RUHR
UNIVERSITÄT
BOCHUM

RUB

Einführung in R

Christian Dudel & Sebastian Jeworutzki

17. Januar 2011

Vorbemerkungen

Dieser Text ist eine anwendungs- und beispielorientierte Einführung in das Statistik-Programm R. Vorkenntnisse in der Bedienung und Programmierung statistischer Software werden nicht vorausgesetzt. Allerdings wird davon ausgegangen, dass der Leser mit den Grundlagen der Statistik vertraut ist.

Am einfachsten dürfte die Erarbeitung dieses Textes sein, wenn man parallel zum durchlesen die angeführten Beispiele selbst in R ausprobiert. Die Beispiele finden sich immer in grauen Boxen wie dieser hier:

```
Dies ist ein Beispiel
Dies ist die zweite Zeile des Beispiels
Und schließlich die dritte Zeile des Beispiels
```

In diesen Boxen sind sowohl die in R einzugebenden Befehle enthalten, als auch die Ergebnisse, die R liefert. Dies dient zum einen dazu, die eigenen Ergebnisse kontrollieren zu können, zugleich ist so aber auch ein Durcharbeiten dieses Textes ohne PC möglich.

Da R eine extrem umfangreiche und flexible Software ist, können in diesem Text nur einige wenige der Möglichkeiten dieses Programms vorgestellt werden und auch dies oftmals nur in abgekürzter Form. Insofern werden viele nützliche Befehle nicht vorgestellt und bei den hier vorgestellten wird wiederum auf einige Optionen verzichtet. Zudem gibt es in R für manche Problemstellungen mehrere Lösungsansätze, wobei sich hier zumeist für den aus unserer Sicht einfachsten, aber nicht unbedingt effizientesten Weg entschieden wurde. Nichtsdestotrotz sollten die hier vermittelten Grundlagen ausreichen, um die meisten Arbeitsschritte der computergestützten statistischen Analyse vornehmen zu können.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Was ist R?

R ist eine kostenlose Open-Source Software für statistische Datenverarbeitung, die über die Website http://www.r-project.org bezogen werden kann. Dabei umfasst R zum einen eine Vielzahl an Möglichkeiten zur Verarbeitung und Auswertung von Daten, die sich ohne großen Aufwand nutzen lassen. Zum anderen kann man statistische Verfahren auch selber programmieren und R hierüber fast beliebig erweitern. Von Anwendern erstellte Erweiterungen werden als Pakete oder *packages* bezeichnet und von ihren Programmierern oftmals über das Internet zugänglich gemacht. Im *Comprehensive R Archive Network* (kurz: CRAN), einem Netz aus Webservern, die Pakete und Code für R bereitstellen, sind über 2000 solcher Pakete gelistet. Daneben wird auch das „Grundgerüst" durch ein Kern-Team von Entwicklern ständig verbessert. Hierdurch ist R in vielen Bereichen immer auf dem neuesten Stand und oftmals sogar das erste Softwarepaket, das neu entwickelte Techniken und Verfahren enthält. Die aktuelle Version von R trägt die Nummer 2.11.1 und steht unter der sogenannten *GNU General Public License*, die eine freie, nicht-kommerzielle Verbreitung ermöglicht.

Die wesentlichen Vorteile von R lassen sich insgesamt wie folgt zusammenfassen:

- R kann kostenlos über das Internet bezogen werden.
- R steht für Windows-, Unix- und Mac-Systeme zur Verfügung.
- R wird von einem Kern-Team von Entwicklern ständig verbessert.
- Es gibt eine Vielzahl von frei zugänglichen Erweiterungen, die von der ständig wachsenden R-Community erstellt werden.
- R kann durch den Nutzer selbst erweitert werden.

Aufgrund dieser Vorteile findet R zunehmend Verbreitung und wird nicht nur im wissenschaftlichen Bereich, sondern auch für Anwendungen in der Wirtschaft eingesetzt, beispielsweise bei den Unternehmen Shell, Google oder Facebook[1], welche die R-Projektgruppe teilweise finanziell unterstützen. Selbst kommerzielle Statistik-Pakete wie SPSS oder SAS besitzen mittlerweile Möglichkeiten, R-Prozeduren und Grafiken einzubinden.

[1] http://dataspora.com/blog/predictive-analytics-using-r/

Ein Nachteil von R dürfte vor allem in der Bedienung liegen, die für viele Einsteiger wahrscheinlich ungewohnt ist und kompliziert wirkt. Dabei gibt es etliche Hilfsangebote, die den Start erleichtern. Eines davon ist dieses Skript. Daneben lassen sich im Internet diverse einführende Texte kostenlos beziehen. Beispielsweise findet sich auf der Website http://www.r-project.org unter dem Punkt „manuals“ eine gelungene Einführung (Venables et al., 2009). Zudem gibt es eine zunehmende Zahl an Lehrbüchern mit unterschiedlichen Zielgruppen, die die Grundlagen von R erläutern. Eine notgedrungen selektive Auswahl umfasst folgende Titel: Behr and Pötter (2010); Crawley (2007); Dalgaard (2002); Dolic (2003); Ligges (2009). Eine umfassendere Liste mit Lehr- und Fachbüchern, die sich mit R beschäftigen, ist unter http://www.r-project.org/doc/bib/R-books.html zu finden. Schließlich sei noch auf zwei wissenschaftliche Zeitschriften verwiesen: Das R-*Journal* ist eine begutachtete Zeitschrift, die über die Website http://journal.r-project.org frei zugänglich ist und regelmäßig neu entwickelte *packages* vorstellt, ebenso wie nützliche Tipps und Tricks und Hinweise auf Veranstaltungen, wie beispielsweise die jährlich stattfindende R-Nutzerkonferenz *useR!*. Ebenfalls als Open-Access-Zeitschrift frei zugänglich ist das *Journal of Statistical Software*, welches über http://www.jstatsoft.org/ abrufbar ist. Ein Großteil der begutachteten Aufsätze hat neue Pakete für R zum Thema.

1.2 Was findet sich in diesem Text?

In diesem Text werden vor allem die Grundlagen der Bedienung von R vorgestellt. Neben einer kurzen Anleitung zur Installation im folgenden Abschnitt findet sich zunächst eine Einführung in grundlegende Aspekte der Bedienung und wichtige Konzepte. Anschließend wird erläutert, wie Daten geladen und bearbeitet werden können. Darauf werden wichtige Befehle für den Bereich der deskriptiven Statistik eingeführt. Es folgt eine erste Einführung in die Erstellung von Grafiken. Darüber hinaus werden noch Inferenzstatistik, lineare Algebra und Grundlagen des Programmierens in R vorgestellt.

1.3 R installieren & starten

Um R zu installieren, muss zunächst die Website www.r-project.org aufgerufen werden. Dort klickt man unter dem Punkt „Download, Packages“ auf CRAN und wählt anschließend einen der angegebenen Server aus – idealerweise sollte es ein Server in Deutschland sein. Als nächstes muss das verwendete Betriebssystem ausgewählt werden. Wird „Windows“ verwendet, muss man auf der nächsten Seite noch das R-Basispaket durch anklicken von „base“ auswählen und schließlich auf dem folgenden Bildschirm nur noch „Download“ wählen. Die heruntergeladene Datei muss anschließend ausgeführt werden und das Setup-Programm führt durch die weiter notwendigen Schritte und Angaben.

Die wesentlichen Schritte der Installation sind immer die gleichen: CRAN Server-Liste aufrufen, Server wählen, Betriebssystem wählen und beispielsweise bei Windows das Basispaket

auswählen. Auf der Website des R-Projekts finden sich unter dem Punkt FAQ weitere Hinweise zur Installation, auch für Unix- und Mac-Systeme. Nach der Installation kann R wie die meisten anderen Programme auch gestartet werden – unter Windows in der Regel durch Doppelklick auf die entsprechende Verknüpfung, bei Unix-Systemen durch Eingabe von R im Shell.

Teil I

Grundlagen

2 Einführung in die Bedienung von R

2.1 R nach dem Start

Nachdem R installiert und gestartet wurde, sollte etwa folgende Ansicht auf dem Bildschirm erscheinen:

```
R version 2.9.0 (2009-04-17)
Copyright (C) 2009 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R ist freie Software und kommt OHNE JEGLICHE GARANTIE.
Sie sind eingeladen, es unter bestimmten Bedingungen weiter zu
verbreiten.
Tippen Sie 'license()' or 'licence()' für Details dazu.

R ist ein Gemeinschaftsprojekt mit vielen Beitragenden.
Tippen Sie 'contributors()' für mehr Information und 'citation()',
um zu erfahren, wie R oder R packages in Publikationen zitiert
werden können.

Tippen Sie 'demo()' für einige Demos, 'help()' für on-line
Hilfe, oder 'help.start()' für eine HTML Browserschnittstelle
zur Hilfe.
Tippen Sie 'q()', um R zu verlassen.

>
```

Zunächst ist die Version von R mit Erscheinungsdatum angegeben – in diesem Beispiel handelt es sich um die Version 2.9.0, die am 17.4.2009 veröffentlicht wurde. Anschließend sind einige Befehle aufgelistet, die bei Aufruf zusätzliche Informationen über R geben. Zunächst aber schauen wir uns an, wie man überhaupt Befehle eingeben kann. Die letzte Zeile in der oberen Box beginnt mit dem Symbol

```
>
```

Dieses Symbol zeigt an, dass R auf eine Befehlseingabe wartet. Tippt man beispielsweise den Befehl licence() ein und drückt schließlich die Eingabetaste, sollte folgender Text zu sehen sein:

```
> licence()

This software is distributed under the terms of the GNU General
```

```
Public License Version 2, June 1991. The terms of this license
are in a file called COPYING which you should have received with
this software and which can be displayed by RShowDoc("COPYING").

If you have not received a copy of this file, you can obtain one
at http://www.R-project.org/licenses/.

A small number of files (the API header files listed in
R_DOC_DIR/COPYRIGHTS) are distributed under the
Lesser GNU General Public LIcense version 2.1.
This can be displayed by RShowDoc("COPYING.LIB"),
or obtained at the URI given.

'Share and Enjoy.'

>
```

Die oberste Zeile der Box zeigt unsere Eingabe an. Die Zeilen, die nicht mit dem Symbol > beginnen, sind das Ergebnis, das wir durch die Befehlseingabe erzeugt haben. In diesem Beispiel handelt es sich um Lizenzinformationen zu R. Die letzte Zeile schließlich, die wieder mit einem > beginnt, zeigt an, dass R wieder auf eine Befehlseingabe wartet. Ebenfalls fällt auf, dass die Ausgabe diesmal in englischer Sprache ist. Von der Startansicht abgesehen ist dies oft der Fall.

Auf dem Startbildschirm von R sind neben dem Befehl licence() noch weitere Befehle angegeben, deren Funktion ebenfalls auf dem Startbildschirm erklärt ist:

```
license()
contributors()
citation()
demo()
```

Zudem sind zwei Befehle zur Nutzung der Hilfsfunktion ausgewiesen, sowie der Befehl zum Beenden von R:

```
help()
help.start()
q()
```

Auf diese Befehle werden wir später zurückkommen.

2.2 R als Taschenrechner

Eine einfache, aber nützliche Möglichkeit der Nutzung von R ist die Verwendung als Taschenrechner. Beispielsweise könnte man 1+1 gefolgt von der Eingabetaste eintippen. Das Resultat sollte in etwa so aussehen:

```
> 1+1
[1] 2
```

Die erste Zeile gibt unsere Eingabe wieder, die zweite, mit [1] beginnende Zeile, enthält das Ergebnis. Auf die Bedeutung der vorangestellten Klammer [1] wird später eingegangen. Weitere Beispiele für einfache Berechnungen, die nach dem gleichen Schema durchgeführt wurden sehen wie folgt aus:

```
> 10+11
[1] 21
> 34-7
[1] 27
> 3*8
[1] 24
> 20/2
[1] 10
```

Die erste Eingabe stellt wieder eine Addition dar, gefolgt von Subtraktion, Multiplikation und Division. Bei Verwendung der beiden letztgenannten Operationen beachtet R die „Punkt-vor-Strich"-Regel:

```
> 2+3*5
[1] 17
```

Klammern lassen sich durch Eingabe von runden Klammern ( ) setzen und auch diese werden entsprechend der üblichen Rechenregeln interpretiert:

```
> (2+3)*5
[1] 25
```

Bei Divisionen ist zusätzlich zu beachten, dass die Division durch Null, die eigentlich eine unzulässige Rechenoperation darstellt, durchaus ein Ergebnis liefert:

```
> 2/0
[1] Inf
> 2/0+1
[1] Inf
```

Inf steht hierbei für „unendlich", also ∞.

Durch die Verwendung eines oder mehrerer Semikola können mehrere Berechnungen in eine Zeile geschrieben werden. Dabei werden die Ergebnisse der Reihe nach ausgegeben:

```
> 1+1;2+2;3+3
[1] 2
[1] 4
[1] 6
```

Um Dezimalzahlen einzugeben, wird entsprechend der englischen Schreibweise ein Punkt als Dezimaltrennzeichen verwendet. Die im deutschen übliche Verwendung des Kommas führt zu einer Fehlermeldung:

```
> 1.5+2
[1] 3.5
> 1,5+2
Fehler: Unerwartetes ',' in "1,"
```

Die letzte Zeile der obigen Box enthält die Fehlermeldung. Diese gibt wieder, dass R das Komma-Zeichen an der verwendeten Stelle nicht verarbeiten kann.

Zum Berechnen von Potenzen bietet R verschiedene Varianten an. Beispielsweise kann 3^2 berechnet werden über 3^2 oder 3**2:

```
> 3^2
[1] 9
> 3**2
[1] 9
```

Zunächst wird die Basis eingegeben, darauf folgt eines der Symbole, welche Potenzierung angeben (^ oder **) und schließlich folgt der Exponent. Die Wurzel einer Zahl kann über den Befehl sqrt(x) gezogen werden, wobei das x für eine positive Zahl steht:

```
> sqrt(9)
[1] 3
> sqrt(16)
[1] 4
```

Der Name des Befehls leitet sich aus dem englischen Begriff *square root* (Quadratwurzel) her. Für Wurzeln mit höherem Wurzelexponenten (beispielsweise die Kubikwurzel) werden die Operatoren der Potenzrechnung verwendet:

```
> 8^(1/3)
[1] 2
> 27^(1/3)
[1] 3
> 32^(1/5)
[1] 2
```

Weitere mathematische Funktionen lassen sich analog zum Befehl sqrt(x) aufrufen. Den Wert der Exponentialfunktion für eine reelle Zahl x erhält man beispielsweise über exp(x), den natürlichen Logarithmus über log(x):

```
> exp(0)
[1] 1
> log(1)
[1] 0
```

Eine Übersicht über die gebräuchlichsten mathematischen Operatoren und Funktionen findet man in Tabelle 2.1 und Tabelle 2.2. Bei den trigonometrischen Funktion ist allgemein zu beachten, dass R für Winkel nicht Grad, sondern Radiant als Einheit verwendet. 2π rad entsprechen dabei $360°$. Insofern ist beispielsweise $\sin(90°)$ in R:

```
> sin(0.5*pi)
[1] 1
```

Bei solchen Aufrufen, die mehrere Elemente wie Funktionen, Zahlen oder ähnliches enthalten, ist zu beachten, dass R Leerzeichen zwischen Elementen ignoriert:

Tabelle 2.1: Eine Übersicht über die wichtigsten mathematischen Operatoren

Operator	Operation	Beispiel	Output
+	Addition	> 1+3	[1] 4
-	Subtraktion	> 3-1	[1] 2
*	Multiplikation	> 2*5	[1] 10
/	Division	> 10/2	[1] 5
^	Potenz	> 3^3	[1] 27
**	Potenz	> 3**3	[1] 27

Tabelle 2.2: Eine Übersicht über die wichtigsten mathematischen Funktionen

Befehl	Funktion	Beispiel	Output
exp(x)	Exponentialfunktion	> exp(2)	[1] 7.389056
log(x)	natürlicher Logarithmus	> log(3)	[1] 1.098612
log2(x)	Logarithmus zur Basis 2	> log2(3)	[1] 1.584963
log10(x)	Logarithmus zur Basis 10	> log10(3)	[1] 0.4771213
sin(x)	Sinusfunktion	> sin(10)	[1] -0.5440211
cos(x)	Kosinusfunktion	> cos(10)	[1] -0.8390715
tan(x)	Tangensfunktion	> tan(10)	[1] 0.6483608
abs(x)	Absolutwert	> abs(-10)	[1] 10

Tabelle 2.3: Übersicht über logische Operatoren

Symbol	Bedeutung	Beispiel	Output
==	Gleich	> 1==2	[1] FALSE
!=	Ungleich	> 1!=2	[1] TRUE
<	Kleiner als	> 1<2	[1] TRUE
>	Größer als	> 1>2	[1] FALSE
<=	Kleiner-gleich	> 1<=2	[1] TRUE
>=	Größer-gleich	> 1>=2	[1] FALSE

```
> sin ( 0.5 * pi )
[1] 1
> sin(0.5*pi)
[1] 1
```

Allerdings dürfen keine einzelnen Elemente durch Leerzeichen unterbrochen werden:

```
> s in(0.5*pi)
Fehler: syntax error, unexpected IN, expecting '\n' or ';' in "s in"
```

Zudem müssen Eingaben nicht immer in einer Zeile erfolgen. Wird im obigen Beispiel nur sin(0.5* eingegeben, folgt nach der mit > beginnenden Eingabezeile eine Zeile, die mit einem + beginnt und in der der Aufruf beendet werden kann:

```
> sin(0.5*
+ pi)
[1] 1
```

Dies ist insbesondere nützlich, wenn längere Befehle eingegeben werden, die so übersichtlich über mehrere Zeilen verteilt werden können.

2.3 Logische Vergleiche

R kann nicht nur Werte berechnen, sondern diese auch vergleichen. Hierfür werden sogenannte logische Operatoren verwendet. Ein einfaches Beispiel soll zur Illustration dienen:

```
> 2==2
[1] TRUE
```

Das doppelte Gleichheitszeichen == steht in R für ein logisches Gleichheitszeichen. Die Eingabe stellt also eine einfache Aussage dar: 2 ist gleich 2. R überprüft dann, ob diese Aussage zutrifft oder nicht, und gibt das Ergebnis im Output wieder. Die Aussage in unserem Beispiel ist selbstredend TRUE („wahr“). Würde man R eine falsche Aussage geben, würde als Ergebnis der Wahrheitswert FALSE („falsch“) erscheinen:

```
> 3==2
[1] FALSE
```

In Tabelle 2.3 findet sich eine Liste mit den von R verwendeten logischen Operatoren.

Für eine Verknüpfung von mehreren Vergleichen werden die Symbole & und | verwendet, sowie der Befehl xor(a,b). Das Symbol & steht für ein logisches „Und“, das Symbol | für ein „Oder“ und der Befehl xor(a,b) wird für ein ausschließendes „Entweder-Oder“ verwendet:

```
> 1==1 & 2==2
[1] TRUE
> 1==1 & 2==3
[1] FALSE
> 1==1 | 2==2
[1] TRUE
```

```
> 1==1 | 2==3
[1] TRUE
> xor(1==1,2==2)
[1] FALSE
> xor(1==1,2==3)
[1] TRUE
```

Beim Vergleichen von Werten muss man allerdings zuweilen Vorsicht walten lassen. Als Beispiel wollen wir überprüfen, ob $(\sqrt{2})^2 = 2$ ist:

```
> sqrt(2)^2==2
[1] FALSE
```

Dies mag auf den ersten Blick überraschend sein, insbesondere wenn man sich das Ergebnis der linken Seite des Aufrufes anzeigen lässt:

```
> sqrt(2)^2
[1] 2
```

Als Resultat der linken Seite ergibt sich also 2, dennoch wird angezeigt, dass dieses Ergebnis nicht gleich 2 ist? Ursache hierfür ist die Genauigkeit, mit der Dezimalzahlen behandelt werden können. $\sqrt{2}$ beispielsweise ist eine irrationale Zahl mit endlos vielen Dezimalstellen, die nicht periodisch sind. Hier wird schnell klar, dass diese Zahl nicht vollständig gespeichert werden kann, sondern die Dezimaldarstellung an einer bestimmten Stelle „abgeschnitten“ werden muss. Allgemein werden sogenannte Gleitkommazahlen verwendet, von denen es unterschiedliche Typen gibt. R verwendet den IEEE 754 Standard mit sogenannter „doppelter Genauigkeit“. In diesem Format ist eine Genauigkeit von etwa 16 Dezimalstellen möglich. Bezogen auf unser obiges Beispiel bedeutet dies, dass $(\sqrt{2})^2$ zwar bei einem einzelnen Aufruf vermeintlich korrekt als 2 angezeigt wird, da R ohne explizite Angabe hierzu nicht 16 Dezimalstellen anzeigt und aufrundet. Zugleich unterscheiden sich die beiden Seiten der logischen Abfrage nichtsdestotrotz minimal:

```
> sqrt(2)^2 - 2
[1] 4.440892e-16
```

Ein Befehl, der dennoch einen korrekten Vergleich ermöglicht ist all.equal(), der als Argumente die zu vergleichenden Werte nimmt:

```
> all.equal(sqrt(2)^2 , 2)
[1] TRUE
```

In den meisten Anwendungskontexten ergeben sich aus der begrenzten Gleitkommagenauigkeit keine Probleme.

2.4 Zuweisungen

Die Ergebnisse von Berechnungen, wie sie beispielsweise in den vorherigen Abschnitten vorgestellt wurden, stellen gegebenenfalls nur Zwischenschritte dar und sollen weiterverwendet

werden. Eine praktischer Ansatz, um eine einfache Weiterverwendung zu ermöglichen, sind sogenannte Zuweisungen (engl.: *assignment*). Bei diesen werden Ergebnisse von Berechnungen mit einem Namen versehen und im Speicher behalten. Dies funktioniert nicht nur mit „einfachen" Ergebnissen, sondern auch mit Datensätzen, Tabellen, Matrizen etc. Es gibt verschiedene Möglichkeiten, Zuweisungen in R auszuführen, wir beschränken uns hier allerdings auf die gebräuchlichste Variante.

Zur Erklärung der grundsätzlichen Funktionsweise soll wieder ein einfaches Beispiel dienen. Angenommen, man möchte das Ergebnis der Berechnung 1+1 speichern. Dann sollte man sich in einem ersten Schritt einen Namen überlegen, dem dieses Ergebnis zugewiesen wird und der es ermöglicht, darauf zurückzugreifen. Wir entscheiden uns für den Namen Ergebnis. Um die Zuweisung durchzuführen, werden die Zeichen <- (ein Kleiner-Als-Zeichen gefolgt von einem Minus) verwendet. Links von diesem „Pfeil" muss der Name stehen, rechts die Berechnung. Anschließend kann man das Ergebnis über die Eingabe des festgelegten Namen aufrufen:

```
> Ergebnis <- 1+1
> Ergebnis
[1] 2
```

Die erste Zeile der obigen Eingabe führt wohlgemerkt zu keinem Output, sondern lässt uns lediglich in eine neue Zeile für die Befehlseingabe wechseln. Mit der zweiten Zeile wird R angegeben, dass das Objekt Ergebnis aufgerufen werden soll. Man hätte statt der Berechnung 1+1 auch direkt das Ergebnis 2 verwenden können:

```
> Ergebnis <- 2
> Ergebnis
[1] 2
```

Darüber hinaus sind auch kompliziertere Eingaben möglich, beispielsweise:

```
> Resultat <- sin(3)+cos(3)*tan(3)
> Resultat
[1] 0.28224
```

Mit den so gespeicherten Ergebnissen kann einfach weiter gerechnet werden:

```
> Ergebnis*2
[1] 4
> 1 + Resultat
[1] 1.28224
> Ergebnis - Resultat
[1] 1.71776
```

Zudem können sie wiederum für neue Zuweisungen verwendet werden:

```
> Ergebnis_2 <- exp(Ergebnis)
> Ergebnis_2
[1] 7.389056
```

Zuweisungen können wie erwähnt nicht nur zum (temporären) Abspeichern von einfachen Berechnungen verwendet werden, sondern eine Vielzahl an verschiedenen Inhalten abdecken.

Beispielsweise könnte man Text eingeben, wobei zu beachten ist, dass dieser in Anführungszeichen stehen muss, da R den angegebenen Text ansonsten als Namen eines abgespeicherten Ergebnisses interpretiert und gegebenenfalls eine Fehlermeldung erscheint:

```
> abc <- Hallo
Fehler: Objekt "Hallo" nicht gefunden
> abc <- "Hallo"
> abc
[1] "Hallo"
```

Das allgemeine Schema bei der Verwendung von Zuweisungen ist immer identisch: Name <- Inhalt. Bei der Namensgebung sind einige grundsätzliche Regeln zu beachten. Namen können im Prinzip aus beliebig vielen Zeichen bestehen, wobei das erste Zeichen immer ein Buchstabe sein muss, hierauf können auch Zahlen und einige Sonderzeichen – wie zum Beispiel der Unterstrich – folgen. Bei Buchstaben wird zwischen Groß- und Kleinschreibung unterschieden, so dass beispielsweise ergebnis ein anderer Name ist als Ergebnis. Wurde ein Name bereits bei einer Zuweisung verwendet, führt die erneute Zuweisung eines Inhaltes zu diesem Namen zum Überschreiben des alten Inhaltes. Einige Beispiele:

```
> a <- 5
> a
[1] 5
> A <- 3
> A
[1] 3
> b <- 7
> b
[1] 7
> b <- 9
> b
[1] 9
> A_1 <- 13
> A_1
[1] 13
> 1_A <- 12
Fehler: Unerwartete Eingabe in "1_"
```

Der Fehler in der letzten Zeile weist darauf hin, dass R eine Zuweisung auf einen Namen, der mit einer Ziffer beginnt, nicht umsetzen kann.

2.5 Objekte in R

Durch die im vorherigen Abschnitt vorgenommenen Zuweisungen wurden sogenannte Objekte erzeugt. Grundsätzlich sind in R alle Arten von behandelbaren Inhalten Objekte. Dabei unterscheidet R zwischen verschiedenen Arten von Objekten, die als Klassen bezeichnet werden.

Im Speicher befinden sich für alle definierten Objekte deren Inhalte, deren Namen sowie Angaben dazu, zu welcher Klasse ein Objekt gehört. Die letztgenannte Information zeigt an, wie ein Objekt von R zu behandeln ist und welche Operationen möglich sind.

Informationen darüber, zu welcher Klasse ein Objekt gehört, gibt der Befehl class(). Beispielsweise könnte man sich für die Klasse des im vorherigen Abschnitt definierten Objekts a interessieren:

```
> class(a)
[1] "numeric"
```

numeric steht hierbei für ganze und reelle Zahlen. Hier wird uns also schlicht angezeigt, dass das Objekt a eine Zahl (ggf. auch mehrere) enthält. Wendet man den Befehl auf das Objekt abc an, wird ausgegeben, dass der Inhalt aus Text besteht (bzw. mehreren Textelementen):

```
> class(abc)
[1] "character"
```

Man könnte sich auch für die Klasse der Funktion sqrt() interessieren. Hier erhält man die Angabe, dass es sich um ein Objekt der Klasse function handelt:

```
> class(sqrt)
[1] "function"
```

Auch die anderen mathematischen Befehle wie beispielsweise exp() weisen die Klasse function auf – ebenso wie der Befehl class selbst. Objekte dieser Klasse werden im weiteren wie bisher als Funktionen oder Befehle bezeichnet.

Wie erwähnt zeigen Klassen an, wie ein Objekt von R verwendet werden kann, und implizieren darüber hinaus eine bestimmte Struktur von Objekten. Der Aufruf von Funktionen folgt beispielsweise allgemein folgender Struktur: name(Argument1,Argument2,...). Zunächst wird der Name der Funktion angegeben und anschließend folgen in runden Klammern einzelne Argumente, wobei sich die Zahl der Argumente von verschiedenen Funktionen unterscheidet. Argumente können dabei entweder einer Funktion ein Objekt übergeben, mit dem diese Funktion etwas machen soll, oder aber Optionen festlegen, über die gesteuert wird, was eine Funktion genau macht. Ein Beispiel für eine Funktion, der ein Argument übergeben wird, war beispielsweise sqrt(2), und ein Beispiel für eine Funktion mit zwei Argumenten ist all.equal(sqrt(2)^2,2) gewesen. Diese hier beispielhaft aufgegriffene Struktur wird uns im weiteren Verlauf immer wieder begegnen.

Mittels des Befehls mode() erfährt man, welchem Datentypus ein Objekt zugeordnet wird. Hiermit ist gemeint, welches Format die Inhalte eines Objektes haben – enthält es beispielsweise Text oder Zahlen? Für das Objekt abc sind Datentypus und Klasse beispielsweise identisch:

```
> mode(abc)
[1] "character"
```

Dies ist allerdings bei weitem nicht immer der Fall. Ein Beispiel sind Matrizen, die wir erst in einem späteren Kapitel besprechen. Diese weisen als Klasse matrix auf, allerdings kann der Datentypus beispielsweise numeric sein.

Schließlich können Objekte zusätzlich zu diesen beiden Informationen noch weitere Eigenschaften besitzen, die (teilweise) mit dem Befehl attributes() abgefragt werden können. Für das Objekt abc ergibt sich beispielsweise:

```
> attributes(abc)
NULL
```

Die Ausgabe NULL steht allgemein für eine leere Menge und gibt in diesem Beispiel an, dass neben Klasse und Datentypus keine weiteren Eigenschaften des Objektes abc gespeichert sind.

Objekte werden von R in sogenannten Umgebungen gespeichert. Von diesen gibt es verschiedene und manche Befehle erzeugen sogar neue Umgebungen. Am wichtigsten ist aber der sogenannte Workspace, in dem in der Regel die vom Nutzer erzeugten Objekte abgelegt sind. Über den Befehl ls() lässt sich der aktuelle Inhalt des Workspace abrufen, wobei zu beachten ist, dass die Klammern des Befehlsaufrufes hierbei leer bleiben:

```
> ls()
 [1] "a"        "A"        "A_1"      "abc"      "b"        "Ergebnis"
 [7] "Ergebnis2" "Resultat"
```

Alle in den vorherigen Abschnitten erstellten Objekte werden in zwei Zeilen aufgelistet. Die Angabe [1] in der ersten Zeile zeigt an, dass diese Zeile beim ersten Objekt beginnt. Die Angabe [7] zu Beginn der zweiten Zeile bedeutet, dass diese mit dem siebten Objekt beginnt. Dabei werden Objekte in alphabetischer Reihenfolge angezeigt. Das gleiche Ergebnis lässt sich auch über den Befehl objects() erreichen.

Mittels des Befehls rm(obj) können Objekte aus dem Speicher entfernt werden, wobei obj für den Namen eines beliebigen Objektes im Workspace steht:

```
> rm(Ergebnis2)
> ls()
[1] "a"     "A"     "A_1"     "abc"     "b"     "Ergebnis"     "Resultat"
```

Dies ist zuweilen sinnvoll, da manche Objekte, wie beispielsweise empirische Datensätze, recht groß sein können und viel Speicherplatz belegen. Wird ein solcher Datensatz nicht mehr benötigt, kann er aus dem Speicher entfernt werden, um Ressourcen zu sparen. Möchte man alle erzeugten Objekte auf einmal aus dem Workspace löschen, kann folgender Aufruf verwendet werden:

```
> rm(list=ls())
```

Hierbei wird dem Befehl rm() eine Liste von Objekten übergegeben, die gelöscht werden sollen, wobei diese Liste gerade gleich ls(), also dem Inhalt des Workspace ist.

Beendet man R, in dem man entweder den Button zum schließen des Fensters oder aber den Befehl q() verwendet, erscheint eine Abfrage, ob der Inhalt des Workspace gespeichert werden soll. Wird mit „Ja“ geantwortet, steht der Inhalt des Workspace beim nächsten Start von R wieder zur Verfügung. Als allgemeine Faustregel gilt, dass dies nicht unbedingt sinnvoll ist, da gegebenenfalls nicht mehr benötigte Objekte ebenfalls gespeichert werden und daher ein gezieltes abspeichern einzelner Objekte sinvoller ist (s. hierfür Abschnitt 3.4.1).

2.6 Vektoren

2.6.1 Erstellen von Vektoren

Die Verwendung von Vektoren ist in R von immenser Bedeutung. Ein Vektor wurde beispielsweise bereits im letzten Abschnitt über den Befehl ls() abgerufen:

```
> ls()
 [1] "a"         "A"         "A_1"       "abc"       "b"         "Ergebnis"
 [7] "Ergebnis2" "Resultat"
```

Dieser Befehl zeigt die Objekte, die im Workspace abgespeichert sind, als Vektor an. Dabei ist ein Vektor etwas vereinfacht eine Auflistung von einzelnen Elementen. Eine Möglichkeit, Vektoren formal aufzuschreiben, besteht darin, die einzelnen Vektorelemente durch Kommata getrennt hintereinander aufzureihen und in runde Klammern zu setzen. Beispielsweise besteht der Vektor $(1, 2, 3)$ aus den Elementen 1, 2 und 3 und der Vektor $(5, 7, 9)$ aus den Elementen 5, 7 und 9, wobei 5 der erste Eintrag des Vektors ist, 7 der zweite Eintrag und 9 der dritte. Dabei müssen die Einträge von Vektoren nicht notwendigerweise Zahlen sein, wie das obige Beispiel mit ls() verdeutlicht. Ebenso können Vektoren auch nur einen Eintrag enthalten. Um Vektoren von „einfachen“ Zahlen abzugrenzen, bezeichnen wir letztere im weiteren zuweilen als Skalare. Wie man in R selber Vektoren erstellen und manipulieren kann ist Gegenstand dieses Abschnittes. Dabei stellen Vektoren nur eine von vielen speziellen Objektklassen dar. Weitere werden im Verlauf des Textes eingeführt, wie beispielsweise Matrizen im Kapitel zur linearen Algebra.

Vektoren lassen sich in R über den Befehl c(element1,element2,...) eingeben. Das c steht hierbei für *concatenate*, was soviel bedeutet wie „verketten“. Innerhalb der Klammern werden die einzelnen Elemente des zu erstellenden Vektors durch Kommata getrennt eingetragen. Um beispielsweise den Vektor $(1, 2, 3)$ einzugeben, wird folgender Befehl verwendet:

```
> x <- c(1,2,3)
> x
[1] 1 2 3
> class(x)
[1] "numeric"
```

Die Klasse numeric ist bereits im vorherigen Abschnitt zu Objekten angesprochen worden und es wurde bereits angedeutet, dass diese Klasse für Objekte steht, die einen oder mehrere numerische Werte enthalten. Nun dürfte klar sein, dass diese Klasse für Vektoren steht, die Zahlen enthalten. Auch Skalare, die im Workspace abgelegt sind, werden als Vektoren mit nur einem Eintrag gespeichert. Das Ergebnis des Aufrufes class(x) ist ebenfalls ein Vektor mit nur einem Element:

```
> class(class(x))
[1] "character"
```

Die Klasse character steht hier wie bereits angedeutet für einen Vektor, dessen Elemente aus Text bestehen.

Ein zweites Beispiel für einen Vektor, der aus Zahlen besteht, ist $(5, 2, 4)$:

```
> y <- c(5,2,4)
> y
[1] 5 2 4
> class(y)
[1] "numeric"
```

Vektoren können an sich wie erwähnt beliebige Datentypen und somit auch Text enthalten, wobei dieser in Anführungszeichen gesetzt sein muss:

```
> z <- c("eins","zwei","drei")
> z
[1] "eins" "zwei" "drei"
> class(z)
[1] "character"
```

Möchte man Zahlenfolgen von einer Zahl a bis zu einer Zahl b mit Schrittgröße 1 erstellen, reicht es die erste und die letzte Zahl dieser Sequenz getrennt durch einen Doppelpunkt anzugeben:

```
> x <- 1:3
> x
[1] 1 2 3
```

Möchte man eine andere Schrittgröße als 1 verwenden, kommt der Befehl $\mathrm{seq(a,b,by=s)}$ zur Anwendung. Dieser nimmt als Argumente die erste Zahl der Sequenz a, die letzte Zahl der Sequenz b und die Schrittgröße s, die über das Argument by angegeben wird:

```
> x1 <- seq(1,3,by=0.5)
> x1
[1] 1.0 1.5 2.0 2.5 3.0
```

Soll eine Zahl a insgesamt w-mal wiederholt werden, kann der Befehl $\mathrm{rep(a,w)}$ benutzt werden:

```
> x2 <- rep(1,5)
> x2
[1] 1 1 1 1 1
```

Diese Befehle lassen sich auch kombinieren:

```
> x3 <- c(rep(1,3),seq(1,3,by=0.5),4:7)
> x3
 [1] 1.0 1.0 1.0 1.0 1.5 2.0 2.5 3.0 4.0 5.0 6.0 7.0
```

Die Zahl der Elemente eines Vektors lässt sich über den Befehl $\mathrm{length(vektor)}$ auslesen:

```
> length(x)
[1] 3
> length(y)
[1] 3
```

Erzeugt man einen Skalar und lässt sich dessen Länge anzeigen, wird klar, dass Skalare wie oben bereits erwähnt als Vektoren mit nur einem Eintrag gespeichert werden:

```
> a <- 5
> length(a)
[1] 1
```

Auch das Ergebnis des Aufrufes length(x) ist ein Vektor mit einem Element:

```
> length(length(x))
[1] 1
```

2.6.2 Rechnen mit Vektoren

Addition, Subtraktion, Multiplikation und Division von Vektoren mit Skalaren erfolgt über die Symbole der Grundrechenarten, wobei Addition und Multiplikation kommutativ sind:

```
> x <- c(1,2,3)
> x+1
[1] 2 3 4
> 1+x
[1] 2 3 4
```

Der Wert 1 wird also einfach zu allen Elementen des Vektors hinzuaddiert. Der zweite und der dritte Aufruf zeigen die oben erwähnte Kommutativität: In welcher Reihenfolge Skalar und Vektor addiert werden spielt für das Ergebnis keine Rolle. Subtraktion funktioniert zwar analog zur Addition, allerdings ist diese nicht kommutativ:

```
> x-1
[1] 0 1 2
> 1-x
[1]  0 -1 -2
```

Im ersten Fall wird von jedem Eintrag des Vektors x der Wert 1 abgezogen, so dass das Ergebnis dem Vektor $(1-1, 2-1, 3-1)$ entspricht. Im zweiten Fall wird von 1 jeder Eintrag des Vektors x abgezogen, also genau umgekehrt vorgegangen, so dass man $(1-1, 1-2, 1-3)$ als Resultat erhält.

Multiplikation und Division funktionieren analog, wobei Multiplikation wieder kommutativ ist, Division hingegen nicht:

```
> x*2
[1] 2 4 6
> 2*x
[1] 2 4 6
> x/2
[1] 0.5 1.0 1.5
> 2/x
[1] 2.0000000 1.0000000 0.6666667
```

Addition, Subtraktion, Multiplikation und Division von zwei Vektoren miteinander erfolgt über die selben Symbole:

```
> y <- c(5,2,4)
> x+y
[1] 6 4 7
> x-y
[1] -4  0 -1
> x*y
[1] 5 4 12
> x/y
[1] 0.20 1.00 0.75
```

Hierbei wird jeweils elementweise vorgegangen. Beispielsweise wird bei x+y das erste Element von x zum ersten Element von y addiert, das zweite Element von x zum zweiten Element von y usw. Zu beachten ist dabei, dass R diese Operationen auch durchführt, wenn die beiden verwendeten Vektoren nicht gleicher Länge sind:

```
> a <- c(1,2,3,4,5,6)
> a*x
[1] 1  4  9 4 10 18
> length(a)
[1] 6
> length(x)
[1] 3
```

Die Elemente des kürzeren Vektors werden dabei so oft wiederholt, bis die Länge des längeren Vektors erreicht ist, so dass im vorherigen Beispiel letztlich Vektoren $(1,2,3,1,2,3)$ und $(1,2,3,4,5,6)$ multipliziert wurden. Ist die Länge des kürzeren Vektors kein natürlicher Teiler der Länge des längeren Vektors geht R zwar analog vor, allerdings wird zusätzlich noch eine Warnung ausgegeben:

```
> b <- c(1,2,3,4,5)
> b*x
[1]  1  4  9  4 10
Warning message:
In b * x : Länge der längeren Objekts
        ist kein Vielfaches der Länge der kürzeren Objektes
```

Möchte man die Summe aller Elemente eines Vektors bilden, kann der Befehl sum() eingesetzt werden:

```
> sum(x)
[1] 6
```

Spezielle mathematische Funktionen lassen sich auf alle Elemente eines Vektors anwenden, indem dieser als Argument der gewünschten Funktion verwendet wird. Beispielsweise lassen sich die Wurzeln aller Einträge eines Vektors errechnen über:

```
> sqrt(x)
[1] 1.000000 1.414214 1.732051
```

Im Kapitel zur linearen Algebra werden einige Rechenoperationen wie das Tensorprodukt zweier Vektoren ergänzt.

2.6.3 Indizierung von Vektoren

Um einzelne Elemente von Vektoren aufzurufen, wird hinter dem Namen des Vektors der Index des Elements in eckigen Klammern angegeben. Um beispielsweise das erste Element des Vektos x aus dem letzten Unterabschnitt aufzurufen, würde man folgende Eingabe verwenden:

```
> x[1]
[1] 1
```

Ganz analog könnte man das dritte Element des Vektors y aufrufen über:

```
> y[3]
[1] 4
```

In beiden Fällen ist das Ergebnis ein Vektor, der nur ein Element enthält. Nun dürfte wahrscheinlich auch klar sein, warum viele der bisher erzeugten Ausgaben mit [1] beginnen: die Ausgabe stellt einen Vektor dar und die Anzeige beginnt beim ersten Element dieses Vektors. Ist der angezeigte Vektor relativ lang und wird über mehrere Zeilen hinweg angezeigt, steht vor jeder Zeile die Nummer des Elements, mit dem diese Zeile beginnt.

Möchte man mehrere Elemente eines Vektors aufrufen, muss innerhalb der eckigen Klammern ein Vektor angegeben werden, dessen Elemente den Indizes der Elemente des ursprünglichen Vektors entsprechen, die man aufrufen möchte. Das erste und das dritte Element von x erhält man zum Beispiel über:

```
> x[c(1,3)]
[1] 1 3
```

Hierbei ist zu beachten, dass die Reihenfolge der Nummern innerhalb der eckigen Klammern von Bedeutung ist:

```
> y[c(1,3)]
[1] 5 4
> y[c(3,1)]
[1] 4 5
```

Im ersten Fall erhält man erst das erste und dann das dritte Element, im zweiten Fall das dritte und erst dann das erste Element. Auf diesem Weg lassen sich einzelne Elemente auch mehrmals ausgeben:

```
> y[c(1,1,1)]
[1] 5 5 5
> y[c(3,3)]
[1] 4 4
```

Durch die Verwendung eines Minuszeichens lässt sich erreichen, dass ein Vektor *ohne* die entsprechenden Elemente angezeigt wird:

```
> x[-2]
[1] 1 3
> y[-c(1,3)]
[1] 2
```

Durch das Angeben von Indizes in Kombination mit Zuweisungen lassen sich einzelne Elemente von Vektoren ändern:

```
> x
[1] 1 2 3
> x[1] <- 13
> x
[1] 13  2  3
```

Ebenso lässt sich mit einzelnen Elementen eines Vektors rechnen, wie beispielsweise:

```
> x[2]^2
[1] 4
```

2.6.4 Logische Vergleiche mit Vektoren

Vektoren können auch für logische Vergleiche verwendet werden. Hierbei muss zunächst unterschieden werden, ob ein Vektor mit einem einfachen Wert oder aber einem anderen Vektor verglichen wird. Im ersten Fall werden alle Einträge des Vektors mit diesem Wert verglichen und als Ergebnis wird ein Vektor mit Wahrheitswerten ausgeben, der die Resultate der Vergleiche zeigt:

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

Beim ersten Vergleich wird für alle Einträge von x überprüft, ob diese gleich 1 sind, was nur für den ersten Eintrag zutrifft. Beim zweiten Aufruf wird auf Gleichheit mit 2 kontrolliert.

