ist bekannt und müsste nicht jedes Mal neu aufgeschrieben werden, so dass eine ständige Liste mit Marken und die aktuelle Einkaufsliste zusammengefügt werden können:

```
> Markenzuordnung <-
+     data.frame(Produkt = c("Quark", "Joghurt", "Wasser", "Limo"),
+         Marke = c("R-Milch", "R-Milch", "R-Wasser", "R-Wasser"))
```

```
> merge(Einkaufen, Markenzuordnung, all.x = TRUE)
     Produkt  Abteilung Menge    Marke
 1 Apfelsaft   Getränke     4     <NA>
 2      Bier   Getränke     2     <NA>
 3   Joghurt Milchprod.     2  R-Milch
 4     Quark Milchprod.     2  R-Milch
 5  Schinken  Fleischw.     1     <NA>
 6    Wasser   Getränke     3 R-Wasser
 7     Wurst  Fleischw.     1     <NA>
```

Man sieht, dass die „Marke“ passend angefügt wird. Wo keine passende Eintragung zu finden ist, wird konsequenterweise ein fehlender Wert gesetzt.

Ein detailliertes Beispiel, das sich zur Nacharbeitung empfiehlt, gibt es auf der Hilfeseite `?merge`.

Einfaches zeilen- oder spaltenweises Zusammenfügen zweier Datensätze geschieht mit den Funktionen `rbind()` und `cbind()`.

2.9.6 Objekte für formale S4 Klassen

Objekte, die S4 Klassen angehören, haben eine ganz eigene Datenstruktur. Sowohl S4 Klassen, die durch das Paket **methods** bereitgestellt werden, als auch die zugehörige Datenstruktur für entsprechende Objekte werden in Abschn. 6.2 detailliert besprochen. Da man jedoch auf ein einer S4 Klasse angehörendes Objekt treffen kann, ohne wissentlich mit S4 Klassen zu arbeiten, z.B. als Ergebnis einer auf S4 Standards basierenden Funktion, soll hier der Umgang und insbesondere der Zugriff auf solche Objekte kurz vorgestellt werden.

Zunächst reicht es aus, sich ein solches S4 Objekt als eine Liste vorzustellen, denn es hat sehr ähnliche Eigenschaften. Genau wie Listen als Elemente beliebige andere Objekte enthalten, haben S4 Klassen sogenannte *Slots*, die beliebige Objekte enthalten, meist also Vektoren, Matrizen, Dataframes, Listen usw. Der Zugriff erfolgt im Unterschied zu Listen mit Hilfe des `@`-Operators oder der mächtigeren und mehr Schreibarbeit erfordernden Funktion `slot()`.

Die Details zur Erzeugung einer S4 Klasse im folgenden Beispiel werden in Abschn. 6.2 erläutert. Hier soll zunächst nur der Zugriff auf diese Objekte in den letzten Zeilen des Beispiels interessieren.

```
> setClass("neu", representation(x = "numeric", y = "numeric"))
 [1] "neu"
> (n1 <- new("neu", x = c(5, 7), y = 1:10))
 An object of class "neu"
 Slot "x":
  [1] 5 7
```

```
Slot "y":
 [1]  1  2  3  4  5  6  7  8  9 10

> n1[1]; n1$x; n1[[1]]      # Kein "herkömmlicher" Zugriff klappt
 [[1]]
 NULL

 NULL
 Error in n1[[1]] : subscript out of bounds
> n1@x                      # der @-Operator hilft
 [1] 5 7
```

Einer der Unterschiede von diesen Objekten zu Listen ist, dass jedes Objekt einer S4 Klasse dieselbe Struktur hat, d.h. dieselben vordefinierten *Slots*, die wieder dieselben Datenstrukturen und -typen besitzen. Mehr zu S4 Klassen gibt es in Abschn. 6.2.

2.10 Konstrukte

Eine moderne Programmiersprache kommt nicht ohne Konstrukte für die Steuerung des Programmablaufs aus. Schleifen werden für die Wiederholung gleichartiger Abläufe, z.B. in iterativen Algorithmen oder in Simulationen, benötigt, während der Einsatz bedingter Anweisungen zur Abbildung von Fallunterscheidungen erforderlich ist.

2.10.1 Bedingte Anweisungen

Bedingte Anweisungen werden zur Abbildung von Fallunterscheidungen in Algorithmen verwendet. Dabei sind nicht nur Fallunterscheidungen im mathematischen Sinne gemeint, sondern auch Abbruchbedingungen in Schleifen (s. Abschn. 2.10.2). Ebenso gibt es häufig Fallunterscheidungen zur Überprüfung auf zulässige Benutzereingaben. Dabei wird abhängig von einer Bedingung, die meist zu einem logischen Wert ausgewertet wird, ein jeweils entsprechendes Programmsegment ausgeführt.

Für bedingte Anweisungen gibt es in R im Wesentlichen die zwei Konstrukte

```
(1)    if(Bedingung){Ausdruck1} else{Ausdruck2}
(2)    ifelse(Bedingung, Ausdruck1, Ausdruck2)
```

und für eine etwas andere Art der Fallunterscheidung die Funktion `switch()` (s.u.).

Zunächst wird die *Bedingung* ausgewertet, bei der es sich um einen gültigen Ausdruck handeln muss. Ist diese wahr (`TRUE`), so wird *Ausdruck1* ausgewertet, sonst (`FALSE`) wird *Ausdruck2* ausgewertet. Der jeweils andere Ausdruck bleibt unberücksichtigt.

Die geschweiften Klammern (`{}`) werden benutzt, um mehrere Ausdrücke syntaktisch zusammenzufassen. Das ist nützlich, damit der so zusammengefasste Ausdruck mehrere Zeilen in Anspruch nehmen kann, denn ein Zeilenumbruch ist sonst Zeichen für den Beginn eines neuen Ausdrucks.

if ... else

Mit `if(){} else{}` (1) steht eine bedingte Anweisung für recht komplexe Ausdrücke zur Verfügung, wobei die *Bedingung* im Gegensatz zu `ifelse()` aber nicht vektorwertig sein darf. Vielmehr wird nur das erste Element im Fall einer vektorwertigen *Bedingung* verwendet (mit Warnung). Der Teil `else{Ausdruck2}` darf auch weggelassen werden, wenn kein Ausdruck für den entsprechenden Fall benötigt wird.

Im folgenden Beispiel ist der erste Ausdruck zwei Zeilen lang und wird mit geschweiften Klammern zusammengefasst. Da der zweite Ausdruck (im `else` Teil) nur über eine Zeile geht, sind dort keine geschweiften Klammern nötig:

```
> x <- 5
> if(x == 5){          # falls x == 5 ist:
+     x <- x + 1       #   x um 1 erhöhen und
+     y <- 3           #   y auf 3 setzen
+ } else               # sonst:
+     y <- 7           #   y auf 7 setzen
> c(x = x, y = y)     # x wurde um 1 erhöht und y ist gleich 3
 x y
 6 3

> if(x < 99) print("x ist kleiner als 99")
 [1] "x ist kleiner als 99"
```

Zum Auskommentieren längerer (aber notwendigerweise syntaktisch korrekter) Programmteile eignet sich `if(FALSE){Programmteil}`, da der Programmteil wegen der immer falschen Bedingung niemals ausgeführt werden kann. Hier möchte man häufig nicht vor sehr viele Zeilen Programmcode das Kommentarzeichen `#` setzen müssen.

ifelse

Die Funktion `ifelse()` (2) zeichnet sich durch vektorwertige Operationen aus, d.h. neben einer vektorwertigen *Bedingung* sind auch vektorwertige Ausdrücke

möglich, in dem Fall sind es häufig einfache Vektoren. Allerdings sind durch die Notation in runden Klammern komplexe Ausdrücke nicht besonders einfach zu spezifizieren, so dass ein Umweg, z.B. über lokal definierte Funktionen, nötig ist.

Der aktuelle Wert von `x` aus dem letzten Beispiel ist 6, so dass der Ausdruck

```
> ifelse(x == c(5, 6), c("A1", "A2"), c("A3", "A4"))
 [1] "A3" "A2"
```

erwartungsgemäß komponentenweise ausgewertet wird:
In der ersten Komponente gilt, dass `x` gleich 5 *falsch* ist, daher wird *Ausdruck2*, also `c("A3", "A4")`, in der ersten Komponente (`"A3"`) zurückgegeben.
In Komponente zwei gilt, dass `x` gleich 6 *wahr* ist, daher wird *Ausdruck1*, also `c("A1", "A2")`, in der zweiten Komponente (`"A2"`) zurückgegeben.

switch

Eine andere Art der Fallunterscheidung wird von `switch(EXPR, ...)` bereitgestellt. Diese Funktion bietet sich an, wenn eine ganze Reihe von möglichen Fällen abgeprüft werden muss, von denen eine große Anzahl möglicher Ausgaben abhängig ist. Die Funktionsweise wird am Beispiel schnell deutlich:

```
> switch(2, a=11, b=12, cc=13, d=14)    # Gibt das 2. Objekt in
 [1] 12                                 #   "..." aus
> switch("c", a=11, b=12, cc=13, d=14)  # Kein Objektname passt
 NULL
> switch("cc", a=11, b=12, cc=13, d=14) # Gibt Objekt "cc" aus
 [1] 13
```

Als `EXPR` kann ein numerischer Ausdruck angegeben werden, der spezifiziert, das wievielte Objekt aus dem „`...`“-Teil ausgegeben werden soll.

Alternativ kann das auszugebende Objekt auch benannt werden. Wenn die Objekte per Namen angesprochen werden, so wird in dem Fall, dass kein Name passt, `NULL` zurückgegeben. Sollte ein letztes unbenanntes Objekt existieren, so wird dieses ausgegeben.

Wie bei allen Funktionen gilt auch hier, dass die Hilfeseite (`?switch`) mehr Einzelheiten verrät.

2.10.2 Schleifen

Schleifen sind unverzichtbar, um eine größere Anzahl sich wiederholender Befehle aufzurufen. Das ist beispielsweise bei Wiederholung von Programmen mit unterschiedlichen Parametern oder Startwerten nötig, vor allem aber auch

bei iterativen Algorithmen, in denen Eingabewerte eines späteren Iterationsschritts von einem vorherigen abhängig sind. Bei Simulationen kommen solche Schleifen sehr häufig zum Einsatz, weil gewisse Befehlsfolgen und Funktionen immer wieder mit unterschiedlichen Zufallszahlen gestartet werden müssen.

In R gibt es drei Varianten von Schleifen sowie zwei wesentliche Kontrollbefehle, die in Tabelle 2.5 zusammengefasst sind und hier detailliert vorgestellt werden. Am häufigsten wird die `for`-Schleife (s. S. 53) benutzt.

Tabelle 2.5. Schleifen und zugehörige Kontrollbefehle

Schleife bzw. Kontrollwort	Beschreibung
`repeat{Ausdruck}`	Wiederholung des *Ausdrucks*
`while(Bedingung){Ausdruck}`	Wiederholung, solange *Bedingung* erfüllt
`for(i in M){Ausdruck}`	Wiederhole *Ausdruck* für jedes $i \in M$
`next`	Sprung in den nächsten Iterationsschritt
`break`	Sofortiges Verlassen der Schleife

repeat, next und break

Die zunächst einfachste Schleifenkonstruktion wird mit `repeat{Ausdruck}` bewerkstelligt, wobei der *Ausdruck* immer wieder wiederholt wird. Wie in Abschn. 2.10.1 werden auch hier geschweifte Klammern (`{}`) benutzt, um mehrere Ausdrücke syntaktisch zusammenzufassen.

Die Schleife wird endlos laufen, solange sie nicht mit dem Kontrollwort `break` beendet wird. Natürlich wird die Schleife andererseits durch ein einfach eingefügtes `break` sofort im ersten Durchlauf abbrechen. Hier macht es also Sinn eine bedingte Anweisung (s. Abschn. 2.10.1) einzubauen, die das `break` nur unter bestimmten Bedingungen zum Abbruch der Schleife veranlasst. Da eine vektorwertige Bedingung keinen Sinn macht, sieht man also in `repeat` typischerweise eine Zeile wie `if(Bedingung) break`.

Alle folgenden Beispiele zum Thema „Schleifen“ sind ausschließlich als Demonstration der Funktionen, nicht aber als guter Programmierstil oder gar effizient (s. Kap. 5) anzusehen.

```
> i <- 0
> repeat{
+   i <- i + 1              # addiere 1 zu i
+   if(i == 3) break        # stoppe, falls i = 3 ist
+ }
> i
 [1] 3
```

Hier wurde zunächst `i` auf 0 gesetzt. Die Schleife startet, zu `i` wird 1 addiert, und dessen Summe wird wieder dem Objekt `i` zugewiesen. Als Nächstes wird die Bedingung `(i == 3)` überprüft. Weil `i` im ersten Durchlauf noch 1 ist, wird das `break` nicht ausgeführt und der zweite Schleifendurchlauf beginnt, indem der Code innerhalb der Schleife wieder von vorne abgearbeitet wird. Im dritten Durchlauf ist `i` gleich 3, `break` wird ausgeführt und damit die Schleife verlassen.

Das Kontrollwort `next` bewirkt, dass der aktuelle Schleifendurchlauf abgebrochen und zum Anfang der Schleife in den nächsten Durchlauf gesprungen wird. Dadurch wird in

```
> i <- 0
> repeat{
+   i <- i + 1          # addiere 1 zu i
+   if(i < 3) next      # springe zum Anfang, falls i < 3
+   print(i)            # gibt aktuelles i aus
+   if(i == 3) break    # stoppe, falls i = 3 ist
+ }
 [1] 3
```

im ersten und zweiten Durchlauf der Schleife bereits von der Zeile

```
if(i < 3) next
```

zum Anfang der Schleife für den nächsten Durchlauf zurückgesprungen. Im dritten Durchlauf wird weitergearbeitet, so dass erst hier `print(i)` wirksam wird. Es ist nur eine Ausgabe zu sehen, weil die Schleife direkt danach abgebrochen wird.

Zur Wiederholung identischer Abläufe in Simulationen bietet sich auch die in Abschn. 5.2.2 auf S. 107 beschriebene Funktion `replicate()` an.

while

Anstatt eine `repeat`-Schleife zu verwenden, die eine Abbruchbedingung direkt an ihrem Anfang oder Ende hat, kann man auch (einfacher) die Konstruktion mit `while` (s. Tabelle 2.5) verwenden.

Die Schleife aus dem ersten Beispiel kann ersetzt werden durch:

```
> i <- 0
> while(i < 3)
+   i <- i + 1      # Solange i < 3 ist, erhöhe i um 1
> i
 [1] 3
```

Solange die *Bedingung* (hier „`i < 3`“) am Anfang der Schleife erfüllt ist, wird der *Ausdruck* (hier nur aus der Zeile „`i <- i + 1`“ bestehend) immer wieder ausgewertet. Ist die *Bedingung* nicht (mehr) erfüllt, wird die Schleife verlassen.

Die Kontrollwörter `next` und `break` sind in `while`-Schleifen verwendbar.

for

Mit der Schleife `for(i in M){Ausdruck}` nimmt `i` zunächst das erste Element von `M` an und der *Ausdruck* wird mit diesem Wert von `i` ausgeführt, wobei `i` darin nicht notwendigerweise verwendet werden muss. Als Nächstes nimmt `i` das zweite Element von `M` an usw.:

```
> x <- c(3, 6, 4, 8, 0) # Vektor der Länge 5 (=length(x))
> for(i in x)           # i nimmt nacheinander die Werte von x an
+     print(i^2)        # Ausgabe auf Konsole
 [1] 9
 [1] 36
 [1] 16
 [1] 64
 [1] 0
```

Es entsteht häufig der Wunsch, einen Laufindex in der Schleife zu verwenden, z.B. zur Indizierung von Objekten. Das letzte Beispiel, modifiziert unter Verwendung eines Laufindex, sieht dann wie folgt aus:

```
> for(i in seq(along = x))
+     print(x[i]^2)     # für alle i im Vektor seq(along=x)
 [1] 9
 . . . .               # usw.
```

Beide Beispiele können natürlich einfacher, übersichtlicher und deutlich schneller durch ein vektorwertiges `x^2` ersetzt werden!

Im zweiten Beispiel wurde ein Indexvektor zum Vektor `x` mit Hilfe von `seq(along = x)` (s. Abschn. 2.9.1) erzeugt. Häufig sieht man stattdessen die *gefährliche* Verwendung von `1:length(x)`. Die Gefahr liegt darin, dass ein Objekt `x` der Länge 0 existieren könnte. In dem Fall sollte offensichtlich die Schleife nicht ausgeführt werden, was bei Verwendung von `seq(along = x)` auch nicht eintritt, `1:length(x)` liefert jedoch unerwünscht einen Vektor der Länge 2 (`c(1, 0)`).

Schleifen vermeiden

Bei R handelt es sich um eine interpretierte Sprache, d.h. jeder Befehl wird erst zur Laufzeit interpretiert und nicht wie bei kompilierten Sprachen beim Vorgang des Übersetzens. In Schleifen muss jede Zeile in jedem Durchlauf erneut interpretiert werden. Daher können Schleifen im Vergleich zu vektorwertigen Operationen, in denen diese Zeilen nur einmal interpretiert werden müssen, deutlich langsamer sein. Auch die Anzahl von Transfers von Objekten wird deutlich durch Einsatz vektorwertiger Operationen reduziert.

Wo eine (z.B. vektorwertige) Alternative offensichtlich ist, sollte daher möglichst auf Schleifen verzichtet werden. Das ist eine der grundsätzlichen

Regeln für effizientes Programmieren mit R, die in Kap. 5 und insbesondere Abschn. 5.2 genauer beschrieben werden. Dort wird auch auf die für vektorwertiges Programmieren wesentlichen Funktionen `apply()`, `lapply()` usw. eingegangen.

2.11 Zeichenketten

In R sind mächtige Werkzeuge zur Manipulation von Zeichenketten vorhanden, wie sie in Objekten vom Datentyp (s. Abschn. 2.8) *character* vorkommen.

Solche Möglichkeiten überraschen zunächst, wenn man R als eine Sprache für Statistik betrachtet, sie werden jedoch in den verschiedensten Situationen immer wieder benötigt. Der Anwender soll schließlich einfach zum Programmierer werden können, also kein anderes Werkzeug benutzen müssen, und R bietet auch für Zeichenketten mehr als nur das Nötigste. Eine Zusammenfassung einiger wesentlicher Funktionen wird in Tabelle 2.6 gegeben.

Tabelle 2.6. Funktionen zum Umgang mit Zeichenketten

Funktion	Beschreibung
`cat()`	Ausgabe in Konsole und Dateien
`deparse()`	*expression* in Zeichenfolge konvertieren
`formatC()`	sehr allgemeine Formatierungsmöglichkeiten
`grep()`	Zeichenfolgen in Vektoren suchen
`match(), pmatch()`	Suchen von (Teil-)Zeichenketten in anderen
`nchar()`	Anzahl Zeichen in einer Zeichenkette
`parse()`	Konvertierung in eine *expression*
`paste()`	Zusammensetzen von Zeichenketten
`strsplit()`	Zerlegen von Zeichenketten
`sub(), gsub()`	Ersetzen von Teil-Zeichenfolgen
`substring()`	Ausgabe und Ersetzung von Teilzeichenfolgen
`toupper(), tolower()`	Umwandlung in Groß- bzw. Kleinbuchstaben

Die Repräsentation eines Objekts als Zeichenkette(n) haben wir bisher bereits häufig gesehen. So wird bei einfacher Eingabe eines Objektnamens in der Konsole und damit implizites (oder explizites) Aufrufen der Funktion `print()` das Objekt in adäquat repräsentierende Zeichenketten umgewandelt und als solche ausgegeben.

Eine andere Möglichkeit zur Ausgabe von Zeichenketten auf die Konsole oder in Dateien bietet `cat()`. Diese Funktion konvertiert alle durch Kommata getrennt eingegebenen Objekte in Zeichenketten, verknüpft diese mit dem im

Argument `sep` angegebenen Zeichen (Voreinstellung: Leerzeichen) und gibt das Ergebnis aus. Dieses Vorgehen ist typisch zur Ausgabe von Informationen in selbst geschriebenen Funktionen. Als Verknüpfungselement der einzelnen Objekte eignet sich oft auch ein Tabulator, den man mit `"\t"` erzeugt. Nach einem Aufruf von `cat()` wird kein Zeilenumbruch erzeugt, so dass sich am Ende einer Ausgabe das explizite Erzeugen eines Zeilenumbruchs mit `"\n"` anbietet:

```
> x <- 8.25
> cat("Das Objekt x hat den Wert:", x, "\n", sep = "\t")
 Das Objekt x hat den Wert:      8.25
```

Die Syntax der Funktion `paste()` ist dazu analog, wobei sie aber dem Zusammenfügen zu einem neuen Objekt dient, nicht aber der Ausgabe.

Manchmal möchte man Zeichenketten auch anders verarbeiten, etwa zusammensetzen, zerlegen, darin suchen oder diese für die Ausgabe angemessen formatieren. Typische Anwendungen von Zeichenkettenoperationen sind:

- Zusammensetzen von Buchstabenkombinationen und Nummern, etwa um eine Liste durchnummerierter Dateien nacheinander automatisch bearbeiten zu können:

  ```
  > paste("Datei", 1:3, ".txt", sep = "")
   [1] "Datei1.txt" "Datei2.txt" "Datei3.txt"
  ```

- Das Zerlegen von Zeichenketten, die durch Sonderzeichen (im Beispiel: Leerzeichen) voneinander getrennt sind, in einzelne Teile:

  ```
  > x <- "Hermann Müller"
  > strsplit(x, " ")
   [[1]]
   [1] "Hermann" "Müller"
  ```

- Suchen, ob in einer Liste von Zeichenfolgen bestimmte Zeichenkombinationen vorhanden sind.
- Benutzerfreundliche, übersichtliche Ausgabe von Informationen in tabellarischer Form.

In Tabelle 2.6 ist eine Zusammenstellung einiger sehr nützlicher Funktionen für solche Anwendungen gegeben. Weitere Hinweise erhält man durch Konsultation der entsprechenden Hilfeseiten, die auf andere Funktionen hinweisen, falls der Funktionsumfang nicht ganz dem erwünschten entspricht. Exemplarisch werden hier einige weitere Beispiele für die erwähnten Anwendungen angegeben:

```
> x <- "Hermann Müller"
> y <- "Hans_Meier"
```

```
> grep("Hans", c(x, y))              # "Hans" ist im 2. Element
 [1] 2
> sub("ü", "ue", c(x, y))            # Ersetze "ü" durch "ue"
 [1] "Hermann Mueller" "Hans_Meier"
> nchar(x)                           # Wie viele Zeichen hat x?
 [1] 14
> toupper(x)                         # Bitte alles in Großbuchstaben
 [1] "HERMANN MÜLLER"
> (ep <- parse(text = "z <- 5"))     # Zeichenfolge -> expression
 expression(z <- 5)
> eval(ep)                           # ep kann man jetzt auswerten
```

Insbesondere die letzten beiden Schritte des Beispiels sind interessant, denn so lässt sich eine zusammengesetzte Zeichenkette in einen für R auswertbaren Ausdruck (*expression*) konvertieren. Wie in Abschn. 4.6 beschrieben ist, erfolgt die Auswertung eines solchen Ausdrucks mit `eval()`.

Anführungszeichen

Wie selbstverständlich sind bisher doppelte Anführungszeichen (") benutzt worden, um Zeichenketten zu spezifizieren. Auch R selbst gibt Zeichenketten mit doppelten Anführungszeichen umschlossen aus. Alternativ können Zeichenketten aber auch von einfachen Anführungszeichen (') umschlossen werden. Das macht vor allem dann Sinn, wenn Zeichenketten selbst Anführungszeichen enthalten sollen.

Eine von einfachen Anführungszeichen umschlossene Zeichenkette kann doppelte Anführungszeichen enthalten und umgekehrt. Sollen in einer von doppelten Anführungszeichen umschlossenen Zeichenkette weitere doppelte Anführungszeichen enthalten sein, so ist diesen ein Backslash (\) voranzustellen. Analoges gilt für einfache Anführungszeichen. Hier werden vier Alternativen für Anführungszeichen in Zeichenketten angegeben:

```
> "Eine 'Zeichenkette' in einer Zeichenkette"
 [1] "Eine 'Zeichenkette' in einer Zeichenkette"
> "Eine \"Zeichenkette\" in einer Zeichenkette"
 [1] "Eine \"Zeichenkette\" in einer Zeichenkette"
> 'Eine "Zeichenkette" in einer Zeichenkette'
 [1] "Eine \"Zeichenkette\" in einer Zeichenkette"
> 'Eine \'Zeichenkette\' in einer Zeichenkette'
 [1] "Eine 'Zeichenkette' in einer Zeichenkette"
```

Die Nützlichkeit von Anführungszeichen in Zeichenketten zeigt sich z.B. bei folgender *SQL* Anfrage aus Abschn. 3.6, bei der ein irregulärer, durch Anführungszeichen zu umschließender Name in der *SQL* Zeichenkette enthalten ist:

```
> sqlQuery(channel, "select * from \"iris$\"")
```

Einfache rückwärts gerichtete Anführungszeichen (`), s.g. Backticks, können für den Zugriff auf Objekte mit nicht regulären Objektnamen benutzt werden (s. auch Abschn. 2.3). Dazu wird die Zeichenkette (der Objektname) wie in folgendem Beispiel in Backticks gesetzt:

```
> `1x / 5y` <- 3
> `1x / 5y` - 2
 [1] 1
```

Der Objektname besteht dann tatsächlich aus der ganzen Zeichenfolge

```
`1x / 5y`
```

Solche Objektnamen sollten vermieden werden!

2.12 Datum und Zeit

Die Darstellung von Datum und Uhrzeit mit Hilfe von Objekten kann für eine Reihe von Anwendungen interessant sein. Einen ersten Übersichtsartikel zu Klassen und Werkzeugen zur Erzeugung und Behandlung von Datum- und Zeitobjekten haben Ripley und Hornik (2001) verfasst. Insbesondere werden die *POSIX* Klassen *POSIXt*, *POSIXct* und *POSIXlt* in dem Artikel besprochen. Eine aktuellere Zusammenstellung geben Grothendieck und Petzoldt (2004), die neben den POSIX Klassen vor allem die Klassen *Date* und *chron* vergleichen. Hier erfolgt eine kurze Übersicht über diese Klassen.

Die Klasse *Date* repräsentiert das Datum ohne eine Zeitangabe, wobei die interne Repräsentation als Anzahl Tage seit dem 01.01.1970 erfolgt.

Das Paket **chron** (James und Pregibon, 1993) stellt die gleichnamige Klasse *chron* und zugehörige Funktionen zur Verfügung. Objekte der Klasse *chron* repräsentieren Datum und Uhrzeit, können allerdings keine Informationen zu Zeitzonen beinhalten.

Die *POSIX* Klassen *POSIXt*, *POSIXct* und *POSIXlt* unterstützen neben Datum und Uhrzeit auch Zeitzonen und können zwischen Winter- und Sommerzeit unterscheiden. Das ist nicht nur von Belang für Dateisysteminformationen des Betriebssystems oder Synchronisation mit Datenbanken, sondern z.B. auch im Bereich der Datenanalyse in Finanzmärkten. Intern wird die Zeit in Anzahl an Sekunden seit dem 01.01.1970 repräsentiert. Die große Mächtigkeit von Klassen kann gerade dann von Nachteil sein, wenn keine Informationen zu Zeitzonen oder Sommer- und Winterzeit benötigt werden. Diese Informationen können wegen verschiedener Zeiteinstellungen auf verschiedenen Betriebssystemen bzw. an verschiedenen Orten zu Fehlern führen, wenn Benutzer nicht auf korrekte und allgemeine Spezifikationen achten. Die-

ses Problem wird auch von Grothendieck und Petzoldt (2004) diskutiert. Details zu *POSIX* Klassen geben der oben erwähnte Artikel und die Hilfeseite `?DateTimeClasses`, die auf eine Reihe nützlicher Funktionen verweist.

Eine der wichtigsten Funktionen in diesem Zusammenhang ist sicherlich `strptime()`, die eine Zeichenfolge in ein Objekt der Klasse *POSIXlt* konvertiert. Das folgende leicht erweiterte Beispiel von der entsprechenden Hilfeseite, deren Lektüre sehr zu empfehlen ist, zeigt die Funktionsweise:

```
> dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92")
> times <- c("23:03:20", "22:29:56", "01:03:30", "18:21:03")
> x <- paste(dates, times)
> (z <- strptime(x, "%m/%d/%y %H:%M:%S"))
 [1] "1992-02-27 23:03:20" "1992-02-27 22:29:56"
 [3] "1992-01-14 01:03:30" "1992-02-28 18:21:03"
```

Während in dem ersten Argument der Funktion `strptime()` die zu konvertierende Zeichenkette angegeben wird, enthält das zweite Argument eine Angabe darüber, wie das Format der Spezifikation von Datum und Uhrzeit zu interpretieren ist. Es ist also der Monat (`%m`) durch einen Schrägstrich (`/`) jeweils von Tag (`%d`) und Jahr (`%y`) getrennt. Nach einem Leerzeichen folgen dann durch Doppelpunkte (`:`) getrennt Stunde (`%H`), Minute (`%M`) und Sekunde (`%S`).

Das Rechnen mit *POSIXlt* Objekten ist leicht möglich, da entsprechende Methoden existieren. Zeitdifferenzen lassen sich beispielsweise wie folgt berechnen:

```
> z[1] - z[2]
 Time difference of 33.4 mins
```

Es sei bemerkt, dass 33.4 Minuten gerade 33 Minuten und 24 Sekunden entsprechen.

Auch eine Grafik kann einfach erzeugt werden. Beispielsweise lassen sich die Werte `1:4` einfach gegen die Daten aus `z` abtragen mit `plot(z, 1:4)`.

Zuletzt wird mit Hilfe der Funktion `as.Date()` das Objekt der Klasse *POSIXlt* in ein Objekt der Klasse *Date* konvertiert:

```
> as.Date(z)
 [1] "1992-02-27" "1992-02-27" "1992-01-14" "1992-02-28"
```

3

Ein- und Ausgabe von Daten

Wer R als Werkzeug für Datenanalyse einsetzen will, muss diese Daten zunächst einmal einlesen und vermutlich andere Daten wieder ausgeben. In der Praxis wird man mit Datensätzen konfrontiert, die in den verschiedensten Formaten vorliegen. R bietet die Möglichkeiten für den Zugriff auf sehr viele Formate, wie z.B. Textdateien (im ASCII Format, Abschn. 3.1), Binärdateien (Abschn. 3.2), R Dateien (Abschn. 3.3) und Datenbanken (Abschn. 3.5), aber auch die Möglichkeit, eigene Import- und Exportfunktionen zu schreiben (Abschn. 3.4).

In der Praxis begegnet man recht häufig Dateien, die im Format des Produkts Excel der Firma Microsoft gespeichert sind. Zwar gibt es in R keine direkte Import- bzw. Exportfunktionen für dieses Format, dafür gibt es aber je nach Situation einige mehr oder weniger komfortable Umwege, die in Abschn. 3.6 beschrieben werden.

Wenn es Probleme beim Einlesen und Ausgeben von Daten gibt, hilft das Handbuch „R Data Import/Export“ (R Development Core Team, 2006b) weiter.

3.1 ASCII – Dateien

Das Austauschen von Dateien im ASCII Format ist sicherlich die einfachste Möglichkeit, relativ kleine Datenmengen von einem System in das andere zu transportieren, denn nahezu jedes Programm kann diese Art von Daten lesen.

Textdateien lesen

Die Funktion `read.table()` ist gedacht für das Einlesen eines Datensatzes (*data frame*), der in Tabellenform vorliegt. Hier einige wesentliche Argumente dieser Funktion:

- `file`: (Pfad und) Dateiname
- `header`: Sind Spaltennamen vorhanden? Voreinstellung: `FALSE`
- `sep`: Trennzeichen zwischen zwei Spalten. Nach Voreinstellung, `""`, wird jeglicher „Leerraum“, also Leerzeichen oder Tabulatoren, als Trennzeichen verwendet.
- `quote`: In der Datei verwendete Anführungszeichen – Voreinstellung: `"\"'"` (`'` und `"` werden als Anführungszeichen erkannt)
- `dec`: Dezimalzeichen, Voreinstellung: `"."`
- `colClasses`: Mit diesem Vektor kann angegeben werden, welche Datentypen (z.B. *character*, *numeric*, ...) die einzelnen Spalten des Datensatzes haben. Das geschieht entweder dann, wenn die automatische Bestimmung des Datentyps fehl schlägt, oder es geschieht zur Steigerung der Geschwindigkeit des Einlesens.

Informationen zu vielen weiteren mächtigen Argumenten findet man in der Hilfe (`?read.table`). Ein Beispiel zu den hier beschriebenen Funktionen folgt am Ende dieses Abschnitts.

Bei sehr großen Datensätzen empfiehlt es sich, das Argument `colClasses` zu setzen, so dass die Funktion nicht für jeden Tabelleneintrag die Konsistenz der Datentypen prüfen muss. Sollte `read.table()` trotzdem noch träge reagieren, kann das Einlesen durch Verwendung von `scan()` (s.u.) weiter beschleunigt werden, die Benutzung ist jedoch entsprechend weniger komfortabel.

Praktische Modifikationen der Voreinstellungen von `read.table()` für häufig vorkommende Formate bieten `read.csv()` (`header = TRUE, sep = ",", dec = "."`) und `read.csv2()` (`header = TRUE, sep = ";", dec = ","`). Letzteres als typische Voreinstellung für das im deutschsprachigen Raum recht gebräuchliche Format mit dem Komma als Dezimalzeichen und dem Semikolon als Spaltentrennzeichen.

Textdateien mit fester Spaltenbreite (*fixed width formatted*) ohne gesonderte Trennzeichen, wie sie z.B. bei SAS (SAS Institute Inc., 1999) Anwendern beliebt sind, kann die Funktion `read.fwf()` lesen. Hier müssen die Spaltenbreiten aller Variablen des Datensatzes angegeben werden.

Eine noch mächtigere Funktion zum Lesen von Textdateien ist `scan()`. Ihre Benutzung zum Einlesen einspaltiger Textdateien ist trivial. Aber es können mit etwas trickreich spezifizierten Argumenten auch Dateien mit sehr verschiedenen und komplizierten Datenstrukturen eingelesen werden. Ein Blick in die Hilfe (`?scan`) lohnt sich.

Zum Lesen von Textdateien unterschiedlich langer Zeilen eignet sich die Funktion `readLines()`. Die damit eingelesenen Zeilen können dann oft sehr

gut mit den in Abschn. 2.11 beschriebenen Funktionen zum Umgang mit Zeichenketten bearbeitet werden.

Textdateien schreiben

Analog zu den oben beschriebenen Funktionen zum Einlesen von Dateien im ASCII-Format gibt es auch solche zum Schreiben:

- `write.table()` eignet sich als Analogon zu `read.table()` für das Schreiben von Datensätzen und hat sehr ähnliche Argumente.
- `write()` kann als „Partner“ von `scan()` angesehen werden. Allerdings ist die Voreinstellung des Arguments `ncolumns` (Spaltenanzahl der Ausgabe), `if(is.character(x)) 1 else 5`, etwas überraschend, da man für einen numerischen Vektor `x` bei Aufruf von `write(x)` meist nicht eine 5-spaltige Ausgabe erwartet.
- Das Analogon zu `read.fwf()` zum Schreiben mit fester Spaltenbreite findet sich mit `write.matrix()` im Paket **MASS**.
- `sink()` gehört nicht unbedingt zu den Funktionen des Datenexports. Hiermit kann vielmehr die Ausgabe eines R Prozesses in eine Datei geschrieben werden.

Hier nun ein Beispiel, in dem die bereits in Abschn. 2.5 benutzten *iris* Daten verwendet werden, die hier zunächst exportiert und danach wieder eingelesen werden. Es ist sehr empfehlenswert, sich die im Beispiel erzeugten Dateien in einem beliebigen Editor anzusehen, denn so kann man besser ein „Gefühl“ für das Verhalten der Funktionen gewinnen.

```
> write.table(iris, file = "iris.txt")
> x <- read.table("iris.txt")
> write.table(iris, file = "iris2.txt", sep = "\t", dec = ",")
> x <- read.table("iris2.txt", sep = "\t", dec = ",")
```

Hier wurden keine vollständigen Pfade spezifiziert, sondern die Dateien wurden im aktuellen Arbeitsverzeichnis (s. Abschn. 2.6, auch für Dateinamen) angelegt bzw. daraus gelesen.

3.2 Binärdateien

Neben ASCII-Dateien gibt es eine ganze Reihe verschiedener und z.T. proprietärer Formate für Binärdateien, die von ebenso verschiedenen Softwareprodukten gelesen oder geschrieben werden. Importfunktionalität erscheint vor allem für Formate anderer Statistiksoftware-Produkte interessant. Excel ist der eigene Abschn. 3.6 gewidmet.

Das Paket **foreign** stellt Importfunktionen zum Lesen von Binärdateien folgender Formate bereit:

- Epi „Info Data“
- Minitab „Portable Worksheet“
- S3 (für „alte“ S-PLUS Dateien)
- SAS Datensätze (benötigt eine vorhandene SAS Installation und benutzt den Filter für SAS XPORT Dateien)
- SAS XPORT
- SPSS
- Stata

Zum Beispiel kann ein in einer Datei `SPSSdaten.sav` gespeicherter SPSS (SPSS Inc., 2005) Datensatz wie folgt als data frame eingelesen werden:

```
> library("foreign")              # Laden des Pakets
> daten <- read.spss("SPSSdaten.sav", to.data.frame = TRUE)
```

Für eine Reihe anderer Binärdateien gibt es auch Importfilter, die man im WWW und insbesondere auf CRAN suchen sollte. Als Beispiel sei das Paket **tuneR** genannt, das u.a. Funktionen für das Lesen und Schreiben von Wave-Dateien bietet.

Kann man keine Importfunktion finden, so ist es meist am einfachsten, aus der Originalsoftware in ein ASCII-Format zu exportieren und die in Abschn. 3.1 erwähnten Funktionen zu nutzen oder den Datentransfer über eine Datenbank (z.B. im Fall vieler zu transferierender Datensätze; s. Abschn. 3.5) abzuwickeln. Ist das nicht oder nur erschwert möglich, so besteht immer noch die in Abschn. 3.4 beschriebene Möglichkeit, eigene Importfunktionen zu schreiben.

3.3 R Objekte lesen und schreiben

In Abschn. 2.6 wurde bereits erwähnt, dass ein R *Workspace* mit Hilfe von `save.image()` gespeichert und mit `load()` wieder explizit geladen werden kann. Das Speichern geschieht dabei als Voreinstellung in einem Binärformat, um Platz zu sparen. Alternativ kann man den *Workspace* mit Hilfe des Arguments `ASCII` als ASCII Repräsentation speichern und mit `compress` weiter komprimieren. Die ASCII Repräsentation ist besonders dann geeignet, wenn der Austausch binärer Workspace-Dateien zwischen Betriebssystemen fehlschlägt.

Mit der Funktion `save()` lässt sich eine Auswahl von R Objekten speichern. Das Format ist analog zu dem von `save.image()` gespeicherten Format, d.h. es gibt die Auswahl zwischen binärer und ASCII Repräsentation sowie die Möglichkeit der Komprimierung. Ebenso lassen sich mit `save()` erzeugte Dateien mit `load()` lesen, so dass die Objekte dem aktuellen *Workspace* hinzugefügt werden.

Weitere Möglichkeiten zum Austausch von Objekten zwischen verschiedenen R Versionen oder zwischen unterschiedlichen Betriebssystemen bieten die Paare

- `dump()` zum Export und `source()` zum Einlesen eines R Objekts als ASCII Repräsentation sowie
- `dput()` und `dget()` (hier wird der Objektname nicht mit gespeichert; auch zu Funktionen gehörende *environments* werden nicht zusammen mit den Funktionen gespeichert).

Diese vier Funktionen bieten sich auch für den Austausch von Objekten zwischen S-PLUS und R an.

Der Datensatz *iris* lässt sich speichern durch

```
> dump("iris", file = "iris.txt")
```

mit folgender ASCII Repräsentation:

```
"iris" <-
structure(list(Sepal.Length = c(5.1, 4.9, 4.7, 4.6, 5, 5.4, 4.6,
5, 4.4, 4.9, 5.4, 4.8, 4.8, 4.3, 5.8, 5.7, 5.4, 5.1, 5.7, 5.1,
. . . .
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3), .Label =
c("setosa", "versicolor", "virginica"), class = "factor")),
.Names = c("Sepal.Length", "Sepal.Width", "Petal.Length",
"Petal.Width", "Species"), row.names = c("1", "2", "3", "4", "5",
. . . .
"146", "147", "148", "149", "150"), class = "data.frame")
```

Den Lesern sei empfohlen, das Vorgehen auszuprobieren und die resultierende Datei vollständig zu betrachten. Man sieht, dass die Datei einen gültigen R Ausdruck enthält, der ausgeführt werden kann. Die Funktion `source("iris.txt")` führt tatsächlich den enthaltenen Ausdruck aus, womit das Objekt `iris` im *Workspace* neu erzeugt wird.

Neben der Möglichkeit zum *Austausch* von Objekten bietet sich das Speichern als ASCII-Datei mit Hilfe von `dump()` auch an, wenn Objekte außerhalb von R einfach editiert werden sollen. Letzterer Punkt findet insbesondere bei Funktionen (s. Abschn. 4.1) Verwendung, da es sehr lästig ist, längere Funktionen in der R Kommandozeile zu bearbeiten. Es wird daher i.A. auf externe Editoren zurückgegriffen (für Details zu Editoren s. Anhang B). Auch hier wird `source()` verwendet, um die Funktionen (oder andere Objekte) wieder einzulesen.

3.4 Spezielle Datenformate

Für Formate, für die keine Importfunktion erhältlich ist, kann man leicht eigene Import- und Exportfunktionen schreiben, indem man s.g. *Connections* (Ripley, 2001a) zusammen mit Funktionen benutzt, die einzelne Zeilen, Zeichen oder Bytes lesen und schreiben können.

Bei *Connections* handelt es sich um Verbindungen zum Lesen bzw. Schreiben von Informationen z.B. in Dateien oder Geräte. Tabelle 3.1 enthält eine Liste von Funktionen, die solche Verbindungen aufbauen können. Das wirkliche Lesen und Schreiben der Daten aus bzw. zu *Connections* geschieht dann mit

- `readBin()` und `writeBin()` für den Zugriff auf einzelne Bytes und
- `readChar()` und `writeChar()` für den Zugriff auf einzelne Zeichen.

Stattdessen können auch die in Abschn. 3.1 beschriebenen Funktionen verwendet werden.

Die meisten Formate bestehen aus einem Dateikopf (*header*), in dem die relevanten Informationen über das Format des folgenden Datenteils (*body*) der Datei enthalten sind. Eine übliche Importfunktion würde also zunächst den Kopf lesen, diesen interpretieren und dann korrekt den Datenteil lesen, während eine Exportfunktion den Kopf aus den zu schreibenden Daten errechnen muss. Als einfache Beispiele mögen die Funktionen `readWave()` und `writeWave()` aus dem Paket **tuneR** dienen.

Im Rahmen eines Projekts hat der Autor einen Importfilter für ein Dateiformat geschrieben, in dem binär die Messdaten mehrerer Sensoren gleichzeitig mit z.T. unterschiedlichen Samplingraten aufgezeichnet waren. Der Import mit Hilfe einer R Funktion ist i.d.R. zwar langsamer als etwa mit einem auf *C* basierenden Importfilter, aber sehr einfach zu implementieren.

Tabelle 3.1. *Connections* zum Lesen und Schreiben von Daten

Funktion	Beschreibung
`file()`	Datei auf dem lokalen Rechner / im lokalen Netzwerk
`pipe()`	Direkte Eingabe aus einem anderen Programm
`fifo()`	*First In First Out*, Ein- / Ausgabe aus einem anderen laufenden Prozess (nicht unter Windows)
`url()`	Zugriff auf Daten im Internet, z.B. `http`, `ftp`, ...
`gzfile()`, `bzfile()`	Direktes Lesen und Schreiben komprimierter Daten
`socketConnection()`	Zugriff auf ein „Gerät“ im Sinne des Betriebssystems
`open()`	Erneutes Öffnen einer geschlossenen Verbindung
`close()`	Schließen einer offenen Verbindung

3.5 Zugriff auf Datenbanken

Wer mit wirklich großen Datenbeständen arbeitet wie bei

- Kundendatenbanken (z.B. in Einzelhandelsketten),
- Mobilfunk (z.B. Abrechnung, Netzentwicklung),
- Computerchipherstellung oder
- Genomforschung (z.B. DNA-Sequenzierung),

hat keine Daten der Größe Kilo- oder MegaByte, sondern kommt in Bereiche von Giga- oder TeraByte. Man kann also nicht mehr den vollständigen Datensatz in den Hauptspeicher des Rechners laden und Auswertungen machen, sondern man muss Teilmengen der Daten von einer Datenbank anfordern und evtl. schon elementare Rechnungen vom Datenbanksystem durchführen lassen. Für solche Datenbanksysteme und deren Verwaltung wird seit einiger Zeit nicht nur von Softwareunternehmen und deren Kunden der Begriff *Data Warehousing* geprägt.

Für die Kommunikation mit Datenbanken wird meist die Sprache *SQL* (**S***tructured* **Q***uery* **L***anguage*, manchmal mit datenbankspezifischen Abwandlungen) verwendet. Schnittstellen zu verschiedenen Datenbanksystemen mit der Möglichkeit zum Absetzen von *SQL* Anfragen bieten u.a. die Pakete **PostgreSQL**, **RmSQL**, **RMySQL**, **RODBC**, **RPgSQL** und **RSQLite**. Details sind in der entsprechenden Dokumentation zu finden. Der Name des jeweiligen Pakets gibt implizit an, für welches zugehörige Datenbanksystem eine Schnittstelle geliefert wird, so kann man z.B. mit Hilfe des Pakets **RMySQL** von R auf MySQL Datenbanken zugreifen. Einen allgemeinen Artikel zur Benutzung von Datenbanken mit R hat Ripley (2001b) verfasst.

Das Paket **DBI** (Hothorn et al., 2001b) bietet ein höheres Interface zwischen Benutzer und einigen der o.g. Systeme (z.B. **RPgSQL** und **ROracle**). Damit wird es ermöglicht, eine identische Syntax für verschiedene Datenbanksysteme zu benutzen.

Unter Windows eignet sich auch das Paket **RODBC** (Schnittstelle zu Microsofts Datenbank-Protokoll ODBC), um auf MySQL Datenbanken zuzugreifen, wofür der entsprechende *MyODBC* Treiber installiert sein muss.

Das folgende Beispiel zeigt die Benutzung einer MySQL Datenbank mit Hilfe von **RODBC** unter den Annahmen, dass man Zugriff auf eine solche hat (inklusive der Berechtigung, Tabellen anzulegen), unter Windows arbeitet und einen geeigneten *MyODBC* Treiber installiert hat. Die Datenquelle, im Beispiel „Mining“, sollte zunächst unter Windows in der Systemsteuerung als *DSN* (*Data Source Name*) bekannt gemacht werden.

```
> library("RODBC")                  # das Paket laden
> channel <- odbcConnect("Mining")  # Verbindung öffnen
```

An dieser Stelle werden in einem Windows-Dialog u.a. Benutzername und Kennwort abgefragt. Als Nächstes soll eine Tabelle „iristab“ auf dem Datenbank-Server angelegt werden, die die bereits mehrfach verwendeten *iris* Daten enthält (s. Abschn. 2.5):

```
> # Datensatz "iris" in Tabelle "iristab" schreiben:
> sqlSave(channel, iris, "iristab")

> sqlTables(channel)                # vorh. Tabelle(n) anzeigen
   Table_qualifer Table_owner Table_name Table_type     Remarks
 1                              iristab      TABLE MySQL table
```

Aus der eben neu erzeugten Tabelle auf unserem Datenbank-Server soll nun einmal der ganze Datensatz (im ersten Ausdruck) abgefragt werden. Danach sollen nur diejenigen Beobachtungen abgefragt werden, bei denen die Variable „Species“ den Wert „virginica“ hat und zugleich der Wert von „PetalLength“ größer als 6 ist.

```
> sqlQuery(channel, "select * from iristab")
  rownames SepalLength SepalWidth PetalLength PetalWidth Species
1        1         5.1        3.5         1.4        0.2  setosa
2        2         4.9        3.0         1.4        0.2  setosa
3        3         4.7        3.2         1.3        0.2  setosa
. . . . # usw.

> sqlQuery(channel,
+     "select * from iristab where Species = 'virginica'
              and PetalLength > 6")  # muss in einer Zeile stehen!
  rownames SepalLength SepalWidth PetalLength PetalWidth   Species
1      106         7.6        3.0         6.6        2.1 virginica
2      108         7.3        2.9         6.3        1.8 virginica
3      110         7.2        3.6         6.1        2.5 virginica
. . . . # usw.

> close(channel)
```

Zuletzt sollte man die Verbindung zur Datenbank mit `close()` wieder schliessen. Ein weiteres Beispiel für den Datenbankzugriff findet man im folgenden Abschnitt für den Zugriff auf Excel-Dateien.

3.6 Zugriff auf Excel-Daten

In diesem Abschnitt wird das Importieren und Exportieren von Microsofts häufig verwendetem Excel-Format beschrieben.

Umweg über Textdateien

Der einfachste Umweg zum Importieren und Exportieren von vereinzelten, nicht allzu großen Excel-Dateien führt über das ASCII-Format. Daten können von Excel als Text des Formats „Tabstopp-getrennt“ exportiert werden. In R kann eine solche Datei dann mit

```
read.table(Dateiname, header = TRUE, sep = "\t", dec = ",")
```

eingelesen werden. Aus R mit

```
write.table(Objekt, file = Dateiname, row.names = FALSE,
    sep = "\t", dec = ",")
```

exportierte Daten können von Excel direkt wieder gelesen werden.

Lesen mit `read.xls()` *aus dem Paket* **gdata**

In dem auf CRAN erhältlichen Paket **gdata** von Gregory R. Warnes gibt es die Funktion `read.xls()` für das Lesen (nicht aber zum Schreiben) von Excel-Dateien. Die Funktion basiert auf dem Perl Skript `xls2csv.pl` desselben Autors, das die Excel-Tabelle in ein durch Kommata getrenntes ASCII-Format konvertiert, das letztendlich mit Hilfe von `read.csv()` in R eingelesen wird. Zur Verwendung der Funktion `read.xls()` wird eine auf dem Rechner vorhandene Perl Installation benötigt.

Excel als Datenbank

Wenn man eine größere Anzahl an Excel-Dateien nach R überführen möchte oder die Daten einer Excel-Datei immer wieder aktualisieren muss, bietet es sich an, Excel als Datenbank (s. Abschn. 3.5) zu verwenden. Das Paket **RODBC** kann mit Hilfe des ODBC Protokolls nicht nur auf Access Datenbanken zugreifen, sondern auch Excel als Datenbank verwenden.

Im folgenden Beispiel soll der Zugriff auf eine Excel-Datei (sie sei zu finden unter `c:\irisdat.xls`) gezeigt werden, die eine Tabelle mit den *iris* Daten enthalte, wobei der Tabellenname entsprechend „iris“ sei. Der Zugriff kann völlig analog zum Beispiel in Abschn. 3.5 erfolgen, wird hier aber mit Hilfe der sehr komfortablen Funktion `odbcConnectExcel()` durchgeführt. Voraussetzung dazu ist, dass englischsprachige ODBC-Treiber installiert sind. Es gibt auch für direkten Zugriff auf Access analog eine Funktion `odbcConnectAccess()`.

```
> library("RODBC")                                # das Paket laden
> channel <- odbcConnectExcel("C:/irisdat.xls") # Verbindung öffnen
> sqlTables(channel)                              # Name der Tabellen
     TABLE_CAT TABLE_SCHEM TABLE_NAME   TABLE_TYPE REMARKS
 1 C:\\irisdat        <NA>      iris$ SYSTEM TABLE    <NA>
```

```
> sqlQuery(channel, "select * from \"iris$\"")  # Datensatz lesen
     Sepal#Length Sepal#Width Petal#Length Petal#Width    Species
 1            5.1         3.5          1.4         0.2     setosa
 2            4.9         3.0          1.4         0.2     setosa
 3            4.7         3.2          1.3         0.2     setosa
 . . . . # usw.

> close(channel)                              # Verbindung schließen
```

Es fällt auf, dass der Name der Tabelle durch das ODBC Protokoll zu „iris$“ verfälscht wird, wodurch es leider notwendig wird, den Namen in Anführungszeichen (s. auch Abschn. 2.11, S. 56) zu setzen, die selbst wieder durch das Zeichen „\“ eingeleitet werden müssen, weil sie innerhalb anderer Anführungszeichen stehen und weitergeleitet werden sollen. Außerdem wird der ursprüngliche Punkt in den Spaltennamen (z.B. in `Petal#Length`) offensichtlich in eine Raute (`#`) umgewandelt.

Zugriff per DCOM

Das R-Excel Interface[1] von Baier und Neuwirth (2003), das mit Hilfe des R-(D)COM[2] Servers (Baier, 2003) kommuniziert, dient u.a. dazu, R als Server für Berechnungen in Excel-Tabellen zu benutzen. Es kann auch dazu verwendet werden, Daten zwischen beiden Programmen auszutauschen. Details findet man in den o.g. Quellen und der jeweiligen Dokumentation. Ein solcher Datenaustausch empfiehlt sich aber nur, wenn man ohnehin eine Kommunikation über das DCOM Protokoll verwenden möchte, da die Installation im Vergleich zur Kommunikation über ODBC deutlich aufwändiger ist.

[1] `http://CRAN.R-project.org/other-software.html`
[2] (D)COM: *(Distributed) Component Object Model*

4

Die Sprache im Detail

Grundlagen für das interaktive Arbeiten mit R wurden in den ersten Kapiteln vermittelt. In diesem Kapitel werden Details zur Sprache beschrieben, deren Kenntnis zum Schreiben eigener Funktionen, Simulationen und Pakete notwendig oder zumindest von Vorteil ist. Dabei sind neben einer formalen Einführung von Funktionen (s. Abschn. 4.1) vor allem die Regeln der *Lazy Evaluation* (verzögerte Auswertung, s. Abschn. 4.2) und die Regeln zu *Environments* (Umgebungen, s. Abschn. 4.3) wesentlich, da deren Ausnutzung nützlich ist, ihre Missachtung aber zu überraschenden Fehlern führen kann. Ein wesentlicher Bestandteil des Programmierens ist das Auffinden und Beseitigen von Fehlern (s. Abschn. 4.4), die gerade bei längeren Funktionen unvermeidlich sind, aber auch der Umgang mit zu erwartenden Fehlern. Die Abschnitte 4.5 zur Rekursion, 4.6 zum Umgang mit Sprachobjekten und 4.7 zum Vergleich von Objekten runden das Kapitel ab.