Möchte man feststellen, ob zumindest einer der Einträge des Vektors dem gewählten Wert entspricht, wird der Befehl any() verwendet:

```
> any(x==3)
[1] TRUE
> any(x==4)
[1] FALSE
```

Dieser Befehl nimmt als Argument einen Vektor mit Wahrheitswerten und überprüft, ob zumindest einer dieser Wahrheitswerte TRUE ist. Beim ersten Aufruf beispielsweise wird also letztlich überprüft, ob mindestens eines der Elemente des Vektors x gleich 3 ist.

Auf ähnliche weise funktioniert der Befehl all(), der überprüft, ob alle Einträge eines Vektors mit Wahrheitswerten TRUE sind:

```
> all(x==1)
[1] FALSE
> y <- c(1,1,1)
> all(y==1)
[1] TRUE
```

Beim Vergleich zweier Vektoren werden die einzelnen Einträge paarweise verglichen. Der erste Eintrag der beiden Vektoren wird also verglichen und ein Wahrheitswert resultiert, der zweite Eintrag der Vektoren wird verglichen und es resultiert ein Wahrheitswert und so weiter. Als Ergebnis erhält man somit wieder einen Vektor mit Wahrheitswerten:

```
> x==y
[1]  TRUE FALSE FALSE
> z <- c(1,2,3)
> x==z
[1] TRUE TRUE TRUE
```

Um zu überprüfen, ob mindestens ein Eintrag oder alle Einträge identisch sind, können wieder die Befehle any() und all() verwendet werden.

2.7 Die Hilfefunktion von R

Benötigt man Informationen über die Funktionsweise eines Befehls in R, lässt sich eine Hilfsseite zu diesem Befehl über den Aufruf help(befehl) einsehen. Würde man beispielsweise wissen wollen, wie der Befehl sqrt() funktioniert, würde folgende Eingabe nötig sein:

```
> help(sqrt)
```

Unter Windows sollte sich die Hilfe von R in einem neuen Fenster öffnen.

Mittels des Befehls help.search("suche") lässt sich die Hilfe durchsuchen. Dabei ist zu beachten, dass der in der Hilfe zu suchende Text in Anführungszeichen stehen sollte:

```
> help("regression")
```

Allerdings ist diese integrierte Suchfunktion nicht immer hilfreich. Ursache hierfür ist, dass nur in der Hilfe zum R-Basispaket und von installierten Erweiterungen gesucht wird. Ist ein benötigtes Verfahren in einem Paket vorhanden, dass nicht installiert ist, lässt sich dies mit der integrierten Suchfunktion nicht feststellen. Hier bieten sich mehrere Alternativen an. Zunächst empfiehlt sich die R-Reference-Card, die unter http://cran.r-project.org/doc/contrib/Short-refcard.pdf abgerufen werden kann und eine Übersicht über die wichtigsten Befehle liefert. Des weiteren kann unter http://www.rseek.org/ auf eine speziell auf R zugeschnittene Suchmaschine zurückgegriffen werden. Schließlich enthält die in Abschnitt 1 angegebene Literatur oftmals nützliche Befehlsreferenzen und Stichwortverzeichnisse. Sollte auch dies nicht helfen bleibt immer noch die Möglichkeit, eine allgemeine Suchmaschine zu benutzen.

2.8 R und Editoren

Im Laufe einer R-Sitzung werden zuweilen viele verschiedene Befehle ausgeführt. Damit das Vorgehen nachvollziehbar und reproduzierbar bleibt, empfiehlt es sich, R in Kombination mit einem speziellen Editor zu verwenden, von denen mehrere zur Auswahl stehen. Diese erlauben

es, sogenannte Skriptdateien zu erstellen, in denen Befehle abgelegt werden können. Nachdem eine Skriptdatei erstellt wurde, kann diese zur Ausführung an R geschickt werden, welches die enthaltenen Befehle der Reihe nach abarbeitet. Durch das Speichern und Laden von Skripten kann man Auswertungen auch nach einiger Zeit wiederholen und anderen zugänglich machen.

Die Programme Emacs, Vim und RKWard stehen unter Windows und Unix zur Verfügung, das Programm Tinn-R ist nur für Windows erhältlich. Es gibt zwar noch weitere Editoren, allerdings sind die genannten die gebräuchlichsten. Für Einsteiger empfiehlt sich Tinn-R, welches unter http://www.sciviews.org/Tinn-R/ kostenlos heruntergeladen werden kann. Wird Unix verwendet, ist das unter http://rkward.sourceforge.net/ zu beziehende RKWard das einfacher zugängliche Programm. Emacs und Vim sind hingegen nur für fortgeschrittene Anwender zu empfehlen und können unter http://www.gnu.org/software/emacs/ bzw. http://www.vim.org bezogen werden. Die Funktionsweise der genannten Programme ist den entsprechenden Dokumentationen zu entnehmen.

2.9 Kommentare

Unabhängig davon, welchen Editor man verwendet, ist es sinnvoll seine Skriptdateien mit Kommentaren zu versehen. Diese werden grundsätzlich mit dem Zeichen # kenntlich gemacht. Sämtliche Eingaben, die in einer Zeile nach diesem Symbol folgen, werden von R nicht interpretiert. Hierüber kann man Anmerkungen und Erklärungen in eine Skriptdatei setzen, die es einem selbst und anderen ermöglichen, die verwendeten Befehle auch im Nachhinein besser zu verstehen, was insbesondere bei komplizierteren Berechnungen sinnvoll ist. Ein einfaches Beispiel:

```
> 1+1 # Eine einfache Berechnung
[1] 2
```

Wird hingegen keine Raute verwendet, erscheint eine Fehlermeldung:

```
> 1+1 Eine einfache Berechnung
Fehler: Unerwartetes Symbol in "1+1 Eine"
```

2.10 Erweiterungen für R

Wie bereits in der Einleitung erwähnt lässt sich R durch zusätzliche *packages* erweitern. Eine Vielzahl solcher Zusatzpakete ist über das *Comprehensive R Archive Network* (CRAN) zugänglich. Die Installation kann dabei in der Regel direkt aus R heraus erfolgen. Beispielsweise könnte man sich für das Paket network interessieren. Um dieses zu installieren wird folgender Befehl verwendet:

```
> install.packages("network")
```

Hierbei ist auf die Anführungszeichen zu achten, da sonst eine Fehlermeldung erscheint. Nach der Eingabe dieses Befehls wird man aufgefordert, einen CRAN-Server auszuwählen, von dem das gewünschte Paket heruntergeladen werden soll. Nachdem ein Server gewählt wurde beginnt automatisch das Herunterladen und die Installation des Paketes. Nach der erfolgreichen Installation kann das Paket mittels

```
> library("network")
```

geladen werden, wobei hier die Anführungszeichen auch weggelassen werden können. Dieses gezielte Aufrufen des Paketes ist notwendig, damit auf dessen Funktionen zurückgegriffen werden kann. Nach jedem Neustart von R ist ein erneutes Laden des Paketes über library() notwendig, um es zu verwenden. Auch die R-interne Hilfe zum Paket ist erst nach dem Laden verfügbar. Das Installieren und Laden von anderen Erweiterungen erfolgt ganz analog zum hier vorgestellten Vorgehen, lediglich der Name des Paketes muss ausgetauscht werden. Unter http://cran.r-project.org/web/views/SocialSciences.html findet sich eine (nicht unbedingt vollständige) Auflistung von *packages*, die für Sozialwissenschaftler von Interesse sein können.

3 Daten einlesen, bearbeiten, importieren

3.1 Daten einlesen

Daten lassen sich grundsätzlich in verschiedenen Anordnungen abspeichern. Die üblichste Variante besteht darin, Daten in einer Art Matrix anzuordnen, bei der die Zeilen einzelne Beobachtungen und die Spalten jeweils Variablen repräsentieren. Angenommen, man hätte Informationen zu den Noten von fünf Studenten in drei Seminaren. Dann könnte eine Datenmatrix wie Tabelle die folgende Tabelle aussehen:

Tabelle 3.1: Ein Beispiel für eine einfache Datentabelle

	Seminar A	Seminar B	Seminar C
ω_1	$1,3$	$1,7$	$1,0$
ω_2	$3,3$	$2,7$	$2,7$
ω_3	$2,7$	$4,0$	$3,7$
ω_4	$1,7$	$1,3$	$2,0$
ω_5	$1,3$	$1,3$	$1,3$

In der ersten Zeile der Tabelle stehen die Namen der Variablen. Die folgenden Zeilen beginnen mit symbolischen Namen (ω_1 bis ω_5) für die einzelnen Studenten, jeweils gefolgt von den Noten in den einzelnen Veranstaltungen. Beispielsweise hat Student ω_1 das Seminar A mit der Note $1,3$ abgeschlossen, das Seminar B mit der Note $1,7$ und das Seminar C mit $1,0$.

Diese Anordnung wird von allen Statistik-Programmen als Standard verwendet. Allerdings unterscheiden sich die einzelnen Softwarepakete in den Formaten, die zum Abspeichern von Daten verwendet werden. Insofern ist beim Laden von Daten zunächst darauf zu achten, in welchem Format diese abgelegt sind.

Eine Möglichkeit besteht darin, Daten einfach als „Textdatei“ abzuspeichern und zu öffnen. Diese Möglichkeit bietet jedes Statistik- oder Tabellenkalkulationsprogramm. Solche Dateien können bereits mit einfachen Editoren und Textverarbeitungsprogrammen betrachtet und erzeugt werden. Grundsätzlich werden hierbei Daten wie in Tabelle 3.1 angeordnet. Einzelne Datenzeilen werden durch einfache Zeilenumbrüche kenntlich gemacht. Einzelne Spalten können dabei entweder durch Leerzeichen oder aber andere Symbole getrennt werden. Üblich sind hierbei Kommata, Semikola oder „Tab-Stopps“. Weiterhin ist zu beachten, ob als Dezimaltrennzeichen Punkte oder Kommata verwendet werden und wie Textdaten kenntlich gemacht wer-

den. Für letzteres werden zumeist die einfachen Anführungszeichen, die sich auf der Tastatur finden, benutzt: "Text". Einige Beispiele dafür, wie die Daten aus Tabelle 3.1 in verschiedenen Varianten abgespeichert werden könnten, finden sich in Abbildung 3.1.

Um solche einfachen Textdateien einzulesen stellt R mehrere Befehle zur Verfügung. Ein allgemein anwendbarer Befehl ist read.table. Dieser nimmt als Argument zunächst den Pfad und den Namen der Datei. Dieser muss in Anführungszeichen angegeben werden. Zudem ist zu beachten, dass statt eines \ in der Pfadangabe ein einfacher Schrägstrich / verwendet wird. Die Optionen sep und dec können dazu verwendet werden, um das Spalten- und das Dezimaltrennzeichen anzugeben. Dabei muss nach Angabe der Option ein Gleichheitszeichen folgen (z.B. sep=) und das gewünschte Zeichen in Anführungszeichen gesetzt werden (z.B. dec=","). Werden diese Optionen nicht spezifiziert, wird von Punkten als Dezimal- und Leerzeichen als Spaltentrennzeichen ausgegangen. Schließlich muss R mitgeteilt werden, ob die erste Zeile der Datei die Namen der Variablen enthält (wie in unseren Beispielen). Hierzu wird die Option header verwendet, die die logischen Werte TRUE und FALSE verarbeiten kann. Wird diese Option nicht spezifiziert, werden Variablennamen nur erkannt, wenn es einen Variablennamen weniger als Spalten gibt. Für die in Abbildung 3.1 angegebenen Beispiele sollte also header=TRUE verwendet werden.

Angenommen, man hätte die Daten aus Tabelle 3.1 in einer Textdatei abgelegt, die wie in Beispiel 1 in Abbildung 3.1 aufgebaut ist. Diese Datei sei unter dem Namen data.csv im Verzeichnis C:\daten gespeichert. Dann könnte man diese Daten über

```
# Beispiel 1
> dat <- read.table("C:/daten/data.csv",header=TRUE)
```

aufrufen. Die Daten sind dann in einem Objekt mit dem Namen dat abgelegt und können hierüber aufgerufen und weiter verarbeitet werden. Die anderen Beispiele aus Abbildung 3.1 könnte man aufrufen mit:

```
# Beispiel 2
> dat <- read.table("C:/daten/data.csv",dec=",",header=TRUE)
# Beispiel 3
> dat <- read.table("C:/daten/data.csv",sep=",",header=TRUE)
# Beispiel 4
> dat <- read.table("C:/daten/data.csv",sep=";",header=TRUE)
# Beispiel 5
> dat <- read.table("C:/daten/data.csv",dec=",",sep="\t",header=TRUE)
```

Die Verwendung des Tab-Stopp bei Beispiel 5 wird hierbei durch \t kenntlich gemacht.

Weitere Befehle zum Einlesen von Textdateien funktionieren analog zum Befehl read.table, allerdings unterscheiden sich die Standardeinstellungen für die Optionen sep und dec, die in der folgenden Auflistung mit angegeben sind:

```
read.csv("Dateiname",sep=",",dec=".")
read.csv2("Dateiname",sep=";",dec=",")
read.delim("Dateiname",sep="\t",dec=".")
read.delim2("Dateiname",sep="\t",dec=",")
```

Abbildung 3.1: Diverse Beispiele für in einfachen Textdateien abgelegten Daten; [...] deutet die Auslassung der Beobachtungen 3 bis 5 an;

Beispiel 1; Spalten durch Leerzeichen getrennt, Punkt als Dezimaltrennzeichen:

```
"Seminar A"  "Seminar B"  "Seminar C"
1.3  1.7  1.0
3.3  2.7  2.7
[...]
```

Beispiel 2; Spalten durch Leerzeichen getrennt, Komma als Dezimaltrennzeichen:

```
"Seminar A"  "Seminar B"  "Seminar C"
1,3  1,7  1,0
3,3  2,7  2,7
[...]
```

Beispiel 3; Spalten durch Kommata getrennt, Punkt als Dezimaltrennzeichen:

```
"Seminar A", "Seminar B", "Seminar C"
1.3, 1.7, 1.0
3.3, 2.7, 2.7
[...]
```

Beispiel 4; Spalten durch Semikola getrennt, Punkt als Dezimaltrennzeichen:

```
"Seminar A"; "Seminar B"; "Seminar C"
1.3; 1.7; 1.0
3.3; 2.7; 2.7
[...]
```

Beispiel 5; Spalten durch Tab-Stopps getrennt, Komma als Dezimaltrennzeichen:

```
"Seminar A"	"Seminar B"	"Seminar C"
1,3	1,7	1,0
3,3	2,7	2,7
[...]
```

Spalten, die nicht numerische Werte enthalten, werden beim Import von R automatisch in einen sogenannten factor umgewandelt, d.h. als kategoriale Variable behandelt (s. Abschnitt 4.2.3). Um dies zu verhindern, muss die Option stringsAsFactors auf FALSE gesetzt werden.

Möchte man nicht immer den vollständigen Dateipfad eingeben müssen, kann der Befehl setwd() verwendet werden, dem als Argument das Verzeichnis angegeben wird, aus dem Daten eingelesen werden sollen:

```
> setwd("C:/daten")
> dat <- read.table("data.csv")
```

Dies ist beispielsweise praktisch, wenn mehrere Dateien, die in einem Verzeichnis liegen, geladen werden sollen. Die Option beeinflusst nicht nur den Suchpfad für zu ladende Dateien, sondern bestimmt das aktuelle Arbeitsverzeichnis. Wird in einem Befehl der Verzeichnispfad nicht angegeben, sucht oder speichert R alle Dateien in diesem Verzeichnis.

Um Dateien einzulesen, die mit Statistikprogrammen wie beispielsweise SPSS oder Stata erstellt wurden, kann das Paket foreign verwendet werden. Für mit SPSS abgespeicherte Daten bietet es den Befehl read.spss und mit Stata erzeugte Datensätze können mit read.dta geöffnet werden. Dieses Thema wird auf den übernächsten Abschnitt verschoben, da wir uns vorher noch anschauen wollen, wie man Daten betrachten und bearbeiten kann.

3.2 Daten betrachten und bearbeiten

3.2.1 Daten anzeigen

Wir gehen nun davon aus, dass ein Datensatz geladen wurde. Als Beispiel verwenden wir die Klausurdaten aus Tabelle 3.1 und laden sie mit folgendem Befehl:

```
> dat <- read.table("C:/daten/data.csv",header=TRUE)
```

Nun befindet sich ein Objekt mit dem Namen dat im Speicher. Die Klasse dieses Objektes ist data.frame, was darauf hinweist, dass es sich bei diesem Objekt um einen Datensatz handelt:

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

Rufen wir dieses Objekt auf, bekommen wir die Daten angezeigt:

```
> dat
  Seminar.A Seminar.B Seminar.C
1       1.3       1.7       1.0
2       3.3       2.7       2.7
3       2.7       4.0       3.7
4       1.7       1.3       2.0
5       1.3       1.3       1.3
```

Zunächst fällt auf, dass die Leerzeichen in den Variablennamen durch Punkte ersetzt wurden. Dies ist den in Abschnitt 2.3 beschriebenen Namensregeln geschuldet. Insbesondere wird auch

hier zwischen Groß- und Kleinschreibung unterschieden. Zusätzlich zu den eigentlichen Datenspalten, die mit den Variablennamen überschrieben sind, wird in der ersten Spalte die Nummer der Datenzeile angezeigt.

Gerade bei sehr umfangreichen Datensätzen, die sowohl eine sehr große Zahl an Beobachtungen als auch eine große Zahl an Variablen aufweisen können, empfiehlt es sich nicht, wie im obigen Beispiel den kompletten Datensatz anzeigen zu lassen. Stattdessen könnte man sich auf einige Variablen oder Beobachtungen beschränken. Hierfür gibt es mehrere Möglichkeiten. Der Befehl `str()` zeigt allgemeine Informationen und einen Auszug aus den Daten an:

```
> str(dat)
'data.frame':   5 obs. of  3 variables:
 $ Seminar.A: num  1.3 3.3 2.7 1.7 1.3
 $ Seminar.B: num  1.7 2.7 4 1.3 1.3
 $ Seminar.C: num  1 2.7 3.7 2 1.3
```

Zunächst ist die Klasse von `dat` angegeben, sowie die Zahl der Beobachtungen (`obs.` steht für *observations*) und die Zahl der Variablen. In den folgenden Zeilen finden sich jeweils der Name einer Variable, Angaben zur Datenart (hier: `num` für numerisch) sowie die ersten zehn Werte, die diese Variable im Datensatz aufweist. Da der hier verwendete Datensatz lediglich fünf Fälle aufweist, werden auch nur diese angezeigt.

Um gezielt einzelne Fälle oder Variablen aufzurufen, kann eine direkte Indizierung verwendet werden. Hierbei werden Beobachtungen und Variablen über Nummern angezeigt. Beobachtungen werden über die bereits weiter oben vorgekommenen Nummern von Datenzeilen angesprochen. Die erste Beobachtung hat die Nummer 1, die zweite Beobachtung die Nummer 2 und so fort. Variablen werden ebenfalls einfach durchnummeriert, so dass die Variable, die der ersten Datenspalte entspricht beispielsweise die Nummer 1 hat. Bei den Klausurdaten ist dies die Variable `Seminar.A`. Die Variable `Seminar.B` hat die Nummer 2 und die Variable `Seminar.C` hat die Nummer 3.

Zum Aufruf einzelner Beobachtungen oder Variablen werden die Nummern direkt hinter den Objektnamen der Daten geschrieben, und zwar in der Form `[i,j]`, wobei `i` die Zeilennummer und `j` die Spaltennummer darstellt. Die Note des ersten Studenten in Seminar A kann man beispielsweise aufrufen über:

```
> dat[1,1]
[1] 1.3
```

Wird nur eine Zeilennummer angegeben, werden alle Werte der entsprechenden Beobachtung ausgegeben:

```
> dat[1,]
  Seminar.A Seminar.B Seminar.C
1       1.3       1.7         1
```

Hierbei ist darauf zu achten, dass das Komma, das Zeilen- und Spaltennummer trennt, auf jeden Fall angegeben werden muss. Auf gleiche Weise kann man sich auch alle Werte einer Variablen anzeigen lassen:

```
> dat[,1]
[1] 1.3 3.3 2.7 1.7 1.3
```

Anstelle einzelner Zeilen- oder Spaltennummern kann man auch mehrere Nummern angeben, wobei hierfür Vektoren verwendet werden. Möchte man sich beispielsweise die zweite und die dritte Beobachtung anzeigen lassen, wird hinter den Objektnamen [2:3,] gesetzt:

```
> dat[2:3,]
  Seminar.A Seminar.B Seminar.C
2       3.3       2.7       2.7
3       2.7       4.0       3.7
```

Ganz analog lassen sich nur Klausurnoten in den Seminaren A und B anzeigen:

```
> dat[,1:2]
  Seminar.A Seminar.B
1       1.3       1.7
2       3.3       2.7
3       2.7       4.0
4       1.7       1.3
5       1.3       1.3
```

Eine einfache Möglichkeit, sich die ersten Beobachtungen in einem Datensatz anzuschauen, bietet der Befehl head(), welcher immer die ersten sechs Fälle anzeigt. Analog hierzu zeigt der Befehl tail() die letzten sechs Fälle an.

Einzelne Variablen können zudem aufgerufen werden, indem hinter den Namen des Objekts ein $ gefolgt vom Namen der Variable gesetzt wird:

```
> dat$Seminar.B
[1] 1.7 2.7 4.0 1.3 1.3
```

Die Werte einzelner Beobachtungen erreicht man durch eine Indizierung wie bei Vektoren:

```
> dat$Seminar.B[2]
[1] 2.7
> dat$Seminar.B[-1]
[1] 2.7 4.0 1.3 1.3
```

Eine Verkürzung lässt sich durch das Anwenden des Befehls attach auf den Datensatz erreichen:

```
> attach(dat)
```

Nun können die Variablen direkt über ihre Namen abgerufen werden:

```
> Seminar.B
[1] 1.7 2.7 4.0 1.3 1.3
```

Möchte man die Wirkung von attach rückgängig machen, wird der Befehl detach verwendet:

```
> detach(dat)
> Seminar.B
Fehler: Objekt 'Seminar.B' nicht gefunden
```

Bei der Verwendung des Befehls attach sollte beachtet werden, dass die Variablen zwar über ihre Namen aufgerufen werden können, aber immer noch ein „Teil“ des Datensatzes und keine eigenständigen Objekte sind. Wenn bereits vor dem Ausführen des Befehls ein Objekt im Workspace ist, welches den selben Namen wie eine Variable hat, führt attach zu einem Namenskonflikt:

```
> ls()
[1] "dat"
> Seminar.B <- 1
> ls()
[1] "dat"         "Seminar.B"
> attach(dat)

        The following object(s) are masked _by_ .GlobalEnv :

         Seminar.B

> Seminar.B
[1] 1
> dat$Seminar.B
[1] 1.7 2.7 4.0 1.3 1.3
```

Das Objekt Seminar.B, welches sich bereits vor dem Ausführen des Befehls im Speicher befand, wird nicht überschrieben und bleibt weiterhin über den vergebenen Namen abrufbar, während sich hingegen die Variable Seminar.B aus dem Datensatz nicht direkt über ihren Namen aufrufen lässt.

3.2.2 Daten ändern

Bei einem wie im vorherigen Unterabschnitt beschriebenen Fall ist es nützlich, die Namen der Variablen zu ändern. Die Namen der Variablen, die in einem Datensatz vorkommen, lassen sich über den Befehl names() aufrufen:

```
> names(dat)
[1] "Seminar.A" "Seminar.B" "Seminar.C"
```

Das Ergebnis ist ein Vektor, dessen einzelne Einträge den Namen entsprechen und der wie im Abschnitt zu Vektoren beschrieben indiziert werden kann:

```
> names(dat)[2]
[1] "Seminar.B"
```

Zum Ändern von Variablenbezeichnungen werden nun einfach Zuweisungen auf diesen Vektor verwendet, wobei zu beachten ist, dass der neue Variablenname in Anführungszeichen stehen muss:

```
> names(dat)[2] <- "Y"
> names(dat)
[1] "Seminar.A"   "Y"   "Seminar.C"
```

Hierbei muss der Befehl attach nicht mehrmals angewendet werden, um eine Variable über ihren neuen Namen anzusprechen. Möchte man mehrere oder alle Variablennamen ändern, müssen diese Namen als Vektor angegeben werden:

```
> names(dat) <- c("X","Y","Z")
> names(dat)
[1] "X" "Y" "Z"
```

Wohlgemerkt wird hierüber nur das Datenobjekt im Workspace geändert, nicht jedoch die eigentliche Datei, die geladen wurde. Gleiches gilt grundsätzlich auch für die Veränderung der Werte von Variablen.

Möchte man Werte einzelner Variablen umkodieren, spielen wieder Zuweisungen eine Rolle. Angenommen, man würde gerne die Klausurnoten aus Seminar A auf die nächste ganze Note runden. Dann könnte man den Befehl round() auf die Variable Seminar.A anwenden und dieses wiederum der Variable selbst zuweisen. Wurde der Befehl attach auf die Daten angewandt, würde dies folgendermaßen aussehen:

```
> dat$Seminar.A <- round(Seminar.A)
```

Die rechte Seite der Zuweisung erzeugt einen Vektor, der die gerundeten Klausurergebnisse aus Seminar A enthält. Beim Ziel der Zuweisung ist abermals zu beachten, dass über den Befehl attach keine eigenständigen Objekte erzeugt wurden. Eine einfache Zuweisung auf ein Objekt Seminar.A würde ein neues Objekt mit diesen Namen erzeugen, welches nicht zum Datensatz gehört – allerdings wie oben beschrieben über den Namen Seminar.A aufrufbar wäre. Dieser Fall wäre interessant, wenn man den Datensatz aus bestimmten Gründen noch in seiner ursprünglichen Fassung im Speicher haben möchte. Um im weiteren Verlauf Uneindeutigkeiten zu vermeiden, wird grundsätzlich die Schreibweise datensatz$variable verwendet und von der Verwendung des Befehls attach abgesehen.

Die Veränderung von Variablenwerten kann man auf solche Werte beschränken, die bestimmte Bedingungen erfüllen. Hierfür werden eckige Klammern und logische Operatoren verwendet. Möchte man beispielsweise nur die Noten in Seminar B bei Beobachtungen verändern, die in diesem Seminar die Note $1,3$ haben, wird wie folgt vorgegangen:

```
> dat$Seminar.B
[1] 1.7 2.7 4.0 1.3 1.3
> dat$Seminar.B[dat$Seminar.B==1.3] <- 1.0
> dat$Seminar.B
[1] 1.7 2.7 4.0 1.0 1.0
```

Die Noten der Beobachtungen, die den Wert $1,3$ aufweisen, wurden also auf den Wert $1,0$ gesetzt. Dabei liefert die Abfrage in den eckigen Klammern einen Vektor, der Wahrheitswerte enthält:

```
> dat$Seminar.B==1.3
[1] FALSE FALSE FALSE  TRUE  TRUE
```

Nur für die Beobachtungen, für die der Wert TRUE resultiert – also Beobachtungen Nummer 4 und 5, die mit den entsprechenden Einträgen dieses Vektors korrespondieren – wird der Variablenwert geändert.

3.2.3 Faktoren

Möchte man, dass für eine Variable keine numerischen Werte, sondern Text angezeigt wird, kann der Befehl factor verwendet werden. Beispielsweise könnte man sich anstatt der Drittelnoten ausgeschriebene Schulnoten – „Sehr Gut“, „Gut“ usw. – anzeigen lassen. Dies ändern wir nun für die Variable Seminar.A:

```
> dat$Seminar.A <- factor(dat$Seminar.A, labels=
+ c("Sehr Gut","Gut","Befriedigend","Ausreichend"))
```

Wie leicht ersichtlich ist, wurde der Befehl auf zwei Zeilen aufgeteilt. Dem Befehl wird zunächst die Variable angegeben, deren Werte als Text angezeigt werden sollen. Aus dieser Variable zieht der Befehl lediglich die vorhandenen Werte – in diesem Fall also 1.3, 1.7, 2.7 und 3.3. Das zweite Argument ist ein Vektor der den Text der Faktoren enthält. Dabei werden die einzelnen Werte des Vektors der Reihe nach den der Größe nach geordneten Werten der Variablen zugeordnet – also "Sehr Gut" zu 1.3, "gut" zu 1.7 und so fort. Nun wird bei Aufruf der Daten – und auch beispielsweise bei deskriptiven Auswertungen – der Text anstelle des numerischen Wertes angezeigt:

```
> dat[1,]
  Seminar.A Seminar.B Seminar.C
1  Sehr Gut       1.7         1
> dat$Seminar.A
[1] Sehr Gut      Ausreichend   Befriedigend Gut           Sehr Gut
Levels: Sehr Gut Gut Befriedigend Ausreichend
```

Der erste Aufruf zeigt alle Merkmalswerte der ersten Beobachtung an. Bei der Variablen Seminar.A wird anstatt des numerischen Wertes nun der spezifizierte Text angezeigt, während für die anderen beiden Variablen die numerischen Werte beibehalten wurden. Der zweite Aufruf listet die Werte einzelner Beobachtungen bei der Variable Seminar.A der Reihe nach auf. Zusätzlich ist noch eine mit Levels beschriebene Zeile vorhanden, die die spezifizierten Faktorwerte auflistet.

3.2.4 Fehlende Werte

Fehlende Werte werden in R durch die Zeichenkombination NA kenntlich gemacht. NA steht hierbei für *not available*. In Zuweisungen kann dieses Symbol wie andere auch verwendet werden:

```
> dat$Seminar.B[1] <- NA
> dat$Seminar.B
[1]  NA 2.7 4.0 1.0 1.0
```

In diesem Beispiel wurde der Wert der Variablen Seminar.B der ersten Beobachtung auf NA gesetzt. Bei der Kombination von logischen Operatoren und NAs ist allerdings Vorsicht geboten:

```
> dat$Seminar.B[1]
[1] NA
> dat$Seminar.B[1] == NA
[1] NA
> NA == NA
[1] NA
```

Zunächst lassen wir uns den Wert der ersten Beobachtung bei der Variablen Seminar.B anzeigen. Anschließend soll überprüft werden, ob dieser Wert gleich NA ist. Anstatt eines Wahrheitswertes erhält man allerdings wieder ein NA als Ergebnis. Problematisch wird dies, wenn beispielsweise Variablenwerte über logische Abfragen geändert werden sollen:

```
> dat$Seminar.B
[1]  NA 2.7 4.0 1.3 1.3
> dat$Seminar.B[dat$Seminar.B==NA] <- 1.0
> dat$Seminar.B
[1]  NA 2.7 4.0 1.3 1.3
```

Der Wert der ersten Beobachtung sollte eigentlich wieder auf den Wert 1.0 gesetzt werden. Da aber die Angabe in den eckigen Klammern zu keinem Wahrheitswert führt, bleibt die Note unverändert auf NA gesetzt. Um dennoch eine Veränderung zu erreichen, muss der Befehl is.na() angewandt werden:

```
> dat$Seminar.B[is.na(dat$Seminar.B)==TRUE] <- 1
> dat$Seminar.B
[1] 1.0 2.7 4.0 1.3 1.3
```

Dieser Befehl liefert Wahrheitswerte, die dann mittels logischen Operatoren innerhalb einer Bedingung verwendet werden können:

```
> NA == NA
[1] NA
> is.na(NA)
[1] TRUE
```

3.2.5 Neue Variablen erstellen

Das erstellen neuer Variablen erfolgt einfach über Zuweisungen. Man möchte beispielsweise den Notendurchschnitt der Noten in den Seminaren B und C berechnen. Diese neue Variable soll den Namen durchschnitt bekommen. Um anzuzeigen, dass diese Variable zu unserem Datenobjekt gehören soll, schreiben wir dat$durchschnitt und geben die gerade genannte Berechnung ein:

```
> dat$durchschnitt <- (dat$Seminar.B+dat$Seminar.C)/2
> dat$durchschnitt
```

```
[1] 1.35 2.70 3.85 1.65 1.30
> dat
  Seminar.A Seminar.B Seminar.C durchschnitt
1       1.3       1.7       1.0        1.35
2       3.3       2.7       2.7        2.70
3       2.7       4.0       3.7        3.85
4       1.7       1.3       2.0        1.65
5       1.3       1.3       1.3        1.30
```

Das Objekt dat enthält nun eine entsprechende Datenspalte. Bei der Berechnung der Werte dieser Variablen werden Seminar.B und Seminar.C wie Vektoren behandelt, die angegebenen Rechenoperationen also elementweise durchgeführt. Dass Ergebnis dieser Berechnung wird dann in die neue Datenspalte gefüllt.

Anstelle einer Berechnungsvorschrift können auch direkt Werte eingegeben werden, wobei hierfür Vektoren verwendet werden. Beispielsweise könnte man eine Variable erstellen, die die Alter der einzelnen Studenten erfasst. Dann ließe sich wie folgt vorgehen:

```
> dat$alter <- c(23,20,21,30,25)
```

Die neu erstellte Variable trägt den sinnfälligen Namen alter und weist für den ersten Studenten den Wert 23 auf, für den zweiten Studenten den Wert 20 und so fort.

3.3 Daten importieren

3.3.1 SPSS und Stata

Zum importieren von Daten, die in speziellen Formaten, wie dem SPSS .sav-Format oder dem von Stata benutzten .dta-Format, vorliegen, kann wie bereits erwähnt das Paket foreign verwendet werden, welches für diese beiden Formate die Befehle read.spss() und read.dta() bereitstellt. Zudem werden noch weitere Formate unterstützt. Als Beispiel wird hier ein SPSS-Datensatz eingelesen. Verwendet wird das Campus-File des Mikrozensus 2002, welches frei zugänglich unter www.forschungsdatenzentrum.de heruntergeladen werden kann.

Die Datei heißt mz02_cf.sav, wobei die Dateiendung .sav auf den von SPSS verwendeten Dateitypus hinweist. Diese sei im Verzeichnis C:\daten abgelegt. Zunächst laden wir das Paket foreign, welches nicht zusätzlich installiert werden muss, und legen unser Arbeitsverzeichnis fest:

```
> library(foreign)
> setwd("C:/daten")
```

Eine einfache Möglichkeit, die Daten einzulesen, besteht darin lediglich den Dateinamen an den Befehl zu geben:

```
dat <- read.spss("mz02_cf.sav")
```

Nun betrachten wir, wie die geladenen Daten aussehen und beschränken uns hierfür auf die erste Beobachtung und die ersten drei Variablen:

```
> dat[1,1:3]
Fehler in dat[1, 1:3] : falsche Anzahl von Dimensionen
```

Hier hat sich anscheinend ein Problem eingeschlichen:

```
> class(dat)
[1] "list"
```

Anscheinend sind die Daten nicht in der richtigen Klasse data.frame abgespeichert. Um dies zu erreichen, muss beim read.spss Befehl noch die Option to.data.frame=TRUE angegeben werden. Davor löschen wir allerdings die bereits geladenen Daten aus dem Speicher:

```
> rm(dat)
> dat <- read.spss("mz02_cf.sav",to.data.frame=T)
> class(dat)
[1] "data.frame"
> dat[1,1:3]
                  EF1 EF3 EF4
1 Schleswig-Holstein   1   1
```

Die Dateien sind nun mit der richtigen Klasse abgespeichert und wir können die Werte der ersten drei Variablen im Datensatz für die erste Beobachtung abrufen. Die Variable EF1 gibt das Bundesland wider, in der die befragte Person lebt, EF3 und EF4 enthalten Informationen zur Haushalts- beziehungsweise Personenidentifikation. Bei EF1 fällt auf, dass kein numerischer Wert, sondern Text angegeben wird, während dies bei EF3 und EF4 nicht der Fall ist:

```
> class(dat$EF1)
[1] "factor"
> class(dat$EF3)
[1] "numeric"
```

Anscheinend ist also EF1 als Faktor gespeichert. Die Ursache hierfür ist relativ einfach. In SPSS besteht die Möglichkeit numerischen Werten einer Variable sogenannte Labels zuzuweisen, ähnlich den Faktoren in R. Beim einlesen in R werden diese Wertelabels in Faktoren umgewandelt. Möchte man stattdessen aber numerische Werte haben, kann man entweder einzelne Variablen entsprechend bearbeiten, oder aber beim importieren der Datei beim Befehl read.spss() die Option use.value.labels=FALSE angeben:

```
> rm(list=ls())
> dat <- read.spss("mz02_cf.sav",to.data.frame=T,use.value.labels=F)
> dat[1,1:3]
  EF1 EF3 EF4
1   1   1   1
> class(dat$EF1)
[1] "numeric"
> class(dat$EF3)
[1] "numeric"
```

Nun sind auch die Werte der Variablen EF1 direkt numerisch aufrufbar.

3.3.2 Excel

In MS Excel oder ähnlichen Programmen erstellte Dateien mit der Endung .xls lassen sich nicht mit dem bisher vorgestellten Vorgehen öffnen. Eine Möglichkeit, dennoch an die Daten zu kommen, besteht darin, solche Daten mit MS Excel oder ähnlicher Software als .csv-Datei zu exportieren und über den read.table-Befehl einzulesen.

Eine andere Möglichkeit erlaubt das Einlesen von .xls-Dateien über die Zwischenablage. In diese werden Inhalte abgelegt, die zum kopieren markiert wurden – also indem die Inhalte erst ausgewählt wurden und anschließend beispielsweise die Tastenkombination Strg C gedrückt wurde. Wurden die im letzten Satz genannten Schritte in einer Excel-Datei ausgeführt, reicht folgender Befehlsaufruf, um diese Daten zu laden:

```
> dat <- read.table(file="clipboard",sep="\t",header=T)
```

Diese Möglichkeit besteht allerdings nur unter MS Windows. Unter Unix-Systemen wird der Inhalt des R-Clipboards geladen – also Inhalte die innerhalb von R zum kopieren markiert wurden.

Mittlerweile gibt es auch einige Erweiterungspakete, die es ermöglichen direkt auf Excel-Dateien zuzugreifen, wie beispielsweise das nur für Windows verfügbare Paket xlsReadWrite, welches Excel-Dateien lesen und schreiben kann.

3.4 Daten & Ergebnisse speichern

3.4.1 Objekte speichern

Hat man Objekte erstellt und möchte diese so speichern, dass R sie wieder direkt in den Workspace laden kann, können die Befehle save() und load() verwendet werden. Dem Befehl save() übergibt man als Argumente die Objekte, die gespeichert werden sollen, sowie unter der Option file die Datei, in der diese abgelegt werden sollen.

Beispielsweise hat man zwei Objekte erzeugt, die abgespeichert werden sollen:

```
> a <- 2
> b <- 3
```

Möchte man diese im Verzeichnis C:\daten in einer Datei objekte.rda speichern, ist folgender Aufruf nötig:

```
> save(a,b,file="C:/daten/objekte.rda")
```

Dabei ist die Dateiendung beliebig wählbar. Zum Laden der Objekte werden einfach Dateiname und Pfad an den Befehl load() übergeben:

```
> rm(list=ls())
> a
Fehler: objekt "a" nicht gefunden
> b
Fehler: objekt "b" nicht gefunden
```

```
> load("C:/daten/objekte.rda")
> a
[1] 2
> b
[1] 3
```

3.4.2 Daten speichern

Hat man Daten, die als Objekt der Klasse data.frame geladen sind, bearbeitet und möchte diese Abspeichern, lässt sich dies einfach über den Befehl write.table() erreichen. Diesem wird einfach ein data.frame-Objekt sowie eine Pfadangabe übergeben. Sei beispielsweise ein Datensatz als Objekt dat geladen. Dieser soll unter C:\daten\daten-neu.dat abgespeichert werden:

```
> write.table(dat, file="C:/daten/daten-neu.dat")
```

Anschließend können diese Daten einfach wieder mit read.table aufgerufen werden:

```
> dat <- read.table("C:/daten/daten-neu.dat")
```

Mit dem Befehl write.table erzeugte Dateien lassen sich relativ unproblematisch auch von anderen Statistik-Programmen verwenden: Es handelt sich um Textdateien, bei denen in der Standardeinstellung einzelne Spalten durch Leerzeichen getrennt sind und ein Punkt als Dezimaltrennzeichen verwendet wird, wie bei Beispiel 1 in Abbildung 3.1. Solche Dateien lassen sich mit jedem Statistik-Programm öffnen.

4 Deskriptive Statistik

4.1 Tabellen

4.1.1 Tabellen erstellen

Für dieses Kapitel wird angenommen, dass die Daten aus Tabelle 3.1 geladen sind. Sind diese im Verzeichnis C:\daten im csv-Format mit Leerzeichen als Spaltentrenner und Punkten als Dezimaltrennzeichen abgelegt, lassen sie sich wie bereits gezeigt aufrufen:

```
> setwd("C:\daten")
> dat <- read.table("noten.csv",header=T)
```

Zunächst wollen wir uns eine Übersicht über die Verteilung der Noten in Seminar A verschaffen. Hierfür verwenden wir den Befehl table(), dem wir als Argument den Namen der betreffenden Variable angeben:

```
> table(dat$Seminar.A)

1.3 1.7 2.7 3.3
  2   1   1   1
```

Als Ergebnis werden zwei Zeilen ausgegeben. In der ersten Zeile sind die auftretenden Merkmalswerte festgehalten, in der zweiten Zeile die jeweilige absolute Häufigkeit des Auftretens. Die Note $1,3$ liegt bei zwei Studenten vor, die Note $1,7$ bei einem Studenten und so weiter. Das gleiche Vorgehen lässt sich auch für die anderen Variablen anwenden, beispielsweise:

```
> table(dat$Seminar.C)

  1 1.3   2 2.7 3.7
  1   1   1   1   1
```

Wurden Faktoren zur Benennung auftretender Werte verwendet, werden diese anstatt der numerischen Werte angezeigt:

```
> dat$Seminar.A <- factor(dat$Seminar.A,labels =
+ c("Sehr Gut","Gut","Befriedigend","Ausreichend"))
> table(dat$Seminar.A)

    Sehr Gut     Gut Befriedigend     Ausreichend
           2       1             1               1
```

Weist eine Variable viele Ausprägungen auf, kann es unter Umständen sinnvoll sein, nicht alle Ausprägungen zu tabellieren. Dies wird über das Argument exclude erreicht:

```
> table(dat$Seminar.A, exclude="Gut")

     Sehr Gut Befriedigend   Ausreichend
             2             1             1
> table(dat$Seminar.C, exclude="1.3")

  1   2 2.7 3.7
  1   1   1   1
```

In diesen Beispielen wurden die Ausprägungen „Gut“ und „1,3“ ausgespart.