4.1 Funktionen

Als Grundlage zu den folgenden Abschnitten werden zunächst Funktionen und deren Eigenschaften detailliert beschrieben. Zwar wurde bereits eine ganze Reihe verschiedener Funktionen benutzt, Syntax, Definition und Eigenschaften von Funktionen wurden in Kapitel 2 aber nur implizit bzw. verkürzt eingeführt.

Es sei zunächst noch einmal darauf hingewiesen, dass R eine *funktionale Sprache* ist. Prinzipiell werden alle Operationen, selbst Zuweisungen oder die Ausgabe von Werten auf die Konsole, durch Funktionen bearbeitet. So gibt es spezielle Funktionen mit Kurzformen, z.B. ist auch der Operator „`+`“ eine Funktion, so dass die Addition `3 + 5` in voller Form als `"+"(3, 5)` angegeben werden müsste. Der Einfachheit halber ist jedoch die Benutzung als Operator

(`3 + 5`) möglich. Ebenso müsste (und kann!) die Zuweisung `x <- 3` in Funktionsform mit `"<-"(x, 3)` angegeben werden. Insbesondere geschieht jegliches Arbeiten mit Daten mit Hilfe von Funktionen.

4.1.1 Funktionsaufruf

Ein Funktionsaufruf hat die Form

```
funktionsname(Argument1 = Wert1, Argument2 = Wert2)
```

wie es bereits aus den ersten Kapiteln bekannt ist. Dem Funktionsnamen folgen in runden Klammern kein, ein oder mehrere durch Kommata getrennte Argumente, die benannt oder unbenannt spezifiziert werden können.

Argumente

Formal in der Funktion definierte Argumente können s.g. *defaults* (Voreinstellungen) haben, die nicht beim Funktionsaufruf mit angegeben werden müssen. Ein Argument ohne Voreinstellung muss beim Funktionsaufruf i.d.R. angegeben werden, eine Ausnahme bildet der Fall, dass innerhalb einer Funktion auf das Fehlen eines formalen Arguments mit `missing()` getestet und entsprechend gehandelt wird.

Es gibt auch Funktionen ohne Argumente, z.B. gibt `getwd()` das aktuelle Arbeitsverzeichnis an. Namen für Argumente und Voreinstellungen zu Argumenten sind in der Regel auf der Hilfeseite der entsprechenden Funktion (`?funktionsname`) angegeben. Man muss also zwischen den *formal* in der Funktion definierten Argumenten und den *tatsächlich* beim Funktionsaufruf gegebenen Argumenten unterscheiden. Details dazu werden an folgendem Beispiel erläutert.

Die Funktion zur Berechnung des Medians eines Vektors hat gemäß ihrer Hilfeseite die vollständige Syntax `median(x, na.rm = FALSE)`. Die hier formal definierten Argumente sind `x` (ohne Voreinstellung) und `na.rm`, dessen Voreinstellung `FALSE` ist. Die folgenden Aufrufe von `median()` sind dann z.B. möglich:

```
> a <- c(3, 1, 5, NA)
> median(a)                    # 1. Möglichkeit
 [1] NA
> median(a, TRUE)              # 2. Möglichkeit
 [1] 3
> median(na.rm = TRUE, x = a)  # 3. Möglichkeit
 [1] 3
> median(na = TRUE, a)         # 4. Möglichkeit
 [1] 3
```

Die Reihenfolge von formal definierten und angegebenen Argumenten muss also nicht unbedingt übereinstimmen.

Die Regeln für die Zuordnung von spezifizierten zu formalen Argumenten werden in der folgenden Reihenfolge angewandt:

- Alle Argumente mit vollständigem Namen werden zugeordnet (z.B. in (3): `x = a`, `na.rm = TRUE`; offensichtlich können benannte Argumente also in beliebiger Reihenfolge stehen).
- Argumente mit teilweise passendem Namen werden den übrigen formalen Argumenten zugeordnet (z.B. in (4): `na = TRUE`). Hierbei müssen die Anfangsbuchstaben übereinstimmen und (nach Anwendung des o.g. Punktes) eindeutig zu einem noch nicht benutzten formalen Argument passen.
- Alle unbenannten Argumente werden der Reihe nach den noch übrigen formalen Argumenten zugeordnet (z.B. in (2) der Reihenfolge entsprechend `a` zu `x` und `TRUE` zu `na.rm` ; bzw. in (4): `a` zu `x`, weil `na.rm` bereits gesetzt ist).
- Übrige Argumente, die jetzt noch nicht von einer der vorherigen Regeln erfasst wurden, werden dem evtl. vorhandenen formalen „Dreipunkte"-Argument „`...`" zugeordnet (s. Abschn. 4.1.2).

4.1.2 Eigene Funktionen definieren

Eigene Funktionen sind immer dann sinnvoll, wenn eine Folge von anderen Funktionsaufrufen (unter einem Namen) zusammengefasst werden soll, z.B. für mehrmaliges Ausführen mit verschiedenen Parametern. Da Funktionen meist aus mehreren oder vielen Zeilen an Code bestehen, sollte man Funktionen nicht in der R-Konsole schreiben, sondern einen geeigneten Editor verwenden (s. Anhang B), der das Programmieren deutlich vereinfacht. Das Einlesen einer komplett innerhalb einer Datei definierten Funktion erfolgt mit Hilfe von `source()`.

Eine Funktionsdefinition geschieht mittels `function()` und hat die Form

```
MeineFunktion <- function(Argumente){
    # Befehlsfolge / "body" der Funktion
}
```

Dabei werden die `Argumente` mit oder ohne Voreinstellung angegeben. Beim Aufruf der Funktion werden die `Argumente` als Objekte an die `Befehlsfolge`, den s.g. *body* der Funktion, weitergereicht. Nicht nur bei der Definition von Funktionen, sondern auch bei allen anderen Konstruktionen (z.B. `for()` und `if()`, s. Abschn. 2.10) können Befehlsfolgen, solange sie in geschweiften Klammern stehen, aus mehreren Zeilen bestehen. Eine typische Definition einer Funktion sieht damit, wieder am Beispiel der Medianberechnung (unwesentlich editierte Funktion `median()` aus R), wie folgt aus:

```
median <- function(x, na.rm = FALSE){
  if(mode(x) != "numeric")
    stop("need numeric data")
  if(na.rm)
    x <- x[!is.na(x)]
  else if (any(is.na(x)))
    return(NA)
  n <- length(x)
  if(n == 0)
    return(NA)
  half <- (n + 1)/2
  if(n %% 2 == 1){
    sort(x, partial = half)[half]
  } else{
    sum(sort(x, partial =
        c(half, half + 1))[c(half, half + 1)]) / 2
  }
}
```

Rückgabe von Objekten und Ausgabe von Informationen

Das zuletzt in einer Funktion erzeugte Objekt wird nach deren Aufruf zurückgegeben. Die explizite Rückgabe, die ästhetischer ist und auch an anderer Stelle als am Ende möglich ist, erfolgt mit `return()`, wobei die Funktion automatisch beendet wird. Die Funktion `invisible()` kann verwendet werden, wenn die Rückgabe „unsichtbar“ erfolgen soll, d.h. ohne Ausgabe auf die Konsole. Wenn mehrere Objekte zurückgegeben werden sollen, werden sie üblicherweise als (benannte) Elemente einer Liste (Funktion `list()`, s. Abschn. 2.9.4) zusammengefasst. Der Wert einer Funktion (also das mit `return()` zurückgegebene Objekt) kann dann bei Funktionsaufruf einem neuen Objekt zugewiesen werden. Wenn keine Zuweisung erfolgt, wird der Wert bei interaktiver Benutzung auf der Konsole als Textrepräsentation ausgegeben. Alle anderen Objekte, die innerhalb einer Funktion erzeugt wurden, sind nach Beenden der Funktion wieder gelöscht (s. Abschn. 4.3).

Zur Ausgabe von Informationen oder Objekten in Textform auf die Konsole oder in Text-Dateien eignen sich die in Abschn. 2.11 beschriebenen Funktionen `cat()` und `print()`. Man beachte, dass mit `cat()` oder `print()` auf die Konsole geschriebene Informationen nicht einem Objekt zugewiesen werden können.

Das „Dreipunkte“-Argument

Der Einsatz des formalen „Dreipunkte“-Arguments ermöglicht es, *tatsächlich* angegebene Argumente ohne zugehörige korrespondierende formale Argumen-

te an andere Funktionen durch den Aufruf mittels „...“ weiterzuleiten, sie können aber auch innerhalb der Funktion weiterverarbeitet werden.

Am Beispiel einer sonst trivialen Funktion `punkte()` soll das Prinzip erläutert werden. In der Funktionsdefinition gibt es die formalen Argumente `x` und „...“, die beide direkt an die Funktion `matrix()` weitergeleitet werden, deren Wert anschließend ausgegeben wird.

```
> punkte <- function(x, ...){
+     matrix(x, ...)
+ }
> a <- c(3, 5, 7, 2)            # Es gibt kein Argument,
> punkte(a)                     #  das dem "..." zugeordnet wird
      [,1]
 [1,]    3
 [2,]    5
 [3,]    7
 [4,]    2
> punkte(a, ncol = 2, byrow = TRUE)  # ncol und byrow werden
      [,1] [,2]                     #  an matrix() weitergeleitet
 [1,]    3    5
 [2,]    7    2
```

Im letzten Aufruf wird das erste im Aufruf angegebene Argument `a` dem formalen Argument `x` zugeordnet. Alle weiteren beim Aufruf genutzten Argumente werden dem formalen „...“Argument zugeordnet, das sie direkt an `matrix()` weiterleitet. Dadurch erspart man sich in einer neuen Funktion u.U. die Definition vieler formaler Argumente.

Das „Dreipunkte“-Argument wird besonders häufig in Grafikfunktionen verwendet, wenn z.B. Einstellungen zu Schriftgrößen oder Farben an viele innerhalb einer solchen Funktion verwendete Funktionen gleichzeitig weitergeleitet werden sollen, das Aufführen aller möglichen Argumente aber einen großen Aufwand erfordern würde.

Dokumentation

Leider wird die Dokumentation eigener Funktionen, gerade wenn sie „mal eben“ geschrieben werden, immer wieder vergessen. Es ist aber einfach und schnell möglich, Kommentare (s. Abschn. 2.1) zur Beschreibung von Programmstücken, Argumenten und Ausgabe einer Funktion einzufügen. Ein eindringlicher Appell zur Dokumentation erfolgt in Abschn. 5.1.

Gerade für Funktionen, die über einen längeren Zeitraum immer wieder verwendet werden sollen oder an Dritte weitergegeben werden, empfiehlt sich dringend die Erstellung von Hilfeseiten oder gar die Zusammenstellung als

Paket. In Abschn. 10.7 werden Formatvorgaben für die Hilfeseiten und Werkzeuge, die bei der Erstellung solcher Hilfeseiten sehr große Hilfestellungen geben, beschrieben.

Eine Übung zu Funktionen

Als einfache Übung zum Schreiben einer eigenen Funktion kann z.B. das Newton'sche Iterationsverfahren zur Bestimmung der Quadratwurzel einer Zahl programmiert werden. Wer üben möchte, möge sich das folgende Programm zunächst nicht anschauen.

Bei der Suche der Nullstelle einer Funktion $f(x)$ führt das Verfahren zur Vorschrift $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. Um also $\sqrt{y}$ zu bestimmen, kann man die Nullstelle von $f(x) = x^2 - y$ suchen, so dass die Vorschrift lautet: $x_{n+1} = x_n - \frac{x_n^2 - y}{2x_n}$. Es fehlt dann noch ein Abbruchkriterium, das die Iteration bei Erreichen eines genügend genauen Wertes abbricht. Dazu wird gewartet, bis sich das Ergebnis fast nicht mehr ändert, die Differenz zweier Iterationsschritte also sehr klein ist.

```
> Newton.Wurzel <- function(y, Startwert, eps = 10^(-7)){
+     x <- Startwert                        # Mit Startwert beginnen
+     repeat{
+         x.neu <- x - (x^2 - y) / (2 * x) # Iterationsvorschrift
+         if(abs(x - x.neu) < eps) break   # Abbruchkriterium
+         x <- x.neu
+     }
+     return(x.neu)
+ }
> Newton.Wurzel(4, 1)
 [1] 2
```

Es gibt in R natürlich sowohl bessere Funktionen zur Berechnung der Quadratwurzel (z.B. `sqrt()`) als auch effizienter programmierte Optimierungsverfahren (z.B. `optimize()` oder `optim()`).

4.2 Verzögerte Auswertung – *Lazy Evaluation*

In manchen Situationen, insbesondere wenn statt eines einfachen Objekts ein Ausdruck als Argument spezifiziert wird, fällt auf, dass Argumente in Funktionsaufrufen der *Lazy Evaluation* (wörtlich: „faule Auswertung") unterliegen, also einer verzögerten Auswertung. Argumente und darin spezifizierte Ausdrücke werden nämlich erst ausgewertet, wenn das jeweilige Argument innerhalb der Funktion zum ersten Mal benutzt wird. Als Beispiel diene zunächst die Funktion `faul()`:

```
> faul <- function(x, rechnen = TRUE) {
+     if(rechnen) x <- x + 1
+     print(a)
+ }
> faul((a <- 3), rechnen = FALSE)          # (1)
 Error in print(a) : Object "a" not found
> faul(a <- 3)                             # (2)
 [1] 3
```

Im ersten Aufruf ist das Argument `rechnen = FALSE`, wobei innerhalb der Funktion nur `print(a)` ausgewertet wird, nicht aber `x <- x + 1`. Da `x` noch nicht ausgewertet wurde, wurde auch die im Argument benutzte Zuweisung `a <- 3` nicht durchgeführt, das Objekt `a` ist also unbekannt.

Im zweiten Aufruf hingegen ist per Voreinstellung `rechnen = TRUE`, die Zeile `x <- x + 1` wird damit ausgewertet. Weil das formale Argument `x` hier konkret mit dem Ausdruck `a <- 3` angegeben wurde, wird dieser ausgewertet, `a` ist im Schritt `print(a)` also bekannt.

Es macht i.A. also wenig Sinn (und ist gefährlich!) Zuweisungen in Argumenten durchzuführen. Hingegen zeigt sich die Nützlichkeit der verzögerten Auswertung an folgendem Beispiel:

```
> aufruf <- function(x)
+     return(list(Aufruf = substitute(x), Wert = x))
> aufruf(1 + 2)
 $Aufruf
 1 + 2

 $Wert
 [1] 3
```

Die triviale Funktion `aufruf()` enthält nur ein `return()`, welches eine Liste der Länge 2 mit den Elementen `Aufruf` und `Wert` zurückgibt. Weil das Argument `x` erst bei seinem Aufruf ausgewertet wird, kann `substitute(x)` (s. Abschn. 4.6) noch den urspünglichen Ausdruck extrahieren, bevor `x` als `Wert` benutzt und zurückgegeben wird. Dieser „Trick“ wird sehr häufig in Grafikfunktionen als Voreinstellung für zum Argument korrespondierende Beschriftungen verwendet.

4.3 Umgebungen und deren Regeln – *Environments* und *Scoping Rules*

Eine wichtige Frage im Zusammenhang mit Funktionen ist, wann welche Objekte existieren bzw. sichtbar sind. Wenn man direkt in der Kommandozei-

le von R arbeitet, werden alle neu erzeugten Objekte im Workspace abgelegt. Werden aber Funktionen aufgerufen, so möchte man natürlich nicht, dass alle innerhalb einer Funktion durch Zuweisung erzeugten Objekte auch im Workspace liegen und dort liegende Objekte überschreiben. Andererseits sollen natürlich anstelle von gleichnamigen Objekten im Workspace die zuletzt innerhalb der Funktion erzeugten Objekte benutzt werden. Gerade in komplexeren Funktionen werden doch sehr viele Objekte erzeugt, die nur vorübergehend benötigt werden. Daher macht es Sinn, Funktionen in einer eigenen Umgebung (*environment*) auszuführen, so dass nicht alle Objekte im Workspace abgelegt werden und zu Unübersichtlichkeit und Speicherverschwendung führen. Diese Umgebungen sollten i.d.R. (außer wenn Funktionen zurückgegeben werden, s. S. 79) nach Beenden der Funktion zusammen mit den darin enthaltenen Objekten gelöscht werden – abgesehen von (z.B. mit `return()`) zurückgegebenen Objekten. Weniger formal formuliert handelt es sich bei einer Umgebung also um eine virtuelle Hülle, in der Objekte, nahezu ohne Interaktionen zu anderen gleichzeitig existierenden Hüllen, gebündelt werden können.

Die Regeln im Überblick

Der Artikel „Lexical Scope and Statistical Computing“ von Gentleman und Ihaka (2000) ist grundlegend für die auch in R verwendeten Regeln des *Lexical Scoping*. Die *Scoping Rules* sind diejenigen Regeln, die u.a. festlegen, in welcher Reihenfolge verschiedene Umgebungen nach Objekten (passend nach Namen) durchsucht werden und wann bzw. wie Umgebungen erzeugt werden.

Gerade bei den *Scoping Rules* unterscheiden sich R und das hier nicht beschriebene S-PLUS sehr stark. Für einen Vergleich sei auf Venables und Ripley (2000) verwiesen. Darin findet man auch einige hier nicht beschriebene Details und Beispiele, die besonders interessant werden, wenn komplexe Funktionen, die direkt oder indirekt auf diese Regeln zugreifen, unter beiden Programmen gleichermaßen laufen sollen.

Die wesentlichen für Benutzer „sichtbaren“ Regeln sind:

- Alle in R erzeugten Umgebungen werden im Hauptspeicher (RAM) des Rechners abgelegt.
- Standardmäßig werden Objekte, die auf der Konsole bzw. in einem Skript (im nicht interaktiven *Batch* Modus) erzeugt werden, im Workspace („`.GlobalEnv`“, s. auch 2.6) gespeichert. Das gilt nicht für innerhalb einer Funktion erzeugte Objekte.
- Die `.GlobalEnv` Umgebung steht im *Suchpfad* an Stelle 0 („in der Mitte“), wobei man sich unter einem Suchpfad einen Pfad vorstelle, entlang dessen nach Objekten gesucht wird.

Im Suchpfad werden alle aktuellen Umgebungen angegeben. Das sind in erster Linie Pakete, um darin enthaltene Funktionen nutzen zu können, oder Datensätze, um auf deren einzelne Variablen zugreifen zu können. An letzter Stelle im Suchpfad steht das Paket **base**. Eingehängte Objekte (mit `library()` oder `attach()`, s. Abschn. 2.9.5) sind zwischen `.GlobalEnv` und **base** angeordnet.
Der aktuelle Suchpfad kann mit Hilfe von `search()` ausgegeben werden und lautet bei standardmäßigem Aufruf von R zunächst wie folgt (man ignoriere die Nummerierung der Elemente des Vektors):

```
> search()
 [1] ".GlobalEnv"        "package:methods"   "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:datasets"
 [7] "package:utils"     "Autoloads"         "package:base"
```

Zwischen `.GlobalEnv` (Stelle 0) und Paket **base** (hier an Stelle -8) sieht man die beim R–Start standardmäßig geladenen anderen Pakete.

- Bei Aufruf einer Funktion wird eine eigene Umgebung kreiert, die dem Suchpfad an Stelle 1 vorangestellt wird. Eine innerhalb einer ersten Funktion aufgerufene Funktion erhielte wieder eine eigene Umgebung (Stelle 2, vor 1 im Suchpfad), weil die Umgebung der ersten Funktion wegen der Verschachtelung noch nicht geschlossen wurde.
  Das Voranstellen im Suchpfad hat den Sinn, dass der Pfad dann vom Anfang zum Ende kontinuierlich abgearbeitet werden kann. Dadurch kann angefangen von der Umgebung der aktuellen Funktion, über den Workspace, bis zum Paket **base** so gesucht werden, dass immer das aktuellste (zuletzt definierte) Objekt unter gleichnamigen Objekten auch als Erstes gefunden wird; denn das in der gerade laufenden Funktion ist neuer als das im Workspace. Des Weiteren sollten Objekte, die man selbst im Workspace definiert hat, natürlich auch vor in Paketen definierten Objekten gefunden werden.
  Eine Ausnahme bzw. Erweiterung für den hier beschriebenen Suchpfad wird durch Namespaces (s. S. 81) gebildet.

Als Beispiel generieren wir eine einfache Funktion `einfach()`, die das Argument `x` und dessen Median ausgibt:

```
> einfach <- function(x){
+     med <- median(x)
+     return(list(x = x, median = med))
+ }
```

```
> einfach(1:3)
 $x
 [1] 1 2 3

 $median
 [1] 2
```

Bei Aufruf von `einfach(1:3)` wird zunächst eine Umgebung (1) erzeugt, in der die Funktion `einfach()` Objekte ablegt, zunächst einmal die übergebenen Argumente, hier also `x`. Dann wird durch Aufruf von `median()` eine weitere Umgebung (2) erzeugt, in der zunächst wieder ein `x` ist. Innerhalb von `median()` werden wieder Funktionen aufgerufen (s. Abschn. 4.1.2), z.B. `sort()`, wofür wieder eine Umgebung (3) erzeugt wird. Die an dieser Stelle existierende Liste von Umgebungen sieht wie folgt aus:

```
-8 package:base
-7 Autoloads
-6 package:utils
-5 package:datasets
-4 package:grDevices
-3 package:graphics
-2 package:stats
-1 package:methods
 0 .GlobalEnv           # Workspace
 1 environment 1        # Funktion einfach()
 2 environment 2        # Funktion median()
 3 environment 3        # Funktion sort()
```

Wenn `sort()` beendet ist, wird deren Umgebung (3) wieder gelöscht und nach Beenden von `median()` wird auch hier die zugehörige Umgebung (2) gelöscht, das Ergebnis jedoch einem Objekt `med` in der zu `einfach()` gehörenden Umgebung zugewiesen (dahin „gerettet“). Wenn `einfach()` beendet wird, wird entsprechend auch hier die zugehörige Umgebung mit den enthaltenen Objekten `med` und `x` gelöscht – abgesehen von der mit `return()` in die `.GlobalEnv` zurückgegebenen Liste.

Falls in einer verschachtelten Funktion (z.B. auf Ebene 2) auf ein Objekt zugegriffen werden soll, so werden auch alle darunter liegenden Umgebungen (2, 1, 0, -1, ...) durchsucht, wie z.B. in:

```
> x <- 5
> scope <- function(do.it = TRUE){
+     if(do.it) x <- 3
+     innen <- function()
+         print(x)
+     innen()
+ }
```

```
> scope()
 [1] 3
> scope(FALSE)
 [1] 5
```

Im Workspace gibt es ein Objekt `x` mit Wert `5`. Innerhalb der Funktion `scope()` wird je nach Wahl des Arguments `do.it` ein Objekt `x` mit Wert `3` erzeugt oder nicht. In `scope()` wird eine Funktion `innen()` definiert und aufgerufen, die auf ein Objekt `x` zugreift. `innen()` sucht zunächst in der Umgebung, die bei ihrem Aufruf kreiert wurde, in der aber kein `x` existiert. Dann wird in der Umgebung der aufrufenden Funktion `scope()` gesucht, in der nur dann ein `x` gefunden wird, falls `do.it TRUE` ist. Bei Aufruf `scope()` wird also dieses `x` mit `print()` ausgegeben und nicht das aus dem Workspace. Im Fall des Aufrufs `scope(FALSE)` hingegen wird auch in der Umgebung von `scope()` kein `x` existieren. Deshalb wird als Nächstes im Workspace nach `x` gesucht. Das dort gefundene `x` wird ausgegeben.

Aus diesem Grund sollte man Objekte, die innerhalb einer Funktion benötigt werden, nicht nur aus ästhetischen Gründen als Argumente übergeben. Sonst wird darauf vertraut, dass das jeweilig richtige Objekt in der richtigen Umgebung gefunden wird, obwohl es mehrere gleichnamige Objekte in verschiedenen Umgebungen geben kann.

An dieser Stelle sollen die Grundlagen der *Scoping Rules* für das alltägliche Programmieren klar werden, weswegen einige Punkte leicht vereinfacht dargestellt sind. Weitere Details zum Erzeugen und Abfragen von bzw. zum Zugriff auf Umgebungen findet man auf den Hilfeseiten `?environment` und `?sys.parent`, in R Development Core Team (2006d) sowie bei Venables und Ripley (2000).

Lexical Scoping

Eine nützliche und verblüffende Eigenschaft des *Lexical Scoping* lässt sich wie folgt beschreiben[1]: Funktionen, die in einer bestimmten Umgebung (*environment*) erzeugt und dann einer anderen zugewiesen wurden, „kennen“ weiterhin die Objekte der ursprünglichen Umgebung. In diesem Fall (Ausgabe einer Funktion) wird eine Umgebung nämlich bei Beenden einer umschließenden Funktion nicht gelöscht und die ausgegebene Funktion kennt die Umgebung, in der sie erzeugt wurde. Eine solche Funktion zusammen mit einer Umgebung heißt auch *(function) closure*. Eine Umgebung wird also nur dann wirklich gelöscht, wenn die zugehörige beendete Funktion keine Funktion zurückgegeben hat, die einer anderen Umgebung zugewiesen wurde.

[1] Diese Eigenschaft des Lexical Scoping macht einen wesentlichen Unterschied zwischen R und S-PLUS aus.

Die Funktion `l.scope()` dient als kurzes Beispiel:

```
> l.scope <- function(){
+     nur.hier <- "Ich bin nur hier definiert!"

+     innen <- function()
+         print(nur.hier)
+     return(innen)
+ }
```

Es werden zwei Objekte (ausschließlich) in der Umgebung der Funktion `l.scope()` definiert, und zwar `nur.hier`, das eine selbsterklärende Zeichenkette enthält, und eine Funktion `innen()`, die bei Aufruf nichts anderes tut, als eine Variable `nur.hier` auf die Konsole auszugeben. Diese Funktion (`innen()`) wird nicht in `l.scope()` aufgerufen, sondern zurückgegeben. Bei Aufruf von `l.scope()` und Zuweisung in ein Objekt, z.B. `Ausgabe`, wird die Funktion `innen()` (aber nicht deren Wert) diesem Objekt zugewiesen:

```
> Ausgabe <- l.scope()
> ls()
 [1] "Ausgabe"  "l.scope"
> Ausgabe
 function() print(nur.hier)
 <environment: 015E8D70>
> testen <- function()
+     print(nur.hier)
> testen()
 Error in print(nur.hier) : Object "nur.hier" not found
> Ausgabe()
 [1] "Ich bin nur hier definiert!"
> ls(environment(Ausgabe))
 [1] "innen"    "nur.hier"
```

Wenn man sich die neue Funktion `Ausgabe()` anschaut, fällt auf, dass eine zugeordnete Umgebung (*environment*) mit angezeigt wird. Die angezeigte Zeichenfolge (hier: `015E8D70`) ist zufällig und eindeutig, denn eine Funktion kann mehrfach aufgerufen werden und sollte, wenn Umgebungen noch nicht wieder gelöscht sind, alte Umgebungen nicht überschreiben. Das Objekt `nur.hier` wurde nicht im Workspace definiert, denn es wurde in der Umgebung der Funktion `l.scope` erzeugt. Eine Funktion `testen()`, die abgesehen von der verknüpften Umgebung mit `Ausgabe()` identisch ist, kann das Objekt `nur.hier` also nicht finden. `Ausgabe()` hingegen durchsucht die verknüpfte Umgebung und findet das Objekt. Die Objekte in der Umgebung von `Ausgabe()` können mit `ls(environment(Ausgabe))` angezeigt werden, wobei das erste Argument von `ls()` die Umgebung angibt, deren Objekte

aufgelistet werden sollen (Voreinstellung ist die aktuelle Umgebung, meist also der Workspace), während man mit `environment()` nach der Umgebung einer Funktion fragt.

Diese für manche komplexe Funktionen sehr nützliche Eigenschaft des *Lexical Scoping* wird z.B. sinnvoll in dem Paket **scatterplot3d** (Ligges und Mächler, 2003) eingesetzt (s. auch Kap. 8). Bei der enthaltenen gleichnamigen Grafikfunktion wird eine Projektion aus einem 3-dimensionalen Raum in einen 2-dimensionalen Raum durchgeführt. Wenn später Elemente zu einer so erzeugten Grafik hinzugefügt werden sollen, müssen einige Parameter der Projektion weiterhin bekannt sein. Das wird erreicht, indem Funktionen zum Hinzufügen von Elementen innerhalb von `scatterplot3d()` erzeugt und dann ausgegeben werden, die die Parameter aus dem zugehörigen `scatterplot3d()` Aufruf wegen der noch existierenden („verknüpften“) Umgebung weiterhin kennen.

Ein anderes anschauliches Beispiel zum Scoping in R wird durch die Eingabe von `demo(scoping)` vorgeführt.

Abschließend sei nochmals erwähnt, dass die verschiedenen *Scoping Rules* von R und S-PLUS beachtet werden müssen, wenn komplexere Funktionen, die (wenn auch nur implizit) von den Scoping Rules Gebrauch machen, in beiden Programmen benutzt werden sollen.

Namespaces

Zusätzliche Regeln zu der oben beschriebenen üblichen Suchreihenfolge in verschiedenen Umgebungen (*environments*) wurden durch die Einführung von *Namespaces* (Tierney, 2003) geschaffen. Da die Anzahl an Zusatzpaketen seit einiger Zeit schon rasant steigt und es daher unvermeidlich zu Konflikten zwischen Funktionen gleichen Namens in gleichzeitig benutzten Paketen kommen kann, wurde mit R-1.7.0 der Namespace Mechanismus eingeführt und seither erweitert.

Namespaces definieren, welche Objekte für den Benutzer und andere Funktionen (im Suchpfad) sichtbar sind und welche nur innerhalb des eigenen Namespace sichtbar sind. Funktionen, die aus einem Paket mit Namespace nicht explizit *exportiert* werden, sind nur für andere Funktionen innerhalb desselben Namespace sichtbar. Auf diese Weise können Funktionen, die nur zur Strukturiertheit dienen und nicht für Aufruf durch den Benutzer gedacht sind, auch „versteckt“ werden. Sie tragen damit nicht zur Unübersichtlichkeit bei.

Man schafft also durch Namespaces eine gewisse Unabhängigkeit von Objektnamen innerhalb des jeweiligen Namespace von anderen Namespaces. Bei Funktionen, die von außen benutzt werden sollen, muss aber nach wie vor auf Eindeutigkeit der Benennungen geachtet werden.

Wer in einem Paket eine Funktion

```
beispiel <- function(x)
    sin(2 * pi * x)
```

schreibt, erwartet vermutlich, dass die darin verwendeten Objekte `sin()` und `pi` aus dem Paket **base** stammen. Gleich lautende Funktionen in anderen Paketen oder im Workspace würden sie aber wegen der Scoping Rules „überschreiben“, wie z.B. in:

```
> beispiel <- function(x)
+     sin(2 * pi * x)
> beispiel(1:5)         # erwartetes Ergebnis:
[1] -2.449213e-16 -4.898425e-16 . . . .
> sin <- sum
> pi <- 0.5
> beispiel(1:5)         # Summe der Zahlen 1:5
[1] 15
```

Durch Namespaces wird sichergestellt, dass keine Objekte des Pakets **base** maskiert werden. Ebenso kann man auch Objekte aus beliebigen anderen Paketen *importieren*, die dann ihrerseits nicht mehr durch neue Objekte, z.B. im Workspace, maskiert werden können. Pakete, die nur durch im Namespace angegebene *Imports* geladen werden, werden nicht in den Suchpfad gehängt.

Eine in einem Namespace definierte Funktion sucht nach Objekten gemäß dem auf S. 77 angegebenen Suchpfad – mit folgender Ausnahme: Zuallererst wird im eigenen Namespace gesucht, dann in den in den Namespace importierten Objekten, danach im **base** Paket und danach erst in dem oben angegebenen Suchpfad.

Für expliziten Zugriff auf ein Objekt in einem konkreten Paket kann der Operator „`::`“ benutzt werden, der den Namen des Namespace von dem Objektnamen trennt. So greift man z.B. mit `stats::ks.test` auf das Objekt (Funktion) `ks.test` in Namespace `stats` zu.

In seltenen Fällen, z.B. um Fehler in einem Paket zu finden oder ein Paket zu entwickeln, möchte man auf eine nicht aus einem Namespace exportierte Funktion zugreifen, was durch die Funktion `getFromNamespace()` ermöglicht wird. Auch der Operator „`:::`“ bietet Zugriff auf nicht exportierte Objekte eines Namespace (Benutzung analog zu „`::`“).

Die Funktion `getS3method()` bietet die Möglichkeit, direkt auf eine zu einer generischen Funktion gehörenden Methoden-Funktion (nach `S3` Standard, s. Kap. 6) zuzugreifen, die nicht aus einem Namespace exportiert wird. Da Benutzer stets generische Funktionen aufrufen sollten, werden Methoden-Funktionen nur in Ausnahmefällen aus einem Namespace exportiert. Durch einen Registrierungsmechanismus (s. Abschn. 10.6) werden die Methoden-Funktionen den generischen Funktionen bekannt gemacht.

Mit `fixInNamespace()` kann das Ersetzen einer nicht exportierten Funktion geschehen. Abschließend sei die Funktion `getAnywhere()` erwähnt, die alle Objekte innerhalb des Suchpfads und in geladenen Namespaces findet, auch wenn das gesuchte Objekt nicht exportiert wird bzw. der geladene Namespace nicht im Suchpfad eingehängt ist.

Für weitere Details zu Namespaces verweise ich auf Tierney (2003). Das Einrichten eines Namespace für ein selbst definiertes Paket, bzw. das Ändern von Namespaces anderer Pakete, wird in Abschn. 10.6 beschrieben.

`assign()` *und* `get()`

Mit der Funktion `assign()` können Objekte in beliebigen existierenden Umgebungen erzeugt werden. Als Spezialfall erzeugt der Operator „`<<-`“ Objekte immer in der `.GlobalEnv` Umgebung. Man sollte ihn nur benutzen, wenn man sich wirklich über alle Konsequenzen im Klaren ist, insbesondere ist seine Verwendung für das alltägliche Programmieren i.A. unnötig. Das folgende Beispiel dient ausschließlich der Demonstration:

```
> zuweisen <- function(){
+     a <- 1                # lokale Zuweisung
+     b <<- 2               # Zuweisung in die .GlobalEnv
+     assign("d", 3, pos = ".GlobalEnv")
+     return(c(a, b, d))    # a lokal; b und d aus dem Workspace
+ }
> zuweisen()
 [1] 1 2 3
> print(a)                  # a ist nicht im Workspace
 Error in print(a) : Object "a" not found
> print(b)
 [1] 2
> print(d)
 [1] 3
```

Ohne Angabe des Arguments `pos` erzeugt `assign()` das gegebene Objekt in der aktuellen Umgebung. Analog zu `assign()` kann mit `get()` auf Objekte in beliebigen Umgebungen zugegriffen werden.

Eine häufig auf den Mailinglisten gestellte Frage ist, wie viele Objekte mit gleich strukturierten Namen (z.B.: `Objekt1`, `Objekt2`, `Objekt3`, ...) automatisch erzeugt werden können und wie man darauf wieder zugreift. Auch hier helfen `assign()` und `get()`, denn man kann darin die Objektnamen aus Zeichenketten zusammensetzen, wie das folgende Beispiel zeigt:

```
> for(i in 1:3)
+     assign(paste("Buchstabe", i, sep = ""), LETTERS[i])
> ls()                                    # Objekte im Workspace
 [1] "Buchstabe1" "Buchstabe2" "Buchstabe3" "i"
```

```
> Buchstabe2
 [1] "B"
> for(i in 3:1)
+     print(get(paste("Buchstabe", i, sep = "")))
 [1] "C"
 [1] "B"
 [1] "A"
```

Meist ist es jedoch wesentlich einfacher und sehr empfehlenswert, statt einer Vielzahl von Objekten besser deren Inhalt als Elemente einer einzigen Liste zu verwalten (s. Abschn. 2.9.4). Bei dem oben gezeigten Beispiel könnte Gleiches auch effizienter ohne Schleifen erreicht werden, es dient hier aber als Anschauung für komplexere Aufgaben.

4.4 Umgang mit Fehlern

Die Tatsache, dass jeder Fehler macht, der eigene Programme oder Funktionen schreibt, lässt sich nicht verleugnen. Bei sehr kurzen und einfachen Funktionen findet man den oder die Fehler meist (aber leider nicht immer) recht schnell. In Abschn. 4.4.1 sollen neben einem allgemeinen Vorgehen zur Fehlersuche und -beseitigung auch Werkzeuge und Strategien gezeigt werden, die den Anwender oder Programmierer beim Auffinden von Fehlern möglichst komfortabel unterstützen – auch in langen und komplexen Funktionen.

Das Abfangen und Behandeln von Fehlern, die erwartet werden, wird in Abschn. 4.4.2 erläutert.

4.4.1 Finden und Beseitigen von Fehlern – *Debugging*

Allgemeines Vorgehen

Wie auch in anderen Programmiersprachen kann man durch Ausgabe kurzer Texte (z.B. mit `cat()`, s. Abschn. 2.11) auf die Konsole anzeigen, welche Stellen die Funktion bereits fehlerfrei passiert hat, so dass der Fehler eingegrenzt werden kann. Auch das Anzeigen (z.B. mit `print()`) von Objekten, bei denen der Fehlerverursacher vermutet wird, ist oft nützlich.

Niemals sollte man mehr als 2-10 Zeilen Code schreiben, ohne diese einzeln in der Konsole daraufhin zu überprüfen, dass sie das erwartete Ergebnis produzieren. Immer wieder sollte überprüft werden, ob das bisher Geschriebene im Zusammenhang noch funktioniert, bevor Neues hinzugefügt wird. Vielmehr sollte man recht kurze Funktionen schreiben, die sich aufrufen, und somit modular programmieren, was auch eher der Herangehensweise an objektorientiertes Programmieren entspricht.

Tabelle 4.1. Funktionen zur Fehlersuche und Schlüsselwörter für den Browser

Funktion, Schlüsselwort	Beschreibung
`browser()`	der Browser wird explizit gestartet
`debug()`, `undebug()`	eine Funktion wird komplett im Browser ausgeführt
`debugger()`	Durchsuchen von Umgebungen, z.B. solchen, die `dump.frames()` in `last.dump` schreibt
`dump.frames()`	alle Umgebungen werden zur Fehlerzeit gespeichert
`recover()`	im Fehlerfall kann der Benutzer in eine zu untersuchende Umgebung springen
`trace()`, `untrace()`	Code für die Fehleranalyse dynamisch in Funktionen integrieren
`traceback()`	Anzeige des Pfads zu der fehlermeldenden Funktion durch alle Umgebungen hindurch
`c`	weitere Ausdrücke ohne Unterbrechung ausführen
`n`	den Ausdruck auswerten und zum nächsten springen
`where`	aktuelle Schachtelung von Umgebungen anzeigen
`Q`	den Browser beenden

Das Strukturieren von Code durch Leerzeilen und Einrückungen (z.B. bei Konstrukten), wie sie bereits in den Beispielen dieses Buches gesehen werden konnten, das Einfügen von Kommentaren sowie die Benutzung von Leerzeichen um Zuweisungssymbole und Operatoren erhöht die Lesbarkeit und Übersichtlichkeit drastisch und hilft somit auch bei der Fehlersuche.

Werkzeuge

Da einfache Hilfsmittel, wie etwa die Ausgabe von Informationen auf die Konsole, nicht immer schnell zum Ziel führen, gibt es einige Werkzeuge, die die Fehlersuche vereinfachen (s. Tabelle 4.1). Generell kann dynamisch Code zur Fehlersuche ohne Änderung an der eigentlichen Funktion mit Hilfe von `trace()` hinzugefügt werden.

Die Funktion `traceback()` zeigt an, welche Funktion den letzten Fehler verursacht hat und gibt auch den „Pfad“ der Funktionsaufrufe bis dorthin an. So kann in verschachtelten Strukturen der Schuldige gefunden werden:

```
> foo2 <- function(x)
+     x[-7] + 5
> foo1 <- function(x)
+     foo2(x)
> foo1(1:5)
 Error in foo2(x) : subscript out of bounds
```

```
> traceback()
 2: foo2(x)
 1: foo1(1:5)
```

In diesem Fall wird innerhalb von `foo1()` eine weitere Funktion `foo2()` aufgerufen, in der ein Fehler auftritt. Der Fehlermeldung sieht man zwar an, dass `foo2()` die Schuldige ist, bei komplexen mehrfach verschachtelten Funktionen ist aber oft gar nicht klar, von welcher anderen Funktion `foo2()` aufgerufen wurde. Hier zeigt `traceback()`, dass zunächst `1: foo1(1:5)` und darin `2: foo2(x)` aufgerufen wurde.

Mit `debug(foo2)` wird die Funktion `foo2()` ab sofort immer im Browser (s.u.) ausgeführt, bis `undebug(foo2)` diesen Modus beendet.

```
> debug(foo2)
> foo1(1:5)
 debugging in: foo2(x)
 debug: x[-7] + 5
Browse[1]> ls()          # Welche Objekte gibt es?
 [1] "x"
Browse[1]> x             # Die Definition von x ansehen
 [1] 1 2 3 4 5
Browse[1]> where         # An welcher Stelle sind wir noch gerade?
 where 1: foo2(x)
 where 2: foo1(1:5)

Browse[1]> n             # Mit 'n' wird der Schritt ausgeführt
 Error in foo2(x) : subscript out of bounds

> undebug(foo2)          # Beim nächsten Aufruf nicht mehr debuggen
```

Man erkennt schon an diesem einfachen Beispiel die Funktionsweise des *Browsers*: Nachdem `debug(foo2)` gesetzt ist (`foo2` wurde mit `traceback()` als Fehlerursache ermittelt) und `foo2(x)` innerhalb von `foo1()` aufgerufen worden ist, wird `foo2(x)` nicht normal ausgeführt, sondern Ausdruck für Ausdruck abgearbeitet. Es wird der nächste auszuführende Ausdruck angezeigt (hier zuerst: `debug: x[-7] + 5`) und eine neue Browser-Umgebung (für jeden Ausdruck) geschaffen, die durch den auf der Konsole erscheinenden Text `Browse[1]>` angezeigt wird. Der Ausdruck ist aber noch nicht ausgeführt worden, denn die Ausführung könnte ja bereits zu einem Fehler führen. Vielmehr kann man nun in der aktuellen Umgebung der Funktion „browsen“. Insbesondere kann man sich, z.B. mit `ls()`, alle darin definierten Objekte anschauen und diese Objekte auch auf der Konsole ausgeben lassen. In unserem Beispiel sieht man, dass `x` nur 5 Elemente hat, im nächsten Schritt aber das siebte (nicht existierende) Element entfernt werden soll.

Das Schlüsselwort (s. Tabelle 4.1 auf S. 85) `where` bewirkt, dass die aktuelle (zum Zeitpunkt des Aufrufs) Schachtelung von Umgebungen angezeigt wird. Durch Eingabe von `n` wird der Ausdruck dann ausgeführt und ggf. zum nächsten Ausdruck gesprungen. Die Eingabe von `c` (continue) im Browser bewirkt, dass alle weiteren Ausdrücke ohne erneute Generierung einer Browser-Umgebung und ohne Unterbrechung abgearbeitet werden. Der Browser kann mit `Q` direkt beendet werden.

Wenn innerhalb einer Funktion der Aufruf `browser()` erfolgt, wird an dieser Stelle der Browser gestartet. Man muss also nicht unbedingt mit „`debug()`“ eine lange Liste von Programmzeilen über sich ergehen lassen, bevor man an eine interessante (vermutlich fehlerhafte) Stelle kommt, sondern kann direkt vor einer solchen Stelle die Zeile `browser()` einfügen.

Mit der Einstellung `options(error = recover)`, wird im Fehlerfall die Funktion `recover()` ausgeführt. Diese startet den Browser so, dass eine beliebige zu untersuchende Umgebung aus den durch verschachtelte Funktionsaufrufe erzeugten Umgebungen (bei Auftreten des Fehlers) ausgewählt werden kann:

```
> options(error = recover)
> foo1(1:5)
 Error in foo2(x) : subscript out of bounds

 Enter a frame number, or 0 to exit

 1:foo1(1:5)
 2:foo2(x)

Selection: 2
 Called from: eval(expr, envir, enclos)
Browser[1]> Q
```

In unserem Fall springt man durch Eingabe von `2` also in die zur Funktion `foo2()` gehörende Umgebung, man könnte aber auch in die Umgebung von `foo1()` springen, um zu überprüfen, ob die richtigen Objekte für die Übergabe als Argumente an `foo2()` erzeugt wurden.

Eine Alternative zu `recover()` gibt es mit der Funktion `dump.frames()`, die auch mit `options(error = dump.frames)` zur Fehlerverarbeitung eingestellt werden kann. Sie hat den Vorteil, dass nicht bei jedem Fehler ein Browser gestartet wird, denn sie speichert gemäß Voreinstellung alle zur Fehlerzeit existierenden Umgebungen in einem Objekt `last.dump`. In diesem Objekt können mit `debugger()` wie folgt die Umgebungen durchsucht werden:

```
> options(error = dump.frames)
> foo1(1:5)
 Error in foo2(x) : subscript out of bounds
```

```
> debugger()
 Message:  Error in foo2(x) : subscript out of bounds
 Available environments had calls:
 1: foo1(1:5)
 2: foo2(x)

 Enter an environment number, or 0 to exit  Selection: 2
 Browsing in the environment with call:
    foo2(x)
 Called from: debugger.look(ind)
Browse[1]> ls()
 [1] "x"
Browse[1]> Q
```

Der Nachteil von `dump.frames()` ist, dass die gespeicherten Umgebungen u.U. sehr große Objekte enthalten, die viel Platz im Hauptspeicher verbrauchen.

4.4.2 Fehlerbehandlung

Es kommt durchaus vor, dass eine Funktion in einer anderen aufgerufen wird und je nach Übergabe von Argumenten einen Fehler verursachen kann. In dem Fall wird unter normalen Umständen sofort auch die äußere Funktion beendet und ein Fehler zurückgegeben, wie z.B. in dem erweiterten Beispiel des letzten Abschnitts:

```
> foo2 <- function(x)
+     x[-7] + 5
> foo1 <- function(x){
+     wert <- foo2(x)
+     cat("Ich werde immer noch ausgeführt...\n")
+     return(wert)
+ }
> foo1(1:5)
 Error in foo2(x) : subscript out of bounds
```

In komplexeren Situationen (z.B. bei Optimierungsverfahren) kann es vorkommen, dass sich die Fehler selbst durch geschickte Überprüfung der Argumente vor Aufruf der problematischen Funktion (`foo2()`) nicht vermeiden lassen. Dann möchte man aber die äußere Funktion (`foo1()`) im Fehlerfall nicht beenden lassen, sondern den Fehler abfangen können. Dabei hilft die Funktion `try()`, die einfach den Wert des als Argument übergebenen Ausdrucks enthält, falls kein Fehler auftritt. Im Fehlerfall des übergebenen Ausdrucks gibt sie ein Objekt der Klasse `try-error` zurück, das die Fehlermeldung enthält. Die umgebene Funktion wird dann aber fortgesetzt:

```
> foo2 <- function(x)
+     x[-7] + 5
> foo1 <- function(x){
+     wert <- try(foo2(x))
+     cat("Ich werde immer noch ausgeführt...\n")
+     if(class(wert) == "try-error")
+         wert <- "Ein Fehler ist aufgetreten"
+     return(wert)
+ }

> foo1(1:5)                          # Fehlerfall:
 Error in foo2(x) : subscript out of bounds
 Ich werde immer noch ausgeführt...
 [1] "Ein Fehler ist aufgetreten"

> foo1(1:10)                         # Ohne Fehler:
 Ich werde immer noch ausgeführt...
 [1]  6  7  8  9 10 11 13 14 15
```

Als Alternative zu `try()` bieten sich für die Fehlerbehandlung die etwas mächtigere Funktion `tryCatch()` und einige auf der Hilfeseite `?tryCatch` beschriebene zugehörige Funktionen an.

4.5 Rekursion

Eine in einigen höheren Programmiersprachen (aber leider nicht in allen Statistiksprachen) übliche Konstruktion ist die Rekursion, d.h. eine Funktion kann sich selbst aufrufen. Die Berechnung der Fibonacci-Zahlen (s. z.B. Bronstein et al., 2000) ist ein bekanntes Beispiel für rekursive Programmierung:

```
> fib.rek <- function(n){
+     if(n == 1) return(0)
+     else if(n == 2) return(1)
+     else return(fib.rek(n - 1) + fib.rek(n - 2))
+ }
> fib.rek(10)
 [1] 34
```

Der Funktion `fib.rek()` muss als Argument der Index der gewünschten Fibonacci-Zahl übergeben werden, danach schreibt man lediglich die Definition der Fibonacci-Zahlen als R Programm auf. Wenn der Index 1 oder 2 ist, so ist das Ergebnis laut Definition 0 bzw. 1 (und es folgen keine weiteren Aufrufe), andernfalls ist das Ergebnis die Summe der beiden vorhergehenden Fibonacci-Zahlen. Sollte der Index größer als 2 sein, wird die Funktion also selbst wieder aufgerufen, jedoch mit entsprechend kleineren Indizes.

Rekursionstiefe

Zu Problemen mit zu großer Rekursionstiefe kann es u.a. wegen der *Scoping Rules* kommen. Man bedenke, dass ein Funktionsaufruf eine eigene Umgebung kreiert, die erst bei Beenden der Funktion wieder gelöscht wird. Da eine Funktion bei der Rekursion sich selbst immer wieder aufruft und eine neue Instanz erzeugt, ohne dass die vorherige beendet ist, wird also in jedem Rekursionsschritt eine *zusätzliche* Umgebung erzeugt. Wenn viele und/oder große Objekte innerhalb der Funktion erzeugt werden oder an sie übergeben werden, so summiert sich der Speicherverbrauch also über alle Rekursionsschritte, auch wenn die Objekte z.T. identisch sind!

Während dieses Problem bei Berechnung der Fibonacci-Zahlen nahezu vernachlässigt werden kann, kann bei Verwendung eines großen Datensatzes schon bei einer sehr geringen Rekursionstiefe der Hauptspeicher eines modernen Rechners zu klein werden. Des Weiteren wird natürlich auch Laufzeit für die Speicheroperationen benötigt.

Bei den Fibonacci-Zahlen zeigt sich bei größerer Rekursionstiefe ein anderes Problem: Die Funktion ruft sich zunächst zweimal selbst auf, jeder neue Aufruf löst aber wieder zwei Aufrufe aus usw. Die Komplexität des Algorithmus ist also $O(n) = 2^n$.

Man sollte insgesamt nur Probleme rekursiv programmieren, die eine überschaubare Rekursionstiefe haben. Wenn möglich, ist iteratives Programmieren angebrachter, gerade bei Funktionen, die häufiger verwendet werden sollen. Wenn es bei einer Rekursion zu den beschriebenen Speicher- bzw. Laufzeitproblemen kommt, sollte auf jeden Fall versucht werden, eine iterative Lösung zu finden oder die Rekursion z.B. in C (s. Kap. 9) auszulagern.

Zum Vergleich wird die iterative Variante zur Erzeugung der Fibonacci-Zahlen an dieser Stelle dargestellt:

```
> fib.it <- function(n){
+     FZ <- numeric(n)                  # x: Platz für alle Fib.-Z.
+     FZ[1:2] <- 0:1                    # FZ[1] und FZ[2] laut Def.
+     for(i in seq(along = FZ)[-(1:2)]) # FZ[1] und FZ[2] auslassen
+         FZ[i] <- FZ[i-1] + FZ[i-2]    # laut Definition
+     return(FZ[n])
+ }
> fib.it(10)
 [1] 34
```

4.6 Umgang mit Sprachobjekten

Da in R alles ein Objekt ist (s. Abschn. 2.3), können auch Elemente der Sprache, wie etwa Funktionsaufrufe, Aufrufe, Ausdrücke und Zuweisungen, als Objekte behandelt und bearbeitet werden.

Für die alltägliche Benutzung von R für die Datenanalyse und Visualisierung ist der Umgang mit Werkzeugen zur Bearbeitung von Sprachobjekten zwar meist unnötig, trotzdem gewinnt man beim Studium der Möglichkeiten mehr Einsicht in die Sprache. Wer beim Schreiben eigener Pakete ein paar Tricks anwenden möchte, kommt nicht um den expliziten Umgang mit Sprachobjekten herum.

Was man in R auch tut, es werden ständig Aufrufe (*calls*, von Funktionen) und Ausdrücke (*expressions*, ein oder mehrere zusammenhängende Aufrufe) verwendet, wie etwa in:

```
> round(12.3456, digits = 2)
 [1] 12.35
```

Es gibt u.U. aber Situationen, in denen ein solcher Aufruf nicht explizit angegeben werden kann oder soll, sondern in denen er erst konstruiert werden muss. Das kann beispielsweise der Fall sein beim Zusammensetzen von Formeln (s. Abschn. 7.4) oder bei Konstruktion mathematischer Beschriftung in Grafiken und Legenden (s. Kap. 8).

Einen solchen, noch nicht ausgewerteten Aufruf kann man mit `call()` aus einzelnen Elementen zusammensetzen und ihn dann explizit mit `eval()` auswerten:

```
> (mycall <- call("round", 12.3456, digits = 2))
 round(12.3456, digits = 2)
> mode(mycall)
 [1] "call"
> eval(mycall)
 [1] 12.35
```

Ein direktes Zusammensetzen und Auswerten geschieht mit `do.call()`:

```
> do.call("round", list(12.3456, digits = 2))
 [1] 12.35
```

Hierbei müssen die Argumente des Funktionsaufrufs als Liste übergeben werden.