Werden zwei Variablen als Argumente angegeben, wird eine Kreuztabelle erstellt:

```
> table(dat$Seminar.A, dat$Seminar.C)

                1 1.3 2 2.7 3.7
  Sehr Gut      1   1 0   0   0
  Gut           0   0 1   0   0
  Befriedigend  0   0 0   0   1
  Ausreichend   0   0 0   1   0
```

Hieraus ist beispielsweise ersichtlich, dass ein Student in Seminar A die Note „Sehr Gut“ und in Seminar B die Note „1“ hat. Die Merkmalsausprägungen der als erstes Argument angegebenen Variable befinden sich also in der ersten Spalte, die Ausprägungen der zweiten Variable in der ersten Zeile. Die Namen der Variablen können über das Argument `dnn` hinzugefügt werden:

```
> table(dat$Seminar.A, dat$Seminar.C, dnn=c("Seminar A","Seminar B"))
             Seminar B
Seminar A       1 1.3 2 2.7 3.7
  Sehr Gut      1   1 0   0   0
  Gut           0   0 1   0   0
  Befriedigend  0   0 0   0   1
  Ausreichend   0   0 0   1   0
```

Mit dem Befehl `table()` erstellte Tabellen lassen sich über die Verwendung des Zuweisungsoperators auch als eigenständige Objekte abspeichern:

```
> tab1 <- table(dat$Seminar.A, dat$Seminar.C, dnn=c("Seminar A","Seminar B"))
> class(tab1)
[1] "table"
> tab1
             Seminar B
Seminar A       1 1.3 2 2.7 3.7
  Sehr Gut      1   1 0   0   0
  Gut           0   0 1   0   0
  Befriedigend  0   0 0   0   1
  Ausreichend   0   0 0   1   0
```

Im obigen Beispiel wird die Kreuztabelle der Variablen `Seminar.A` und `Seminar.B` unter dem Namen `tab1` im Workspace abgelegt, wobei dieses Objekt die Klasse `table` aufweist. Durch Eingabe des Objektnamens kann dieses wie gewohnt aufgerufen werden.

Das Erstellen eines Objektes ist von Vorteil bei der weiteren Bearbeitung und Verwendung von (Kreuz-)Tabellen. Ein Tabellenobjekt lässt sich beispielsweise als Argument für den Befehl prop.table() verwenden, der die bisher in den Tabellen enthaltenen absoluten Häufigkeiten in relative Häufigkeiten umwandelt:

```
> prop.table(tab1)
                Seminar B
Seminar A          1 1.3   2 2.7 3.7
  Sehr Gut       0.2 0.2 0.0 0.0 0.0
  Gut            0.0 0.0 0.2 0.0 0.0
  Befriedigend   0.0 0.0 0.0 0.0 0.2
  Ausreichend    0.0 0.0 0.0 0.2 0.0
```

Beispielsweise ist nun ersichtlich, dass 20% der Beobachtungen in Seminar A die Note „Sehr Gut“ und in Seminar B die Note 1 aufweisen. Der prop.table() Befehl lässt sich auch direkt mit dem table()-Befehl kombinieren:

```
> prop.table(table(dat$Seminar.C))

  1 1.3   2 2.7 3.7
0.2 0.2 0.2 0.2 0.2
```

Bei der aus dem Objekt tab1 erstellten Kreuztabelle mit relativen Häufigkeiten summieren sich alle Einträge auf den Wert 1. Um die Summe aller Tabelleneinträge zu generieren, lässt sich der Befehl margin.table() verwenden:

```
> margin.table(prop.table(tab1))
[1] 1
```

In manchen Fällen interessieren aber eventuell nur bedingte Verteilungen. Beispielsweise könnte man sich für die Häufigkeit bestimmter Noten im Seminar B interessieren, bezogen auf alle Fälle, die in Seminar A die Note „Sehr Gut“ aufweisen. In diesem Fall ergeben die Zeilensummen der Tabelle den Wert 1. Dies lässt sich auch umdrehen, so dass die Noten in Seminar A in Abhängigkeit der Noten aus Seminar B betrachtet werden, so dass die Spaltensummen der Tabelle 1 ergeben. In beiden Fällen wird das Argument margin verwendet. Sollen die Zeilensummen den Wert 1 ergeben wird margin=1 angegeben, im zweiten Fall wird margin=2 benutzt:

```
> prop.table(tab1,margin=1)
                Seminar B
Seminar A          1 1.3   2 2.7 3.7
  Sehr Gut       0.5 0.5 0.0 0.0 0.0
  Gut            0.0 0.0 1.0 0.0 0.0
  Befriedigend   0.0 0.0 0.0 0.0 1.0
  Ausreichend    0.0 0.0 0.0 1.0 0.0
```

In diesem Beispiel ergeben die Zeilensummen den Wert 1. Aus dieser Tabelle ist dann unter anderem zu ersehen, dass von den Personen, die in Seminar A die Note „Sehr Gut“ haben, 50% in Seminar B die Note 1 und ebenfalls 50% die Note $1,3$ haben. Verwendet man nun den Befehl margin.table() erhält man folgendes Ergebnis:

```
> margin.table(prop.table(tab1,margin=1))
[1] 4
```

Dieses Ergebnis ergibt sich, da es bei den Noten in Seminar A vier Ausprägungen gibt, für die sich die relativen Häufigkeiten jeweils auf 1 summieren. Dies lässt sich ebenfalls mit dem margin.table()-Befehl überprüfen, indem ganz analog zu prop.table() das Argument margin angegeben wird:

```
> margin.table(prop.table(tab1,margin=1),margin=1)
Seminar A
     Sehr Gut            Gut Befriedigend   Ausreichend
            1              1              1              1
```

Zunächst wird über den Befehl prop.table() unter Angabe von margin=1 eine Kreuztabelle mit relativen Häufigkeiten erstellt, wobei sich diese zeilenweise zu 1 aufsummieren. Über den Befehl margin.table() werden die Summen der Einträge der Tabelle gebildet, wobei das Argument margin=1 dazu führt, dass dies zeilenweise geschieht.

Die Summen der einzelnen Spalten und Zeilen einer Tabelle lassen sich auch direkt mit dieser ausgeben. Hierfür kann der Befehl addmargins() benutzt werden:

```
> addmargins(tab1)
                Seminar B
Seminar A         1 1.3 2 2.7 3.7 Sum
  Sehr Gut        1   1 0   0   0   2
  Gut             0   0 1   0   0   1
  Befriedigend    0   0 0   0   1   1
  Ausreichend     0   0 0   1   0   1
  Sum             1   1 1   1   1   5
```

Hier wird wieder die Tabelle mit den absoluten Häufigkeiten verwendet. In der mit Sum überschriebenen Spalte finden sich die Zeilensummen und in der untersten, ebenfalls mit Sum bezeichneten Zeile finden sich die Spaltensummen. Der Eintrag ganz unten rechts in der Tabelle mit dem Wert 5 gibt die Summe aller Einträge der Tabelle wieder.

Sollen nur Spalten- oder nur Zeilensummen ausgegeben werden, kann das bereits bekannte Argument margin wieder verwendet werden:

```
> addmargins(tab1,margin=1)
                Seminar B
Seminar A         1 1.3 2 2.7 3.7
  Sehr Gut        1   1 0   0   0
  Gut             0   0 1   0   0
  Befriedigend    0   0 0   0   1
  Ausreichend     0   0 0   1   0
  Sum             1   1 1   1   1
> addmargins(tab1,margin=2)
                Seminar B
Seminar A         1 1.3 2 2.7 3.7 Sum
  Sehr Gut        1   1 0   0   0   2
  Gut             0   0 1   0   0   1
```

```
  Befriedigend 0   0 0   0   1   1
  Ausreichend  0   0 0   1   0   1
```

Bei den bisher erzeugten Tabellen sind viele Einträge gleich 0. Um die Lesbarkeit zu verbessern, können diese Einträge mit einem Punkt oder einem anderen Symbol ersetzt werden. Hierfür wird dem Befehl print() als Argument eine Tabelle mit absoluten Häufigkeiten angegeben und als weiteres Argument zero.print spezifiziert:

```
> print(tab1,zero.print=".")
             Seminar B
Seminar A      1 1.3 2 2.7 3.7
  Sehr Gut     1   1 .   .   .
  Gut          .   . 1   .   .
  Befriedigend .   . .   .   1
  Ausreichend  .   . .   1   .
```

Der print()-Befehl dient allgemein dazu, Objekte anzuzeigen, und wird immer aufgerufen, wenn einfach der Name eines Objektes zum anzeigen desselben eingegeben wird. Diese letztere Möglichkeit ist praktisch eine Abkürzung, die die Bedienung von R erleichtern soll. Würde man also print(tab1) eingeben, wäre das Resultat dasselbe wie bei der Eingabe von tab1. Allerdings bietet der print()-Befehl je nach aufzurufendem Objekt noch weitere Optionen, die sich nur bei direktem Aufruf dieses Befehls spezifizieren lassen. Beim aktuellen Beispiel ist dies die Option zero.print, die für Tabellen zur Verfügung steht.

Das obige Beispiel lässt sich auch mit dem addmargins()-Befehl kombinieren:

```
> print(addmargins(tab1,margin=1),zero.print="-")
             Seminar B
Seminar A      1 1.3 2 2.7 3.7
  Sehr Gut     1   1 -   -   -
  Gut          -   - 1   -   -
  Befriedigend -   - -   -   1
  Ausreichend  -   - -   1   -
  Sum          1   1 1   1   1
```

Die korrekte Anzeige von Zeilen- oder Spaltensummen ist also durch die Aussparung der Nullen nicht betroffen.

4.1.2 Tabellen exportieren

Tabellen, die mit den im letzten Unterabschnitt vorgestellten Befehlen erzeugt wurden, sollen in der Regel in HTML- oder Textdokumenten weiterverarbeitet werden, was weitere Formatierung nötig macht. Wird LaTeX als Textsatzsystem benutzt oder soll HTML eingesetzt werden, kann das Paket xtable zur Erzeugung von formatierten Tabellen verwendet werden. Dieses muss zunächst installiert und geladen werden:

```
> install.packages("xtable")
> library(xtable)
```

Dieses Paket stellt den Befehl xtable() zur Verfügung. Diesem können eine Vielzahl an verschiedenen Objekten wie Tabellen übergeben werden. Als Output erhält man zunächst LaTeX-Syntax. Wendet man diesen Befehl auf die im vorherigen Unterabschnitt unter dem Namen tab1 erzeugte Tabelle an, erhält man folgendes Ergebnis:

```
> xtable(tab1)
% latex table generated in R 2.10.1 by xtable 1.5-6 package
% Tue Aug 17 13:19:42 2010
\begin{table}[ht]
\begin{center}
\begin{tabular}{rrrrr}
  \hline
 & 1 & 1.3 & 2 & 2.7 & 3.7 \\
  \hline
Sehr Gut &  1 &  1 &  0 &  0 &  0 \\
  Gut &  0 &  0 &  1 &  0 &  0 \\
  Befriedigend &  0 &  0 &  0 &  0 &  1 \\
  Ausreichend &  0 &  0 &  0 &  1 &  0 \\
   \hline
\end{tabular}
\end{center}
\end{table}
```

Kopiert man dies in ein LaTeX-Dokument, erscheint die Tabelle wie folgt:

	1	1.3	2	2.7	3.7
Sehr Gut	1	1	0	0	0
Gut	0	0	1	0	0
Befriedigend	0	0	0	0	1
Ausreichend	0	0	0	1	0

Über das Argument caption lässt sich zusätzlich noch eine Beschriftung der Tabelle erzeugen, indem beispielsweise caption="Name der Tabelle" angegeben wird.

Auch das mit xtable() erzeugte Ergebnis lässt sich als Objekt abspeichern. Dies erlaubt eine einfache Weiterverarbeitung mittels des Befehls print(). Wird diesem ein xtable-Objekt übergeben, stehen etliche Argumente zur Verfügung. Über das Argument type lässt sich zunächst das Ausgabeformat festlegen. Der Standardwert ist "latex", alternativ kann auch "html" angegeben werden. Das Argument erlaubt die Angabe einer Datei, in die die formatierte Tabelle geschrieben werden soll. Wird "" angegeben, also doppelte Anführungszeichen, wird das Ergebnis auf dem Bildschirm ausgegeben. Ansonsten kann zwischen den Anführungszeichen ein Dateiname stehen. Möchte man beispielsweise die Tabelle tab1 im HTML-Format in einer Datei namens tabelle.html im Pfad C:\ergebnis ablegen, muss folgendes eingegeben werden:

```
> tab2 <- xtable(tab1)
> print(tab2,type="html",file="C:/ergebnis/tabelle.html")
```

Auf gleiche Weise lassen sich auch LaTeX- Tabellen erstellen, die anschließend in LaTeX-Dokumente eingebunden werden können.

Möchte man Tabellen in Kombination mit Office-Applikationen weiterverwenden, gibt es keine vergleichbar einfache Handhabe wie bei LaTeX oder HTML. Hier empfiehlt sich bei Verwendung von MS Word folgendes Vorgehen: Über den xtable()-Befehl wird in Kombination mit print() eine HTML-Tabelle erzeugt und abgespeichert. Diese wird anschließend im Browser geöffnet, markiert und in ein Word-Dokument kopiert. Bei den erscheinenden Einfüge-Optionen wird „Nur den Text übernehmen“ gewählt. Anschließend wird der so eingefügte Text markiert und „Text in Tabelle“ gewählt (zu finden unter „Tabelle“, „Umwandeln“). Hier sollte es nun reichen, einfach mit „OK“ zu bestätigen. Bei Open Office kann praktisch identisch vorgegangen werden.

4.2 Lagemaße: Mittelwerte & Co.

Im weiteren wird wieder davon ausgegangen, dass die Seminardaten unbearbeitet im Speicher unter dem Namen dat liegen. Insbesondere wurde keine Variable in Faktoren umkodiert – der Grund hierfür wird weiter unten diskutiert.

Zunächst sollen für die Variable Seminar.A diverse Lagemaße berechnet werden. Der bekannteste dürfte das arithmetische Mittel sein, welches sich über den Befehl mean() berechnen lässt:

```
> mean(dat$Seminar.A)
[1] 2.06
```

Der Mittelwert der in Seminar A erreichten Noten beträgt somit $2,06$. Soll für alle Variablen der Mittelwert berechnet werden – beispielsweise zu Vergleichszwecken – reicht es aus, dass Datenobjekt dat als Argument zu verwenden:

```
> mean(dat)
Seminar.A Seminar.B Seminar.C
     2.06      2.20      2.14
```

Die durchschnittlichen Werte der einzelnen Variablen sind einfach hintereinander aufgereit.

Eine gängige Alternative zum arithmetischen Mittel ist der Median, welcher über den gleichnamigen Befehl ausgegeben wird:

```
> median(dat$Seminar.A)
[1] 1.7
```

Hier sei angemerkt, dass dieser Befehl nicht direkt auf das Objekt dat angewendet werden kann. Stattdessen muss die Berechnung bei Verwendung dieses Befehls für alle Variablen einzeln erfolgen. Eine Alternative hierzu wird weiter unten besprochen.

Neben Mittelwerten ist weiterhin die Streuung einer Variable von Interesse. Diese kann unter anderem über die Standardabweichung, die Varianz sowie die mittlere absolute Abweichung vom Mittelwert festgestellt werden. Zur Berechnung der Standardabweichung kann der Befehl sd() benutzt werden (sd für *standard deviation*), zur Berechnung der Varianz der Befehl var():

```
> sd(dat$Seminar.A)
[1] 0.8988882
> var(dat$Seminar.A)
[1] 0.808
```

Hier lässt sich auch leicht überprüfen, dass die Varianz der quadrierten Standardabweichung entspricht:

```
> all.equal(var(dat$Seminar.A),sd(dat$Seminar.A)^2)
[1] TRUE
```

Der Befehl all.equal() wird verwendet, da die Ergebnisse nur innerhalb der Gleitkommagenauigkeit identisch sind, womit ein direkter Vergleich über den entsprechenden Operator == als Ergebnis FALSE liefern würde (s. auch Abschnitt 2.3).

Die Standardabweichung entspricht der Wurzel der durchschnittlichen quadrierten Abweichung vom Mittelwert. Durch die Quadrierung erhalten allerdings Werte, die relativ weit vom Mittelwert „entfernt" liegen einen großen Einfluss auf das Ergebnis, wodurch die Streuung bei Betrachtung der Standardabweichung hoch erscheinen kann, obwohl die meisten Werte nahe beim Mittelwert liegen. Eine Alternative zur Standardabweichung, die dieses Problem nicht aufweist, ist die durchschnittliche absolute Abweichung vom Mittelwert (*mean absolute deviation*), welche über den Befehl mad() berechnet werden kann:

```
> mad(dat$Seminar.A)
[1] 0.59304
```

Weiterhin von Interesse sind die Quantile der Verteilung einer Variablen. Diese können über den Befehl quantile() aufgerufen werden. Als Standardeinstellung werden das Minimum, das erste Quartil, der Median, das dritte Quartil und das Maximum angezeigt:

```
> quantile(dat$Seminar.A)
  0%  25%  50%  75% 100%
 1.3  1.3  1.7  2.7  3.3
```

Sollen andere Quantile ausgegeben werden, beispielsweise bestimmte Dezile, kommt das Argument probs zum Einsatz, bei welchem die entsprechenden Quantile als Vektor angegeben werden. Sollen beispielsweise das erste und das neunte Dezil angezeigt werden, lässt sich folgender Aufruf verwenden:

```
> quantile(dat$Seminar.A,probs=c(0.1,0.9))
 10%  90%
1.30 3.06
```

Das Minimum und das Maximum können auch über die Befehl min() und max() aufgerufen werden:

```
> min(dat$Seminar.A)
[1] 1.3
> max(dat$Seminar.A)
[1] 3.3
```

Bisher wurden für einzelne Lagemaße einzelne Befehle vorgestellt. Eine Auswahl von wichtigen Lagemaßen lässt sich allerdings einfach über den Befehl summary() ausgeben:

```
> summary(dat$Seminar.A)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.30    1.30    1.70    2.06    2.70    3.30
```

Diesem Befehl kann auch der vollständige Datensatz dat übergeben werden, wodurch die Kennwerte für alle Variablen des Datensatzes ausgegeben werden:

```
> summary(dat)
   Seminar.A       Seminar.B       Seminar.C
 Min.   :1.30   Min.   :1.3   Min.   :1.00
 1st Qu.:1.30   1st Qu.:1.3   1st Qu.:1.30
 Median :1.70   Median :1.7   Median :2.00
 Mean   :2.06   Mean   :2.2   Mean   :2.14
 3rd Qu.:2.70   3rd Qu.:2.7   3rd Qu.:2.70
 Max.   :3.30   Max.   :4.0   Max.   :3.70
```

In beiden Varianten werden das Minimum, das erste Quartil, der Median, der Mittelwert, das dritte Quartil und das Maximum berechnet.

Liegen fehlende Werte vor, lassen sich die meisten der gerade vorgestellten Befehle nicht mehr ohne weiteres verwenden. Um dies zu demonstrieren, setzen wir den Wert der Variablen Seminar.B bei der ersten Beobachtung auf NA:

```
> dat$Seminar.B[1] <- NA
```

Wird nun der Befehl mean() angewandt, erhält man folgendes Ergebnis:

```
> mean(dat$Seminar.B)
[1] NA
```

Als Ergebnis wird NA ausgegeben, was darauf hinweist, das fehlende Werte vorliegen. Zwar liegt für eine Beobachtung kein Wert vor, allerdings könnte aus den Werten der vier anderen Beobachtungen ein Mittelwert berechnet werden. Um dies zu bewerkstelligen, muss das Argument na.rm auf TRUE gesetzt werden. na.rm steht für *NA remove* und weist als Standard den Wert FALSE auf. Hierdurch werden fehlende Werte bei der Berechnung des arithmetischen Mittelwerts nicht ausgeschlossen, was eine Ermittlung des Wertes nicht möglich macht. Ist hingegen na.rm=TRUE werden fehlende Werte nicht berücksichtigt, so dass die Berechnung durchgeführt werden kann:

```
> mean(dat$Seminar.B, na.rm=TRUE)
[1] 2.325
```

Bei den meisten anderen oben vorgestellten Befehlen tritt dieses Problem ebenfalls auf. Allerdings verschafft auch hier das Argument na.rm=TRUE abhilfe. Als Beispiele seien die Befehle min() und quantile() betrachtet:

```
> min(dat$Seminar.B)
[1] NA
> min(dat$Seminar.B, na.rm=TRUE)
```

```
[1] 1.3
> quantile(dat$Seminar.B)
Fehler in quantile.default(dat$Seminar.B) :
  fehlende Werte und NaNs sind nicht erlaubt, wenn 'na.rm' = FALSE
> quantile(dat$Seminar.B, na.rm=TRUE)
   0%   25%   50%   75%  100%
1.300 1.300 2.000 3.025 4.000
```

Bei der Verwendung des Befehls summary() werden fehlende Werte automatisch ausgeschlossen, ohne das ein weiteres Argument nötig wäre. Zudem wird nun zusätzlich die Zahl der fehlenden Werte ausgegeben:

```
> summary(dat$Seminar.B)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  1.300   1.300   2.000   2.325   3.025   4.000   1.000
```

Die in diesem Abschnitt vorgestellten Befehle lassen sich wohlgemerkt nicht auf Faktoren anwenden. Ist beispielsweise die Variable Seminar.A in einen Faktor umgewandelt, erhält man bei Anwendung des Befehls mean() eine Fehlermeldung:

```
> dat$Seminar.A <- factor(dat$Seminar.A, labels =
+ c("Sehr Gut","Gut","Befriedigend","Ausreichend"))
> mean(dat$Seminar.A)
[1] NA
Warnmeldung:
In mean.default(dat$Seminar.A) :
  Argument ist weder numerisch noch boolesch: gebe NA zurück
```

Dies lässt sich nicht ohne weiteres Umgehen, weshalb es sich in diesem Fall empfiehlt, zunächst auf die Umwandlung in einen Faktor zu verzichten.

4.3 Kovarianz und Korrelation

Kovarianz und Korrelation sind deskriptive Kennwerte für den Zusammenhang zweier Variablen. Zunächst gehen wir wieder davon aus, dass die Seminardaten in ihrer ursprünglichen Fassung im Workspace unter dem Namen dat abgespeichert sind, also keine fehlenden Werte vorliegen oder Variablen in Faktoren umgewandelt wurden.

Die Kovarianz zweier Variablen lässt sich entweder über den Befehl var(), der schon zur Berechnung der Varianz einer Variable verwendet wurde, oder über den Befehl cov() berechnen. cov steht hierbei für *covariance*. In beiden Fällen werden dem gewählten Befehl die beiden interessierenden Variablen als Argumente übergeben:

```
> var(dat$Seminar.A, dat$Seminar.C)
[1] 0.812
> cov(dat$Seminar.A, dat$Seminar.C)
[1] 0.812
```

Man kann auch einfach das Objekt dat als einziges Argument verwenden, zum Beispiel:

```
> cov(dat)
           Seminar.A Seminar.B Seminar.C
Seminar.A      0.808     0.790     0.812
Seminar.B      0.790     1.340     1.135
Seminar.C      0.812     1.135     1.193
```

Die Verwendung von var(dat) führt zum selben Ergebnis. Ausgegeben wird die so genannte Kovarianzmatrix der Variablen. Die einzelnen Einträge geben die Kovarianz der Variablen wieder, die die entsprechende Zeile und Spalte benennen. Beispielsweise ist $0,790$ die Kovarianz der Variablen Seminar.A und Seminar.B. Die Kovarianzen der drei Variablen mit sich selbst – also beispielsweise die Kovarianz von Seminar.A mit Seminar.A – entsprechen den Varianzen der Variablen, was sich auch leicht überprüfen lässt:

```
> cov(dat$Seminar.A, dat$Seminar.A)
[1] 0.808
> var(dat$Seminar.A)
[1] 0.808
```

Liegen fehlende Werte vor, kann bei var() wieder das Argument na.rm=T angegeben werden. Beim Befehl cov() hingegen muss als Argument use verwendet werden, welches eine genauere Steuerung erlaubt. Im allgemeinen empfiehlt sich bei fehlenden Werten die Verwendung des Befehls cov() mit dem kompletten Datensatz beziehungsweise allen interessierenden Variablen als Argument, was an einem Beispiel verdeutlicht werden soll. Zunächst wird der Wert der Variable Seminar.B der ersten Beobachtung wieder auf NA gesetzt:

```
> dat$Seminar.B[1] <- NA
```

Als nächstes wird die Kovarianz zwischen den Variablen Seminar.A und Seminar.B, zwischen den Variablen Seminar.A und Seminar.C sowie die komplette Kovarianzmatrix berechnet. Hierfür wird der Befehl var() verwendet:

```
> var(dat$Seminar.A, dat$Seminar.B)
[1] NA
> var(dat$Seminar.A, dat$Seminar.B, na.rm=T)
[1] 0.895
> var(dat$Seminar.A, dat$Seminar.C, na.rm=T)
[1] 0.812
> var(dat, na.rm=T)
           Seminar.A Seminar.B Seminar.C
Seminar.A 0.8366667  0.895000 0.7216667
Seminar.B 0.8950000  1.682500 1.2758333
Seminar.C 0.7216667  1.275833 1.0491667
```

Hier fällt auf, dass die beiden Ergebnisse für die Kovarianz zwischen Seminar.A und Seminar.C voneinander abweichen. Bei der „einzelnen“ Berechnung erhält man $0,812$, in der Kovarianzmatrix findet sich hingegen der Wert $0,7216667$. Ursache hierfür ist, dass bei der Berechnung der Kovarianzmatrix nur die vier Fälle verwendet werden, die keine fehlenden Werte aufweisen. Die Beobachtung mit dem fehlenden Wert bei der Variable Seminar.B wird also komplett von

der Berechnung der Matrix ausgeschlossen. Bei der direkten Berechnung der Kovarianz von Seminar.A und Seminar.C werden hingegen alle fünf Beobachtungen genutzt, da fehlende Werte nur bei der Variablen Seminar.B vorliegen. Zwar ist es durchaus möglich, den Befehl var() zu verwenden und die Beobachtung mit dem fehlenden Wert auszuschließen, allerdings ist dies deutlich umständlicher als die Verwendung von cov() in Kombination mit dem Argument use.

Das Argument use weist als Standard den Wert "everything" auf. In diesem Fall werden Variablen mit fehlenden Werten komplett ausgeschlossen und als Ergebnis für die Varianz von beziehungsweise Kovarianz mit solchen Variablen NA ausgegeben:

```
> cov(dat)
           Seminar.A Seminar.B Seminar.C
Seminar.A      0.808        NA      0.812
Seminar.B         NA        NA         NA
Seminar.C      0.812        NA      1.193
```

Die Angabe von use="complete.obs" führt dazu, dass Beobachtungen mit fehlenden Werten komplett ausgeschlossen werden, was äquivalent ist zu var(dat,na.rm=T):

```
> cov(dat,use="complete.obs")
           Seminar.A Seminar.B Seminar.C
Seminar.A 0.8366667  0.895000 0.7216667
Seminar.B 0.8950000  1.682500 1.2758333
Seminar.C 0.7216667  1.275833 1.0491667
```

Wird use="pairwise.complete.obs" angegeben, werden Beobachtungen nur bei der Berechnung von Kovarianzen zwischen zwei Variablen ausgeschlossen, wenn sie bei einer der Variablen fehlende Werte aufweisen:

```
> cov(dat,use="pairwise.complete.obs")
           Seminar.A Seminar.B Seminar.C
Seminar.A      0.808  0.895000  0.812000
Seminar.B      0.895  1.682500  1.275833
Seminar.C      0.812  1.275833  1.193000
```

In diesem Fall kann die Zahl der bei der Berechnung der Kovarianzen berücksichtigten Beobachtungen variieren.

Bei der Berechnung von Korrelationen zwischen Variablen ist zu beachten, dass es hierfür mehrere Varianten gibt. Die Berechnung des Korrelationskoeffizienten nach Pearson erfolgt durch die Verwendung des Befehls cor():

```
> cor(dat$Seminar.A,dat$Seminar.C)
[1] 0.8270469
```

Zur Handhabung von fehlenden Werten steht wieder das Argument use zur Verfügung, welches wie beim cov()-Befehl verwendet werden kann:

```
> cor(dat)
           Seminar.A Seminar.B Seminar.C
Seminar.A 1.0000000        NA 0.8270469
Seminar.B         NA         1        NA
```

```
Seminar.C 0.8270469         NA 1.0000000
> cor(dat,use="pairwise.complete.obs")
           Seminar.A Seminar.B Seminar.C
Seminar.A 1.0000000 0.7543437 0.8270469
Seminar.B 0.7543437 1.0000000 0.9602718
Seminar.C 0.8270469 0.9602718 1.0000000
```

Die Interpretierbarkeit des Korrelationskoeffizienten nach Pearson ist mit etlichen Anforderungen an die Daten verknüpft, die oftmals nicht erfüllt sind. Deshalb wurden diverse Alternativen entwickelt. Zwei bekannte Varianten, die in R implementiert sind, sind Spearmans ρ und Kendalls τ, bei denen es sich um so genannte Rangkorrelationskoeffizienten handelt. Diese können durch Angabe der Argumente method="spearman" beziehungsweise method="kendall" beim Befehl cor() verwendet werden. Hierbei kann das Argument use auf gleiche Weise wie bisher ebenfalls benutzt werden:

```
> cor(dat,method="spearman",use="pairwise.complete.obs")
           Seminar.A Seminar.B Seminar.C
Seminar.A 1.0000000 0.7378648 0.8720816
Seminar.B 0.7378648 1.0000000 0.9486833
Seminar.C 0.8720816 0.9486833 1.0000000
> cor(dat,method="kendall",use="pairwise.complete.obs")
           Seminar.A Seminar.B Seminar.C
Seminar.A 1.0000000 0.5477226 0.7378648
Seminar.B 0.5477226 1.0000000 0.9128709
Seminar.C 0.7378648 0.9128709 1.0000000
```

Wie aus den Ergebnissen ersichtlich ist, unterscheiden sich die Werte der drei Arten von Korrelationskoeffizienten teils nicht unerheblich. Insofern muss bei der Wahl eines geeigneten Korrelationskoeffizienten vorsichtig vorgegangen werden.

5 Einfache Grafiken

5.1 Grafiken für eine Variable

5.1.1 Histogramme

Um einfache Grafiken zu erstellen, gehen wir zunächst wieder davon aus, dass die Seminardaten unverändert als Objekt dat geladen sind. Zunächst werden wir einfache Histogramme erzeugen. Bei diesen werden wir etliche Argumente zur Steuerung von Grafikeigenschaften kennen lernen, die bei den meisten Grafiken angewandt werden können.

Für Histogramme kann der Befehl hist() verwendet werden. Im einfachsten Fall wird diesem Befehl als Argument einfach eine Variable übergeben:

```
> hist(dat$Seminar.A)
```

Wird dieser Befehl ausgeführt, öffnet sich ein neues Fenster, in dem die Grafik dargestellt ist. Das Resultat findet sich in Abbildung 5.1a. Aus dieser Grafik ist ersichtlich, dass zwei Studenten eine Note im Bereich von 1 bis $1,5$ haben, ein Student eine Note im Bereich von $1,5$ bis 2 und so fort. Dabei beginnt die x-Achse mit dem Wert 1 und hört beim Wert $3,5$ auf, wobei die Intervallbreite bei $0,5$ liegt. Ferner ist die x-Achse mit dem Namen der betrachteten Variable versehen, die y-Achse mit der Beschriftung „Frequency“ und der Titel der Grafik lautet „Histogram of dat$Seminar.A“. Die eingezeichneten Werte der y-Achse reichen von 0 bis 2 mit einer Schrittgröße von $0,5$. Alle diese Eigenschaften der Grafik lassen sich leicht ändern.

Die Zahl der Intervalle lässt sich über das Argument breaks steuern. Entweder gibt man die Zahl der Intervalle direkt an, wobei in diesem Fall die Anfangs- und Endpunkte der Intervalle von R automatisch festgelegt werden, oder aber man gibt die Intervallgrenzen selbst als Vektor ein. Möchte man 12 Intervalle haben, führt man folgenden Befehl aus:

```
> hist(dat$Seminar.A, breaks=12)
```

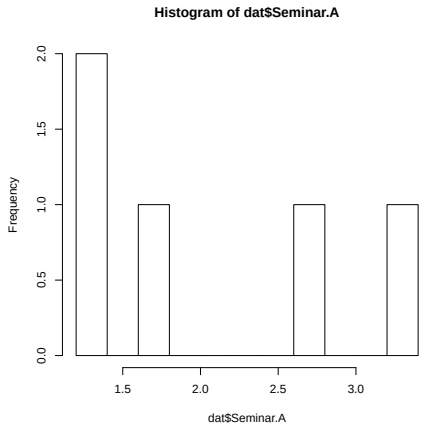
Das Ergebnis findet sich in Abbildung 5.1b.

Möchte man hingegen drei Intervalle haben, wobei das erste von 1 bis 2 reicht, das zweite von 2 bis 3 und das dritte von 3 bis 4, kann wie folgt vorgegangen werden:

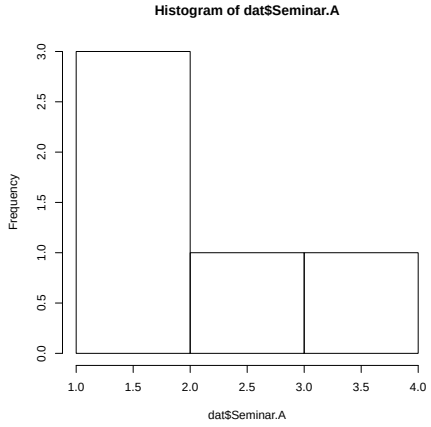
```
> hist(dat$Seminar.A, breaks=c(1,2,3,4))
```

Das Resultat ist in Grafik 5.1c abgebildet. Das erste Intervall wird also aus dem ersten und dem zweiten angegebenen Wert gebildet, dass zweite Intervall aus dem zweiten und dem dritten Wert und so fort.

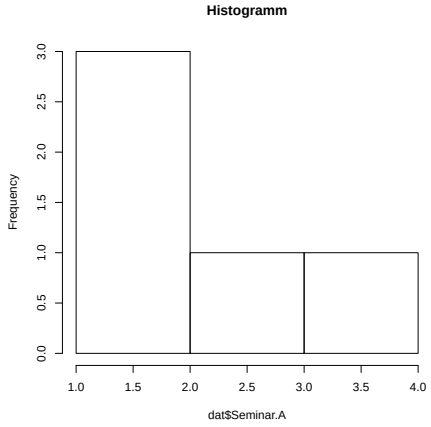
(a)



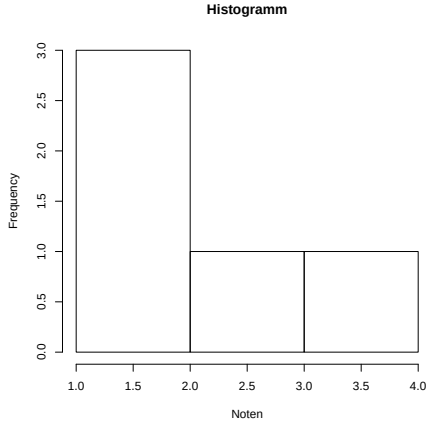
(b)



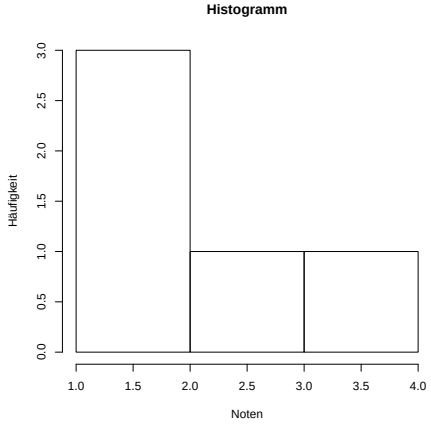
(c)



(d)



(e)



(f)

Abbildung 5.1: Abbildungen zu den Histogrammbeispielen (a) – (f)

Als nächstes wollen wir die Beschriftung der Grafik ändern – also die Beschriftung der Achsen sowie die Überschrift der Grafik. Die Überschrift kann über das Argument main festgelegt werden, wobei die Überschrift in Anführungszeichen gesetzt werden muss. Soll der Titel der Grafik beispielsweise einfach „Histogramm“ lauten, wird folgendes eingegeben:

```
> hist(dat$Seminar.A,main="Histogramm",breaks=c(1,2,3,4))
```

Das Ergebnis findet sich in Abbildung 5.1d.

Die Benennung der Achsen lässt sich über die Argumente xlab und ylab steuern. Auch hier müssen die Namen wieder in Anführungszeichen gesetzt werden. Soll die x-Achse mit „Noten“ beschriftet werden, wird folgendes ausgeführt:

```
> hist(dat$Seminar.A,xlab="Noten",main="Histogramm",breaks=c(1,2,3,4))
```

Soll ferner noch die y-Achse mit „Häufigkeit“bezeichnet werden, wird das Argument ylab entsprechend ergänzt:

```
> hist(dat$Seminar.A,ylab="Häufigkeit",xlab="Noten",main="Histogramm",
+      breaks=c(1,2,3,4))
```

Die resultierenden Grafiken finden sich in den Abbildungen 5.1e und 5.1f.

Um die auf den Achsen abgetragenen Werte zu steuern stehen die Argumente xlim, ylim, xaxp und yaxp zur Verfügung.

Über xlim und ylim wird angegeben, über welchen Wertebereich die x- und die y-Achse gezeichnet werden sollen. Dies beeinflusst die Wertebeschriftung (*ticks*) der selbigen wohlgemerkt nur indirekt. Das händische Festlegen des Wertebereichs der Achsen kann sinnvoll sein, wenn mehrere Grafiken erstellt werden, die miteinander verglichen werden sollen, die automatische Festlegung aber zu unterschiedlichen Wertebereichen führt, was den Vergleich erschwert. In Abbildung 5.1f beginnt die y-Achse beim Wert 0 und reicht bis zum Wert 3. Soll sie vom Wert 0 bis 4 reichen, wird folgendes eingegeben:

```
> hist(dat$Seminar.A,ylab="Häufigkeit",xlab="Noten",main="Histogramm",
+      breaks=c(1,2,3,4),ylim=c(0,4))
```

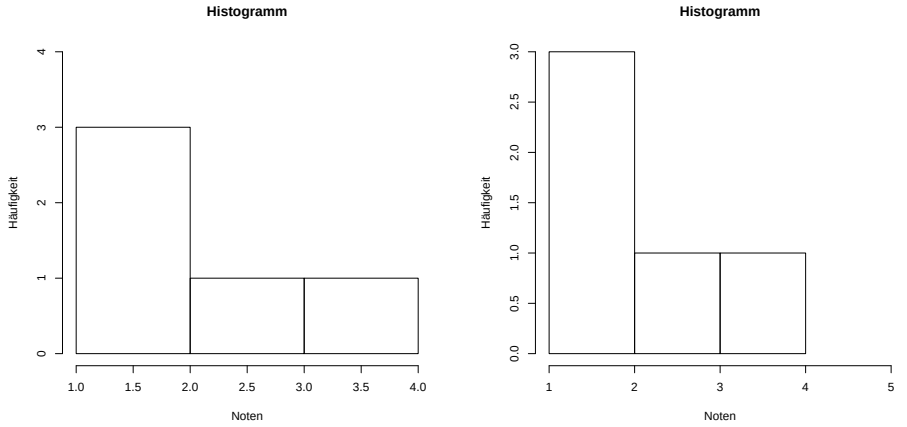
Die Ergebnisse finden sich in Abbildung 5.2a. Wie man sieht reicht die y-Achse nun von 0 bis 4. Zudem wurden die an der Achse eingezeichneten Werte automatisch verändert.

Auf gleiche Weise kann auch der Bereich der x-Achse verändert werden:

```
> hist(dat$Seminar.A,ylab="Häufigkeit",xlab="Noten",main="Histogramm",
      breaks=c(1,2,3,4),xlim=c(1,5))
```

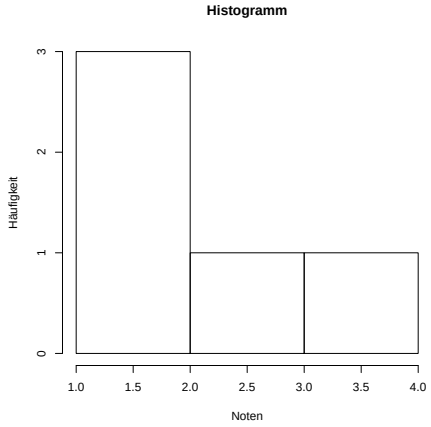
Das Ergebnis dieses Befehlsaufrufs ist in Abbildung 5.2b zu finden.

Betrachtet man wieder Abbildung 5.1f und befindet die Wertebereiche der Achsen für akzeptabel, scheint es dennoch notwendig die eingezeichneten Werte an den Achsen, die so genannten *ticks*, zu ändern. Beispielsweise sind an der y-Achse Werte wie $1,5$ und $2,5$ abgetragen, die aber bei den absoluten Häufigkeiten, die dargestellt werden, nicht sinnvoll sind – zumal es keine „halben“ Beobachtungen gibt. Um diese Werte zu ändern können die Argumente xaxp und yaxp verwendet werden. In beiden Fällen wird ein Vektor angegeben, der drei Werte enthält. Der

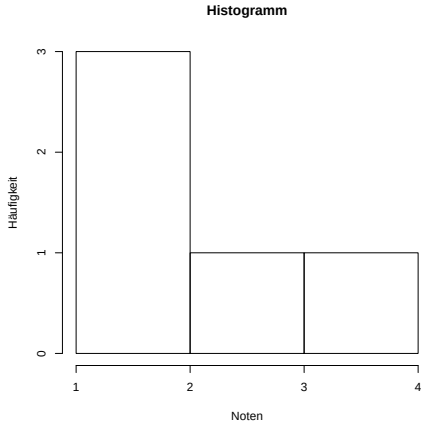


(a)

(b)



(c)



(d)

Abbildung 5.2: Abbildungen zu den Beispielen für Achsenbeschriftungen (a) – (d)

erste Werte ist der niedrigste darzustellende *tick*, der zweite ist der höchste abzubildende Wert und der dritte Wert gibt die Zahl der Intervalle zwischen diesen Extremwerten wieder. Würde man beispielsweise yaxp=c(0,3,3) angeben, würden *ticks* bei den Werten 0, 1, 2 und 3 erscheinen. Der niedrigste Wert ist 0, der höchste 3 und es gibt drei Intervalle, nämlich von 0 zu 1, von 1 zu 2 und von 2 zu 3. Möchte man dieses Beispiel auf Abbildung 5.1f anwenden, gibt man folgendes ein:

```
> hist(dat$Seminar.A, breaks=c(1,2,3,4), yaxp=c(0,3,3),
+       ylab="Häufigkeit", xlab="Noten", main="Histogramm")
```

Das Ergebnis sieht man in Abbildung 5.2c. Auf gleiche Weise kann man auch die *ticks* der x-Achse verändern:

```
> hist(dat$Seminar.A, breaks=c(1,2,3,4), xaxp=c(1,4,3), yaxp=c(0,3,3),
+       ylab="Häufigkeit", xlab="Noten", main="Histogramm")
```

In diesem Beispiel wurden alle verwendeten Argumente angegeben, also auch die Argumente für die Namen der Grafik und der Achsen. Das Ergebnis ist in Abbildung 5.2d zu sehen.