Mit `quote()` kann man einen Aufruf vor der Auswertung schützen:

```
> quote(round(12.3456, digits = 2))
 round(12.3456, digits = 2)
```

Eine Zeichenfolge, die einen auszuwertenden Ausdruck als Textrepräsentation enthält, kann mit `parse()` in einen auswertbaren Ausdruck umgewandelt werden, so dass dieser letztendlich bearbeitet oder mit `eval()` ausgewertet werden kann:

```
> (zeichen <- "round(12.3456, digits = 2)")
 [1] "round(12.3456, digits = 2)"
> mode(zeichen)
 [1] "character"
> (ausdruck <- parse(text = zeichen))
 expression(round(12.3456, digits = 2))
> mode(ausdruck)
 [1] "expression"
> eval(ausdruck)
 [1] 12.35
```

Ein Ausdruck ist eine zusammenhängende Folge von Aufrufen, oder genauer, eine Liste von Aufrufen. Im vorigen Beispiel handelt es sich also um einen aus einem Aufruf bestehenden Ausdruck, einer einelementigen Liste. Ein solcher Ausdruck (hier aus zwei Aufrufen) lässt sich mit `expression()` erzeugen:

```
> (myexpr <- expression(x <- 12.3456, round(x, digits = 2)))
 expression(x <- 12.3456, round(x, digits = 2))
> mode(myexpr)
 [1] "expression"
> eval(myexpr)
 [1] 12.35
> (call1 <- myexpr[[1]])    # erstes Element (Aufruf) der Liste
 x <- 12.3456
> mode(call1)
 [1] "call"
> print(eval(call1))
 [1] 12.3456
```

Das Gegenstück zur Funktion `parse()`, womit eine Zeichenkette in einen Ausdruck umgewandelt wird, ist `deparse()`. Diese Funktion wandelt einen noch nicht ausgewerteten Ausdruck in eine Zeichenkette um:

```
> deparse(call("round", 12.3456, digits = 2))
 [1] "round(12.3456, digits = 2)"
```

Das ist z.B. dann hilfreich, wenn der Aufruf zu Dokumentationszwecken zusammen mit seinem Ergebnis gespeichert werden soll, was etwa bei `lm()` (Bearbeitung linearer Modelle, s. Abschn. 7.5) und ähnlichen Funktionen üblich ist. In solchen Fällen wird `deparse()` zusammen mit `substitute()` eingesetzt und wirkt wegen der verzögerten Auswertung von Argumenten (*Lazy Evaluation*, s. Abschn. 4.2) wie folgt:

```
> foo <- function(x)
+     return(list(Aufruf = deparse(substitute(x)), Ergebnis = x))
> foo(sin(pi / 2))
 $Aufruf
 [1] "sin(pi/2)"

 $Ergebnis
 [1] 1
```

Häufig wird dieser Trick auch als Voreinstellung zur Beschriftung von Achsen in Grafiken benutzt.

Die Funktion `substitute()` kann jedoch viel mehr. Als erstes Argument erwartet sie einen Ausdruck, in dem sie alle Symbole ersetzt, die in ihrem zweiten Argument `env` spezifiziert sind, wobei die aktuelle Umgebung als Voreinstellung genommen wird. Dieses zweite Argument kann eine Umgebung sein oder eine Liste von Objekten, deren Elementnamen die Zuordnung bestimmen. Eine Ausnahme ist die Umgebung `.GlobalEnv`, deren Symbole nicht als Voreinstellung zur Ersetzung herangezogen werden. Häufig wird diese Funktion zur dynamischen mathematischen Beschriftung in Grafiken eingesetzt (s. Abschn. 8.1.5), wenn nicht ausgewertete Ausdrücke an die Grafikfunktion übergeben werden müssen, in die man jedoch zur Laufzeit Zahlenwerte einsetzen möchte:

```
> lambda <- pi / 2          # sei aus einer Berechnung
> (mycall <- substitute(sin(lambda), list(lambda = lambda)))
 sin(1.57079632679490)
> mode(mycall)
 [1] "call"
> eval(mycall)
 [1] 1
```

Einen sehr guten Artikel („Mind Your Language“) zum angewandten Umgang mit Sprachobjekten hat Venables (2002) verfasst. Weitere Details findet man auch in Venables und Ripley (2000).

4.7 Vergleich von Objekten

Im Gegensatz zu dem Vergleichsoperator „`==`“, der elementweise Vergleiche durchführt, bietet sich die Funktion `identical()` für den Vergleich zweier beliebiger Objekte auf exakte Gleichheit an. Im folgenden Beispiel wird deutlich, wie unterschiedlich der Vergleich der Vektoren `x` und `y`, die nicht den gleichen Datentyp haben, ausfallen kann:

```
> x <- c(1, 2, 3)
> y <- c("1", "2", "3")
> x == y
 [1] TRUE TRUE TRUE
> all(x == y)
 [1] TRUE
> identical(x, y)
 [1] FALSE
> identical(x, as.numeric(y))
 [1] TRUE
```

Häufig sollen Objekte aber gar nicht auf exakte Gleichheit, sondern auf Gleichheit im Sinne der Rechengenauigkeit hin überprüft werden. Wird z.B. die Matrix X

```
> (X <- matrix(c(3, 4, 8, 7, 1, 8, 2, 9, 9), 3))
     [,1] [,2] [,3]
[1,]    3    7    2
[2,]    4    1    9
[3,]    8    8    9
```

invertiert, so erhält man bei Berechnung von XX^{-1} mit

```
> X %*% solve(X)
              [,1]          [,2]          [,3]
[1,] 1.000000e+00 1.110223e-16 5.551115e-17
[2,] 2.220446e-16 1.000000e+00 0.000000e+00
[3,] 4.440892e-16 0.000000e+00 1.000000e+00
```

fast die Einheitsmatrix bei einer maximalen Abweichung von nicht mehr als 10^{-15}. Die Routinen zum Invertieren einer Matrix sind numerisch recht komplex und führen wegen Darstellungsproblemen mit Gleitkommazahlen (Lange, 1999) in diesem Beispiel zu Fehlern. Überprüft man nun das Ergebnis auf Gleichheit mit der Einheitsmatrix (`diag(3)`), so liefern `identical()` bzw. „`==`“:

```
> identical(diag(3), X %*% solve(X))
 [1] FALSE
> diag(3) == X %*% solve(X)
      [,1]  [,2]  [,3]
[1,]  TRUE FALSE FALSE
[2,] FALSE FALSE  TRUE
[3,] FALSE  TRUE  TRUE
```

Die Funktion `all.equal()` hilft hier weiter, denn sie überprüft auf Gleichheit im Sinne der Rechengenauigkeit:

```
> all.equal(diag(3), X %*% solve(X))
 [1] TRUE
> all.equal(3, 3.1)
 [1] "Mean relative  difference: 0.03333333"
```

Wegen der analysierenden Ausgabe von `all.equal()` in Fällen, in denen keine Gleichheit vorliegt, sollte innerhalb von Funktionen die Überprüfung zweier Objekte `x` und `y` auf Gleichheit im Sinne der Rechengenauigkeit mit

```
> isTRUE(all.equal(x, y))
```

vorgenommen werden. Die Funktion `isTRUE()` liefert `TRUE` zurück, wenn auch ihr Argument den Wert `TRUE` hat, sonst aber `FALSE`.

5

Effizientes Programmieren

Vektorwertiges Programmieren möge der Leser als Gesetz ansehen! Warum das so ist, wird sich im weiteren Verlauf dieses Kapitels zeigen.

Sowohl die Funktionen, die man sehr häufig verwenden oder veröffentlichen möchte, als auch diejenigen, die mit großen Datenbeständen oder als Bestandteil von (z.T. längeren) Simulationen großen Einfluss auf die Rechenzeit haben, sollten hinsichtlich ihrer Geschwindigkeit und ihres Speicherverbrauchs optimiert werden. Der Begriff „Effizienz“ wird von der Informatik geprägt und wird für Algorithmen und Programme verwendet, die für kaum weiter optimierbar gehalten werden.

Man bedenke, dass manchmal eine simple Verdoppelung der Geschwindigkeit bedeutet, dass Programme statt zweier Tage nur noch einen Tag Rechenzeit benötigen, oder dass Berechnungen statt über Nacht auch eben während der Mittagspause laufen können. Eine Verringerung des Speicherverbrauchs bringt auch Geschwindigkeit, vor allem, wenn vermieden werden kann, dass virtueller Speicher verwendet werden muss. Virtueller Speicher ist Speicher, der auf der Festplatte in s.g. *Swap*-Bereichen erzeugt wird. Er erweitert *virtuell* den Hauptspeicher des Rechners – mit dem Nachteil, dass er um ein Vielfaches (Größenordnung: 10^3) langsamer als echter Hauptspeicher ist. Manchmal kann eine Verringerung des Speicherverbrauchs das Lösen von Problemen erst möglich machen, wenn sonst z.B. gar nicht genügend Hauptspeicher zur Verfügung stünde.

Bei sehr großen Datensätzen ist es manchmal nötig, nur mit Teildatensätzen zu arbeiten, u.a. weil R alle Objekte im Hauptspeicher hält und sonst ein Problem mit der Speicherkapazität auftritt. Dann helfen Datenbanken weiter. Außerdem können triviale Rechenoperationen oft auch direkt vom Datenbankserver ausgeführt werden. Näheres zur Benutzung von Datenbanken findet man in Abschn. 3.5 sowie bei Ripley (2001b).

Es ist zu bedenken, dass einfache Arbeitsplatzrechner heute leicht auf 2–4 Gigabyte RAM aufgerüstet werden können. Im Zuge der Massenproduktion von 64-bit Prozessoren und der Entwicklung passender Betriebssysteme wird sich diese Grenze in den nächsten Jahren auch für sehr preisgünstige Rechner schnell weiter nach oben verschieben. Gerade bei moderaten Datensätzen ist eine Hauptspeicheraufrüstung eine Investition, die häufig viel andere Arbeit spart.

R ist eine interpretierte Sprache, d.h. die Auswertung von Code erfolgt erst zur Laufzeit. Der daraus entstehende Geschwindigkeitsnachteil kann z.T. dadurch ausgeglichen werden, dass möglichst vektorwertig programmiert wird und so Schleifen vermieden werden. Einige Tricks zum vektororientierten Programmieren und Hinweise, wann Schleifen vielleicht doch Sinn machen, werden in Abschn. 5.2 gegeben.

Hilfsmittel zur Effizienzanalyse werden in Abschn. 5.3 beschrieben. Sie können aufdecken, wo die Schwachstellen (der „Flaschenhals“) eines Programms liegen. Einige einfache Regeln für effizientes Programmieren kann man, solange es leicht fällt, gleich von Anfang an beachten. Andere Feinheiten müssen je nach Gegebenheit getestet werden, wenn eine Optimierung nötig erscheint. Sicherlich sollte man keine komplizierte Optimierung versuchen, solange noch nicht klar ist, ob man das gerade programmierte Verfahren später noch einsetzen möchte.

Als generelle Regeln sollte man beachten:

- *Vektorwertiges Programmieren möge der Leser als Gesetz ansehen!*
- Bereits implementierte Funktionen, wie etwa `optim()` (eine Sammlung von Optimierungsverfahren), greifen häufig auf vorhandenen schnellen C oder Fortran Code zurück. Hier sollte man i.d.R. das Rad nicht neu erfinden.
- Sehr zeitkritische Teile des Codes können, idealerweise nachdem sie auf Richtigkeit und Funktionalität überprüft worden sind, gut in C, C++ oder Fortran Code ausgelagert werden. R unterstützt, wie in Kap. 9 beschrieben, das Erstellen und Einbinden kompilierter (und dadurch schneller) Bibliotheken.
- Wenn das Optimieren nicht hilft, so bleibt als einziger Ausweg der Kauf eines größeren Computers (schneller, mehr Speicher) oder die Benutzung von Rechenclustern (Stichwort: MOSIX) und Multiprozessor-Rechnern. Einige Pakete bieten inzwischen Schnittstellen zu Werkzeugen, die paralleles Rechnen vereinfachen. An dieser Stelle seien die Pakete **Rmpi**[1] (Yu, 2002) als Interface zu MPI (*Message Passing Interface*) und **rpvm** (Li und

[1] `http://www.stats.uwo.ca/faculty/yu/Rmpi/`

Rossini, 2001) als Interface zu PVM (*Parallel Virtual Machine*) genannt sowie das darauf aufsetzende Paket **snow**[2] von Luke Tierney.

Es spielen aber auch Lesbarkeit und Verständlichkeit von Code und Dokumentation eine besonders große Rolle, gerade wenn auch andere mit den Funktionen arbeiten sollen und sie verändern müssen. Klarer Code und eindeutige Dokumentation führen nicht nur zu drastisch kürzeren Einarbeitungszeiten für andere Benutzer und Programmierer (s. Abschn. 5.1), sondern helfen auch dem ursprünglichen Programmierer, seine Funktionen nach gewisser Abstinenz selbst wieder zu verstehen.

5.1 Programmierstil

Guter Programmierstil ist recht schwierig zu definieren, es gehören aber sicherlich mehrere Punkte dazu. Auf folgende Punkte soll näher eingegangen werden:

- *Wiederverwendbarkeit:* Eine Funktion sollte möglichst allgemein geschrieben sein.
- *Nachvollziehbarkeit:* Eine Funktion sollte so viele Kommentare wie möglich enthalten, am besten eigene Dokumentation wie z.B. Hilfeseiten.
- *Lesbarkeit – Kompaktheit:* Sehr kompakt geschriebenen Code kann man schlecht verstehen, genauso wie zu wenig kompakten Code.
- *Lesbarkeit – Schrift:* Code sollte vom Schriftbild gut lesbar sein, d.h. es sollte mit Einrückungen bei Konstrukten und Leerzeichen für Übersichtlichkeit gesorgt werden.
- *Effizienz:* Bereits zu Anfang des Kapitels wurde erwähnt, dass Funktionen, die man sehr häufig verwenden will und Bestandteil von (längeren) Simulationen sind, hinsichtlich der Geschwindigkeit (und des Speicherverbrauchs) optimiert werden sollten.

Wiederverwendbarkeit

Eine Funktion sollte möglichst allgemein geschrieben sein, so dass sie nicht nur auf den aktuellen Datensatz, sondern auf beliebigen Datensätzen anwendbar ist. Sie sollte nicht nur genau *jetzt* und zu *diesem* Zweck, sondern auch bei ähnlichen Aufgabenstellungen (gleiche Aufgabenklasse) in Zukunft ohne (oder zumindest ohne starke) Anpassung verwendet werden können.

Wenn eine Funktion geschrieben wird, liegt meist eine sehr konkrete Problemstellung vor. Oft wird man mit einem aktuell vorliegenden Datensatz

[2] `http://www.stat.uiowa.edu/~luke/R/cluster/cluster.html`

Berechnungen anstellen oder ein sehr konkretes Phänomen simulieren wollen. Als triviales Beispiel stelle man sich eine einfache Würfelsimulation vor, bei der simuliert werden soll, wie häufig die Zahl 8 bei 100 Würfen eines Würfels mit 10 Seiten fällt (schlechter Stil!):

```
> wuerfel <- function(){
+     x <- sample(1:10, 100, replace = TRUE)
+     sum(x == 8)
+ }
> wuerfel()                 # von Zufallszahlen abhängig!
 [1] 9
```

Wenn sich nun die Frage nach der Häufigkeit der Zahl 6 bei 1000 Würfen eines Würfels mit 6 Seiten stellt, müsste die Funktion neu geschrieben werden. Stattdessen hätte man die Funktion mit entsprechenden Argumenten auch von Beginn an allgemein programmieren können – für einen Würfel mit beliebig vielen Seiten, beliebiger Augenzahl und für beliebig viele Würfe (guter Stil):

```
> wuerfel <- function(Seiten, N, Augenzahl){
+     # Generieren von N Würfen eines Würfels mit "Seiten" Seiten:
+     x <- sample(1:Seiten, N, replace = TRUE)
+     # Zählen, wie oft die Augenzahl "Augenzahl" vorkommt:
+     sum(x == Augenzahl)
+ }
> wuerfel(10, 100, 8)       # von Zufallszahlen abhängig!
 [1] 4
> wuerfel(6, 1000, 6)       # von Zufallszahlen abhängig!
 [1] 176
```

Dieses Beispiel ist natürlich stark vereinfacht. Prinzipiell wird man immer wieder feststellen, dass etwas für einen konkreten Fall programmiert wird. Das ist zunächst gut und oft die einzige Möglichkeit, ein Programm zu beginnen. Man möge aber nicht den Schritt zur Verallgemeinerung des Codes vergessen, der meistens später zur Arbeitserleichterung verhilft, weil Funktionen nur so wieder verwendet werden können. Eine Verallgemeinerung ist viel leichter, wenn eigene Ideen, die während des Programmierens entstanden sind, noch bekannt sind.

Nachvollziehbarkeit

Eine Funktion sollte so viele Kommentare wie möglich enthalten, am besten sogar eigene Dokumentation wie z.B. Hilfeseiten. Nur so kann man selbst nach einigen Wochen oder gar Jahren den Code noch verstehen. Vor allem auch andere können nur kommentierte und dokumentierte Funktionen einfach benutzen und deren Code so nachvollziehen und ändern. Bei der Anfertigung von

Hilfeseiten wird man von dem System zur Paketerstellung (s. Abschn. 10.7) von R unterstützt.

Sollte man Funktionen anderer Programmierer lesen oder eigenen Code, der vor Monaten oder Jahren geschrieben wurde, erneut bearbeiten müssen, so gelten provokativ formuliert die folgenden Sätze:

- *Satz 1:* Kein Programmierer kommentiert seinen Code ausreichend detailliert.
- *Satz 2:* Sollte eine Funktion erstaunlicherweise kommentiert sein, so sind die Kommentare so knapp und unverständlich, dass der Programmierer sie schon fast hätte weglassen können.

Der Leser möge so oft es geht versuchen, Gegenbeispiele für diese Sätze bei seinen eigenen Funktionen zu erzeugen!

Lesbarkeit

Sehr kompakt geschriebenen Code kann man nur schlecht lesen und verstehen, weil in jeder Zeile zu viele Operationen enthalten sind, die man nicht gleich überblicken und entschlüsseln kann. Es macht daher, etwas überspitzt formuliert, wenig Sinn zu versuchen, ganze Programme in einer Zeile unterzubringen. Genauso ist aber auch zu wenig kompakter Code schlecht zu verstehen, denn man verliert schnell die Übersicht, wenn sehr viele Zeilen und sehr viele (z.T. temporäre) Variablen für einfache Operationen benötigt werden. Zwischen diesen beiden Punkten zur *Kompaktheit* von Code muss also ein Kompromiss gefunden werden.

Eine andere Art von Lesbarkeit wird vom *Schriftbild* verlangt, das für eine gute Übersichtlichkeit sorgen sollte. Es ist dringend anzuraten, dass je nach Kontext mit Einrückungen gearbeitet wird, die den Code strukturieren, so dass z.B. bei Funktionen, bedingten Anweisungen und Schleifen sofort klar wird, welche Codeteile zu einem bestimmten Konstrukt bzw. zu einer bestimmten Gliederungsebene gehören. Ebenso sollten alle Kommentare möglichst gleich weit eingerückt werden, so dass quasi das Bild einer Tabelle entsteht.

Auch mit Leerzeichen sollte für Übersichtlichkeit gesorgt werden, insbesondere um Zuweisungszeichen und Gleichheitszeichen herum sowie nach Kommata. Zeilen mit mehr als 60-80 Zeichen führen zur Unübersichtlichkeit und können nicht mehr gut in beliebigen Editoren dargestellt werden. Ganz generell sollte immer für Programmcode gelten, dass auf dem Bildschirm und im Ausdruck eine Schrift mit fester Zeichenbreite, z.B. *Courier New* unter Windows, verwendet wird, damit untereinander geordnete Programmteile auch exakt untereinander stehen.

Als Beispiele ziehe man beliebige Beispiele aus diesem Buch heran, bei denen stets auf gute Lesbarkeit geachtet wurde, wie etwa bei der Funktion

```
> wuerfel <- function(){
+     x <- sample(1:10, 100, replace = TRUE)
+     sum(x == 8)
+ }
```

die sicherlich niemand aus der folgenden, schlecht lesbaren Form entschlüsseln möchte:

```
> wuerfel<-function(){x<-sample(1:10,100,replace=TRUE)
+ sum(x==8)}
```

5.2 Vektorwertiges Programmieren und Schleifen

Es gilt vektorwertig zu programmieren, um Schleifen zu vermeiden. In diesem Abschnitt werden einige Hilfsmittel dazu vorgestellt. Schleifen sind aber nicht immer schlecht, wenn sie richtig eingesetzt und ein paar Regeln beachtet werden.

5.2.1 Sinnvolles Benutzen von Schleifen

In vielen Kommentaren zu R findet man Bemerkungen dazu, dass Schleifen generell „schlecht“ sind. Das ist so aber nicht wahr. Richtig ist, dass so oft wie möglich vektorwertig programmiert werden sollte, es jedoch einige Ausnahmen gibt.

Nicht jedes Problem kann ohne Schleifen bearbeitet werden. Es gibt auch Situationen, in denen eine Schleife schneller sein kann als ein vektorwertig programmiertes Problem. Das ist der Fall, wenn durch vektorwertiges Programmieren der Verbrauch an Speicherplatz den zur Verfügung stehenden Hauptspeicher sprengt und das Betriebssystem zum Auslagern von Speicher zwingt bzw. selbst das nicht mehr möglich ist. Hier macht es Sinn, doch eine Schleife zu benutzen, die das Problem in kleinere Teile zerlegt, die noch in den Hauptspeicher passen.

Schleifen sind dann sogar sehr angebracht, wenn eine rekursiv (s. Abschn. 4.5) programmierte Funktion durch eine iterativ programmierte ersetzbar ist.

Wenn man Schleifen benutzt, z.B. in den oben beschriebenen Situationen oder bei Simulationen, sollte man einige Regeln – wenn möglich – unbedingt beachten. Durch die folgenden Regeln können nämlich z.T. erhebliche Geschwindigkeitsgewinne erzielt werden (für einige Laufzeitvergleiche s. Abschn. 5.3):

- *Zu Beginn Objekte vollständig initialisieren, die pro Schleifendurchlauf wachsen.*
  Wenn einem Objekt, z.B. einem Vektor oder einer Liste, in jedem Schleifendurchlauf ein neues Element zugewiesen werden soll, sollte dieses Objekt vor Beginn der Schleife vollständig initialisiert werden, falls die endgültige Länge des Objekts bereits vor Beginn der Schleife bekannt ist. Wenn man es jedoch pro Schleifendurchlauf verlängert, muss jedes Mal erneut Speicher bereitgestellt werden und der Inhalt u.U. intern kopiert werden, wozu Rechenzeit benötigt wird. Anstelle von

  ```
  a <- NULL
  for(i in 1:n)
      a <- c(a, foo(i))   # langsam!
  ```

  (`foo()` sei hier eine beliebige Funktion, die ein passendes Objekt zurückliefert) sollte man also lieber schreiben:

  ```
  a <- numeric(n)
  for(i in 1:n)
      a[i] <- foo(i)      # schneller!
  ```

  Zur Initialisierung des Objekts `a` vom Typ `numeric` wurde hier die Funktion `numeric()` verwendet. Weitere komfortable Funktionen zur Erzeugung von Vektoren bestimmter Datentypen (s. auch Abschn. 2.8) sind `logical()`, `integer()`, `complex()` und `character()`. Der Datentyp ergibt sich aus dem jeweiligen Funktionsnamen.
  Eine allgemeine Funktion zur Erzeugung von Vektoren beliebigen Datentyps ist `vector()`. Der Aufruf `vector(mode = "list", length = n)` erzeugt eine (leere) Liste der Länge n.
- *Nicht in Schleifen unnötige Fehlerüberprüfungen durchführen.*
  Argumente sollte man auf ihre Richtigkeit vor einer Schleife und in der Schleife berechnete Ergebnisse nach Ende der Schleife möglichst vektorwertig überprüfen.
- *Keine Berechnung mehrfach ausführen – vor allem nicht in Schleifen.*
  Als sehr einfaches Beispiel sei die folgende Schleife für beliebige (passende) Objekte `a`, `n`, `pi` und `foo()` definiert:

  ```
  for(i in 1:n)
      a[i] <- 2 * n * pi * foo(i)
  ```

  Die Berechnung `2 * n * pi` wird hier in jedem Schleifendurchlauf ausgeführt. Man kann die Ausführung durch folgende Umstellung beschleunigen:

  ```
  for(i in 1:n)
      a[i] <- foo(i)
  a <- 2 * n * pi * a
  ```

Es werden nun drei Multiplikationen pro Schleifendurchlauf gespart, die anstelle von n-Mal nur am Ende einmal nötig sind.

5.2.2 Vektorwertiges Programmieren – mit `apply()` und Co

Vektorwertiges Programmieren möge der Leser als Gesetz ansehen!
Nicht nur wegen der häufigen Wiederholung des letzten Satzes wird inzwischen klar sein, dass vektorwertiges Programmieren meist schnellere Funktionen liefert als die Benutzung von Schleifen.

Viele vektorwertige Operationen sind direkt ersichtlich. Die üblichen komponentenweisen Rechenoperationen für Vektoren und Matrizen und einige in Tabelle 5.1 angegebene Operatoren und Funktionen (z.B. `%*%` oder `%o%`) wird niemand durch Schleifen ersetzen wollen, da Ihre Benutzung direkt klar ist. Auf einige andere Funktionen, deren Benutzung vielleicht weniger ersichtlich ist, wird hier näher eingegangen.

Tabelle 5.1. Operatoren und Funktionen für vektorwertiges Programmieren

Operator, Funktion	Beschreibung
`%*%`	Vektor- oder Matrixmultiplikation
`%o%`, `outer()`	äußeres Produkt
`%x%`, `kronecker()`	Kronecker-Produkt
`colSums()`, `rowSums()`	schnelle Spalten-, Zeilensummen
`colMeans()`, `rowMeans()`	schnelle Spalten-, Zeilenmittel
`apply()`	spalten- und zeilenweises Anwenden einer Funktion auf Matrizen bzw. Arrays
`lapply()`	elementweises Anwenden einer Funktion auf Listen, Datensätze und Vektoren
`sapply()`	wie `lapply()`, gibt „einfaches“ Objekt zurück
`mapply()`	multivariates `sapply()`
`tapply()`	Tabellen gruppiert nach Faktoren

Es soll nämlich auch vektorwertiges Arbeiten möglich sein, wenn die Lösung nicht sofort offensichtlich ist. So ist es wünschenswert, gewisse Funktionen auf einige Objekte (z.B. Vektoren, Matrizen, Arrays oder Listen) element-, spalten- oder zeilenweise anwenden zu können. Dabei helfen die Funktionen `apply()`, `lapply()`, `sapply()`, `mapply` und `tapply()` (s. auch Tabelle 5.1).

Matrizen und Arrays: `apply()`

Mit der Funktion `apply()` (engl.: anwenden) kann vektorwertig auf Matrizen bzw. Arrays gearbeitet werden, und zwar können geeignete Funktionen auf jede Spalte oder Zeile einer Matrix angewendet werden, ohne dass Schleifen eingesetzt werden müssen. Die allgemeine Form von `apply()` ist

```
apply(X, MARGIN, FUN, ...)
```

Als Argumente werden dabei mit `X` das Array bzw. die Matrix, mit `MARGIN` die beizubehaltende Dimensionsnummer (1 für Zeilen, 2 für Spalten usw.) und mit `FUN` die komponentenweise in ihrem ersten Argument anzuwendende Funktion erwartet. Außerdem können mit „`...`“ Argumente an die in `FUN` spezifizierte Funktion übergeben werden.

Als einfaches Beispiel werden zunächst die Maxima jeder Zeile einer Matrix sowie die Spannweiten jeder Spalte berechnet:

```
> (X <- matrix(c(4, 7, 3, 8, 9, 2), nrow = 3))
      [,1] [,2]
 [1,]    4    8
 [2,]    7    9
 [3,]    3    2
> apply(X, 1, max)
 [1] 8 9 3
> apply(X, 2, function(x) diff(range(x)))
 [1] 4 7
```

Im Fall der Spannweitenberechnung wird hier mit

```
function(x) diff(range(x))
```

eine s.g. *anonyme* (unbenannte) Funktion definiert, die nach Aufruf von `apply()`, weil sie keinem Objekt zugewiesen wurde, nicht mehr bekannt ist. Die Spannweite ist die Differenz (`diff()`) von Maximum und Minimum (gegeben durch `range()`), man muss also beide Funktionen kombinieren. Analog zum ersten Beispiel, in dem die Zeilen (`1`) von `X` an die Funktion `max()` übergegeben werden, werden im zweiten Beispiel die Spalten (`2`) von `X` an die anonyme Funktion weitergeleitet. Es könnte entsprechend auch eine längere, nicht anonyme Variante verwendet werden:

```
> spannweite <- function(x)
+     diff(range(x))
> apply(X, 2, spannweite)
 [1] 4 7
```

Data frames, Listen und Vektoren: `lapply()` *und* `sapply()`

Mit der Funktion `lapply()` (*l* für *l*ist) kann man eine geeignete Funktion elementweise mit hoher Geschwindigkeit auf Listen, Datensätze (*data frames*)

und Vektoren anwenden. Die Argumente, abgesehen vom nicht benötigten `MARGIN`, sind analog zu `apply()`. Es wird eine Liste ausgegeben, deren Länge der des bearbeiteten ursprünglichen Objekts entspricht.

Die Funktion `sapply()` (*s* für *s*implify) arbeitet völlig analog zu `lapply()` mit der Ausnahme, dass versucht wird, das auszugebende Objekt zu vereinfachen. Sind die auszugebenden Werte pro Element Skalare, so wird anstelle einer Liste ein Vektor von entsprechender Länge ausgegeben.

Als Beispiel wird hier der berühmte Datensatz von Anscombe (1973) zur linearen Regression verwendet (s. auch Abschn. 7.5):

```
> attach(anscombe)                        # Hinzufügen zum Suchpfad
> anscombe                                # Datensatz anschauen
    x1 x2 x3 x4    y1   y2    y3    y4
1  10 10 10  8  8.04 9.14  7.46  6.58
2   8  8  8  8  6.95 8.14  6.77  5.76
3  13 13 13  8  7.58 8.74 12.74  7.71
. . . . .
> ans.reg <- vector(4, mode = "list")   # leere Liste erzeugen
> # 4 Regressionen (y_i gegen x_i) in Liste speichern:
> for(i in 1:4){
+     x <- get(paste("x", i, sep = ""))
+     y <- get(paste("y", i, sep = ""))
+     ans.reg[[i]] <- lm(y ~ x)
+ }
> detach(anscombe)  # Datensatz wieder aus dem Suchpfad entfernen
```

Es werden hier zunächst mit `lm()` vier einfache lineare Regressionen durchgeführt (für Details s. Abschn. 7.5) und die ausgegebenen Regressionsobjekte (Objekte der Klasse `lm`) einer Liste `ans.reg` zugewiesen. Für die Benutzung von `get()` s. Abschn. 4.3 auf S. 83. Nun sollen nachträglich für alle Regressionen die Parameterschätzungen des Modells ausgegeben werden, die man für ein einzelnes Objekt mit Hilfe der Funktion `coef()` erhält:

```
> lapply(ans.reg, coef)
 [[1]]
 (Intercept)           x
   3.0000909   0.5000909
 [[2]]
 (Intercept)           x
    3.000909    0.500000
 [[3]]
 (Intercept)           x
   3.0024545   0.4997273
 [[4]]
 (Intercept)           x
   3.0017273   0.4999091
```

```
> sapply(ans.reg, coef)
                    [,1]     [,2]      [,3]      [,4]
 (Intercept)   3.0000909 3.000909 3.0024545 3.0017273
 x             0.5000909 0.500000 0.4997273 0.4999091
```

Wie oben beschrieben, liefert `lapply()` eine Liste mit 4 Elementen (Vektoren), die die Parameterschätzungen für Achsenabschnitt (Intercept) und Steigung enthalten. Eine zu einer Matrix vereinfachte Form derselben Ergebnisse wird durch `sapply()` erzeugt. Das geschieht immer dann, wenn die Ergebnisse Vektoren gleicher Länge sind.

Die Frage, wie man immer die i-ten Spalten aller derjenigen Matrizen (analog die i-ten Elemente derjenigen Vektoren usw.) extrahiert, die Elemente einer Liste L sind, wird häufig auf den Mailinglisten gestellt. Die Lösung ist erstaunlich einfach, denn die Indizierungsklammern können sowohl als Funktionen benutzt werden als auch in anonyme Funktionen eingebaut werden. Man kann dann die `i`-ten Spalten aller Matrizen einer Liste `L` wie folgt extrahieren:

```
lapply(L, "[", , i)               # "[" als Funktion aufgefasst
sapply(L, function(x) x[ , i])    # Anonyme Funktion benutzt
```

Es gilt zu bedenken, dass alles in R mit Funktionen geschieht, auch die Indizierung. Mit `lapply(L, "[", , i)` wird jedes Element von `L` an die Funktion `"["()` weitergegeben. Das zweite Argument zu `"["()` ist leer (bedeutet: alle Zeilen) und das dritte ist `i` und wählt die entsprechenden Spalten aus. Man möge zur Verdeutlichung das folgende Beispiel betrachten:

```
> (X <- matrix(1:4, 2))
      [,1] [,2]
 [1,]    1    3
 [2,]    2    4
> "["(X, , 2)           # a) - gleich b)
 [1] 3 4
> X[ , 2]               # b) - gleich a)
 [1] 3 4
```

Wiederholungen mit `replicate()`

Die Funktion `replicate()` arbeitet analog zu `sapply()` mit der Ausnahme, dass als erstes Argument eine positive ganze Zahl n erwartet wird. Der Ausdruck im zweiten Argument wird dann n-Mal ausgewertet. Die Benutzung von `replicate()` bietet sich daher insbesondere für Simulationen an, bei denen Zufallszahlen generiert werden müssen. Beispielsweise könnte man sich wie folgt die Verteilung von 1000 Mittelwerten von jeweils 100 Würfelwürfen mit Hilfe eines Histogramms anschauen:

```
hist(replicate(1000, mean(sample(1:6, 100, replace = TRUE))))
```

Multivariates `mapply()`

Bei der Funktion `mapply()` handelt es sich um eine Art multivariate Version von `sapply()`, wobei die Reihenfolge der Argumente nicht übereinstimmt. Das erste Argument ist nämlich der Funktionsname derjenigen Funktion, die auf die im Folgenden als Argumente spezifizierten Objekte elementweise angewendet werden soll. Und zwar zunächst auf die ersten Elemente aller angegebenen Objekte, dann auf die zweiten Elemente aller angegebenen Objekte, usw. Das folgende einfache Beispiel soll die Funktionsweise verdeutlichen:

```
> mapply(sum, 1:10, 10:1, 5)        # 1+10+5, 2+9+5, 3+8+5, ...
 [1] 16 16 16 16 16 16 16 16 16 16
```

Wie üblich werden Objekte auch hier automatisch verlängert, wenn sie nicht passen. Das Argument `5` wird also implizit wie mit `rep(5, 10)` auf die passende Länge 10 gebracht.

Tabellen mit `tapply()`

Die Verwendung der praktischen Funktion `tapply()`, mit der sehr gut kleine Statistiken von Daten tabellarisch zusammengefasst werden können, wird am Beispiel der bereits in Abschn. 2.5 verwendeten *iris* Daten gezeigt. Als Argumente werden ein Vektor, ein Faktor, nach dem gruppiert werden soll und der damit die gleiche Länge wie der erste Vektor haben muss, und die anzuwendende Funktion erwartet.

```
> attach(iris)                          # Datensatz anhängen
> tapply(Sepal.Length, Species, mean)
 setosa versicolor  virginica
  5.006      5.936      6.588
> tapply(Sepal.Width, Species, range)
 $setosa
 [1] 2.3 4.4

 $versicolor
 [1] 2.0 3.4

 $virginica
 [1] 2.2 3.8
> detach(iris)
```

Hier wurde zunächst für jede Art (`Species`) der Mittelwert der jeweiligen Kelchblattlänge (`Sepal.Length`) ermittelt. Anschließend, in der Ausgabe nicht mehr so schön vereinfacht, werden durch die Funktion `range()` pro Art das Minimum und Maximum der Kelchblattbreiten ausgegeben.

Schnelle Summen und Mittelwerte

Sehr häufig muss bei Matrizen oder Arrays eine Zeilen- bzw. Spaltensumme oder ein Zeilen- bzw. Spaltenmittel berechnet werden. Dafür sind die speziellen Funktionen `rowSums()`, `colSums()`, `colMeans()` und `rowMeans()` gedacht, die auf sehr hohe Geschwindigkeit für diese Operationen optimiert sind.

Optimierte Bibliotheken und Pakete

Für Operationen (z.B. Multiplikation, Invertierung) auf großen Matrizen bieten optimierte Bibliotheken, z.B. ATLAS[3], eine besonders hohe Geschwindigkeit (Details s. Anhang A.1). Optimierte spezielle Matrixoperationen werden auch durch das Paket **Matrix**[4] bereitgestellt.

5.3 Hilfsmittel zur Effizienzanalyse

Wer seine Funktionen optimieren möchte, muss die Leistungsfähigkeit messen können, da sonst verschiedene Varianten einer Funktion nicht vergleichbar sind. Zunächst werden Möglichkeiten zur Zeitmessung vorgestellt, mit denen einige in vorherigen Abschnitten angesprochene Punkte für effizientes Programmieren verglichen werden. Möglichkeiten, den Speicherverbrauch zu messen, gehören ebenfalls zu den notwendigen Hilfsmitteln. In Abschn. 5.3.1 wird dann professionelles Werkzeug für das s.g. *Profiling* vorgestellt.

Zeitmessung

Die Funktion `proc.time()` gibt an, wie viel Zeit der aktuell laufende R Prozess bisher benötigt hat. Bei einer vor kurzer Zeit gestarteten R Sitzung erhält man z.B.:

```
> (pt1 <- proc.time())  # Sitzung läuft seit ca. 5 Min. (300/6)
 [1]   1.91   1.44 307.19     NA     NA
> plot(1:10)            # R etwas zu tun geben ...
> (pt2 <- proc.time())
 [1]   1.92   1.53 307.31     NA     NA
> pt2 - pt1             # wenige hundertstel Sek. wurden benötigt
 [1] 0.01 0.09 0.12   NA   NA
```

[3] Automatically Tuned Linear Algebra Software
[4] Das Paket **Matrix** von Douglas Bates ist auf CRAN erhältlich.

Die angegebenen Werte sind die benötigten Zeiten in Sekunden für Benutzeranteil des Prozesses (*user*), Systemanteil des Prozesses (*system*), Gesamtzeit des Prozesses (*total*, Sekunden seit dem Start von R – keine Rechenzeit) sowie zwei Werte für die summierte Zeit aller abhängig von R gestarteten Prozesse (*user*, *system*). Die Genauigkeit der Zeitmessung ist vom Betriebssystem abhängig. Unter Windows sind die beiden letztgenannten Werte nicht erhältlich und daher `NA`.

Eine komfortablere Funktion zur Zeitmessung von Ausdrücken wird mit `system.time()` bereitgestellt. Sie misst die von dem im Argument angegebenen Ausdruck benötigten Zeiten. Als Beispiel werden hier Geschwindigkeitsunterschiede zwischen den auf S. 103 angegebenen Schleifen zur (in)effizienten Initialisierung von Objekten analysiert. In den Schleifen sollten Objekte elementweise gefüllt werden. Konkret wird hier für $i = 1, \ldots, n$ der Wert i^2 in einen Vektor a an Stelle i eingefügt, und zwar

1. zunächst ohne a auf die korrekte Länge zu initialisieren (nur `a <- NULL`),
2. mit einem auf Länge n initialisierten a (`a <- numeric(n)`) und
3. vektorwertig, ganz ohne Schleife.

```
> Zeit1 <- function(n){         # Beispiel 1: ganz schlecht
+     a <- NULL
+     for(i in 1:n) a <- c(a, i^2)
+ }
> Zeit2 <- function(n){         # Beispiel 2: etwas besser
+     a <- numeric(n)
+     for(i in 1:n) a[i] <- i^2
+ }
> Zeit3 <- function(n){         # Beispiel 3: viel besser
+     a <- (1:n)^2
+ }

> system.time(Zeit1(30000))
 [1] 8.87 0.07 8.96   NA   NA
> system.time(Zeit2(30000))
 [1] 0.19 0.00 0.19   NA   NA
> system.time(Zeit3(30000))
 [1]  0  0  0 NA NA
```

Einige Punkte werden hier ganz deutlich. Die Effizienzanalysen in Form von Zeitmessungen lohnen sich, da nahezu gleich lange Funktionen deutlich unterschiedliche Rechenzeiten benötigen.

Die Initialisierung von Objekten auf die endgültige Länge ist sehr wichtig. In diesem Beispiel konnte eine etwa 45-fach schnellere Bearbeitung nur durch

die geeignete Initialisierung und dadurch unnötige vielfache Allozierung von Speicher erreicht werden.

Der zuvor oft wiederholte Satz „*Vektorwertiges Programmieren möge der Leser als Gesetz ansehen!*“ hat seine Berechtigung, denn die benötigte Rechenzeit der vektorwertigen Version (3) ist nicht messbar ($< 1/100$ Sek.) und der Geschwindigkeitsvorteil ist damit enorm. Hier muss auf das Objekt `a` nämlich nur einmal zugegriffen werden, anstelle von n Zugriffen bei den Schleifenversionen. Man bedenke aber, dass nicht alle Funktionen vektorisierbar sind, diese Möglichkeit also nicht immer besteht.

Auch der Sinn des auf S. 103 angegebene Punktes, dass konstante Berechnungen nicht innerhalb sondern vor der Schleife ausgeführt werden sollen, kann mit `system.time()` verdeutlicht werden.

```
> Zeit4 <- function(n){         # Beispiel 4: ganz schlecht
+     a <- numeric(n)
+     for(i in 1:n) a[i] <- sin(2 * pi * i)
+ }
> Zeit5 <- function(n){         # Beispiel 5: etwas besser
+     a <- numeric(n)
+     pi2 <- 2 * pi
+     for(i in 1:n) a[i] <- sin(pi2 * i)
+ }
> Zeit6 <- function(n){         # Beispiel 6: viel besser
+     a <- sin(2 * pi * 1:n)
+ }

> system.time(Zeit4(30000))
 [1] 0.34 0.00 0.35   NA   NA
> system.time(Zeit5(30000))
 [1] 0.3 0.0 0.3   NA   NA
> system.time(Zeit6(30000))
 [1]  0  0  0 NA NA
```

Auch hier ist eine deutliche Verringerung der Rechenzeit festzustellen, wenn die Berechnung `2 * pi` nur einmal ausgeführt wird, und anstatt der weiteren 29999 neuen Berechnungen desselben Wertes jeweils auf das Objekt `pi2` zugegriffen wird. Natürlich ist die vektorwertige Version wieder am schnellsten.

Speichernutzung

Die Speichernutzung von R kann schon beim Start durch Angabe von Minima und Maxima eingegrenzt werden (s. `?Memory`, auch für das Verständnis der Speicherallozierung) oder zur Laufzeit mit Hilfe von `mem.limits()` nach oben korrigiert werden. In der Regel ist das jedoch nicht nötig.

Die aktuell verwendete Menge an Speicher kann mit `gc()` angezeigt werden. Durch den Funktionsaufruf wird gleichzeitig auch explizit der *garbage collector* („Müllsammler“) gestartet, der Speicherplatz für nicht mehr verwendete und gelöschte Objekte wieder freigibt. Implizit wird die Freigabe innerhalb von R automatisch und dynamisch je nach Notwendigkeit und je nach Alter von Objekten durchgeführt. Man beachte, dass das Löschen eines Objektes, etwa mit `rm()` nicht direkt wieder Speicherplatz freigibt.

Ein großer „Speicherfresser“ kann das gerne vergessene `.Last.value` Objekt sein, das das Ergebnis des letzten Ausdrucks für weitere Verwendung speichert (s. auch Abschn. 2.3). Bei einem sehr großen Ergebnis (z.B. einer großen Matrix) lohnt es sich manchmal, dieses Objekt zu löschen. Das geschieht z.B. auch bei Aufruf von `gc()`, denn dessen Rückgabe ist ein sehr kleines Objekt, wovon `.Last.value` überschrieben wird. Bei zwei aufeinander folgenden Aufrufen von `gc()` wird der Speicherverbrauch deswegen manchmal beim zweiten Aufruf nochmals drastisch reduziert, weil dann der Speicher des alten (bereits überschriebenen) `.Last.value` Objekts freigegeben wird.

Unter Windows gibt es ein voreingestelltes Maximum des von R verwendeten Hauptspeichers, und zwar das Minimum von vorhandenem Hauptspeicher des Rechners und 1024 MB. Diese Einstellung kann mit Hilfe der Startoption `--max-mem-size` (s. Anhang A.2) heraufgesetzt werden, so dass sie in einer Verknüpfung angegeben werden kann und nicht bei jedem Start neu gesetzt werden muss. Auch zur Laufzeit kann die maximal zulässige Hauptspeichermenge erhöht werden, und zwar mit Hilfe der Funktion `memory.limit()`, die auch das aktuelle Limit anzeigt:

```
> memory.limit()         # Limit der Speichernutzung in Bytes
 [1] 1073074176
> memory.limit(1536)     # Setzt neues Limit (1.5 GB) in Megabytes
 NULL
> (ml <- memory.limit()) # Überprüfen ergibt richtiges Limit
 [1] 1610612736
> ml / (1024^2)
 [1] 1536
```

Man beachte, dass das aktuelle Limit in Bytes angezeigt wird, das Setzen eines neuen Maximums aber in Megabytes erfolgt. Die Funktion `memory.size()` gibt die gerade verwendete Menge an Speicher unter Windows an.

5.3.1 Laufzeitanalyse – *Profiling*

Eine professionelle Laufzeitanalyse ist durch das s.g. *Profiling* möglich (für weitere Details s. auch Venables, 2001). Damit wird zu einer Funktion ein Profil erstellt, das aussagt, wie viel Zeit in welchem darin enthaltenen Funk-

tionsaufruf verwendet wird, und zwar rekursiv bis zur letzten Stufe der verschachtelt enthaltenen Funktionsaufrufe hinab.

Sollte eine der enthaltenen Funktionen einen Großteil der Zeit verbrauchen, also u.U. einen „Flaschenhals“ darstellen, kann dort optimiert werden. Optimierungen von Funktionen, die fast keine Rechenzeit verbrauchen, lohnen sich entsprechend weniger. Man kann seine Bemühungen auf die relevanten Optimierungen konzentrieren.

Mit der Funktion `Rprof()` wird das Aufzeichnen von Profiling-Informationen über ausgewertete Ausdrücke gestartet und beendet (`Rprof(NULL)`). Die zugehörigen aufgezeichneten Informationen werden gemäß Voreinstellung in der Datei `Rprof.out` abgelegt. Diese Datei kann dann auf zwei Wegen analysiert werden:

- Die Funktion `summaryRprof()` kann die aufgezeichneten Informationen innerhalb von R auswerten und zeigt sie an.
- Schneller, aber etwas weniger komfortabel, kann man an die Auswertung kommen, wenn man R aus der *shell* des Betriebssystems mit

  ```
  R CMD Rprof Rprof.out
  ```

  aufruft. Insbesondere unter Windows ist zu beachten, dass Perl installiert sein muss und der Pfad zum R Unterverzeichnis `\bin` gesetzt sein sollte.

Ausgegeben wird die Liste der im Ausdruck aufgerufenen Funktionen in zwei Sortierungen. Die eine Sortierung erfolgt nach der Gesamtrechenzeit (*total*), d.h. inkl. der Rechenzeit der in einer Funktion aufgerufenen anderen Funktionen, die andere Sortierung nach der ausschließlich von einer Funktion verbrauchten Zeit (*self*).

Der Leser möge das Beispiel aus der Hilfe (`?Rprof`) studieren. Darin ist insbesondere die Ausgabe von `summaryRprof()` interessant, die wegen ihrer Länge im hier angegebenen und leicht abgewandelten Beispiel nicht vollständig gedruckt wird:

```
>     # Aufzeichnung für Profiling starten:
> Rprof(tmp <- tempfile(), interval = 0.01)
> example(glm)      # Auszuwertender Code
 . . . . .          # Lange Ausgabe ...
> Rprof(NULL)       # Aufzeichnung für Profiling beenden
> summaryRprof(tmp) # Aufzeichnung auswerten
 $by.self
                      self.time self.pct total.time total.pct
 index.search              0.03     13.6       0.03      13.6
 !                         0.01      4.5       0.01       4.5
 [.factor                  0.01      4.5       0.01       4.5
 . . . . .
```

```
$by.total
                      total.time total.pct self.time self.pct
example                     0.22     100.0      0.00      0.0
source                      0.16      72.7      0.00      0.0
eval.with.vis               0.10      45.5      0.00      0.0
glm                         0.06      27.3      0.00      0.0
. . . . .

$sampling.time
[1] 0.23
```

Die hier eingesetzte Funktion `tempfile()` generiert einen Dateinamen für eine temporäre Datei, die beim Beenden der R Sitzung gelöscht wird.

In der Sortierung „`by.self`“ ist keine Funktion mit auffällig großer Rechenzeit vorhanden. Wird `example(glm)` nochmals gestartet und erneut analysiert, kann es bei so kleinen Unterschieden schon zu einer völlig anderen Sortierung kommen, denn es wurden für das Profiling nur alle 0.01 Sekunden Informationen aufgezeichnet. Daran sieht man, dass die Funktion `glm()` zur Anpassung generalisierter linearer Modelle bereits sehr effizient in R implementiert ist und hier keine Optimierung nötig ist.

In der Sortierung „`by.total`“ vereint `example()` 100% der Laufzeit für sich, weil alle anderen Funktionen schließlich in Abhängigkeit von `example()` gestartet wurden, obwohl diese Funktion selbst durch eigene Laufzeit fast keinen Anteil zur Gesamtlaufzeit beiträgt. Die Gesamtlaufzeit beträgt laut Auswertung 0.23 Sekunden.

Luke Tierney arbeitet an dem Paket **proftools**[5], das noch weiter ausgefeilte Werkzeuge für die Laufzeitanalyse bietet.

[5] Das Paket **proftools** ist z.Zt. erhältlich unter der URL `http://www.stat.uiowa.edu/~luke/R/codetools/`.

6

Objektorientiertes Programmieren

Objektorientiertes Programmieren, kurz OOP, ist inzwischen in sehr vielen Sprachen möglich. Zwei Fakten zu R sind aus den ersten Kapiteln bekannt. Zum einen ist *alles* ein Objekt (jegliche Art von Daten und Funktionen, s. auch Abschn. 2.3), zum anderen handelt es sich um eine *objektorientierte* Sprache.

Objektorientiert Programmieren bedeutet, dass eine *generische* Funktion für verschiedene *Klassen* von Objekten jeweils angepasste *Methoden* mitbringen kann. Der Vorteil ist, dass der Benutzer nicht viele Funktionen für verschiedene Objektklassen kennen muss, sondern einfach die generische Funktion benutzen kann, ohne Details zur Objektklasse zu kennen.

Ein einfaches Beispiel ist die Funktion `print()`, die beliebige Klassen von Objekten darstellen muss, denn jedes Objekt soll auf der Konsole ausgegeben werden können. Obwohl Datensätze und Listen sehr ähnliche Datenstrukturen sind, werden sie doch recht unterschiedlich dargestellt, nämlich Datensätze in Tabellenform und Listen als Auflistung. Das Verhalten von `print()` hängt also von der Klasse des Objekts ab. Und anstatt alle möglichen Arten von Objekten innerhalb von `print()` per Fallunterscheidung zu behandeln, schreibt man für jede Klasse von Objekten eine Methode zu der generischen Funktion `print()`, die entsprechend der Klasse eines Objekts ihre zugehörige Methode aufruft.

R bietet zwei Ansätze zum objektorientierten Programmieren. In Abschn. 6.1 wird der als S3 bekannte Ansatz beschrieben, während der als S4 (Chambers, 1998) bekannte Ansatz, der von S3 verschieden und wesentlich formaler ist, in Abschn. 6.2 besprochen wird.

Beide Ansätze werden heute parallel benutzt, man sollte also die jeweiligen Grundlagen grob kennen. Vieles in R basiert auf den „alten“ S3-Methoden, während einige neu implementierte Pakete mit S4-Methoden arbeiten. Während man mit S3 sehr schnell durch implizite Angaben von Klassen zum Ziel kommt, braucht man mit S4 deutlich mehr Code, hat dafür dann aber auch einwandfreie Definitionen. S3 eignet sich zum Ergänzen vorhandener

Funktionen und Klassen und zum schnellen Ausprobieren. S4 sei demjenigen ans Herz gelegt, der strukturiert programmieren möchte – insbesondere bei der Entwicklung eines neuen Pakets.

6.1 OOP mit S3-Methoden und -Klassen

Aus Abschn. 2.3 ist bekannt, dass Objekte Attribute besitzen können. Das Attribut „class“ gibt die *Klasse* des Objekts an, wobei es auch eine Liste mehrerer Klassennamen enthalten kann, wenn Klassen eine hierarchische Struktur besitzen. Das Klassenattribut wird mit Hilfe von `class()` abgefragt und auch gesetzt (der exakte Funktionsname ist dann „`class<-`“). Eine Zusammenstellung weiterer nützlicher Funktionen für das Programmieren mit S3-Klassen gibt es in Tabelle 6.1.

Tabelle 6.1. Auswahl von Funktionen für objektorientiertes Programmieren mit S3-Methoden und -Klassen

Funktion	Beschreibung
`attributes()`	Erfragen und Setzen aller Attribute eines Objekts
`attr()`	Erfragen und Setzen konkreter Attribute
`class()`	Erfragen und Setzen des Klassen–Attributs
`getS3method()`	nicht aus einem Namespace exportierte Methode abfragen
`inherits()`	von einer anderen Klasse *erben*
`methods()`	alle zu einer generischen Funktion gehörenden Methoden
`NextMethod()`	Verweis auf die nächste in Frage kommende Methode
`UseMethod()`	Verweis von der generischen Funktion an die Methode

Als Beispiel wird hier, wie bereits in Abschn. 5.2.2, der Datensatz von Anscombe (1973) zur linearen Regression verwendet (s. auch Abschn. 7.5):

```
> class(anscombe$x1)
 [1] "numeric"
> print(anscombe$x1)  # analog zur Eingabe von 'anscombe$x1'
 [1] 10  8 13  9 11 14  6  4 12  7  5
> ansreg1 <- lm(y1 ~ x1, data = anscombe)
> class(ansreg1)
 [1] "lm"
> print(ansreg1)      # analog zur Eingabe von 'ansreg1'

 Call:
 lm(formula = y1 ~ x1, data = anscombe)
```

```
Coefficients:
(Intercept)           x1
     3.0001       0.5001
```

Während es sich beim Objekt `x1` um einen Vektor reeller Zahlen (Klasse `numeric`) handelt, ist das Objekt `ansreg1` (Klasse `lm`) als Ergebnis einer linearen Regression offensichtlich komplexerer Natur. Die generische Funktion `print()` leitet ein Objekt entsprechend seiner Klasse an eine passende Methode mit Hilfe der Funktion `UseMethod()` weiter:

```
> print
function (x, ...)
UseMethod("print")
<environment: namespace:base>
```

Als Argumente werden hier `x`, also das Objekt einer bestimmten Klasse, sowie das Dreipunkte-Argument verwendet, das gerade hier von Vorteil ist, da so auch unterschiedliche weitere Argumente an die jeweiligen Methoden weitergegeben werden können.