5.1.2 Balkendiagramme

Balkendiagramme werden auf der Grundlage von Tabellen erstellt. Deshalb wird zunächst ein Tabellenobjekt für die Variable Seminar.A erzeugt:

```
> tab1 <- table(dat$Seminar.A)
```

Anschließend kann über den Befehl barplot ein Balkendiagramm erstellt werden:

```
> barplot(tab1)
```

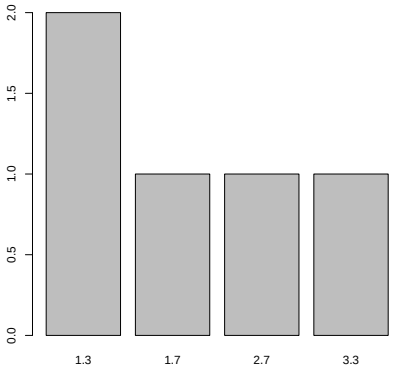
Das Ergebnis findet sich in Abbildung 5.3a. Für jede auftretende Merkmalsausprägung ist ein Balken vorhanden, der die absolute Häufigkeit des Auftretens anzeigt, wobei jeder Balken mit der entsprechenden Merkmalsausprägung beschriftet ist.

Wird die Variable Seminar.A in einen Faktor umgewandelt, werden die Namen der Faktoren als Beschriftung der Balken verwendet:

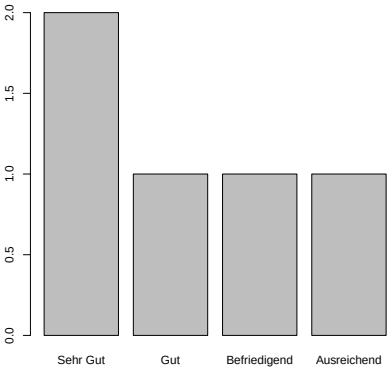
```
> dat$Seminar.A <- factor(dat$Seminar.A, labels =
+ c("Sehr Gut","Gut","Befriedigend","Ausreichend"))
> dat$Seminar.A
[1] Sehr Gut      Ausreichend   Befriedigend  Gut           Sehr Gut
Levels: Sehr Gut Gut Befriedigend Ausreichend
> tab2 <- table(dat$Seminar.A)
> barplot(tab2)
```

Das Ergebnis ist in Abbildung 5.3b zu finden.

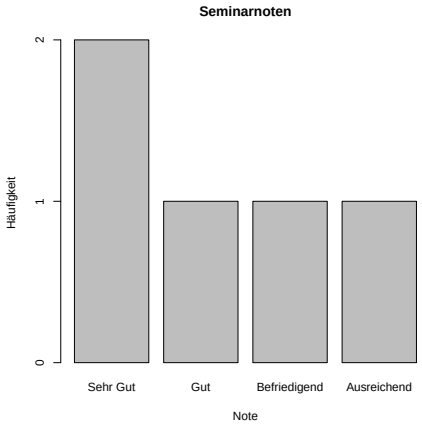
Um Namen für die Achsen und eine Überschrift für die Grafik einzufügen, können die bekannten Argumente xlab, ylab und main verwendet werden. Gleiches gilt für das Setzen von *ticks* der y-Achse mit yaxp:



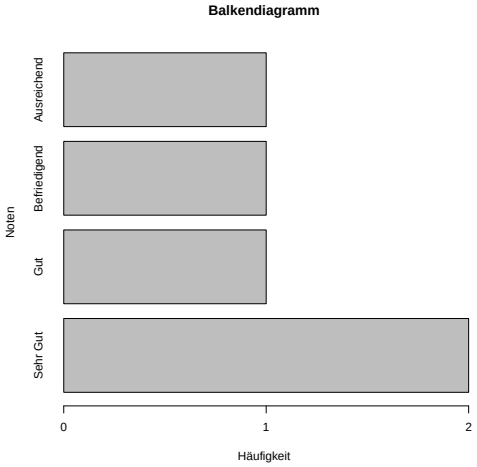
(a)



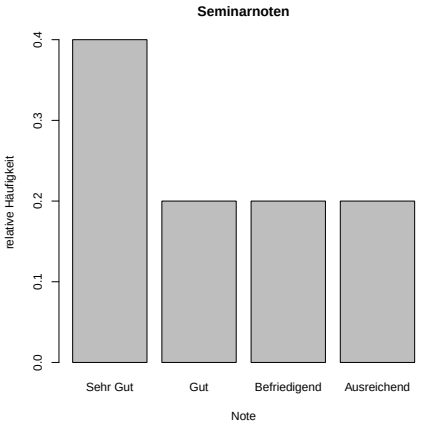
(b)



(c)



(d)



(e)

Abbildung 5.3: Beispiele für Balkendiagramme (a) – (e)

```
> barplot(tab2,main="Seminarnoten",xlab="Note",
+         ylab="Häufigkeit",yaxp=c(0,2,2))
```

Der Grafik wird die Überschrift „Seminarnoten“ gegeben, die x-Achse bekommt die Bezeichnung „Noten“, an die y-Achse wird die Beschriftung „Häufigkeit“ gesetzt, was in Abbildung 5.3c zu sehen ist.

Sollen die Balken horizontal anstatt vertikal dargestellt werden, muss dass Argument horiz auf den Wert TRUE gesetzt werden:

```
> barplot(tab2,main="Seminarnoten",xlab="Häufigkeit",ylab="Note",
+         horiz=T,xaxp=c(0,2,2))
```

Das Ergebnis ist in 5.3d abgebildet. Zu beachten ist, dass nun anstelle von yaxp das Argument xaxp verwendet werden muss. Zudem muss nun verglichen mit den vorherigen Balkendiagrammen die Beschriftung von x- und y-Achse getauscht werden.

Sollen anstelle von absoluten Häufigkeiten relative Häufigkeiten verwendet werden, muss die verwendete Tabelle mittels prop.table() entsprechend umgewandelt werden:

```
> tab3 <- prop.table(tab2)
> barplot(tab3,main="Seminarnoten",xlab="Note",
+         ylab="relative Häufigkeit")
```

Wie in Abbildung 5.3e zu sehen, stehen an der y-Achse nun relative Werte.

Die Farben der Balken lassen sich über das Argument col steuern. Beispiele finden sich in Abbildung 5.4. Abbildungen 5.4a und 5.4b lassen sich wie folgt erzeugen:

```
> barplot(tab2,main="Seminarnoten",xlab="Note",ylab="Häufigkeit",
+         yaxp=c(0,2,2),col="white")
> barplot(tab2,main="Seminarnoten",xlab="Note",ylab="Häufigkeit",
+         yaxp=c(0,2,2),col="blue")
```

Beim Argument col wird die zu verwendende Farbe in Anführungszeichen angegeben. Eine Übersicht der zur Verfügung stehenden Farben beziehungsweise deren Namen erhält man über Eingabe des Befehls colors():

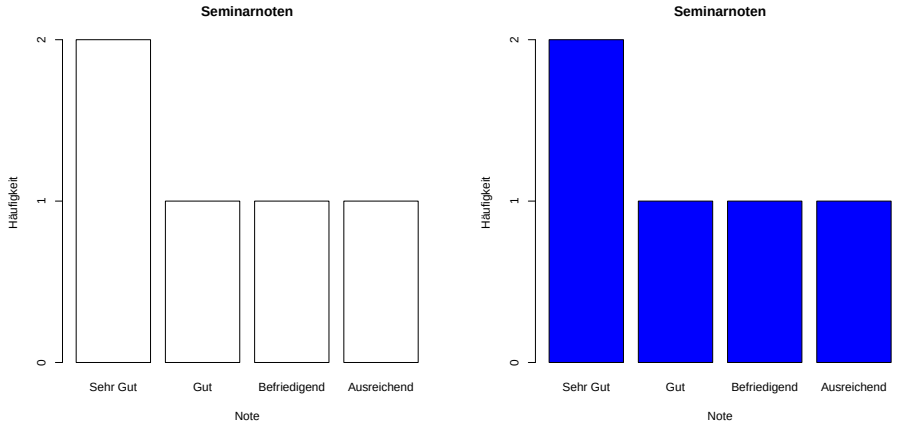
```
> colors()
```

Sollen die einzelnen Balken unterschiedliche Farben aufweisen, wird ein Vektor von Farben angegeben. Soll jeder der vier Balken eine eigene Farbe bekommen, werden vier Farben in einen Vektor geschrieben, wobei diese Farben der Reihe nach den Balken von links nach rechts zugeordnet werden:

```
> barplot(tab2,col=c("grey40","grey50","grey60","grey70"))
```

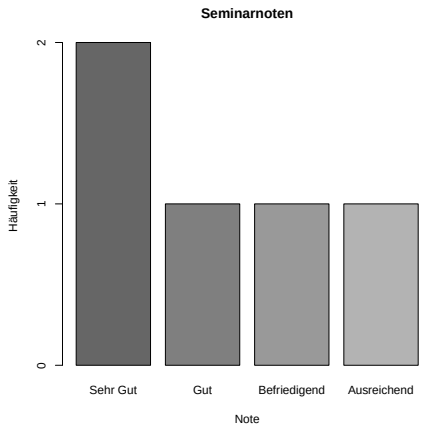
Hier werden den Balken unterschiedliche Grautöne zugeordnet, was in Abbildung 5.4c zu sehen ist. Weitere verwendete Argumente wurden hier ausgespart.

Werden weniger Farben als Balken angegeben, wird der Farbvektor wiederholt. Sind beispielsweise vier Balken vorhanden und nur zwei Farben angegeben, werden die ersten beiden Balken entsprechend der angegebenen Farben eingefärbt und für die zwei weiteren Balken wird der Farbvektor erneut verwendet. Ein Beispiel ist in Abbildung 5.4d zu sehen:

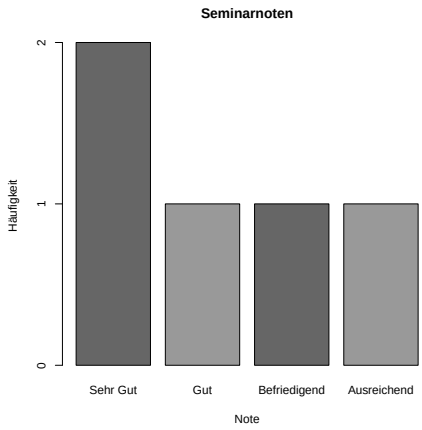


(a)

(b)



(c)



(d)

Abbildung 5.4: Beispiele für Farben bei Balkendiagrammen (a) – (d)

```
> barplot(tab2, col=c("grey40","grey60"))
```

Auf die gleiche Art und Weise können auch Histogramme eingefärbt werden.

5.2 Grafiken für zwei Variablen

5.2.1 Streudiagramme

Eine Möglichkeit, um die Verteilung von zwei Variablen zu visualisieren sind sogenannte Streudiagramme (engl.: *scatter plot*). Diese lassen sich in R mittels des Befehls plot() erstellen. Im einfachsten Fall wird die Form plot(x,y) verwendet, wobei x für die Variable steht, die an die x-Achse gezeichnet wird und y entspricht den Werten der y-Achse.

Hat man die Klausurdaten als Objekt dat im Workspace geladen und möchte die Noten aus Seminar A als x-Variable und die Noten aus Seminar B als y-Variable verwenden, lässt sich einfach wie folgt vorgehen:

```
> plot(dat$Seminar.A, dat$Seminar.B)
```

Hiermit hat man dem plot()-Befehl im Prinzip zwei Vektoren übergeben. Die Länge der Vektoren entspricht der Zahl der einzuzeichnenden Punkte. Der erste Vektor enthält die Werte der x-Dimension, der zweite Vektor die Werte der y-Dimension. Das Ergebnis wird wie gewohnt in einem gesonderten Grafikfenster ausgegeben und ist in Abbildung 5.5a dargestellt. Für jeden Studenten ist entsprechend der Klausurnoten in den beiden Seminaren ein kleiner Kreis eingezeichnet. x- und y-Achse sind mit den eingegebenen Klausurnamen bezeichnet und ein Titel ist nicht vorhanden. Um diese Beschriftungen zu ändern können wieder die Argumente main, ylab und xlab verwendet werden, wie in Abbildung 5.5b zu sehen ist:

```
> plot(dat$Seminar.A, dat$Seminar.B, main="Streudiagramm",
+      xlab="Note in Seminar A", ylab="Note in Seminar B")
```

Ebenso können wieder die Argumente xlim, ylim, xaxp und yaxp benutzt werden. Ein Beispiel:

```
> plot(dat$Seminar.A, dat$Seminar.B, main="Streudiagramm",
+      xlab="Note in Seminar A", ylab="Note in Seminar B",
+      xlim=c(1,5), ylim=c(1,5))
```

Das Ergebnis ist in Abbildung 5.5c zu sehen.

Um die Größe der eingezeichneten Kreise zu verändern, lassen sich die Argumente cex und lwd benutzen. Über das Argument cex wird ein numerischer Faktor angegeben, um den die eingezeichneten Symbole verglichen zur Standardausgabe vergrößert werden sollen. Möchte man also verglichen mit den bisherigen Grafiken doppelt so große Kreise haben, würde man cex=2 angeben:

```
> plot(dat$Seminar.A, dat$Seminar.B, main="Streudiagramm",
+      xlab="Note in Seminar A", ylab="Note in Seminar B",
+      cex=2)
```

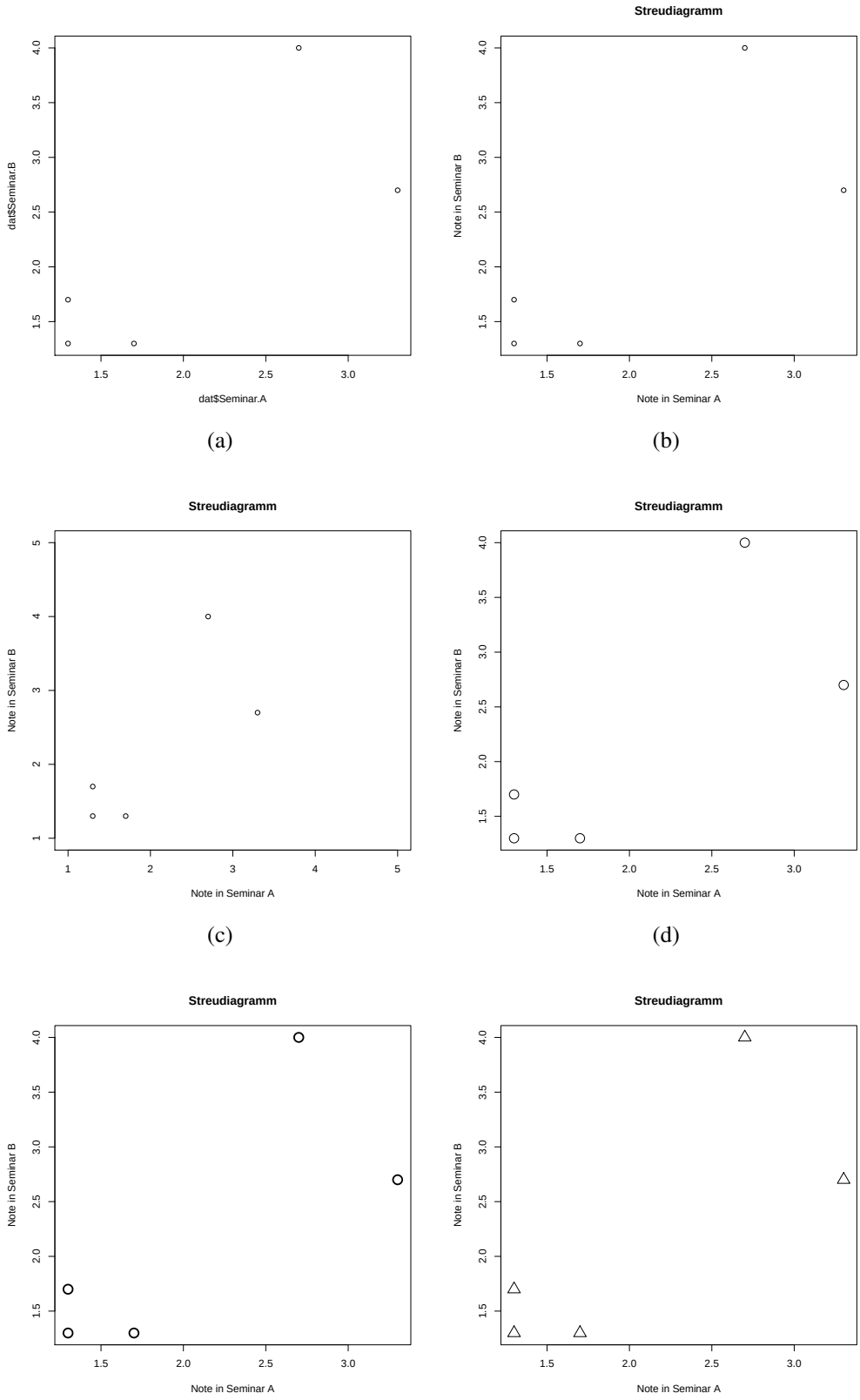


Abbildung 5.5: Beispiele für Streudiagramme (a) – (f)

In Abbildung 5.5d ist das Resultat zu finden.

Über das Argument lwd wird die Dicke der Linien gesteuert, aus denen die Kreise bestehen. Auch hier ist wird wieder ein relativer Faktor angegeben, der sich auf die „Standard-Dicke“ bezieht. Wählt man beispielsweise lwd=2.5 und kombiniert dies mit cex=2, erhält man Grafik 5.5e:

```
> plot(dat$Seminar.A,dat$Seminar.B,main="Streudiagramm",
+       xlab="Note in Seminar A",ylab="Note in Seminar B",
+       cex=2,lwd=2.5)
```

Möchte man anstelle der Kreise andere Symbole einzeichnen, muss das Argument pch benutzt werden, wobei pch für *plotting character* steht. Diesem Argument kann entweder einfach ein als Text angegebenes Symbol übergeben werden – beispielsweise in der Form pch="+". Oder aber es wird eine Zahl angegeben, wobei bestimmte Zahlen mit bestimmten Symbolen korrespondieren. Beispielsweise sorgt pch=2 dafür, dass Dreiecke eingezeichnet werden, wie in Abbildung 5.5f:

```
> plot(dat$Seminar.A,dat$Seminar.B,main="Streudiagramm",
+       xlab="Note in Seminar A",ylab="Note in Seminar B",
+       cex=2,pch=2)
```

Eine Übersicht über mögliche Symbole bietet Abbildung 5.6. Dabei lassen sich die Symbole verwenden, indem beim Argument pch die Zahl angegeben wird, die sich aus dem Produkt der angegebenen Zeilen- und Spaltennummer ergibt. Das Symbol $<$ befindet sich in der mit der Nummer 12 versehenen Zeile und der mit der Nummer 5 versehenen Spalte, womit die zu verwendende Zahl $12 \cdot 5 = 60$ ist und das Argument pch=60 benutzt wird.

Zum farbigen Einzeichnen der Symbole kann wiederum das bekannte Argument col verwendet werden. Auch hier können wieder nur eine oder aber mehrere Farben angegeben werden. Möchte man alle eingezeichneten Symbole rot färben, lässt sich folgende Syntax benutzen:

```
> plot(dat$Seminar.A,dat$Seminar.B,main="Streudiagramm",
+       xlab="Note in Seminar A",ylab="Note in Seminar B",
+       cex=2,lwd=2.5,col="red")
```

Sollen die einzelnen Symbole in unterschiedlichen Farben eingezeichnet werden, muss ein Vektor angegeben werden:

```
> plot(dat$Seminar.A,dat$Seminar.B,main="Streudiagramm",
+       xlab="Note in Seminar A",ylab="Note in Seminar B",
+       cex=2,lwd=2.5,col=c("red","green","blue","orange","brown"))
```

Die Resultate der letzten beiden Aufrufe sind in 5.7a und 5.7b dargestellt. Werden über einen Vektor mehrere Farben angegeben ist wieder zu beachten, dass falls dieser Vektor weniger Elemente als eingezeichnete Beobachtungen enthält, Farben wiederholt werden.

Um die bisher erstellten Streudiagramme wurden bisher immer Boxen gezeichnet – die Diagramme sind also an allen Seiten mit Linien begrenzt. Ob und wie Boxen eingezeichnet werden, lässt sich über das Argument bty steuern. Für dieses können die Werte "o" , "l", "7", "c", "u" und "]" angegeben werden. Die Form der resultierenden Box entspricht dabei grob der Form des ein-

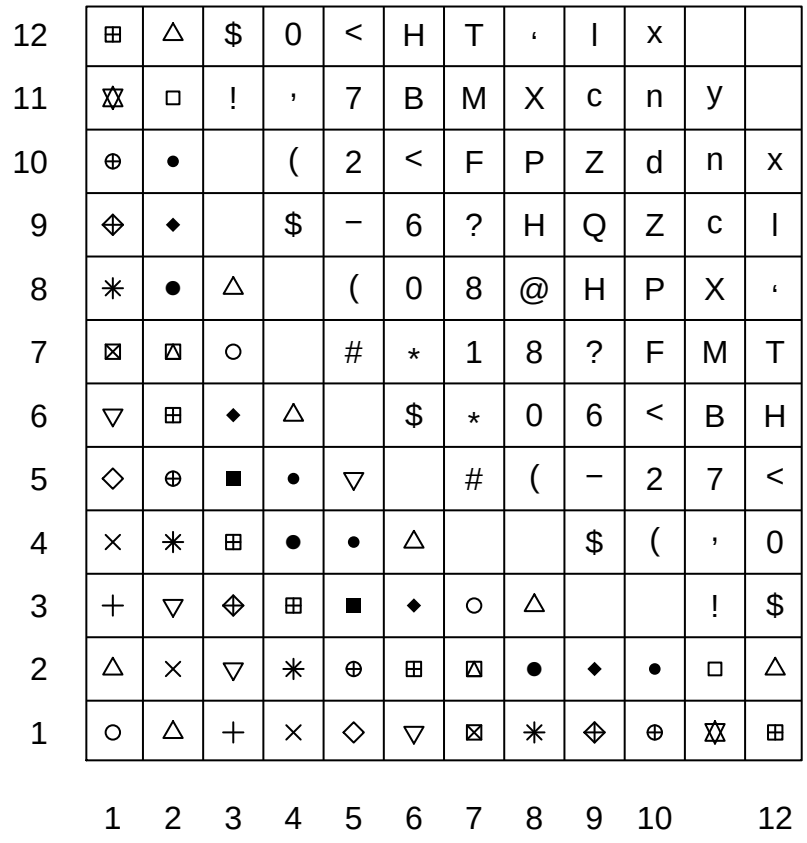


Abbildung 5.6: Symbole zur Visualisierung von Datenpunkten

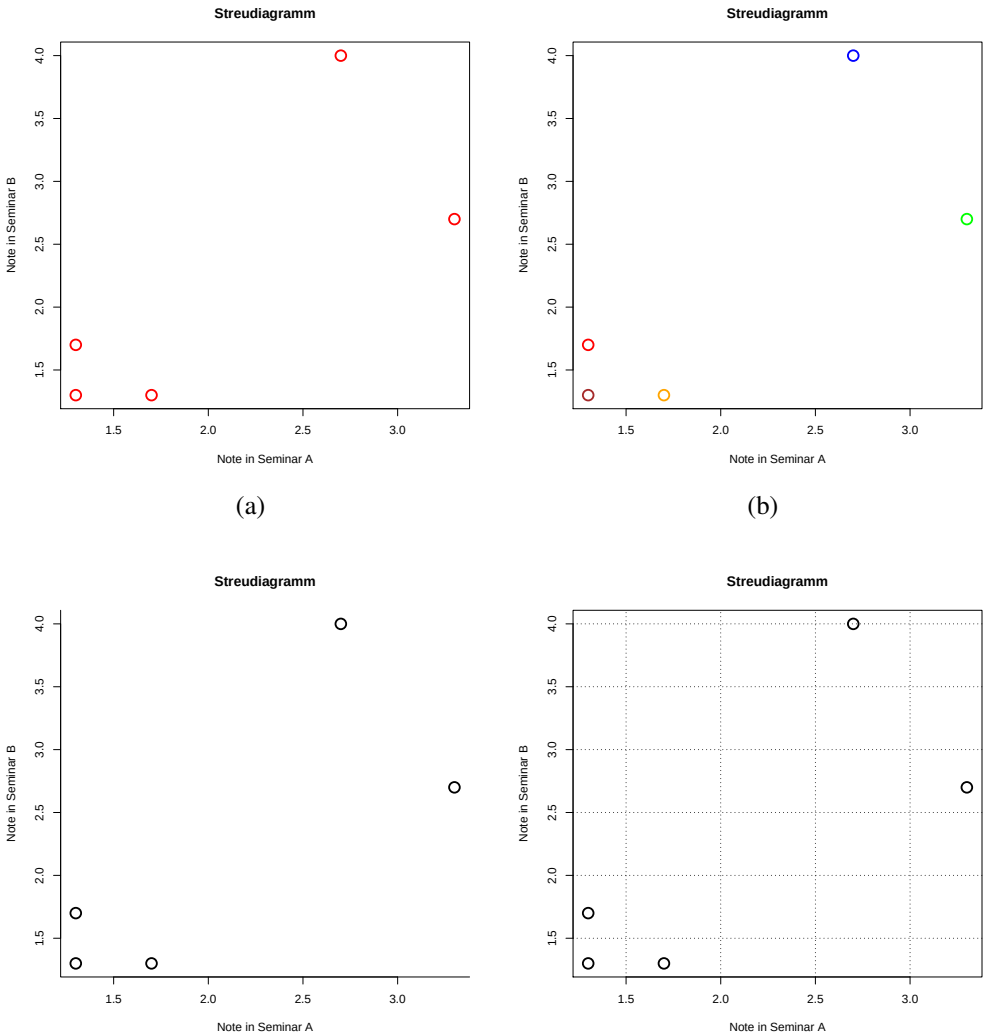


Abbildung 5.7: Weitere Beispiele für Streudiagramme (a) – (d)

gegebenen Symbols. Soll beispielsweise eine oben und rechts offene Box verwendet werden, wird bty="l" angegeben und man erhält Abbildung 5.7c:

```
> plot(dat$Seminar.A,dat$Seminar.B,main="Streudiagramm",
+      xlab="Note in Seminar A",ylab="Note in Seminar B",
+      cex=2,lwd=2.5,bty="l")
```

Ein weiteres nützliches Argument ist panel.first. Diesem Argument können Befehle übergeben werden, deren Resultate in die Grafik gesetzt werden, bevor die Symbole für die einzelnen Beobachtungen eingezeichnet werden. Ein Befehl, für den dies nützlich ist, ist grid(). Dieser Befehl zeichnet Linien in eine Grafik, die von den einzelnen *ticks* der x- und y-Achse ausgehen, was eine Art Raster ergibt und Grafiken einfacher lesbar macht. Wenn ein Grafikfenster geöffnet ist, kann dieser Befehl einfach direkt eingegeben werden, und das Raster wird in die Grafik aus dem aktuellen Grafikfenster eingefügt. Beispielsweise:

```
> plot(dat$Seminar.A,dat$Seminar.B)
> grid()
```

Die Linien des Rasters werden in diesem Beispiel allerdings gegebenenfalls über eingezeichnete Symbole gesetzt und überlagern diese gewissermaßen. Wird hingegen der Weg über das Argument panel.first verwendet, ist dies nicht der Fall. Ein möglicher Aufruf könnte lauten:

```
> plot(dat$Seminar.A,dat$Seminar.B,main="Streudiagramm",
+      xlab="Note in Seminar A",ylab="Note in Seminar B",
+      cex=2,lwd=2.5,panel.first=grid(col="black"))
```

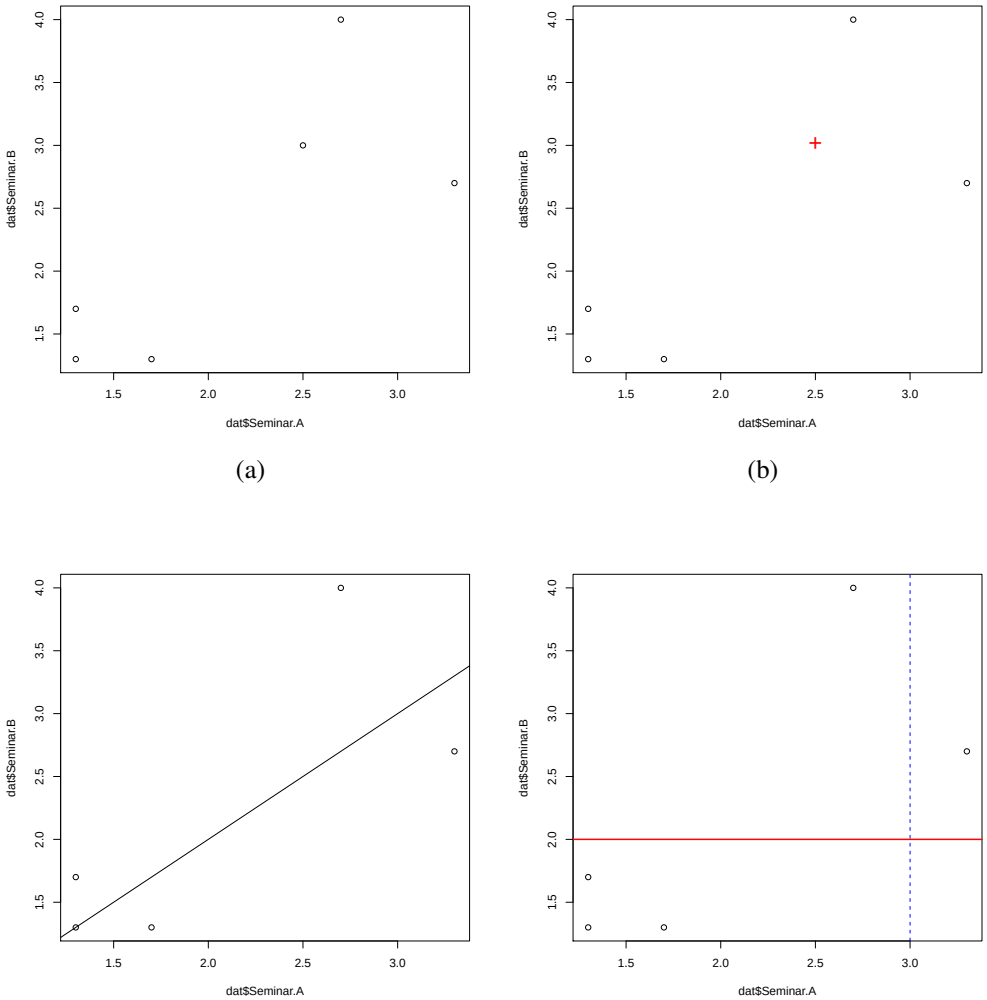
Das Ergebnis findet sich in Abbildung 5.7d. Innerhalb des Befehls grid() wurde das Argument col auf "black" gesetzt, da dieses für den betrachteten Befehl als Standard auf eine graue Farbe gesetzt ist, die bei der hier gewählten relativ kleinen Darstellung der Grafiken schwer zu erkennen ist.

5.2.2 Ergänzungen zu Streudiagrammen

Im letzten Unterabschnitt wurde mit dem grid() Befehl bereits eine Möglichkeit vorgestellt, über die in eine Grafik in einem offenen Grafikfenster zusätzliche Elemente eingezeichnet werden können. Beim genannten Beispiel war es ein Raster. Neben dem Befehl grid() gibt es noch einige weitere Befehle, die die Ergänzung von Grafiken ermöglichen.

Einer dieser Befehle ist points(). Über diesen können zusätzliche Symbole in eine Grafik eingezeichnet werden. Die x- und y-Koordinaten dieser Symbole werden wie beim plot()-Befehl auch über zwei Vektoren spezifiziert. Möchte man beispielsweise in ein einfaches Streudiagramm der Klausurdaten einen Punkt an der x-Koordinate 2,5 und der y-Koordinate 3 hinzufügen, gibt man folgendes ein:

```
> plot(dat$Seminar.A,dat$Seminar.B)
> points(2.5,3)
```



(a) (b)

(c) (d)

Abbildung 5.8: Ergänzungen zu Streudiagrammen (a) – (a)

Das Ergebnis ist in Abbildung 5.8a zu sehen. In dieser Grafik sind nun sechs anstelle von fünf Kreisen eingezeichnet.

Der points()-Befehl kann mit allen bisher kennengelernten Argumenten verwendet werden, die das Aussehen eingezeichneter Symbole steuern, wie beispielsweise pch, lwd, cex oder col. Die gewählte Darstellung der Symbole gilt dann nur für die über den points()-Befehl zusätzlich eingezeichneten Symbole, während die in der zu ergänzenden Grafik bereits vorhandenen unverändert bleiben (Abbildung 5.8b):

```
> plot(dat$Seminar.A, dat$Seminar.B)
> points(2.5,3,pch="+",cex=2,col="red")
```

Zu beachten ist, dass die Koordinaten von zusätzlich eingezeichneten Punkten im angezeigten Wertebereich der ursprünglichen Grafik liegen. Wenn also beispielsweise in der Grafik die y-Achse die Werte von 1 bis 5 abdeckt, sei es durch R automatisch gewählt oder durch ylim von Hand spezifiziert, einer der einzuzeichnenden Punkte allerdings einen y-Wert von 10 aufweist, dann wird dieser nicht in der Grafik erscheinen. Um dies dennoch zu ermöglichen, muss beim Aufruf des plot()-Befehls das Argument ylim entsprechend spezifiziert werden.

Neben Punkten lassen sich auch Linien in Grafiken ergänzen. Hierfür gibt es mehrere Möglichkeiten. Eine einfache besteht in der Verwendung des Befehls abline(). Soll eine Gerade mit Achsenabschnitt a und Steigung b eingezeichnet werden, wird ein Aufruf der Form abline(a,b) benutzt:

```
> plot(dat$Seminar.A, dat$Seminar.B)
> abline(0,1)
```

Das Resultat ist in Abbildung 5.8c dargestellt. In diesem Beispiel ist eine Gerade mit Achsenabschnitt 0 und Steigung 1 eingezeichnet. Punkte, die auf dieser Linie liegen weisen in beiden Seminaren die selbe Note auf.

Zum Einzeichnen von horizontalen oder vertikalen Linien reicht die Spezifikation eines Argumentes. Angegeben wird, an welchem Punkt die Gerade entweder die x- oder die y-Achse schneidet. Für horizontale Geraden wird der Schnittpunkt mit der x-Achse über das Argument h gesteuert, bei vertikalen Geraden wird v verwendet:

```
> plot(dat$Seminar.A, dat$Seminar.B)
> abline(h=2,col="red",lwd=2)
> abline(v=3,col="blue",lty=2)
```

Die resultierende Grafik findet sich in Abbildung 5.8d. Bei den beiden eingezeichneten Geraden wurden weitere Argumente spezifiziert. Die Argumente col und lwd sind bereits bekannt und steuern die Farbe und die Dicke der eingezeichneten Gerade. Das Argument lty steuert die Art der ausgegebenen Linie – lty=2 steht für eine gestrichelte Linie. Der Standardwert ist lty=1 und steht für eine durchgezogene Linie. Über lty=3 erhält man eine gepunktete Linie, ähnlich den Rasterlinien in Abbildung 5.7d, und über lty=4 erhält man eine Linie, die abwechselnd aus Punkten und Strichen besteht.

5.3 Grafiken speichern & weiterverwenden

Grafiken können in R auf verschiedene Art und Weise gespeichert werden. Eine Möglichkeit besteht darin, im Grafikfenster über die übliche Menüsteuerung „Speichern“ beziehungsweise „Exportieren“ zu wählen und hierüber auf Dateinamen, -pfad und -format anzugeben. Beim Dateiformat ist zu beachten, dass unterschiedliche Formate nicht die gleiche Qualität aufweisen. Insbesondere von der Verwendung von Dateien im JPEG-Format ist abzuraten. Vektorformate wie PDF oder PostScript sind wesentlich besser zur Weiterverwendung geeignet.

R bietet nebem dem menügeführtem Speichern die Möglichkeit, Grafiken über Befehle abzuspeichern. Hierbei wird allgemein in drei Schritten vorgegangen: über einen ersten Befehlsaufruf wird eine Grafikdatei vom gewünschten Typus geöffnet; hierauf folgen ein oder mehrere Grafikbefehle, die automatisch in die geöffnete Datei geschrieben werden; abschließend wird die Grafikdatei mit einem speziellen Befehl geschlossen und fertiggestellt.

Eine PDF-Datei lässt sich über den Befehl pdf() erstellen. Über das Argument file wird der Name und der Pfad der Datei festgelegt. Im einfachsten Fall reicht diese Angabe. Anschließend wird entsprechend des oben vorgestellten Schemas eine Grafik erzeugt. Mit dem Befehlsaufruf dev.off() wird die Datei geschlossen, wobei keine weiteren Argumente nötig sind. Ein Beispiel:

```
> pdf(file="C:/grafik/bild.pdf")
> plot(dat$Seminar.A,dat$Seminar.B)
> dev.off()
null device
          1
```

Bei diesem Beispiel wird eine PDF-Datei mit dem Namen bild.pdf im Verzeichnis C:\grafik angelegt. In diese Datei wird ein Streudiagramm der Noten in Seminar A und Seminar B abgelegt. Der letzte Aufruf führt zum Abschluss der Datei. Ohne diesen existiert die Datei zwar, kann allerdings nicht gelesen werden. Die abgeschlossene Datei kann anschließend weiterverwendet werden. Hierbei ist zu beachten, dass der Datei die richtige Dateiendung gegeben wurde, da es ansonsten bei manchen Betriebssystemen zu Schwierigkeiten bei der Verwendung der Grafik kommen kann.

Über die Optionen width und height wird die Größe der gespeicherten Grafik in Zoll festgelegt. Ein Zoll entspricht $2,54$cm. width steuert die Breite der Grafik und height die Höhe. Beide Optionen weisen als Standardwert 7 Zoll auf. Möchte man eine größere Grafik erzeugen, beispielsweise mit einer Höhe und Breite von 15 Zoll, lässt sich folgender Aufruf verwenden:

```
> pdf(file="C:/grafik/bild.pdf",width=15,height=15)
> plot(dat$Seminar.A,dat$Seminar.B)
> dev.off()
null device
          1
```

Zum Erzeugen von PostScript-Grafiken kann der Befehl postscript() verwendet werden. Für JPEG-Grafiken benutzt man jpeg(), für Bitmap-Dateien bmp(), für Dateien im PNG-Format png()

und für TIFF-Dateien kann tiff() benutzt werden. Der Befehl für die PostScript-Ausgabe kann analog zum Befehl pdf() verwendet werden:

```
> postscript(file="C:/grafik/bild.ps",width=15,height=15)
+ plot(dat$Seminar.A,dat$Seminar.B)
+ dev.off()
null device
          1
```

Die anderen gerade aufgelisteten Befehle unterscheiden sich leicht von der bisher verwendeten Form. Dateiname und -pfad werden über das Argument filename gesetzt. Die Optionen width und height stehen zwar wieder zur Verfügung, allerdings erfassen diese die Größe der Grafik nun standardmäßig in Bildpunkten und weisen den Standardwert 480 auf. Soll die Einheit geändert werden, kann man das Argument units benutzen. Hierbei kann "px" für Bildpunkte, "in" für Zoll, "cm" für Zenti- und "mm" für Millimeter angegeben werden. Wird eine andere Einheit als der Standard "px" benutzt, muss zusätzlich noch das Argument res angegeben werden, welches die nominale Auflösung der Grafik in dpi erfasst. Zumeist sollte es ausreichend sein, Höhe und Breite einfach über die Zahl der Bildpunkte festzulegen. Wird eine andere Einheit verwendet, sind in der Regel maximal 300dpi für ein qualitativ hochwertiges Ergebnis ausreichend. Der Befehl jpeg() erlaubt zusätzlich noch das Argument quality, welches die Komprimierung und hierüber die Qualität der zu erstellenden Grafik steuert. Der minimale Wert 0 steht für die schlechteste Qualität, aber am meisten komprimierte und somit kleinste Datei, während 100 für die beste Qualität steht.

Neben diesen Abweichungen bei einigen Argumenten kann wie bei den obigen Beispielen vorgegangen werden:

```
> jpeg(filename="C:/grafik/bild.jpeg",quality=100)
> plot(dat$Seminar.A,dat$Seminar.B)
> dev.off()
null device
          1

> bmp(filename="C:/grafik/bild.bmp",width=700,height=700)
> plot(dat$Seminar.A,dat$Seminar.B)
> dev.off()
null device
          1

> png(filename="C:/grafik/bild.png",width=9,height=9,units="in",res=70)
> plot(dat$Seminar.A,dat$Seminar.B)
> dev.off()
null device
          1

> tiff(filename="C:/grafik/bild.tiff")
> plot(dat$Seminar.A,dat$Seminar.B)
> dev.off()
```

```
null device
          1
```

Anstelle von der Benutzung von Befehlen besteht auch die Möglichkeit, Grafiken über das Grafikfenster zu speichern. Unter Windows ist dies durch anklicken von „Datei“ und „Speichern unter“ möglich.

6 Hypothesentests

Es gibt eine Vielzahl an Verfahren zum Testen von Hypothesen, die je nach Fragestellung und Testproblem zum Einsatz kommen. In diesem Kapitel werden nur einige ausgewählte Verfahren und deren Umsetzung in R behandelt, die üblicherweise in Einführungsveranstaltungen in die Statistik besprochen werden:

- t-Tests in diversen Varianten (inklusive Welchs t-Test)
- Wilcoxon-Tests in diversen Varianten (inklusive Mann-Whitney U-Test)
- χ^2-Methoden (inklusive McNemar-Test)

Zu jedem Ansatz wird lediglich eine kurze Einführung gegeben und im wesentlichen davon ausgegangen, dass die Verfahren und die zu Hypothesentests gehörenden Fachtermini bekannt sind. Detaillierte Erklärungen zu den hier besprochenen Methoden und Hinweise zu weiteren Verfahren findet man bei Kähler (2002) und insbesondere bei Bortz (2005) und Bortz et al. (2008). Eine knappe Übersicht liefern Sauerbier and Voß (2002). Ansätze, die bei kleinen Stichprobenumfängen eingesetzt werden können, werden umfassend von Bortz and Lienert (2008) behandelt.

6.1 t-Tests für intervallskalierte Variablen

6.1.1 Eine Stichprobe

Wiederholung t-Tests lassen sich bei diversen Testproblemen einsetzen. Im einfachsten Fall liegt aus einer Stichprobe eine intervallskalierte Variable X vor. Der Mittelwert dieser Variable $\bar{x}$ soll mit einem (ggf. rein hypothetischen) Wert verglichen μ werden. Hierüber soll überprüft werden, ob die Stichprobe aus einer Grundgesamtheit mit Mittelwert μ stammt. Die zu untersuchende Nullhypothese lautet also $H_0 : \bar{x} = \mu$. Zu überlegen bleibt, wie die Alternativhypothese formuliert werden soll. Hier bestehen drei Möglichkeiten:

1. $H_1 : \bar{x} \neq \mu$
2. $H_1 : \bar{x} > \mu$
3. $H_1 : \bar{x} < \mu$

Unabhängig von der Alternativhypothese wird davon ausgehend, dass die Standardabweichung der Variable X, s_x bekannt ist, die interessierende Prüfgröße t wie folgt berechnet:

$$t = \frac{\bar{x} - \mu}{s_x/\sqrt{n}} \tag{6.1}$$

wobei n den Stichprobenumfang bezeichnet. Bei großen Stichproben ist diese unabhängig von der eigentlichen Verteilung des Merkmals in der Stichprobe t-verteilt mit $n-1$ Freiheitsgraden.