Um zu erfahren, welche *Methoden* es zu einer generischen Funktion gibt, kann die Funktion `methods()` benutzt werden, deren Ausgabe für die generische Funktion `print()` hier wegen der großen Menge an Methoden gekürzt ist:

```
> methods(print)
   [1] print.acf*      print.anova        print.aov*
   [4] print.aovlist*  print.ar*          print.Arima*
 . . . . .
  [37] print.default   print.dendrogram*  print.density
 . . . . .
  [67] print.listof    print.lm           print.loadings*
 . . . . .
 [127] print.xgettext* print.xngettext*   print.xtabs*

    Non-visible functions are asterisked
```

Hier wird deutlich, wie man eine S3-Methode zu einer bestimmten Klasse definiert, nämlich einfach und ausschließlich durch den Namen der Funktion, der die Form

```
NameDerGenerischenFunktion.NameDerKlasse
```

haben muss, also die Namen der generischen Funktion und der Klasse durch einen Punkt getrennt. Die Methode von `print()` für die Klasse `lm` ist damit in `print.lm()` definiert (man möge sich die Definition durch Eingabe von `print.lm` anschauen). Sollte keine passende Methode gefunden werden, so wird von `UseMethod()` immer die *default* Methode gewählt. Im Beispiel von oben wird `print.default()` für das Objekt `x1` der Klasse `numeric` verwendet.

Analog mögen die Leser die Ergebnisse der Aufrufe

```
> plot(anscombe$x1)
> plot(ansreg1)
> summary(anscombe$x1)
> summary(ansreg1)
```

ansehen und sich die Methoden der generischen Funktionen `plot()` und `summary()` anschauen, die neben `print()` die meisten Methoden mitbringen.

Alle in der Ausgabe von `methods()` mit einem Stern versehene Methoden sind in den jeweiligen Namespaces versteckt. Zum Anschauen von versteckten Methoden bietet sich die Funktion `getS3method()` an (s. auch Abschn. 4.3, S. 81). Beispielsweise erhält man mit `getS3method("print", "acf")` den Code für die Methode `print.acf()`.

Die Klassenabhängigkeit einer generischen Funktion muss nicht unbedingt durch ihr erstes Argument ausgedrückt werden, sondern kann auch durch Angabe des Arguments `object` in der Funktion `UseMethod()` für beliebige Argumente einer generischen Funktion erfolgen.

Vererbung

Vererbung bedeutet, dass eine Klasse als Spezialisierung einer anderen Klasse aufgefasst werden kann und die spezialisierte Klasse dann von der anderen „erbt“. Z.B. ein durch Anpassung eines generalisierten Linearen Modells entstandenes `glm`-Objekt kann als Spezialisierung eines `lm`-Objekts aufgefasst werden (im mathematischen Sinne ist es natürlich eine Verallgemeinerung), denn ein `glm`-Objekt enthält alle Elemente, die auch ein `lm`-Objekt enthält. In manchen Fällen könnte eine für ein Objekt der Klasse `lm` geschriebene Methode auch für die Verarbeitung eines Objekts der Klasse `glm` verwendet werden. Daher erbt ein `glm`-Objekt auch von der Klasse `lm`, und man erhält:

```
> class(glmObjekt)
 [1] "glm" "lm"
```

Mit Hilfe von `inherits()` kann man fragen, ob ein Objekt von einer bestimmten Klasse erbt:

```
> inherits(glmObjekt, "lm")
 [1] TRUE
> inherits(glmObjekt, "glm")
 [1] TRUE
> inherits(glmObjekt, "numeric")
 [1] FALSE
```

Häufig wird bei Spezialisierungen von einer Methode auf die Methode für die übergeordnete Klasse verwiesen (z.B. von einer Methode für `glm`-Objekte an eine für `lm`-Objekte). Dazu wird die Funktion `NextMethod()` verwendet,

die die Methode für den nächsten Eintrag im Attribut `class` des Objekts auswählt. Wenn keine weitere Klasse in der Liste vorhanden ist, wird die `default` Methode verwendet.

So könnte man ein `glm`-Objekt z.B. durch die Definition

```
> print.glm <- function(x)
+     NextMethod()
```

genauso wie ein `lm`-Objekt ausgeben lassen, was aber wegen der besseren vorhandenen Funktion hier wenig Sinn macht. Details zu `lm` und `glm` Objekten sind in Kap. 7 zu finden.

Bei-Spiel – Aufgabenstellung

Es folgt ein kleines Beispiel, dass keine nützliche Anwendung beinhaltet, aber das Verständnis für objektorientiertes Programmieren in R fördert. Die Leser mögen sich zur Übung erst eine Lösung überlegen, bevor sie in die Auflösung am Ende dieses Abschnitts schauen.

Die Aufgabe besteht nun darin, durch objektorientierte Programmierung möglichst kurzen Code zu schreiben, der Folgendes ermöglicht: Durch Eingabe des Buchstabens `Q` (ohne Klammern, aber mit Abschluss der Eingabe durch `Enter`) in der Konsole soll R direkt geschlossen werden, also ohne Nachfrage, ob der Workspace gespeichert werden soll.

Bei-Spiel – Lösung

Die Idee ist, dass bei Eingabe des Buchstabens `Q` ein Objekt gleichen Namens mit `print()` bearbeitet wird (i.d.R. Ausgabe von Text mit relevanten Informationen des Objekts auf der Konsole). Wenn ein solches Objekt einer Klasse angehört, so kann man eine Methode zur generischen Funktion `print()` für diese Klasse schreiben, die R schließt:

```
> Q <- 1                      # beliebiges Objekt Q
> class(Q) <- "quit"          # Q hat jetzt die Klasse "quit"
> print.quit <- function(x) # Eine Methode für print() zur Klasse
+     q("no")                 #   "quit"
> Q                           # R wird geschlossen
```

Durch Eingabe von `Q` wird zunächst die Funktion `print()` aufgerufen, die als generische Funktion über `UseMethod("print")` die Klasse von `Q` bestimmt. Dann wird gemäß der Namenskonventionen die passende Methode `print.quit()` zur Klasse `quit` gefunden und ausgeführt. Anstatt etwas auszugeben, ruft diese Funktion aber einfach `q("no")` auf. R wird also beendet.

Es sind auch noch kürzere Lösungen[1] möglich. Die hier gezeigte soll dem Verständnis des objektorientierten Programmierens dienen.

6.2 OOP mit S4-Methoden und -Klassen

S4-Methoden und -Klassen werden im grünen Buch (Chambers, 1998) definiert. Durch das von Chambers selbst entwickelte Paket **methods**, das als Voreinstellung beim Start von R mitgeladen wird, ist S4 implementiert. Das in R verwendete Modell für S4 weicht in wenigen Punkten von der ursprünglichen Definition ab.

S4-Methoden und -Klassen werden von Venables und Ripley (2000), auch anhand von Beispielen, ausführlich beschrieben. Darüber hinaus gibt die Hilfeseite `?Methods` detaillierte Auskünfte und Verweise auf weitere Hilfeseiten. Bates (2003) und Bates und DebRoy (2003) erläutern, u.a. am Beispiel ihres Pakets **nlme**, wie man Pakete von S3- auf S4-Methoden und -Klassen umstellt. Sie erläutern neben den Vor- und Nachteilen für das Paket auch die bei der Umstellung auftretenden Probleme.

Ein ausführliches Beispiel zur Benutzung der im Folgenden erwähnten Funktionen und Definitionen von Klassen wird in Abschn. 6.2.1 gegeben. Durch Nachvollziehen des Beispiels wird hier Beschriebenes viel schneller ersichtlich als durch lange Erklärungen, auf die verzichtet wird. Weitere Beispiele findet man auch in der o.g. Literatur.

Vor- und Nachteile

Der Vorteil der neuen Methoden ist, dass ganz formal Klassen, Methoden und generische Funktionen definiert werden können. Dabei wird u.a. genau spezifiziert, welche Struktur ein Objekt einer bestimmten Klasse haben muss, während man bei einer S3-Klasse die Struktur eines Objekts nur implizit festlegt. Dazu gibt es die passenden Werkzeuge, z.B. zur Validierung eines Objekts und dessen Klassenzugehörigkeit. Eine (unvollständige) Aufstellung dieser Werkzeuge gibt es in Tabelle 6.2.

Die formalen Definitionen helfen bei der Entwicklung komplexer Pakete und Funktionen, weil man sich schon zu Beginn der Entwicklung Gedanken zu der Struktur des Problems und der Daten machen muss. Es werden sowohl spätere Fehler als auch die Notwendigkeit der Umstrukturierung der Klassen vermieden. Ebenso treten keine Probleme wegen versehentlicher, ungünstiger Namensgebungen von Funktionen auf, die evtl. fälschlicherweise als S3-Methode erkannt werden, obwohl sie eigenständig sein sollen.

[1] Eine sehr schöne, allerdings nicht leicht zu verstehende Lösung wurde auf der Mailingliste *R-help* von Henrik Bengtsson am 17. September 2003 vorgeschlagen.

Tabelle 6.2. Auswahl von Funktionen für objektorientiertes Programmieren mit S4-Methoden und -Klassen

Funktion	Beschreibung
`setClass()`, `getClass()`	Definition/Abfrage einer Klasse
`representation()`	Representation für eine Klassendefinition
`prototype()`	Prototyp des Objekts einer Klasse
`setValidity()`, `getValidity()`	Definition/Abfrage einer Validitäts-Prüfung
`new()`	Erzeugen eines neuen Objekts einer Klasse
`slot()`, `@`	Zugriff auf Slots
`slotNames()`	Name aller Slots eines Objekts
`setGeneric()`	Definition einer generischen Funktion
`setMethod()`	Definition einer Methode
`signature()`	Zuordnung von Methoden zu Klassen
`selectMethod()`	explizite Methodenwahl

Die Nachteile von S4 liegen zum Teil in der Formalität, die zuvor auch als Vorteil beschrieben wurde. Es ist nämlich auch ein viel größerer Aufwand für die Implementation notwendig als bei S3-Methoden und -Klassen.

Definition von Klassen und Objekten

Alle Objekte einer bestimmten Klasse haben dieselben *Slots* (s. Abschn. 2.9.6). Es kann aber auch Klassen ohne Slots geben. Sowohl die Namen, als auch die Klassen der Objekte innerhalb der Slots sind Bestandteil der Definition einer Klasse, wobei die Slots meist eine einfachere Klasse, etwa Vektoren der Klasse `numeric` enthalten. Zur Erinnerung sei bemerkt, dass man mit Hilfe des `@`-Operators oder der Funktion `slot()` auf Slots zugreift.

Die Definition einer neuen Klasse geschieht mit Hilfe von `setClass()` (für Details bitte unbedingt die zugehörige Hilfe lesen!), wobei die Definition der zugehörigen Slots mit `representation()` geschieht. Weiter können mit `prototype()` Voreinstellungen für ein neues Objekt dieser Klasse definiert werden.

Ein Objekt, das einer bestimmten Klasse angehört, kann mit `new()` erzeugt werden.

In S4 hat jedes Objekt genau eine Klasse, während es in reinem S3 beliebig viele (bzw. gar keine) haben konnte. Vererbung wird stattdessen innerhalb der Definition der Klasse selbst ausgedrückt. Im Argument `contains` der Funktion `setClass()` kann man angeben, von welcher Klasse die neu definierte Klasse erbt. Slots der übergeordneten Klasse werden automatisch übernommen.

Es soll nicht vorkommen, dass ein Objekt einer bestimmten Klasse erzeugt wird, das nicht die erforderliche Struktur hat. Dazu kann man eine

Funktion schreiben, die die Gültigkeit (Validität) eines Objekts bzgl. der Klassendefinition überprüft und entsprechend `TRUE` oder `FALSE` zurückgibt. Eine solche Funktion wird mit `setValidity()` als vorgegebene Methode zur Validitätsprüfung definiert.

Generische Funktionen und Methoden

Beliebige Funktionen werden zu generischen Funktionen nach S4 durch entsprechendes Setzen mit Hilfe von `setGeneric()`. Zugehörige Funktionen für Methoden können dann durch `setMethod()` spezifiziert werden, wobei mindestens die ersten drei Argumente angegeben werden müssen:

- `f`, der Name der entsprechenden generischen Funktion,
- `signature`, der Name bzw. die Namen der zu bearbeitenden Klasse(n) als Zeichenkette,
- `definition`, diejenige Funktion, die als Methode deklariert wird.

Wegen dieser formalen Definitionen können die generische Funktion und ihre Methoden in S4 völlig unabhängig voneinander benannt sein (identische Benennung ist natürlich nicht möglich). Trotzdem sei zu geeigneter Namensgebung ähnlich der S3 Konvention geraten, da das zur Übersichtlichkeit beiträgt. Wenn man S3 Konventionen komplett beachtet, kann man in vielen Fällen parallel S3 und S4 durch dieselben Funktionen abdecken, auch wenn das meist nicht erforderlich ist.

Es sei abschließend erwähnt, dass zum Anzeigen von Objekten auf der Konsole in S4 laut Chambers (1998) anstelle von `print()` die generische Funktion `show()` verwendet werden soll.

6.2.1 Beispiel: Eine Klasse Wave und Methoden

Wave Dateien, die digitalisierte Daten der Schwingungen eines Klangs – sei es Musik oder Geräusch – enthalten, stellen kein Format dar, dass man häufig in Statistik-Sprachen findet.

Dieses Beispiel zeigt anhand von Wave Dateien, wie eine (S4) Klasse als R Repräsentation einer Wave Datei definiert werden kann. Zusätzlich werden generische Funktionen und Methoden im Beispiel definiert, mit denen dann entsprechende Wave-Objekte behandelt werden können. Der Code stammt aus dem Paket **tuneR** (erhältlich auf CRAN), das sich u.a. auch für den Import und Export von Wave Dateien eignet.

Definition von Klassen und Objekten

Zunächst soll mit der Definition der Klasse begonnen werden, die möglichst natürlich an die in einer Wave Datei enthaltenen Daten angepasst werden soll.

Dazu gehören als Slots zunächst Daten der Kanäle (jeweils Zahlenwerte, also von Klasse `numeric`), wobei für Mono-Aufnahmen nur der linke Kanal, sonst linker und rechter Kanal verwendet werden. Die Anzahl der Kanäle kann durch einen logischen Wert kodiert werden (Stereo bei `TRUE`, sonst Mono). Wichtige weitere Größen sind die Samplingrate der Aufnahme und die Auflösung.

```
> setClass("Wave",
+     representation = representation(left = "numeric",
+         right = "numeric", stereo = "logical",
+         samp.rate = "numeric", bit = "numeric"),
+     prototype = prototype(stereo = TRUE, samp.rate = 44100,
+         bit = 16))
 [1] "Wave"
```

In der Definition (mit `setClass()`) wurde hier mit `representation()` festgelegt, welchen Typ die jeweiligen Slots besitzen. Außerdem wurde mit `prototype()` festgelegt, dass bei Erzeugung eines neuen Wave-Objekts dieses die typischen Einstellungen für CD-Qualität haben soll (Stereo, 44100 Hertz Samplingrate, 16-Bit).

Wie erwartet sieht ein neues Objekt, `waveobj`, dann wie folgt aus:

```
> (waveobj <- new("Wave"))  # Erzeugen eines Wave Objekts
 An object of class "Wave"
 Slot "left":
 numeric(0)

 Slot "right":
 numeric(0)

 Slot "stereo":
 [1] TRUE

 Slot "samp.rate":
 [1] 44100

 Slot "bit":
 [1] 16
```

Validität

Als Nächstes soll eine Validitätsprüfung möglich gemacht werden, denn mehrere logische Werte für `Stereo` oder mehrere bzw. negative Samplingraten machen sicherlich keinen Sinn.

Die im Folgenden definierte (anonyme) Funktion gibt `TRUE` zurück, wenn das Wave-Objekt gültig ist, sonst wird sie mit der Fehlermeldung abgebrochen, die als Zeichenfolge von `return()` zurückgegeben wird:

```
> setValidity("Wave", function(object){
+     if(!is(object@left, "numeric"))
+         return("channels of Wave objects must be numeric")
+     if(!(is(object@stereo, "logical") &&
+          (length(object@stereo) < 2)))
+         return("slot 'stereo' of a Wave object must be a logical
+                 of length 1")
+     if(object@stereo){
+         if(!is(object@right, "numeric"))
+             return("channels of Wave objects must be numeric")
+         if(length(object@left) != length(object@right))
+             return("both channels of Wave objects must have
+                     the same length")
+     }
+     else if(length(object@right))
+         return("'right' channel of a wave object is not supposed
+                 to contain data if slot stereo==FALSE")
+     if(!(is(object@samp.rate, "numeric") &&
+        (length(object@samp.rate) < 2) && (object@samp.rate > 0)))
+         return("slot 'samp.rate' of a Wave object must be a
+                 positive numeric of length 1")
+     if(!(is(object@bit, "numeric") &&
+        (length(object@bit) < 2) && (object@bit %in% c(8, 16))))
+         return("slot 'bit' of a Wave object must be a
+                 positive numeric (either 8 or 16) of length 1")
+     return(TRUE)
+ })

 Slots:

 Name:       left     right    stereo samp.rate       bit
 Class:   numeric   numeric   logical   numeric   numeric
```

Die Funktion wird als Methode zur Validitätsprüfung für die Klasse `Wave` durch den Aufruf `setValidity("Wave", Funktion)` deklariert. Sicherlich kann man auch noch weitere Einschränkungen für ein gültiges Wave-Objekt finden.

Mit S4 wurde die hier benutzte Funktion `is()` eingeführt, die testet, ob ihr erstes Argument als Objekt der Klasse des zweiten Arguments behandelt werden kann. Analog gibt es auch eine Funktion `as()` zum Erzwingen einer anderen Klasse.

Die folgende ungültige Zuweisung wird schon durch Überprüfung der Deklaration der Klasse Wave abgefangen:

```
> waveobj@stereo <- "Ja"     # ungültig, leicht editierte Ausgabe
 Error in checkSlotAssignment(object, name, value) :
     Assignment of an object of class "character" is not valid for
     slot "stereo" in an object of class "Wave";
     is(value, "logical") is not TRUE
```

Bereits bei der Deklaration der Klasse wurde schließlich festgelegt, dass der `stereo`-Slot ein logischer Wert sein muss. Die folgende Zuweisung hingegen wird erst durch die Validitätsprüfung, die mit der Funktion `validObject()` erzwungen werden kann, als ungültig erkannt:

```
> waveobj@samp.rate <- -1000
> validObject(waveobj)       # ungültig, leicht editierte Ausgabe
Error in validObject(waveobj) :
    invalid class "Wave" object: slot 'samp.rate' of a Wave object
    must be a positive numeric of length 1
```

Hier ist der Wert -1000 nämlich entsprechend der Definition der Klasse als numerischer Wert gültig, und erst mit der Validitätsprüfung wird der negative Wert als ungültig erkannt. Entsprechend werden Aufrufe von `validObject()` in wichtigen Funktionen benutzt. In der Funktion `new()` erfolgt die Valididätsprüfung immer.

Vererbung

Als Beispiel zur Vererbung soll hier eine Klasse Song definiert werden, die von der Klasse Wave erbt, aber einen zusätzlichen Slot für den Text eines Liedes hat:

```
> setClass("Song",
+     representation = representation(text = "character"),
+     contains = "Wave")
 [1] "Song"
> new("Song")
 An object of class "Song"
 Slot "text":
 character(0)

 Slot "left":
 . . . . .  Der Rest ist identisch mit einem Wave Objekt
```

Methoden

Der nächste zu bearbeitende Punkt ist die geeignete Ausgabe eines `Wave`-Objekts auf die Konsole. Dabei sollte man als Voreinstellung lieber nicht die

Daten beider Kanäle ausgeben lassen, die bei einem nur einminütigen Stück in CD-Qualität zusammen 5292000 Zahlen umfassen (man multipliziere 60 Sekunden, 2 Kanäle, 44100 Hertz). Eine Ausgabe der Gesamtlänge und der sonstigen Eigenschaften des Objekts erscheinen also geeigneter.

Dazu entwickeln wir eine Methode zur generischen Funktion `show()`:

```
> setMethod("show", signature(object = "Wave"),
+     function(object){
+         l <- length(object@left)
+         cat("\nWave Object")
+         cat("\n\tNumber of Samples:     ", l)
+         cat("\n\tDuration (seconds):    ",
+             round(l / object@samp.rate, 2))
+         cat("\n\tSamplingrate (Hertz):  ", object@samp.rate)
+         cat("\n\tChannels (Mono/Stereo):",
+             if(object@stereo) "Stereo" else "Mono")
+         cat("\n\tBit (8/16):            ", object@bit, "\n\n")
+     }
+ )
 [1] "show"
```

Mit `setMethod()` wurde die im dritten Argument angegebene anonyme Funktion als Methode zu `show()` (im ersten Argument angegeben) gesetzt. Dabei gibt `signature()` im zweiten Argument von `setMethod()` an, dass diese Methode gewählt werden soll, wenn das Argument `object` die Klasse `Wave` hat.

Ausgaben auf die Konsole durch die generische Funktion `show()` sehen mit dieser Definition wie folgt aus:

```
> show(waveobj)   # analog: "print(waveobj)" oder einfach "waveobj"

Wave Object
        Number of Samples:      0
        Duration (seconds):     0
        Samplingrate (Hertz):   44100
        Channels (Mono/Stereo): Stereo
        Bit (8/16):             16
```

Als Nächstes könnte man eine geeignete Methode für die graphische Ausgabe zur generischen Funktion `plot()` schreiben, die die Daten als Grafiken der Zeitreihen wiedergibt. Dabei sollten für Mono-Daten eine und für Stereo-Daten zwei untereinander angeordnete Bilder entstehen. Einige Samples müssen dann u.U. „intelligent“ ausgelassen werden, da die grafische Darstellung sehr großer Datensätze Rechenzeit und Speicherplatz kostet.

7

Statistik mit R

Bei R handelt es sich um eine „Sprache und Umgebung für Statistisches Rechnen“ (R Development Core Team, 2006a). Ein Kapitel über die Anwendung statistischer Verfahren mit R darf und soll in diesem Buch nicht fehlen. Der Fokus des Buches ist jedoch auf die Programmierung mit R gerichtet, und es soll in erster Linie das Verständnis der Sprache vermittelt werden. Daher werden nur häufig verwendete und wichtige Verfahren kurz erklärt sowie syntaktische Feinheiten der Modellspezifikation beschrieben.

Die Theorie der meisten hier erwähnten statistischen Verfahren kann bei Hartung et al. (2005) nachgeschlagen werden. Einen genaueren und umfassenderen Überblick über die Umsetzung statistischer Verfahren in R gibt es in dem darauf spezialisierten Buch „Modern Applied Statistics with S“ (Venables und Ripley, 2002) sowie in „A Handbook of Statistical Analysis Using R“ (Everitt und Hothorn, 2006).

Die Leser haben bereits gelernt, nach benötigten Funktionen im Hilfesystem zu suchen, Funktionen auf Daten anzuwenden und Ergebnisse der Analysen in Objekten zu speichern, die dann u.U. weiter verwendet werden können.

Zunächst werden häufig verwendete elementare Verfahren und wichtige Funktionen in Abschn. 7.1 beschrieben. Abschn. 7.3 ist dem Umgang mit Verteilungen und Ziehen von Stichproben gewidmet, dem auch das Generieren von Pseudo-Zufallszahlen zu Grunde liegt (s. Abschn. 7.2). In Abschn. 7.4 wird die Formelnotation zur einfachen Spezifikation statistischer Modelle eingeführt. Diese Formelnotation wird sowohl für lineare Modelle (Abschn. 7.5) als auch für viele andere Verfahren benötigt, z.B. bei den in der Zusammenfassung (Abschn. 7.6) erwähnten Klassifikationsverfahren.

7.1 Grundlegende Funktionen

In diesem Abschnitt werden einige grundlegende Funktionen beschrieben, die für die Datenanalyse wichtig sind. Neben einigen statistischen und mathematischen Hilfsfunktionen gehören dazu auch Funktionen zur Berechnung deskriptiver Kenngrößen.

Sortieren, Anordnungen, Ränge und Bindungen

Für das Sortieren eines Vektors steht die Funktion `sort()` zur Verfügung, die per Voreinstellung aufsteigend sortiert. Durch Angabe des Arguments `decreasing = TRUE` kann aber auch absteigend sortiert werden. Ein Vektor kann mit `rev()` umgekehrt werden. Das Ergebnis von `rev(x)` für einen aufsteigend sortierten Vektor `x` ist demnach ein absteigend sortierter Vektor:

```
> x <- c(5, 7, 2, 7, 8, 9)
> sort(x)
 [1] 2 5 7 7 8 9
> sort(x, decreasing = TRUE)
 [1] 9 8 7 7 5 2
> rev(sort(x))
 [1] 9 8 7 7 5 2
```

Die Anordnung der Werte innerhalb eines Vektors kann auch von Bedeutung sein. Mit der Funktion `order()` kann man sich einen Indexvektor erzeugen lassen, bei dessen Benutzung als Index des Ausgangsvektors ein sortierter Vektor entsteht:

```
> (index <- order(x))
 [1] 3 1 2 4 5 6
> x[index]
 [1] 2 5 7 7 8 9
```

Das ist vor allem zur Sortierung von Datensätzen nach darin enthaltenen Vektoren nützlich. Bei Angabe von mehreren Vektoren als Argumente zu `order()` wird versucht, Bindungen im ersten Vektor durch den zweiten Vektor aufzulösen:

```
> X <- data.frame(x = c(2, 2, 1, 1), y = c(1, 2, 2, 1))
> (temp <- order(X[,1], X[,2]))
 [1] 4 3 1 2
> X[temp, ]
   x y
4 1 1
3 1 2
1 2 1
2 2 2
```

Ränge lassen sich mit Hilfe der Funktion `rank()` bestimmen:

```
> rank(x)
 [1] 2.0 3.5 1.0 3.5 5.0 6.0
```

Mehrfach vorhandene Werte in einem Vektor können mit `duplicated()` identifiziert oder mit Hilfe von `unique()` gleich entfernt werden:

```
> duplicated(x)
 [1] FALSE FALSE FALSE  TRUE FALSE FALSE
> unique(x)
 [1] 5 7 2 8 9
```

Lage-, Streu- und Zusammenhangsmaße

In Tabelle 7.1 sind Funktionen zusammengefasst, mit denen Lage-, Streu- und Zusammenhangsmaße berechnet werden können.

Tabelle 7.1. Lage-, Streu- und Zusammenhangsmaße

Funktion	Beschreibung
`mean()`	arithmetisches Mittel, $\bar{x}$
`median()`	Median, $\tilde{x}_{0.5}$
`quantile()`	Quantile
`summary()`	Zusammenfassung eines Objekts
`mad()`	Median Absolute Deviation
`range()`	Minimum und Maximum (Spannweite)
`var()`	Varianz
`cor()`	Korrelationskoeffizient
`cov()`	Kovarianz

Die meisten Funktionen sind selbsterklärend, es sei jedoch bemerkt, dass zur Berechnung der Spannweite der Werte in einem Vektor `x` einfach die Differenz aus Maximum und Minimum mit `diff(range(x))` gebildet werden kann:

```
> x <- c(3, 5, 7, 6)
> diff(range(x))
 [1] 4
```

Für eine deskriptive Zusammenfassung lässt sich für viele Arten von Objekten `summary()` verwenden, z.B.:

```
> x <- c(3, 5, 7, 6)
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   3.00    4.50    5.50    5.25    6.25    7.00
```

Diese sehr mächtige generische Funktion liefert bei komplexeren Objekten aber auch andere Arten von Zusammenfassungen (s. z.B. Abschn. 6.1).

Die *Korrelationskoeffizienten* von Pearson, Kendall (τ) und Spearman (ρ) können mit `cor()` berechnet werden, die *Kovarianzen* mit `cov()`.

Weitere nützliche Helfer

Die *kumulierte Summe*, d.h. die Folge der Summen $s_j := \sum_{i=1}^{j} x_i$ mit $j \in \{1, \ldots, n\}$, kann mit Hilfe der Funktion `cumsum()` berechnet werden. Analog berechnet `cumprod()` die Folge der Produkte:

```
> x <- c(3, 5, 7, 6)
> cumsum(x)
 [1]  3  8 15 21       # 3, 3+5, 3+5+7, 3+5+7+6
> cumprod(x)
 [1]   3  15 105 630   # 3, 3*5, 3*5*7, 3*5*7*6
```

Eine effiziente Berechnung des *Binomialkoeffizienten* erfolgt mit `choose()`. Die *Fakultät* lässt sich mit `factorial()` ausgeben, wobei für ganze Zahlen $n \in \mathbb{N}$ gilt: $n! = \Gamma(n+1)$. Damit lässt sich die Fakultät also auch über die Gammafunktion, `gamma()`, berechnen. Im folgenden Beispiel wird der Binomialkoeffizient $\binom{6}{4}$ mit Hilfe dieser drei Funktionen berechnet:

```
> choose(6, 4)
 [1] 15
> factorial(6) / (factorial(4) * factorial(2))
 [1] 15
> gamma(7) / (gamma(5) * gamma(3))
 [1] 15
```

Für die *Diskretisierung* eines numerischen Vektors bietet sich die Funktion `cut()` an, die daraus ein Objekt der Klasse `factor` erzeugt. Es kann angegeben werden, ob an den Klassengrenzen (Argument `breaks`) rechts-offene (`right = FALSE`) oder links-offene Intervalle verwendet werden sollen. Die Namen der Faktorstufen werden gemäß den gewählten Intervallgrenzen vergeben. Anschließend kann dann, wie bei jedem anderen Objekt mit Klasse `factor`, die Anzahl des Auftretens der verschiedenen Faktorstufen mit `table()` gezählt und in einer *Tabelle* ausgegeben werden:

```
> set.seed(123)       # Initialisiert den Zufallszahlengenerator
> x <- rnorm(10)      # Erzeugt 10 standard-normalvert. Zufallsz.
```

```
> (xd <- cut(x, breaks = c(-Inf, -2, -0.5, 0.5, 2, Inf)))
 [1] (-2,-0.5]  (-0.5,0.5] (0.5,2]    (-0.5,0.5] (-0.5,0.5]
 [6] (0.5,2]    (-0.5,0.5] (-2,-0.5]  (-2,-0.5]  (-0.5,0.5]
Levels: (-Inf,-2] (-2,-0.5] (-0.5,0.5] (0.5,2] (2,Inf]

> table(xd)
xd
 (-Inf,-2]  (-2,-0.5] (-0.5,0.5]    (0.5,2]    (2,Inf]
         0          3          5          2          0
```

Die Erzeugung von Pseudo-Zufallszahlen gemäß bestimmter Verteilungen wird in den beiden folgenden Abschnitten behandelt.

Die Form eines Datensatzes, der Messwiederholungen enthält, z.B. bei verschiedenen Faktorstufen oder zu verschiedenen Zeitpunkten, kann mit Hilfe von `reshape()` sowohl aus der breiten Darstellung (pro Messwiederholung eine Variable) in die lange Darstellung (pro Messwiederholung eine neue Zeile / Beobachtung) überführt werden, als auch von der langen in eine breite Darstellung. Für Details sei auf die Hilfeseite `?reshape` verwiesen.

7.2 Zufallszahlen

Nicht nur das Ziehen von Stichproben beruht auf der Erzeugung von Pseudo-Zufallszahlen, sondern auch in Simulationen werden häufig Pseudo-Zufallszahlen benötigt, die einer bestimmten Verteilung unterliegen. Diese beiden Anwendungen werden in Abschn. 7.3 beschrieben.

Zu Pseudo-Zufallszahlen (s. z.B. Lange, 1999) sei angemerkt, dass sie

- von s.g. Pseudo-*Zufallszahlen-Generatoren* erzeugt werden,
- möglichst (fast) *keine Regelmäßigkeiten* enthalten sollen,
- möglichst *schnell* erzeugt werden sollen und
- *reproduzierbar* sein sollen, um z.B. eine Simulation wiederholen und Ergebnisse nachvollziehen und bestätigen zu können.

Von Rechnern erzeugte Pseudo-Zufallszahlen sind jedoch alles andere als zufällig, denn Sie müssen (durch Funktionen) berechnet werden und sollen außerdem reproduzierbar sein.

Leider gibt es nicht den „optimalen“ Zufallszahlen-Generator. In R ist der Zufallszahlen-Generator *Mersenne-Twister* (Matsumoto und Nishimura, 1998) Standard, der einen Kompromiss eingeht zwischen möglichst wenig Regelmäßigkeiten und zugleich möglichst hoher Geschwindigkeit. Eine Reihe anderer Generatoren steht auch zur Verfügung und kann mit der Funktion `RNGkind()` ausgewählt werden. Details und Literaturstellen zu den verschiedenen Generatoren werden in der Hilfe `?RNGkind` erwähnt.

Der Zufallszahlen-Generator wird normalerweise mit der Systemuhr initialisiert. Wenn aber reproduzierbare Ergebnisse gefordert sind (z.B. in einigen Beispielen in diesem Buch, damit die Leser sie reproduzieren können), kann mit der Funktion `set.seed()` ein Startwert gesetzt werden:

```
> set.seed(1234)            # Startwert definieren
> rnorm(2)                  # Ergebnis A
 [1] -1.2070657  0.2774292
> rnorm(2)                  # Ergebnis B
 [1]  1.084441 -2.345698
> set.seed(1234)            # Startwert redefinieren
> rnorm(2)                  # wieder Ergebnis A
 [1] -1.2070657  0.2774292
> RNGkind("Wichmann-Hill")  # anderen Generator wählen
> set.seed(1234)            # Startwert re-definieren
> rnorm(2)                  # nicht Ergebnis A (anderer Generator!)
 [1] -0.2160838  0.8444022
```

Dabei erzeugt `rnorm(2)` jeweils 2 standard-normalverteilte Pseudo-Zufallszahlen.

7.3 Verteilungen und Stichproben

Verteilungen

Für die Berechnung von Dichte-, Verteilungsfunktion, Pseudo-Zufallszahlen und Quantilen der gebräuchlichen Verteilungen sind in R bereits meist vier Typen von Funktionen implementiert, denen jeweils derselbe Buchstabe vorangestellt ist, und zwar

`d` (*d*ensity) für Dichtefunktionen,
`p` (*p*robability) für Verteilungsfunktionen,
`q` (*q*uantiles) für die Berechnung von Quantilen und
`r` (*r*andom) für das Erzeugen von Pseudo-Zufallszahlen (s. auch Abschn. 7.2).

Nach diesen „magischen“ Buchstaben folgt dann der Name der Verteilung bzw. dessen Abkürzung, z.B. `norm` für die Normalverteilung oder `unif` (uniform) für die Rechteckverteilung. Tabelle 7.2 enthält eine Aufstellung der Verteilungen, mit denen R direkt umgehen kann, wobei der erste Buchstabe (angedeutet durch „_“) der jeweiligen Funktion gemäß der Auflistung weiter oben zu ersetzen ist. Parameter der Verteilungen können jeweils durch entsprechende Argumente der Funktionen spezifiziert werden.

Die Funktion `rnorm()` erzeugt somit normalverteilte Pseudo-Zufallszahlen, während `punif()` die Verteilungsfunktion der Rechteckverteilung berechnen kann. Einige Beispiele dazu sind:

Tabelle 7.2. Verteilungen

Funktion	Verteilung
`_beta()`	Beta-
`_binom()`	Binomial-
`_cauchy()`	Cauchy-
`_chisq()`	χ^2–
`_exp()`	Exponential-
`_f()`	F-
`_gamma()`	Gamma-
`_geom()`	Geometrische-
`_hyper()`	Hypergeometrische-
`_logis()`	Logistische-
`_lnorm()`	Lognormal-
`_multinom()`	Multinomial- (nur `rmultinom()`, `dmultinom()`)
`_nbinom()`	negative Binomial-
`_norm()`	Normal-
`_pois()`	Poisson-
`_signrank()`	Verteilung der Wilcoxon (Vorzeichen-) Rangsummen Statistik (Ein-Stichproben-Fall)
`_t()`	t-
`_unif()`	Rechteck-
`_weibull()`	Weibull-
`_wilcox()`	Verteilung der Wilcoxon Rangsummen Statistik (Zwei-Stichproben-Fall)

```
> set.seed(123)     # mit Standard-Generator Mersenne-Twister
> # 5 Pseudo-Zufallszahlen einer R[3,5]-Verteilung:
> runif(5, min = 3, max = 5)
 [1] 3.575155 4.576610 3.817954 4.766035 4.880935
> # 0.25-Quantil der R[3,5]-Verteilung:
> qunif(0.25, min = 3, max = 5)
 [1] 3.5
> # Verteilungsfunktion an der Stelle 0 der N(0,1)-Verteilung:
> pnorm(0, mean = 0, sd = 1)
 [1] 0.5
```

Funktionen zur Behandlung weiterer Verteilungen sind in einigen Paketen (s. Kap. 10) vorhanden. Als ein Beispiel für viele sei hier das auf CRAN erhältliche Paket **mvtnorm** (Hothorn et al., 2001a) erwähnt, das Funktionen zum Umgang mit multivariater Gauss- (z.Zt.: `d`, `p`, `q`, `r`) und multivariater t-Verteilung (z.Zt.: `p`, `q`, `r`) bereitstellt.

Stichproben

Für das Ziehen von Stichproben kann die Funktion `sample()` verwendet werden. Deren vollständige Syntax lautet:

```
sample(x, size, replace = FALSE, prob = NULL)
```

Damit wird aus dem Vektor `x` eine Stichprobe der Größe `size` ohne Zurücklegen (`replace = FALSE`) gezogen. Sollen die Auswahlwahrscheinlichkeiten der einzelnen Elemente aus `x` nicht gleich sein, so kann zusätzlich das Argument `prob` spezifiziert werden. Die Benutzung wird anhand des folgenden Beispiels verdeutlicht:

```
> set.seed(54321)
> # Stichprobe aus den Zahlen 1:10 der Größe 4:
> sample(1:10, 4)                 # ohne Zurücklegen
 [1]  5 10  2  8
> sample(1:10, 4, replace = TRUE) # mit Zurücklegen
 [1] 3 9 1 3
> sample(letters, 5)              # geht mit beliebigen Objekten
 [1] "i" "j" "d" "p" "a"
```

7.4 Modelle und Formelnotation

Die Formelnotation zur einfachen Spezifikation statistischer Modelle wurde mit S3 im weißen Buch (Chambers und Hastie, 1992) eingeführt. Da die Modellbildung ein zentraler Punkt in der Statistik ist, baut der Erfolg der Sprache S auch auf der Formelnotation auf. So können sehr einfach selbst komplexe Zusammenhänge z.B. in (generalisierten) linearen Modellen (Abschn. 7.5) oder den in Abschn. 7.6 beschriebenen Klassifikationsverfahren mit Hilfe der Formelnotation spezifiziert werden.

Generell werden Formeln in der Form

```
y ~ model
```

angegeben. Dabei trennt die Tilde[1] („~“) die auf der linken Seite angegebene Zielvariable (abhängige Variable, response) von dem auf der rechten Seite angegebenen Rest des Modells (`model`).

Im Allgemeinen wird in der rechten Seite der Formel mit Hilfe von mathematischen Symbolen das Modell spezifiziert, das den Zusammenhang zwischen den erklärenden Variablen und der abhängigen Variablen beschreibt. Bei linearen Modellen wird die Designmatrix dann entsprechend dem Modell

[1] Wegen des Textsatzes ist die Tilde (~) hier nach oben verschoben gedruckt, in R erscheint sie aber vertikal zentriert (etwa so: ∼).

aufgebaut. Details werden im folgenden Abschn. (7.5) erläutert, der auch als Beispiel für die Formelnotation im Allgemeinen dienen möge. Dort wird auf die besondere Bedeutung der mathematischen Operatoren in der Formelnotation eingegangen.

Auch bei Erzeugung von **lattice** Grafiken spielt die Formelnotation eine große Rolle (s. Abschn. 8.2). Dort wird durch die rechte Seite einer Formel die Abszisse („x-Achse“) beschrieben, wobei als zusätzlicher Operator der vertikale Strich („`|`“) bedingende Variablen ankündigt.

7.5 Lineare Modelle

In diesem Abschnitt wird kurz auf die Thematik der linearen Modelle, der generalisierten linearen Modelle, der Regressionsanalyse und der Varianzanalyse im Zusammenhang mit der Modellspezifikation durch Formelnotation eingegangen. Ausführlichere Beschreibungen dazu findet man in Chambers und Hastie (1992) sowie in Venables und Ripley (2002). In die Regressionsanalyse mit R wird sehr leicht verständlich und detailliert von Fox (1997, 2002) eingeführt. Pinheiro und Bates (2000) geben eine umfassende Erklärung zu linearen und nichtlinearen Modellen mit (und ohne) gemischten Effekten, insbesondere im Zusammenhang mit dem Paket **nlme**, das im Laufe der Zeit durch das schon jetzt viel genutzte Paket **lme4** (Bates und Sarkar, 2006) ersetzt werden soll.

Formelnotation im linearen Modell

In Tabelle 7.3 wird die Bedeutung mathematischer Operatoren in der Formelnotation für lineare und generalisierte lineare Modelle zusammengefasst. Daraus geht hervor, dass eine multiple Regression mit zwei quantitativen erklärenden Variablen der Form

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \varepsilon_i, \qquad i = 1, \ldots, n \tag{7.1}$$

einfach durch die Formel

```
y ~ x1 + x2
```

beschrieben werden kann. Die Hinzunahme weiterer Variablen erfolgt mit dem Plus („`+`“).

Häufig sollen *alle* Variablen in das Modell aufgenommen werden, die in einem Datensatz vorliegen. Datensätze können meist mit Hilfe des Arguments `data` in den Funktionen, die Formeln benutzen, angegeben werden. Der Punkt („`.`“) gibt an, alle Variablen aufzunehmen – mit Ausnahme derer, die bereits

Tabelle 7.3. Bedeutung mathematischer Operatoren in der Formelnotation für lineare und generalisierte lineare Modelle

Operator	Bedeutung
`+`	Hinzunahme einer Variablen
`-`	Herausnahme einer Variablen (`-1` für Achsenabschnitt)
`:`	Wechselwirkung/Interaktion von Variablen
`*`	Hinzunahme von Variablen *und* deren Wechselwirkungen
`/`	hierarchisch untergeordnet („nested“). `y/z` bedeutet: `z` hat nur Wirkung innerhalb der Stufen von `y`, aber nicht global.
`^`	alle Interaktionen bis zum angegebenen Grad
`.`	alle Variablen aus dem Datensatz in das Modell aufnehmen
`I()`	innerhalb von `I()` behalten arithmetische Operatoren ihre ursprüngliche Bedeutung („Inhibit Interpretation“)

in der linken Seite des Modells (vor der Tilde) spezifiziert sind. Weitere Ausnahmen können durch das Minus („`-`“) angegeben werden. Das Modell (7.1) kann also für einen Datensatz, der genau die Elemente `y`, `x1` und `x2` enthält, noch einfacher mit der Formel `y ~ .` spezifiziert werden.

In (7.1) ist der Achsenabschnitt β_0 enthalten, dieser wurde jedoch nicht in der Formel `y ~ x1 + x2` angegeben. Tatsächlich ist der Achsenabschnitt zunächst als Voreinstellung immer im Modell enthalten, solange er nicht explizit mit „`-1`“ ausgeschlossen wird. Die Formel

```
y ~ x1 + x2 - 1
```

entspricht dann also dem Modell

$$y_i = \beta_1 x_{i1} + \beta_2 x_{i2} + \varepsilon_i, \qquad i = 1, \ldots, n. \tag{7.2}$$

Die Modellierung von Wechselwirkungen (Interaktionen) erfolgt mit dem Doppelpunkt („`:`“). Variablen, die durch ein „`*`“ voneinander getrennt sind, werden gleich inklusive ihrer Interaktionen in das Modell genommen. Die folgenden Formeln sind damit äquivalent:

```
y ~ x1 + x2 + x3 + x1:x2 + x1:x3 + x2:x3
y ~ x1 * x2 * x3 - x1:x2:x3         # ohne 3-fach Wechselwirkung
y ~ (x1 + x2 + x3)^2
```

In der letzten Formel wird der „`^`“-Operator benutzt, der in diesem Fall angibt, dass die Variablen `x1`, `x2`, `x3` inklusive aller Interaktionen bis zum Grad 2 (2-fach Wechselwirkungen) in das Modell aufgenommen werden sollen.

Wenn innerhalb einer Formel die in Tabelle 7.3 aufgeführten Symbole in ihrer Bedeutung als arithmetische Operatoren benutzt werden sollen, hilft die Funktion `I()`, innerhalb derer diese Operatoren die arithmetische Bedeutung behalten. Während die Formel

```
y ~ x1 + x2
```

zwei Variablen `x1` und `x2` in das Modell aufnimmt, wird durch die Formel

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

nur die Summe von `x1` und `x2` als einzige erklärende Variable aufgenommen.

Modellanpassung, Regression

Für die Modellanpassung stehen eine Reihe von Funktionen zur Verfügung. Im Zusammenhang mit linearen Modellen, der Regressions- und der Varianzanalyse sind besonders die in Tabelle 7.4 aufgeführten Funktionen wichtig.

Einfache lineare Regression wurde bereits in den Abschnitten 5.2.2 und 6.1 anhand des Datensatzes von Anscombe (1973) durchgeführt, dessen künstlich erzeugte Daten sehr schön die Wichtigkeit der Modelldiagnose aufzeigen. An dieser Stelle werden jene beiden Beispiele zusammengefasst und erweitert, so dass die Benutzung der Funktionen zum Arbeiten mit linearen Modellen verdeutlicht wird. Es werden zunächst mit `lm()` vier einfache lineare Regressionen durchgeführt. Die ausgegebenen Regressionsobjekte `lm1`, ..., `lm4` sind Objekte der Klasse `lm`:

```
> lm1 <- lm(y1 ~ x1, data = anscombe)
> lm2 <- lm(y2 ~ x2, data = anscombe)
> lm3 <- lm(y3 ~ x3, data = anscombe)
> lm4 <- lm(y4 ~ x4, data = anscombe)
```

Alle weiteren Funktionen werden im Beispiel nur auf das Objekt `lm1` angewendet, die Leser sollten sie aber zu Übungszwecken auch auf die anderen Objekte anwenden. Es wird sich herausstellen, dass Parameterschätzungen, p-Werte, R^2 und weitere Größen für alle Regressionen nahezu identisch sind, die Residuen sich aber stark voneinander unterscheiden. Insbesondere ist Struktur in den zu `lm2`, ..., `lm4` zu Grunde liegenden Daten vorhanden, die man mit einfachen Plots schon vor der Durchführung einer (eigentlich nicht angebrachten) Regression und spätestens bei der Residualanalyse finden sollte.

Zunächst wird im folgenden Code grobe Information zu dem Objekt `lm1` angezeigt und Koeffizienten werden mit `coef()` ausgegeben. Danach werden mehr Details zur Regression mit `summary()` angezeigt. Neben dem exakten Aufruf zu `lm()` sind das deskriptive Statistiken zu den Residuen, die Koeffizientenschätzungen in einer Tabelle zusammen mit Teststatistik und p-Werten von t-Tests, R^2 und R^2_{adj} sowie Teststatistik und p-Wert des F-Tests für das gesamte Modell:

Tabelle 7.4. Auswahl von Funktionen für die Modellanpassung sowie für die Arbeit mit Modellen und zugehörigen Objekten

Funktion	Bedeutung
`anova()`	Varianzanalyse für ein oder mehrere Objekte der Klasse `lm` bzw. `glm`
`aov()`	Varianzanalyse inkl. Aufruf von `lm()` auf allen Schichten
`glm()`	Anpassung eines generalisierten linearen Modells
`lm()`	Anpassung eines linearen Modells
`step()`	Schrittweise Regression auf einem Objekt (meist `lm` oder `glm`)
`add1()`	einen Term zu einem Modell hinzufügen - zeigt mögliche Änderungen
`coef()`	geschätzte Koeffizienten des Modells
`drop1()`	einen Term aus einem Modell entfernen - zeigt mögliche Änderungen
`fitted()`	angepasste Werte laut Modell
`formula()`	zum Erstellen von Formeln aus anderen Objekten und zum Extrahieren von Formeln aus Modellen
`influence()`	Berechnung von Einflussgrößen zur Modelldiagnose
`influence.measures()`	Berechnung von Einflussgrößen zur Modelldiagnose
`model.matrix()`	zu einer Formel oder einem Modell passende Design-Matrix
`plot()`	Grafiken zur Modelldiagnose (z.B. Residualplot, QQ-Plot)
`predict()`	Vorhersage eines neuen Wertes mit Hilfe eines angepassten Modells
`residuals()`	Residuen
`rstandard()`	standardisierte Residuen
`rstudent()`	studentisierte Residuen
`summary()`	Umfassende Informationen zum Modell (Koeffizienten, Statistiken zu Residuen, p-Werte, R^2, ...)
`update()`	komfortable Anpassung eines Modells durch kleine Änderungen

```
> lm1                          # grobe Information zum Objekt lm1
 Call:
 lm(formula = y1 ~ x1, data = anscombe)
 Coefficients:
 (Intercept)           x1
      3.0001       0.5001
```

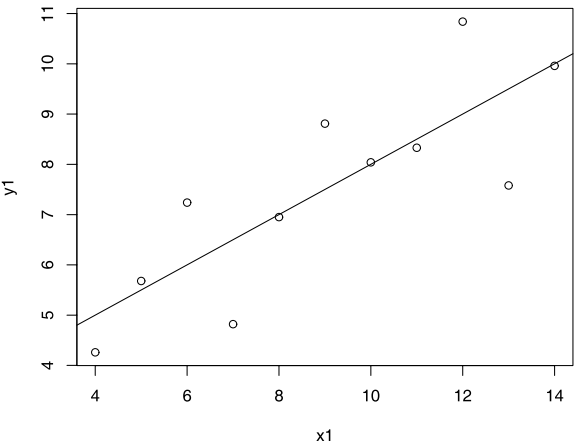

Abb. 7.1. Daten und zugehörige Regressionsgerade

```
> coef(lm1)                    # geschätzte Koeffizienten
 (Intercept)          x1
   3.0000909   0.5000909

> summary(lm1)                 # detaillierte Information zu lm1
 Call:
 lm(formula = y1 ~ x1, data = anscombe)

 Residuals:
      Min       1Q   Median       3Q      Max
 -1.92127 -0.45577 -0.04136  0.70941  1.83882

 Coefficients:
             Estimate Std. Error t value Pr(>|t|)
 (Intercept)   3.0001     1.1247   2.667  0.02573 *
 x1            0.5001     0.1179   4.241  0.00217 **
 ---
 Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

 Residual standard error: 1.237 on 9 degrees of freedom
 Multiple R-Squared: 0.6665,     Adjusted R-squared: 0.6295
 F-statistic: 17.99 on 1 and 9 DF,  p-value: 0.002170
```

Daten und Regressionsgerade werden wie folgt als Grafik (Abb. 7.1) ausgegeben:

```
> with(anscombe, plot(x1, y1))
> abline(lm1)
```

Dabei wird zunächst ein Plot von `y1` gegen `x1` erzeugt, wobei beide Variablen aus den *anscombe* Daten stammen. Mit `abline()` wird dann eine Gerade mit den Koeffizienten aus dem Objekt `lm1` erzeugt.

Die Residuen können mit `residuals()` ausgegeben werden. Für die standardisierte bzw. studentisierte Variante sind die Funktionen `rstandard()` und `rstudent()` zu benutzen. Für Modell `lm1` sind das z.B.:

```
> rbind(res = residuals(lm1), rsta = rstandard(lm1),
+       rstu = rstudent(lm1))
              1           2         3        4          5 .....
res  0.03900000 -0.05081818 -1.921273 1.309091 -0.1710909 .....
rsta 0.03324397 -0.04331791 -1.777933 1.110288 -0.1481007 .....
rstu 0.03134464 -0.04084477 -2.081099 1.126800 -0.1398012 .....
```

Die erste Zeile kann leicht nachvollzogen werden, indem die an den jeweiligen Stellen x angepassten Werte $\hat{y}$ (`fitted(lm1)`) von den wahren Werten y (`anscombe$y1`) abgezogen werden:

```
> anscombe$y1 - fitted(lm1)     # Residuen explizit berechnen
              1           2         3        4          5 .....
     0.03900000 -0.05081818 -1.921273 1.309091 -0.1710909 .....
```

Wichtige Hilfsmittel in der Residualanalyse sind Grafiken. Die generische Funktion `plot()` bringt daher Methoden für Objekte der Klassen `lm` und `glm` mit. Der Aufruf

```
> plot(lm1)                     # Grafiken für die Residualanalyse
```

erzeugt vier Grafiken (vgl. Abb. 7.2):

- *Residualplot* (Residuen gegen angepasste Werte),
- *QQ-Plot* der standardisierten Residuen (ϵ_{sta}) gegen die theoretischen Quantile einer Standard-Normalverteilung,
- *Scale-Location Plot* von $\sqrt{|\epsilon_{sta}|}$ gegen die angepassten Werte und
- *Leverage* Plot mit Höhenlinien der *Cook's Distance*. Diese Grafik identifiziert Punkte mit sehr großem individuellen Einfluss auf die Anpassung.

Für weitere Details zu Grafiken sei auf Kap. 8 verwiesen.