Umsetzung in R Um das Verfahren in R anzuwenden, kann der Befehl t.test() benutzt werden. Zur Demonstration sollen die Klausurdaten dienen. Diese seien unter dem Namen dat im Workspace abgelegt:

```
> dat <- read.table("C:/daten/noten.csv",header=T)
```

Ausgehend von den Noten in Seminar A soll untersucht werden, ob die Studenten in unseren Daten eine Stichprobe aus einer Grundgesamtheit mit Mittelwert $\mu = 2$ sind. Die Alternativhypothese sei ungerichtet, also $H_1 : \bar{x} \neq 2$. Hierfür wird wie folgt vorgegangen:

```
> t.test(dat$Seminar.A,mu=2)

        One Sample t-test

data:  dat$Seminar.A
t = 0.1493, df = 4, p-value = 0.8886
alternative hypothesis: true mean is not equal to 2
95 percent confidence interval:
 0.9438829 3.1761171
sample estimates:
mean of x
     2.06
```

Der oben genannte Befehl t.test() wird verwendet. An diesen wird zunächst die interessierende Variable übergeben. Treten bei dieser fehlende Werte auf, werden diese bei der Berechnung automatisch ausgeschlossen. Der hypothetische Mittelwert der Grundgesamtheit wird über das Argument mu angegeben. Bei $\mu = 0$ muss dieses Argument nicht spezifiziert werden, da es als Standardwert 0 hat.

Bei der resultierenden Ausgabe wird zunächst angegeben, dass es sich um einen t-Test für eine Stichprobe handelt und es werden die verwendeten Daten angezeigt. Anschließend ist der resultierende t-Wert zu sehen. In diesem Beispiel beträgt er 0.1493 bei 4 Freiheitsgraden, welche als nächstes unter df ausgewiesen sind. Dies korrespondiert zu einer Überschreitungswahrscheinlichkeit von knapp 0.89. Gegeben einem üblichen Signifikanzniveau von $\alpha = 5\%$ kann die Nullhypothese also nicht verworfen werden. Neben diesen Angaben ist die angenommene Alternativhypothese angezeigt, Konfidenzintervalle für den Mittelwert sowie der Stichprobenmittelwert, der in diesem Beispiel bei 2.06 liegt.

Soll eine andere Alternativhypothese verwendet werden, kann dies über das Argument alternative geschehen. Dieses kann auf einen der drei Werte "two.sided", "greater" oder "less" gesetzt werden. "two.sided" ist der Standardwert und entspricht einer ungerichteten Alternativhypothese, weshalb das Argument im vorherigen Beispiel nicht spezifiziert werden musste. "greater" entspricht $H_1 : \bar{x} > \mu$ und "less" steht für $H_1 : \bar{x} < \mu$. Soll in unserem Beispiel die Alternativhypothese verwendet werden, dass der Mittelwert größer als 2 ist, wird folgendes eingegeben:

```
> t.test(dat$Seminar.A,mu=2,alternative="greater")

        One Sample t-test

data:  dat$Seminar.A
t = 0.1493, df = 4, p-value = 0.4443
alternative hypothesis: true mean is greater than 2
95 percent confidence interval:
 1.203008      Inf
sample estimates:
mean of x
     2.06
```

Der t-Wert sowie die Zahl der Freiheitsgrade bleibt unverändert, allerdings weist die bei p-value ausgegebene Überschreitungswahrscheinlichkeit einen deutlich anderen Wert auf, auch wenn sich die Interpretation bei einem Signifikanzniveau von $\alpha = 5\%$ nicht ändert. Zudem ist nun unsere neue Alternativhypothese ausgegeben und auch die Konfidenzintervalle wurden durch diese stark beeinflusst.

Das hier verwendete Beispiel dient lediglich der Demonstration des Vorgehens, denn eigentlich dürfte der t-Test nicht verwendet werden. Zum einen handelt es sich bei Seminarnoten nicht um ein intervallskaliertes Merkmal, zum anderen ist dieses weder normalverteilt, noch ist der Stichprobenumfang von der nötigen Größe.

6.1.2 Zwei unabhängige Stichproben

Wiederholung Beim t-Test für zwei unabhängige Stichproben soll überprüft werden, ob der Mittelwert eines Merkmals X aus der ersten Stichprobe $\bar{x}_1$ gleich dem Mittelwert der zweiten Stichprobe $\bar{x}_2$ ist. Die Nullhypothese lautet also $H_0 : \bar{x}_1 = \bar{x}_2$. Hier lassen sich dann relativ analog zum Fall einer Stichprobe drei Alternativhypothesen formulieren:

1. $H_1 : \bar{x}_1 \neq \bar{x}_2$
2. $H_1 : \bar{x}_1 > \bar{x}_2$
3. $H_1 : \bar{x}_1 < \bar{x}_2$

Je nachdem, ob davon ausgegangen wird, dass die Standardabweichungen der Stichproben s_{x1} und s_{x2} identisch sind, werden unterschiedliche Prüfgrößen berechnet und auch bei der Ermittlung der Zahl der Freiheitsgrade wird unterschiedlich vorgegangen.

Wird angenommen, dass $s_{x1} = s_{x2}$ gilt, wird der t-Wert berechnet über:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\left(\frac{1}{n_1} + \frac{1}{n_2}\right)\left(\frac{(n_1-1)s_{x1}^2+(n_2-1)s_{x2}^2)}{n_1+n_2-2}\right)}} \tag{6.2}$$

wobei n_1 den Umfang der ersten Stichprobe wiedergibt und n_2 steht entsprechend für den Umfang der zweiten Stichprobe. Die Zahl der Freiheitsgrade beträgt $n_1 + n_2 - 2$.

Kann nicht davon ausgegangen werden, dass die Standardabweichungen identisch sind, wird die Prüfgröße wie folgt berechnet:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_{x1}^2}{n_1} + \frac{s_{x1}^2}{n_1}}} \tag{6.3}$$

Verglichen mit der vorherigen Formel für den t-Wert ist diese deutlich einfacher, allerdings verkompliziert sich nun die Berechnung der Zahl der Freiheitsgrade deutlich:

$$\frac{\left(\frac{s_{x1}^2}{n_1} + \frac{s_{x2}^2}{n_2}\right)^2}{\frac{\left(\frac{s_{x1}^2}{n_1}\right)^2}{n_1-1} + \frac{\left(\frac{s_{x2}^2}{n_2}\right)^2}{n_2-1}} \tag{6.4}$$

Dieses Vorgehen wird nach seinem Entwickler oftmals als Welchs t-Test bezeichnet.

Umsetzung in R Zur Berechnung der beiden Varianten kann wieder der t.test() Befehl benutzt werden. Diesem werden nun einfach zwei Variablen übergeben und über das Argument var.equal kann durch Angabe eines Wahrheitswertes festgelegt werden, ob davon ausgegangen werden soll, dass die Standardabweichungen identisch sind. Als Standardwert wird var.equal=FALSE verwendet. Alternativhypothesen können wieder über das Argument alternative ausgewählt werden.

Als Beispiel verwenden wir wieder die Seminardaten und nehmen der Einfachheit halber an, dass die Noten aus den Seminaren A und B unterschiedlichen Studierenden zugerechnet werden können. Die zu untersuchende Frage wäre hier dann, ob die Studierenden aus Seminar A die selbe Durchschnittsnote wie Studierende aus Seminar B aufweisen. Zunächst wollen wir unterstellen, dass die Standardabweichungen identisch sind und anschließend diese Annahme aufgeben. Die Berechnung beider Varianten geschieht wie folgt:

```
> t.test(dat$Seminar.A, dat$Seminar.B, var.equal=T)

        Two Sample t-test

data:  dat$Seminar.A and dat$Seminar.B
t = -0.2136, df = 8, p-value = 0.8362
alternative hypothesis: true difference in means is not equal to 0
```

```
95 percent confidence interval:
 -1.651445  1.371445
sample estimates:
mean of x mean of y
     2.06      2.20

> t.test(dat$Seminar.A, dat$Seminar.B)

        Welch Two Sample t-test

data:  dat$Seminar.A and dat$Seminar.B
t = -0.2136, df = 7.538, p-value = 0.8365
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -1.667736  1.387736
sample estimates:
mean of x mean of y
     2.06      2.20
```

Die Ausgabe unterscheidet sich in beiden Fällen nicht sonderlich von der im Falle einer Stichprobe. Zunächst ist angegeben, um welchen t-Test es sich handelt. Hierauf folgend werden die verwendeten Daten angezeigt. Anschließend sind der Wert der Prüfgröße, die Zahl der Freiheitsgrade und die Überschreitungswahrscheinlichkeit zu sehen. Es folgt die Alternatichypothese, Konfidenzintervalle für die Differenz $\bar{x}_1 - \bar{x}_2$ sowie die konkreten Werte von $\bar{x}_1$ und $\bar{x}_2$.

In beiden Fällen kann die Nullhypothese, dass die beiden Mittelwerte identisch sind, nicht verworfen werden. Die Annahme, dass die Standardabweichungen identisch sind, hat bei diesem Beispiel nur einen minimalen Einfluss auf die Resultate.

Auch dieses Beispiel ist eigentlich wieder ungültig. Zum einen gelten die im vorherigen Abschnitt bereits genannten Probleme, zum anderen stammen die Daten nicht aus zwei unabhängigen Stichproben, sondern beziehen sich eigentlich auf die gleichen Studenten. In solch einem Fall kann der t-Test für zwei abhängige Stichproben benutzt werden.

6.1.3 Zwei abhängige Stichproben

Wiederholung Liegen wiederholte Beobachtungen des selben Merkmals bei den selben Individuen vor, kann der t-Test für zwei abhängige Stichproben verwendet werden, um zu überprüfen, ob sich diese Messungen signifikant voneinander unterscheiden. Wenn X_1 und X_2 die Variablen sind, die die beiden Merkmalswerte sind, wird für alle Beobachtungen die Differenz $d = X_1 - X_2$ gebildet. Anschließend werden der Mittelwert der Differenzen $\bar{d}$ und die Standardabweichung dieser s_d berechnet. Gegeben einer hypothetischen Differenz μ_d wird die Nullhypothese $H_0 : \bar{d} = \mu_d$ geprüft. Hierfür wird folgende Größe verwendet:

$$t = \frac{\bar{d} - \mu_d}{s_d / n} \tag{6.5}$$

Dabei entspricht n der Zahl der paarweisen Beobachtungen. Wurden beispielsweise 100 Personen zwei mal befragt, ist $n = 100$. Die Zahl der Freiheitsgrade ist gleich $n - 1$.

Umsetzung in R Zur Anwendung in R wird abermals der Befehl t.test() benutzt. Es kann relativ analog zu den beiden vorherigen Varianten vorgegangen werden, wobei das Argument paired auf den Wert TRUE gesetzt werden muss. Als Beispiel soll die Hypothese überprüft werden, dass die durchschnittliche Differenz der Noten in den Seminaren A und C nicht signifikant von Null abweicht, womit $\mu_d = 0$ ist. Hier kann wie folgt vorgegangen werden:

```
> t.test(dat$Seminar.A, dat$Seminar.C, paired=T)

        Paired t-test

data:  dat$Seminar.A and dat$Seminar.C
t = -0.2913, df = 4, p-value = 0.7853
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.8423857  0.6823857
sample estimates:
mean of the differences
                  -0.08
```

μ_d wurde hier nicht explizit spezifiziert, da hierfür wieder das Argument mu verwendet wird, welches den Standardwert 0 hat. Die Ausgabe entspricht in weiten Teilen den bisherigen Beispielen. In diesem Fall beträgt die Prüfgröße etwa -0.29 bei 4 Freiheitsgraden, was einer Überschreitungswahrscheinlichkeit von etwa 0.79 entspricht. Auch bei diesem Beispiel kann die Nullhypothese also nicht verworfen werden. Andere Alternativhypothesen hätte man wieder mit alternative festlegen können.

Auch bei diesem Beispiel stellt sich wieder das Problem, dass die Daten eigentlich nicht intervallskaliert sind.

6.2 Wilcoxon-Tests für ordinalskalierte Variablen

6.2.1 Zwei unabhängige Stichproben (U-Test nach Mann-Whitney)

Wiederholung Das in diesem Abschnitt vorgestellte Verfahren wurde zunächst von Wilcoxon entwickelt und anschließend von Mann und Whitney erweitert. In der Fachliteratur kursieren diverse Namen wie Wilcoxon-Rangsummentest, Wilcoxon-Mann-Whitney-Test oder Mann-Whitney U-Test. Wir werden hier relativ analog zur Bezeichnung in R vom Wilcoxon-Test für zwei unabhängige Stichproben sprechen.

Gegeben sind zwei Stichproben vom Umfang n_1 und n_2, wobei für beide Stichproben ein ordinalskaliertes Merkmal X beobachtet wurde. Alle beobachteten Werte werden nun in eine (in der Regel aufsteigende) Rangfolge gebracht, also der Reihe nach geordnet. Für jede der beiden

Stichproben werden nun die Ränge der Beobachtungen aufsummiert, wobei R_1 die Summe der Ränge in Stichprobe 1 und R_2 die Summe der Ränge in Stichprobe 2 sei. Sei R gleich dem kleineren der beiden Werte und n_R sei die Zahl der Beobachtungen in der entsprechenden Stichprobe. Dann wird die Prüfgröße U berechnet als:

$$U = R - n_R \frac{n_R + 1}{2} \tag{6.6}$$

Diese Größe ist bei großen Stichprobe näherungsweise normalverteilt. Nimmt man an, dass sich die beiden Stichproben nicht unterscheiden, lässt sich zeigen, dass U den Wert $n_1 n_2/2$ aufweisen sollte. Ferner sind alle möglichen U-Werte symmetrisch um diesen Wert verteilt mit Standardabweichung $\sqrt{n_1 n_2 (n_1 + n_2 + 1)/12}$. Hierüber lässt sich eine standardisierte Variante von U berechnen, welche standardnormalverteilt ist:

$$\frac{U - n_1 n_2/2}{\sqrt{n_1 n_2 (n_1 + n_2 + 1)/12}} \tag{6.7}$$

Hierüber lassen sich für den gefundenen U-Wert Überschreitungswahrscheinlichkeiten angeben unter der Nullhypothese $H_0 : U = n_1 n_2/2$. Alternativhypothesen lassen sich wie bisher spezifizieren. Wird die Nullhypothese verworfen, unterscheiden sich die beiden Verteilungen.

Für den Fall, dass sogenannte Bindungen auftreten, was bedeutet, dass sich mehrere Beobachtungen einen Rangplatz „teilen“, gelten andere Berechnungsvorschriften, die der Literatur entnommen werden können.

Umsetzung in R In R kann der Befehl wilcox.test() zur Umsetzung benutzt werden. Diesem werden im einfachsten Fall die Daten der zu vergleichenden Stichproben übergeben. Als Beispiel wollen wir wieder die Seminardaten verwenden und nehmen an, dass sich die Noten aus den Seminaren A und C auf unterschiedliche Studierende beziehen. Dann kann wie folgt vorgegangen werden:

```
> wilcox.test(dat$Seminar.A,dat$Seminar.C)

        Wilcoxon rank sum test with continuity correction

data:  dat$Seminar.A and dat$Seminar.C
W = 12.5, p-value = 1
alternative hypothesis: true location shift is not equal to 0

Warnmeldung:
In wilcox.test.default(dat$Seminar.A, dat$Seminar.C) :
  kann bei Bindungen keinen exakten p-Wert Berechnen
```

Die Ausgabe beginnt mit der Bezeichnung des Tests, wobei R eine spezielle Korrektur zur Berechnung der Überschreitungswahrscheinlichkeit verwendet, was mit continuity correction angegeben ist. Anschließend werden die verwendeten Daten angegeben. Der Wert der Prüfgröße

ist unter W zu finden und die zu diesem gehörende Überschreitungswahrscheinlichkeit unter p-value. In diesem Beispiel unterscheiden sich die Verteilungen also nicht signifikant.

Am Ende der Ausgabe findet sich eine Warnmeldung, die darauf hinweist, dass Bindungen aufgetreten sind. Dies ist insofern problematisch, als das R die Überschreitungswahrscheinlichkeit bei kleinen Stichprobenumfängen nicht über die oben genannten Formeln berechnet, die nur näherungsweise und bei großen Stichproben gelten, sondern eine exakte Wahrscheinlichkeit berechnet, die auch bei kleinen Stichproben Gültigkeit hat. Beim vorliegen von Bindungen kann diese exakte Wahrscheinlichkeit allerdings nicht berechnet werden, weshalb die Warnmeldung ausgegeben wird.

Dies kann über das Argument exact verhindert werden. Dieses steuert, ob die exakte Überschreitungswahrscheinlichkeit berechnet werden soll. Bei kleinen Stichprobenumfängen wird dieses Argument automatisch auf TRUE gesetzt, falls es nicht spezifiziert wird. Setzt man es in unserem Beispiel explizit auf FALSE wird die Überschreitungswahrscheinlichkeit approximativ berechnet und die Warnmeldung bleibt aus:

```
> wilcox.test(dat$Seminar.A, dat$Seminar.C, exact=F)

        Wilcoxon rank sum test with continuity correction

data:  dat$Seminar.A and dat$Seminar.C
W = 12.5, p-value = 1
alternative hypothesis: true location shift is not equal to 0
```

6.2.2 Zwei abhängige Stichproben

Wiederholung Der Wilcoxon-Test für zwei abhängige Stichproben kann angewendet werden, wenn beispielsweise für ein ordinalskaliertes Merkmal zwei Beobachtungen für die selben Individuen vorliegen. Wenn X_1 die erste und X_2 die zweite Messung erfasst, werden zunächst Differenzen $d = X_1 - X_2$ gebildet. Anschließend werden die absoluten Differenzen $|d|$ in eine Rangfolge gebracht. Nun werden zwei Summen gebildet: die Summe aller Ränge, für die die ursprüngliche Differenz positiv ist, und die Summe aller Ränge, für die die ursprüngliche Differenz negativ ist. Die kleinere der beiden Summen wird als W bezeichnet. Eine näherungsweise standardnormalverteile Prüfgröße lässt sich aus dieser berechnen über:

$$\frac{W - \frac{1}{4}n\,(n-1)}{\sqrt{\frac{n(n+1)(2n+1)}{24}}} \tag{6.8}$$

Hierüber lässt sich eine Überschreitungswahrscheinlichkeit für die Nullhypothese $H_0 : W = 0$ berechnen. Eine Ablehnung dieser Nullhypothese deutet darauf hin, dass sich die Mediane der beiden Messungen X_1 und X_2 signifikant voneinander unterscheiden.

Sind viele der Differenzen zwischen X_1 und X_2 gleich Null gehen diese nicht mit in die Berechnung ein. Trifft dies nur einen geringen Anteil der Beobachtungen kann dies ignoriert werden.

Ansonsten sind aber Korrekturen des Testverfahrens nötig, die in der einschlägigen Literatur zu finden sind.

Umsetzung in R Hier kann abermals der wilcox.test() Befehl benutzt werden. Dieser kann wie im vorherigen Abschnitt angewandt werden, wobei zusätzlich das Argument paired auf den Wert TRUE gesetzt werden muss. Soll die Gleichheit der Mediane der Noten in den Seminaren A und B untersucht werden, kann relativ analog zum letzten Beispiel vorgegangen werden:

```
> wilcox.test(dat$Seminar.A, dat$Seminar.C, paired=T, exact=F)

        Wilcoxon signed rank test with continuity correction

data:  dat$Seminar.A and dat$Seminar.C
V = 4.5, p-value = 1
alternative hypothesis: true location shift is not equal to 0
```

Auch hier wird wieder der Name des Tests ausgegeben. Der Wert der Prüfgröße W findet sich unter V und die Überschreitungswahrscheinlichkeit wieder unter p-value. In diesem Beispiel zeigt sich kein signifikanter Unterschied in den Mediannoten der beiden Seminare.

6.3 χ^2-Methoden für nominalskalierte Variablen

6.3.1 Eine Stichprobe

Wiederholung Im Falle einer Stichprobe kann der χ^2-Test eingesetzt werden um zu untersuchen, ob die Verteilung eines nominalskalierten Merkmals von einer hypothetisch gegebenen Verteilung abweicht. Sei n die Zahl der Beobachtungen in der Stichprobe, X die interessierende Variable und $\Pr(X = x)$ die hypothetische Wahrscheinlichkeit für Merkmalsausprägung x, wobei es insgesamt r Merkmalsausprägungen der Variable gibt. n_x gibt wieder, wieviele Individuen in der Stichprobe Merkmalsausprägung x aufweisen. Dann wird folgende Prüfgröße berechnet:

$$U = \sum_{i=1}^{r} \frac{(n_i - n\Pr(X = i))^2}{n\Pr(X = i)} \tag{6.9}$$

Dieser Wert ist χ^2-verteilt mit $r - 1$ Freiheitsgraden.

Umsetzung in R Um dieses Verfahren in R umzusetzen, kann der Befehl chisq.test() benutzt werden. Hierfür benötigen wir eine empirische und eine theoretische Verteilung, die an den Befehl übergeben werden. Als Beispiel verwenden wir wieder die Seminardaten und betrachten die Verteilung der Noten bei Seminar A. Diese lässt sich über den Befehl table() anzeigen:

```
> table(dat$Seminar.A)
```

```
1.3  1.7  2.7  3.3
  2    1    1    1
```

Zu bemerken ist, dass diese Tabelle nur vier Merkmalsausprägungen umfasst, also nicht alle möglichen Noten, was uns bei diesem Beispiel aber nicht stören soll.

Wollen wir die Hypothese untersuchen, dass die Studierenden gleichmäßig auf die vier Merkmalsausprägungen verteilt sind, müssen wir hierfür keine hypothetische Verteilung spezifizieren, da R dies als Standard setzt. Es reicht dann, einfach die Tabelle an den chisq.test()-Befehl zu übergeben:

```
> chisq.test(table(dat$Seminar.A))

        Chi-squared test for given probabilities

data:  table(dat$Seminar.A)
X-squared = 0.6, df = 3, p-value = 0.8964

Warnmeldung:
In chisq.test(table(dat$Seminar.A)) :
  Chi-Quadrat-Approximation kann inkorrekt sein
```

Wir bekommen wie bei den bisher kennengelernten Testverfahren ausgegeben, um welche Methode es sich handelt und welche Daten verwendet wurden. Der Wert der Prüfgröße ist unter X-squared angezeigt und beträgt in diesem Beispiel 0.6. Anschließend werden die Zahl der Freiheitsgrade und die Überschreitungswahrscheinlichkeit angezeigt. Die Hypothese, dass die empirische mit der hypothetischen Verteilung übereinstimmt, kann in diesem Beispiel nicht verworfen werden. Zu beachten ist allerdings, dass dieses Testverfahren für unser Beispiel eigentlich unzulässig ist, worauf auch die Warnmeldung hinweist: keines der Produkte $n\mathrm{Pr}(X = x)$ in der Formel zur Berechnung von U sollte kleiner als 5 sein. Dies ist in unserem Beispiel nicht der Fall.

Deshalb wollen wir ein zweites Beispiel betrachten, bei dem wir die empirische Verteilung nicht als Tabelle, sondern als Vektor eingeben. In einem großen Unternehmen wird ein Teil der Mitarbeiter zufällig ausgewählt und befragt. Insgesamt wurden 85 Frauen und 134 Männer erfasst. Die Frage sei nun, ob diese Verteilung mit einer hypothetischen Gleichverteilung der Geschlechter in Einklang zu bringen ist. In R übergibt man die empirische Verteilung als Vektor an den Befehl chisq.test(), wobei aufgrund der hypothetischen Gleichverteilung wieder keine theoretische Verteilung spezifiziert werden muss:

```
> chisq.test(c(85,134))

        Chi-squared test for given probabilities

data:  c(85, 134)
X-squared = 10.9635, df = 1, p-value = 0.0009293
```

Die Ausgabe entspricht dem vorherigen Beispiel, allerdings erscheint diesmal keine Warnmeldung. In diesem Fall kann die Hypothese der Gleichverteilung klar verworfen werden.

Angenommen aber, dass in unserem Beispielunternehmen gar keine Gleichverteilung der Geschlechter zu erwarten sei, da dieses zu einer Branche gehört, in der überproportional Männer beschäftigt sind. In diesem Fall müsste man eine hypothetische Verteilung spezifizieren, die dieses widerspiegelt. Beispielsweise seien in besagter Branche 40% der Beschäftigten weiblich und entsprechend 60% männlich. Geprüft werden soll nun, ob die empirische Verteilung im Beispielunternehmen mit dieser Verteilung übereinstimmt. Hierfür kann in R die theoretische Verteilung mittels des Arguments p als Vektor an den Befehl chisq.test() übergeben werden. Die Reihenfolge der theoretischen Anteilswerte sollte der der Merkmale im Vektor der empirischen Häufigkeiten entsprechen. In unserem Beispiel wäre dies also zunächst der Anteil an Frauen und dann der Anteil an Männern:

```
> chisq.test(c(85,134),p=c(0.4,0.6))

        Chi-squared test for given probabilities

data:  c(85, 134)
X-squared = 0.1286, df = 1, p-value = 0.7199
```

In diesem Beispiel kann die Hypothese, dass die Verteilungen übereinstimmen, nicht verworfen werden. Bei der Angabe der theoretischen Verteilung ist zu beachten, dass sich die Elemente des Vektors zu 1 summieren. Gibt man in unserem Beispiel die theoretische Verteilung beispielsweise in Prozentpunkten an (die sich zu 100 summieren) und nicht als Dezimalzahlen, erhält man eine Fehlermeldung:

```
> chisq.test(c(85,134),p=c(40,60))
Fehler in chisq.test(c(85, 134), p = c(40, 60)) :
  Wahrscheinlichkeiten müssen sich zu 1 addieren
```

Damit c(40,60) trotzdem an den Befehl übergeben werden kann, muss das Argument rescale.p auf den Wert TRUE gesetzt werden. Dies führt dazu, dass der über p angegebene Vektor so skaliert wird, dass sich die einzelnen Einträge zu 1 summieren und man erhält wieder das selbe Ergebnis wie oben:

```
> chisq.test(c(85,134),p=c(40,60),rescale.p=T)

        Chi-squared test for given probabilities

data:  c(85, 134)
X-squared = 0.1286, df = 1, p-value = 0.7199
```

6.3.2 Zwei unabhängige Stichproben

Wiederholung Sollen die Verteilungen eines nominalskalierten Merkmals in zwei unabhängigen Stichproben verglichen werden, wird ähnlich zur Problemstellung mit einer Stichprobe vorgegangen: es werden wieder beobachtete mit erwarteten Häufigkeiten verglichen. Hierfür wird die Randverteilung des betrachteten Merkmals in den zusammengelegten Stichproben

betrachtet. $Pr(X = x)$ sei der Anteilswert von Merkmalsausprägung x in den zusammengelegten Stichproben. n_{x1} sei die absolute Häufigkeit von Merkmalsausprägung x in der ersten Stichprobe und entsprechend n_{x2} die absolute Häufigkeit von Merkmalsausprägung x in der zweiten Stichprobe. n_1 sei die Zahl der Beobachtungen in Stichprobe 1, n_2 die Zahl der Beobachtungen in der zweiten Stichprobe. Die Zahl der Merkmalsausprägungen sei wieder gleich r. Dann wird die Prüfgröße berechnet als:

$$U = \sum_{i=1}^{r} \frac{(n_{i1} - n_1 \mathrm{Pr}(X = i))^2}{n_1 \mathrm{Pr}(X = i)} + \frac{(n_{i2} - n_2 \mathrm{Pr}(X = i))^2}{n_2 \mathrm{Pr}(X = i)} \tag{6.10}$$

Diese Größe ist wieder approximativ χ^2-verteilt mit $r - 1$ Freiheitsgraden.

Umsetzung in R Hier kann wieder der chisq.test() Befehl benutzt werden. Als Beispiel wollen wir wieder auf die Mitarbeiterbefragung zurückgreifen, bei der 85 Personen weiblich und 134 männlich waren. Es wurden nun zusätzlich Mitarbeiter in einem zweiten Unternehmen befragt. Insgesamt wurden 200 Personen befragt, 64 sind weiblich und 136 männlich. Überprüft werden soll die Hypothese, dass die Geschlechterverteilungen in den beiden Unternehmen übereinstimmen.

Hier muss zunächst überlegt werden, wie die Daten an den Befehl übergeben werden können, was über einen Vektor nicht möglich ist. Eine Möglichkeit besteht in der Verwendung einer Kreuztabelle oder alternativ einer Matrix. Da Matrizen erst später besprochen werden, wird hier die erste Möglichkeit genutzt. Die Tabelle wird über folgende Schritte erstellt:

```
> g1 <- c(rep(0,85),rep(1,134))
> g2 <- c(rep(0,64),rep(1,136))
> u1 <- rep(0,length(g1))
> u2 <- rep(1,length(g2))
> dat <- data.frame(c(g1,g2),c(u1,u2))
> names(dat) <- c("G","U")
> head(dat)
  G U
1 0 1
2 0 1
3 0 1
4 0 1
5 0 1
6 0 1
> tail(dat)
    G U
414 1 2
415 1 2
416 1 2
417 1 2
418 1 2
419 1 2
```

```
> tab <- table(dat$U,dat$G)
> tab

     0   1
  1 85 134
  2 64 136
```

Ganz am Ende liegt eine Kreuztabelle vor, deren Zeilen sich auf die beiden Unternehmen und deren Spalten sich auf die Geschlechter beziehen. Hierfür werden zunächst zwei Vektoren g1 und g2 erzeugt. g1 enthält für jede Person aus dem ersten Unternehmen deren Geschlecht, wobei 0 für weiblich und 1 für männlich stehen soll. Anschließend werden zwei Vektoren u1 und u2 erzeugt, die für jede Person anzeigen, zu welchem Unternehmen sie gehören – entweder Unternehmen 1 oder Unternehmen 2. Darauf wird der Befehl data.frame() benutzt, um diese Vektoren zu einem Datensatz zusammenzufassen. Zunächst werden die Vektoren mit den Geschlechtern und die Vektoren mit der Unternehmenszugehörigkeit „aneinandergeklebt". Diese beiden Vektoren werden nun in einem Datensatz als Datenspalten nebeneinander gelegt. Die Variable, die das Geschlecht erfasst, wird mit dem folgenden Aufruf als G bezeichnet, die Unternehmenszugehörigkeit wird mit U bezeichnet. Die ersten und die letzten sechs Datenzeilen werden durch die Befehle head() und tail() aufgerufen. Die Ausgaben verdeutlichen die Struktur des Datensatzes. Schließlich wird mittels des table() Befehls die eigentliche Kontingenztabelle erstellt und über den letzten Befehlsaufruf in der Box angezeigt.

Dieses Vorgehen ist etwas umständlich und lässt sich durch die Verwendung einer Matrix deutlich verkürzen:

```
> tab <- matrix(c(85,64,134,136),ncol=2)
> tab
     [,1] [,2]
[1,]   85  134
[2,]   64  136
```

Dieser Befehlsaufruf wird im entsprechenden Kapitel beschrieben.

Unabhängig davon, ob man eine Tabelle oder eine Matrix erzeugt hat, kann nun der Befehl chisq.test() angewendet werden:

```
> chisq.test(tab)

        Pearson's Chi-squared test with Yates' continuity correction

data:  tab
X-squared = 1.8305, df = 1, p-value = 0.1761
```

Hier wird zunächst wieder der Name des verwendeten Verfahrens angezeigt (zur Kontinuitätskorrektur s. bspw. Bortz (2005: 159)). Anschließend folgt ein Hinweis auf die verwendeten Daten. Der Wert der Prüfgröße ist wieder bei X-squared ausgegeben und beträgt hier 1.83. Bei einem Freiheitsgrad kann die Nullhypothese, dass die beiden Verteilungen übereinstimmen, nicht verworfen werden.

6.3.3 Zwei abhängige Stichproben (McNemar-Test)

Wiederholung Beim McNemar-Test wird die Hypothese überprüft, dass sich die Merkmalsausprägungen einer dichotomen Variablen von Personen, die zweimal befragt werden, nicht ändern. Die interessierende dichotome Variable X sei dabei binär kodiert (dies dient lediglich der Vereinfachung der Notation und ist allgemein nicht notwendig). $n_{0,0}$ sei die Zahl der Personen, die bei beiden Befragungen die Merkmalsausprägung 0 aufweisen. $n_{1,1}$ sei ganz analog die Zahl der Personen, die bei beiden Befragungen die Merkmalsausprägung 1 aufweisen. Dann wird folgende Prüfgröße berechnet:

$$M = \frac{(n_{0,0} - n_{1,1})^2}{n_{0,0} + n_{1,1}} \tag{6.11}$$

Diese ist näherungsweise χ^2-verteilt mit 1 Freiheitsgrad. Je höher der Wert der Prüfgröße, desto eher kann die Hypothese verworfen werden, dass es zwischen den beiden Befragungen keine Wechsel der Merkmalsausprägungen gab.

Umsetzung in R Zur Berechnung des Verfahrens kann der Befehl mcnemar.test() benutzt werden. Diesem kann entweder wie beim letzten Beispiel eine Kontingenztabelle beziehungsweise Matrix übergeben werden, oder aber zwei Vektoren. Dabei bezieht sich der erste Eintrag der beiden Vektoren immer auf die selbe Person und gibt für diese die Merkmalsausprägungen zu beiden Befragungen wieder. Hierbei können Vektoren entweder von Hand eingegeben werden oder aber Vektoren aus Datenmatrizen benutzt werden. Wir werden hier die Variante mit Vektoren nutzen und geben folgende ein:

```
> b1 <- c(rep(0,50),rep(1,50))
> b2 <- c(rep(0,30),rep(1,70))
```

Der Vektor b1 soll die erste Befragung erfassen, der Vektor b2 die zweite Befragung. Beide Vektoren haben die Länge 100 und die ersten 30 und die letzten 50 Einträge sind identisch. Bei diesen Beobachtungen kommt es zu keinem Wechsel der Merkmalsausprägung. Bei den Einträgen 31 bis 50 weist der Vektor b1 den Wert 0 auf, der Vektor b2 den Wert 1. Bei diesen Beobachtungen kommt es also zu einem Wechsel der Merkmalsausprägung.

Die beiden Vektoren können nun einfach an den Befehl mcnemar.test() übergeben werden, wobei die Reihenfolge unerheblich ist:

```
> mcnemar.test(b1,b2)

        McNemar's Chi-squared test with continuity correction

data:  b1 and b2
McNemar's chi-squared = 18.05, df = 1, p-value = 2.152e-05
```

Die Ausgabe wird wie bei den bisherigen Beispielen gelesen. Der Wert der Prüfgröße beträgt 18.05, die dazugehörige Überschreitungswahrscheinlichkeit liegt bei 0.00002, womit die Hypothese des Gleichbleibens der Merkmalsausprägungen verworfen werden kann.

Teil II

Fortgeschrittene Grafiken

7 Weitere Grafiktypen

7.1 Eine Variable

7.1.1 Kreisdiagramme

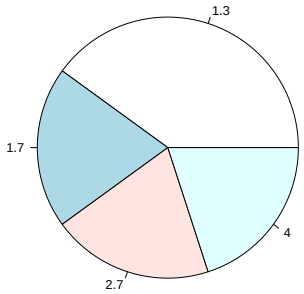
Kreis- oder Tortendiagramme (englisch: pie chart) sind zwar weit verbreitet, weisen jedoch einige Nachteile auf: a) Unterschiede zwischen den Anteilswerte sind weniger gut erkennbar, da dazu die Fläche der Kreissegmente verglichen werden muss, b) bei vielen Kategorien wird die Darstellung schnell unübersichtlich, c) sehr kleine Anteilswerte können oftmals nicht im Kreisdiagramm dargestellt werden. Aufgrund dieser Nachteile bietet sich die Verwendung von Kreisdiagramme nur in selten Fällen an – meist liefern Punkt- oder Balkendiagramme bessere Darstellungen (vgl. (Cleveland and McGill, 1984: 545)).

Um in R Kreisdiagramme zu erstellen, verwendet man den Befehl `pie()`. Diesem Befehl wird ein Vektor mit absoluten Häufigkeiten oder relativen Häufigkeiten übergeben, aus dem die Grafik erstellt wird. Entweder können solche Vektoren von Hand eingegeben werden, oder über Befehle wie `table()` erzeugt werden. Im folgenden Beispiel werden die Seminardaten verwendet, um ein einfaches Kreisdiagramm zu erstellen:

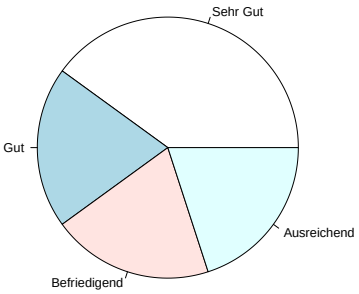
```
> dat$Seminar.B
[1] 1.7 2.7 4.0 1.3 1.3
> tab <- table(dat$Seminar.B)
> tab

1.3 1.7 2.7   4
  2   1   1   1
> pie(tab)
```

Zunächst werden die einzelnen Noten in Seminar B zur Erinnerung aufgerufen. Anschließend werden diese als Grundlage für eine Tabelle verwendet, die unter dem Namen `tab` abgespeichert und anschließend aufgerufen wird. Schließlich wird diese Tabelle einfach ohne weitere Argumente an den Befehl `pie` übergeben, wodurch man Abbildung 7.1b erhält.

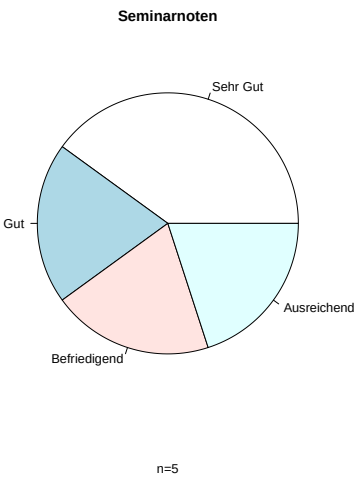
Um die Beschriftung der einzelnen Kategorien zu verändern, die sich momentan noch an den in den Daten vorkommenden Werten orientiert, kann man das Argument `labels` verwenden. Die Labels werden dabei der Reihe nach dem numerischen Vektor zugeordnet, der die Häufigkeiten enthält, aus denen die Grafik erzeugt wird. Beim vorherigen Beispiel war dies der Vektor `tab`. In diesem steht zunächst die Häufigkeit für die Note 1.3, dann für die Note 1.7 und so fort. In entsprechender Reihenfolge müssen die Labels in einem Vektor angegeben werden:



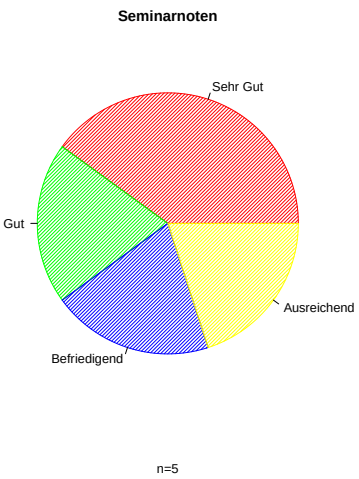
(a)



(b)



(c)



(d)

Abbildung 7.1: Abbildungen zu den Beispielen für Kreisdiagramme (a) – (d)

```
> pie(tab,labels=c("Sehr Gut","Gut","Befriedigend","Ausreichend"))
```

Hier wurden die üblichen Bezeichnungen für die Noten verwendet. Zu sehen ist das Ergebnis in Abbildung 7.1b.

Als nächstes werden über die bereits bekannten Grafikargumente main und sub noch eine Überschrift und eine Bildunterschrift hinzugefügt (s. Abb. 7.1c:

```
> noten <- c("Sehr Gut","Gut","Befriedigend","Ausreichend")
> pie(tab,labels=noten,main="Seminarnoten",sub="n=5")
```

Damit der Aufruf von pie() übersichtlich bleibt, wird zunächst ein Vektor namens noten erstellt, der die einzelnen Kategoriebezeichnungen enthält. Beim eigentlichen Befehlsaufruf wird dann dieser Vektor an das Argument labels übergeben. Dies hat ferner den Vorteil, dass man diesen Vektor auch für weitere Grafiken nutzen kann.

Schließlich nutzen wir das ebenfalls bereits bekannte Grafikargument col um die Farben zu verändern. Wir wollen anstelle der als Standard verwendeten Pastelltöne einige kräftigere Farben nutzen:

```
> farben <- c("red","green","blue","yellow")
> pie(tab,labels=noten,main="Seminarnoten",sub="n=5",density=30,col=farben)
```

Wie beim vorherigen Beispiel wird zunächst ein Vektor erstellt, der diesmal farben heißt. Dieser wird anschließend an das Argument col übergeben. Ferner wird noch das Argument density spezifiziert. Dieses kann verwendet werden, um zu erreichen, dass die Grafik nicht komplett mit Farbe ausgefüllt, sondern die Füllung gestrichelt eingezeichnet wird. Je höher der Wert, desto mehr Striche werden verwendet. Dieses Beispiel ist in Abbildung 7.1d zu sehen. Bei Verwendung des density Arguments ist darauf zu achten, dass ebenfalls das Argument col spezifiziert ist. Ist dies nicht der Fall, wird die Grafik nur in Schwarz und Weiß dargestellt.

7.1.2 Punktdiagramme

7.1.3 Dichteschätzer

7.2 Zwei Variablen

7.2.1 Sunflower-Plots

7.2.2 Box-Whisker-Plots

7.2.3 Spiderweb-Plots

8 Technische Grundlagen

8.1 Ausgabegeräte

Die Grafikausgabe in R erfolgt über sogenannte *graphic devices*, kurz Ausgabegeräte. Bereits in der Standardinstallation sind verschiedene Ausgabegeräte mit unterschiedlichen Aufgabengebieten und Fähigkeiten verfügbar. Einige sind plattformspezifisch wie X11, Quartz oder windows, andere wie pdf sind unter allen Betriebssystemen verfügbar.

Während sich pdf, xfig, tkiz oder svg auf die Dateiausgabe beschränken, verfügen die plattformspezifischen Ausgabegeräte meistens über die Fähigkeit, sowohl für die Bildschirmausgabe als auch für die Dateiausgabe zu sorgen.

Für LaTeX-Benutzer sind insbesondere die Ausgabegeräte tikz und pictex von Interesse, da mit ihnen LaTeX-Code für die tkizpicture- bzw. pictex-Umgebung generiert werden kann und die Grafiken sich so nahtlos in das LaTeX-Dokument einfügen. Eine nicht zwangsläufig vollständige Übersicht der verschiedenen Ausgabegeräte in R und die entsprechenden Pakete sind in Tabelle 8.1 aufgeführt.

Wird kein Ausgabegerät explizit aufgerufen, erfolgt die Grafikausgabe auf dem Standardgerät. Unter Windows ist dies windows, unter Mac OSX Quartz und unter Linux X11. Beim Aufruf der verschiedenen Ausgabegeräte können zusätztliche Optionen angegeben werden – diese werden im Folgenden für einige Ausgabegeräte vorgestellt.

8.1.1 windows

Die grafische Ausgabe erfolgt unter Windows automatisch über das windows-Device. Wird beispielsweise der plot() Befehl aufgerufen, erscheint das Ergebnis mittels dieses Ausgabegerätes. Dabei ist gewissermaßen immer nur eine Instanz des Ausgabegerätes aktiv – werden beispielsweise zwei plot() Aufrufe hintereinander ausgeführt, werden beide in diese Instanz geschrieben, was praktisch dazu führt, dass der zweite Aufruf den ersten überschreibt.