Nach einer Residualanalyse besteht häufig der Wunsch nach einer Anpassung des Modells, etwa Hinzunahme oder Entfernen von Variablen bzw. deren Interaktionen oder auch einfachen Transformationen. Für solche Anpassungen bietet sich bei der interaktiven Arbeit die Benutzung der Funktion `update()` an. Die Zeile

```
> lm1u <- update(lm1, log(.) ~ . - 1)
```

passt das bereits existierende Modell `lm1` so an, dass die linke Seite (früher einfach `y1`) nun durch `log(y1)` ersetzt wird. Dabei steht der Punkt (`.`) für

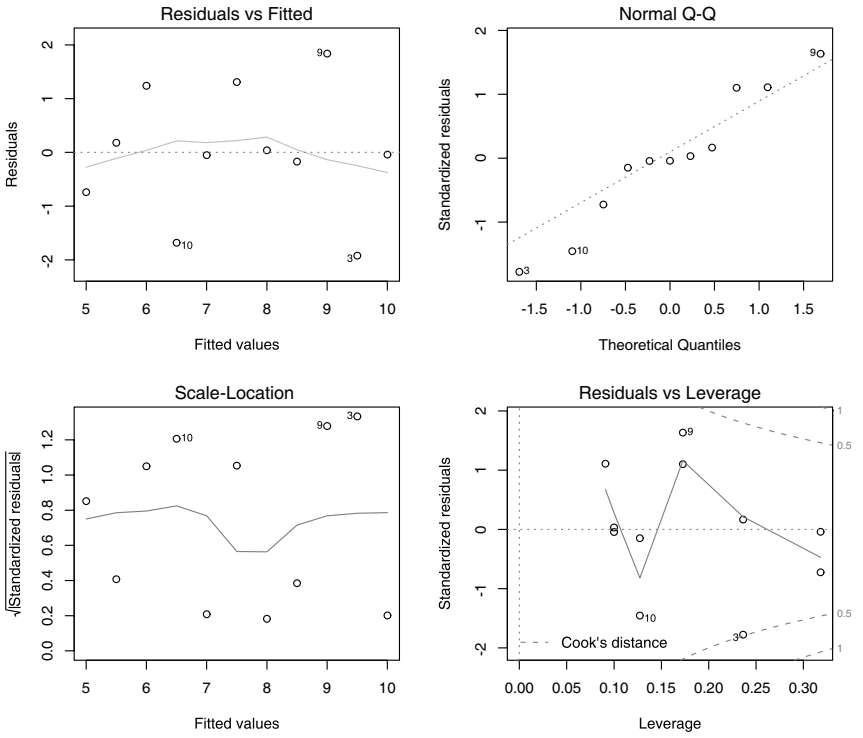

Abb. 7.2. Grafiken für die Residualanalyse

alle Elemente der alten linken Seite. Von der rechten Seite wird der Achsenabschnitt mit „`-1`“ entfernt, während der durch den Punkt spezifizierte Rest der rechten Seite des alten Modells (hier nur `x1`) erhalten bleibt:

```
> formula(lm1u)
 log(y1) ~ x1 - 1

> lm1u
 Call:
 lm(formula = log(y1) ~ x1 - 1, data = anscombe)

 Coefficients:
     x1
 0.2035
```

Die wirkliche Nützlichkeit von `update()` zeigt sich bei großen Modellen mit vielen Variablen, wenn sehr viel Tipparbeit gespart werden kann. Es ist zu beachten, dass das zuletzt angepasste Modell deutlich schlechter als die erste Anpassung ist.

kategorielle Variablen

Die Behandlung kategorieller Variablen in Modellen ist denkbar einfach. Eine kategorielle Variable sollte dazu in R als ein Objekt der Klasse `factor` repräsentiert werden. Das geschieht meist direkt automatisch beim Einlesen der Daten oder durch explizite Umwandlung in einen Faktor mit Hilfe von `factor()` (s. Abschn. 2.8).

Ist ein solcher Faktor Teil eines Modells, so werden automatisch entsprechende Dummy-Variablen pro Faktorstufe in die Design-Matrix aufgenommen, wie in dem folgenden Beispiel zu sehen ist. Dort wird mit Hilfe von `aov()` eine Varianzanalyse auf den von Beall (1942) beschriebenen Daten zur Wirksamkeit von sechs verschiedenen Insekten-Sprays durchgeführt[2].

```
> str(InsectSprays)
 'data.frame':   72 obs. of  2 variables:
  $ count: num  10 7 20 14 14 12 10 23 17 20 ...
  $ spray: Factor w/ 6 levels "A","B","C","D",..: 1 1 1 1 1 ...
> aovobj <- aov(count ~ spray, data = InsectSprays)
> summary(aovobj)
             Df Sum Sq Mean Sq F value    Pr(>F)
spray         5 88.438  17.688  44.799 < 2.2e-16 ***
Residuals    66 26.058   0.395
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
> model.matrix(aovobj)        # Design-Matrix (Ausgabe gekürzt!)
   (Intercept) sprayB sprayC sprayD sprayE sprayF
1            1      0      0      0      0      0
2            1      0      0      0      0      0
..    .      .      .      .      .      .      .
12           1      0      0      0      0      0
13           1      1      0      0      0      0
..    .      .      .      .      .      .      .
24           1      1      0      0      0      0
25           1      0      1      0      0      0
..    .      .      .      .      .      .      .
36           1      0      1      0      0      0
37           1      0      0      1      0      0
..    .      .      .      .      .      .      .
48           1      0      0      1      0      0
49           1      0      0      0      1      0
..    .      .      .      .      .      .      .
```

[2] Mehr Details zur Analyse der Daten gibt es auf der Hilfeseite `?InsectSprays` und in den dort angegebenen Literaturstellen.

```
60          1      0      0      0      1      0
61          1      0      0      0      0      1
..    .     .      .      .      .      .      .
72          1      0      0      0      0      1
attr(,"assign")
[1] 0 1 1 1 1 1
attr(,"contrasts")
attr(,"contrasts")$spray
[1] "contr.treatment"
```

Die erste Faktorstufe (`sprayA`) wird als Referenzwert betrachtet. Für jede weitere Stufe wird eine Dummy-Variable in die Design-Matrix aufgenommen, die zuletzt mit Hilfe von `model.matrix()` ausgegeben wurde. Die Namen der Dummy-Variablen setzen sich dann aus dem Namen der ursprünglichen Variablen (`spray`) und der jeweiligen Faktorstufe (`B`, `C`, ...) zusammen.

Kontraste können mit der Funktion `constrasts()` gesetzt werden, deren Hilfeseite (`?contrasts`) weitere Details liefert. Insbesondere kann mit `options(contrasts = .....)` eine Voreinstellung für Kontraste definiert werden.

7.6 Überblick: Weitere spezielle Verfahren

Einen sehr breiten Überblick über statistische Verfahren in R geben Venables und Ripley (2002) in „Modern Applied Statistics with S“. Neben den vielen in den Standard-Paketen von R implementierten Verfahren gibt es eine schier unüberschaubare Anzahl an Verfahren in anderen Paketen auf CRAN. Eine Übersicht über die wichtigsten Pakete wird in Abschn. 10.2 gegeben. Hier soll als Auswahl noch eine Übersicht über häufig benutzte Tests, Klassifikations- und Optimierungsverfahren folgen.

Übersicht über Tests

Eine Übersicht über häufig verwendete Tests in der Statistik gibt Tabelle 7.5. Die dort aufgeführten Tests sind alle im Standard-Paket **stats** zu finden. Der Aufruf der Funktionen ist intuitiv, und Antwort auf Detailfragen liefert meist die jeweilige Hilfeseite.

Wegen seiner Wichtigkeit sei hier des Weiteren der Permutationstest erwähnt, der durch die Funktion `perm.test()` in dem CRAN Paket **exactRankTests** (Hothorn, 2001) zur Verfügung gestellt wird.

Tabelle 7.5. Tests

Funktion	Beschreibung
`binom.test()`	exakter Binomialtest
`chisq.test()`	χ^2-Test für Kontingenztafeln
`fisher.test()`	Exakter Test von Fisher
`friedman.test()`	Friedman-Rangsummen-Test
`kruskal.test()`	Kruskal-Wallis-Rangsummen-Test
`ks.test()`	Kolmogorov-Smirnov-Test (1- u. 2-Stichproben)
`mcnemar.test()`	Test nach McNemar (Zeilen- u. Spaltensymmetrie in 2D-Kontingenztafeln)
`t.test()`	t-Test
`var.test()`	F-Test
`wilcox.test()`	Wilcoxon-Test (1- u. 2-Stichproben)

Klassifikationsverfahren

Hastie et al. (2001) geben eine Übersicht über Klassifikationsverfahren und zu Grunde liegende Theorie. Die Anwendung vieler dieser Verfahren in R wird von Venables und Ripley (2002) beschrieben. In Tabelle 7.6 sind Funktionen zur Durchführung bekannter und häufig verwendeter Klassifikationsverfahren und zugehörige Hilfsfunktionen zusammengestellt. Außerdem wird dort angegeben, in welchem Paket die jeweilige Funktion zu finden ist.

Die Pakete **class**, **MASS** und **nnet** gehören zu den empfohlenen Paketen in der Sammlung **VR** (Ripley, 1996; Venables und Ripley, 2002, s. auch Abschn. 10.2), die in regulär veröffentlichten R Versionen bereits enthalten ist. Dasselbe gilt für das Paket **rpart** für Klassifikations- und Regressionsbäume (Breiman et al., 1984). Diese und alle weiteren Pakete sind auch auf CRAN erhältlich.

An der TU Wien wird das Paket **e1071** entwickelt, welches eine Kollektion verschiedenartiger Funktionen enthält, darunter solche für auf *LIBSVM* (Chang und Lin, 2001) basierende Support-Vektor-Maschinen (SVM) und Naive-Bayes-Klassifikation. Karatzoglou et al. (2004) geben einen Überblick über verschiedene SVM Implementationen in R. Das Paket **ipred** (Peters et al., 2002) enthält u.a. Methoden für indirekte Klassifikation, Bagging und für die Schätzung des Klassifikationsfehlers mittels Kreuzvalidierung oder Bootstrapping. Weitere Klassifikationsverfahren, etwa eine Anbindung an SVM^{light}[3] oder Regularisierte Diskriminanzanalyse (Friedman, 1989), sowie Werkzeuge zur Beurteilung und Verbesserung von Klassifikation, darunter die von Garczarek (2002) eingeführten Gütemaße, sind in dem Paket **klaR** ent-

[3] `http://svmlight.joachims.org/`

Tabelle 7.6. Klassifikation

Funktion	Paket	Beschreibung
`knn()`	class	k-Nearest Neighbour
`lda()`	MASS	Lineare Diskriminanzanalyse
`naiveBayes()`	e1071	Naive-Bayes-Klassifikation
`NaiveBayes()`	klaR	erweiterte Funktionalität der o.g. Naive-Bayes-Klassifikation
`nnet()`	nnet	Neuronale Netze mit einer versteckten Schicht
`qda()`	MASS	Quadratische Diskriminanzanalyse
`rda()`	klaR	Regularisierte Diskriminanzanalyse
`rpart()`	rpart	Bäume für Rekursive Partitionierung und Regressionsbäume
`svm()`	e1071	Support-Vektor-Maschine (benutzt LIBSVM)
`svmlight()`	klaR	Support-Vektor-Maschine (benutzt SVM^{light})
`errorest()`	ipred	Schätzung des Vorhersagefehlers (z.B. Fehlklassifikationsrate) per Kreuzvalidierung oder Bootstrap
`predict()`		Vorhersage neuer Beobachtungen mit Hilfe gelernter Klassifikationsregeln
`stepclass()`	klaR	schrittweise Variablenselektion
`ucpm()`	klaR	Gütemaße für die Klassifikation

halten. In dem CRAN Paket **randomForest** (Liaw und Wiener, 2002) wird durch die Funktion `randomForest()` Klassifikation und Regression mit der gleichnamigen Methode nach Breiman (2001) bereitgestellt.

Die bereits in Abschn. 2.5 benutzten, von Anderson (1935) gesammelten *iris* Daten, sind im Zusammenhang mit Klassifikationsverfahren berühmt geworden und werden auch in dem hier folgenden Beispiel benutzt, das zeigt, wie Klassifikationsverfahren angewendet werden können.

Zunächst wird der Datensatz in einen Trainingsdatensatz (`train`, 90% der Daten) und einen Testdatensatz (`test`, 10% der Daten) aufgeteilt, damit die Güte der später zu lernenden Klassifikationsregel eingeschätzt werden kann[4]:

```
> set.seed(123)
> index <- sample(nrow(iris))
> train <- iris[-index[1:15], ]
> test  <- iris[ index[1:15], ]
```

Nun wird mit Hilfe der Formelnotation (s. Abschn. 7.4) ein Modell spezifiziert, das aussagt, welche die Zielvariable (eine kategorielle Variable) ist, und welche

[4] Besser wäre es, die Fehlklassifikationsrate kreuzvalidiert zu bestimmen, z.B. mit Hilfe der Funktion `errorest()` aus dem Paket **ipred**.

weiteren Variablen die Zielvariable erklären sollen. In unserem Beispiel ist das eine der beiden folgenden Formeln:

```
Species ~ .
Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
```

Damit wird spezifiziert, dass die Schwertlilien-Sorte (`Species`) durch alle anderen („`.`“) im Datensatz vorkommenden Variablen erklärt werden soll. Diese Formel kann in fast allen Funktionen, die ein Klassifikationsverfahren implementieren, zur Spezifikation des Modells genutzt werden, z.B. in:

```
> library("MASS")
> library("rpart")
> ldao <- lda  (Species ~ ., data = train)
> rpo  <- rpart(Species ~ ., data = train)
```

Mit den soeben gelernten Klassifikationsregeln kann dann mittels `predict()` für die Testdaten eine Klasse (bzw. Klassenwahrscheinleichkeiten) vorhergesagt und mit den wahren Werten verglichen werden:

```
> ldap <- predict(ldao, newdata = test)$class
> rpp  <- predict(rpo,  newdata = test, type = "class")
> mean(ldap != test$Species)
 [1] 0.06666667
> mean(rpp  != test$Species)
 [1] 0.1333333
```

Für die konkrete Aufteilung in Trainings- und Testdaten macht der Klassifikationsbaum also doppelt so viele Fehler wie die lineare Diskriminanzanalyse.

Zu Übungszwecken sei empfohlen, weitere Klassifikationsverfahren an diesem Datensatz auszuprobieren, Kreuzvalidierung mit Hilfe von `errorest()` durchzuführen und auch verschiedene Konstellationen von erklärenden Variablen zu benutzen, die Auswahl also auf 2-3 Variablen zu beschränken.

Optimierungsverfahren

In vielen Fällen erweisen sich Funktionen als nützlich, die andere (mathematische) Funktionen numerisch optimieren können. In Tabelle 7.7 sind einige dieser Funktionen zusammengestellt. Für zu optimierende Funktionen, die von mehr als einer Variablen abhängen, ist `optim()` zu empfehlen. Bei dieser Funktion handelt es sich um ein wahres Multitalent, das *Nelder-Mead*, *quasi-Newton*, Optimierung unter Restriktionen und einige weitere (auch gradientenfreie) Verfahren beherrscht, darunter auch *Simulated Annealing*.

Generell gilt, dass statt der voreingestellten Minimierung durch Umkehrung des Vorzeichens auch maximiert werden kann. Nullstellen können meist durch Minimierung der quadrierten Funktion gefunden werden. Weiter sei

Tabelle 7.7. Optimierungsverfahren

Funktion	Beschreibung
`nlm()`	Minimierung einer Funktion nach Newton (auch mehrdimensional)
`optim()`	Sammlung von Optimierungsverfahren (s. Text)
`optimize()`	(eindimensionale) Optimierung innerhalb vorgegebener Intervallgrenzen
`polyroot()`	komplexe Nullstellen von Polynomen
`uniroot()`	Nullstellen einer Funktion

bemerkt, dass numerische Optimierungsverfahren je nach zu optimierender Funktion im Fall von vorhandenen lokalen Extremstellen u.U. Probleme beim Finden des globalen Optimums haben (siehe z.B. Lange, 1999). Ebenso können auch nicht stetige Funktionen zu Problemen führen.

Im Zusammenhang mit Optimierungsverfahren sei die Funktion `deriv()` zur analytischen partiellen Differenzierung einer Funktion erwähnt.

8

Grafik

Ihaka und Gentleman (1996) beschreiben R mit dem Titel „R: A Language for Data Analysis and *Graphics*“ und auch Becker und Chambers (1984) schrieben schon von „S, an Interactive Environment for Data Analysis and *Graphics*“. Demnach liegt eine der besonderen Stärken von R im Grafikbereich. Es ist nicht nur möglich, explorative Grafiken für die interaktive Analyse von Daten sehr schnell und einfach zu erstellen, sondern auch Grafiken sehr hoher Qualität für Publikationen oder Präsentationen zu erzeugen. Als Programmiersprache besitzt R die Fähigkeit, Grafikproduktion zu automatisieren. So können viele Grafiken desselben Typs ohne Eingreifen des Benutzers erzeugt werden, z.B. zur täglichen Zusammenfassung von Berechnungen auf einer Web-Seite. Murrell (2005), der Autor der R Grafik Systeme, widmet dem Thema ein ganzes Buch mit vielen Beispielen.

Zu unterscheiden gilt es die „konventionellen“ Grafikfunktionen (s. Abschn. 8.1) und die in Cleveland (1993) eingeführten *Trellis* Grafiken, die in R in dem Paket **lattice**, welches auf **grid** basiert, implementiert sind (s. Abschn. 8.2). Die drei Begriffe „lattice“, „grid“ und „trellis“ lassen sich alle mit „Gitter“ übersetzen. Entsprechend stellen Trellis Grafiken oft viele Grafiken desselben Typs innerhalb eines Gitters dar, z.B. je Kategorie einer kategoriellen Variablen im Datensatz eine Grafik.

Für dynamische und interaktive Grafik ist das Grafiksystem von R nicht ausgelegt. Es gibt allerdings inzwischen eine Reihe von Paketen, von denen entsprechende Funktionalität bereitgestellt wird. Diese werden in Abschn. 8.3 besprochen.

8.1 Konventionelle Grafik

In diesem Abschnitt werden die „konventionellen“ Grafikfunktionen beschrieben. Dabei handelt es sich nicht nur um Funktionen, die die verschiedensten Arten kompletter Grafiken produzieren können (z.B. Boxplots, QQ-Plots, Stabdiagramme; s. Abschn. 8.1.2), sondern auch um solche Funktionen, mit denen Elemente zu Grafiken hinzugefügt werden können bzw. Grafiken von Grund auf konstruiert werden können (8.1.4, 8.1.6). Insbesondere wird auch auf die Ausgabe von Grafiken (8.1.1), die Einstellmöglichkeiten von Parametern (8.1.3) und mathematische Beschriftung (8.1.5) von Grafiken eingegangen.

8.1.1 Ausgabe von Grafik – *Devices*

Wenn eine Grafik erzeugt wird, muss zunächst geklärt werden, auf welchem Gerät (*Device*) die Ausgabe erfolgen soll und dieses gestartet werden. Tabelle 8.1 listet einige mögliche Devices auf.

Sollte eine Grafik erzeugt werden, ohne dass zuvor ein Device gestartet wurde, so wird das in `options("device")` eingetragene Gerät gestartet. Als Voreinstellung wird beim interaktiven Arbeiten mit R bei der Erzeugung einer Grafik das Device für Bildschirmgrafik (z.B. `X11()`) gestartet, während

Tabelle 8.1. Devices

Funktion	Beschreibung
`bitmap()`	Erzeugt mit `postscript()` eine Grafik, die von GhostScript in das gewünschte (nicht nur Bitmap-)Format konvertiert wird (erfordert Konfiguration unter Windows).
`jpeg()`	JPEG
`pdf()`	PDF
`pictex()`	PicTEX zum Import in LaTeX-Dokumente
`png()`	PNG (ähnlich GIF)
`postscript()`	PostScript
`xfig()`	XFig – Grafik kann als solche einfach bearbeitet werden
`X11()`	Grafik in Fenster auf dem Bildschirm
	nur unter Windows*:*
`bmp()`	Bitmap
`win.metafile()`	Windows Metafile
`win.print()`	Ausgabe an einen angeschlossenen Drucker
`windows()`	Grafik in Fenster auf dem Bildschirm (s. `X11()`)

im nicht interaktiven (BATCH) Betrieb oder bei nicht vorhandener Möglichkeit der Bildschirmausgabe die Ausgabe an das `postscript()` Device in eine PostScript Datei 'Rplots.ps' im aktuellen Arbeitsverzeichnis erfolgt. Im nicht interaktiven Modus sollte man bei der Grafikerzeugung besser immer explizit ein Device starten, auch wenn eine PostScript Ausgabe gewünscht ist.

Einige Devices (z.B. `postscript()` und `pdf()`) unterstützen auch mehrseitige Ausgaben. Für jede in das Bild gezeichnete neue Grafik wird dann eine neue Seite angefangen. Auf diese Weise kann auch eine große Anzahl an Bildern in einer Datei zusammengefasst werden.

Nachdem die auszugebenden Grafiken von R erzeugt worden sind, darf nicht vergessen werden, das Device mit `dev.off()` zu schließen. Bei vielen Formaten ist die resultierende Datei sonst nicht korrekt interpretierbar.

Devices verhalten sich wie Papier, auf das mit einem Stift gemalt wird. Einmal erzeugte Elemente in der Grafik lassen sich nicht löschen, sondern nur weiter übermalen. Das zuletzt eingefügte Element übermalt alle zuvor eingefügten Elemente. Es macht daher Sinn, den Code zur Erzeugung von Grafiken in einem Editor zu schreiben, so dass die Grafik einfach durch das Ausführen einiger gespeicherter Zeilen Code schnell reproduziert werden kann, nachdem ein Fehler gemacht worden ist.

Es ist möglich, mehrere Devices gleichzeitig zu öffnen, also mehrere Bildschirmgrafiken zu betrachten bzw. eine Bildschirmgrafik zu betrachten und zwischendurch eine Grafikausgabe in einer Datei zu erzeugen. Das zuletzt geöffnete Device ist das aktive, in das „hineingemalt" wird.

Als Beispiel wird hier eine einfache Grafik in eine PDF Datei 'testgrafik.pdf' im aktuellen Arbeitsverzeichnis ausgegeben:

```
> pdf("testgrafik.pdf")          # Device starten
> plot(1:10)                     # Grafik(en) erzeugen
> #  ...  weitere Grafik(en) erzeugen bzw. Elemente hinzufügen
> dev.off()                      # Device wieder schließen
 null device
           1
```

Die Ausgabe von `dev.off()` enthält die Information, welches Device nach dem gerade erfolgten Schließen des zuletzt aktiven Devices das jetzt aktive ist. Im Beispiel war kein weiteres Device mehr geöffnet (`null device`). Wäre unter Windows noch eine Bildschirmgrafik geöffnet gewesen, hätte `dev.off()` die Information

```
windows
      2
```

ausgegeben.

Der übliche Weg zum Erzeugen einer Grafik ist, diese zunächst interaktiv als Bildschirmgrafik ausgeben zu lassen, bis der Grafik-Inhalt den eigenen

Wünschen entspricht. Der dazu verwendete Code wird dabei in einem Editor geschrieben, so dass er später erneut ausgeführt werden kann. Erst wenn alles den Wünschen entspricht, wird man ein anderes Device öffnen, z.B. eines für die Ausgabe als PDF, den Code erneut ausführen und das Device wieder schließen. Abschließend können die Ergebnisse in der Datei mit einem entsprechenden Betrachter kontrolliert werden.

In R unter Windows ist es auch möglich, eine Bildschirmgrafik durch „Klicken“ im Menü in verschiedenen Formaten zu speichern. Es ist aber zu empfehlen, besser sofort geeignete Devices zu verwenden.

Einige weitere Funktionen erleichtern das Arbeiten mit Devices. Ein konkretes offenes Device kann mit `dev.set()` als „aktiv“ markiert werden, wobei zu beachten ist, dass immer nur genau ein Device aktiv ist. Die Liste aller geöffneten Devices (außer dem `null device`) wird von `dev.list()` ausgegeben. Mit `graphics.off()` werden alle geöffneten Devices geschlossen.

Mit Hilfe von `dev.copy()` kann der Inhalt eines Devices in ein anderes kopiert werden. Das neue Device wird daraufhin aktiv und muss noch geschlossen werden, falls keine weiteren Änderungen erfolgen sollen. Kopieren des Inhalts von Devices ist z.B. dann nützlich, wenn die aktuelle evtl. nicht reproduzierbare Bildschirmgrafik in einer Datei gespeichert werden soll. Analog kopiert `dev.print()` in ein PostScript Device und startet anschließend den Druck. Die spezielle Form `dev.copy2eps()` kopiert direkt in eine (Encapsulated) PostScript Datei. Auf den Hilfeseiten dieser Funktionen werden auch Hinweise zu weiteren, weniger häufig benutzten Funktionen gegeben.

Einige Eigenschaften von Grafiken hängen von dem verwendeten Device ab. Dazu gehören insbesondere die Möglichkeit zum Erzeugen teilweise transparenter Objekte (s. Abschn. 8.1.3) mit Hilfe eines *alpha*-Kanals, um Überlagerungen zu zeigen, und das Verwenden spezieller Schriftarten (s. jeweils Murrell, 2004). Das Einbinden von Schriftarten in PostScript oder PDF Dateien, wie es viele Verlage fordern, kann mit Hilfe der Funktion `embedFonts()` erfolgen (Murrell und Ripley, 2006).

Konkrete Eigenschaften und Parameter eines Devices können mit Hilfe der Funktion `par()`, die in Abschn. 8.1.3 beschrieben wird, abgefragt und gesetzt werden.

8.1.2 *High-level* Grafik

High-level Grafikfunktionen sind solche Funktionen, die eine vollständige Grafik mit Achsen, Beschriftungen und mit einer Darstellung für die Daten erzeugen können. Einige wichtige und nützliche *High-level* Grafikfunktionen für bestimmte Darstellungsformen sind in Tabelle 8.2 aufgelistet.

Tabelle 8.2. *High-level* Grafikfunktionen (Auswahl)

Funktion	Beschreibung
`plot()`	kontextabhängig – generische Funktion mit vielen Methoden
`barplot()`	Stabdiagramm
`boxplot()`	Boxplot
`contour()`	Höhenlinien-Plot
`coplot()`	Conditioning-Plots
`curve()`	Funktionen zeichnen
`dotchart()`	Dotplots (Cleveland)
`hist()`	Histogramm
`image()`	Bilder (3. Dim. als Farbe)
`mosaicplot()`	Mosaikplots (kategorielle Daten)
`pairs()`	Streudiagramm-Matrix
`persp()`	perspektivische Flächen
`qqplot()`	QQ–Plot

Einige dieser Funktionen, insbesondere `plot()`, sind generisch und bringen eine Reihe von Methoden mit (s. auch Kap. 6). Daher können sie auf verschiedene Klassen von Objekten angewendet werden und reagieren „intelligent“, so dass für jedes dieser Objekte eine sinnvolle Grafik erzeugt wird. Die Funktion `plot()` erzeugt beispielsweise ein Streudiagramm (Scatterplot) bei Eingabe von zwei Vektoren, Grafiken zur Modelldiagnose (z.B. Residualplot) bei Eingabe eines *lm*-Objektes (lineares Modell, s. Abschn. 7.5) und eine per Linie verbundene Zeitreihe bei Zeitreihenobjekten. Es ist also mit einem kurzen Aufruf sehr schnell ein Streudiagramm, Histogramm, Boxplot o.Ä. erstellt, was gerade für die interaktive Datenanalyse sehr wichtig ist.

Die meisten Argumente von Grafikfunktionen können als Vektoren angegeben werden. Dazu gehören nicht nur Argumente, die die Datenpunkte repräsentieren, z.B. `x` und `y` zur Darstellung eines Streudiagramms, sondern auch Argumente, die zur Spezifikation von Farben und Symbolen (s. Abschn. 8.1.3) dienen. Dadurch werden Schleifen für das nachträgliche Einfügen von einzelnen Elementen in die Grafik oft unnötig.

Das folgende Beispiel zeigt – erneut anhand der *iris* Daten – wie einfach es ist, recht übersichtliche Grafiken bei interaktiver Datenanalyse zu erstellen (s. Abb. 8.1):

```
> attach(iris)                  # Daten in den Suchpfad einhängen
> plot(Petal.Length, Petal.Width, pch = as.numeric(Species))
> hist(Petal.Length)
> detach(iris)
```

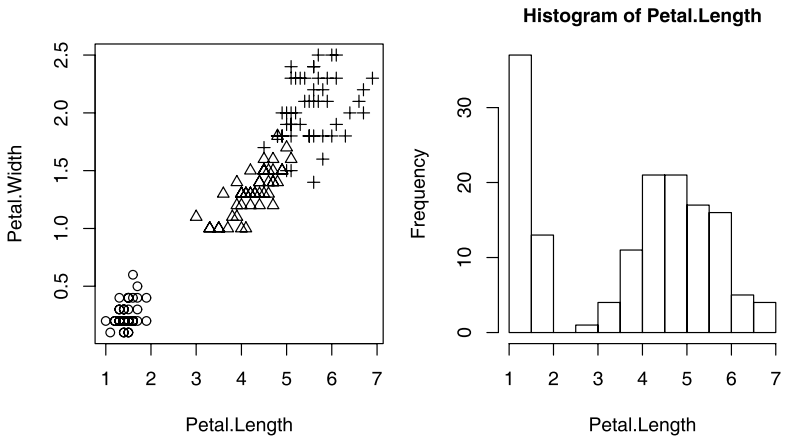

Abb. 8.1. Beispielgrafiken (Streudiagramm, Histogramm) mit den *iris* Daten

Mit dem Argument `pch` (von „*p*oint *ch*aracter“, s. auch Abschn. 8.1.3) innerhalb des Aufrufs von `plot` wird je Pflanzenart ein anderes Symbol vergeben. Anstelle von `pch` könnte bei interaktiver Arbeit am Bildschirm auch das Argument `col` für eine Farbkodierung verwendet werden.

Als weitere Beispiele für *High-level* Grafikfunktionen sind Boxplots in der Beispielsitzung (Abschn. 2.5, Abb. 2.1 auf S. 22) und ein Streudiagramm mit Regressionsgerade und Diagnoseplots für das lineare Modell in Abschn. 7.5 (Abb. 7.1 und 7.2 auf S. 139 f.) zu finden.

Mehrdimensionale Daten

Es gibt verschiedene Möglichkeiten, Daten höherer Dimension zu visualisieren. In 2-dimensionalen Darstellungen können Farbe (über das Argument `col`), Form (`pch`) und Größe (`cex`) von Symbolen verwendet werden, um weitere Dimensionen darzustellen.

Höhenlinien in zwei Dimensionen stellt `contour()` dar, während `image()` die Höhe von Punkten auf einem 2-dimensionalen Gitter mit Hilfe von Farben darstellt. Eine Kombination beider Ansätze wird durch die Funktion `filled.contour()` bereitgestellt.

All diese Funktionen (`contour()`, `image()` und `filled.contour()`) erwarten die drei Argumente `x`, `y` und `z`, wobei `z` eine Matrix der Werte über einem Gitter der 2-dimensionalen Grundfläche ist. Die Argumente `x` und `y` sind Vektoren der Werte an den Achsen der Grundfläche, zu denen die Einträge in `z` korrespondieren. Alternativ kann nur eine Matrix `x` angegeben werden. In dem Fall wird angenommen, dass die Werte der Matrix `x` an den Achsen der Grundfläche zu Punkten, die äquidistant zwischen 0 und 1 verteilt sind,

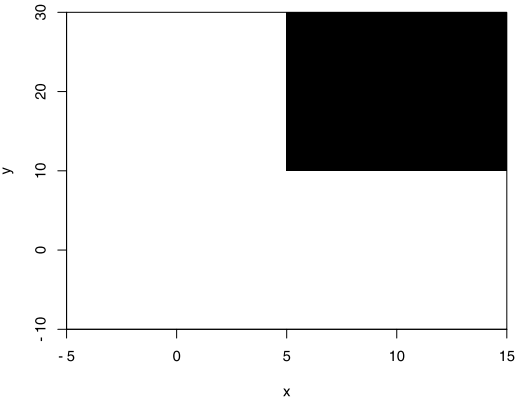

Abb. 8.2. Beispielgrafik für die Funktion `image()`

korrespondieren. Als Beispiel werden hier `x` und `y` die Werte 0 und 10 bzw. 0 und 20 annehmen. Die Matrix `z` nimmt das äußere Produkt beider Vektoren an und wird dann von `image()` dargestellt:

```
> x <- c(0, 10)
> y <- c(0, 20)
> (z <- outer(x, y))
      [,1] [,2]
 [1,]    0    0
 [2,]    0  200
> image(x, y, z, col = c("transparent", "black"))
```

Das entstehende Bild (Abb. 8.2) hat nun 4 Flächen (jeweils eine Fläche pro Eintrag in der Matrix `z`), die entsprechend den Werten für `x` und `y` an den jeweiligen Achsen zentriert sind. Drei der Flächen, nämlich die mit Wert 0, erhalten transparente Farbe, während die vierte Fläche mit Wert 200 in schwarz eingezeichnet wird. An welchen Stellen sich die Farbwerte ändern bzw. von `contour()` Linien eingezeichnet werden, wird automatisch bestimmt, kann jedoch über entsprechende Argumente (`breaks` für `image()` und `nlevels` oder `levels` für die anderen Funktionen) manuell gesetzt werden.

Drei-dimensionale perspektivisch gezeichnete Flächen können mit Hilfe von `persp()` erstellt werden. Die Spezifikation der Argumente `x`, `y` und `z` erfolgt analog zu der zuvor beschriebenen Vorgehensweise für `image()`. Die Flächen werden dargestellt mit Hilfe von vertikal und horizontal über der Grundfläche gespannten Linien, die in `z`-Richtung eine entsprechende Höhe annehmen. Damit werden durch diese Linien viereckige Facetten im 3D-Raum gebildet. Sollen diese Facetten eingefärbt werden, so kann das durch Angabe einer Matrix mit Farbwerten (s. Abschn. 8.1.3) erfolgen. Dabei ist zu beach-

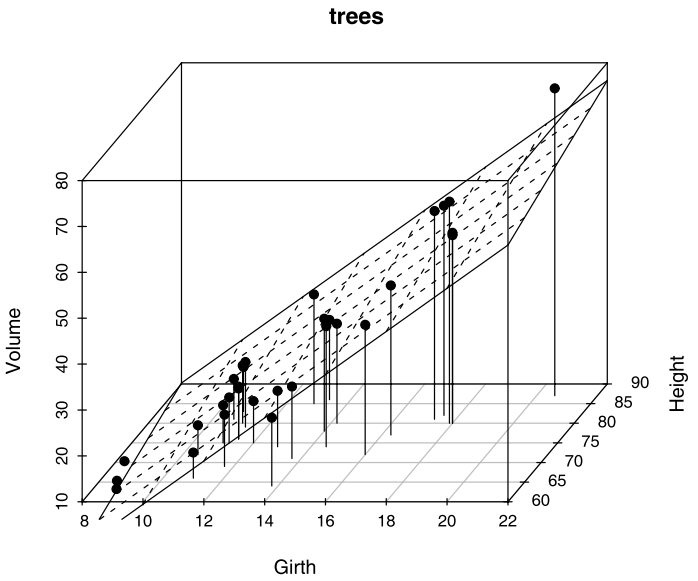

Abb. 8.3. Beispielgrafik für die Funktion `scatterplot3d()`

ten, dass es für eine $n \times m$ Matrix `z` genau $(n-1) \times (m-1)$ Facetten zwischen den Linien gibt. Die Matrix der Farbwerte muss also eine Zeile und eine Spalte weniger als `z` enthalten. Eine Reihe von Beispielen für `persp()` erhält man durch Eingabe von `demo(persp)` und auf der Hilfeseite.

Das CRAN-Paket **scatterplot3d** (Ligges und Mächler, 2003) enthält die gleichnamige Funktion `scatterplot3d()`, mit der Punktwolken in drei Dimensionen dargestellt werden können. Die Funktion liefert selbst wieder Funktionen zurück, mit denen weitere Elemente, z.B. weitere Punkte oder eine Regressionsebene, zu der bestehenden Grafik hinzugefügt werden können. Das folgende Beispiel (s. Abb. 8.3) verdeutlicht das Vorgehen, bei dem eine der in dem von `scatterplot3d()` zurückgegebenen Objekt `s3d` enthaltenen Funktionen benutzt wird:

```
> install.packages("scatterplot3d") # Paket bei Bedarf installieren
> library("scatterplot3d")          # ... und laden
> s3d <- scatterplot3d(trees, type = "h", angle = 55,
                       scale.y = 0.7, pch = 16, main = "trees")
> my.lm <- with(trees, lm(Volume ~ Girth + Height))
> s3d$plane3d(my.lm, lty.box = "solid") # Regressionsebene zeichnen
```

In dem Paket **lattice**, welches in Abschn. 8.2 beschrieben wird, gibt es die Funktionen `wireframe()` (ähnlich `persp()`) zur Darstellung von Flächen und `cloud()` (ähnlich `scatterplot3d()`) zur Darstellung von Punktwolken.

Einige weitere Pakete und externe Werkzeuge für dynamische und interaktive Grafiken, die auch 3D-Darstellungen beherrschen, werden in Abschn. 8.3 beschrieben.

8.1.3 Konfigurierbarkeit – `par()`

Während die im Beispiel des letzten Abschnitts erzeugten Grafiken (Abb. 8.1) für die interaktive Datenanalyse völlig ausreichen, möchten Benutzer die Grafiken für Präsentationen oder Publikationen aber häufig noch anpassen. Das kann geschehen durch Wahl anderer Beschriftungen (z.B. Überschrift, Achsenbeschriftung), anderer Linienstärken, Skalierung oder Farben (z.B. weiße Linien auf dunklem Hintergrund für PowerPoint oder PDF Präsentationen).

Es gibt eine Vielzahl von Parametern, die an die meisten Grafikfunktionen in Form von Argumenten übergeben werden können. Dazu gehören vor allem diejenigen, die in den Hilfen `?plot` und `?plot.default` aufgeführt sind, aber zum größten Teil auch die in `?par` aufgeführten Argumente. Mit der Funktion `par()` werden die wichtigsten Voreinstellungen im Grafikbereich durchgeführt. Für Anfänger im Grafikbereich von R empfiehlt es sich, die Hilfe dieser mächtigen Funktion (evtl. auch zusätzlich die Hilfe `?plot.default`) einmal auszudrucken und durchzulesen, damit ein Überblick über die Funktionalität der vielen verschiedenen Argumente gewonnen werden kann.

Insgesamt ist zu empfehlen, einmalige Einstellungen bei dem Aufruf der Grafikfunktion selbst anzugeben, eine Änderung für mehrere Grafiken aber mit `par()` vorzunehmen. Einige Einstellungen können ausschließlich mit `par()` geändert werden, während einige spezielle Grafikfunktionen hingegen gewisse Voreinstellungen von `par()` überschreiben.

Einige der am häufigsten gebrauchten Argumente in Grafikfunktionen und `par()` sind in Tabelle 8.3 zu finden.

Farben und Symbole

Farben können auf verschiedene Art und Weise angegeben werden. Dabei bezeichnet eine ganze Zahl die Farbe an entsprechender Stelle in der aktuellen Farbpalette, die mit Hilfe von `palette()` abgefragt und gesetzt werden kann. In der voreingestellten Palette steht 1 für Schwarz, 2 für Rot usw.:

```
> palette()
 [1] "black"   "red"     "green3"  "blue"
 [5] "cyan"    "magenta" "yellow"  "gray"
```

R kennt offenbar auch eine ganze Reihe von Farben, deren Namen (als Zeichenfolge in Anführungszeichen anzugeben, z.B.: `"green"`) entsprechenden RGB (Rot/Grün/Blau) Werten zugeordnet werden. Die Funktion `colors()` gibt

Tabelle 8.3. Einige häufig benutzte Argumente in Grafikfunktionen und `par()`

Funktion	Beschreibung
`axes`	Achsen sollen (nicht) eingezeichnet werden
`bg`	Hintergrundfarbe
`cex`	Größe eines Punktes bzw. Buchstaben
`col`	Farben
`las`	Ausrichtung der Achsenbeschriftung
`log`	Logarithmierte Darstellung
`lty, lwd`	Linientyp (gestrichelt, ...) und Linienbreite
`main, sub`	Überschrift und „Unterschrift“
`mar`	Größe der Ränder für Achsenbeschriftung etc.
`mfcol, mfrow`	mehrere Grafiken in einem Bild
`pch`	Symbol für einen Punkt
`type`	Typ (l=Linie, p=Punkt, b=beides, n=nichts)
`usr`	Ausmaße der Achsen auslesen
`xlab, ylab`	x-/y-Achsenbeschriftung
`xlim, ylim`	zu plottender Bereich in x-/y- Richtung
`xpd`	in die Ränder hinein zeichnen

eine vollständige Liste der bekannten Namen aus. Ausgenommen ist transparente Farbe (`"transparent"`), da diese Farbe nicht auf allen Devices zur Verfügung steht.

Als dritte Möglichkeit bietet es sich an, die RGB Werte direkt in Hexadezimaldarstellung anzugeben, wenn eine Feineinstellung im gesamten RGB Farbraum benötigt wird. Dabei hat Schwarz den Wert `"#000000"`, Weiß `"#FFFFFF"` und Rot `"#FF0000"`. Die Funktionen `col2rgb()`, `hcl()`, `hsv()`, `rgb()` und `rainbow()` helfen dabei, aus einer jeweils intuitiven Größe die RGB Darstellung zu berechnen.

Ein besonderer Einsatz von „Farbe“ ist die Verwendung des *alpha*-Kanals, der es ermöglicht, teilweise transparente Objekte zu erzeugen, so dass Überlagerungen verschiedener Objekte sichtbar werden. Objekte mit teilweiser Transparenz lassen sich allerdings nur auf den Devices *Quartz* (auf dem Macintosh) und PDF erzeugen, wobei die PDF Version mindestens 1.4 sein muss. Dazu muss unbedingt das Argument `version = "1.4"` im Aufruf von `pdf()` gesetzt werden. Die Stärke der Transparenz kann unter anderem bei den Funktionen `hcl()`, `hsv()` und `rgb()` mit dem Argument `alpha`, das zwischen 0 (transparent) und 1 (nicht transparent) liegen muss, angegeben werden. Details beschreibt Murrell (2004). Das Ergebnis des folgenden Beispiels kann in Abb. 8.4 betrachtet werden:

```
> x <- rnorm(2000)
> y <- rnorm(2000)
```

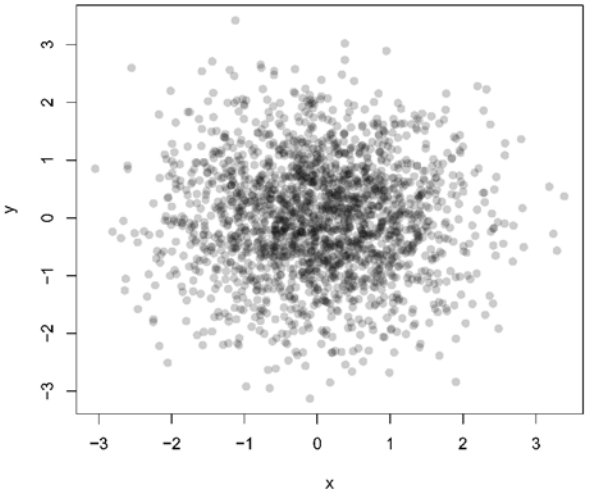

Abb. 8.4. Sichtbare Überlagerung durch teilweise transparente Punkte

```
> pdf("Transparenz.pdf", version = "1.4")
> plot(x, y, col = rgb(0, 0, 0, alpha = 0.2), pch = 16)
> dev.off()
 null device
           1
```

Bei Symbolen kann analog zu Farben der Symboltyp entweder durch eine ganze Zahl oder durch ein in Anführungszeichen gesetztes einzelnes Zeichen angegeben werden. Wenn ein Symbol aus mehr als einem Zeichen bestehen soll, so kann nach Erstellung einer Grafik ohne die Eintragung von Daten (durch Setzen des Arguments `type = "n"`) Text mit Hilfe von `text()` (s. Abschn. 8.1.4) an konkreten Stellen hinzugefügt werden.

Das Setzen von `type = "n"` ist immer dann sinnvoll, wenn Beschriftungen, Achsen und Koordinatensystem einer Grafik (oder Teile davon) initialisiert, aber Datenpunkte oder andere Elemente hinterher einzeln hinzugefügt werden sollen.

Ein Beispiel

In dem folgenden Beispiel sollen einige Konfigurationsmöglichkeiten vorgestellt werden, wobei wegen des schwarz-weiß Druckes dieses Buches die Verwendung von Farben unterbleibt:

```
> set.seed(123)
> x <- rnorm(100)       # 100 N(0,1)-verteilte Zufallszahlen
> par(las = 1)          # alle Achsenbeschriftungen horizontal
```

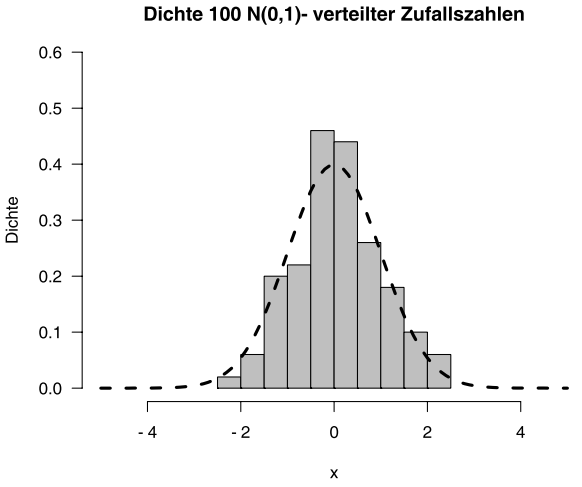

Abb. 8.5. Beispielgrafik – Histogramm und Dichtefunktion

```
> # Beschriftetes und manuell skaliertes Histogramm:
> hist(x, main = "Dichte 100 N(0,1)-verteilter Zufallszahlen",
+      freq = FALSE, col = "grey", ylab = "Dichte",
+      xlim = c(-5, 5), ylim = c(0, 0.6))
> # Hinzufügen der theor. Dichtefunktion - dick und gestrichelt:
> curve(dnorm, from = -5, to = 5, add = TRUE, lwd = 3, lty = 2)
```

Nachdem 100 Zufallszahlen gemäß einer Standard-Normalverteilung gezogen wurden, wird mit Hilfe von `par(las = 1)` für alle weiteren Aktionen im aktuellen Device die Ausrichtung der Achsenbeschriftungen auf 1 (d.h. horizontal) gesetzt. Danach wird ein Histogramm erzeugt, bei dem explizit die Überschrift und die Beschriftung der y-Achse gesetzt werden. Außerdem wird angegeben, dass die Flächen in grauer Farbe (`col = "grey"`) gefüllt und keine absoluten Häufigkeiten (`freq = FALSE`) verwendet werden sollen. Die x-Achse soll sich von -5 bis 5 erstrecken, und die y-Achse soll auf dem Intervall von 0 bis 0.6 dargestellt werden. Schließlich wird mit `curve()` die Dichte einer Standard-Normalverteilung (`dnorm`, ohne Argumente) zu der Grafik hinzugefügt, indem durch `add = TRUE` erzwungen wird, dass keine eigenständige neue Grafik erzeugt wird. Diese Funktion wird auch auf der x-Achse von -5 bis 5 mit einer dicken (`lwd = 3`), gestrichelten (`lty = 2`) Linie gezeichnet. Das Ergebnis dieser Aufrufe ist in Abb. 8.5 zu sehen.

Regionen, Ränder und mehrere Grafiken in einem Bild

Ein Grafik Device enthält mehrere Regionen (s. Abb. 8.6). Die „*device region*“ enthält die gesamte Grafik, welche aus mehreren Teil-Grafiken und den *äußе-*

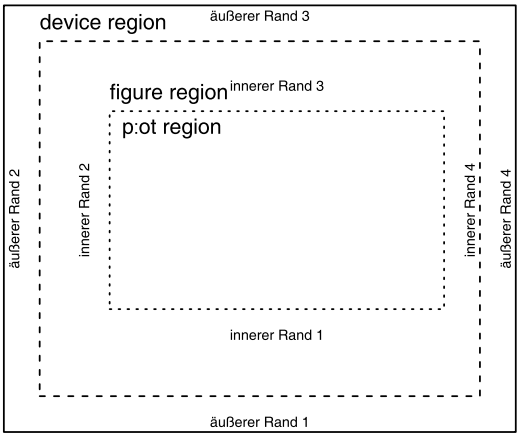

Abb. 8.6. Regionen und Ränder

ren Rändern bestehen kann (s.u.). Eine „*figure region*“ enthält jeweils eine Teil-Grafik. Als Voreinstellung enthält die „*device region*“ genau die „*figure region*“. Innerhalb einer „*figure region*“ gibt es neben den inneren Rändern, in denen Beschriftungen (Achsen, Titel) zu finden sind, die „*plot region*“. In der „*plot region*“ sind schließlich die zu zeichnenden Daten zu finden.

Wenn Parameter bezüglich der Ränder gesetzt werden müssen, oder Achsen an einem der Ränder hinzugefügt werden sollen, so ist der entsprechende Rand meist mit seiner Nummer anzugeben. Gezählt wird im Uhrzeigersinn vom unteren (1) Rand angefangen über den linken (2) und den oberen (3) bis zum rechten (4) Rand (s. Abb. 8.6).

Abbildung 8.6 wurde mit der Standard-Einstellung

```
> par(mar = c(5, 4, 4, 2) + 0.1)
```

erzeugt. Der untere innere Rand ist somit etwas mehr als 5 Textzeilen hoch und der obere Rand hat eine Höhe von etwas mehr als 4 Textzeilen, was sehr gut am Abstand der Beschriftungen zur äußeren Begrenzung nachvollzogen werden kann. Das Hinzufügen von 0.1 ist sinnvoll, um die Grafik nicht direkt am Beschriftungstext abzuschneiden. Zusätzlich zu den üblichen inneren Rändern wurden außerdem mit

```
> par(oma = c(2, 2, 2, 2) + 0.1)
```

äußere Ränder der Breite 2.1 hinzugefügt.

Es macht in vielen Situationen Sinn, mehrere Grafiken neben- oder untereinander anzuordnen. So sind z.B. in Abb. 8.1 auf S. 154 zwei Grafiken nebeneinander zu sehen. Bei der Erzeugung jener Grafiken wurde vor allen weiteren Aufrufen, die die Grafiken erzeugt haben, tatsächlich der Aufruf

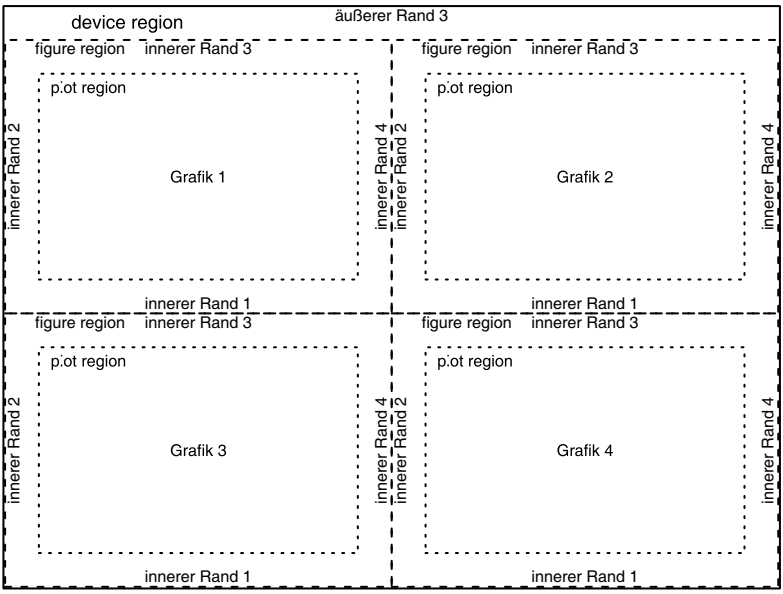

Abb. 8.7. Mehrere Grafiken in einem Device mit Regionen und Rändern

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

vorangestellt, der das aktuelle Device in eine Zeile und zwei Spalten einteilt. In dieses Gitter werden dann nacheinander die einzelnen Grafiken eingezeichnet. Analog erfolgte vor dem Erzeugen der Grafiken in Abb. 7.2 auf S. 141 der Aufruf `par(mfrow = c(2, 2))`, der das Device in zwei Zeilen und zwei Spalten (also für alle vier Grafiken) eingeteilt hat.

Sind mehrere Grafiken in einem Device angeordnet, soll häufig noch ein äußerer Rand, etwa für eine globale Überschrift, erzeugt werden. Das kann durch Aufruf von `par(oma = ...)` geschehen. Wenn ein Rand der Zeilenhöhe 2 über der Grafik eingefügt werden soll, ist `par(oma = c(0, 0, 2, 0))` der zugehörige Aufruf.

Abbildung 8.7 enthält eine Illustration, in der abgesehen von den Regionen und Rändern auch die Reihenfolge der Erzeugung der Grafik dargestellt wird. Für die Ränder und die Aufteilung des Devices wurde für diese Abbildung die Einstellung

```
> par(mfrow = c(2, 2), oma = c(0, 0, 2, 0) + 0.1,
+     mar = c(2, 2, 2, 2) + 0.1)
```

gewählt.

Eine Methode, mehrere Grafiken *unterschiedlicher Größe* in einem Device anzuordnen, wird durch die Funktion `layout()` bereitgestellt. Details und Beispiele findet man auf der zugehörigen Hilfeseite und bei Murrell (1999).

Ein alternativer Ansatz (im Gegensatz zum konventionellen Grafiksystem) wird durch das Paket **grid** (Murrell, 2002) implementiert. Dieses Paket ist dadurch bekannt, dass es grundlegende (*Low-level*) Grafikfunktionen für das Paket **lattice** (s. Abschn. 8.2) bereitstellt. Im Zusammenspiel mit dem Paket **gridBase** kann die bereitgestellte Funktionalität auch mit konventionellen Grafikfunktionen kombiniert werden. Beispiele, wie dadurch Grafiken gedreht, kompliziert beschriftet und aus mehreren gleichartigen aber komplexen Elementen zusammengesetzt werden können, beschreibt Murrell (2003). Im Wesentlichen werden dabei s.g. *Viewports* eingerichtet, die innerhalb des Devices oder innerhalb anderer *Viewports* angelegt werden können, und in die dann gezeichnet wird.

8.1.4 *Low-level* Grafik

Mit *Low-level* Grafikfunktionen können Grafiken initialisiert werden (z.B. Bereitstellung des Koordinatensystems). Außerdem können Elemente, etwa zusätzliche Punkte, Linien für Konfidenzintervalle oder Beschriftungen, zu einer Grafik hinzugefügt werden. *Low-level* Grafikfunktionen helfen auch bei der Berechnung von geeigneten Achsenbeschriftungen oder bei dem Hinzufügen einer Legende. Sie dienen damit als Grundlage und zur Erweiterung von *High-level* Grafikfunktionen, die meist für Standard-Grafiken und zur schnellen interaktiven Datenanalyse ausreichen.

Eine Auswahl solcher *Low-level* Funktionen ist in Tabelle 8.4 angegeben. Die meisten dieser Funktionen lassen sich nach kurzer Konsultation der entsprechenden Hilfeseite leicht anwenden. Konkrete Beispiele zu einigen Funktionen sind in den beiden folgenden Abschnitten zu finden.

Die Funktion `abline()` zeichnet horizontale (Argument `h`), vertikale (Argument `v`) sowie durch Achsenabschnitt und Steigung (Argumente `a`, `b`) angegebene Linien ein. Zusätzlich akzeptiert sie als Argument u.a. Objekte der Klasse `lm`, aus denen die Parameter für das Einzeichnen einer Regressionsgeraden extrahiert werden.