Über den Befehl dev.list() kann man eine Liste aller aktiven Ausgabegeräte anzeigen, wobei hierfür keine Argumente nötig sind. Ist kein Grafikfenster geöffnet, sollte das Ergebnis wie folgt aussehen:

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

Tabelle 8.1: Grafikausgabegeräte in R. Quelle: Murrel (2009)

Gerät	Betriebssystem	Verfügbarkeit
PostScript(& Bitmap)	alle	R-Base
pictex	alle	R-Base
pdf	alle	R-Base
xfig	alle	R-Base
tikz	alle	tikzDevice (CRAN)
Java	alle	RJavaDevice (CRAN)
GTK	alle	gtkDevice (CRAN)
Cairo	alle	cairoDevice (CRAN)
libgd	alle	GDD (RForge)
SVG	alle	RSvgDevice (CRAN)
X11 (& PNG & JPEG)	UNIX	R-Base
GNOME	UNIX	R-Base
windows	Windows	R-Base
proxy	Windows	R-Base
Quartz	MacOS X	R-Base

NULL, also ein leeres Objekt, zeigt an, dass kein Ausgabegerät aktiv ist. Wir laden nun die Seminardaten, erstellen eine einfache Grafik und rufen den Befehl erneut auf, wobei dass durch den plot() Aufruf geöffnete Grafikfenster nicht geschlossen werden sollte:

```
> dat <- read.table("C:/noten.csv",header=T)
> plot(dat$Seminar.A,dat$Seminar.B)
> dev.list()
windows
      2
```

Der Aufruf von dev.list() zeigt uns nun an, dass ein windows Ausgabegerät aktiv ist. Dabei werden einzelne Ausgabegeräte durchnummeriert. Das im Beispiel aktive hat die Nummer 2. Die Nummer 1 ist für das sogenannte null device reserviert, welches eine Art Platzhalterfunktion hat und aus technischen Gründen implementiert ist. Wird das Grafikfenster geschlossen, sollte bei Aufruf von dev.list() wieder NULL als Ergebnis erscheinen. Dabei kann das Grafikfenster entweder durch einen Mausklick, oder aber durch den Befehl dev.off() geschlossen werden.

Über den Befehl windows() können zusätzliche Instanzen des windows Ausgabegerätes aktiviert werden. Dies ermöglicht es, mehrere Grafikfenster gleichzeitig geöffnet zu haben. Damit ein neues Ausgabegerät aktiviert wird, muss bereits ein erstes in Verwendung sein – also beispielsweise bereits ein Grafikfenster geöffnet sein. Im einfachsten Fall wird der Befehl dann einfach ohne Argumente ausgeführt:

```
> plot(dat$Seminar.A,dat$Seminar.B)
> dev.list()
windows
```

```
      2
> windows()
> plot(dat$Seminar.A, dat$Seminar.C)
> dev.list()
windows windows
      2       3
> dev.off()
windows
      2
> dev.off()
null device
          1
```

Zunächst wird mittels plot() eine einfache Grafik erstellt und wir vergewissern uns mittels des dev.list() Befehls, dass eine Instanz des Ausgabegerätes aktiv ist. Anschließend öffnen wir über windows() eine weitere Instanz und erstellen in diese einen zweiten Plot. Nun sollten zwei Grafikfenster geöffnet sein. Dies sieht man auch beim erneuten Aufruf des dev.list() Befehls – es sind zwei Instanzen des Ausgabegerätes windows aktiv, wobei eines die Nummer 2 und eines die Nummer 3 besitzt. Führen wir nun den Befehl dev.off() aus, wird das aktuell aktive Ausgabegerät geschlossen – dies ist windows 3. Die Ausgabe beim Aufruf dieses Befehls zeigt an, dass noch das Ausgabegerät windows 2 aktiv ist. Es wird also *nicht* ausgegeben, welches Ausgabegerät geschlossen wurde, sondern welche noch vorhanden sind.

Hat man mehrere Ausgabegeräte geöffnet, werden Grafikbefehle immer auf das aktive Ausgabegerät angewandt. Möchte man das aktive Ausgabegerät wechseln, sind die Befehle dev.cur() und dev.set() zu verwenden. Der erstgenannte Befehl wird ohne Argumente aufgerufen und gibt an, welches Ausgabegerät momentan aktiv ist. Durch den zweiten Befehl kann durch Angabe der Nummer eines Ausgabegerätes zu diesem Ausgabegerät gewechselt werden. Damit das folgende Beispiel funktioniert, sollte zu Beginn kein Grafikfenster aktiv sein:

```
> plot(dat$Seminar.A, dat$Seminar.B)
> windows()
> plot(dat$Seminar.A, dat$Seminar.C)
> dev.cur()
windows
      3
> dev.set(2)
windows
      2
> dev.cur()
windows
      2
> dev.off()
windows
      3
> dev.off()
null device
          1
```

Es werden wie beim Beispiel vorher zwei Ausgabegeräte geöffnet. Zunächst ist dann Ausgabegerät 3 aktiv. Durch Aufruf von dev.set(2) wird dann zu Ausgabegerät 2 gewechselt. Dieser Wechsel wird beim Aufruf von dev.cur() und auch beim Schließen der Ausgabegeräte mittels dev.off() deutlich.

Der Befehl windows() kann auch in Kombination mit eltichen Argumenten aufgerufen werden, die die Eigenschaften des Grafikfensters (und ggf. auch von Grafiken) beeinflussen. Beispielsweise lässt sich über das Argument pointsize die Größe der Beschriftung und von weiteren Grafikelementen steuern:

```
> windows(pointsize=5)
> plot(dat$Seminar.A,dat$Seminar.B)
```

Bei diesem Beispiel erhält man eine Grafik, bei der die Beschriftung und die Punkte im Streudiagramm relativ klein ausfallen. Weitere mögliche Argumente sind der Dokumentation des Befehls zu entnehmen.

8.1.2 Ausgabe in Dateien

Die Ausgabe in eine Datei kann auf zwei Arten erfolgen: entweder durch "kopieren" des Inhalts des Grafikfensters in eine Datei oder durch das direkte Schreiben in die Ausgabedatei.

Das Kopieren des Inhalts eines Grafikfensters geschieht über dev.copy(). Der Befehl verwendet dabei die folgenden Argumente:

```
dev.copy(device=Dateityp, file="Dateiname", width=x, height=y)
```

Als Dateityp sind unter Windows postscript, pdf, png oder jpeg zulässig. Die Rasterformate PNG und JPEG sollten nicht für Ausdrucke sondern nur für die Bilschirmdarstellung genutzt werden. Die Optionen width und height bestimmen die Dimensionen der Ausgabedatei. Bei den Rasterformaten sind die Angaben auf Pixel bezogen und bei PDF und Postscript wird die Breite und Höhe in Inch angegeben. Unter Windows besteht zusätzlich die Möglichkeit, den Inhalt des Grafikfensters über das Menü abzuspeichern oder direkt zu drucken.

Die zweite Variante zum Abspeichern von Grafiken ist besonders hilfreich, wenn Grafiken im Rahmen von Skripten erstellt werden sollen. Bei dieser Variante wird zuerst ein Grafikausgabegerät gestartet:

```
> pdf(file="C:/grafik/bild.pdf")
> plot(dat$Seminar.A, dat$Seminar.B)
> dev.off()
null device
1
```

Danach folgen die Plot-Anweisungen und erst wenn mit dev.off() die Datei geschlossen wird, ist der Inhalt auch auf die Festplatte geschrieben und kann betrachtet werden. Dieses Vorgehen wurde bereits in Kapitel 5 behandelt.

8.1.3 Cairo

Das R-Paket „Cairo“ stellt unter Windows und Unix ein Gerät für die Ausgabe von hochauflösenden PNG-, JPEG- und TIFF-Bitmap-Grafiken bereit sowie qualitativ hochwertigen PDF-, SVG- und PostScript-Dateien. Das Cairo-Device unterstützt für alle Ausgabeformate viele Grafikeigenschaften wie beispielsweise *alpha blending* oder *anti-aliasing*. Eine weitere Besonderheit ist die Unterstützung von im System installierten TrueType-Schriften.

Damit das Paket verwendet werden kann, muss es zunächst wie in Kapitel 2 beschrieben installiert und geladen werden:

```
> install.packages("Cairo")
> library(Cairo)
```

Ein interaktives Grafikfenster muss im Gegensatz zur Standardausgabe explizit aufgerufen werden. Unter Windows geschieht dies mit CairoWin() und unter Unix mit CairoX11(). Um unter Windows eine Grafik über dieses Ausgabegerät zu erstellen, geht man dann beispielsweise wie folgt vor:

```
> CairoWin()
> plot(dat$Seminar.A,dat$Seminar.B)
> dev.list()
Cairo
    2
> dev.off()
null device
          1
```

Ferner stellt das Paket einige Befehle zur Verfügung, um Grafiken in den oben genannten Formaten abzuspeichern. Dies sind:

```
CairoPNG()
CairoJPEG()
CairoTIFF()
CairoPDF()
CairoSVG()
CairoPS()
```

Diese Befehle funktionieren mehr oder weniger analog zu den in Kapitel 5 vorgestellten Befehlen zum abspeichern von Grafiken. Möchte man Beispielsweise eine Grafik im PNG-Format erstellen, geht man wie folgt vor:

```
> CairoPNG(file="C:/grafik.png",height=480,width=480,
+          units="px",bg="transparent")
> plot(dat$Seminar.A,dat$Seminar.B)
> dev.off()
null device
          1
```

Zunächst wird über CairoPNG() das entsprechende Ausgabegerät geöffnet. Als erstes Argument wird der Name und der Pfad der Datei angegeben, unter dem die Grafik gespeichert wer-

den soll. Anschließend wird über height und width die Höhe und Breite der Grafik festgelegt, wobei über units="px" festgelegt wird, dass diese Angaben in Bildpunkten gemacht werden. Schließlich wird über festgelegt, dass der Hintergrund transparent sein soll. Bei den anderen Befehlen zum Abspeichern von Grafiken wird im Prinzip analog vorgegangen. Beispielsweise lässt sich eine PDF-Grafik wie folgt erstellen:

```
> CairoPDF(file="C:/grafik.pdf",height=12,width=12)
> plot(dat$Seminar.A,dat$Seminar.B)
> dev.off()
null device
          1
```

Die Einheit, in der die Höhe und Breite der Grafik eingegeben wird, ist nun Zoll. Für weitere Details zu diesem und den weiteren oben genannten Befehlen sei hier auf die Dokumentation des Pakets verwiesen.

8.1.4 Welches Ausgabeformat ist das Richtige?

Die Vielzahl der möglichen Ausgabegeräte lässt natürlich die Frage aufkommen, welches Gerät sich für welche Aufgaben eignet. Generell lässt sich folgende Unterscheidung treffen: Für die Druckausgabe sollten immer Vektorgrafikformate verwendet werden und für die Bildschirmausgabe können die Bitmapformate genutzt werden.

Vektorformate wie PDF, Postscript oder SVG nutzen grafische Primitive wie Linien, Kreise oder allgemeiner Polygone und Splines zur Darstellung des Bildinhaltes. Dieses Vorgehen hat mehrere Vorteile: zum einem sind die resultierenden Dateien verhältnismäßig klein, da beispielsweise für eine gerade Linie nur der Start- und Endpunkt gespeichert werden müssen. Zum anderen lassen sich die Grafiken im Gegensatz zu Bitmapgrafiken ohne Qualitätsverlust skalieren. Welches der verschiedenen Vektorformate schließlich genutzt werden sollte, hängt vor allem von der zur Weiterverabeitung genutzen Software ab. Sollen die Grafiken beispielsweise in ein Microsoft Word Dokument eingefügt werden, bieten sich die Formate EPS oder WMF an, da diese nativ von Word unterstützt werden. Für LaTeX-Anwender eignen sich sowohl Postscript- und PDF-Dateien, wie auch die Ausgabegeräte tikzDevice oder pictex, wobei letztere direkt LaTeX-Code produzieren.

Für die Einbindung von Grafiken in Powerpoint- oder OpenOffice-Bilschirmpräsentationen können sämtliche Bitmapformate genutzt werden. Bitmap- oder Pixelgrafiken verwenden ein Raster von Bildpunkten, um den Bildinhalt darzustellen. Dies lässt sich gut mit einem Bildbearbeitungsprogramm nachvollziehen: stellt man die Vergrößerung auf die höchste Stufe ein, lassen sich die einzelnen farbigen Quadrate erkennen, aus denen das Bild zusammengesetzt ist. Dieser Effekt tritt auch leicht auf, wenn Bilder mit zu geringer Auflösung im Druck verwendeten werden. Bei der Skalierung werden die einzelnen Bildpunkte größer und können so schnell die Darstellung trüben. Auch bei der Verwendung in Bildschirmpräsentationen ist es sinvoll vorab zu überlegen, auf welchen physischen Ausgabegerät die Darstellung erfolgt. Die meisten Beamer arbeiten heute mit einer Auflösung von 1024 mal 786 Bildpunkten, eine Grafik die etwas

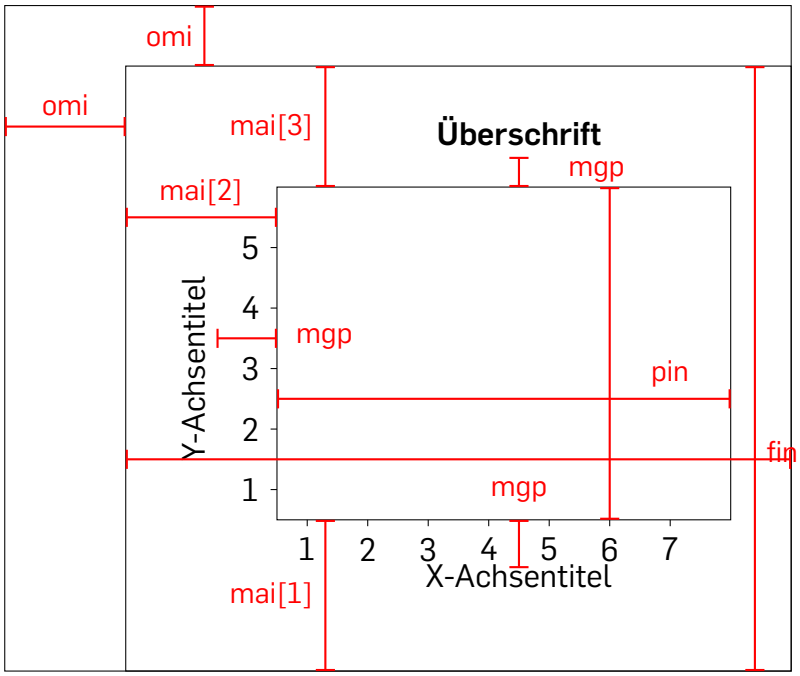


Abbildung 8.1: Bezeichnungen für Ränder und Abstände bei Grafiken

die Hälfte der Bildfläche einnimmt muss also mindestens 512 mal 393 Pixel groß sein, um in bester Qualität dargestellt zu werden. Ist die Bilddatei größer als die Auflösung des Ausgabemediums ergeben sich meist keine Probleme, da die Verkleinerung meist ohne sichtbaren Qualitätsverlust erfolgt. Lediglich die Dateigröße wird unnötigerweise erhöht. Die Frage welches der Bitmapformate verwendet werden sollte, richtet sich wieder nach der weiteren Verwendung. Generell ist das PNG-Format zu empfehlen, da es im Gegensatz zu JPEG Tranzparenz unterstützt und auch nicht wie GIF-Dateien in der Anzahl der nutzbaren Farben eingeschränkt ist.

8.2 Einstellungen für Ränder und Abstände

Die Grafikausgabe ist in verschiedene Bereiche eingeteilt, die jeweils eigene Einstellungen für Ränder und Bereichsgrößen haben. Die Bezeichnung der einzelnen Ränder und Abstände sieht man in Abbildung 8.1. Beispielsweise bezieht sich `fin` auf die Höhe *und* die Breite des kompletten Plots inklusive aller Elemente, während `omi` sich auf die Ränder um den Plot bezieht.

Die Einstellungen für diese Ränder und Abstände werden mit der Funktion `par()` für die gesamte Sitzung festgelegt für das aktuelle Ausgabegerät festgelegt. Dabei sind die einzelnen Abstände und Ränder in der Hilfe zu diesem Befehl dokumentiert. Um also beispielsweise Informationen über `omi` zu bekommen, muss `help(par)` aufgerufen werden.

Die einzelnen Abstände und Ränder werden diesem Befehl als Argumente übergeben, für die jeweils unterschiedliche Vektoren spezifiziert werden müssen. Beispielsweise muss für omi ein Vektor mit vier Werten angegeben werden. Die einzelnen Werte stehen der Reihe nach für den Rand unten, den Rand links, den Rand oberhalb der Grafik und den Rand rechts neben der Grafik. Es müssen numerische Werte angegeben werden, die den Rand in Zoll spezifizieren. Ein Beispiel:

```
> dat <- read.table("C:/daten/noten.csv",header=T)
> plot(dat$Seminar.A,dat$Seminar.B)
> windows()
> par(omi=c(1,1,1,1))
> plot(dat$Seminar.A,dat$Seminar.B)
> plot(dat$Seminar.A,dat$Seminar.C)
> dev.off()
> dev.off()
```

Zunächst werden die Seminardaten geladen, die hier wieder als Beispiel dienen. Anschließend wird ein Plot mit den üblichen Standardeinstellungen erstellt, der zum Vergleich in Abbildung 8.2a zu sehen ist. Durch den Befehl windows() wird ein zusätzliches Ausgabegerät geöffnet. Für dieses werden die Ränder um den Plot auf jeweils einen Zoll gesetzt. Die erste Grafik, die in den Plot gesetzt wird, sieht man in Abbildung 8.2b. Der dritte Aufruf des plot() Befehls wird ebenfalls in das Ausgabegerät mit den veränderten Rändern gesetzt und ist in Abbildung 8.2c zu sehen. Nun sollten zwei Grafikfenster offen sein – eines mit den Standardeinstellungen und eines mit den veränderten Rändern. Beide werden durch den wiederholten Aufruf von dev.off() geschlossen. Würde man nun wieder eine Grafik erstellen, würde sich ein neues Plotfenster öffnen, dass wieder die Standardeinstellungen aufweist.

Ein zweites Beispiel für die Anwendung des par() Befehls sollen die Ränder, in die die Beschriftungen der Grafik gesetzt werden, verändert werden. Hierfür wird das Argument mai benutzt, welches wie omi einen Vektor mit vier Einträgen verarbeitet, der die einzelnen Ränder in Zoll festlegt:

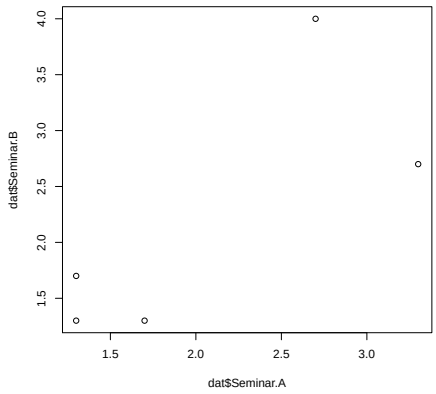
```
> par(mai=c(2,2,2,0.5))
> plot(dat$Seminar.A,dat$Seminar.B,main="Titel")
> dev.off()
```

Der Rand unter der Grafik wird auf 2 Zoll gesetzt, ebenso wie der Rand links neben und oberhalb der Grafik. Der Rand rechts neben der Grafik wird auf 0.5 Zoll festgelegt. Das Resultat sieht man in Abbildung 8.2d.

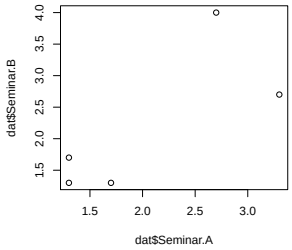
8.3 Farben & Formen

8.3.1 Definition von Farben

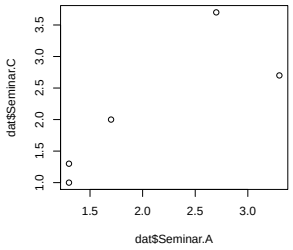
Bisher haben wir Farben über Namen angegeben – beispielsweise steht "red" für einen bestimmten Rotton. Farbangaben können aber auch in verschiedenen speziellen Farbmodellen



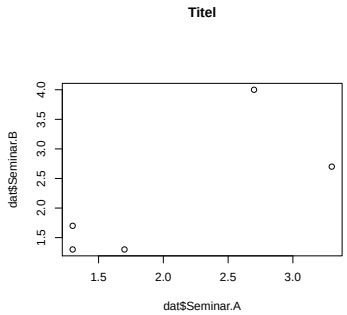
(a)



(b)



(c)



(d)

Abbildung 8.2: Abbildungen zu den Beispielen zu unterschiedlichen Rändern(a) – (d)

definiert werden. Eine übliche und geläufige Möglichkeit ist das RGBA-Model, in dem Farben aus unterschiedlichen Anteilen der Grundfaben Rot (R), Grün (G) und Blau (B) gemischt werden. Das A in RGBA steht für den Alpha-Kanal, der die Tranzparenz einer Farbe definiert. In R dient der Befehl rgb zur Definition von Farben nach diesem Farbmodel. Weitere Farbmodelle und die entsprechenden Funktionen sind hsv (Hue-Saturation-Value) oder hcl (Hue-Chroma-Luminance).

Unabhängig davon, mit welcher Funktion Farben definiert wurden, speichert R Farbwerte in hexadezimaler Form, wie sie beispielsweise auch in HTML-Seiten Verwendung finden. Dabei werden die Prozentangaben durch eine Zahl im Intervall von 0 bis 255 bzw. 0 bis FF beschrieben. Die Farbe Rot wird zum Beispiel durch "#FF0000" angegeben, in dezimaler Form entspricht dies (255,0,0) – also 100% Rot-Anteil und 0% Blau- und Grün-Anteil. Die meisten Bildverarbeitungsprogramme sind in der Lage, Farbwerte in hexadezimaler Form anzuzeigen, so dass auch eine „visuelle Farbmischung“ in diesen Programmen möglich ist.

Der oben eingeführte rgb() Befehl akzeptiert unter anderem die Argumente red, green und blue sowie alpha. Bei diesen Argumenten können Werte zwischen 0 und 1 angegeben werden. Je höher der Wert, desto stärker geht die entsprechende Grundfarbe mit in die Mischung ein. Beim Argument alpha bedeutet der Wert 0 völlige Transparenz, der Wert 1 steht für Intransparenz. Transparenz wird dabei nicht von allen Ausgabegeräten unterstützt. Mit dem folgenden Beispiel erhält man einen violetten Farbton:

```
> rgb(red=0.7,green=0,blue=0.9)
[1] "#B200E6"
```

Die Ausgabe entspricht dem Farbton in hexadezimaler Form, wie oben beschrieben. Möchte man diese Farbe in einer Grafik verwenden, kann man die Farbe entweder als Objekt speichern und an das entsprechende Grafikargument übergeben. Oder aber man kann den rgb() Befehl auch direkt verwenden:

```
> x <- c(1,2,3)
> y <- c(1,2,3)

> violett <- rgb(red=0.7,green=0,blue=0.9)
> plot(x,y,pch=15,col=violett)

> plot(x,y,pch=15,col=rgb(red=0.7,green=0,blue=0.9))
```

Zunächst werden drei Punkte spezifiziert, die grafisch dargestellt werden sollen. Beide folgenden plot() Aufrufe erzeugen die selbe Grafik. Im zweiten Fall ist der rgb() Befehl innerhalb des plot() Befehls geschachtelt, im ersten Fall wird vorab ein Objekt definiert, das die Farbdefinition enthält. Zu beachten ist hier, dass der Name des Objektes ohne Anführungszeichen angegeben werden muss.

8.3.2 Farbpaletten und Farbverläufe

Sollen mehrere Farben verwendet werden, empfiehlt sich die Verwendung von vordefinierten Farbpaletten, die aufeinander abgestimmte Farbtöne bereitstellen. In R bereits implementierte Farbpaletten sind rainbow, heat.colors, terrain.colors, topo.colors und cm.colors. . Farben aus diesen Paletten lassen sich aufrufen, indem die gleichnamigen Befehle aufgerufen werden, wobei in allen Fällen als Argument die Zahl der Farben anzugeben ist. Beispielsweise lassen sich 10 Farben aus der Palette rainbow aufrufen über:

```
> rainbow(10)
 [1] "#FF0000FF" "#FF9900FF" "#CCFF00FF" "#33FF00FF" "#00FF66FF" "#00FFFFFF"
 [7] "#0066FFFF" "#3300FFFF" "#CC00FFFF" "#FF0099FF"
```

Die Befehle lassen sich direkt in Kombination mit Grafikbefehlen verwenden, um Farben festzulegen:

```
> plot(1:10,1:10,pch=15,cex=2,col=rainbow(10),main="rainbow")
> plot(1:10,1:10,pch=15,cex=2,col=heat.colors(10),main="heat")
> plot(1:10,1:10,pch=15,cex=2,col=terrain.colors(10),main="terrain")
> plot(1:10,1:10,pch=15,cex=2,col=topo.colors(10),main="topo")
> plot(1:10,1:10,pch=15,cex=2,col=cm.colors(10),main="cm")
```

Die Ergebnisse sind in den Grafiken 8.3a bis 8.3e zu sehen.

Als praktisches Anwendungsbeispiel greifen wir auf ein Balkendiagramm für die Seminardaten aus Kapitel 5 zurück:

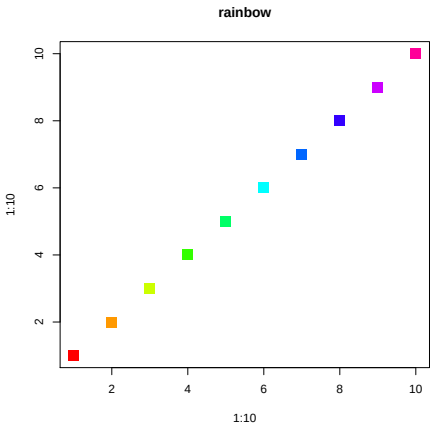
```
> dat <- read.table("noten.csv",header=T)
> tab <- table(dat$Seminar.A)
> barplot(tab,main="Seminarnoten",xlab="Note",ylab="Häufigkeit",
+         yaxp=c(0,2,2),col=cm.colors(4))
```

Hier wird die Farbpalette cm.colors genutzt, um Abbildung 8.3f zu erzeugen. Genauere Beschreibungen der ansonsten verwendeten Befehle findet man in Kapitel 5.

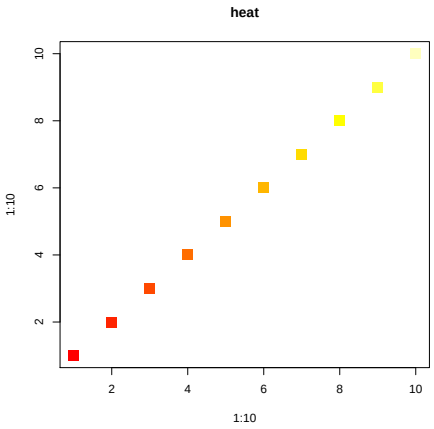
Das Paket RColorBrewer enthält weitere vordefinierte Farbpaletten. Bevor es benutzt werden kann, muss es über install.packages("RColorBrewer") installiert und über library(RColorBrewer) geladen werden. Der Befehle display.brewer.all() gibt eine Übersicht über die bereitsgestellten Paletten. Um eine dieser Farbpaletten zu nutzen, erzeugt man mit brewer.pal(n, name) einen Vektor mit n Farbwerten aus der jeweiligen Palette. So liefert der Aufruf von brewer.pal(11,"Spectral") folgende Farbwerte:

```
[1] "#9E0142" "#D53E4F" "#F46D43" "#FDAE61"
[5] "#FEE08B" "#FFFFBF" "#E6F598" "#ABDDA4"
[9] "#66C2A5" "#3288BD" "#5E4FA2"
```

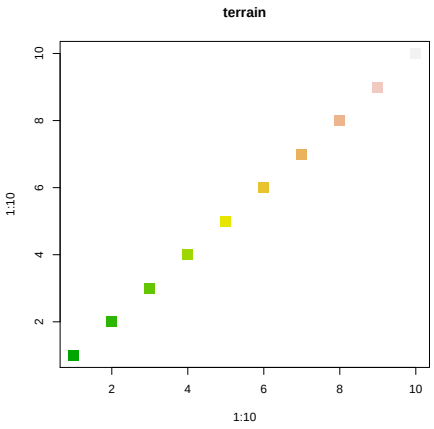
In der folgenden Grafik sind exemplarisch einige Farbpaletten dargestellt, an erster Stelle findet sich auch die soeben erstellte Palette Spectral:



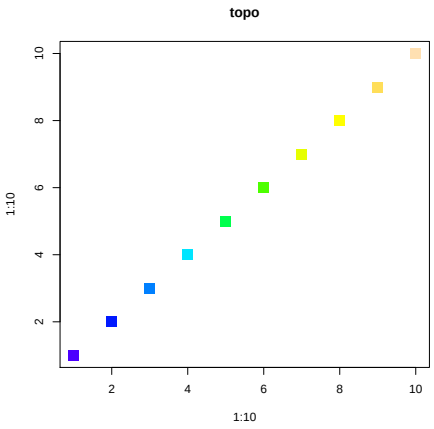
(a)



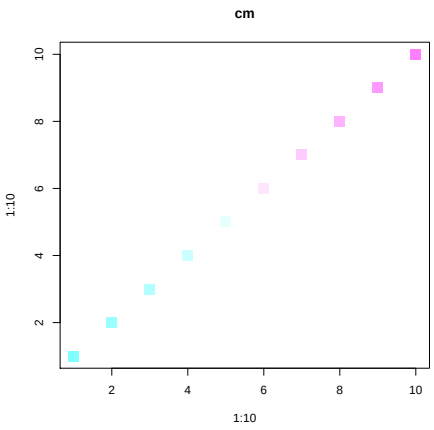
(b)



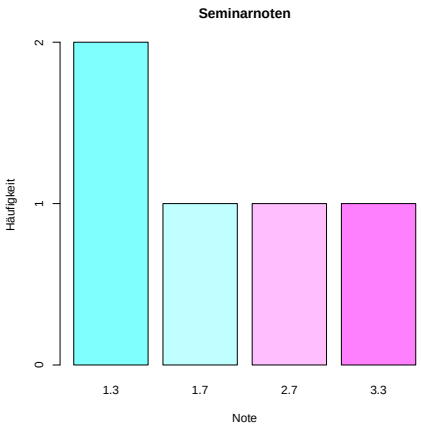
(c)



(d)

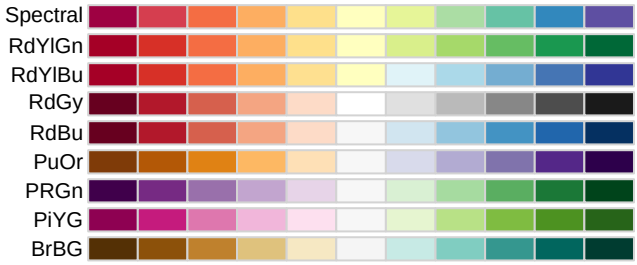


(e)



(f)

Abbildung 8.3: Abbildungen zu den Beispielen zu unterschiedlichen Farbpaletten(a) – (f)

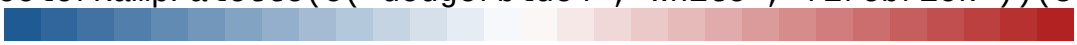


Farbverläufe lassen sich auf einfache Weise mit colorRampPalette() erstellen. Die Funktion hat ungewöhnlicherweise zwei Klammerpaare zur Angabe von Optionen. Das erste Klammerpaar nimmt die Start-, Zwischen- und Endfarben als Vektor der Form c("1. Farbe","2. Farbe") entgegen. Zusätzlich kann mit den Argumenten space="rgb" oder "lab" der Farbraum und mit interpolate="linear" oder "spline" die Berechnung der Farbverläufe beeinflusst werden. Im zweiten Klammernpaar wird anschließend die Anzahl der Farbstufen festgelegt. Hier einige Beispiele zur Verdeutlichung, wobei bei den einzelnen Befehlsaufrufen direkt die erzeugten Farbverläufe dargestellt werden:

```
colorRampPalette(c("black","white"))(30)
```

```
colorRampPalette(c("dodgerblue4","white"))(30)
```

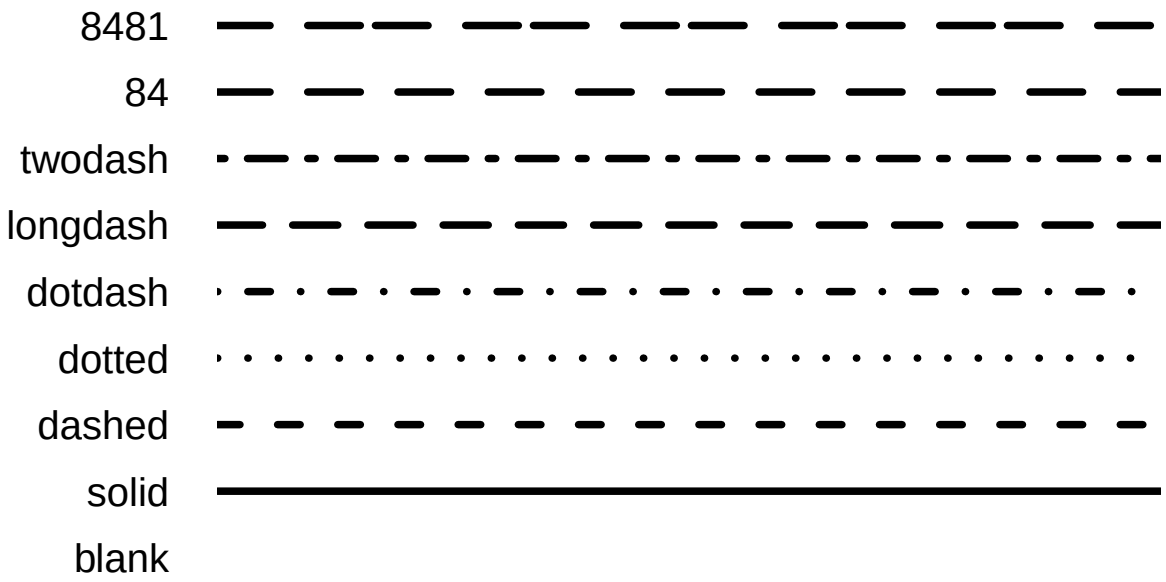
```
colorRampPalette(c("dodgerblue4","white","firebrick"))(30)
```



8.3.3 Symbol- und Linientypen

Die in R verfügbaren Symbole zum Einzeichnen von Punkten wurden bereits in Abbildung 5.6 im Kapitel 5 vorgestellt. Die Größe dieser Symbole wird über den bereits bekannten Parameter cex gesteuert. Der Standardwert ist cex=1, ein Wert von 0.5 halbiert die Größe der Symbole und ein Wert von 2 verdoppelt die Symbolgröße, wie hier dargestellt:

Linienstile werden über das ebenfalls bereits bekannte Argument lty ausgewählt. Die Grafik unten zeigt die standardmäßig verfügbaren Stile, die am einfachsten über ihren Namen ausgewählt werden können, wie beispielsweise lty="dashed". Linienstile können auch auf einfache Weise selbst definiert werden. Die ersten beiden Linien der Grafik zeigen selbstdefinierte Linientypen. Die erste Linie wird mit lty="8481" erstellt. Die Bedeutung der Zahlen ist einfach zu erkennen: Die erste 8 steht für einen acht Einheiten langen Strich, die 4 zeichnet eine vier Einheiten lange "Pause", die nächste 8 wieder eine Linie der entsprechenden Länge und die 1 erzeugt eine kurze Pause. Danach wird diese Abfolge für den Reste der zu zeichnenden Linie wiederholt:



Die einzelnen Linienstile können auch über numerische Werte aufgerufen werden. Anstelle von lty="dashed" lässt sich beispielsweise lty=2 angeben. Folgende numerischen Werte können verwendet werden:

1 solid

2 dashed

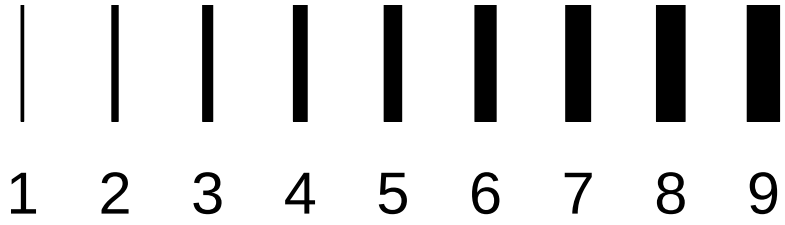
3 dotted

4 dotdash

5 longdash

6 twodash

Die Linienstärke lässt sich über das Argument lwd steuern. Dieser Parameter wird genauso wie cex behandelt und ist standardmäßig 1:



8.4 Schriften

Die in den Grafiken verwendeten Schriften lassen sich wie andere Standardeinstellungen ebenfalls ändern. In R sind drei Schriftschnitte direkt verfügbar: serif, sans und mono, wobei sans der Standardschriftart entspricht. Diese drei Schriftarten können über die Option family ausgewählt werden, welche beispielsweise bei den Befehlen plot() und text() angegeben werden kann:

```
> x <- c(1,2,3)
> y <- c(1,2,3)
> plot(x,y,main="Überschrift Sans",xlab="x-Achse",
+      ylab="y-Achse",family="sans")
> plot(x,y,main="Überschrift Serif",xlab="x-Achse",
+      ylab="y-Achse",family="serif")
> plot(x,y,main="Überschrift Mono",xlab="x-Achse",
+      ylab="y-Achse",family="mono")
> text(2.1,2.1,"Hier steht Text",family="serif")
```

Die Ergebnisse dieser Aufrufe sieht man in den Abbildungen 8.4a bis 8.4c.

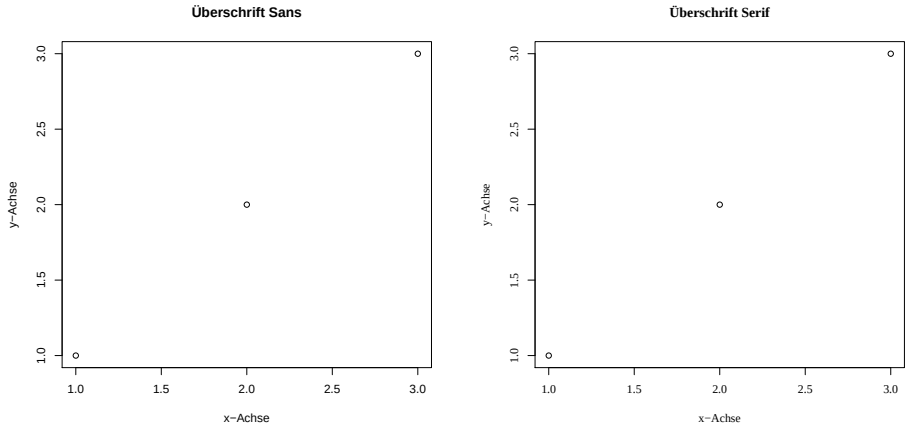
Unter Windows lassen sich zusätzlich auf einfache Weise im System installierte Truetype-Schriften nutzen. Dazu können mit der Funktion windowsFonts() Schriftfamilien definiert werden. Dazu muss jeweils der Name der neuen Schriftfamilie angegeben werden und die ausgewählte Schriftart:

```
> windowsFonts(WS="Arial Black")
> plot(x,y,main="Überschrift",xlab="x-Achse",ylab="y-Achse",family="WS")
```

Der Name der Schriftfamilie wird in diesem Beispiel als WS gewählt, wobei auch ein anderer Name hätte gewählt werden können. Als Schriftart wird Arial Black ausgewählt. Eine Übersicht über alle installierten Schriftarten findet man in der Systemsteuerung unter dem Punkt „Schriftarten“. Nachdem die Schriftfamilie definiert wurde, wird mit dieser eine Grafik erzeugt. Das Ergebnis sieht man in Abbildung 8.4d. Zu beachten ist, dass diese Grafik nicht als PDF- oder PostScript-Datei gespeichert werden kann, da die Schriftart bei diesen Formaten nicht zur Verfügung steht. Stattdessen kann man beispielsweise auf das PNG Format zurückgreifen.

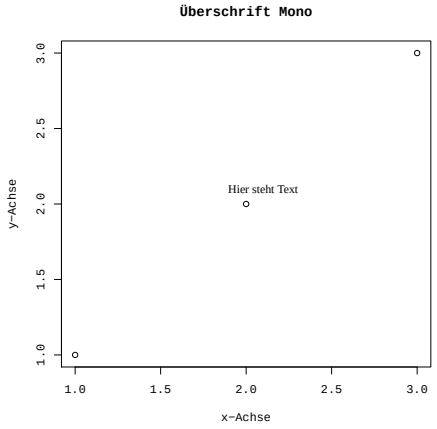
Man kann den Befehl windowsFonts() auch nutzen, um mehrere Schriftfamilien gleichzeitig zu definieren. Diese müssen dabei durch Kommata getrennt werden:

```
> windowsFonts(CS="Comic Sans MS",KS="My Underwood")
> plot(x,y,main="Überschrift",xlab="x-Achse",ylab="y-Achse",family="CS")
```

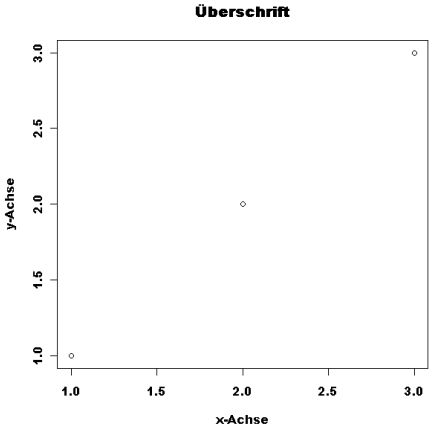


(a)

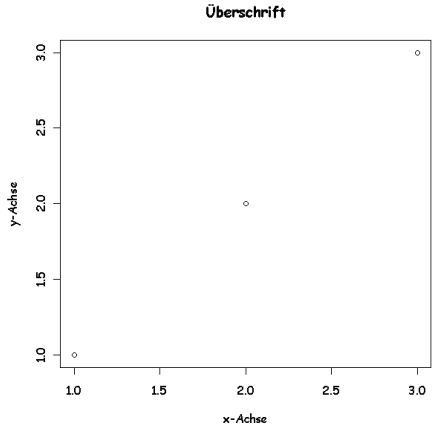
(b)



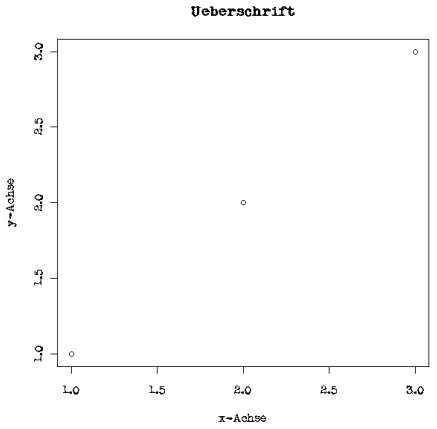
(c)



(d)



(e)



(f)

Abbildung 8.4: Abbildungen zu den Beispielen zu unterschiedlichen Schriftarten(a) – (f)

```
> plot(x,y,main="Ueberschrift",xlab="x-Achse",ylab="y-Achse",family="KS")
```

Bei diesem Beispiel werden die Schriftarten Comic Sans MS und MS Underwood benutzt. Zumindest letztere ist nicht standardmäßig bei Windows enthalten, sondern muss extra installiert werden. Beide Schriftarten werden für Grafiken genutzt, die man in den Abbildungen 8.4e und 8.4f sieht. Auch hier ist wieder zu beachten, dass diese Grafiken nicht im PDF- oder PostScript-Format gespeichert werden können.