Während `lines()` Linien zeichnet, die die spezifizierten Koordinatenpunkte miteinander verbinden, zeichnet `segments()` mehrere voneinander unabhängige Linien vektorwertig ein.

Wer mit der von R festgelegten Einteilung der Achsen oder deren Beschriftung nicht zufrieden ist, wird in der *High-level* Grafikfunktion für die x-Achse das Argument `xaxt = "n"` oder für die y-Achse das Argument `yaxt = "n"` setzen, so dass die jeweilige Achse zunächst nicht gezeichnet wird. Alternativ kann mit `axes = FALSE` das Zeichnen beider Achsen und des die Grafik

umgebenden Kastens unterbunden werden. Der Kasten kann mit `box()` wieder hinzugefügt werden. Mit `axis()` kann eine einzelne Achse den eigenen Wünschen entsprechend hinzugefügt werden. So zeichnet

```
> axis(1, at = c(3, 4, 5), labels = c("drei", "vier", "fünf"))
```

an der x-Achse (unten: Rand 1) eine Achse, die Einteilungen an den Stellen 3, 4 und 5 des Koordinatensystems enthält. Diese Einteilungen werden mit den Zeichenfolgen „drei“, „vier“ und „fünf“ beschriftet.

Ein Beispiel – Fortsetzung

An dieser Stelle wird das Beispiel aus Abschn. 8.1.3 (S. 159, s. auch Abb. 8.5) fortgesetzt. Zu dem Histogramm und der darüber gelegten Kurve der Dichtefunktion der theoretischen Verteilung wird eine Legende an den Koordinatenpunkten $[-4.5, 0.55]$ (linker oberer Rand der Legende) hinzugefügt (s. Abb. 8.8 auf S. 167):

```
> legend(-4.5, 0.55, legend = c("emp. Dichte", "theor. Dichte"),
+        col = c("grey", "black"), lwd = 5)
```

Die Legende enthält als Beschreibungen die Zeichenketten „emp. Dichte“ und „theor. Dichte“. Daneben sind Linien zu sehen, weil die Spezifizierung des Arguments `lwd = 5` (für Liniendicke 5) impliziert, dass Linien erwünscht sind (und nicht etwa Punkte oder Kästen). Die Unterscheidung erfolgt anhand der Farbe, so dass die Linie neben der Zeichenkette „emp. Dichte“ in grau (`"grey"`), die neben „theor. Dichte“ in schwarz erscheint (`"black"`).

Tabelle 8.4. *Low-level* Grafikfunktionen (Auswahl)

Funktion	Beschreibung
`abline()`	„intelligente“ Linie
`arrows()`	Pfeile
`axis()`	Achsen
`grid()`	Gitternetz
`legend()`	Legende
`lines()`	Linien (schrittweise)
`mtext()`	Text in den Rändern
`plot.new()`	Grafik initialisieren
`plot.window()`	Koordinatensystem initialisieren
`points()`	Punkte
`polygon()`	(ausgefüllte) Polygone
`pretty()`	berechnet „hübsche“ Einteilung der Achsen
`segments()`	Linien (vektorwertig)
`text()`	Text
`title()`	Beschriftung

8.1.5 Mathematische Beschriftung

In mehreren der vorhergehenden Abschnitte in diesem Kapitel wurden bereits Beschriftungen erwähnt. Als Besonderheit für Beschriftungen beherrscht R mathematische Notation (Murrell und Ihaka, 2000), d.h. es können komplexe Formeln mit mathematischen Symbolen und griechischen Buchstaben schon bei der Erzeugung der Grafik verwendet werden. Für Details zu der mathematischen Notation empfiehlt sich ein Blick auf die Hilfeseite `?plotmath`. Eine Reihe von Kniffen wird auch von Ligges (2002) vorgestellt.

In sehr vielen Grafikfunktionen kann mathematische Notation dort verwendet werden, wo es um Beschriftung geht, zum Beispiel in den Argumenten `main`, `sub`, `xlab` und `ylab` der meisten Grafikfunktionen (wie etwa `plot()`), oder auch in der Funktion `text()`.

Mathematische Notation wird nicht in Form von Zeichenfolgen, sondern in Form von nicht ausgewerteten R Ausdrücken angegeben. Das kann in der einfachsten Form mit Hilfe von `expression()` geschehen. Weitere Details zu entsprechenden Sprachobjekten und zugehörige Literaturstellen werden in Abschn. 4.6 gegeben.

Eine Formel wird nahezu in R Syntax spezifiziert, wobei einige Schlüsselwörter an LaTeX angelehnt sind, z.B. `frac(Zähler, Nenner)` zur Spezifikation von Brüchen. Griechische Buchstaben werden in Lateinischer Schrift ausgeschrieben, wobei die Groß- und Kleinschreibung des Anfangsbuchstabens relevant ist, also `sigma` für σ und `Sigma` für Σ.

Die Formel $y_i = \beta_0 + \beta_1 x_i + e$ kann beispielsweise als Unterschrift wie folgt in eine Grafik eingefügt werden:

```
> plot(1:10, sub = expression(y[i] == beta[0] + beta[1] * x[i] + e))
```

Das Gleichheitszeichen wird dabei wie ein Vergleich (`==`) erzeugt und Indizes werden entsprechend der R Syntax mit eckigen Klammern spezifiziert.

Variablen in Formeln ersetzen

Manchmal sollen einige Variablen in Formeln durch Werte ersetzt werden, die erst in der Funktion berechnet werden, die auch die Grafik erzeugt. Da ein nicht ausgewerteter Ausdruck spezifiziert werden muss, jedoch eine Auswertung des Objekts, das die Variable ersetzen soll, erst noch erfolgen muss, scheint die Ersetzung zunächst schwierig zu sein. An dieser Stelle hilft die Funktion `substitute()` weiter, die Variablen mit Werten ersetzen kann, die in einer Umgebung oder einer Liste als weiteres Argument übergeben werden.

Als Beispiel sei in vorhergehenden Berechnungen die Variable `wert` erzeugt worden, die hier der Einfachheit halber auf 0.5 gesetzt wird:

```
> wert <- 0.5
> substitute(sigma == s, list(s = wert))
 sigma == 0.5
```

In dem Aufruf von `substitute()` wird in dem als erstes Argument angegebenen Ausdruck `sigma == s` das `s` durch den Wert `wert` ersetzt. Es existiert nämlich ein benanntes Argument gleichen Namens (`s`) mit Wert `wert` in der Liste, die als zweites Argument von `substitute()` spezifiziert wurde. Das von `substitute()` zurückgegebene Sprachobjekt kann von den Grafikfunktionen für die mathematische Notation entsprechend interpretiert werden.

Komplexere Ersetzungen, etwa zum Übergeben von Ausdrücken (in direkter Form oder in Form von Zeichenketten) an Funktionen, werden von Ligges (2002) beschrieben. In jenem Artikel wird auch gezeigt, wie mehrere Ausdrücke spezifiziert werden können, die verschiedene Formeln innerhalb einer Legende erzeugen.

Ein Beispiel – Fortsetzung

Das in Abschn. 8.1.3 (S. 159, s. auch Abb. 8.5) eingeführte und in Abschn. 8.1.4 (S. 164) fortgesetzte Beispiel wird hier erweitert.

In der linken Hälfte der Grafik soll nun zusätzlich die Formel für die bereits eingezeichnete theoretische Dichtefunktion

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

der Normalverteilung erscheinen. Weiter sollen in der rechten Hälfte der Grafik die Werte der Parameter ($\mu = 0, \sigma = 1$) angegeben werden. Beides kann mit den folgenden beiden Aufrufen von `text()` geschehen:

```
> text(-5, 0.3, adj = 0, cex = 1.3,
+     expression(f(x) == frac(1, sigma * sqrt(2*pi)) ~~
+         e^{frac(-(x - mu)^2, 2 * sigma^2)}))
> text(5, 0.3, adj = 1, cex = 1.3,
+     expression(paste("mit ", mu == 0, ", ", sigma == 1)))
```

Die doppelte Tilde (`~~`) erzeugt dabei einen kleinen Zwischenraum zwischen Bruch und Exponentialfunktion. Das Ergebnis dieser mathematischen Beschriftung ist in Abb. 8.8 zu sehen.

8.1.6 Eigene Grafikfunktionen definieren

Wer eine selbst entwickelte oder eine noch nicht in R enthaltene Visualisierungsform implementieren möchte, wird die neue *High-level* Grafikfunktion aus *Low-level* Funktionen zusammensetzen wollen. Alternativ kann die Basis

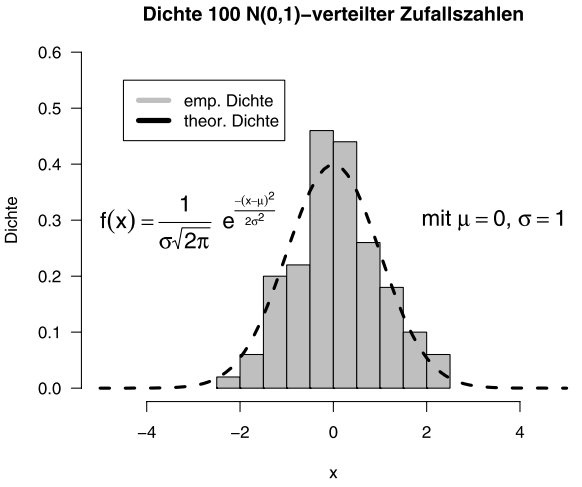

Abb. 8.8. Beispielgrafik – Histogramm und Dichtefunktion mit Legende und mathematischer Beschriftung

von einer anderen *High-level* Funktion gebildet werden, die dann mit Hilfe von *Low-level* Funktionen erweitert wird.

High-level Grafikfunktionen basieren meist auf einer Abfolge von *Low-level* Funktionen. Nach der Initialisierung des Grafik-Device wird das Koordinatensystem eingerichtet, Daten werden eingezeichnet, Achsen hinzugefügt und letztendlich wird die Grafik beschriftet.

Wie aus Abschn. 8.1.3 bereits bekannt ist, gibt es eine sehr große Anzahl an Argumenten (z.B. die aus `par()` und `plot.default()` bekannten), mit denen Grafiken konfiguriert werden können. Wer eine eigene Grafikfunktion implementiert, möchte sicher nicht alle diese Argumente an alle innerhalb der Funktion verwendeten *Low-level* Funktionen einzeln weiterleiten. Gerade hier bietet sich die Verwendung des Dreipunkte-Arguments „`...`“ an (s. Abschn. 4.1.2). So können die meisten Argumente einfach an viele (oder gar alle) *Low-level* Funktionen weitergeleitet werden. Nur diejenigen Argumente, die in der Funktion beeinflusst werden sollen, müssen somit als formale Argumente implementiert sein. Gerade bei Grafikfunktionen besteht wegen der vielen Argumente die Gefahr, dass nicht allgemein programmiert wird, sondern zunächst sinnvoll erscheinende Einstellungen fest implementiert werden.

Als Beispiele mögen die vielen Methoden der generischen Funktion `plot()` dienen, die man nicht nur in den Basis-Paketen von R findet. Als weiteres Beispiel sei die Funktion `scatterplot3d()` im Paket **scatterplot3d** (Ligges und Mächler, 2003) erwähnt, die trotz ihrer Komplexität ausschließlich auf *Low-level* Grafikfunktionen basiert.

Im Folgenden wird die `plot`-Methode zum Zeichnen von Objekten der Klasse `histogram`, wie sie etwa mit `hist()` erzeugt werden, betrachtet. Der im Wesentlichen vollständige Code der im Namespace des Pakets **graphics** „versteckten“ Funktion kann durch Eingabe von `graphics:::plot.histogram` angesehen werden. Einige interessante Zeilen wurden hier extrahiert:

```
plot.histogram <- function (x, freq = equidist, density = NULL,
    angle = 45, col = NULL, border = par("fg"), lty = NULL,
    main = paste("Histogram of", x$xname), sub = NULL,
    xlab = x$xname, ylab, xlim = range(x$breaks), ylim = NULL,
    axes = TRUE, labels = FALSE, add = FALSE, ...)
# [... hier wurde Code ausgelassen  ...] #
    if (!add) {
      if (is.null(ylim)) ylim <- range(y, 0)
# [... hier wurde Code ausgelassen  ...] #
      plot.new()
      plot.window(xlim, ylim, "")
      title(main = main, sub = sub, xlab = xlab, ylab = ylab, ...)
      if (axes) {
        axis(1, ...)
        axis(2, ...)
      }
    }
    rect(x$breaks[-nB], 0, x$breaks[-1], y, col = col,
      border = border, angle = angle, density = density, lty = lty)
# [... hier wurde Code ausgelassen  ...] #
}
```

Die Funktion beginnt mit der Definition vieler formaler Argumente. In den ersten extrahierten Zeilen wird überprüft, ob das Argument `ylim` als `NULL` eingegeben wurde, und gegebenenfalls sinnvoll ersetzt. Danach ist die Abfolge der Grafikfunktionen sehr einfach. Nach der Initialisierung mit `plot.new()` wird das Koordinatensystem von `plot.window()` vorbereitet (`plot.new()` und `plot.window()` sind keine Methoden zu `plot()`). Dann erfolgt die Beschriftung (`title()`), und Achsen werden mit `axis()` eingezeichnet. Zuletzt werden von der Funktion `rect()` die Rechtecke gezeichnet, die die eigentlichen Daten des Histogramms darstellen. Während alle Argumente von `rect()` formal im Kopf der Funktion `plot.histogram()` definiert sind, wird zur Weiterleitung an `axis()` und `title()` auch das Dreipunkte-Argument verwendet.

8.2 Trellis Grafiken mit lattice

Trellis Grafiken wurden von Cleveland (1993) eingeführt und von Becker et al. (1996) weiter diskutiert. Sie werden in R durch das Paket **lattice** (Sarkar,

2002) implementiert, welches auf **grid** (Murrell, 2001, 2002) basiert. Die drei Begriffe „lattice“, „grid“ und „trellis“ lassen sich alle mit „Gitter“ übersetzen. Entsprechend stellen Trellis Grafiken (im Folgenden auch als Lattice Grafiken bezeichnet) oft viele Grafiken desselben Typs innerhalb eines Gitters dar, wobei in jeder Grafik nur eine Teilmenge der Daten zu sehen ist, z.B. wird je Faktoreinstellung einer kategoriellen Variablen im Datensatz eine Grafik erzeugt (s. z.B. Abb. 8.10 auf S. 175). Alternativ dazu können viele Variablen gleichzeitig visualisiert werden, z.B. in Form einer Streudiagramm-Matrix. Das Paket **grid** wird automatisch mitgeladen, wenn mit

```
> library("lattice")
```

Trellis Grafikfunktionen verfügbar gemacht werden.

Tabelle 8.5 gibt eine Übersicht über wichtige *High-level* und *Low-level* Grafikfunktionen des Pakets **lattice** sowie über wichtige Funktionen zur Kontrolle des Trellis Device.

Das Paket **grid** wird in Abschn. 8.2.2 kurz vorgestellt, nachdem in Abschn. 8.2.1 auf die Unterschiede zwischen konventionellen Grafiken und Trellis Grafiken eingegangen wurde. Zwei dieser Unterschiede, die Ausgabe von Trellis Grafiken auf ein Device (s. Abschn. 8.2.3) und das Formelinterface (s. Abschn. 8.2.4), werden näher erläutert. In Abschnitt 8.2.5 wird die Konfiguration und Erweiterbarkeit von Trellis Grafiken nehandelt.

8.2.1 Unterschiede zu konventioneller Grafik

Ein wesentlicher Unterschied zu konventionellen Grafiken ist der, dass das im Paket **grid** enthaltene Grafiksystem von **lattice** verwendet wird, nicht aber das konventionelle Grafiksystem. Damit werden auch andere *Low-level* Grafikfunktionen zum Zusammensetzen von *High-level* Grafikfunktionen genutzt. Insbesondere wird eine hohe Modularität dadurch erreicht, dass häufig genutzte Elemente als *panel*-Funktionen zur Verfügung stehen, die später miteinander kombiniert werden können (s. Abschn. 8.2.5).

Obwohl prinzipiell alle in Abschn. 8.1.1 beschriebenen Devices genutzt werden können, wird meist doch auf das in Abschn. 8.2.3 eingeführte Trellis Device zurückgegriffen. Anders als bei den konventionellen Grafiken, bei denen Element für Element nacheinander einzeln gezeichnet wird, wird ein Trellis Objekt erst generiert und bearbeitet, bevor es gezeichnet wird.

Die Benutzer-Schnittstelle zu Trellis Grafikfunktionen ist als Formelinterface so angelegt, dass Zusammenhänge der Daten als Formel beschrieben werden (s. Abschn. 8.2.4). Damit kann auch die Anordnung der nebeneinander liegenden Grafiken sehr einfach ausgedrückt werden.

Tabelle 8.5. Auswahl von Lattice Grafikfunktionen für *High-level*, *Low-level* und Kontrolle des Trellis Device

Funktion	Beschreibung
`barchart()`	Balkendiagramme
`bwplot()`	Boxplots
`cloud()`	3D-Punktewolken
`contourplot()`	Höhenlinien-Plots
`densityplot()`	Dichten
`dotplot()`	Punkteplots
`histogram()`	Histogramme
`levelplot()`	Levelplots
`piechart()`	Kuchendiagramme
`qq()`	QQ–Plots
`splom()`	Streudiagramm-Matrix
`wireframe()`	3D-Flächen
`xyplot()`	sehr allgemeine Grafikfunktion, ähnlich der Funktion `plot()`, z.B. zum Zeichnen eines Streudiagramms
`larrows()`	Pfeile
`llines()`	Linien
`lpoints()`	Punkte
`lsegments()`	mehrere Linien gleichzeitig
`ltext()`	Text
`panel.....()`	bereits vordefinierte Funktionen zum Hinzufügen komplexerer Elemente
`print()`	Trellis Objekt zeichnen
`trellis.device()`	Trellis Device
`trellis.par.get()`	Parameter des Device abfragen
`trellis.par.set()`	Parameter des Device einstellen

8.2.2 Das Paket grid – mehr als nur Grundlage für lattice

Es ist ein Grundproblem des „konventionellen“ Grafiksystems, dass es extrem mühsam ist, allgemein verschiedenartige Grafiken innerhalb einer Grafik zu realisieren. Die Funktion `layout()` oder das einfache `par(mfrow = ...)` können nicht geschachtelt werden, so dass es beispielsweise unmöglich ist, mehrere `coplot()`, `filled.contour()` oder `pairs()` Grafiken zu einer neuen Grafik zusammenzustellen. Die scheinbar sehr flexible Verwendung von `par()` stößt dort an Grenzen oder wird hässlich. Nicht umsonst enthält der Source-Code in der Datei ‘`par.c`’ folgenden Kommentar:

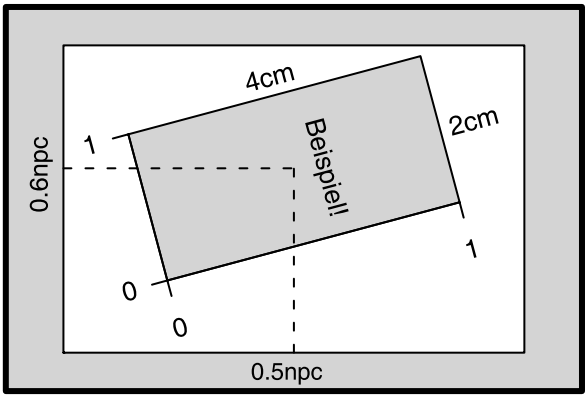

Abb. 8.9. Beispiel zu Funktionen aus Paket **grid**

```
*  "The horror, the horror ..."
*  Marlon Brando in Apocalypse Now.
```

Das Paket **grid** selber arbeitet mit *Grob*s (*Gr*aphical *Ob*jects), *Viewports* und *Units* (Einheiten) und ermöglicht damit ein klares und extrem flexibles System von Grafiken in diversen Koordinatensystemen, bearbeitbare graphische Elemente und beliebige Grafiken innerhalb von einer Grafik.

Abbildung 8.9 zeigt das Ergebnis des folgenden Beispiels:

```
> library(grid)                      # Laden von "grid"
> vp <- viewport(x = 0.5, y = 0.6, angle = 15,
+                w = unit(4, "cm"), h = unit(2, "cm"))
> grid.show.viewport(vp)             # Viewport vp visualisieren
> pushViewport(vp)                   # Viewport auf den Stack legen
> grid.text("Beispiel!", 0.6, 0.4, rot = -90)
> popViewport()                      # einen Viewport zurück
> grid.rect(gp = gpar(lwd = 3))      # äußerer Rahmen
```

Nach dem Laden von **grid** wird hier mit `viewport()` ein Viewport erzeugt, dessen Mittelpunkt in der Gesamtgrafik an der Stelle `x = 0.5, y = 0.6` liegt. Diese Angabe erfolgt in einer zum Koordinatensystem der Gesamtgrafik relativen Einheit. Der Viewport `vp` hat die Breite 4cm und die Höhe 2cm, also absolute Größen als Einheit, statt derer man auch wieder relative Größen (z.B. relativ zur Größe der Gesamtgrafik) hätte angeben können. Hier wird schon deutlich, wie flexibel mit Koordinatensystemen umgegangen werden kann. Zudem ist der Viewport noch um `angle = 15` Grad gedreht.

Nach der Visualisierung des Viewports `vp` wird er mit `pushViewport()` auf den *Stack* gelegt. Den Stack stelle man sich als einen Stapel von Viewports vor, auf den man jeweils einen Viewport legen oder wieder herunter-

nehmen kann. Der zuoberst liegende Viewport kann dabei bearbeitet werden. Im Beispiel wird in diesen Viewport mit `grid.text()` eine Beschriftung eingefügt, die relativ zu dessen Koordinatensystem (und der Drehung!) erfolgt. Mit `popViewport()` gelangt der Fokus zurück zur Gesamtgrafik, um die mit `grid.rect()` ein Rahmen gezeichnet wird.

Es gibt immer mehr Pakete, die die große Flexibilität von **grid** ausnutzen.

8.2.3 Ausgabe von Trellis Grafiken – `trellis.device()`

Das Trellis Device wird analog zu den in Abschn. 8.1.1 beschriebenen Geräten mit Hilfe der Funktion `trellis.device()` gestartet. Dabei wird als erstes Argument der Name des gewünschten zu Grunde liegenden Devices erwartet. Das kann eines der in Tabelle 8.1 (Abschn. 8.1.1, S. 150) aufgelisteten Geräte sein, z.B. also `"x11"` für Bildschirmgrafik oder `"postscript"` und `"pdf"` zur Erzeugung von PostScript oder PDF Dateien. Auch hier muss ein Device mit `dev.off()` wieder geschlossen werden. Der Dateiname einer u.U. auszugebenden Datei kann mit `file = "Dateiname.Endung"` spezifiziert werden. Tatsächlich handelt es sich bei `trellis.device()` um eine Art Meta-Device, das die Trellis Grafik initialisiert, indem es ein passendes anderes Device startet und ein Farbschema setzt. Ein Beispiel ist dazu im folgenden Abschn. 8.2.4 zu finden. Das Device für Bildschirmgrafik wird bei interaktivem Aufruf einer Trellis Grafikfunktion automatisch gestartet. Alternativ zum Trellis Device können in den meisten Fällen die herkömmlichen Devices (s. Abschn. 8.1.1) auch direkt gestartet werden.

Die Voreinstellung des Farbschemas ist je nach gestartetem Device verschieden. Bei der Ausgabe als PostScript wird als Voreinstellung eine schwarz auf weiß gezeichnete Grafik erzeugt. Solche schwarz-weiß Grafiken können durch das Setzen des Argumentes `color = FALSE` in `trellis.device()` erzwungen werden.

Mit Hilfe von `trellis.par.get()` können Einstellungen des Devices abgefragt und mit `trellis.par.set()` gesetzt werden. Häufig soll das ganze Farbschema geändert werden, was für das gerade aktive Device beispielsweise mit

```
> trellis.par.set(col.whitebg())
```

geschehen kann, wobei die Funktion `col.whitebg()` ein Farbschema mit transparentem Hintergrund liefert.

Ein Trellis Objekt wird, anders als bei konventionellen Grafiken, zunächst vollständig generiert und bearbeitet, bevor es endgültig gezeichnet wird. Das Zeichnen geschieht dabei mit Hilfe der Funktion `print()`. Diese generische Funktion `print()` ruft die Methode `print.trellis()` für Objekte der Klasse

`trellis` auf. Diesem Umstand ist die extrem häufig gestellte Frage zu verdanken, warum unter gewissen Umständen die Erstellung einer Trellis Grafik (*anscheinend!*) fehlschlägt. Während der Aufruf einer Trellis Grafik, im einfachsten Fall z.B. `xyplot(1 ~ 1)` (für Details zur Formelnotation s. Abschn. 8.2.4), bei der interaktiven Arbeit eine Grafik zeichnet, führt er bei nicht interaktivem Aufruf bzw. innerhalb einer Funktion nicht zum Zeichnen einer Grafik:

```
> mein.trellis <- function(){
+     xyplot(1 ~ 1)   # hier wurde etwas vergessen!
+     return(1)
+ }
> mein.trellis()      # hier erscheint *keine* Grafik!
 [1] 1
```

Grund ist, dass interaktive Aufrufe stets mit `print()` ausgegeben werden, solange es sich bei dem Aufruf nicht um eine Zuweisung handelt. Das ist innerhalb von Funktionen oder im nicht interaktiven Modus nicht der Fall. Mit `xyplot(1 ~ 1)` wird in der oben definierten Funktion `mein.trellis()` zwar ein Trellis Objekt erzeugt, es wird aber weder gezeichnet noch von der Funktion zurückgegeben. Diese müsste wie folgt korrigiert werden:

```
> mein.trellis <- function(){
+     print(xyplot(1 ~ 1))
+     return(1)
+ }
> mein.trellis()      # jetzt erscheint eine Grafik!
 [1] 1
```

Das folgende Beispiel verdeutlicht noch einmal, dass Trellis Funktionen Objekte der Klasse `trellis` erzeugen, die bearbeitet und erst später gezeichnet werden könnten:

```
> trellisObjekt <- xyplot(1 ~ 1)
> str(trellisObjekt)        # Struktur des Trellis Objektes
 List of 38
  $ formula          :Class 'formula' length 3 1 ~ 1
   .. ..- attr(*, ".Environment")=length 28 <environment>
  $ as.table         : logi FALSE
  $ aspect.fill      : logi TRUE
  . . . . . . . . . . . . . # Hier wurde die Ausgabe gekürzt
  $ prepanel.default :function (x, y, type, subscripts,
                                            groups = NULL, ...)
  $ prepanel         : NULL
  - attr(*, "class")= chr "trellis"
> # Hier könnte das Trellis Objekt manipuliert werden.
> print(trellisObjekt)      # Die Grafik wird jetzt gezeichnet
```

8.2.4 Formelinterface

Trellis Grafikfunktionen erwarten eine Formel, die die Abhängigkeitsstrukturen der Daten beschreibt. Die Formelnotation zur Spezifikation statistischer Modelle wurde bereits in Abschn. 7.4 vorgestellt. Die Formelnotation von **lattice** ist sehr ähnlich und hat die Form `y ~ x | z`.

Links der Tilde (`~`) steht die Zielvariable, die im Allgemeinen auf der Ordinate (y-Achse) abgetragen wird, während die Variablen zwischen der Tilde und dem senkrechten Strich (`|`) auf der Abszisse abgetragen werden. Die linke Seite muss nicht spezifiziert werden, falls nur eine Dimension abgetragen werden soll (z.B. bei Boxplots). Rechts des senkrechten Striches (`|`) stehen bedingende kategorielle Variablen. Für jede Faktoreinstellung wird eine eigene Grafik mit der entsprechenden Teilmenge der Daten erzeugt. Wenn auf mehrere Variablen bedingt werden soll, können jene durch Multiplikationszeichen (`*`, „Wechselwirkung“) getrennt angegeben werden, so dass für jede Kombination der Faktoreinstellungen aller Variablen eine Grafik erzeugt wird.

Die Formel `y ~ x | z1 * z2` kann also wie folgt gelesen werden: „Trage für jede Kombination der Faktoreinstellungen von `z1` und `z2` das `y` gegen das `x` ab.“

Eine einfache Form einer solchen Formel zeigt das folgende Beispiel, in dem erneut (s. auch Abschn. 2.5) die *iris* Daten (Anderson, 1935) verwendet werden. Nachdem Datensatz und Paket geladen sind, wird das Trellis Device geöffnet, mit dem die Abb. 8.10 (als PostScript) erzeugt wurde:

```
> library("lattice")      # Laden des Pakets lattice
> trellis.device("postscript", file = "lattice-beispiel.eps",
+                width = 8, height = 4, horizontal = FALSE)
> xyplot(Petal.Length ~ Sepal.Length | Species, data = iris)
> dev.off()
```

Bevor das Device wieder geschlossen wird, erfolgt die Erzeugung des Trellis Objekts, das im interaktiven Modus so auch gezeichnet wird. Hier werden mit `xyplot()` Streudiagramme erzeugt. Und zwar wird für die Daten jeder Pflanzensorte (`Species`) einzeln ein Streudiagramm der `Petal.Length` gegen die `Sepal.Length` gezeichnet.

Es gibt aber auch die Möglichkeit, auf gewisse Bereiche nicht kategorieller (sogar stetiger) Variablen zu bedingen. Dazu kann zum Beispiel mit Hilfe der Funktion `equal.count()` diskretisiert werden, wobei nach der zu diskretisierenden Variable als weitere Argumente die geforderte Anzahl an Klassen und der Anteil der Überlappung zwischen einzelnen Klassen angegeben werden können.

Als weitere Variable soll die `Sepal.Width` der *iris* Daten in die Analyse aufgenommen werden, indem mit

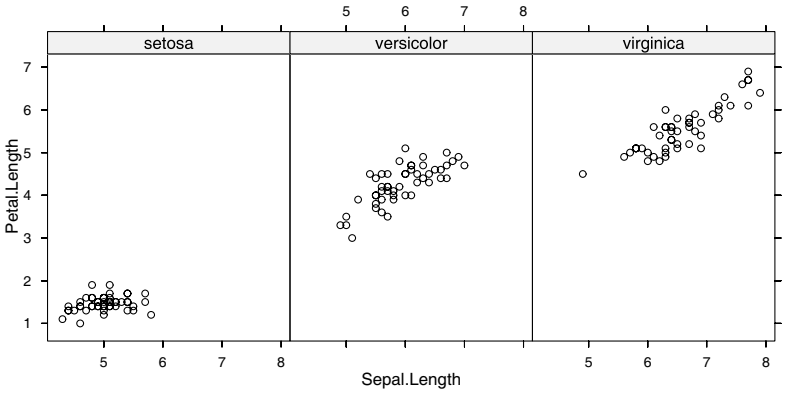

Abb. 8.10. Lattice Grafik der *iris* Daten, bedingt auf `Species`

```
> SW <- equal.count(iris$Sepal.Width, number = 2, overlap = 0.33)
```

ein Objekt `SW` erzeugt wird, das die `Sepal.Width`, unterteilt in zwei Klassen, bei einem Anteil der Überlappung von 1/3 enthält. Dieses neu erzeugte Objekt kann jetzt als weitere bedingende Variable in die Formel aufgenommen werden:

```
> xyplot(Petal.Length ~ Sepal.Length | Species * SW, data = iris)
```

Der Aufruf resultiert jetzt in sechs Streudiagrammen (es gibt sechs Kombinationen der Faktoreinstellungen, s. Abb. 8.11).

Der dunkle Balken hinter der Überschrift „SW“ zeigt an, ob die Klasse der kleinen oder der großen `Sepal.Width` in der jeweiligen Grafik zu sehen ist. Offensichtlich hat die durch das Objekt `SW` angegebene Variable in diesen zwei Klassen keinen großen Einfluss auf den Zusammenhang, denn die oberen und die unteren Grafiken in Abb. 8.11 unterscheiden sich kaum.

8.2.5 Konfiguration und Erweiterbarkeit

Neben den bereits in Abschn. 8.2.3 beschriebenen Möglichkeiten zur Konfiguration des Trellis Device, lassen sich eine große Anzahl an Parametern auch an die meisten Trellis Grafikfunktionen übergeben, wobei die Namen der meisten Argumente identisch sind mit den Namen entsprechender Argumente in konventionellen Grafikfunktionen. Die Hilfe `?xyplot` enthält dazu eine sehr detaillierte Übersicht.

Erweiterungen von den durch **lattice** implementierten Trellis Grafiken sind wegen der durch die *panel*-Funktionen erreichten Modularität sehr einfach vorzunehmen. Dabei kann man sich *panel*-Funktionen als eine sinnvolle Bündelung von *Low-level* Funktionen vorstellen, die eine bestimmte Grafik

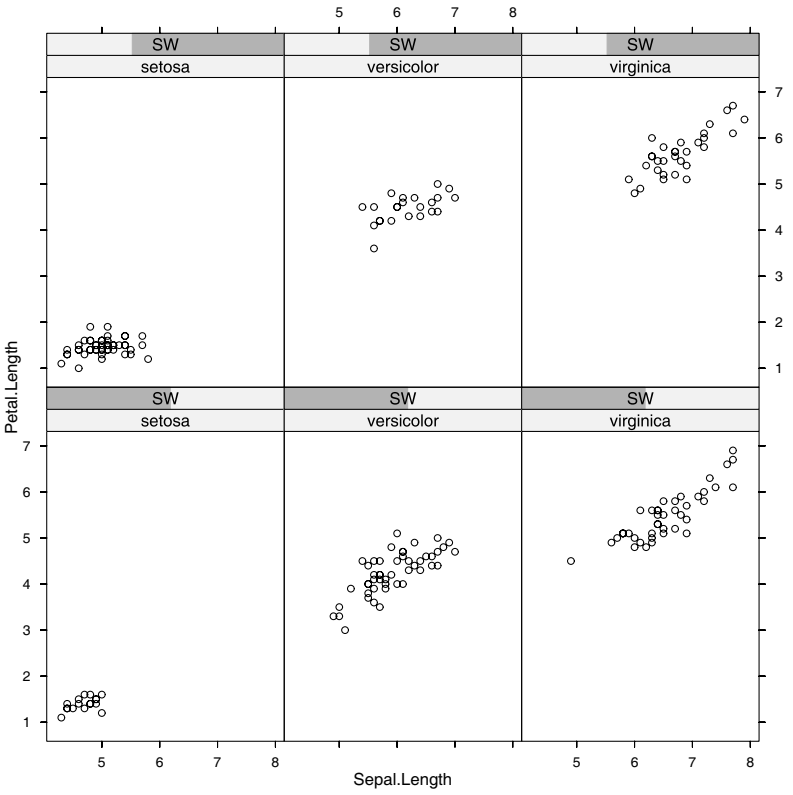

Abb. 8.11. Lattice Grafik der *iris* Daten, bedingt auf `Species` und `Sepal.Width`

erzeugen können. An Stelle einer ausführlichen Beschreibung soll diese Erweiterbarkeit hier anhand eines kurzen Beispiels demonstriert werden.

Zunächst wird eine eigene *panel*-Funktion definiert, die einfach aus zwei anderen *panel*-Funktionen zusammengesetzt ist:

```
> mein.panel <- function(x, y){
+     panel.xyplot(x, y)
+     panel.lmline(x, y, lwd = 3)
+ }
```

Die Funktion `panel.xyplot()` ist die voreingestellte *panel*-Funktion von `xyplot()`, die den inneren Teil von Streudiagrammen zeichnet. Da die Voreinstellung für die *panel*-Funktion später mit `mein.panel()` überschrieben wird, muss die Funktion `panel.xyplot()` hier mit angegeben werden. Die Funktion `panel.lmline()` schätzt intern die Parameter des Modells `lm(y ~ x)` und zeichnet eine entsprechende Regressionsgerade, wie hier gefordert, mit Lini-

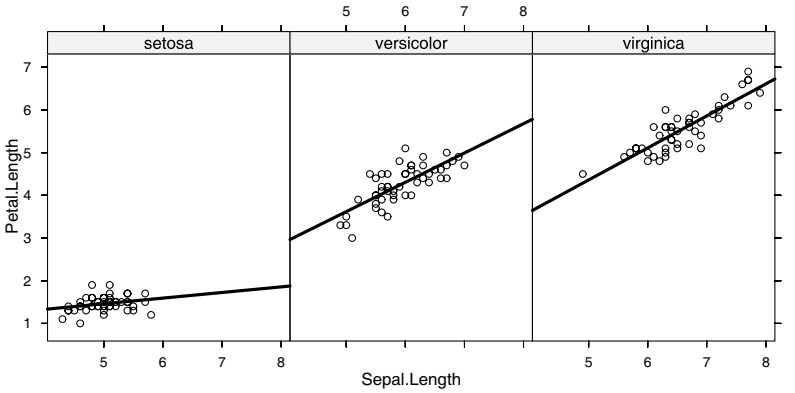

Abb. 8.12. Lattice Grafik der *iris* Daten mit eingezeichneter Regressionsgeraden, bedingt auf `Species`

enstärke 3 (Argument `lwd`). Die so definierte eigene Funktion `mein.panel()` führt in dem Aufruf

```
> xyplot(Petal.Length ~ Sepal.Length | Species, data = iris,
+        panel = mein.panel)
```

dann zu der in Abb. 8.12 dargestellten resultierenden Grafik.

Funktionalität für eine mögliche Kombination von konventionellen Grafiken und Trellis Grafiken wird durch das Paket **gridBase** zur Verfügung gestellt, welches in Abschn. 8.1.3 auf S. 163 kurz eingeführt wurde.

8.3 Dynamische und interaktive Grafik

Das Grafiksystem von R ist zwar nicht für dynamische und interaktive Grafik ausgelegt, jedoch gibt es eine Reihe von Paketen, von denen entsprechende Funktionalität bereitgestellt wird. Die hier vorgestellten Pakete und Programme arbeiten nicht mit den R Grafik Devices zusammen. Daher ist das Aussehen der Grafiken auch von dem der Standard R Grafiken verschieden.

Multidimensionale Visualisierung mit XGobi und GGobi

Das Visualisierungssystem XGobi (Swayne et al., 1998) ist ein eigenständiges Programm, das interaktive Visualisierung multidimensionaler Daten unterstützt, zum Beispiel so genanntes „brush and spin“, Verlinkung oder Rotationstechniken wie die „grand tour“. Das R Paket **XGobi** (Swayne et al., 1991) bietet ein Interface zur Kommunikation mit dem Programm XGobi.

Eine neue Implementierung der Ideen von XGobi findet man in GGobi[1] (Swayne et al., 2003). Auch zu diesem Programm gibt es ein korrespondierendes R Paket namens **rggobi** (Temple Lang und Swayne, 2001). Durch ein bidirektionales Interface ermöglicht **rggobi** es, GGobi innerhalb von R zu benutzen. Damit wird interaktiver Zugriff auf verlinkte und hoch-dimensionale Grafiken innerhalb von R möglich.

Bei *verlinkten* Grafiken können mehrere Grafikfenster geöffnet sein, die dieselben Beobachtungen in unterschiedlicher Form (z.B. als Streudiagramm und Histogramm) bzw. mit unterschiedlichen Variablen repräsentieren. Wird dann eine Auswahl von Beobachtungen in einem Fenster markiert, so werden dieselben Beobachtungen automatisch auch in allen anderen verlinkten Grafikfenstern hervorgehoben.

*3D mit **rgl***

Das auf CRAN erhältliche Paket **rgl** von Adler et al. (2003) bietet eine Schnittstelle zu OpenGL[2]. Unterstützt werden neben interaktiver Navigation der Position des Betrachters im 3-dimensionalen Raum (d.h. Drehung und Zoom) auch verschiedene Lichtquellen, *fogging* und (Teil-)Transparenz (*alpha-blending*), damit auch bei vielen Datenpunkten noch Strukturen im Raum erkannt werden können. *High-level* Grafikfunktionen, die eine schnelle interaktive Datenanalyse unterstützen, wurden mit Hilfe von Duncan Murdoch aus dem etwas älteren Paket **djmrgl**[3] (Murdoch, 2001) portiert. Besonders erwähnenswert sind die Funktionen `plot3d()` zum Zeichnen von Streudiagrammen und `persp3d()` zum Zeichnen von Flächen.

Verlinkte Grafiken mit iPlots

Das Paket **iplots**[4] von Urbanek und Theus (2003) bietet einige Grafikfunktionen zum Erzeugen verlinkter Grafiken für die interaktive Datenanalyse. Unter den unterstützten Grafiken sind Streudiagramme, Histogramme und Balkendiagramme. Das Paket basiert auf Java und benötigt daher das Paket **rJava**. Details sind unter der angegebenen URL[4] zu finden.

[1] `http://www.ggobi.org`
[2] `http://www.opengl.org`
[3] Das Paket **djmrgl** hieß früher auch **rgl**. Es wurde wegen der doppelten Namensgebung und der allgemeineren Implementierung des neueren Paketes **rgl** jedoch umbenannt. Es bietet zudem weniger Funktionalität und läuft nur unter Windows.
[4] `http://www.rosuda.org/iPlots/`

Interaktive Analyse von Bäumen – KLIMT

Für den interaktiven Umgang mit Bäumen wurde das Programm KLIMT[5] (*Kl*assification – *I*nteractive *M*ethods for *T*rees), das auch eine Schnittstelle zu R mitbringt, von Urbanek und Unwin (2002) entwickelt.

Neben der Visualisierung von Bäumen mit verschiedensten Ansichten, Rotation von Bäumen, Zoom-Funktion sowie Umordnung und Verschiebung von Knoten und Blättern, können Bäume interaktiv umgebaut und „beschnitten“ (*pruning*) werden. Auch Unterstützung zur Analyse von Wäldern (*forests*) ist vorhanden.

Zur einfachen Analyse der Knoten können zugehörige Grafiken (Streudiagramme, Histogramme, ...) erstellt werden. Die Knoten sind verlinkt, so dass bei Markierung angezeigt wird, wie viele Beobachtungen von dem markierten Knoten in welchen anderen Knoten wandern.

[5] `http://www.rosuda.org/KLIMT/`

9

Erweiterungen

In diesem Kapitel wird erläutert, wie man Quellcode einbindet (Abschn. 9.1) und die Integration von anderen Programmen in R bzw. R in andere Programme (Abschn. 9.2) erreicht. Auch Erklärungen zu dem Batch Betrieb (Abschn. 9.3) und zu Interaktionen mit dem Betriebssystem (Abschn. 9.4) werden gegeben.

9.1 Einbinden von Quellcode: C, C++, Fortran

Für das Einbinden von Bibliotheken (Libraries), die aus C, C++ oder Fortran Quellcode kompiliert wurden, sowie für das Kompilieren und Linken solcher Bibliotheken stellt R sehr mächtige Werkzeuge zur Verfügung. Große Teile von R selbst sind in C und Fortran geschrieben. Der Vorteil von kompiliertem Code ist in erster Linie Geschwindigkeit (s. Abschn. 5), aber auch die Möglichkeit, verschiedene Schnittstellen zu bauen. Durch das Kompilieren wird eine s.g. *shared library* (Bibliothek) erzeugt, die unter UNIX meist die Endung '.so' (für *shared object*) und unter Windows '.dll' (für *dynamic link library*) hat. Für das Einbinden von Java sei auf Abschnitt 9.2 verwiesen.

Die Funktionen `.C()`, `.Fortran()`, `.Call()` und `.External()` stehen zum Aufruf von Prozeduren in solchen Bibliotheken zur Verfügung. Dabei sind `.C()` und `.Fortran()` recht allgemein und können auch zum Aufruf anderer Sprachen benutzt werden, die entsprechende C kompatible Schnittstellen zur Verfügung stellen können, z.B. C++. Die Funktion `.Call()` bietet die Möglichkeit, mehrere R Objekte direkt an den C Code zu übergeben. Als Verallgemeinerung von `.Call()` kann `.External()` aufgefasst werden, das in der Benutzung sehr ähnlich ist.

Im Folgenden wird ausschließlich der Umgang mit `.Call()` anhand eines Beispiels beschrieben, weil eine vollständige Erörterung des Themas an dieser

Stelle den Rahmen sprengen würde. Für Details sei auf das Handbuch „Writing R Extensions“ (R Development Core Team, 2006e) verwiesen. Beispiele zu den verschiedenen Möglichkeiten des Einbindens gibt es außerdem in vielen der auf CRAN erhältlichen Pakete.

`.Call()` – *anhand eines Beispiels*

Wer C Code für die Benutzung mit `.Call()` schreiben möchte, sollte neben der häufig benötigten Header-Datei ‘R.h’ auch eine der Header-Dateien ‘Rinternals.h’ oder ‘Rdefines.h’ einbinden. Die Datei ‘Rinternals.h’ definiert zusätzliche Makros[1] zum Umgang mit R Objekten in C.

Es ist zu beachten, dass der s.g. *garbage collector* in C erzeugte R Objekte zur Laufzeit „wegwirft“, wenn man diese Objekte nicht mit einem `PROTECT()` schützt (R Development Core Team, 2006e). Ein solcher Schutz wird mit `UNPROTECT()` wieder aufgehoben. Wird das Objekt nicht geschützt, kann es zum Absturz von R kommen. Objekte, die an die C Prozedur übergeben werden, müssen nicht geschützt werden. Der *garbage collector* stellt einmal verwendeten und nicht mehr benötigten Speicherplatz wieder für neue Objekte zur Verfügung.

Als sehr einfaches Beispiel wird die Addition zweier reeller Vektoren a und b mit Hilfe einer von R mit `.Call()` aufgerufenen C Prozedur gezeigt. Üblicherweise verwendet man selbstverständlich `a + b` für diese Addition.

Es sei folgender C Code in einer Datei ‘add.c’ im aktuellen Verzeichnis gegeben:

```
#include <Rinternals.h>
SEXP addiere(SEXP a, SEXP b)
{
    int i, n;
    n = length(a);
    for(i = 0; i < n; i++)
        REAL(a)[i] += REAL(b)[i];
    return(a);
}
```

Hier wird zunächst die Header-Datei ‘Rinternals.h’ eingebunden. Die definierte C Prozedur `addiere` ist, genau wie ihre Argumente `a` und `b`, eine `SEXP` (*Symbolic EXPression*). Der innere Teil enthält neben den üblichen Deklarationen eine Schleife, die zu jedem x_i das korrespondierende y_i für alle $i \in 1, \ldots, n$ addiert. Zurückgegeben wird wieder `a`, das immer noch ein R Objekt ist. Weil kein neues R Objekt erzeugt wurde, wird auch kein `PROTECT()` benötigt.

[1] Makro: Befehlsfolge, analog zu einer Funktion in R.

Aus der C Datei 'add.c' kann nun mit Hilfe des Kommandos `R CMD SHLIB` in der Kommandozeile[2] des Betriebssystems wie folgt eine Bibliothek erzeugt werden:

```
$ R CMD SHLIB add.c
```

Je nach Betriebssystem und Konfiguration unterscheiden sich die nun automatisch ausgeführten Schritte. Unter Windows wird z.B. ausgegeben:

```
making add.d from add.c
gcc -It:/R/include -Wall -O2 -c add.c -o add.o
gcc -shared -s -o add.dll add.def add.o -Lt:/R/bin -lR
```

Neben einigen anderen Dateien ist nun auch die Bibliothek 'add.dll' (bzw. 'add.so' unter UNIX) erzeugt worden, die man nun in R nutzen kann.

Zunächst muss die Bibliothek mit `dyn.load()` geladen werden. Unter der Annahme, dass die Bibliothek im Verzeichnis `c:/` liegt, geschieht dieses Laden mit:

```
> dyn.load("c:/add.dll")
```

bzw. unter UNIX analog z.B. mit

```
> dyn.load("/home/user/add.so")
```

Soll in einer Funktion unabhängig vom Betriebssystem programmiert werden, so empfiehlt sich eine Konstruktion ähnlich der folgenden:

```
> dyn.load(file.path(Verzeichnis,
+     paste("add", .Platform$dynlib.ext, sep = "")))
```

In Paketen sollte zum Laden von Bibliotheken statt `dyn.load()` besser `library.dynam()` innerhalb der Funktion `.First.lib()` des jeweiligen Pakets verwendet werden, oder der entsprechende Namespace Mechanismus (s. Abschn. 10.6).

Das Entladen einer Bibliothek geschieht mit `dyn.unload()` und wird z.B. benötigt, wenn die Bibliothek erneut aus geändertem Quellcode kompiliert werden soll, ohne dass R dazu beendet werden muss. Aber auch bei gleichnamigen Prozeduren in unterschiedlichen Bibliotheken muss u.U. die Bibliothek mit der nicht benötigten Prozedur entladen werden.

Eine R Funktion, die die nun geladene Bibliothek ('add.dll') nutzt, kann dann z.B. wie folgt aussehen:

```
> add <- function(a, b){
+     if(length(a) != length(b))
+         stop("a und b müssen gleich lang sein!")
+     .Call("addiere", as.double(a), as.double(b))
+ }
```

[2] Mit der Kommandozeile ist hier die Benutzereingabe gemeint, die von der Shell (Kommandointerpreter) des Betriebssystems ausgewertet wird.

In `.Call()` wird als erstes Argument der Name der C Prozedur angegeben, danach die beiden zu addierenden Vektoren. Man beachte, dass zur Sicherheit der im C Code durch `REAL` verwendete Datentyp mit `as.double()` explizit erzwungen wird. Das Übergeben eines falschen Datentyps kann nämlich schnell zum Absturz führen. Aus demselben Grund wird auch vor dem Aufruf der C Prozedur überprüft, ob die beiden Vektoren `a` und `b` dieselbe Länge haben.

Das Ausführen der Funktion `add()` liefert dann das erwartete Ergebnis:

```
> add(4:3, 8:9)
 [1] 12 12
```

Quellcode in Paketen

Wenn man C, C++ oder Fortran Code in einem Paket benutzen will, so geschieht das Erstellen der Bibliothek aus dem Quellcode bei der Installation des Source-Pakets automatisch (s. Abschn. 10.3, 10.4). Ein Kompilieren mit `R CMD SHLIB` ist dann also nicht nötig.

Windows

Auf einem typischen Windows System fehlen zunächst einige Werkzeuge, die für das Kompilieren von Quellcode benötigt werden. Auch ist die Shell dieses Betriebssystems anders als übliche Unix Shells.

Wo und wie die benötigten Werkzeuge für das Kompilieren unter Windows gesammelt werden und wie das Betriebssystem konfiguriert werden muss, wird in dem Handbuch „R Installation and Administration“ (R Development Core Team, 2006c) beschrieben. Man sollte diese Beschreibung dringend Zeile für Zeile studieren, da das Befolgen nahezu jeder Zeile entscheidend zum Gelingen von `R CMD SHLIB` beiträgt. Ein konkreteres Beispiel zur Einrichtung der Umgebung geben Ligges und Murdoch (2005). Wer R bereits selbst unter Windows kompiliert hat, hat das System auch auf das Kompilieren von Quellcode vorbereitet.

9.2 Integration

Die Integration von C, C++ und Fortran in R ist, wie bereits in Abschnitt 9.1 beschrieben, möglich. Aber auch die Auswertung von R Ausdrücken in C ist möglich. Hierzu sei auf das Handbuch „Writing R Extensions“ (R Development Core Team, 2006e) verwiesen, in dem detailliert einige Beispiele vorgeführt werden. Weitere Beispiele sind in einigen der auf CRAN vorhandenen Pakete zu finden.

Die Integration von Java in R kann mit Hilfe des CRAN Pakets **rJava** (Urbanek, 2006) und des Pakets **SJava** aus dem Omega Projekt (s. S. 185) erfolgen.

Einige Pakete bieten Funktionalität zur einfachen Integration von anderen Sprachen und Softwareprodukten in R, bzw. von R in andere Softwareprodukte. Dabei gibt es professionell ausgeführte Schnittstellen (Interfaces) wie die oben erwähnten Java Schnittstellen, aber auch recht einfach ausgeführte, die die benötigte Funktionalität durch geschickte Aufrufe der anderen Software steuern.

Als Beispiele für sehr einfach gehaltene „Integration" seien das CRAN Paket **RWinEdt** für den Editor WinEdt (s. Abschn. B.3, hier muss z.B. Code von WinEdt an R geschickt werden) und das auch auf CRAN erhältliche Paket **R2WinBUGS** (Sturtz et al., 2005) genannt. Letzteres generiert durch R Funktionen geschickt Eingabedateien für WinBUGS[3], startet WinBUGS mit Hilfe der Funktion `system()` (s. Abschn. 9.4), lässt sinnvolle Ausgabedateien von WinBUGS erzeugen und liest diese dann wieder in R ein. Mit dem Paket **BRugs** (Thomas et al., 2006) wird derzeit ein komplizierteres aber saubereres Interface zum WinBUGS Nachfolger OpenBUGS[4] (Thomas, 2004) entwickelt.

Das Schreiben solch einfacher „Interfaces" gelingt auch ohne Kenntnis von anderen Programmiersprachen, Softwarearchitektur oder Details der Betriebssysteme.

Das Omega Projekt

Ziel des Omega[5] Projekts (Temple Lang, 2000) ist es, eine interaktive Umgebung von Sprachen und Softwareprodukten für (verteiltes) statistisches Rechnen zu schaffen. Dazu werden Schnittstellen zwischen den verschiedenen Sprachen und Softwareprodukten benötigt, die im Rahmen des Projekts entwickelt werden. Auch Internet-basierte Software soll durch das Omega Projekt unterstützt und entwickelt werden.

Das Projekt stellt bereits eine große Anzahl an Schnittstellen (z.T. bidirektional) zwischen R und anderen Sprachen und Softwareprodukten in Form von Paketen[6] zur Verfügung. Dazu gehören u.a. CORBA, DCOM, GGobi, Gnome, Gnumeric, Gtk, Java, Perl, Postgres, Python, SASXML, SOAP, XML. Für Details zu diesen Schnittstellen sei auf die in der Fußnote angegebene URL verwiesen.

[3] `http://www.mrc-bsu.cam.ac.uk/bugs/winbugs/contents.shtml`
[4] `http://mathstat.helsinki.fi/openbugs/`
[5] `http://www.Omegahat.org`
[6] `http://www.Omegahat.org/download/R/packages/`

9.3 Der Batch Betrieb

Der Batch Betrieb (auch unter dem Begriff Stapelverarbeitung bekannt) erweist sich insbesondere für zwei Einsatzgebiete als nützlich. Zum einen ist es wünschenswert, sich bei sehr lang andauernden Berechnungen, wie etwa tage- oder wochenlangen Simulationen, nach dem Aufruf der Simulation wieder aus dem System ausloggen zu können, während es weiter rechnet. Zum anderen können Prozesse, die regelmäßig (z.B. täglich) automatisch ablaufen sollen, im Hintergrund durch das Betriebssystem gestartet werden, ohne dass eine interaktive R Sitzung gestartet werden muss.

In der Kommandozeile des Betriebssystems ruft man den Batch Modus von R wie folgt auf:

```
$ R CMD BATCH Programmdatei.R Ausgabe.txt
```

Dann wird R den Code in der Datei 'Programmdatei.R' ausführen und die Ausgabe in eine Datei 'Ausgabe.txt' in demselben Verzeichnis schreiben. Wird keine Datei für die Ausgabe spezifiziert, so wird eine Datei mit dem Namen der Programmdatei und angehängtem 'out' erzeugt, also beispielsweise 'Programmdatei.Rout'.

Zum Erzeugen von Grafiken empfiehlt es sich, die entsprechenden Devices explizit zu starten, da sonst alle Grafiken in eine einzige PostScript Datei im aktuellen Verzeichnis geschrieben werden. Insbesondere bei **lattice** (Trellis) Grafiken (s. Abschn. 8.2) gilt es, daran zu denken, dass die Grafiken im nicht interaktiven Betrieb unbedingt mit explizitem Aufruf von `print()` ausgegeben werden müssen. Außerdem sollte man Daten und Ergebnisse, die wieder verwendet werden sollen, besser gleich explizit in eigene Dateien schreiben, da das Suchen in den Ausgabedateien recht mühsam werden kann.

Wer sich, z.B. im Fall langer Simulationen, während des laufenden Prozesses aus dem System ausloggen möchte, so dass dieser nicht dabei beendet wird, kann R unter Unix / Linux mit Hilfe des Betriebssystem-Kommandos `nohup` wie folgt starten:

```
$ nohup nice -n 10 R CMD BATCH Programmdatei.R Ausgabe.txt &
```

Der Befehl `nice -n 10` bewirkt dabei, dass der Prozess eine geringere Priorität (nämlich Priorität 10) bekommt, so dass im Vodergrund laufende Anwendungen (Priorität 0) schneller beendet werden können.

9.4 Aufruf des Betriebsystems

Oftmals erweist es sich als nützlich, aus R heraus Befehle an das Betriebssystem absetzen zu können, sei es um Dateien anzulegen, Inhalt von Verzeichnissen anzuzeigen oder andere Programme zu starten.

Mit Hilfe der Funktion `system()` können Kommandos an das Betriebssystem abgesetzt werden. Neben dem ersten Argument, dem Kommando selbst, gibt es eine Reihe weiterer Argumente. Mit `intern = TRUE` (Voreinstellung: `FALSE`) wird bestimmt, dass die Ausgabe des aufgerufenen Kommandos intern als R Objekt zurückgegeben werden soll. Durch Setzen von `wait = TRUE` (die Voreinstellung) wird R veranlasst, auf die Beendigung des abgesetzten Kommandos zu warten.

Windows

Unter Windows gibt es zwei zusätzliche Funktionen für die Interaktion mit dem Betriebsystem bzw. zum Starten von Programmen.

Für direktes Absetzen von Kommandos an die Shell (Kommandointerpreter) des Betriebssystems ist die Funktion `shell()` geeignet. Sie übernimmt das lästige Spezifizieren der Windows Shell und ersetzt den komplizierteren Aufruf

```
> system(paste(Sys.getenv("COMSPEC"), "/c", "kommando"))
```

eines Shell-Kommandos `kommando` (z.B. das altbekannte `dir`) durch

```
> shell("kommando")
```

Die Funktion `shell.exec()` öffnet die als Argument angegebene Datei mit Hilfe des in der Windows Registry spezifizierten zugehörigen Programms. Der Aufruf

```
> shell.exec("c:/EineWordDatei.doc")
```

würde zum Beispiel auf einem Rechner mit einem installierten Textverarbeitungssystem (z.B. Microsoft Word), mit dem die Endung '.doc' verknüpft ist, jenes Textverarbeitungssystem starten und die Datei 'EineWordDatei.doc' öffnen.

Dateien und Verzeichnisse

Für Operationen auf Dateien und Verzeichnissen ist man nicht unbedingt auf die Funktion `system()` angewiesen. Vielmehr gibt es spezielle Funktionen (s. Tabelle 9.1) in R, die solche Operationen ausführen können. Für mehr Details sei auf die jeweiligen Hilfeseiten verwiesen.

Tabelle 9.1. Funktionen für den Umgang mit Dateien und Verzeichnissen

Funktion	Beschreibung
`file.access()`	Aktuelle Berechtigungen für eine Datei anzeigen.
`file.append()`	Eine Datei an eine andere anhängen.
`file.copy()`	Dateien kopieren.
`file.create()`	Eine neue, leere Datei erzeugen.
`file.exists()`	Prüfen, ob eine Datei bzw. ein Verzeichnis existiert.
`file.info()`	Informationen über eine Datei anzeigen (z.B. Größe, Datum und Uhrzeit des Anlegens bzw. Änderns, ...).
`file.remove()`	Dateien löschen.
`file.rename()`	Eine Datei umbenennen.
`file.show()`	Den Inhalt einer Datei anzeigen.
`file.symlink()`	Eine symbolische Verknüpfung erstellen (nicht unter allen Betriebssystemen).
`basename()`	Dateinamen aus einer vollst. Pfadangabe extrahieren.
`dir.create()`	Ein Verzeichnis erstellen.
`dirname()`	Verzeichnisnamen aus einer vollst. Pfadangabe extrahieren.
`file.path()`	Einen Pfadnamen aus mehreren Teilen zusammensetzen.
`list.files()`	Inhalt eines Verzeichnisses anzeigen (auch: `dir()`).
`unlink()`	Verzeichnis löschen (inkl. Dateien, auch rekursiv).

10

Pakete

Eine große Anzahl Pakete für die verschiedensten Anwendungsgebiete ist inzwischen für R verfügbar. Pakete sind Erweiterungen für R, die zusätzliche Funktionalität (Funktionen, Datensätze usw.) bereitstellen. Zurzeit sind das über 800 Pakete, darunter die Basis-Pakete, die empfohlenen Pakete, die riesige Menge der CRAN Pakete, Pakete der Projekte Omega (Temple Lang, 2000) und BioConductor sowie eine ganze Reihe weiterer Pakete.

Pakete können mit `library()` geladen werden und mit `detach()` wieder aus dem Suchpfad der aktuellen Sitzung entfernt werden. Hilfe zu Paketen (anstelle von Funktionen) gibt es mit `library(help = "Paketname")` oder `help(package = Paketname)`, wobei `Paketname` durch den Namen des jeweiligen Pakets zu ersetzen ist. Neben allgemeinen Informationen zum Paket (Versionsnummer, Autor, Titel, Kurzbeschreibung) gibt es dort auch eine Übersicht über weitere Hilfeseiten zu Funktionen und Datensätzen aus dem Paket. Im folgenden Beispiel werden die genannten Funktionen auf das Paket **survival** angewandt:

```
> library(help = "survival")  # Übersicht, Hilfe
> library("survival")         # Laden des Pakets
> detach("package:survival")  # Entfernen aus dem Suchpfad
```

Nach einer Diskussion zum Sinn von Paketen in Abschn. 10.1 folgt in Abschn. 10.2 eine Übersicht über aktuell verfügbare wichtige Pakete. In Abschn. 10.3 wird auf die Installation und Verwaltung von Paketen unter verschiedenen Betriebssystemen eingegangen.

Die danach folgenden Abschnitte sind für die Entwickler eigener Pakete gedacht, und zwar Abschn. 10.4 zur Struktur, 10.5 zu Funktionen und Daten in Paketen, 10.6 zur Einrichtung eines Namespace sowie Abschn. 10.7 zum Erstellen von Hilfeseiten und umfangreicherer Dokumentation.

10.1 Warum Pakete?

Das Paketsystem von R trägt wesentlich zum Erfolg des gesamten R Projekts bei. Mit Hilfe von Paketen und den Werkzeugen zur Paketverwaltung kann R sehr einfach erweitert werden. Die Entwicklung von Paketen ist nicht fest an das Grundsystem gebunden und kann somit auch verteilt erfolgen. Insbesondere können Benutzer und Entwickler, die nicht an der Entwicklung des Grundsystems von R beteiligt sind, einfach eigene Pakete entwickeln.

Es wurde bereits gezeigt, wie einfach es ist, eigene Funktionen zu erstellen. Ebenso wurde das Einbinden von externem Code (z.B. C, C++ oder Fortran) in Abschn. 9.1 vorgestellt. Mit Hilfe von Paketen kann eine *standardisierte* Sammlung von Funktionen, externem Quellcode, (Beispiel-) Datensätzen und zugehöriger Dokumentation entstehen. Die Erstellung standardisierter Dokumentation wird auf exzellente Art und Weise unterstützt. Weitere Punkte, die für das Paketsystem sprechen, sind:

- Möglichkeit des dynamischen Ladens und Entladens eines Pakets,
- einfache standardisierte Installation (und Updates) von lokalen Datenträgern oder über das Internet, innerhalb von R oder über die Kommandozeile des Betriebssystems,
- einfache Administration: Globale und lokale Einstellungen; globale, lokale und eigene *Library*-Verzeichnisse gleichzeitig nutzbar,
- Validierung: R bietet Werkzeuge zur Überprüfung von Code, Dokumentation, Installierbarkeit sowie zur Überprüfung von Rechenergebnissen,
- einfache Verteilung der Software an Dritte (Standard-Mechanismus).

Ein erstelltes Paket kann einfach an andere Personen, weitere eigene Rechner oder auch an eine noch größere Benutzergemeinde weitergegeben werden. Gerade nach der Entwicklung neuer Verfahren oder dem Schreiben von noch nicht vorhandenen Funktionen macht es Sinn, diese der Allgemeinheit, z.B. auf CRAN, zur Verfügung zu stellen.

10.2 Paketübersicht

Tabelle 10.1 gibt eine Übersicht über sehr wichtige Pakete in R. Bei einer Standard-Installation werden neben den 12 wichtigen Basis-Paketen weitere 13 wegen ihrer Qualität und Wichtigkeit empfohlene Pakete (*recommended packages*) installiert. Die in Tabelle 10.1 kursiv gedruckten Pakete werden, falls eine unveränderte Standard-Installation vorliegt, beim Start von R mitgeladen. Insbesondere sind auch alle in der Paket-Sammlung (*package bundle*) **VR** (von **V**enables und **R**ipley) enthaltenen Pakete aufgeführt.

Tabelle 10.1. Paketübersicht

Paketname	Beschreibung
	Basis-Pakete
base	Notwendige Basis-Funktionalität von R
datasets	Eine Sammlung von Datensätzen, z.B. für Beispiele
graphics	Grundlegende und aufbauende wichtige Grafikfunktionen
grDevices	Grafik-Devices, die von **graphics** und **grid** benötigt werden
grid	Re-Design für komplexes Grafik-Layout; wird vom Paket **lattice** benötigt
methods	S4 Klassen und Methoden nach Chambers (1998)
splines	Anpassen von *splines*
stats	Grundlegende Statistik-Funktionalität
stats4	Auf S4 Klassen basierende Statistik-Funktionalität
tcltk	Interface zur Programmierung von GUI mit *tcl/tk*
tools	Sammlung von Werkzeugen, u.a. zur Qualitätskontrolle von Paketen und zum Erstellen von Dokumentation
utils	Andere nützliche Funktionen
	Empfohlene Pakete
boot	Bootstrap-Verfahren
cluster	Cluster-Verfahren
foreign	Routinen für Import und Export von Daten der Statistikpakete Minitab, S-PLUS, SAS, SPSS, Stata, ...
KernSmooth	Kerndichteschätzung und -glättung
lattice	Implementation von Trellis Grafik
mgcv	Generalisierte additive Modelle
nlme	(Nicht-) Lineare Modelle mit gemischten Effekten
rpart	Rekursive Partitionierung (Klassifikations- und Regressionsbäume)
survival	Funktionalität zur Überlebenszeitanalyse
VR	Sammlung verschiedener Pakete von Venables und Ripley
	Pakete in der **VR** *Sammlung*
class	Funktionen für die Klassifikation
MASS	Funktionen-Sammlung nach Venables und Ripley (2002) *M*odern *A*pplied *S*tatistics with *S*
nnet	Neuronale Netze (feed-forward) mit einer versteckten Schicht und multinomiale log-lineare Modelle
spatial	Räumliche Statistik

Weitere Pakete

Für sehr viele weitere Pakete sei nochmals auf die Seiten von CRAN[1], die Seiten der Projekte Omega[2] (Temple Lang, 2000) und BioConductor[3] (Gentleman et al., 2004, 2005) sowie die Seiten von Lindsey (2001)[4] verwiesen. Diese Projekte entwickeln sich in einer so großen Geschwindigkeit, dass Listen an dieser Stelle auf keinen Fall aktuell sein könnten.

CRAN Task Views

Zeileis (2005) versucht, mit Hilfe so genannter *CRAN Task Views* die über 800 CRAN Pakete zu strukturieren. Für verschiedene Sachgebiete können mit diesem Mechanismus Sammlungen von Paketen zusammengestellt werden, so dass interessierte Benutzer nicht allzu lange nach passenden Paketen suchen müssen. Unter der URL `http://CRAN.R-project.org/src/contrib/Views/` gibt es zu verschiedenen Sachgebieten bereits umfangreiche Zusammenstellungen mit recht ausführlichen Beschreibungen, die von freiwilligen Benutzern aus dem jeweiligen Sachgebiet zusammengetragen wurden. Beispielsweise können sich Benutzer, die sich für das Gebiet „Machine Learning & Statistical Learning“ interessieren, zunächst das Paket **ctv** (für **C**RAN **T**ask **V**iews) installieren und damit dann die Pakete des entsprechenden *Task Views* automatisch installieren:

```
> install.packages("ctv")           # Paket zur Task View Verwaltung
> library("ctv")
> available.views()                 # zeigt vorhandene Task Views
> install.views("MachineLearning") # Task View Pakete installieren
```

10.3 Verwaltung und Installation von Paketen

Ein großer Vorteil von R ist die einfache Installation von Paketen und deren Verwaltung. In diesem Abschnitt werden Möglichkeiten und Vorgehensweisen zum Installieren, Aktualisieren und Administrieren von Paketen in möglicherweise mehr als einer *Library* beschrieben. Insbesondere wird dazu bereits vorhandene Dokumentation zur Paketverwaltung zusammengefasst und erweitert (s. auch Ligges, 2003b; Ripley, 2005b).

Zur weiteren Lektüre muss zunächst die Terminologie geklärt werden. Es gilt zwischen einem Paket (*Package*) und einer *Library* zu unterscheiden: Ein Paket wird aus einer *Library* mit Hilfe von `library()` geladen. Eine *Library* ist also ein Verzeichnis, das (meist mehrere) installierte Pakete enthält.

[1] `http://CRAN.R-project.org`
[2] `http://www.Omegahat.org`
[3] `http://www.BioConductor.org`
[4] `http://popgen.unimaas.nl/~jlindsey/rcode.html`

Dokumentation zur Paketverwaltung

Da die Paketverwaltung sehr grundlegend und wichtig für R Benutzer ist, wird sie in mehreren Dokumenten beschrieben. Das Handbuch „R Installation and Administration“ (R Development Core Team, 2006c) ist die wichtigste Quelle für Benutzer, die Pakete installieren und verwalten möchten.

Auch „The R FAQ“ (Hornik, 2006) widmet der Paketverwaltung einen Abschnitt. Die ergänzende „R for Windows FAQ“ (Ripley und Murdoch, 2006) klärt eine Reihe Windows-spezifischer Punkte in diesem Zusammenhang. Für Macintosh Benutzer wird sich in Zukunft hoffentlich ein Blick in die „R for Mac OS X FAQ“ (Iacus et al., 2006) lohnen, wenn ein entsprechender Abschnitt hinzugefügt sein wird.

10.3.1 Libraries

Falls eine offiziell veröffentlichte Version von R installiert wurde, sind die Basis-Pakete und empfohlenen Pakete (s. Abschn. 10.2) bereits in der Haupt-*Library* im Verzeichnis `R_HOME/library` installiert. Dabei bezeichnet `R_HOME` das Verzeichnis zur entsprechenden Version von R, z.B. `/usr/local/lib/R` oder `c:\Programme\R\R-2.3.1`. Als Voreinstellung werden auch alle weiteren Pakete in dieses Verzeichnis (`R_HOME/library`) installiert.

Es kann nützlich sein, mehr als eine *Library* zu benutzen, je nach Bestimmung des Rechners, auf dem R installiert wurde. Angenommen, R ist so auf einem Server installiert worden, dass der normale Benutzer keine Schreibrechte im Haupt-*Library* Verzeichnis von R hat. Dann muss nicht unbedingt der Administrator um die Installation eines Pakets gebeten werden, denn der Benutzer kann das gewünschte Paket einfach in ein anderes *Library*-Verzeichnis, z.B. `/home/user/meinR/library`, installieren. Außerdem kann eine dritte *Library* nützlich sein, die Pakete enthält, die der Benutzer selbst entwickelt, z.B. `/home/user/meinR/meineLibrary`.

In den oben angegebenen Beispielen werden UNIX Verzeichnisse benutzt, die Idee ist aber für beliebige Betriebssysteme anwendbar: Angenommen, R ist auf einer Netzwerkfreigabe in einem Windows Netzwerk installiert, z.B. `s:\R` (ohne Schreibrechte für den normalen Benutzer). Weiter hat der Benutzer jedoch Schreibrechte unter `d:\user`. Dann ist `d:\user\meinR\library` eine gute Wahl für eine *Library*, die selbst installierte Pakete enthält.

Da R zunächst ausschließlich von seiner Haupt-*Library* weiß, muss die Umgebungsvariable `R_LIBS` entsprechend gesetzt werden. Das kann auf die unter dem jeweiligen Betriebssystem übliche Art und Weise zum Setzen von Umgebungsvariablen geschehen. Besser ist es meist, `R_LIBS` in einer der Dateien zum Setzen von Umgebungsvariablen zu spezifizieren, die während des Starts von R eingelesen werden. Details dazu findet man auf der Hilfeseite `?Startup`. Um R

mitzuteilen, dass eine zweite *Library* `/home/user/meinR/meineLibrary` nach Paketen durchsucht werden soll, kann beispielsweise die Zeile

```
R_LIBS=/home/user/meinR/meineLibrary
```

in der Datei '.Renviron' (s. Hilfeseite `?Startup`) hinzugefügt werden. Evtl. muss diese Datei zunächst noch erzeugt werden. Weitere *Libraries* können als durch Semikolon getrennte Liste angegeben werden.

Während des Starts von R wird der Suchpfad der *Libraries* initialisiert durch Aufruf von `.libPaths()` auf denjenigen Verzeichnissen, die in der Umgebungsvariablen `R_LIBS` definiert sind. Die Haupt-*Library* ist dabei per Definition immer enthalten. Falls `.libPaths()` ohne Argumente aufgerufen wird, gibt diese Funktion den aktuellen Suchpfad der *Libraries* zurück. Der Suchpfad der *Libraries* ist vom Suchpfad der Umgebungen zu unterscheiden.

10.3.2 Source- und Binärpakete

Pakete kann es sowohl als *sources* (Quellcode) als auch als *binaries* (Binärformat) geben.

Source-Pakete sind i.d.R. unabhängig von der verwendeten Plattform, d.h. unabhängig von Hardware (CPU, ...) und Betriebssystem, sofern der Autor eines Pakets keine plattform-spezifischen Details implementiert hat. Selbstverständlich muss R auf der jeweiligen Plattform verfügbar sein. Zur Installation eines Source-Pakets muss als Voraussetzung eine Reihe von Werkzeugen installiert sein, z.B. Perl und, abhängig vom jeweiligen Paket, auch C und Fortran Compiler. Um Pakete einer größeren Gemeinde von anderen Benutzern zur Verfügung zu stellen, müssen Entwickler ihre Pakete als Sources an CRAN[5] übermitteln.

Für Benutzer von UNIX-artigen Systemen (z.B. Linux, Solaris, ...) ist es im Allgemeinen recht einfach, Source-Pakete zu installieren, weil bei diesen Betriebssystemen meist alle benötigten Werkzeuge bereits installiert sind.

Binärpakete sind plattform-spezifisch und können auch abhängig von der verwendeten R Version sein, vor allem dann, wenn kompilierter Code gegen R gelinkt wurde. Binärpakete können aber ohne spezielle zusätzliche Werkzeuge installiert werden, weil *shared object files* (auch als DLL[6] unter Windows bekannt), Hilfeseiten (HTML, Text, ...) usw. in einem solchen Paket bereits vorkompiliert sind.

Um die Installation von Paketen für Benutzer so einfach wie möglich zu machen, werden auf CRAN Binärpakete für die aktuellsten R Versionen auf verschiedenen Plattformen zur Verfügung gestellt, insbesondere MacOS X,

[5] Übermittelte Binärpakete werden auf CRAN nicht akzeptiert.
[6] dynamic link libraries

SuSE Linux und Windows. Zum Beispiel erscheinen Binärpakete für Windows meist innerhalb von zwei Tagen nach den entsprechenden Source-Paketen auf CRAN, falls keine speziellen Abhängigkeiten oder die Notwendigkeit für manuelle Konfiguration bestehen.

Es ist Konvention, dass Source-Pakete die Dateiendung '.tar.gz' haben, während Binärpakete für Windows die Endung '.zip' und Binärpakete für Linux die Endung '.deb' oder '.rpm' haben. Beide Endungen, sowohl '.tar.gz' als auch '.zip', zeigen an, dass es sich um komprimierte Archive[7] handelt, sie wurden nur von verschiedenen Werkzeugen erstellt.

Die in jedem Paket enthaltene Datei 'DESCRIPTION' enthält in einem Binärpaket eine Zeile die mit `Built:` beginnt, z.B.:

```
Built: R 2.3.1; i386-pc-mingw32; 2006-07-30 21:03:20; windows
```

Eine solche Zeile existiert nicht in einem Source-Paket. Auf diese Weise lassen sich Source- und Binärpakete unterscheiden.

Repositories und Spiegelserver

Sowohl Source- als auch Binärpakete können in *Repositories* (standardisierte Archive von R Paketen) zum (automatischen) Herunterladen bereitgestellt werden. Nach der Basis URL des entsprechenden Servers folgt ein standardisierter Pfad zu den Paketen:

- Source-Pakete: `BasisURL/src/contrib`
- Windows Binärpakete: `BasisURL/bin/windows/contrib/MajVer`
- MacOS X Binärpakete (universal für PowerPC und i686 Prozessoren): `BasisURL/bin/macosx/universal/contrib/MajVer`

'MajVer' ist dabei durch den ersten Teil der Versionsnummer der benutzten R Version, z.B. '2.3' für R-2.3.1, zu ersetzen. In jedem dieser Verzeichnisse liegt eine Datei 'PACKAGES', aus der R die benötigten Informationen über den Inhalt der Repositories erhält. Sehr bekannte Repositories stellen CRAN, BioConductor und das Omegahat Projekt bereit.

Mit Hilfe der Funktion `setRepositories()` können zu benutzende Repositories ausgewählt werden. Das CRAN Repository ist bereits voreingestellt.

CRAN ist ein Verbund vieler *Spiegelserver* (engl. *mirror*), die den Inhalt eines Hauptservers spiegeln. Generelle Regel sollte sein, einen örtlich in der Nähe liegenden Spiegelserver anzugeben, wenn R danach fragt. Dadurch wird der Hauptserver entlastet, es wird weniger Datenverkehr erzeugt, und das Herunterladen wird häufig deutlich beschleunigt. Manuell kann man einen Spiegelserver mit Hilfe der Funktion `chooseCRANmirror()` auswählen.

[7] Ein Archiv enthält mehrere Dateien.

Installation und Update von Source-Paketen

Falls man mehr als eine *Library* benutzt, muss diejenige *Library* angegeben werden, in bzw. von der ein Paket installiert, erneuert oder entfernt werden soll. Die Syntax, um ein Source-Paket von der Kommandozeile des Betriebssystems zu installieren, lautet

```
$ R CMD INSTALL -l /Pfad/zur/library Paket
```

Der Teil `-l /Pfad/zur/library` zur Spezifikation der *Library* muss nicht unbedingt angegeben werden. In dem Fall wird die erste *Library* aus der Umgebungsvariablen `R_LIBS` benutzt, falls diese gesetzt ist, sonst die Haupt-*Library*. Es ist zu beachten, dass die Datei '.Renviron' nicht von `R CMD` ausgewertet wird.

Ein Source-Paket `MeinPaket_0.0-1.tar.gz` kann beispielsweise in die *Library* `/home/user/meinR/meineLibrary` mit dem Befehl

```
$ R CMD INSTALL -l /home/user/meinR/meineLibrary
      MeinPaket_0.0-1.tar.gz
```

installiert werden.

In den meisten Fällen ist die folgende Alternative zur Installation jedoch viel komfortabler: Pakete können mit der Funktion `install.packages()` automatisch aus der R Konsole heraus heruntergeladen und installiert werden. Um ein Paket **MeinPaket** von CRAN in die *Library* `/Pfad/zur/library` zu installieren, reicht der Aufruf

```
> install.packages("MeinPaket", lib = "/Pfad/zur/library")
```

Analog zur Kommandozeilen-Variante kann die Spezifikation der *Library* unterbleiben. Dann wird in die erste *Library* im Suchpfad installiert. Genauer sucht `install.packages()`, falls das Argument `type = "source"` (Voreinstellung bei UNIX Installationen von R) gesetzt ist, nach verfügbaren Source-Paketen in den spezifizierten Repositories (z.B. auf einem CRAN Spiegelserver oder BioConductor), lädt die aktuellste Version herunter und installiert sie schließlich mit Hilfe von `R CMD INSTALL`. Andere mögliche Werte für das Argument `type` sind `"win.binary"` und `"mac.binary"` zur Installation von Binärpaketen (s.u.).

Die Funktion `update.packages()` hilft, Pakete auf dem System aktuell zu halten. Es wird dabei die Liste der installierten Pakete mit der Liste der in dem aktuellen Repository verfügbaren Pakete verglichen. Falls dort eine aktuellere Version liegt als die installierte Version eines Pakets, so wird diese Version heruntergeladen und installiert. Der Aufruf

```
> update.packages(checkBuilt = TRUE)
```

führt dazu, dass zusätzlich zur Suche nach neuen Versionen und deren Installation in die jeweils zugehörige *Library* auch überprüft wird, ob das Paket mit der aktuellen R Version installiert wurde. Ist dies nicht der Fall, werden auch solche Pakete neu installiert. Das hat den Vorteil, dass bei Installation einer neuen R Version nicht alle Pakete neu ausgewählt werden müssen, sondern die alte *Library* weiterbenutzt werden kann, indem sie einfach mit der oben gezeigten Zeile aktualisiert wird. Wie `install.packages()` unterstützt auch `update.packages()` das Argument `type`.

Man beachte, dass es geplant ist, die aktuellen Funktionen zum Paketmanagement in Zukunft durch `packageStatus()` (s. auch `?packageStatus`) zu ersetzen. Sowohl `packageStatus()` als auch das Paket **reposTools** vom BioConductor Projekt könnten hervorragende Werkzeuge für modernes Paketmanagement in R werden.

Als Voreinstellung werden unter Windows und MacOS X von Funktionen wie `install.packages()` und `update.packages()` Binärpakete installiert (s.u.).

Das Entfernen von Paketen erfolgt mit `R CMD REMOVE` in der Befehlszeile des Betriebssystems und wird im Handbuch „R Installation and Administration“ (R Development Core Team, 2006c) beschrieben.

Paketmanagement auf dem Macintosh

Aktuelle R Versionen unterstützen ausschließlich MacOS X ab einer minimalen Versionsnummer auf dem Macintosh. In der Kommandozeile dieses Betriebssystems verhält sich R wie unter UNIX.

Die R GUI besitzt Menüs für die Installation und Updates von Source-Paketen und Binärpaketen. Außerdem gibt es separate Menüs für CRAN Pakete, Pakete des BioConductor Projekts und Pakete aus lokalen Verzeichnissen.

Die Funktionen `install.packages()` und `update.packages()` haben unter MacOS X (mit derselben Syntax wie unter UNIX) durch Voreinstellung des Argumentes `type = "mac.binary"` andere Eigenschaften. Sie werten statt der Liste der Source-Pakete die Liste der Binärpakete aus und installieren die aktuellste Version des gewünschten Binärpakets.

Das Repository, in dem Funktionen wie `install.packages()` und `update.packages()` per Voreinstellung nach Binärpaketen suchen, ist zu finden unter: `CRAN/bin/macosx/universal/contrib/MajVer`. Dabei ist `CRAN` zu ersetzen durch den aktuell gewählten CRAN Spiegelserver und ‘MajVer’ durch den ersten Teil der Versionsnummer der benutzten R Version, also z.B. ‘2.3’ für R-2.3.1.

Paketmanagement unter Windows

Auf einem typischen Windows System fehlen zunächst einige Werkzeuge, die zur Installation von Source-Paketen benötigt werden. Auch ist die Shell (Kommandointerpreter) dieses Betriebssystems anders als übliche UNIX Shells.

Wo und wie die benötigten Werkzeuge für die Installation von Source-Paketen unter Windows gesammelt werden und wie das Betriebssystem konfiguriert werden muss, wird in dem Handbuch „R Installation and Administration“ (R Development Core Team, 2006c) beschrieben. Man sollte diese Beschreibung dringend Zeile für Zeile studieren, da das Befolgen nahezu jeder Zeile entscheidend zum Gelingen von `R CMD INSTALL` beiträgt. Ein konkreteres Beispiel zur Einrichtung der Umgebung geben Ligges und Murdoch (2005). Wer R bereits selbst unter Windows kompiliert hat, hat das System auch auf das Kompilieren von Source-Paketen vorbereitet.

Die Funktionen `install.packages()` und `update.packages()` haben unter Windows (mit derselben Syntax wie unter UNIX) durch Voreinstellung des Argumentes `type = "win.binary"` andere Eigenschaften. Sie werten statt der Liste der Source-Pakete die Liste der Binärpakete aus und installieren die aktuellste Version des gewünschten Binärpakets.

Das Repository, in dem diese Funktionen standardmäßig nach Paketen suchen, ist zu finden unter: `CRAN/bin/windows/contrib/MajVer/`. Dabei ist `CRAN` zu ersetzen durch den aktuell gewählten CRAN Spiegelserver. Des Weiteren wird zusätzlich in einem Repository von Brian D. Ripley mit Namen „CRAN (extras)[8]“ gesucht, in dem Pakete liegen, die nur mit größerem Aufwand für Windows kompiliert werden können. Jeweils ist ‘MajVer’ durch den ersten Teil der Versionsnummer der benutzten R Version zu ersetzen, also z.B. ‘2.3’ für R-2.3.1. Die Datei ‘ReadMe’ in diesen Repositories und auch die Datei `CRAN/bin/windows/contrib/ReadMe` enthalten wichtige Informationen zur Verfügbarkeit von Paketen. Dort ist auch beschrieben, was mit Paketen geschieht, die nicht der Standard-Qualitätsprüfung von `R CMD check` genügen.

In der GUI von R unter Windows (‘Rgui.exe’) gibt es das Menü „Pakete“ (in der englischen Version: „Packages“), das eine grafische Oberfläche für die Funktionen `install.packages()`, `update.packages()` und `library()` bereitstellt. Auch Pakete des BioConductor Projekts und von Omegahat (über die Wahl von Repositories) sowie Pakete von der lokalen Festplatte lassen sich mit Hilfe des Menüs installieren.

Es ist zu beachten, dass mit Hilfe des Menüs ausschließlich in die erste Library im Suchpfad (`.libPaths()[1]`) installiert werden kann. Zur Installation in eine andere Library sollte `install.packages()` direkt benutzt

[8] CRAN (extras) ist unter `http://www.stats.ox.ac.uk/pub/RWin/bin/windows/contrib/MajVer/` erreichbar

werden. Zum Beispiel kann das Binärpaket **MeinPaket**, welches sich in einer Datei `c:\meinR\MeinPaket_0.0-1.zip` befindet, wie folgt in die Library `c:\meinR\meineLibrary` installiert werden:

```
> install.packages(
+     "c:/meinR/MeinPaket_0.0-1.zip",
+     lib = "c:/meinR/meineLibrary", repos = NULL)
```

10.4 Struktur von Paketen

Die Struktur eines Pakets ist vor allem für Entwickler eigener Pakete interessant. Details zur Struktur von Paketen und der Entwicklung eigener Pakete sind im Handbuch „Writing R Extensions“ (R Development Core Team, 2006e) zu finden.

Ein Source-Paket besteht aus einigen Standard-Dateien und Verzeichnissen, die bestimmte Arten von Dateien enthalten sollten:

- `DESCRIPTION`, Datei mit standardisierten Erläuterungen zu Autor, Lizenz, Titel, Abhängigkeiten, Versionsnummer des Pakets usw. Die Datei wird automatisch von Skripten weiterverarbeitet.
- `INDEX`, Datei, die automatisch generiert werden kann und den Index aller Daten und Funktionen enthält.
- `NAMESPACE`, Datei, die einen Namespace für das Paket einrichtet (s. Abschn. 4.3 und 10.6).
- `R/`, Verzeichnis mit Dateien, die R Code enthalten (s. Abschn. 10.5).
- `data/`, Verzeichnis mit Datensätze (s. Abschn. 10.5).
- `demo/`, Verzeichnis, das R Code zu Demonstrationszwecken enthält.
- `exec/`, Verzeichnis mit ausführbaren Dateien und Skripts.
- `inst/`, Verzeichnis, dessen Inhalt bei Installation des Pakets in das zentrale Verzeichnis des Binärpakets kopiert wird.
- `man/`, Verzeichnis, das Dokumentation (Dateien im ‘*.Rd’ Format) enthält (s. Abschn. 10.7.1).
- `src/`, Verzeichnis, das C, C++ oder Fortran Sources enthalten kann.
- `tests/`, Verzeichnis mit Dateien für Software-Validierung.

Abgesehen von der Datei ‘DESCRIPTION’ sind alle anderen Punkte obiger Liste optional. Ein nützliches Paket wird aber sowohl Dokumentation als auch R Code und/oder Datensätze enthalten. Insbesondere die Datei ‘DESCRIPTION’ sollte sorgfältig ausgefüllt werden, da sie wichtige Informationen über ein Paket enthält. Auch die Dokumentation sollte in keinem Fall vernachlässigt werden.

Die Erstellung einer wie oben angegebenen Struktur wird Entwicklern von der Funktion `package.skeleton()` abgenommen. Der Aufruf

```
> package.skeleton(name = "MeinPaket", Objektliste, path = ".")
 Creating directories ...
 Creating DESCRIPTION ...
 Creating Read-and-delete-mes ...
 Saving functions and data ...
 Making help files ...
 Created file named 'd://MeinPaket/man/MeinPaket-package.Rd'.
 Edit the file and move it to the appropriate directory.
 Created file named 'd://MeinPaket/man/x.Rd'.
 Edit the file and move it to the appropriate directory.
 Done.
 Further steps are described in d://MeinPaket/Read-and-delete-me
```

legt im spezifizierten Pfad (`"."`) ein Verzeichnis `MeinPaket` an, das die oben angegebene Struktur bereits enthält. Die in der `Objektliste` angegebenen Objekte sind auch schon als R Code bzw. Datensätze enthalten — inklusive vorgefertigter Dokumentationsdateien im '*.Rd' Format, die nur noch ausgefüllt werden müssen.

Pakete packen

Wenn die Dateien wie erforderlich editiert wurden, kann das Paket nun durch Eingabe von

```
$ R CMD build MeinPaket
```

auf der Kommandozeile gepackt und dann mit `R CMD INSTALL` (s. Abschn. 10.3) installiert werden — auch auf anderen Systemen nach dem Verteilen.

Qualitätskontrolle

Wenn das Paket ohne Fehlermeldung mit `R CMD build` gebaut und dann mit `R CMD INSTALL` erfolgreich installiert worden ist, kann es mit `R CMD check` auf Konsistenz, Installierbarkeit, Dokumentation, Ausführbarkeit der Beispiele usw. überprüft werden. Diese Überprüfung mit `R CMD check` muss ein Paket auch bestehen, wenn es auf CRAN veröffentlicht werden soll. Insbesondere können für diese Überprüfung im Verzeichnis `tests/` auch Testfälle definiert werden, die R mit den im Paket definierten Funktionen rechnet. Die Ergebnisse werden dann mit in demselben Verzeichnis vorgegebenen „wahren“ Ergebnissen von `R CMD check` automatisch verglichen.

`R CMD check` bietet also die Möglichkeit einer gewissen Qualitätskontrolle von Paketen, die sehr nützlich ist, denn niemand möchte Pakete abgeben, deren Dokumentation nicht zum Code passt bzw. die sich nicht auf anderen Systemen wegen nicht erfüllter Abhängigkeiten installieren lassen.

Luke Tierney arbeitet an einem Paket **codetools**[9], das erweiterte Möglichkeiten zur Überprüfung von Code in Funktionen bietet. Zum Beispiel kann überprüft werden, ob ungenutzte Objekte erzeugt werden oder ob Objekte, die nicht übergeben werden, im Code verwendet werden.

Einbinden von C, C++, Fortran

Wie das Einbinden von C, C++, und Fortran Code in R gelingt, wurde in Abschn. 9.1 erläutert. Code, der sich, wie dort beschrieben, mit `R CMD SHLIB` kompilieren und einbinden lässt, kann einfach in das Verzeichnis `src/` eines Pakets gelegt werden. Er wird bei Installation des Pakets durch das Kommando `R CMD INSTALL` automatisch kompiliert. Details gibt es im Handbuch „Writing R Extensions“ (R Development Core Team, 2006e).

10.5 Funktionen und Daten in Paketen

Jeder Datensatz (im `data/` Verzeichnis) und jede Funktion (im Verzeichnis `R/`) stehen in einem Paket i.A. in einer separaten Datei. Der Name der Datei wird i.d.R. nach dem des enthaltenen Objekts, sei es Funktion oder Datensatz, gewählt. Auf diese Weise ist der Überblick auch bei einer großen Anzahl an Funktionen leichter zu behalten. Zu jedem Datensatz und jeder Funktion sollte es Dokumentation geben (s. Abschn. 10.7.1).

Es kommt vor, dass eng zusammengehörende Funktionen, etwa eine generische Funktion und zugehörige Methoden, in einer Datei zusammengefasst werden.

Code, der beim Laden eines Pakets mit `library()` direkt ausgeführt werden soll, also nicht unbedingt eine Funktion definiert, wird per Konvention in der Datei ‘R/zzz.R’ abgelegt. Dazu gehört z.B. das Laden von externem Code in einer DLL.

Datensätze können mit der Funktion `data()` geladen werden und müssen dazu in einem der folgenden Formate im `data/` Verzeichnis vorliegen:

- als „rechteckige“ Textdatei (durch Leerzeichen oder Kommata getrennt) mit Endung ‘.csv’, ‘.tab’ oder ‘.txt’,
- als mit `dump()` exportierter R code (ASCII, Endung ‘.r’ oder ‘.R’) oder
- als mit `save()` exportierte R Binärdatei (Endung ‘.rda’ bzw. ‘.RData’).

[9] Das Paket **codetools** ist z.Zt. erhältlich unter der URL `http://www.stat.uiowa.edu/~luke/R/codetools/`.

Lazy Loading

Lazy Loading (Ripley, 2004) ist ein Mechanismus, der es erlaubt, dass Objekte aus Paketen erst dann geladen werden, wenn sie benötigt werden. Eine Funktion braucht beispielsweise erst bei ihrem ersten Aufruf geladen zu werden. Lazy Loading spart damit sowohl Zeit beim Starten von R und beim Laden von Paketen als auch Speicherplatz, weil fast nie alle Funktionen während einer R Sitzung benötigt werden.

Bei der Installation eines Pakets werden die Funktionen in einer Datenbank abgelegt. Beim Laden des Pakets werden dann zunächst nur die Namen aus der Datenbank registriert. Lazy Loading kann für ein Paket explizit durch Setzen des Eintrags `LazyLoad: yes` in der Datei 'DESCRIPTION' aktiviert werden.

Das Laden von Daten mit `data()` ist nicht mehr notwendig, wenn von dem Lazy Loading Mechanismus für Daten Gebrauch gemacht wird, der sich analog zu dem Mechanismus für Pakete verhält. Lazy Loading für Daten kann durch den Eintrag `LazyData: yes` in der 'DESCRIPTION' Datei des Pakets eingeschaltet werden.

Details zur Nutzung von Lazy Loading in Paketen werden von Ripley (2004) und im Handbuch „Writing R Extensions" (R Development Core Team, 2006e) erläutert.

10.6 Namespaces

Die Wirkung und Eigenschaften von Namespaces wurden bereits in Abschn. 4.3 auf S. 81 beschrieben.

Im Wesentlichen erlaubt es der Namespace-Mechanismus zu definieren, welche Objekte exportiert werden, so dass Benutzer sie verwenden können, und welche Objekte aus anderen Namespaces importiert werden. Ein Paket mit einem Namespace wird wie jedes andere Paket geladen und in den Suchpfad eingehängt, wobei aber nur exportierte Objekte in der in den Suchpfad eingehängten Umgebung erscheinen.

An dieser Stelle wird erläutert, wie mit der Datei 'NAMESPACE' (s. Abschn. 10.4) ein Namespace für ein Paket eingerichtet werden kann. In dieser Datei werden also die zu importierenden und zu exportierenden Objekte definiert. Es müssen hier aber auch zu ladender externer Code (in Form einer DLL) und im Paket definierte S3-Methoden angegeben werden.

Mit dem Befehl `export()` (es wird hier von Befehl gesprochen, da Einträge in der Datei 'NAMESPACE' keine R Funktionen sind) wird festgelegt,

welche Objekte exportiert werden sollen, während `exportPattern()` eine ganze Reihe von Objekten exportieren kann, die einer bestimmten Struktur folgen. Die Funktion `import()` importiert alle Objekte aus einem anderen angegebenen Namespace. Ausgewählte Objekte eines Namespace lassen sich mit `importFrom()` importieren.

Mit Hilfe von `S3method()` lassen sich S3-Methoden registrieren, die ohne einen entsprechenden Namespace Eintrag nicht gefunden werden. Externer Code sollte mit `useDynLib()` geladen werden.

Eine 'NAMESPACE' Datei hat dann z.B. folgendes Aussehen:

```
useDynLib(NameDerDLL)
exportPattern("^[^\\.]")
export(".Funktion")
S3method(print, meineKlasse)
import(meinPaket1)
importFrom(meinPaket2, f)
```

Zunächst wird in diesem Beispiel die externe Library mit Namen `NameDerDLL` geladen. Dann werden mit `exportPattern("^[^\\.]")` alle Objekte exportiert, deren Namen nicht mit einem Punkt anfangen. Als Nächstes wird zusätzlich das Objekt `.Funktion` exportiert. Des Weiteren wird eine S3-Methode `print.meineKlasse()` zur generischen Funktion `print()` für die Klasse `meineKlasse` registriert. Außerdem werden alle Objekte aus dem Namespace des Pakets **meinPaket1** importiert sowie das Objekt `f` aus dem Namespace des Pakets **meinPaket2**.

Für den Umgang mit S4-Klassen und -Methoden sind die folgenden Befehle nützlich, die analog zu den oben beschriebenen Befehlen zu benutzen sind: `exportClasses()`, `exportMethods()`, `importClassesFrom()` und `importMethodsFrom()`.

Details zur Definition eines Namespace, insbesondere auch zum Umgang mit S4-Klassen und -Methoden, sind im Handbuch „Writing R Extensions“ (R Development Core Team, 2006e) zu finden.

10.7 Dokumentation

Zu einer guten Dokumentation von Code gehört nicht nur das ausführliche Kommentieren im Code selbst (s. Abschn. 5.1), das zum schnellen Verstehen eines Programms erforderlich ist, sondern auch das Schreiben von Dokumentation für die Benutzer.

Dokumentation in Form von Hilfeseiten für alle Datensätze und Funktionen wird nicht nur von jedem R Paket erwartet, das auf CRAN veröffentlicht wird, sondern sollte auch bei interner Verwendung vorhanden sein. Sol-

che Hilfeseiten lassen sich sehr komfortabel im Rd Format erstellen (s. Abschn. 10.7.1).

Wer einem Paket kurze Berichte und kurze Handbücher beilegen möchte, die über den üblichen Inhalt von Hilfeseiten hinausgehen und höhere Ansprüche an das Layout stellen, sollte erwägen, eine s.g. Vignette mit Hilfe von SWeave (s. Abschn. 10.7.2) zu erstellen.

10.7.1 Das Rd Format

Dokumentation (Dateien im '*.Rd' Format) wird im Verzeichnis `man/` eines Pakets abgelegt.

Bei der Installation eines Pakets mit `R CMD INSTALL` werden daraus eine Reihe anderer Formate erzeugt, wie sie dann auch in Binär-Paketen vorhanden sind, z.B. die Formate HTML, LaTeX und formatierten Text sowie unter Windows zusätzlich '.chm' (Compiled HTML). Einige dieser Formate eignen sich dazu, die Hilfeseiten komfortabel am Rechner zu betrachten (HTML, Compiled HTML), andere eignen sich für wohlformatierten Ausdruck (LaTeX). Zur Benutzung des Hilfesystems sei auf Abschn. 2.4.1 verwiesen.

Dokumentation im '*.Rd' Format kann auch direkt mit den Kommandos `R CMD Rdconv` in die Formate LaTeX, HTML und formatierter Text sowie mit `R CMD Rd2dvi` in die Formate DVI und PDF konvertiert werden. Für die Konvertierung wird die Software Perl benötigt.

Weil Dokumentation extrem wichtig ist, bietet R auch Werkzeuge zum Überprüfen von Dokumentation. Das Kommando `R CMD check` (s. auch Abschn. 10.4) überprüft u.a., ob

- Dokumentation für alle Datensätze und Funktionen eines Pakets existiert,
- alle Argumente einer Funktion dokumentiert sind,
- die Voreinstellungen der Argumente korrekt dokumentiert sind,
- sich die '.Rd'-Dateien in die anderen Formate konvertieren lassen und
- die abgegebenen Beispiele ohne Fehler ausführbar sind.

Die Struktur einer Rd-Datei

Wer zum Erstellen eines Pakets nicht die Funktion `package.skeleton()` benutzt hat oder einem Paket nachträglich Dokumentation zu Datensätzen oder Funktionen hinzufügen möchte, kann die Funktion `prompt()` benutzen, die ein Gerüst für die Dokumentation in einer '.Rd'-Datei erstellt. Einige Einträge der Datei sind dann schon ausgefüllt, etwa der Aufruf einer Funktion oder deren zu dokumentierende Argumente.

'.Rd'-Dateien haben eine LaTeX-ähnliche Syntax und können bzw. müssen u.a. folgende Abschnitte enthalten:

- `\name`, Name der Hilfeseite (oft identisch mit `\alias`),
- `\alias`, Name(n) der Funktion(en), die mit dieser Datei beschrieben wird (werden),
- `\title`, Überschrift der Hilfeseite,
- `\description`, kurze Beschreibung der Funktion,
- `\usage`, wie die Funktion (inkl. aller Argumente) aufgerufen wird,
- `\arguments`, Liste aller Argumente und deren Bedeutung,
- `\value`, Liste der zurückgegebenen Werte,
- `\details`, eine detaillierte Beschreibung der Funktion,
- `\references`, Hinweise auf Literaturstellen,
- `\seealso`, Hinweise und Verknüpfungen zu relevanter Dokumentation anderer Funktionen,
- `\examples`, Beispiele, wie die Funktion benutzen werden kann,
- `\concept`, Schlüsselwörter, anhand derer diese Hilfeseite durch Suchfunktionen gefunden werden soll,
- `\keyword`, ein „Keyword“ (aus einer vorgegebenen Liste an Keywords), damit die Funktion im R Hilfesystem eingeordnet werden kann.

Es gibt weitere (und selbst definierbare) Abschnitte, spezielle Möglichkeiten zum Schreiben von mathematischer Notation, Befehle zum Formatieren von Text, Möglichkeiten zum Einfügen von URLs und Verknüpfungen zu anderen Hilfeseiten sowie viele weitere Möglichkeiten, deren Beschreibung hier den Rahmen sprengen würde. Stattdessen sei auf das Handbuch „Writing R Extensions“ (R Development Core Team, 2006e) als Referenz zum Schreiben von Hilfeseiten verwiesen.

Beispiele zu ‘.Rd’-Dateien findet man in den `man/`-Verzeichnissen bereits vorhandener Pakete.

10.7.2 SWeave

Eingangs wurde erwähnt, dass SWeave sich sehr gut dazu eignet, Handbücher in Form von Vignetten für Pakete zu schreiben. Es ist aber noch viel mehr als das.

Die tägliche Arbeit vieler sich mit Statistik befassender Personen besteht darin, mit Daten zu arbeiten, sie aufzubereiten, zu importieren, zu exportieren, zu analysieren und letztendlich einen Bericht zu verfassen. Oft wird dabei auch Code geschrieben, gerade wenn Abläufe automatisiert werden sollen oder die üblichen Methoden angepasst werden müssen. Rossini (2001) beschreibt Ideen, wie man in solchen Situationen die Datenanalyse mit allen Kommentaren, Code und schließlich dem Bericht in einem Dokument zusammenfassen könnte. Dieses Dokument soll automatisch zu verarbeiten sein, so dass die

komplette Datenanalyse reproduzierbar ist. Aber die Datenanalyse soll auch einfach bei leicht geändertem Datenbestand so wiederholbar sein, dass automatisch alle Grafiken und Zahlen im Bericht abgeändert werden.

Diese Ideen wurden von Leisch (2002a) mit SWeave für R umgesetzt. Und zwar wird hier LaTeX (zum Schreiben des Berichts) und R Code (für die Berechnungen und Erzeugung von Grafiken) gemischt. Ausgewerteter R Code und produzierte Bilder werden automatisch mit dem vorhandenen LaTeX Code zu einer einzigen LaTeX-Datei zusammengestellt.

Als weitere Referenzen sind Leisch (2002b, 2003a) und das SWeave Handbuch (Leisch, 2006) zu erwähnen. Diese Referenzen möge studieren, wer erste SWeave Dokumente schreiben möchte. Auch hier ist es empfehlenswert, sich als Beispiel bereits vorhandene SWeave Dokumente anzuschauen.

Vignetten mit SWeave

Das Erstellen von Vignetten mit SWeave wird von Leisch (2003b) und im Handbuch „Writing R Extensions“ (R Development Core Team, 2006e) beschrieben.

An dieser Stelle sei erwähnt, dass Vignetten in das Verzeichnis `inst/doc/` eines Source-Pakets gelegt werden müssen. Dort werden sie automatisch bei dem Packen des Pakets mit `R CMD build` verarbeitet, d.h. es wird eine PDF-Datei erstellt. Diese PDF-Datei wird bei Installation des Pakets in der Hilfe verlinkt. Das Kommando `R CMD check` überprüft die Ausführbarkeit des in den Vignetten enthaltenen R Codes.

A

R installieren, konfigurieren und benutzen

Dieser Teil des Anhangs soll Hilfestellung beim Herunterladen und bei der Installation von R geben. Außerdem wird gezeigt, wie R gestartet wird. In diesem Zusammenhang werden auch Besonderheiten bei Installation und Start von R auf gängigen Betriebssystemen vorgestellt. Die Installation und Verwaltung von Paketen wurde in Abschn. 10.3 vorgestellt.

Wenn sich das Vorgehen bei verschiedenen Betriebssystemen unterscheidet, so wird in den folgenden Abschnitten zunächst immer „UNIX und Co." vorgestellt. Darunter sind alle UNIX-artigen Betriebssysteme, z.B. Linux und Solaris, zusammengefasst. Für den Macintosh wird zurzeit als einziges Betriebssystem MacOS X unterstützt. Wenn vorhanden, werden wesentliche Unterschiede zu „UNIX und Co." erwähnt. Auch für die große Anzahl der Windows Benutzer gibt es kurze Abschnitte zu Eigenarten von R für Windows.

Für einige hier nicht aufgeführte Details zur Installation und Wartung von R sei auf das Handbuch „R Installation and Administration" (R Development Core Team, 2006c) verwiesen.

A.1 R herunterladen und installieren

Einleitend (Kap. 1) wurde bereits erwähnt, dass es sich bei *CRAN* (*C*omprehensive *R A*rchive *N*etwork) um ein Netz von Servern handelt, das weltweit den Quellcode und die Binärversionen von R für diverse Betriebssysteme bereitstellt.

Der zentrale Server von CRAN ist zu erreichen unter der URL `http://CRAN.R-project.org`. Dort ist eine aktuelle Liste der Spiegelserver[1] (engl. *mirror*) zu finden, aus der man sich einen Server aussucht, der geographisch möglichst nahe am aktuellen Standort liegt. Dadurch erhält man eine

[1] `http://CRAN.R-project.org/mirrors.html`

möglichst schnelle Verbindung und entlastet zusätzlich den zentralen Server. Es macht Sinn, sich den hier ausgewählten Server für den späteren Gebrauch zu merken. Zum Beispiel können von dort zusätzliche Pakete heruntergeladen und auf dem neuesten Stand gehalten werden (s. Abschn. 10.3).

Nachdem ein Server ausgewählt wurde, gibt es die Möglichkeit, den Quellcode (*sources*) oder eine bereits fertig kompilierte Binärversion von R herunterzuladen. Die Installation einer kompilierten Binärversion ist oft etwas unproblematischer und schneller, es gibt sie aber nicht für alle Plattformen.

Wer den Quellcode herunterlädt, muss R noch kompilieren, hat dafür aber auch die Möglichkeit, R für die gegebene Plattform besonders gut anzupassen. Beispielsweise gewinnt R für große Matrixoperationen an Geschwindigkeit, wenn man gegen ATLAS[2] linkt (R Development Core Team, 2006c). Auch andere spezialisierte BLAS[3] Bibliotheken können benutzt werden, zum Beispiel die AMD Core Math Library (ACML, s. Advanced Micro Devices, 2006). Sowohl ATLAS als auch ACML sind speziell für die jeweilige Plattform angepasste und optimierte Bibliotheken, die Routinen für Matrixoperationen bereitstellen.

Als *Plattform* bezeichnet man dabei die jeweilige Kombination aus Rechnertyp (bei den meisten Lesern sicherlich ein PC mit x86-kompatiblem Prozessor von Intel oder AMD) und Betriebssystem (z.B. Windows).

UNIX und Co.

Für viele Linux Distributionen gibt es Binärversionen von R in Form von '.rpm' (z.B. RedHat, SuSE) oder '.deb' (Debian) Dateien auf CRAN.