9 Low-Level Grafiken

Mit den bisher vorgestellten Grafikbefehlen lassen sich vorgegebene Plottypen erstellen und ergänzen. Befehle, die mehr oder weniger fertige Grafiken erstellen, werden als „High Level“ Befehle bezeichnet. Als „Low Level“ Befehle bezeichnet man solche, die lediglich einzelne Elemente einer Grafik beeinflussen beziehungsweise erstellen. Beispiele sind grid() und abline(), die bereits in Kapitel 5 behandelt wurden. Über diese und ähnliche Befehle lassen sich Grafiken im Prinzip von Grund auf selbst gestalten, aber auch wie bei den genannten Beispielen können Grafiken ergänzt werden. Einige dieser Befehle werden in diesem Kapitel besprochen.

9.1 Plots und Achsen

Möchte man Grafiken selber von Hand erstellen, sollte zunächst ein komplett leeres Plot-Fenster geöffnet werden. In dieses werden anschließend Grafikelemente gesetzt. Zur Erstellung eines leeren Fensters wird der Befehl plot.new() ohne Argumente benutzt:

```
> plot.new()
```

Es sollte ein komplett weißes Grafikfenster erscheinen.

Achsen können nun über den Befehl axis() eingezeichnet werden. Dieser besitzt eine Vielzahl an Argumenten, die Position und Aussehen der zu zeichnenden Achse beeinflussen. Im einfachsten Fall könnte lediglich das Argument side spezifiziert werden. Dieses steuert, an welcher Seite der Grafik die Achse erscheint. Dabei steht der Wert 1 für unten, 2 für links, 3 für oben und 4 für rechts. Im folgenden Beispiel wird in das offene leere Grafikfenster eine Achse an die linke Seite gezeichnet:

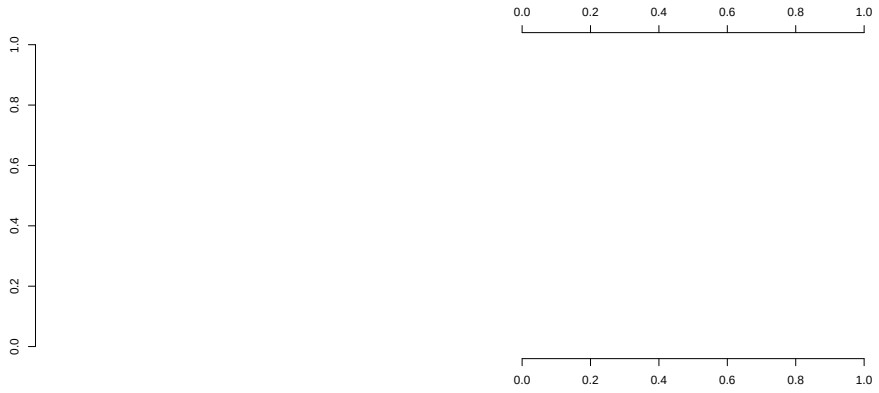
```
> axis(side=2)
```

Das Resultat sieht man in Abbildung 9.1a. Über diesen Befehl lassen sich auch mehrere Achsen in ein Grafikfenster zeichnen:

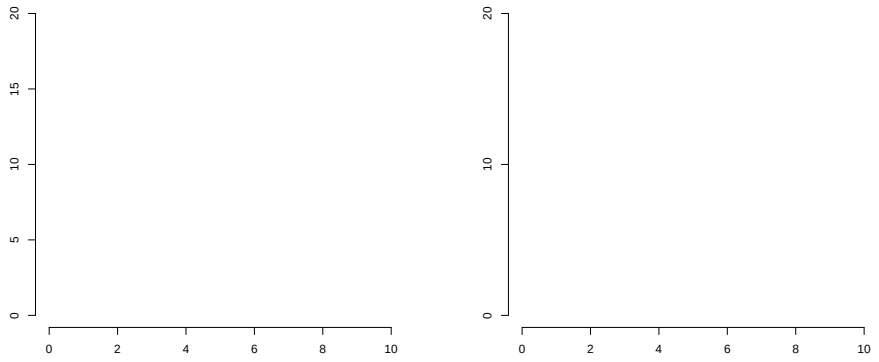
```
> plot.new()
> axis(side=1)
> axis(side=3)
```

Die resultierende Grafik ist in Abbildung 9.1b zu sehen und weist zwei Achsen auf. Wie in den beiden Beispielen gut ersichtlich, gibt es einen bestimmten Bereich der Grafik, der zwischen den Achsen liegt und zum einzeichnen von Punkten, Linien usw. dient.

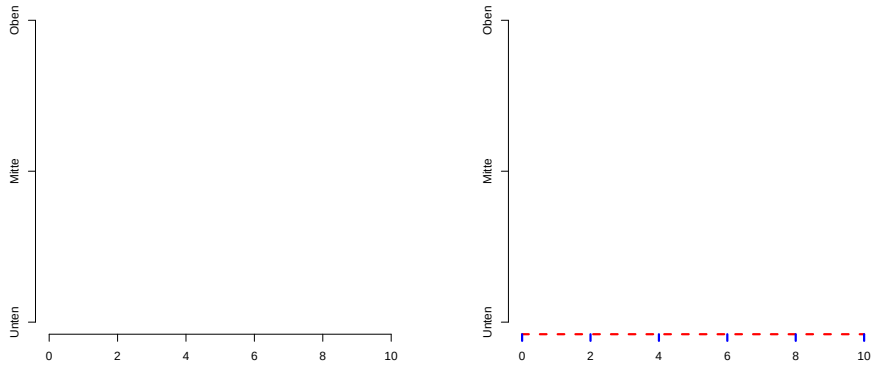
Wie ebenfalls aus diesen Beispielen ersichtlich, erstreckt sich dieser Bereich entlang der x-Achse von 0 bis 1, ebenso entlang der y-Achse. Um einen anderen Wertebereich abzudecken,



(a) (b)



(c) (d)



(e) (f)

Abbildung 9.1: Abbildungen zu den Beispielen zur Erstellung von Achsen (a) – (f)

muss der Befehl plot.window() benutzt werden und zwar nach dem plot.new() Befehl, bevor Achsen oder sonstige Grafikelemente eingezeichnet werden. Dieser besitzt die Argumente xlim und ylim, denen als Vektoren die Wertebereiche übergeben werden können. Soll die (nicht unbedingt eingezeichnete) x-Achse beispielsweise von 0 bis 10 reichen und die (ebenfalls nicht unbedingt eingezeichnete) y-Achse von 0 bis 20, kann wie folgt vorgegangen werden:

```
> plot.new()
> plot.window(xlim=c(0,10),ylim=c(0,20))
> axis(side=1)
> axis(side=2)
```

Das Resultat ist in Abbildung 9.1c zu sehen, wobei zwei Achsen zur Verdeutlichung des Effektes des Befehls eingezeichnet wurden.

Möchte man die Platzierung der Ticks der Achsen verändern, kann das Argument at benutzt werden. Über dieses wird ein Vektor mit Werten angegeben, an denen die Ticks gesetzt werden sollen. Möchte man beispielsweise die Grafik aus dem letzten Beispiel erstellen, allerdings an der y-Achse lediglich Ticks bei den Werten 0, 10 und 20, geht man wie folgt vor:

```
> plot.new()
> plot.window(xlim=c(0,10),ylim=c(0,20))
> axis(side=1)
> axis(side=2,at=c(0,10,20))
```

Das Ergebnis ist in Abbildung 9.1d zu sehen.

Über das Argument labels lässt sich die Beschriftung der Ticks ändern. Als Standard werden die numerischen Werte verwendet, an denen die Ticks gesetzt werden – im letzten Beispiel also 0, 10 und 20. Dem Argument labels wird ein Vektor mit Werten übergeben, die anstelle der numerischen Werte gesetzt werden sollen. Dabei muss der Vektor so viele Elemente enthalten, wie Ticks gesetzt werden sollen. Die einzelnen Elemente des Vektors werden dann der Reihe nach den Ticks zugeordnet. Bei unserem Beispiel müsste der Vektor insgesamt drei Werte enthalten. Der erste Wert wird anstelle der 0 gesetzt, der zweite anstelle der 10 und der dritte anstelle der 20. Bei diesen Werten kann es sich um Zahlen aber auch um Text handeln. Man könnte beispielsweise die 0 mit dem Text „Unten“, die 10 mit dem Text „Mitte“ und die 20 mit dem Text „Oben“ ersetzen:

```
> plot.new()
> plot.window(xlim=c(0,10),ylim=c(0,20))
> axis(side=1)
> axis(side=2,at=c(0,10,20),labels=c("Unten","Mitte","Oben"))
```

Wie in Abbildung 9.1e ersichtlich, ist nun der gewählte Text anstelle der numerischen Werte gesetzt.

Weitere nützliche Argumente, mit denen das aussehen der Achsen und der Ticks verändert werden können sind col, col.ticks, lty und lwd. Über die beiden erstgenannten Argumente lässt sich die Farbe der Achse und der Ticks festlegen. Mittels lty kann die zum Zeichnen der Achse verwendete Linienart festgelegt werden (beispielsweise gestrichelt). Die Dicke der Linie wird

über lwd gesteuert. Die Argumente sollten bereits aus dem Kapitel 5 bekannt sein. Ein Beispiel, in dem das Aussehen der x-Achse über diese Argumente verändert wird, ist in Abbildung 9.1f zu sehen:

```
> plot.new()
> plot.window(xlim=c(0,10),ylim=c(0,20))
> axis(side=1,lty=2,lwd=3,col="red",col.ticks="blue")
> axis(side=2,at=c(0,10,20),labels=c("Unten","Mitte","Oben"))
```

Weitere mögliche Argumente des axis() Befehls sind der Dokumentation zu entnehmen.

9.2 Punkte, Text und Linien

In Kapitel 5 wurden bereits einige Befehle vorgestellt, mit denen Punkte und Linien in Grafiken ergänzt werden können: points() und abline(). Diese werden hier nur der Vollständigkeit halber kurz wiederholt.

Über den Befehl abline() lassen sich einfach vertikal und horizontale Linien sowie Geraden mit beliebiger Steigung und beliebigem Achsenabschnitt einzeichnen:

```
> plot.new()
> plot.window(xlim=c(0,10),ylim=c(0,20))
> axis(side=1)
> axis(side=2)
> abline(h=1,lty=2,col="red")
> abline(v=9,lty=2,col="green")
> abline(a=0,b=1,lty=3,col="blue")
```

Bei diesem Beispiel wird zunächst ein leerer Plot mit x- und y-Achse erstellt. Anschließend werden eine horizontale, eine vertikale sowie eine Linie mit Achsenabschnitt 0 und Steigung 1 eingezeichnet, wie in Abbildung 9.2a

Mittels des Befehls points wiederum lassen sich beliebige Punkte in eine Grafik zeichnen:

```
> x <- c(2,1,8,5,3)
> y <- c(2,11,13,18,7)
> plot.new()
> plot.window(xlim=c(0,10),ylim=c(0,20))
> axis(side=1)
> axis(side=2)
> points(x,y,pch=2)
```

Das Ergebnis ist in Abbildung 9.2b zu sehen.

Eine weitere Variante, um Linien einzuzeichnen, bietet der Befehl lines(). Diesem werden einzelne Koordinaten eingegeben, die dann der Reihe nach mit einer Linie verbunden werden. Ein Anwendungsbeispiel könnte wie folgt aussehen:

```
> x <- 1:10
> y <- c(5,4,2,6,9,10,13,12,12,15)
> z <- y+2
```

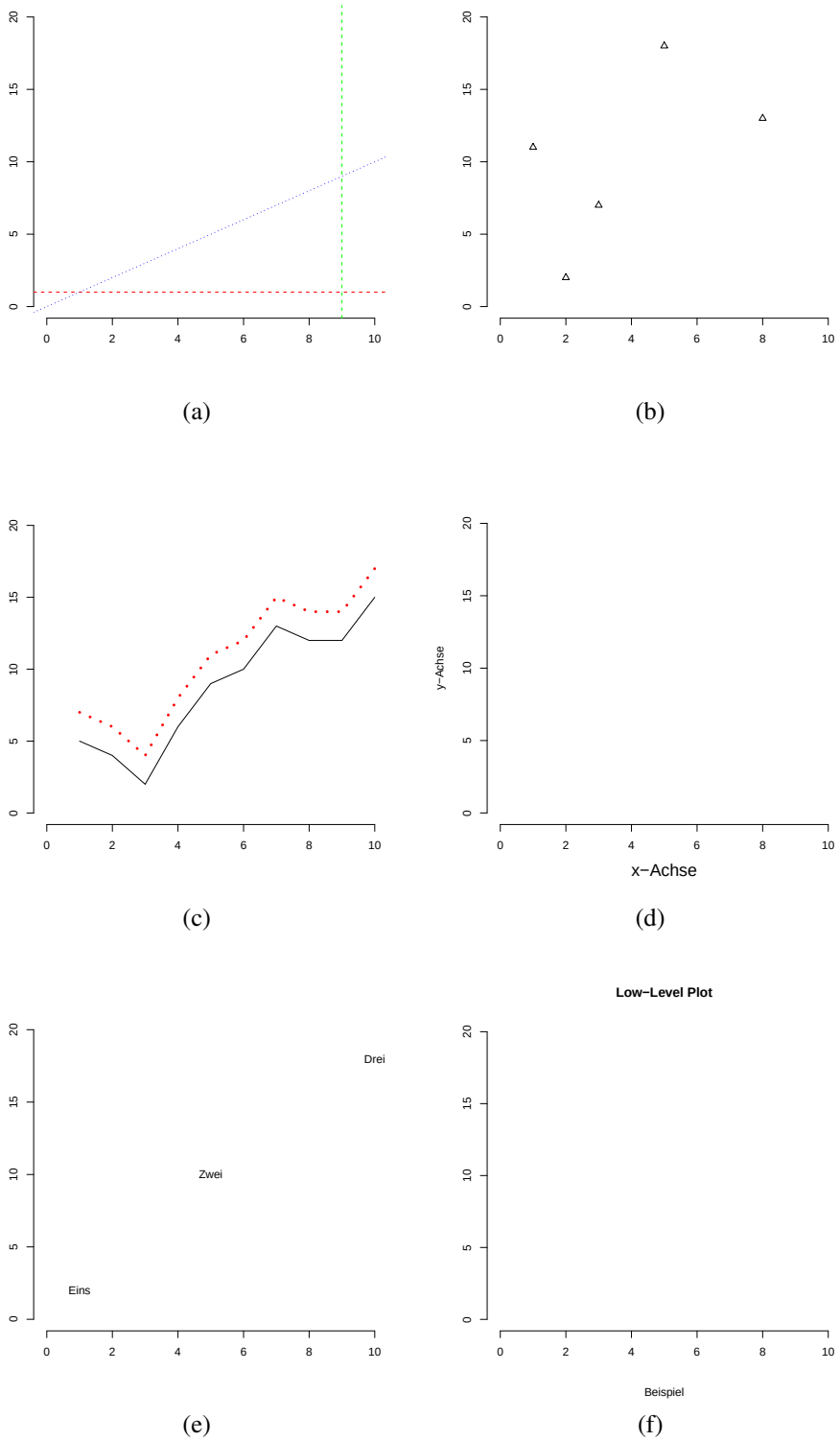


Abbildung 9.2: Abbildungen zu den Beispielen zum Einzeichnen von Punkten und Linien (a) – (c)

```
> plot.new()
> plot.window(xlim=c(0,10),ylim=c(0,20))
> axis(side=1)
> axis(side=2)
> lines(x,y)
> lines(x,z,col="red",lty=3,lwd=4)
```

Das Resultat findet man in Abbildung 9.2c. Zunächst werden drei Vektoren erstellt, aus denen sich hinterher die Koordinaten der Punkte ergeben. Der Vektor x enthält die x-Koordinaten der Punkte, der Vektor y die y-Koordinaten. Der Vektor z enthält die y-Koordinaten einer zweiten Linie, die die selben x-Koordinaten wie die erste haben wird. Anschließend wird mittels plot.new(), plot.window() und axis() ein neues Plotfenster erstellt, in dass die Linien eingezeichnet werden sollen. Zunächst werden die Vektoren x und y an den lines() Befehl übergeben. Dies erstellt die schwarze Linie in Abbildung 9.2c. Die erste Koordinate ist $(1, 5)$ und die zweite $(2, 4)$. Diese beiden werden durch eine Linie verbunden. Ebenso wie die dritte Koordinate $(3, 2)$ mit der zweiten und so fort. Der zweite Aufruf des lines() Befehls verdeutlicht einige Argumente zum Verändern des Aussehens der Linie. Über col="red" wird festgelegt, dass die Linie rot sein soll. Als Linientyp wird über Über lty=3 eine gestrichelte Linie verwendet. Schließlich wird mit Über lwd die Stärke der Linie verändert.

Text lässt sich über die beiden Befehle text() und mtext() ergänzen. Über mtext() kann Text an den Rändern der Grafik eingefügt werden. Die Positionierung dieses Textes wird unter anderem über side und line gesteuert. Mittels side wird ganz analog zum axis() Befehl festgelegt, ob der Text unterhalb (0), links (1), oberhalb (3) oder rechts (4) neben dem Plotbereich gesetzt werden soll. Über line kann der Abstand zur entsprechenden Achse gesteuert werden. Je höher dieser Wert, desto weiter ist der Text von der Achse weg. Ein Beispiel ist in Abbildung 9.2d zu sehen:

```
> plot.new()
> plot.window(xlim=c(0,10),ylim=c(0,20))
> axis(side=1)
> axis(side=2)
> mtext("x-Achse",side=1,line=3,cex=1.5)
> mtext("y-Achse",side=2,line=3)
```

Hier wird der Befehl dazu genutzt, um eine Beschriftung der Achsen zu erstellen. Die Beschriftung der x-Achse wird über das Argument cex noch zusätzlich um den Faktor 1.5 vergrößert.

Mittels des Befehls text() lässt sich Text an beliebige Stellen in der Grafik setzen. Hierfür müssen neben dem Text Koordinaten spezifiziert werden. Dies geschieht über die Argumente x und y. Diesen können entweder einzelne Werte übergeben werden oder aber Vektoren. Über Vektoren lassen sich mehrere Punkte spezifizieren, an die Text gesetzt wird. Der erste Eintrag des an das Argument x übergebenen Vektors bezieht sich auf die x-Koordinate des ersten Punktes, der zweite Eintrag auf die x-Koordinate des zweiten Punktes und so fort. Der an das Argument y übergebene Vektor enthält analog die y-Koordinaten der Punkte. Der eigentliche Text wird über das Argument labels festgelegt. Wurden über x und y mehrere Punkte spezifiziert, muss

auch an dieses Argument ein Vektor übergeben werden. Ein einfaches Beispiel könnte wie folgt gestaltet sein:

```
> plot.new()
> plot.window(xlim=c(0,10),ylim=c(0,20))
> axis(side=1)
> axis(side=2)
> text(x=c(1,5,10),y=c(2,10,18),labels=c("Eins","Zwei","Drei"))
```

Das Ergebnis sieht man in Abbildung 9.2e. An den Punkt $(1, 2)$ wird der Text „Eins“ gesetzt, an den Punkt $(5, 10)$ der Text „Zwei“ und an den Punkt $(10, 18)$ der Text „Drei“. Über das Argument col kann die Farbe des Textes festgelegt werden und über das Argument cex kann die Größe der Schrift skaliert werden.

Ist man sich bezüglich der Koordinaten, an denen Text oder ein anderes Grafikelement gesetzt werden soll, unsicher, kann der Befehl locator() eine große Hilfe sein, die viel ausprobieren ersparen kann. Dieser Befehl wird bei offenem Grafikfenster ohne Argumente eingegeben. Anschließend kann man mit einem Linksklick Koordinaten in der Grafik anklicken. Durch einen Rechtsklick wird der Befehl beendet. Anschließend werden die Koordinaten der angeklickten Punkte der Reihe nach ausgegeben. Ein Beispiel für eine solche Ausgabe:

```
> locator()
$x
[1] 3.112554 7.073593 4.367965

$y
[1] 11.501372  4.784708 18.761295
```

Der Befehl wird ausgeführt und anschließend drei Punkte im Plotfenster angeklickt. Nach einem Rechtsklick erscheinen die unter $x und $y angegebenen Informationen. Der erste angeklickte Punkt hat die x-Koordinate 3.112554 und die y-Koordinate 11.501372. Der zweite Punkt hat die Koordinaten 7.073593 und 4.784708. Die Koordinaten des dritten Punktes sind 4.367965 und 18.761295.

Zum erstellen von Überschriften kann der Befehl title() benutzt werden. Über das Argument main kann eine Überschrift oberhalb des Plots festgelegt werden. Eine „Unterüberschrift“, die unterhalb des Plots erscheint, kann man über sub festlegen:

```
> plot.new()
> plot.window(xlim=c(0,10),ylim=c(0,20))
> axis(side=1)
> axis(side=2)
> title(main="Low-Level Plot",sub="Beispiel")
```

Wie in Abbildung 9.2f zu sehen, wird bei diesem Beispiel „Low-Level Plot“ als Überschrift und „Beispiel“ als Unterüberschrift festgelegt. Die Erscheinung des Textes kann man unter anderem wieder mit col und cex verändern.

9.3 Rechtecke und Polygone

Der Befehl rect() erlaubt es, Rechtecke in ein Plotfenster zu zeichnen. Hierfür müssen insgesamt vier Koordinaten angegeben werden: die x-Koordinate der linken Seite des Rechtecks (über xleft), die x-Koordinate der rechten Seite des Rechtecks (über xright), die y-Koordinate der oberen Seite des Rechtecks (über ytop) und schließlich die y-Koordinate der unteren Seite (über ybottom). Ferner muss eines der Argumente border oder col angegeben werden, damit das Rechteck sichtbar wird. Über das erstgenannte Argument kann die Farbe der Außenlinien des Rechtecks festgelegt werden, über das letztere kann eine Füllfarbe bestimmt werden. Im folgenden Beispiel wird eine schwarze Außenlinie festgelegt und keine Füllung:

```
> plot.new()
> plot.window(xlim=c(0,10),ylim=c(0,20))
> axis(side=1)
> axis(side=2)
> rect(xleft=1,ybottom=2,xright=6,ytop=7,border="black")
```

Das Ergebnis ist in Abbildung 9.3a zu sehen.

Wird über col eine Füllfarbe spezifiziert, kann über zusätzliche Verwendung des Arguments density erreicht werden, dass diese das Rechteck nicht komplett ausfüllt, sondern gestrichelt erscheint, wie in Abbildung 9.3b zu sehen:

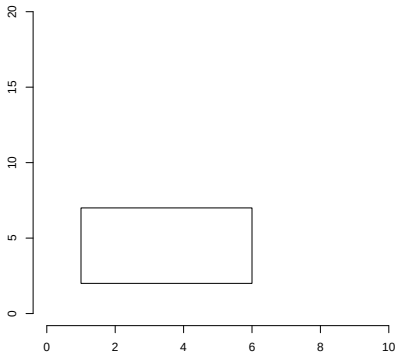
```
> plot.new()
> plot.window(xlim=c(0,10),ylim=c(0,20))
> axis(side=1)
> axis(side=2)
> rect(xleft=1,ybottom=1,xright=2,ytop=5,border="black",col="red")
> rect(xleft=4,ybottom=8,xright=7,ytop=11,border="black",col="blue",
+       density=20,lty=2)
```

Bei diesem Beispiel werden zwei Rechtecke eingezeichnet. Eines mit schwarzen Rahmen und roter Füllung und eines mit schwarzer Umrandung und einer gestrichelten, blauen Füllung. Niedrigere Werte für density führen dazu, dass weniger Linien innerhalb des Rechtecks erscheinen. Zusätzlich wurde über lty festgelegt, dass die Linien dieses Rechtecks gestrichelt sein sollen. Möchte man die Ausrichtung der Fülllinien verändern, kann das Argument angle benutzt werden, welches als Standardwert 45 aufweist. Dieser Wert entspricht dem Winkel der Linien.

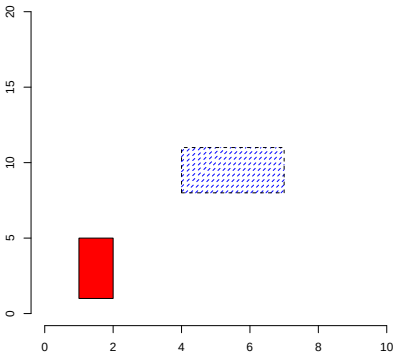
Polygone sind Vielecke, die man erhält, wenn man mindestens drei gegebene Punkte durch Linien miteinander verbindet, so dass eine zusammenhängende Fläche entsteht. In R kann man Polygone über den Befehl polygon() erstellen. Es können beliebig viele Punkte angegeben werden, um ein Polygon zu erstellen. Dies geschieht über zwei Vektoren, wobei der eine Vektor die x-Koordinaten und der andere Vektor die y-Koordinaten der Punkte enthält. Ansonsten können die selben Argumente wie bei rect() verwendet werden.

Bei folgendem Beispiel werden fünf Punkte für die Definition eines Polygons verwendet:

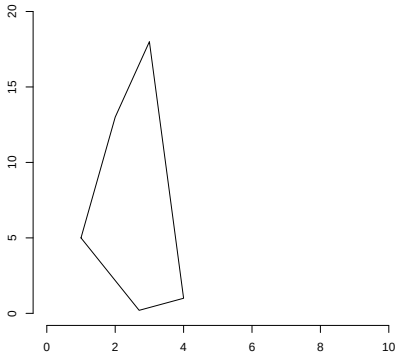
```
> x <- c(1,2,3,4,2.7)
> y <- c(5,13,18,1,0.2)
```



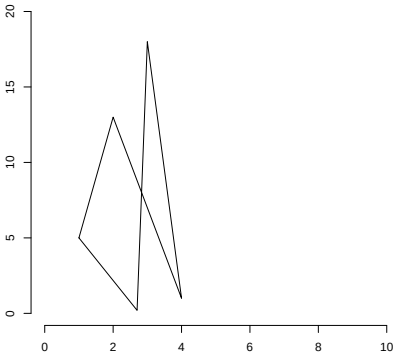
(a)



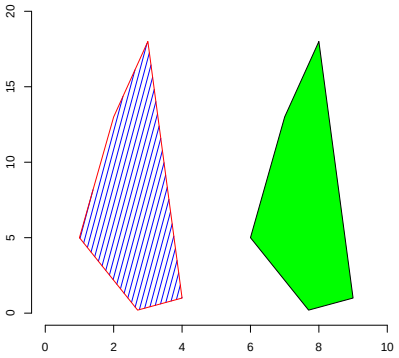
(b)



(c)



(d)



(e)

Abbildung 9.3: Abbildungen zu den Beispielen zu Rechtecken und Polygonen (a) – (e)

```
> plot.new()
> plot.window(xlim=c(0,10),ylim=c(0,20))
> axis(side=1)
> axis(side=2)
> polygon(x,y)
```

Das Ergebnis sieht man in Abbildung 9.3c. Zunächst werden zwei Vektoren erstellt, die die Koordinaten der Punkte erfassen. Der erste Punkt hat die Koordinaten $(1, 5)$, der zweite Punkt die Koordinaten $(2, 13)$ und so fort. Nach einigen Befehlen zum erstellen eines Plotfensters folgt der polygon() Befehl, dem einfach die beiden Vektoren übergeben werden. Zu beachten ist, dass die Punkte der Reihe nach verbunden werden. Als erstes wird eine Linie zwischen den ersten und den zweiten Punkt gezeichnet, dann zwischen den zweiten und dritten Punkt und so weiter bis zur letzten Linie, die den fünften Punkt mit dem ersten Punkt verbindet. Insofern ist die Reihenfolge der Punkte für das Aussehen des Polygons wichtig. Würde man beispielsweise den dritten mit dem vierten Punkt vertauschen, würde man Abbildung 9.3d erhalten.

Zwei Beispiele für Polygone, die weitere Argumente nutzen, die eine farbliche Gestaltung ermögliche, findet man in Abbildung 9.3e:

```
> x <- c(1,2,3,4,2.7)
> y <- c(5,13,18,1,0.2)
> z <- x+5
> plot.new()
> plot.window(xlim=c(0,10),ylim=c(0,20))
> axis(side=1)
> axis(side=2)
> polygon(x,y,border="red",col="blue",density=15,angle=75)
> polygon(z,y,col="green")
```

Die Argumente border, col, density und angle funktionieren wie beim rect() Befehl.

9.4 Ergänzung von Legenden

Um die Interpretation zu erleichtern, werden oftmals Legenden in Grafiken eingezeichnet. In R kann hierfür der Befehl legend() benutzt werden. Als Beispielgrafik, in die eine Legende eingezeichnet werden soll, verwenden wir hier Abbildung 9.2c, die über folgende Befehle erstellt wurde:

```
> x <- 1:10
> y <- c(5,4,2,6,9,10,13,12,12,15)
> z <- y+2

> plot.new()
> plot.window(xlim=c(0,10),ylim=c(0,20))
> axis(side=1)
> axis(side=2)
> lines(x,y)
```

```
> lines(x,z,col="red",lty=3,lwd=4)
```

Um eine Legende zu platzieren, werden zunächst Koordinaten benötigt, an die diese gesetzt werden soll. Diese werden über die Argumente x und y spezifiziert. Die gewählten Koordinaten entsprechen der linken oberen Ecke der Legende. Bei unserem Beispiel könnten wir x=0 und y=18 wählen. Über das Argument legend werden die Bezeichnungen der einzelnen Einträge der Legende festgelegt, wobei hierfür ein Vektor benutzt wird. Für unser Beispiel wählen wir legend=c("Linie 1","Linie 2"). Nachdem die Bezeichnungen festgelegt wurden, muss sich für Symbole für die Einträge entschieden werden. Dies geschieht über die Argumente pch und lty, wobei nicht unbedingt beide spezifiziert werden müssen. Sollen vor den Bezeichnungen in der Legende nur Linien erscheinen, reicht es, lty anzugeben. Diesem Argument muss ein Vektor übergeben werden, dessen einzelnen Einträge sich auf die Linientypen der einzelnen Einträge der Legende beziehen. In unserem Beispiel wählen wir lty=c(1,3). Vor dem ersten Eintrag der Legende mit der Bezeichnung „Linie 1“ wird eine durchgezogene Linie erscheinen, vor dem zweiten Eintrag eine gepunktete Linie. Würde der Plot nicht aus Linien, sondern aus Punkten bestehen, hätte man ganz analog das Argument pch spezifiziert. Über die Argumente lwd und col werden die Stärken und Farben der Linien festgelegt. Auch hier werden wieder Vektoren verwendet. Kombiniert sehen die genannten Argumente wie folgt aus:

```
> legend(x=0,y=18,legend=c("Linie 1","Linie 2"),
+        lty=c(1,3),lwd=c(1,3),col=c("black","red"))
```

Das Ergebnis ist in Abbildung 9.4a zu sehen. Ein Beispiel, welches anstelle von Linien Punkte (bzw. Symbole für Punkte) verwendet, sieht man in Abbildung 9.4b:

```
> legend(x=0,y=18,legend=c("Linie 1","Linie 2"),
+        pch=c(1,3),col=c("black","red"),cex=1.5)
```

Bei diesem Beispiel wurde ferner die Größe der Legende über das Argument cex um den Faktor 1.5 skaliert.

Möchte man verhindern, dass die in den Abbildungen 9.4a und 9.4b zu sehende Box um die Legende gezeichnet wird, muss das Argument bty auf "n" gesetzt werden:

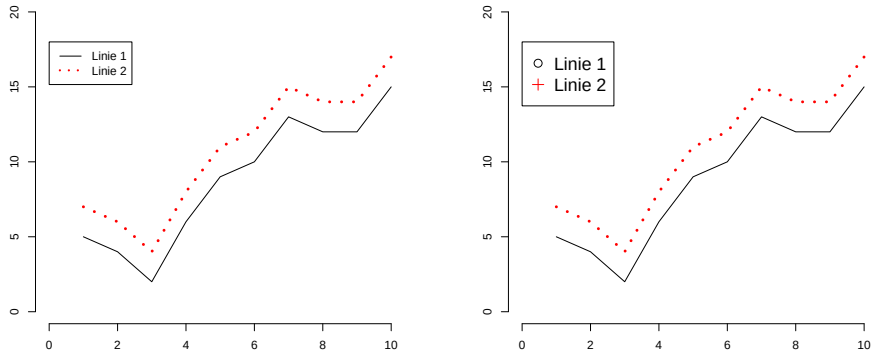
```
> legend(x=0,y=18,legend=c("Linie 1","Linie 2"),
+        lty=c(1,3),lwd=c(1,3),col=c("black","red"),bty="n")
```

Das Ergebnis sieht man in Abbildung 9.4c.

Weitere nützliche Argumente, mit denen sich das Aussehen der Legende steuern lässt, sind bg und title. Über das erstgenannte Argument kann die Hintergrundfarbe der Legende festgelegt werden und über das zweitgenannte kann ein Titel festgelegt werden:

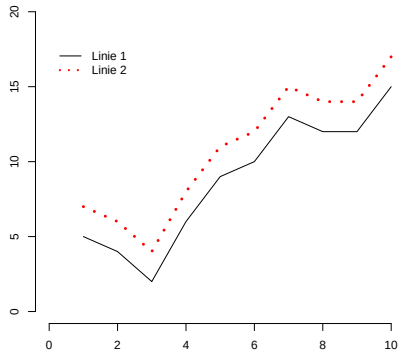
```
> legend(x=0,y=18,legend=c("Linie 1","Linie 2"),
+        lty=c(1,3),lwd=c(1,3),col=c("black","red"),
+        bg="grey",title="Legende")
```

Wie in Abbildung 9.4d zu sehen, wird als Hintergrundfarbe grau gewählt und „Legende“ als Titel eingefügt.

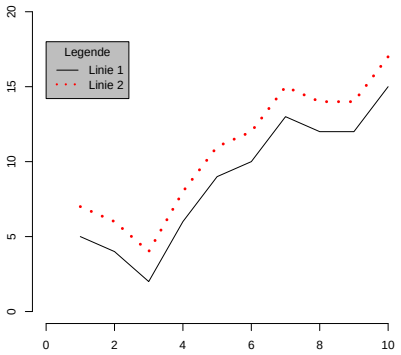


(a)

(b)



(c)



(d)

Abbildung 9.4: Abbildungen zu den Beispielen zu Legenden(a) – (d)

10 Mehrere Plots kombinieren

In R gibt es mehrere Möglichkeiten, wie sich mehrere einzelne Grafiken zusammen in einem Plotfenster anzeigen und bei Bedarf auch speichern lassen. Eine einfache Variante besteht in der Verwendung des Befehls layout(). Diesem wird das gewünschte Layout des Plotfensters übergeben und über den Befehl layout.show() lässt sich das Layout anzeigen.

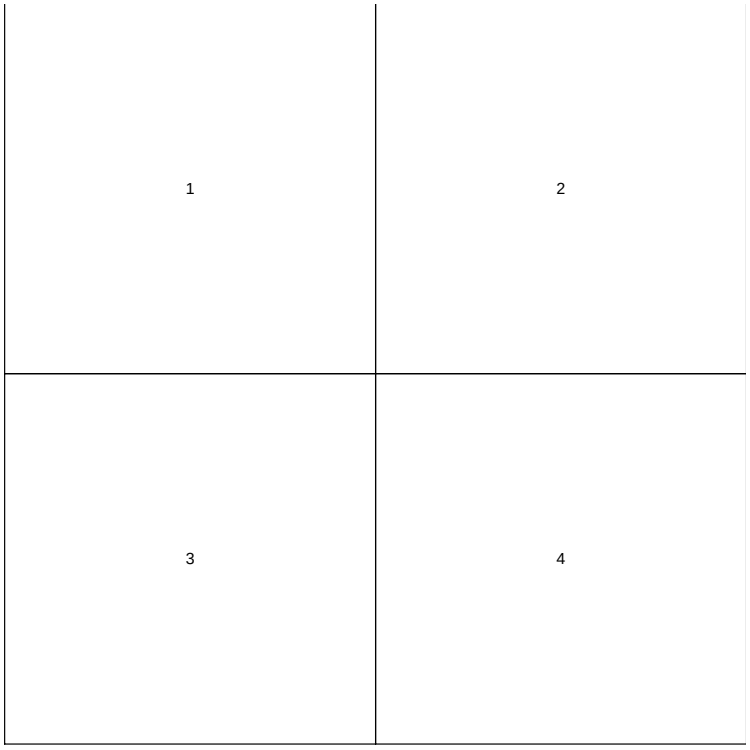
Die Funktionsweise soll hier an einem Beispiel verdeutlicht werden:

```
> l <- rbind(c(1,2),c(3,4))
> l
     [,1] [,2]
[1,]    1    2
[2,]    3    4
> layout(l)
> layout.show(4)
```

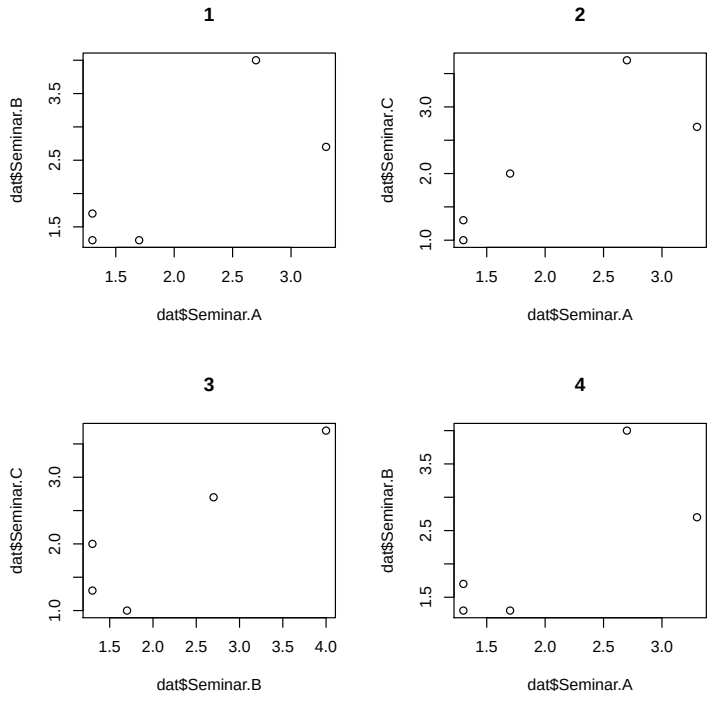
Zunächst wird ein Objekt l erstellt, welches Informationen über das Layout enthält. Wir möchten, dass insgesamt vier Grafiken zusammen in einem Plotfenster erscheinen können. Dabei soll das Plotfenster gewissermaßen in zwei Zeilen unterteilt werden, die je zwei Grafiken enthalten. Für jede Zeile wird ein Vektor verwendet, der Nummern für Grafiken enthält. Der erste Vektor steht für die erste Zeile, welche Grafiken 1 und 2 enthalten soll, der zweite Vektor steht für die zweite Zeile, die die Grafiken 3 und 4 enthalten soll. Diese beiden Vektoren müssen über den rbind() Befehl kombiniert werden. Dieser erstellt aus den beiden Vektoren eine Matrix, wie sie durch Aufruf des Objekts l zu sehen ist. Details zur Erstellung von Matrizen finden sich im entsprechenden Kapitel. Die Matrix verdeutlicht nochmals die Struktur des Layouts: zwei Zeilen mit je zwei Spalten, was Platz für vier durchnummerierte Grafiken bedeutet. Die Matrix mit dem Layout wird nun dem Befehl layout() übergeben, womit es aktiviert wird. Durch den Aufruf von layout.show(4) erhält man Abbildung 10.1a. Das Argument 4 zeigt dabei an, dass das Layout bis Grafik 4 angezeigt werden soll. Gibt man anstelle der 4 beispielsweise eine 3 ein, erscheinen nur die ersten drei Kästchen in der Abbildung.

Nachdem das Layout wie beschrieben umgestellt wurde, können vier Grafikbefehle hintereinander ausgeführt werden, deren Resultate alle in einem Fenster erscheinen. Als Beispiel verwenden wir hier die Seminardaten und erstellen aus diesen vier Streudiagramme:

```
> dat <- read.table("C:/daten/noten.csv",header=T)
> plot(dat$Seminar.A,dat$Seminar.B,main="1")
> plot(dat$Seminar.A,dat$Seminar.C,main="2")
> plot(dat$Seminar.B,dat$Seminar.C,main="3")
> plot(dat$Seminar.A,dat$Seminar.B,main="4")
```



(a)



(b)

Abbildung 10.1: Abbildungen zur Anwendung des layout() Befehls (a) – (b)

Das Ergebnis ist in Abbildung 10.1b zu sehen. Bevor die Grafiken erstellt werden, müssen die Daten gegebenenfalls natürlich noch geladen werden. Die Grafiken werden in der Reihenfolge im Plotfenster sichtbar, wie sie erstellt werden. Dabei kommt die erste Grafik an Position 1, die zweite an Position 2 und so weiter.

Ist das Plotfenster voll und wird ein Grafikbefegl aufgerufen, wird das aktuelle Plotfenster geschlossen und ein neues mit dem selben Layout geöffnet. Das gewählte Layout wird solange verwendet, bis entweder ein neues Layout über den Befehl layout() festgelegt, oder aber mittels layout(1) das Standardlayout mit einer Grafik pro Plotfenster wieder aktiviert wurde.