Man braucht sich aber vor der Installation und dem Kompilieren aus den Quellen nicht zu scheuen. Meist sind nämlich die notwendigen Werkzeuge dazu schon installiert. Notwendig bzw. empfehlenswert sind Perl, C und Fortran Compiler, make und einige Entwickler-Bibliotheken (insbesondere diejenigen für X11 und libreadline sind sinnvoll, fehlen aber manchmal).

Die Quellen für R-2.3.1 sind im Paket 'R-2.3.1.tar.gz' verpackt (die Versionsnummern sind entsprechend anzupassen). Wurde es heruntergeladen, kann wie folgt standardmäßig (1) entpackt, (2) in das Verzeichnis gewechselt, (3) konfiguriert, (4) kompiliert und (5) installiert werden:

```
$ tar xfz R-2.3.1.tar.gz
$ cd R-2.3.1
$ ./configure
$ make
$ make install
```

[2] Automatically Tuned Linear Algebra Software, `http://math-atlas.sourceforge.net`

[3] Basic Linear Algebra Subprograms

Für spezielle Einstellungen und weitere Details sei auf das Handbuch „R Installation and Administration“ (R Development Core Team, 2006c), Hornik (2006) und die Datei ‘INSTALL’ im Hauptverzeichnis der R Quellen verwiesen.

Ein anschließendes

```
$ make check
```

überprüft, ob R richtig kompiliert wurde und korrekt arbeitet.

Mit den Voreinstellungen von `./configure` ist R durch die o.g. Prozedur im Verzeichnis `/usr/local/lib/R` installiert worden. Sollte der Pfad `/usr/local/bin` als Suchpfad für ausführbare Dateien gesetzt sein, kann R nun einfach mit

```
$ R
```

gestartet werden. Es wird derjenige Pfad als Arbeitsverzeichnis gesetzt, aus dem R aufgerufen wird.

Macintosh

Für den Macintosh unter MacOS X empfiehlt sich i.A. die Installation der Binärversion. Seit R-2.3.0 wird die Binärversion als *universal binary* zur Verfügung gestellt, also sowohl für die PowerPC- als auch die i686-Architektur. Für R-2.3.1 liegt die Binärversion beispielsweise als *Diskimage* Datei ‘R-2.3.1.dmg’ auf den CRAN Spiegelservern. Die Installation weist keine besonderen Hürden auf und erfolgt mit einem Doppelklick auf die im Diskimage enthaltene Datei ‘R.mpkg’.

Zum Kompilieren von R aus dem Quellcode empfiehlt sich dringend ein Blick in Iacus et al. (2006).

Windows — *Binärversion*

Die Binärversion von R für Windows wird in Form eines unter diesem Betriebssystem üblichen Setup-Programms bereitgestellt. Für R-2.3.1 ist das die Datei ‘R-2.3.1-win32.exe’ (auch hier ist die Versionsnummer entsprechend anzupassen).

Das Setup-Programm legt eine Verknüpfung im Windows-Startmenü an. Ein Klick auf diese Verknüpfung startet R, genauer das Programm `Pfad_zu_R/bin/RGui.exe`. Das Programm `Pfad_zu_R/bin/Rterm.exe` ist zur Verwendung unter der Windows command shell („Eingabeaufforderung“[4]) gedacht.

Oft möchte man nicht das voreingestellte Arbeitsverzeichnis verwenden, sondern ein eigenes Verzeichnis angeben, oder je nach Projekt eigene Verzeichnisse benutzen. Neben den in Abschn. 2.6 auf S. 25 beschriebenen Methoden

[4] Wird häufig auch als „MS-DOS“ Fenster bezeichnet, obwohl es bei modernen Windows Versionen nichts mit DOS zu tun hat.

zum Setzen des Arbeitsverzeichnisses kann es auch in der Verknüpfung gesetzt werden, mit der R gestartet wird (etwa eine Verknüpfung auf dem Desktop oder im Startmenü, s. Abb. A.1 auf S. 214).

Für Windows sind bereits kompilierte Versionen der Datei 'Rblas.dll' auf CRAN vorhanden[5], die gegen ATLAS gelinkt wurden, und sich bei Operationen auf großen Matrizen durch hohe Geschwindigkeit auszeichnen, manchmal aber auch zu leichten Einbußen führen. Details sind in der entsprechenden 'ReadMe'-Datei nachzulesen.

Windows — R *selbst kompilieren*

Die Vorarbeiten, um R unter Windows aus den Quellen zu kompilieren, erfordern etwas mehr Mühe als unter den anderen Betriebssystemen. Eine Anleitung ist in dem Handbuch „R Installation and Administration" (R Development Core Team, 2006c) zu finden, die Zeile für Zeile möglichst exakt befolgt werden sollte, da nahezu jedes Detail entscheidend für das Gelingen ist. Ein Beispiel zum Einrichten einer entsprechenden Umgebung zum Kompilieren von R wird in Ligges und Murdoch (2005) gegeben. Nach erfolgreichem Kompilieren sollte man noch eine Verknüpfung zur Datei `Pfad_zu_R/bin/RGui.exe` im Startmenü anlegen, die das komfortable Starten von R ermöglicht.

A.2 R konfigurieren

R lässt sich in weiten Teilen konfigurieren. Dazu stehen verschiedene Mechanismen zur Verfügung: Kommandozeilenoptionen beim Aufruf, Umgebungsvariablen und Konfigurationsdateien. Auf diese Mechanismen und Spezialitäten unter Windows wird im Folgenden eingegangen. Insbesondere sei auch die Lektüre des Handbuchs „R Installation and Administration" (R Development Core Team, 2006c) und der Hilfeseite `?Startup` empfohlen. Letztere gibt einen genauen Ablaufplan des Starts von R und zeigt, wann welcher Konfigurationsschritt durchgeführt wird.

Sprache und Schriften

R bietet einige Möglichkeiten zur Internationalisierung (Ripley, 2005a), d.h. entsprechende Konfigurierbarkeit von Sprache und Schrift.

Fehler- und Warnmeldungen sowie Hinweise können in der Sprache des Benutzers ausgegeben werden, sofern entsprechende Übersetzungen vorliegen. Liegt keine entsprechende Übersetzung vor, so wird auf Englisch zurückgegriffen. Per Voreinstellung wählt R die Sprache der aktuell vom Betriebssystem

[5] `http://CRAN.R-project.org/bin/windows/contrib/ATLAS/`

verwendeten Lokalisierung. Dazu werden die Umgebungsvariablen `LANGUAGE`, `LC_ALL`, `LC_MESSAGES` und `LANG` ausgewertet und die erste davon verwertbare Aussage benutzt. Möchte man R dazu bewegen, die deutsche Übersetzung zu verwenden, wenn man unter einem anders lokalisierten Betriebssystem arbeitet, so spezifiziert man die Umgebungsvariable `LANGUAGE = de`. Wer sich andererseits an die englischen Meldungen gewöhnt hat und sich nicht mit den deutschen Meldungen anfreunden kann, wählt `LANGUAGE = en`.

Auf die Umstellung auf andere Schriften, die beispielsweise von Benutzern im arabischen, asiatischen und russischen Raum bevorzugt werden, wird hier nicht eingegangen. Details dazu (ebenso wie zur Sprache) sind zu finden bei Ripley (2005a) und im Handbuch „R Installation and Administration“ (R Development Core Team, 2006c).

Kommandozeilenoptionen

R kann beim Aufruf teilweise mit Hilfe von Kommandozeilenoptionen konfiguriert werden. Der Aufruf

```
$ R --help
```

zeigt eine kurze Hilfe zu diesen Optionen an. Häufiger verwendete Optionen sind:

- `--no-save`: Der Workspace soll bei Beenden einer Sitzung nicht gespeichert werden (ohne Nachfrage). Das ist besonders bei nicht interaktiven Sitzungen nützlich.
- `--vanilla`: Außer der Option `--no-save` werden noch `--no-restore` (nicht einen alten Workspace wiederherstellen) und `--no-site-file` (keine weiteren Konfigurationsdateien einlesen) als Optionen impliziert. Dadurch wird ein minimales R gestartet, was sehr nützlich zur Fehlersuche sein kann.

Auf die Kommandozeilenoptionen, mit denen R gestartet worden ist, kann innerhalb von R mit `commandArgs()` zugegriffen werden. Die Hilfeseite `?Memory` zeigt, wie in der Kommandozeile Minimal- und Maximalwerte für den Speicherverbrauch gesetzt werden können.

Umgebungsvariablen

An verschiedenen Stellen in den R Handbüchern gibt es Hinweise dazu, wie ein bestimmtes Verhalten von R durch das Setzen von Umgebungsvariablen beeinflusst werden kann. Das Setzen der Umgebungsvariablen kann dabei

- in der bei dem jeweiligen Betriebssystem üblichen Form,
- in den im nächsten Teilabschnitt beschriebenen Konfigurationsdateien zum Setzen solcher Variablen,
- beim Aufruf von R in der Kommandozeile und
- innerhalb von R selbst erfolgen.

Das Setzen einer Umgebungsvariablen innerhalb von R gelingt mit der Funktion `Sys.putenv()`. Mit `Sys.getenv()` können Umgebungsvariablen ausgelesen werden, z.B. zeigt `Sys.getenv("R_HOME")` das Verzeichnis an, in dem R installiert ist.

Nützliche und häufig auftretende Umgebungsvariablen sind u.a.:

`LANGUAGE`: Sprache, die R für Meldungen und für die GUI verwenden soll.
`R_ENVIRON`: Hier kann der Ort einer Datei zum Einlesen weiterer Umgebungsvariablen angegeben werden (s.u.).
`R_HOME`: Das Verzeichnis, in dem R installiert ist.
`R_LIBS`: Eine Liste von *Library*-Verzeichnissen, die R Pakete enthalten (s. Kap. 10). Diese Variable sollte am besten in einer Konfigurationsdatei gesetzt werden.
`R_PROFILE`: Hier kann der Ort einer Datei mit Konfigurationseinstellungen angegeben werden (s.u.).

Konfigurationsdateien

Es gibt grundsätzlich zwei Arten von Konfigurationsdateien, nämlich solche zum Setzen von Umgebungsvariablen und solche, die *Profiles* enthalten. Man beachte auch die Ausführungen in der Hilfe `?Startup`.

Umgebungsvariablen (s.o.), die sich auf R beziehen, können auch in entsprechenden Konfigurationsdateien angegeben werden. Diese werden eingelesen, falls die Kommandozeilenoption `--no-environ` nicht gesetzt ist. Dann wird zunächst nach einer Datei '.Renviron' des Benutzers gesucht, und zwar zunächst im aktuellen Arbeitsverzeichnis und dann im *home*-Verzeichnis des Benutzers. Falls eine solche Datei existiert, werden die darin definierten Umgebungsvariablen gesetzt.

Danach wird nach einer globalen Datei gesucht. Dabei wird die in der Umgebungsvariable `R_ENVIRON` angegebene Datei benutzt, sonst die Datei 'R_HOME/etc/Renviron.site', falls sie existiert.

In den *Profile* Dateien steht ausführbarer R Code. Falls nicht die Kommandozeilenoption `--no-site-file` gesetzt wurde, sucht R hier nach einer globalen Datei. Analog zu oben wird zunächst geschaut, ob die Umgebungsvariable `R_PROFILE` einen entsprechenden Dateinamen enthält, sonst wird die Datei 'R_HOME/etc/Rprofile.site' benutzt, falls sie existiert. Man beachte, dass

in dieser globalen *Profile* Datei enthaltener Code direkt in das Paket **base** geladen wird.

Danach wird nach einer '.Rprofile' des aktuellen Benutzers im Arbeitsverzeichnis und im *home*-Verzeichnis des Benutzers gesucht. Diese Datei wird, falls vorhanden, ausgeführt und darin enthaltener Code in den Workspace geladen.

Die Unterscheidung von globalen Dateien und eigenen Konfigurationsdateien ist sehr nützlich. So kann man in Netzwerken nämlich eine zentrale R Installation passend für die Umgebung konfigurieren, während Benutzer trotzdem noch individuelle Einstellungen vornehmen können.

Konfiguration unter Windows

Unter Windows sind zusätzlich zu den oben genannten auch noch folgende Kommandozeilenoptionen wichtig:

- `--max-mem-size`: Mit dieser Option kann das Maximum (unter Windows voreingestellt: Minimum von vorhandenem Hauptspeicher des Rechners und 1024 MB) des von R verwendeten Hauptspeichers gesetzt werden (s. auch Abschn. 5.3, S. 112), z.B. auf 1536MB mit `--max-mem-size=1536M`.
- `--mdi`: Die R GUI wird standardmäßig im MDI Modus gestartet. Hier werden in einem großen, umschließenden R Fenster alle weiteren (z.B. Konsole, Grafikfenster usw.) geöffnet.
- `--sdi`: Im SDI Modus wird jedes Fenster von der R GUI separat von Windows verwaltet, und es gibt kein umschließendes Fenster. Wenige Pakete brauchen diese auch vom Autor favorisierte Einstellung zwingend.

Diese Kommandozeilenoptionen spezifiziert man in der Verknüpfung, mit der R aufgerufen wird. Solche Verknüpfungen können mit einem rechten Maus-Klick geändert werden.

Abb. A.1 zeigt ein Fenster, das zur Änderung von Verknüpfungen dient. In diesem Beispiel erfolgt der Aufruf von R mit den Optionen `--sdi` und `--max-mem-size=1536M`. Der Anfang der Pfadangabe ist aus Platzgründen nicht lesbar. Auch das Arbeitsverzeichnis (`d:\user\ligges`) ist hier explizit gesetzt worden.

Unter Windows gibt es mit 'Rconsole' auch noch eine weitere wichtige Konfigurationsdatei, die das Aussehen des R Konsolenfensters mitbestimmt. Eine globale Datei liegt im Verzeichnis `R_HOME\etc`, während nach einer eigenen Datei des aktuellen Benutzers in dem in der Umgebungsvariablen `R_USER` angegebenen Pfad gesucht wird. Falls `R_USER` nicht gesetzt ist, wird diese Variable auf den Wert der Umgebungsvariablen `HOME` gesetzt, falls `HOME` existiert. Sollten beide Umgebungsvariablen nicht gesetzt sein, so wird als Verzeichnis `HOMEDRIVE:\HOMEPATH` verwendet.

Abb. A.1. Windows Eigenschaftendialog — Setzen von Kommandozeilenoptionen und Arbeitsverzeichnis

Die R GUI unter Windows besitzt im Menü `Bearbeiten` – `GUI Einstellungen ...` (in der englischen Version: `Edit` – `GUI preferences ...`) ein Dialogfeld, mit dem die Einstellungen in der Datei 'Rconsole' komfortabel konfiguriert werden können.

B

Editoren für R

In der Kommandozeile von R kann man zwar sehr bequem einfache Berechnungen durchführen, das Erstellen von Funktionen und längeren Aufrufen fällt jedoch schwer. Es empfiehlt sich daher, einen Editor zu verwenden. Eine Funktion oder eine beliebige andere Ansammlung von Code kann dann in einer Textdatei auf der Festplatte gespeichert werden und mit `source("Dateiname")` geladen und ausgeführt werden. Einfachere Funktionen oder Programmteile können auch mittels *Copy&Paste* übertragen werden.

Drei Editoren mit zugehörigen Erweiterungen für das komfortable Programmieren mit R werden hier vorgestellt: Der Emacs mit ESS, Tinn-R sowie WinEdt mit **RWinEdt** (letzte beiden nur unter Windows). Die Seite `http://www.R-project.org/GUI/` bietet eine Liste weiterer Editoren, die an R angepasst sind.

B.1 Der Emacs mit ESS

Es gibt die Erweiterung ESS (Emacs Speaks Statistics, Rossini et al., 2004) für den unter nahezu allen Betriebssystemen (z.B. Linux, MacOS X, Windows) laufenden, mächtigen und freien Editor Emacs[1] (Stallmann, 1999) bzw. XEmacs[2] (im Folgenden beides als Emacs bezeichnet). Emacs ist als umfassender Editor (nahezu ein eigenes Betriebssystem) bekannt, z.B. für das Schreiben von Dokumenten in LaTeX.

Mit ESS können vom Emacs aus Statistik-Programme (z.B. Lisp-Stat, R, SAS, S-PLUS und Stata) komfortabel gesteuert werden. An der ESS Entwicklung sind auch Mitglieder des R Development Core Teams beteiligt. Falls ESS noch nicht in eine vorhandene Emacs Installation integriert sein sollte, kann

[1] `http://www.gnu.org/software/emacs/emacs.html`
[2] `http://www.xemacs.org/`

diese Erweiterung z.B. im XEmacs mit Hilfe des Menüs über das Internet installiert werden.

Zu den besonders komfortablen Eigenschaften von ESS gehört, dass R innerhalb eines ESS Buffers laufen kann (mit nur minimalen Problemen für R unter Windows). Des Weiteren können markierte Passagen, Zeilen oder Blöcke von Code direkt an R geschickt werden. Die R Ausgabe kann auch direkt weiterverarbeitet werden. Außerdem bietet ESS s.g. *Syntax-Highlighting*. Damit wird Code entsprechend seiner Funktionalität (Zuweisungen, Vergleiche, Kommentare, Funktionen, Konstrukte usw.) automatisch farblich gekennzeichnet oder durch geänderten Schriftstil hervorgehoben. Das ist ein erheblicher Beitrag zur Übersichtlichkeit beim Programmieren.

R wird im Emacs gestartet durch die Tastenkombination:

```
ESC-x, Shift-R, Enter
```

Da einige Windows Benutzer den Emacs wegen der vielen zu merkenden Tastenkürzel (*Shortcuts*) nur sehr ungern benutzen, hat John Fox unter `http://socserv.mcmaster.ca/jfox/Books/Companion/ESS/` neben einer Anleitung zur Installation auch Konfigurationsdateien bereitgestellt, die das Arbeiten mit Emacs und ESS Windows-ähnlicher gestalten.

B.2 Tinn-R

Eine Entwicklungsumgebung unter Windows bietet der Editor Tinn-R (Tinn-R Development Team, 2006). Er unterstützt neben *Syntax-Highlighting* und Kommunikationsmöglichkeiten mit R auch das Erzeugen von SWeave Dokumenten (s. Abschn. 10.7.2) und bietet eine Reihe weiterer nützlicher Funktionalität, die ständig von den sehr aktiven Entwicklern erweitert wird.

Für Windows Benutzer, die bisher weder Emacs noch WinEdt benutzt haben, ist dieser freie (Open Source unter der GPL) Editor sehr empfehlenswert.

B.3 WinEdt mit RWinEdt

Für Windows gibt es u.a. den kommerziellen Editor WinEdt[3] (Alexander, 1999, Shareware), der vor allem LaTeX–Benutzern bekannt sein dürfte. Die durch das CRAN Paket **RWinEdt** von Ligges (2003c) für diesen Editor bereitgestellte Erweiterung (Plug-In) ermöglicht unter anderem *Syntax-Highlighting* und Kommunikationsmöglichkeiten mit R über Tastenkürzel, Menüs und die

[3] `http://www.WinEdt.com`

„Klickwelt“. Die Bedienung geschieht in einer unter Windows gebräuchlichen Art und Weise.

RWinEdt ist allerdings lange nicht so mächtig wie ESS, denn es fehlt z.B. noch die Unterstützung des SWeave Formats (s. Abschn. 10.7.2). Außerdem ist die Schnittstelle zwischen R und WinEdt zwar funktionell, aber nicht professionell.

Für das sehr vollständige *Syntax-Highlighting* werden die folgenden Klassen von syntaktischen Elementen per Schriftstil und/oder Farbe unterschieden:

- Kommentare: `#` und alles was folgt
- Zuweisungen: `<-`, `<<-`, `=`, `->`, `->>` und `;`
- Kontrollwörter für Konstrukte: `if`, `else`, `ifelse`, `for`, `repeat`, `while`, `break` und `next`
- Logische Operatoren: `==`, `<=`, `>=`, `!=`, `!`, `<`, `>`, `&`, `&&`, `|` und `||`
- Namespace Operatoren: `::` und `:::`
- Weitere Kontrollwörter: `function`, `library` und `require`
- Spezielle Werte: `NULL`, `NA`, `NaN`, `Inf`, `F`, `T`, `FALSE`, `TRUE` und die Funktionen `.Internal` und `.C`
- Matrix Operatoren: `%%`, `%*%`, `%/%`, `%in%`, `%o%` und `%x%`
- Zeichenketten zwischen einfachen und doppelten Anführungszeichen (s. auch Abschn. 2.11, S. 56)
- Nicht reguläre Namen zwischen ‘Backticks‘ (s. S. 56)
- Index-Operatoren: `$`, `@`
- Verschiedene Arten von Klammern: `()`, `[]`, `{}`
- Eine Liste aller bekannten R Funktionen in den Basis-Paketen und den empfohlenen Paketen, die mit R geliefert werden
- ... usw.

Durch Abbildung B.1 kann wegen der nicht druckbaren Farben nur ein oberflächlicher Eindruck einer laufenden Sitzung vermittelt werden.

Mit welchen Tastenkürzeln welche Aktionen innerhalb von **RWinEdt** ausgeführt werden können, kann anhand der Hilfestellungen im eigenen R Menü ersehen werden. Die wichtigsten Kürzel sind:

- `Alt+l`: Sendet die aktuelle Zeile
- `Alt+p`: Sendet den aktuell selektierten Bereich
- `Alt+s`: Speichert aktuelle Datei und lädt sie mit `source()` in R ein

Ein „Command Completion Wizard“ für WinEdt ist erhältlich[4] und kann einfach für R angepasst werden. Er ist wegen unbekannter Lizenzbestimmungen aber nicht direkt in **RWinEdt** enthalten.

[4] von Holger Danielsson, `http://www.WinEdt.org/Plugins/complete.php`

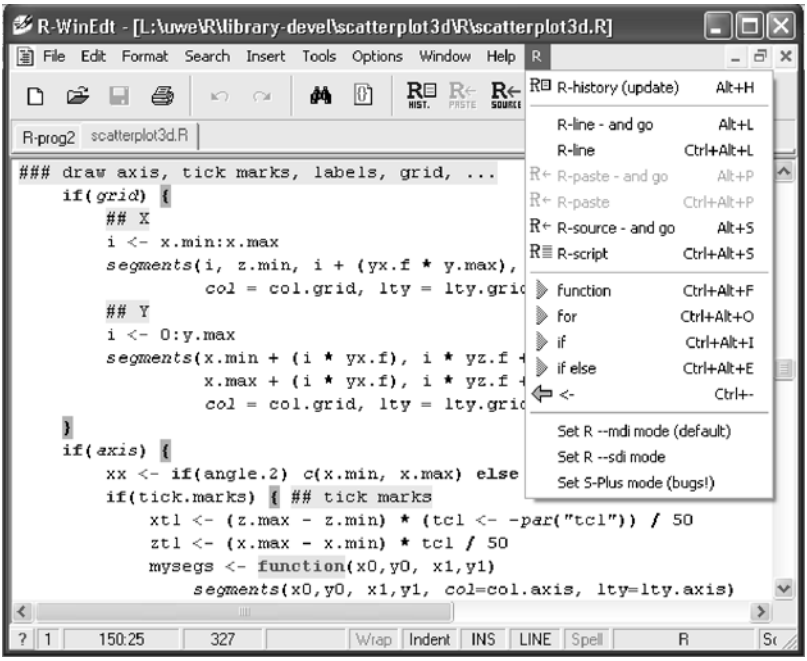

Abb. B.1. Laufende Sitzung: WinEdt mit **RWinEdt**

C

Grafische Benutzeroberflächen (GUI) für R

Eine „vollständige“ grafische Benutzeroberfläche (Graphical User Interface, kurz GUI), von der aus alle möglichen Funktionen über Dialogboxen oder Menüs gesteuert werden könnten, ist in R nicht vorhanden. Nicht nur wegen der Portabilität von R wäre die Entwicklung einer solchen vollständigen GUI sehr aufwändig.

Werkzeuge zur GUI Entwicklung stehen hingegen zur Verfügung, etwa das bereits in der Standard R Distribution enthaltene Paket **tcltk** (Dalgaard, 2001b,a, 2002a) oder das CRAN Paket **RGtk2** (Temple Lang und Lawrence, 2006) zur Erzeugung von Gtk basierter GUI. Der auf **tcltk** basierende R *Commander* wird in Anhang C.1 vorgestellt.

Eine Übersicht über verschiedene Ansätze zu grafischen Benutzeroberflächen und dafür gedachte Entwicklerpakete ist unter der URL `http://www.R-project.org/GUI/` des SciViews Projekts (Grosjean, 2003) zu finden.

Ein Bestandteil des SciViews Projekts ist die *SciViews R Console*, eine GUI, die auf der Windows GUI von R aufsetzt. Sie stellt einen Objekt-Browser, Menüs und einen Skript-Editor bereit. Text- und Grafikausgabe können aus der Konsole in einen integrierten HTML Editor übernommen werden.

JGR[1] (*J*ava *Gui* for *R*; in englischer Aussprache: „Jaguar“) ist eine für verschiedenste Betriebssysteme geeignete und vereinheitlichte Entwicklungsumgebung für R, die auf Java basiert. Dieser viel versprechende Ansatz wurde zuerst im Mai 2004 von den Autoren Markus Helbig, Simon Urbanek and Martin Theus vorgestellt. Unter anderem enthält *JGR*

- einen Editor mit Syntax Highlighting und automatischer Vervollständigung von Funktionsnamen,
- ein integriertes Hilfesystem,
- ein Spreadsheet für Datenansicht und Dateneingabe,

[1] `http://www.rosuda.org/JGR/`

- eine R Konsole, die automatische Vervollständigung von Funktionsnamen unterstützt und Hilfe zur Syntax der Funktion anzeigt
- und einen Objekt-Browser, der nach verschiedenen Objekttypen sortieren kann.

C.1 Der R Commander

Der auf **tcltk** basierende R *Commander* ist eine von Fox (2005) entwickelte GUI, die als Paket **Rcmdr** auf CRAN erhältlich ist. Details gibt es unter der URL `http://socserv.mcmaster.ca/jfox/Misc/Rcmdr/`.

Weil das Paket **tcltk** benutzt wird, ist eine gewisse Plattformunabhängigkeit des **Rcmdr** Pakets gewährleistet. Das Aussehen dieser GUI (s. Abb. C.1) ist durch die **tcltk** Benutzung vorgegeben und hält sich daher nicht unbedingt an die Standards des jeweiligen Betriebssystems.

Neben Zugriff auf Import- und Export-Funktionen sowie Funktionen für Datentransformationen bietet das Paket auch Menüs, die Dialogboxen für die Ausführung elementarer statistischer Verfahren erzeugen. Für Anfänger und für den Einsatz in der Lehre ist das Paket somit sehr gut geeignet. Dazu trägt insbesondere die hervorragende Eigenschaft bei, dass die mit der Maus „geklickten“ Aktionen als Code in einem Fenster erscheinen. Man lernt durch das „Klicken“, wie die gewünschte Aktion durch Aufruf einer Funktion in der Befehlszeile ausgeführt werden kann, so dass sich die GUI möglichst bald selbst überflüssig macht.

C.2 Windows GUI

Die Windows GUI von R hat Menüs für einige elementare Funktionen. Dazu gehören das Laden und Speichern des Workspace, die automatische Installation (inkl. Updates) von Paketen, Zugriff auf Handbücher und Hilfefunktionen sowie ein sehr rudimentärer Dateneditor. Das alles kann auch durch den Aufruf von Funktionen in der Konsole erreicht werden.

Wer Dialoge und Menüs basierend auf der Windows GUI erstellen möchte, kann für sehr einfache Erweiterung der Menüs die Funktionen `winMenuAdd()` und `winMenuAddItem()` verwenden. Für die Erstellung sehr limitierter Dialogboxen sind die Funktionen `winDialog()` und `winDialogString()` verwendbar. Komplexere Dialogboxen können auch erstellt werden. Als Beispiel findet man in den R Quellen im Verzeichnis '.../src/gnuwin32/windlgs' das Paket **windlgs**.

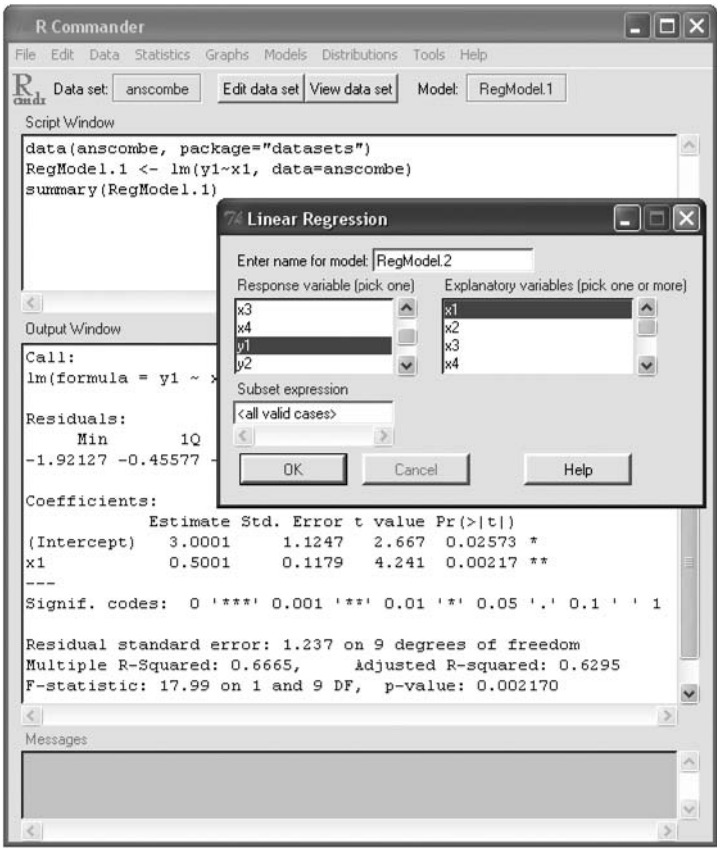

Abb. C.1. Der R Commander

D

Tabelle englischer und deutscher Begriffe

Tabelle D.1. Tabelle englischer und deutscher Begriffe

englischer Begriff	deutscher Begriff bzw. Beschreibung (Verweis: Abschn., Kap.)
Bug	Fehler in einem Programm.
Call	Ein einfacher oder geschachtelter Aufruf von Funktionen. Eine Sequenz von calls nennt man auch expression (s.u.).
Debug	Suchen, Finden und Beheben von Fehlern.
Environment	Eine Umgebung, in der Objekte unabhängig von anderen Umgebungen existieren können, wird z.B. durch Funktionsaufrufe erzeugt (4.3).
Expression	Ein Ausdruck, der aus mehreren Aufrufen bestehen kann.
GUI	Grafische Benutzeroberfläche (Anhang C).
High-level Grafik	Funktionen, die eine vollständige Grafik erzeugen (8).
Lazy Evaluation	Verzögerte Auswertung von Argumenten (4.2).
Lazy Loading	Objekte aus Paketen erst dann in den Hauptspeicher laden, wenn sie benötigt werden.
Lexical Scoping	Besondere Regeln, nach denen R Objekte sucht (4.3).
Library	Bibliothek. Man unterscheide zwischen a) Binärdateien, die schnellen Maschinencode enthalten (z.B. eine *dynamic link library* (DLL)), b) Verzeichnisse, die R Pakete enthalten (10.3.1).
Low-level Grafik	(Hilfs-)Funktionen, die Elemente zu einer Grafik hinzufügen (8).
Namespace	Ein „Raum“ für Objektnamen, der eine gewisse Unabhängigkeit und Flexibilität bei der Benennung von Funktionen bietet (4.3, 10.6).

englischer Begriff	deutscher Begriff bzw. Beschreibung (Verweis: Abschn., Kap.)
Package	Paket, das zusätzliche Funktionen, Daten und zugehörige Dokumentation bieten kann (10).
Scoping Rules	Die Regeln, nach denen Objekte gesucht werden (4.3).
Shell	Kommandointerpreter des Betriebssystems
Sources	Quellcode, Quellen
Workspace	Der Arbeitsplatz, in dem man seine Objekte ablegt (2.6).

Literaturverzeichnis

Adler, D., Nenadic, O., und Zucchini, W. (2003): RGL: A R-library for 3D visualization with OpenGL. In: *Proceedings of the 35th Symposium of the Interface: Computing Science and Statistics*. Salt Lake City. (178)

Advanced Micro Devices (2006): *AMD Core Math Library (ACML)*. Advanced Micro Devices, Inc., Numerical Algorithms Group Ltd. URL `http://developer.amd.com/acml.aspx`. (208)

Alexander, J. (1999): *The WinEdt Hacker's Guide*. URL `http://www.WinEdt.org/`. (216)

Anderson, E. (1935): The irises of the Gaspe Peninsula. *Bulletin of the American Iris Society*, 59, 2–5. (20, 145, 174)

Anscombe, F. (1973): Graphs in statistical analysis. *American Statistician*, 27, 17–21. (106, 116, 137)

Baier, T. (2003): R: Windows Component Services Integrating R and Excel on the COM Layer. In: *Hornik et al. (2003)*. (68)

Baier, T. und Neuwirth, E. (2003): High-Level Interface between R and Excel. In: *Hornik et al. (2003)*. (68)

Bates, D. (2003): Converting Packages to S4. *R News*, 3 (1), 6–8. ISSN 1609-3631. URL `http://CRAN.R-project.org/doc/Rnews/`. (120)

Bates, D. und DebRoy, S. (2003): Converting a Large R Package to S4 Classes and Methods. In: *Hornik et al. (2003)*. (120)

Bates, D. und Sarkar, D. (2006): *lme4: Linear mixed-effects models using S4 classes*. R package version 0.995-2. (135)

Beall, G. (1942): The Transformation of data from entomological field experiments. *Biometrika*, 29, 243–262. (142)

Becker, R. (1994): A Brief History of S. In: P. Dirschedl und R. Ostermann (Hrsg.) *Computational Statistics: Papers Collected on the Occasion of the 25th Conference on Statistical Computing at Schloß Reisenburg*, 81–110. Heidelberg: Physika Verlag. (3)

Becker, R. und Chambers, J. (1984): *S. An Interactive Environment for Data Analysis and Graphics*. Monterey: Wadsworth and Brooks/Cole. (3, 149)

Becker, R., Chambers, J., und Wilks, A. (1988): *The NEW S Language — a Programming Environment for Data Analysis and Graphics*. New York: Chapman & Hall. (3)

Becker, R., Cleveland, W., und Shyu, M. (1996): The Visual Design and Control of Trellis Display. *Journal of Computational and Graphical Statistics*, 5 (2), 123–155. (168)

Breiman, L. (2001): Random Forests. *Machine Learning*, 45 (1), 5–32. (145)

Breiman, L., Friedman, J., Olshen, R., und Stone, C. (1984): *Classification and Regression Trees*. Belmont, CA: Wadsworth. (144)

Bronstein, I., Semendjajew, K., Musiol, G., und Mühlig, H. (2000): *Taschenbuch der Mathematik*. Frankfurt am Main: Verlag Harri Deutsch, 5. Auflage. (89)

Burns, P. (1998): *S Poetry*. Burns Statistics. URL `http://www.burns-stat.com/pages/spoetry.html`. (18)

Chambers, J. (1998): *Programming with Data. A Guide to the S Language*. New York: Springer-Verlag. (3, 4, 5, 11, 12, 115, 120, 122, 191)

Chambers, J. und Hastie, T. (1992): *Statistical Models in S*. New York: Chapman & Hall. (3, 134, 135)

Chang, C.-C. und Lin, C.-J. (2001): *LIBSVM: a library for Support Vector Machines*. URL `http://www.csie.ntu.edu.tw/~cjlin/libsvm`. (144)

Cleveland, W. (1993): *Visualizing Data*. Summit, NJ: Hobart Press. (149, 168)

Dalgaard, P. (2001a): A Primer on the R-Tcl/Tk Package. *R News*, 1 (3), 27–31. URL `http://CRAN.R-project.org/doc/Rnews/`. (219)

Dalgaard, P. (2001b): The R-Tcl/Tk interface. In: *Hornik und Leisch (2001)*. (219)

Dalgaard, P. (2002a): Changes to the R-Tcl/Tk package. *R News*, 2 (3), 25–27. URL `http://CRAN.R-project.org/doc/Rnews/`. (219)

Dalgaard, P. (2002b): *Introductory Statistics with R*. New York: Springer-Verlag. (18)

Everitt, B. und Hothorn, T. (2006): *A Handbook of Statistical Analysis Using R*. Boca Raton, FL: Chapman & Hall/CRC. (18, 127)

Fisher, R. (1936): The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7, 179–188. (20)

Fox, J. (1997): *Applied Regression Analysis, Linear Models, and Related Methods*. Thousand Oaks: Sage. (18, 135)

Fox, J. (2002): *An R and S-PLUS Companion to Applied Regression*. Thousand Oaks: Sage. (18, 135)

Fox, J. (2005): The R Commander: A basic-statistics graphical user interface to R. *Journal of Statistical Software*, 14 (9), 1–42. URL `http://www.jstatsoft.org/`. (220)

Friedman, J. (1989): Regularized Discriminant Analysis. *Journal of the American Statistical Association*, 84, 165–175. (144)

Garczarek, U. (2002): *Classification Rules in Standardized Partition Spaces*. Dissertation, Fachbereich Statistik, Universität Dortmund, Dortmund, Germany. URL `http://hdl.handle.net/2003/2789`. (144)

Gentleman, R., Carey, V., Bates, D., Bolstad, B., Dettling, M., Dudoit, S., Ellis, B., Gautier, L., Ge, Y., Gentry, J., Hornik, K., Hothorn, T., Huber, W., Iacus, S., Irizarry, R., Leisch, F., Li, C., Maechler, M., Rossini, A., Sawitzki, G., Smith, C., Smyth, G., Tierney, L., Yang, J., und Zhang, J. (2004): Bioconductor: Open software development for computational biology and bioinformatics. *Genome Biology*, 5, R80. URL `http://genomebiology.com/2004/5/10/R80`. (5, 192)

Gentleman, R., Carey, V., Huber, W., Irizarry, R., und Dudoit, S. (Hrsg.) (2005): *Bioinformatics and Computational Biology Solutions Using R and Bioconductor*. New York: Springer-Verlag. (192)

Gentleman, R. und Ihaka, R. (2000): Lexical Scope and Statistical Computing. *Journal of Computational and Graphical Statistics*, 9 (3), 491–508. (76)

Grosjean, P. (2003): SciViews: An Object-Oriented Abstraction Layer to Design GUIs on Top of Various Calculation Kernels. In: *Hornik et al. (2003)*. (219)

Grothendieck, G. und Petzoldt, T. (2004): R Help Desk – Date and Time Classes in R. *R News*, 4 (1), 29–32. URL `http://CRAN.R-project.org/doc/Rnews/`. (57, 58)

Hartung, J., Elpelt, B., und Klösener, K.-H. (2005): *Statistik: Lehr- und Handbuch der angewandten Statistik*. München: R. Oldenbourg Verlag, 14. Auflage. (127)

Hastie, T., Tibshirani, R., und Friedman, J. (2001): *The Elements of Statistical Learning. Data Mining Inference and Prediction*. New York: Springer-Verlag. (144)

Hornik, K. (2006): *The R FAQ*. R Foundation for Statistical Computing, Vienna, Austria. URL `http://cran.r-project.org/doc/FAQ`. ISBN 3-900051-08-9. (17, 193, 209)

Hornik, K. und Leisch, F. (Hrsg.) (2001): *Proceedings of the 2nd International Workshop on Distributed Statistical Computing, March 15–17*. Vienna: Technische Universität Wien. ISSN 1609-395X. URL `http://www.ci.tuwien.ac.at/Conferences/DSC-2001/Proceedings/`. (226, 227, 229, 230, 231)

Hornik, K. und Leisch, F. (2002): Vienna and R: Love, marriage and the future. In: R. Dutter (Hrsg.) *Festschrift 50 Jahre Österreichische Statistische Gesellschaft*, 61–70. Österreichische Statistische Gesellschaft. ISSN 1026-597X. (4)

Hornik, K., Leisch, F., und Zeileis, A. (Hrsg.) (2003): *Proceedings of the 3rd International Workshop on Distributed Statistical Computing, March 20–22*. Vienna: Technische Universität Wien. ISSN 1609-395X. URL `http://www.ci.tuwien.ac.at/Conferences/DSC-2003/Proceedings/`. (225, 227, 228, 231)

Hothorn, T. (2001): On Exact Rank Tests in R. *R News*, 1 (1), 11–12. URL `http://CRAN.R-project.org/doc/Rnews/`. (143)

Hothorn, T., Bretz, F., und Genz, A. (2001a): On Multivariate t and Gauß Probabilities in R. *R News*, 1 (2), 27–29. URL `http://CRAN.R-project.org/doc/Rnews/`. (133)

Hothorn, T., James, D., und Ripley, B. (2001b): R/S Interfaces to Databases. In: *Hornik und Leisch (2001)*. (65)

Iacus, S., Urbanek, S., und Goedman, R. (2006): *R for Mac OS X FAQ*. URL `http://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html`. (17, 193, 209)

Ihaka, R. und Gentleman, R. (1996): R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5 (3), 299–314. (1, 4, 149)

Insightful Corporation (2006): *S-PLUS 7*. Insightful Corporation, Seattle, WA, USA. URL `http://www.insightful.com`. (3)

James, D. und Pregibon, D. (1993): Chronological Objects for Data Analysis. In: *Proceedings of the 25th Symposium of the Interface*. San Diego. URL `http://cm.bell-labs.com/cm/ms/departments/sia/dj/papers/chron.pdf`. (57)

Karatzoglou, A., Smola, A., Hornik, K., und Zeileis, A. (2004): kernlab — an S4 package for kernel methods in R. *Journal of Statistical Software*, 11 (9), 1–20. URL `http://www.jstatsoft.org/`. (144)

Lange, K. (1999): *Numerical Analysis for Statisticians*. New York: Springer-Verlag. (94, 131, 147)

Leisch, F. (2002a): Sweave: Dynamic Generation of Statistical Reports Using Literate Data Analysis. In: W. Härdle und B. Rönz (Hrsg.) *Compstat 2002 — Proceedings in Computational Statistics*, 575–580. Heidelberg, Germany: Physika Verlag. (206)

Leisch, F. (2002b): Sweave, Part I: Mixing R and LaTeX. *R News*, 2 (3), 28–31. URL `http://CRAN.R-project.org/doc/Rnews/`. (206)

Leisch, F. (2003a): Sweave and Beyond: Computations on Text Documents. In: *Hornik et al. (2003)*. (206)

Leisch, F. (2003b): Sweave, Part II: Package Vignettes. *R News*, 3 (2), 21–24. URL `http://CRAN.R-project.org/doc/Rnews/`. (206)

Leisch, F. (2006): *Sweave User Manual*. Institut für Statistik und Wahrscheinlichkeitstheorie, Technische Universität Wien, Vienna, Austria. URL `http://www.ci.tuwien.ac.at/~leisch/Sweave`. R Version 2.1.0. (206)

Li, M. und Rossini, A. (2001): RPVM: Cluster Statistical Computing in R. *R News*, 1 (3), 4–7. URL `http://CRAN.R-project.org/doc/Rnews/`. (98)

Liaw, A. und Wiener, M. (2002): Classification and Regression by randomForest. *R News*, 2 (3), 18–22. URL `http://CRAN.R-project.org/doc/Rnews/`. (145)

Ligges, U. (2002): R Help Desk: Automation of Mathematical Annotation in Plots. *R News*, 2 (3), 32–34. ISSN 1609-3631. URL `http://CRAN.R-project.org/doc/Rnews/`. (165, 166)

Ligges, U. (2003a): R Help Desk: Getting Help – R's Help Facilities and Manuals. *R News*, 3 (1), 26–28. ISSN 1609-3631. URL `http://CRAN.R-project.org/doc/Rnews/`. (14)

Ligges, U. (2003b): R Help Desk: Package Management. *R News*, 3 (3), 37–39. ISSN 1609-3631. URL `http://CRAN.R-project.org/doc/Rnews/`. (192)

Ligges, U. (2003c): R-WinEdt. In: *Hornik et al. (2003)*. (216)

Ligges, U. und Mächler, M. (2003): Scatterplot3d – an R Package for Visualizing Multivariate Data. *Journal of Statistical Software*, 8 (11), 1–20. URL `http://www.jstatsoft.org/`. (81, 156, 167)

Ligges, U. und Murdoch, D. (2005): R Help Desk: Make 'R CMD' Work under Windows – an Example. *R News*, 5 (2), 27–28. ISSN 1609-3631. URL `http://CRAN.R-project.org/doc/Rnews/`. (184, 198, 210)

Lindsey, J. (2001): *Nonlinear Models in Medical Statistics*. Oxford: Oxford University Press. (192)

Maindonald, J. (2004): *Using R for Data Analysis and Graphics. Introduction, Code and Commentary*. Australian National University. URL `http://wwwmaths.anu.edu.au/~johnm/`. (17, 18)

Maindonald, J. und Braun, J. (2003): *Data Analysis and Graphics Using R: An Example-Based Approach*. Cambridge: Cambridge University Press. (18)

Matsumoto, M. und Nishimura, T. (1998): Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8 (1), 3–30. (131)

Murdoch, D. (2001): RGL: An R Interface to OpenGL. In: *Hornik und Leisch (2001)*. (178)

Murrell, P. (1999): Layouts: A mechanism for arranging plots on a page. *Journal of Computational and Graphical Statistics*, 8, 121–134. (163)

Murrell, P. (2001): R Lattice Graphics. In: *Hornik und Leisch (2001)*. (169)

Murrell, P. (2002): The grid Graphics Package. *R News*, 2 (2), 14–19. ISSN 1609-3631. URL `http://CRAN.R-project.org/doc/Rnews/`. (163, 169)

Murrell, P. (2003): Integrating grid Graphics Output with Base Graphics Output. *R News*, 3 (2), 7–12. URL `http://CRAN.R-project.org/doc/Rnews/`. (163)

Murrell, P. (2004): Fonts, Lines, and Transparency in R Graphics. *R News*, 4 (2), 5–9. ISSN 1609-3631. URL `http://CRAN.R-project.org/doc/Rnews/`. (152, 158)

Murrell, P. (2005): *R Graphics*. Boca Raton, FL: Chapman & Hall/CRC. (18, 149)

Murrell, P. und Ihaka, R. (2000): An Approach to Providing Mathematical Annotation in Plots. *Journal of Computational and Graphical Statistics*, 9 (3), 582–599. (165)

Murrell, P. und Ripley, B. (2006): Non-Standard Fonts in PostScript and PDF Graphics. *R News*, 6 (2), 41–46. ISSN 1609-3631. URL `http://CRAN.R-project.org/doc/Rnews/`. (152)

Peters, A., Hothorn, T., und Lausen, B. (2002): ipred: Improved Predictors. *R News*, 2 (2), 33–36. URL `http://CRAN.R-project.org/doc/Rnews/`. (144)

Pinheiro, J. und Bates, D. (2000): *Mixed-Effects Models in S and S-PLUS*. New York: Springer-Verlag. (18, 135)

R Development Core Team (2006a): *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL `http://www.R-project.org`. ISBN 3-900051-07-0. (3, 11, 17, 127)

R Development Core Team (2006b): *R Data Import/Export*. R Foundation for Statistical Computing, Vienna, Austria. URL `http://www.R-project.org`. ISBN 3-900051-10-0. (16, 59)

R Development Core Team (2006c): *R Installation and Administration*. R Foundation for Statistical Computing, Vienna, Austria. URL `http://www.R-project.org`. ISBN 3-900051-09-7. (16, 184, 193, 197, 198, 207, 208, 209, 210, 211)

R Development Core Team (2006d): *R Language Definition*. R Foundation for Statistical Computing, Vienna, Austria. URL `http://www.R-project.org`. ISBN 3-900051-13-5. (17, 79)

R Development Core Team (2006e): *Writing R Extensions*. R Foundation for Statistical Computing, Vienna, Austria. URL `http://www.R-project.org`. ISBN 3-900051-11-9. (17, 182, 184, 199, 201, 202, 203, 205, 206)

Ripley, B. (1996): *Pattern Recognition and Neural Networks*. Cambridge, UK: Cambridge University Press. (144)

Ripley, B. (2001a): Connections. *R News*, 1 (1), 16–17. URL `http://CRAN.R-project.org/doc/Rnews/`. (64)

Ripley, B. (2001b): Using Databases with R. *R News*, 1 (1), 18–20. URL `http://CRAN.R-project.org/doc/Rnews/`. (65, 97)

Ripley, B. (2004): Lazy Loading and Packages in R 2.0.0. *R News*, 4 (2), 2–4. ISSN 1609-3631. URL `http://CRAN.R-project.org/doc/Rnews/`. (202)

Ripley, B. (2005a): Internationalization Features of R 2.1.0. *R News*, 5 (1), 2–7. ISSN 1609-3631. URL `http://CRAN.R-project.org/doc/Rnews/`. (210, 211)

Ripley, B. (2005b): Packages and their Management in R 2.1.0. *R News*, 5 (1), 8–11. ISSN 1609-3631. URL `http://CRAN.R-project.org/doc/Rnews/`. (192)

Ripley, B. und Hornik, K. (2001): Date-Time Classes. *R News*, 1 (2), 8–11. URL `http://CRAN.R-project.org/doc/Rnews/`. (57)

Ripley, B. und Murdoch, D. (2006): *R for Windows FAQ*. URL `http://cran.r-project.org/bin/windows/rw-FAQ.html`. (17, 193)

Rossini, A. (2001): Literate Statistical Analysis. In: *Hornik und Leisch (2001)*. (205)

Rossini, A., Heiberger, R., Sparapani, R., Mächler, M., und Hornik, K. (2004): Emacs Speaks Statistics: A Multiplatform, Multipackage Development Environment for Statistical Analysis. *Journal of Computational and Graphical Statistics*, 13 (1), 247–261. (215)

Sarkar, D. (2002): Lattice: An Implementation of Trellis Graphics in R. *R News*, 2 (2), 19–23. ISSN 1609-3631. URL `http://CRAN.R-project.org/doc/Rnews/`. (168)

SAS Institute Inc. (1999): *SAS Language Reference*. Cary, NC. URL `http://www.sas.com/`. (60)

Sawitzki, G. (2005): *Statistical Computing: Einführung in R*. StatLab Heidelberg, Universität Heidelberg. URL `http://www.statlab.uni-heidelberg.de/projects/s/`. (18)

SPSS Inc. (2005): *SPSS Base 14.0 User's Guide*. Chicago, IL. URL `http://www.spss.com`. (62)

Stallmann, R. (1999): *The Emacs Editor*. Boston. URL `http://www.gnu.org`. Version 20.7. (215)

Sturtz, S., Ligges, U., und Gelman, A. (2005): R2WinBUGS: A Package for Running WinBUGS from R. *Journal of Statistical Software*, 12 (3), 1–16. URL `http://www.jstatsoft.org/`. (185)

Swayne, D., Buja, A., und Hubbell, N. (1991): XGobi meets S: Integrating Software for Data Analysis. In: *Computing Science and Statistics: Proceedings of the 23rd*

Symposium on the Interface, 430–434. Fairfax Station, VA: Interface Foundation of North America, Inc. (177)

Swayne, D., Cook, D., und Buja, A. (1998): XGobi: Interactive Dynamic Graphics in the X Window System. *Journal of Computational and Graphical Statistics*, 7 (1), 113–130. URL `http://www.research.att.com/areas/stat/xgobi/`. (177)

Swayne, D., Temple Lang, D., Buja, A., und Cook, D. (2003): GGobi: Evolving from XGobi into an Extensible Framework for Interactive Data Visualization. *Journal of Computational and Graphical Statistics*, 43 (4), 423–444. (178)

Temple Lang, D. (2000): The Omegahat Environment: New Possibilities for Statistical Computing. *Journal of Computational and Graphical Statistics*, 9 (3), 423–451. (185, 189, 192)

Temple Lang, D. und Lawrence, M. (2006): *RGtk2: R bindings for Gtk 2.0*. URL `http://www.ggobi.org/rgtk2,http://www.omegahat.org`. R package version 2.8.5. (219)

Temple Lang, D. und Swayne, D. (2001): GGobi meets R: An Extensible Environment for Interactive Dynamic Data Visualization. In: *Hornik und Leisch (2001)*. (178)

Thomas, A. (2004): *BRugs User Manual, Version 1.0*. Dept of Mathematics & Statistics, University of Helsinki. (185)

Thomas, A., O'Hara, B., Ligges, U., und Sturtz, S. (2006): Making BUGS Open. *R News*, 6 (1). ISSN 1609-3631. URL `http://CRAN.R-project.org/doc/Rnews/`. (185)

Tierney, L. (2003): Name Space Management for R. *R News*, 3 (1), 2–6. ISSN 1609-3631. URL `http://CRAN.R-project.org/doc/Rnews/`. (81, 83)

Tinn-R Development Team (2006): *Tinn-R: A editor for R language and environment statistical computing*. URL `http://sourceforge.net/projects/tinn-r, http://www.sciviews.org/Tinn-R/`. (216)

Urbanek, S. (2006): *rJava: Low-level R to Java interface*. URL `http://www.rosuda.org/software/rJava/`. R package version 0.4-3. (185)

Urbanek, S. und Theus, M. (2003): iPlots – High Interaction Graphics for R. In: *Hornik et al. (2003)*. (178)

Urbanek, S. und Unwin, A. (2002): Making Trees Interactive with KLIMT – a COSADA Software Project. *Statistical Computing and Graphics Newsletter*, 13 (1), 13–16. (179)

Venables, W. (2001): Programmer's Niche. *R News*, 1 (1), 27–30. URL `http://CRAN.R-project.org/doc/Rnews/`. (112)

Venables, W. (2002): Programmer's Niche – Mind Your Language. *R News*, 2 (2), 24–26. URL `http://CRAN.R-project.org/doc/Rnews/`. (93)

Venables, W. und Ripley, B. (2000): *S Programming*. New York: Springer-Verlag. (18, 76, 79, 93, 120)

Venables, W. und Ripley, B. (2002): *Modern Applied Statistics with S*. New York: Springer-Verlag, 4. Auflage. (18, 127, 135, 143, 144, 191)

Venables, W. N., Smith, D. M., und the R Development Core Team (2006): *An Introduction to R*. R Foundation for Statistical Computing, Vienna, Austria. URL `http://www.R-project.org`. ISBN 3-900051-12-7. (16)

Yu, H. (2002): Rmpi: Parallel Statistical Computing in R. *R News*, 2 (2), 10–14. URL `http://CRAN.R-project.org/doc/Rnews/`. (98)

Zeileis, A. (2005): CRAN Task Viwes. *R News*, 5 (1), 39–40. ISSN 1609-3631. URL `http://CRAN.R-project.org/doc/Rnews/`. (192)

Tabellenverzeichnis

Index