Durch eine entsprechende Angabe von Grafiknummern lässt sich auch erreichen, dass eine Grafik eine eigene Zeile erhält, während andere Grafiken zusammen in einer Zeile stehen. Beispielsweise möchte man drei Grafiken kombinieren, wobei die beiden ersten Grafiken zusammen in einer Zeile stehen sollen und die dritte in einer eigenen Zeile. Dies lässt sich über folgenden Aufruf erreichen:

```
> l <- rbind(c(1,2),c(3,3))
> layout(l)
> layout.show(3)
```

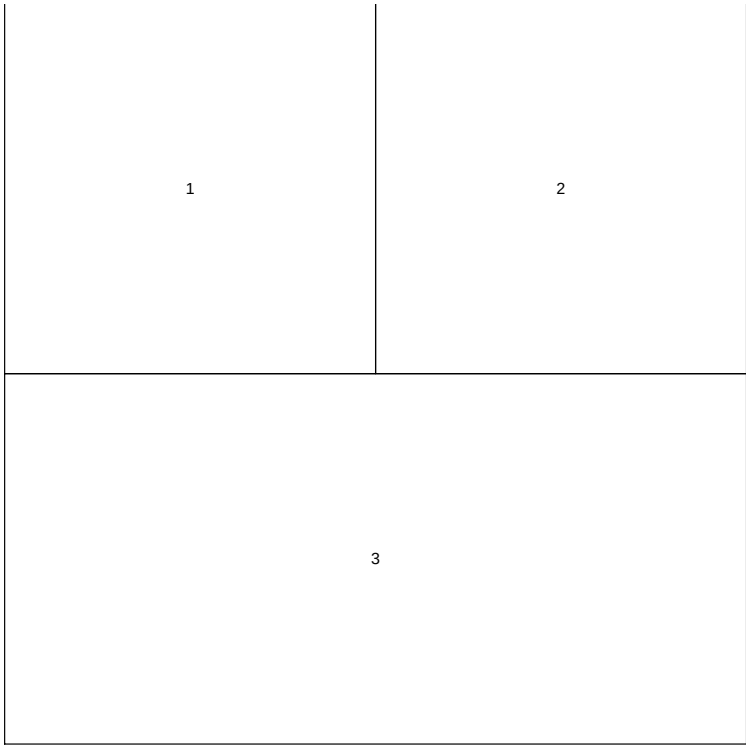
Das Ergebnis des letzten Aufrufs sieht man in Abbildung 10.2a. Nun kann man drei Grafikprozeduren hintereinander ausführen, die zusammen erscheinen. Als Beispiel verwenden wir hier drei einfache Streudiagramme der Klausurdaten, wie in Abbildung 10.2b zu sehen:

```
> plot(dat$Seminar.A,dat$Seminar.B,main="1")
> plot(dat$Seminar.A,dat$Seminar.C,main="2")
> plot(dat$Seminar.B,dat$Seminar.C,main="3")
```

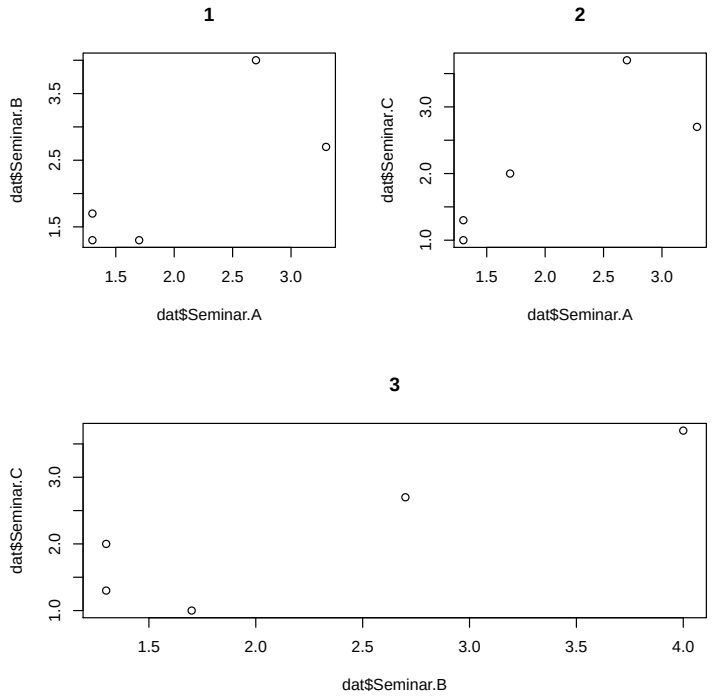
Mit dem layout()-Befehl lassen sich prinzipiell beliebig komplizierte Layouts erstellen:

```
> l <- rbind(c(1,2,3),c(4,4,5),c(6,7,7),c(8,0,9))
> l
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    4    5
[3,]    6    7    7
[4,]    8    0    9
> layout(l,height=c(1,1,1,2),width=c(1,2,1))
> layout.show(9)
```

Der Wert 0 in der letzten Zeile zeigt an, dass an diese Stelle keine Grafik gesetzt werden soll, wie in Abbildung 10.3 zu sehen. Ferner wurden die Argumente width und height genutzt, um die breite der Zeilen und die breite der Spalten festzulegen. Dies geschieht über Vektoren, bei denen jeder Eintrag entweder eine der Zeilen oder eine der Spalten des Layouts repräsentiert. Dabei werden relative Größen angegeben. Beispielsweise bedeutet height=c(1,1,1,2), dass die letzte Zeile des Layouts doppelt so breit sein soll wie die anderen Zeilen. Die Angabe von height=c(2,2,2,4) hätte den gleichen Effekt. width=c(1,2,1) wird im Prinzip gleich gelesen und bedeutet, dass die zweite Spalte doppelt so breit sein soll wie die erste und die dritte Spalte.



(a)



(b)

Abbildung 10.2: Abbildungen zur Anwendung des layout() Befehls (a) – (b)

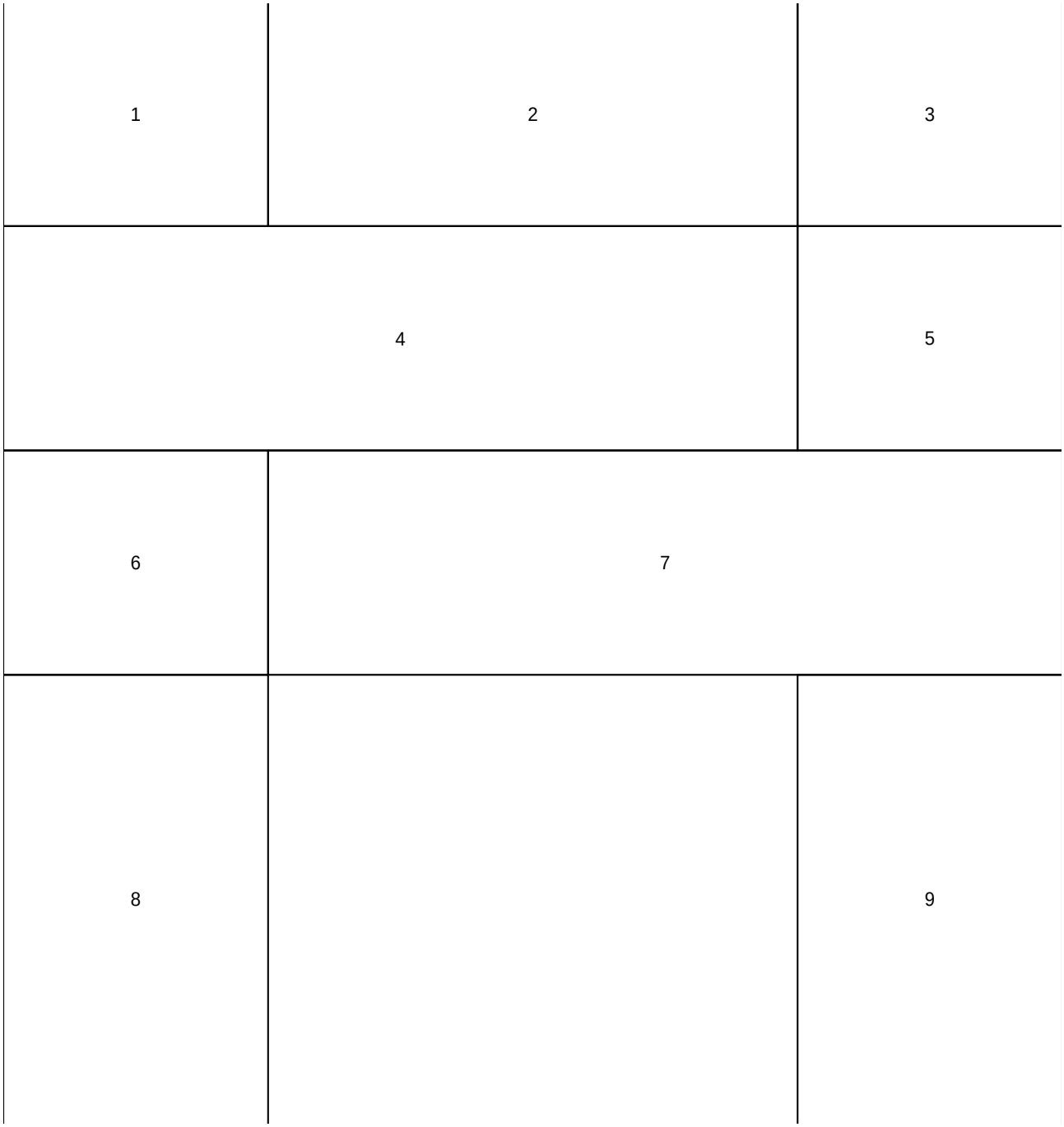


Abbildung 10.3: Abbildung zur Anwendung des layout() Befehls

11 Lattice Plots

Hier wird etwas zu Lattice Plots stehen.

12 Karten erstellen

Hier wird beschrieben, wie man Karten mit R erstellt.

13 Netzwerke & Graphen

Hier wird beschrieben, wie man Graphen erstellt.

Teil III

Mathematik & Programmieren

14 Lineare Algebra

14.1 Matrizen erstellen

Zur Erzeugung von Matrizen wird der Befehl matrix() verwendet. Diesem Befehl wird als erstes Argument ein Vektor übergeben, der die Elemente der Matrix enthält. Anschließend lässt sich die Zahl der Zeilen und die Zahl der Spalten angeben. Die Elemente des Vektors werden dann spaltenweise in die Matrix „gefüllt“. Zur Eingabe der Matrix

$$\mathbf{X} = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

würde man beispielsweise

```
> X <- matrix(c(1,2,3,4),nrow=2,ncol=2)
```

verwenden:

```
> X
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Die Argumente nrow und ncol geben die Zahl der Zeilen bzw. die Zahl der Spalten wieder. In diesem Beispiel soll es eine $(2, 2)$-Matrix sein. Die Elemente des angegebenen Vektors c(1,2,3,4) werden dann wie erwähnt spaltenweise in die Matrix geschrieben: Das erste Element des Vektors wird in die erste Spalte und erste Zeile geschrieben. Das zweite Element des Vektors wird in die erste Spalte und zweite Zeile geschrieben. Nun ist die erste Spalte voll und R springt zur nächsten Spalte und füllt diese analog auf.

Das resultierende Objekt X besitzt folgende Eigenschaften:

```
> class(X)
[1] "matrix"
> mode(X)
[1] "numeric"
> attributes(X)
$dim
[1] 2 2
```

Der Output lässt sich wie folgt interpretieren: X ist eine Matrix, deren Elemente numerisch – also Zahlen – sind. Sie besitzt das Attribut dim, welches die Zahl der Zeilen und die Zahl der Spalten enthält. Letztere lassen sich auch durch Benutzung des Befehls dim() anzeigen:

```
> dim(X)
[1] 2 2
```

Ein zweites Beispiel

```
> Y <- matrix(1:6,nrow=3)
```

liefert die Matrix

$$\mathbf{Y} = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

Hier wurde nur die Zahl der Zeilen n angegeben. Die Zahl der Spalten ergibt sich dann aus der Länge des Datenvektors l – in diesem Fall 6 – dividiert durch n:

```
> attributes(Y)
$dim
[1] 3 2
> dim(Y)
[1] 3 2
```

Man hätte auch nur die Zahl der Spalten m angeben können und die Zahl der Zeilen wäre als l/m berechnet worden:

```
> Z <- matrix(7:12,ncol=2)
> Z
     [,1] [,2]
[1,]    7   10
[2,]    8   11
[3,]    9   12
```

Möchte man, dass die angegebenen Werte nicht spalten- sondern zeilenweise in die Matrix gesetzt werden, muss die Option byrow=TRUE angegeben werden:

```
> Z <- matrix(7:12,ncol=2,byrow=TRUE)
> Z
     [,1] [,2]
[1,]    7    8
[2,]    9   10
[3,]   11   12
```

Nun wird zuerst die erste Zeile aufgefüllt, anschließend die zweite Zeile und so fort. Auch hier reicht es, entweder die Zahl der Zeilen oder die Zahl der Spalten anzugeben.

Eine Matrix, deren Einträge alle einem beliebigen Wert x entsprechen, kann man erzeugen, indem nur dieser Wert x in Kombination mit der Zahl der Zeilen und der Zahl der Spalten angegeben wird:

```
> B <- matrix(1,ncol=3,nrow=3)
> B
     [,1] [,2] [,3]
```

```
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1
```

Eine weitere Möglichkeit zur Erzeugung von Matrizen besteht darin, Vektoren zusammenzufügen. Über den Befehl cbind(x,y) werden zwei Vektoren x und y als Spalten einer Matrix verwendet, durch rbind(x,y) fungieren sie als Zeilen:

```
> x <- c(1,2,3)
> y <- c(5,2,4)
> cbind(x,y)
     x y
[1,] 1 5
[2,] 2 2
[3,] 3 4
> rbind(x,y)
  [,1] [,2] [,3]
x    1    2    3
y    5    2    4
```

Dabei verwendet R die Namen der ursprünglichen Vektoren entweder als Namen der Spalten oder als Namen der Zeilen.

14.2 Rechnen mit Matrizen

Addition, Subtraktion, Multiplikation und Division mit Skalaren erfolgt analog zum Vorgehen bei Vektoren:

```
> X
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> X+1 # Addition
     [,1] [,2]
[1,]    2    4
[2,]    3    5
> X*2 # Multiplikation
     [,1] [,2]
[1,]    2    6
[2,]    4    8
> X-1 # Subtraktion
     [,1] [,2]
[1,]    0    2
[2,]    1    3
> 1-X
     [,1] [,2]
[1,]    0   -2
[2,]   -1   -3
```

```
> X/2  # Division
     [,1] [,2]
[1,]  0.5  1.5
[2,]  1.0  2.0
> 2/X
     [,1]      [,2]
[1,]    2 0.6666667
[2,]    1 0.5000000
```

Die Berechnung erfolgt auch hier wieder elementweise.

Auf gleiche Art und Weise lassen sich elementweise Operationen mit einer Matrix und einem Vektor sowie zwei Matrizen durchführen. Werden eine Matrix und ein Vektor verwendet, erfolgt die Berechnung spaltenweise, wobei die Elemente des Vektors gegebenenfalls wiederholt werden, um die nötige Länge zu erreichen:

```
> Y
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> Y+x
     [,1] [,2]
[1,]    2    5
[2,]    4    7
[3,]    6    9
> Y-x
     [,1] [,2]
[1,]    0    3
[2,]    0    3
[3,]    0    3
> Y*x
     [,1] [,2]
[1,]    1    4
[2,]    4   10
[3,]    9   18
> Y/x
     [,1] [,2]
[1,]    1  4.0
[2,]    1  2.5
[3,]    1  2.0
```

Werden zwei Matrizen verwendet, müssen beide die gleiche Größe haben:

```
> Z
     [,1] [,2]
[1,]    7   10
[2,]    8   11
[3,]    9   12
> Y+Z
     [,1] [,2]
```

```
[1,]    8   14
[2,]   10   16
[3,]   12   18
> Y-Z
     [,1] [,2]
[1,]   -6   -6
[2,]   -6   -6
[3,]   -6   -6
> Y*Z
     [,1] [,2]
[1,]    7   40
[2,]   16   55
[3,]   27   72
> Y/Z
          [,1]      [,2]
[1,] 0.1428571 0.4000000
[2,] 0.2500000 0.4545455
[3,] 0.3333333 0.5000000
> Y-X
Fehler in Y - X : nicht passende Arrays
```

Zur Matrixmultiplikation wird wieder %*% verwendet. Hier muss abermals auf die Größe von Matrizen und auch von Vektoren geachtet werden:

```
> Y%*%X
     [,1] [,2]
[1,]    9   19
[2,]   12   26
[3,]   15   33
> X%*%Y
Fehler in X %*% Y : nicht passende Argumente
```

Die Matrizen $\mathbf{X}$ und $\mathbf{Y}$ können nicht multipliziert werden, da $\mathbf{X}$ eine $(2, 2)$-Matrix ist, $\mathbf{Y}$ aber eine $(3, 2)$-Matrix. $\mathbf{XY}'$ hingegen könnte man berechnen. Um eine Matrix (oder einen Vektor) zu transponieren, wird die Funktion t() verwendet:

```
> t(Y)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> X%*%t(Y)
     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   18   24   30
```

Das Skalarprodukt zweier Vektoren $\mathbf{a}$ und $\mathbf{b}$ lässt sich ebenfalls über %*% oder den Befehl crossprod(a,b) berechnen:

```
> x <- c(1,2,3)
> y <- c(5,2,4)
> x %*% y
```

```
     [,1]
[1,]   21
> crossprod(x,y)
     [,1]
[1,]   21
```

Das Ergebnis entspricht also $\mathbf{ab}'$, wobei $\mathbf{a}$ ein Zeilen- und $\mathbf{b}'$ ein Spaltenvektor (transponierter Zeilenvektor) ist. Zur Berechnung von $\mathbf{a}'\mathbf{b}$ gibt es mehrere Alternativen, eine Möglichkeit wäre:

```
> x %*% t(y)
     [,1] [,2] [,3]
[1,]    5    2    4
[2,]   10    4    8
[3,]   15    6   12
```

Die Transponierung wird aufgrund der internen Behandlung von Vektoren in R notwendig, bei der Vektoren zunächst als Spalten- und nicht als Zeilenvektoren aufgefasst werden. Werden zwei Vektoren über %*% multipliziert, geht R zunächst davon aus, dass das Skalarprodukt gewünscht ist. Zur Berechnung des Tensorproduktes muss hingegen explizit angegeben werden, dass es sich beim zweiten Vektor um einen Zeilenvektor handelt.

Auch die Aufrufe

```
cbind(x)%*%y
x%*%rbind(y)
crossprod(t(x),t(y))
```

würden hier zum gleichen Resultat führen. In allen Fällen ist das Ergebnis eine Matrix, wobei R neben den einzelnen Elementen auch Zeilen- und Spaltennummern ausgibt. Bereits das Ergebnis der Skalarmultiplikation x%*%y wird als Matrix behandelt:

```
> class(x%*%y)
[1] "matrix"
```

Zur Berechnung von Determinanten wird der Befehl det() verwendet, zur Berechnung von Inversen der Befehl solve(). $\det(\mathbf{X})$ und $\mathbf{X}^{-1}$ erhält man dann über:

```
> det(X)
[1] -2
> solve(X)
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

Sollen alle Elemente einer Matrix aufsummiert werden, kann wie bei Vektoren der Befehl sum() benutzt werden:

```
> sum(X)
[1] 10
```

Möchte man hingegen die Summen der Elemente der Spalten beziehungsweise Zeilen berechnen, werden die Befehle colSums() und rowSums() benutzt:

```
> colSums(X)
[1] 3 7
> rowSums(X)
[1] 4 6
```

Im ersten Fall werden die Summen der Elemente der beiden Spalten der Matrix X ausgegeben, im zweiten Fall werden diese Summen über die Zeilen gebildet.

14.3 Indizierung von Matrizen

Zur Indizierung von Matrizen werden eckige Klammern ganz analog zur Indizierung von Datensätzen verwendet. Um beispielsweise das Element $x_{1,2}$ der Matrix $\mathbf{X}$ zu erhalten, ist folgende Eingabe nötig:

```
> X[1,2]
[1] 3
```

Um also allgemein das Element $x_{i,j}$ aus Zeile i und Spalte j zu erhalten, muss man $\mathrm{X[i,j]}$ eingeben. Möchte man eine ganze Spalte bzw. eine ganze Spalte extrahieren, wird nur die Spalten- bzw. Zeilennummer angegeben, wobei auch hier darauf geachtet werden muss, dass weiterhin das Komma in der eckigen Klammer steht:

```
> X[,2]
[1] 3 4
> X[2,]
[1] 2 4
```

Über den Befehl $\mathrm{diag()}$ lässt sich die Hauptdiagonale einer Matrix auslesen:

```
> diag(X)
[1] 1 4
```

Dieser Befehl lässt sich zudem auch zum Erstellen von Diagonalmatrizen verwenden. Zum Beispiel lässt sich eine Einheitsmatrix der Größe $(5, 5)$ erstellen über:

```
> I <- diag(1,5)
> I
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1
```

14.4 Logische Vergleiche mit Matrizen

Auch mit Matrizen können logische Vergleiche durchgeführt werden. Wird eine Matrix mit einem Skalar verglichen, wird für jedes Element der Matrix überprüft, ob es diesem Skalar entspricht:

```
> X <- matrix(c(1,2,3,4),ncol=2)
> X==1
      [,1]  [,2]
[1,]  TRUE FALSE
[2,] FALSE FALSE
```

Wird eine Matrix mit einem Vektor verglichen, werden die einzelnen Einträge der Matrix spaltenweise mit den Einträgen des Vektors verglichen. Der Eintrag in der ersten Zeile und ersten Spalte der Matrix wird mit dem ersten Eintrag des Vektors verglichen, der Eintrag in der ersten Spalte und zweiten Zeile der Matrix wird mit dem zweiten Eintrag verglichen und so fort. Entspricht die Zahl der Einträge des Vektors nicht der Zahl der Einträge der Matrix, werden die Einträge des Vektors gegebenenfalls wiederholt. Besitzt der Vektor mehr Einträge als die Matrix, erscheint eine Fehlermeldung:

```
> x <- c(1,2)
> X==x
      [,1]  [,2]
[1,]  TRUE FALSE
[2,]  TRUE FALSE
> z <- 1:7
> X==z
Fehler: Dimensionen [Produkt 4] passen nicht zur Länge des Objektes [7]
```

Beim Vergleich zweier Matrizen ist darauf zu achten, dass diese die gleiche Größe haben, ansonsten ist kein Vergleich möglich. Ansonsten werden die einzelnen Einträge der Matrizen direkt miteinander verglichen:

```
> Y <- matrix(c(1,4,3,2),ncol=2)
> Y==X
      [,1]  [,2]
[1,]  TRUE  TRUE
[2,] FALSE FALSE
```

Auch bei Vergleichen mit Matrizen können die Befehle all() und any() wie bei Vektoren angewendet werden.

15 Programmieren mit R

15.1 Schleifen und Wiederholungen

15.1.1 Schleifen mit for

Möchte man Berechnungen wiederholt ausführen – ggf. mit unterschiedlichen Eingabewerten – bieten sich sogenannte Schleifen an. Besonders wichtig sind hierbei die Befehle for und while, die oft auch in Kombination mit dem Befehl if verwendet werden.

Der Befehl for soll an einem kleinen Beispiel eingeführt werden:

```
> for(i in 1:5) print(i)
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Für jedes i aus den Zahlen 1 bis 5 wird der Reihe nach der Befehl print() ausgeführt, der einfach das gerade betrachtete i ausgibt. Zunächst wird in den Klammern hinter dem Befehl angegeben, was für ein Index verwendet werden soll und über welche Zahlen dieser läuft – in diesem Beispiel eben i und die Zahlen 1 bis 5. Anschließend wird die Aktion angegeben, die jeweils durchgeführt werden soll. Dies können im Prinzip beliebige Befehle sein.

Es können auch mehrere Befehle ausgeführt werden. Hierfür sollte auf den for()-Befehl eine geschweifte Klammer folgen, die einzelnen auszuführenden Befehle werden danach jeweils in eine Zeile geschrieben und es folgt eine abschließende Zeile mit einer schließenden geschweiften Klammer:

```
> for(j in 1:3) {
+ a <- j*2
+ print(a)
+ }
[1] 2
[1] 4
[1] 6
```

Hier wird der Index als j bezeichnet und läuft über die Werte 1 bis 3. Diese Werte werden jeweils mit 2 multipliziert und in einem Objekt mit dem Namen a abgelegt, welches anschließend über den print()-Befehl angezeigt wird. Der zuletzt berechnete Wert des Objektes a ist zudem immer noch im Speicher:

```
> a
[1] 6
```

Hier benötigen wir keinen print()-Befehl, um das Objekt a anzuzeigen, in einer Schleife ist dies aber notwendig.

15.1.2 Logische Abfragen mit if

Der Befehl if() funktioniert von der Struktur her ähnlich wie der for()-Befehl. Allerdings wir hierbei in Klammern eine logische Abfrage angegeben. Liefert diese den Wert TRUE, werden angegebene Befehle ausgeführt:

```
> a <- 2
> if(a==2) print(a)
[1] 2
> a <- 3
> if(a==2) print(a)
>
```

Dies lässt sich auch mit einer for()-Schleife kombinieren:

```
> for(i in 1:4) {
+ if(i<3) print(i)
+ }
[1] 1
[1] 2
```

Der aktuelle Wert von i wird hier nur ausgegeben, wenn er niedriger als 3 ist.

Schließlich können auch mehrere Befehle in Kombination mit dem if-Befehl angegeben werden, wobei hier idealerweise wieder mehrere Zeilen und geschweifte Klammern verwendet werden:

```
> for(i in 1:4) {
+   if(i>2) {
+   a <- 2*i
+   print(a)
+   }
+ }
[1] 6
[1] 8
```

Wir verwenden hier also zwei Klammersätze – einmal für die for()-Schleife und einmal für den if()-Befehl, wobei der letztere innerhalb der Schleife „geschachtelt“ ist. Damit bei solch längeren Aufrufen die Übersichtlichkeit nicht verloren geht, werden untergeordnete Ebenen – in diesem Fall die if()-Abfrage – über einen Tabulator (*tab stop*) eingerückt. Prinzipiell lassen sich beliebig viele Ebenen von for und if ineinander verschachteln:

```
> for(i in 1:2) {
+   for (j in 3:4) {
```

```
+      for (k in 5:6) {
+        if(i+j+k==10) {
+          print(i)
+        }
+      }
+    }
+ }
[1] 1
[1] 1
[1] 2
```

In Kombination mit if lässt sich über den Befehl else angeben, ob und welche Befehle ausgeführt werden sollen, wenn die logische Abfrage als Ergebnis FALSE liefert. Hierbei muss der Befehl else so angegeben werden, dass erkennbar ist, zu welcher if-Abfrage er gehört. Werden keine geschweiften Klammern verwendet, muss der Befehl in der selben Zeile wie der if-Befehl stehen, ansonsten in der Zeile der schließenden geschweiften Klammer:

```
> a <- 2
> if(a==2) print(a) else print("x")
[1] 2
> a <- 3
> if(a==2) print(a) else print("x")
[1] "x"
```

oder

```
> for(i in 1:4) {
+    if(i>2) {
+    a <- 2*i
+    print(a)
+    } else {
+    print("x")
+    }
+ }
[1] "x"
[1] "x"
[1] 6
[1] 8
```

15.1.3 Wiederholungen mit while

Der Befehl while() schließlich funktioniert ebenfalls analog zu for(). Über diesen können ein Befehl oder mehrere Befehle wiederholt ausgeführt werden, solange eine bestimmte Bedingung zutrifft:

```
> a <- 0
> while(a<10) a <- a+1
> a
[1] 10
```

Durch die Verwendung mehrerer Zeilen und geschweifter Klammern lassen sich mehrere Befehle ausführen. Der while-Befehl kann beliebig mit den Befehlen for und if kombiniert werden.

Einerseits sind hierdurch zwar sehr komplexe Konstruktionen möglich, andererseits ist zu beachten, dass insbesondere Schleifen, die über eine große Anzahl an Werten laufen, beziehungsweise viele ineinander verschachtelte Schleifen sehr, rechenaufwendig sind. Solange es einen R Befehl gibt, der dasselbe vermag, wie die Schleifenkonstruktion, ist dieser immer vorzuziehen. Dies soll an einem einfachen Beispiel illustriert werden. Angenommen man möchte alle Zahlen von 1 bis 50000 quadrieren und anschließend aufsummieren. Dann kann man entweder eine Schleife verwenden

```
> a <- 0
> for(i in 1:50000) a <- a+i^2
> a
[1] 4.166792e+13
```

oder aber den Befehl sum()

```
sum((1:50000)^2)
```

Zunächst ist leicht ersichtlich, dass die letztere Variante einfacher einzugeben ist. Um Unterschiede in der Geschwindigkeit der Berechnung festzustellen, kann der Befehl system.time() verwendet werden:

```
> system.time(for(i in 1:50000) a <- a+i^2)
   user  system elapsed
   0.07    0.00    0.16
> system.time(sum((1:50000)^2))
   user  system elapsed
      0       0       0
```

Interessant ist hier für uns nur die Angabe unter elapsed, die die insgesamt benötigte Rechenzeit in Sekunden wiedergibt. Für den sum-Befehl ist diese so gering, dass 0 ausgegeben wird. Verglichen damit ist die Schleife mit 0.16 Sekunden relativ langsam. Zwar ist dies immer noch nicht viel, aber bei wesentlich aufwendigeren Berechnungen wird der Unterschied schnell erheblich.

15.2 Zufallszahlen

Zuweilen kann die zufällige Generierung von Werten, welche dann als Zufallszahlen bezeichnet werden, nötig sein. Ein Beispiel ist die Erstellung von Simulationen. Hier muss man sich zunächst klarmachen, dass solche Zahlen mit einem PC eigentlich nicht erzeugt werden können: Es muss immer eine klare Regel geben, mit der Zahlen generiert werden, weswegen diese eigentlich nicht wirklich „zufällig“ im engeren Sinne sein können.[1] Hingegen lassen sich aller-

[1] Es besteht jedoch die Möglichkeit, „echte“ Zufallszahlen mittels materieller Zufallsgeneratoren zu erzeugen, wie beispielsweise durch das Werfen eines Würfels oder durch Messung von physikalischen Größen mit zufälligem Charakter, wie etwa dem elektromagnetischen Rauschen in der Atmosphäre. Die Internetseite www.random.org stellt solche Daten zur Verfügung, auf die auch über das R-Paket random zugegriffen werden kann.

dings sogenannte Pseudo-Zufallszahlen erzeugen, die dann wie Zufallszahlen behandelt werden. Hierfür bietet R etliche Möglichkeiten.

Zunächst muss man sich überlegen, welche Zahlen überhaupt vorkommen sollen und mit welcher Wahrscheinlichkeit. Zahlen aus einem Intervall $[a, b]$, die alle die gleiche Ziehungswahrscheinlichkeit aufweisen, lassen sich mit dem Befehl runif(n,min=a,max=b) erzeugen, wobei n für die Anzahl der zu generierenden Werte steht und als Standardeinstellung a=0 und b=1 benutzt wird:

```
> runif(1)
[1] 0.4299770
> runif(1)
[1] 0.2943444
> runif(1)
[1] 0.3639257
> runif(5)
[1] 0.01569455 0.18813462 0.02528612 0.81635473 0.98686035
> runif(5,min=1,max=2)
[1] 1.663770 1.625648 1.457643 1.135046 1.626135
```

Durch die Verwendung von Zuweisungen lassen sich die resultierenden Vektoren im Speicher ablegen und gegebenenfalls weiter bearbeiten:

```
> x <- runif(3)
> x
[1] 0.8294750 0.6747684 0.4168807
> y <- 1-x^2
> y
[1] 0.3119712 0.5446876 0.8262105
```

Der Name des Befehls runif leitet sich aus r für *random deviates* (Zufallswerte) und unif für *uniform distribution* (Gleichverteilung) her.

Neben gleichverteilten Werten über ein Intervall $[a, b]$ kann R Zufallswerte aus vielen anderen Verteilungen erzeugen. Beispielsweise lassen sich Zufallszahlen, die einer Normalverteilung folgen, mit dem Befehl rnorm(n,mean=0,sd=1) erzeugen, wobei n wieder die Zahl der zu generierenden Werte angibt, mean den Mittelwert und sd die Standardabweichung der Normalverteilung spezifiziert. Als Standardeinstellung werden Werte aus der sogenannten Standard-Normalverteilung gezogen:

```
> rnorm(1)
[1] 1.079016
> rnorm(1)
[1] 0.5739131
> rnorm(1,mean=5,sd=2)
[1] 9.32352
> rnorm(3,mean=5,sd=2)
[1] 9.349027 3.833291 3.759596
```

Bei den Optionen mean und sd lassen sich nicht nur einzelne Werte angeben, sondern auch Vektoren:

```
> rnorm(6,mean=c(0,10),sd=c(1,2))
[1] -1.2213204 11.2962079  0.1581048  8.6038313 -1.1265376 11.4683644
```

In diesem Beispiel sind zwei Normalverteilungen spezifiziert – die Standardnormalverteilung mit Mittelwert 0 und Standardabweichung 1 und eine Normalverteilung mit Mittelwert 10 und Standardabweichung 2. Der erste Zufallswert wird aus der Standardnormalverteilung gezogen, der zweite Wert aus der zweiten Normalverteilung, der dritte Wert wieder aus der Standardnormalverteilung und so fort.

Zufallszahlen aus weiteren Verteilungen lassen sich mit den folgenden Befehlen generieren, wobei der Name der Verteilung immer als Kommentar angegeben ist und immer ein Zufallswert als Beispiel angezeigt wird:

```
> runif(1,min=0,max=1) # Gleichverteilung
[1] 0.06070439
> rnorm(1,mean=0,sd=1) # Normalverteilung
[1] -1.265533
> rf(1,df1=1,df2=3) # F-Verteilung
[1] 4.905246
> rgamma(1,shape=1,rate=1) # Gamma-Verteilung
[1] 0.7026316
> rgeom(1,prob=0.5) # Geometrische Verteilung
[1] 1
> rhyper(1,m=3,n=3,k=1) # Hypergeometrische Verteilung
[1] 0
> rlnorm(1,meanlog=0,sdlog=1) # Log-Normalverteilung
[1] 0.5061668
> rlogis(1,location=0,scale=1) # Logistische Verteilung
[1] -1.711113
> rmultinom(1,size=2,prob=c(0.5,0.5)) # Multinomialverteilung
     [,1]
[1,]    2
[2,]    0
> rnbinom(1,size=2,mu=1) # Negativ-Binomialverteilung
[1] 0
> rpois(1,lambda=1) # Poisson-Verteilung
[1] 1
> rbeta(1,shape1=1,shape2=2,ncp=0) # Beta-Verteilung
[1] 0.03780954
> rbinom(1,size=3,prob=0.5) # Binomialverteilung
[1] 1
> rcauchy(1,location=0,scale=1) # Cauchy-Verteilung
[1] 5.23698
> rchisq(1,df=2,ncp=0) # Chi-Quadrat-Verteilung
[1] 0.03397483
> rexp(1,rate=1) # Exponentialverteilung
[1] 0.09731328
> rt(1,df=2,) # t-Verteilung
[1] 0.2820218
```

```
> rweibull(1,shape=1,scale=1) # Weibull-Verteilung
[1] 3.187787
```

Ein weiterer nützlicher Befehl zur Erzeugung von zufälligen Werten ist sample(). Die Syntax sieht allgemein wie folgt aus: sample(x, size, replace = FALSE, prob = NULL). Auf die Optionen replace und prob gehen wir weiter unten ein. Das erste Argument x muss ein Vektor sein, der die Zahlen enthält, aus denen gezogen werden soll. size gibt die Zahl der Ziehungen wieder. Werden nur diese beiden Optionen angegeben, ist die Ziehungswahrscheinlichkeit aller Werte aus x gleich. Um einen einfachen Würfelwurf zu simulieren, geben wir für x einen Vektor an, der die Zahlen 1 bis 6 enthält und setzen size=1:

```
> sample(1:6,size=1)
[1] 6
> sample(1:6,size=1)
[1] 3
> sample(1:6,size=1)
[1] 1
> sample(1:6,size=1)
[1] 1
> sample(1:6,size=1)
[1] 4
```

Für ein mehrfaches Werfen des Würfels setzen wir size entsprechend hoch und erhalten als Ergebnis Vektoren:

```
> sample(1:6,2)
[1] 6 2
> sample(1:6,4)
[1] 2 4 6 3
> sample(1:6,6)
[1] 2 5 4 1 6 3
```

In diesen einfachen Beispielen reicht es, für size einfach einen Wert einzugeben, wie beispielsweise 1 oder 5 – auf size= kann hier also verzichtet werden. Der Klarheit halber werden wir dies aber im weiteren immer angeben.

Ein Problem bei unserer Simulation eines einfachen Würfelwurfs wird aber deutlich, wenn man size=7 wählt:

```
> sample(1:6,size=7)
Fehler in sample(length(x), size, replace, prob) :
        kann keine Stichprobe größer als die Grundgesamtheit nehmen
 wenn 'replace = FALSE'
```

Standardmäßig wird aus dem angegebenen Vektor ohne zurücklegen gezogen. In unserem Beispiel enthält der Vektor zur ersten Ziehung 6 Werte. Beispielsweise wird dann die Zahl 2 gezogen. Für die zweite Ziehung werden dann nur noch die Werte 1, 3, 4, 5 und 6 verwendet. Nach insgesamt sechs Ziehungen sind dann alle Werte aus dem angegebenen Vektor aufgebraucht, so dass keine siebte Ziehung möglich ist. Wird die Option replace=TRUE gewählt, werden die ge-

zogenen Werte wieder zurückgelegt, so dass bei jeder Ziehung jeder angegebene Wert möglich ist:

```
> sample(1:6,size=7,replace=T)
[1] 6 5 6 5 6 2 1
```

Über die Option prob können bestimmte Ziehungswahrscheinlichkeiten für die einzelnen Werte aus x spezifiziert werden. Hierbei wird ein Vektor angegeben, der genau so viele Elemente enthält wie x. Der erste Wert dieses Vektors ist dann die Ziehungswahrscheinlichkeit für den ersten Wert aus x, der zweite Wert bezieht sich auf den zweiten Wert aus x und so fort. Ein einfaches Beispiel, bei dem x nur die Werte 0 und 1 umfasst, die mit einer Wahrscheinlichkeit von 0.1 respektive 0.9 gezogen werden können:

```
> sample(c(0,1),size=20,replace=T,prob=c(0.1,0.9))
 [1] 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
```

15.3 Funktionen, Listen, Klassen, Methoden

15.3.1 Erstellen von eigenen Funktionen

Objekte in R wurden in Kapitel 2.5 vorgestellt. In diesem Rahmen wurden auch Funktionen angesprochen. Diesen werden Objekte und Optionen übergeben, worauf diese bestimmte Operationen durchführen. In diesem Abschnitt schauen wir uns an, wie man Funktionen selber definieren kann. Möchte man aufwendigere Berechnungen mehrfach mit unterschiedlichen Objekten oder Parametern durchführen, ist es ausreichend, eine entsprechende Funktion einmal zu spezifizieren und diese dann mehrfach auszuführen, wodurch eine erhebliche Reduktion der nötigen Syntax möglich ist.

Zur Erläuterung schauen wir uns ein simples Beispiel an:

```
> beispiel <- function(x) {
+   x+2
+   }
> beispiel(1)
[1] 3
```

In den Zeilen 1 bis 3 wird die Funktion definiert. In Zeile 4 wird diese aufgerufen und in Zeile 5 steht das Ergebnis dieses Aufrufs. Über die Zuweisung in der ersten Zeile wird der Name der Funktion festgelegt – hier einfach beispiel. Über den Befehl function auf der rechten Seite der Zuweisung wird festgelegt, dass das Objekt beispiel eine Funktion sein soll. In den runden Klammern hinter dem Befehl function werden die Argumente der Funktion aufgelistet und benannt. In unserem Beispiel gibt es nur ein Argument, welches den Namen x hat. Zwischen den geschweiften Klammern in der ersten und der dritten Zeile stehen die Operationen, die die Funktion durchführt. In diesem Beispiel wird dem Argument x der Wert 2 hinzu addiert. Damit ist die Funktion beispiel fertig definiert und kann unter Angabe eines Argumentes für x wie in Zeile 4

aufgerufen werden. Wir können uns auch vergewissern, dass es sich beim Objekt beispiel um eine Funktion handelt:

```
> class(beispiel)
[1] "function"
```

Möchte man für das Argument x einen Standardwert definieren, gibt man diesen mittels eines Gleichheitszeichens an:

```
> beispiel <- function(x=1) {
+   x+2
+   }
> beispiel()
[1] 3
> beispiel(3)
[1] 5
```

Durch die Angabe dieses Standardwertes kann man die Funktion aufrufen, ohne einen Wert für x zu spezifizieren – wenn dieser fehlt, wird einfach der Wert 1 angenommen. Nichtsdestotrotz kann für x immer noch ein anderer Wert angegeben werden.

Nun ein Beispiel mit zwei Argumenten:

```
> beispiel <- function(x=1,y) {
+   x+y
+   }
> beispiel(y=1)
[1] 2
> beispiel(x=2,y=5)
[1] 7
```

Die Funktion nimmt nun die Argumente x und y an. Im vorherigen Beispiel mit nur einem Argument haben wir lediglich den Wert für x eingegeben, nicht beispielsweise beispiel(x=1). Wenn mehrere Argumente verwendet werden ist dies prinzipiell zwar noch immer möglich, sollte aber um Fehler zu vermeiden nicht genutzt werden, so dass hier bei den Werten immer angegeben wird, auf welches Argument man sich bezieht. Da für x ein Standardwert gewählt wurde, muss dieses Argument bei einem Funktionsaufruf nicht unbedingt angegeben werden. y hingegen muss allerdings immer angegeben werden, da ansonsten eine Fehlermeldung erscheint:

```
> beispiel(x=1)
Fehler in beispiel(x = 1) : Argument "y" fehlt (ohne Standardwert)
> beispiel()
Fehler in beispiel() : Argument "y" fehlt (ohne Standardwert)
```

Mit der gerade definierten Funktion können auch Zuweisungen vorgenommen werden:

```
> a <- beispiel(y=1)
> a
[1] 2
```

Angenommen, eine Funktion errechnet mehrere Ergebnisse bzw. Zwischenergebnisse. Als Beispiel soll folgende Funktion dienen:

```
> beispiel <- function(x) {
+   3*x+2
+   x*x
+ }
```

Rufen wir diese Funktion auf, wird lediglich das Ergebnis von Zeile 3 angezeigt bzw. für Zuweisungen verwendet:

```
> beispiel(1)
[1] 1
> beispiel(3)
[1] 9
> a <- beispiel(5)
> a
[1] 25
```

Es wird also immer das Ergebnis der letzten Berechnung ausgegeben. Um dieses Problem zu lösen, nehmen wir in einem ersten Schritt innerhalb der Funktion Zuweisungen vor:

```
> beispiel <- function(x) {
+   y <- 3*x+2
+   z <- x*x
+ }
```

Hierbei ist wichtig, dass die Objekte y und z nur temporär erzeugt werden, also nach Aufruf der Funktion *nicht* im Workspace gespeichert sind. Wenn wir die Funktion aufrufen, wird noch immer nur das letzte Ergebnis angezeigt und der Versuch, y und z anzusprechen, führt zu einer Fehlermeldung:

```
> a <- beispiel(5)
> a
[1] 25
> y
Fehler: objekt "y" nicht gefunden
> z
Fehler: objekt "z" nicht gefunden
```

Um anzugeben, welches der temporären Objekte ausgegeben wird, kann der Befehl return() verwendet werden:

```
> beispiel <- function(x) {
+   y <- 3*x+2
+   z <- x*x
+   return(y)
+ }
> beispiel(1)
[1] 5
```

Es wurden also zunächst zwei temporäre Objekte erzeugt und anschließend angegeben, welches die Funktion ausgeben soll. Allerdings erhalten wir immer noch nur eines der Objekte.

15.3.2 Verwendung von Listen

Um mehrere Ergebnisse ausgegeben zu bekommen, empfiehlt sich die Verwendung von sogenannten Listen, genauer von Objekten der Klasse list. Listen bestehen aus einzelnen Listeneinträgen, wobei diese Listeneinträge wiederum beliebige Objektklassen aufweisen können. Man könnte also eine Liste erstellen, deren erster Eintrag eine Matrix ist, deren zweiter Eintrag ein Vektor ist und so fort. Ein einfaches Beispiel:

```
> x <- matrix(0,nrow=2,ncol=2)
> y <- c("a","b","c")
> z <- 5
>
> a <- list(x,y,z)
> a
[[1]]
     [,1] [,2]
[1,]    0    0
[2,]    0    0

[[2]]
[1] "a" "b" "c"

[[3]]
[1] 5

> class(a)
[1] "list"
```

Zunächst werden drei Objekte x, y und z definiert. Über den Befehl list() werden diese zu einem Objekt der Klasse list namens a zusammengefasst. Wird dieses aufgerufen, werden die einzelnen Einträge angezeigt.

Um einzelne Einträge von Listen anzusprechen, verwendet man doppelte eckige Klammern:

```
> a[[2]]
[1] "a" "b" "c"
> a[[1]]
     [,1] [,2]
[1,]    0    0
[2,]    0    0
```

Hinter diesen doppelten eckigen Klammern können einfache eckige Klammern gesetzt werden, um innerhalb des Listeneintrags zu indizieren, wobei hier auf eine dem Format des Listeneintrags entsprechende Indizierung geachtet werden muss:

```
> a[[2]][2]
[1] "b"
> a[[1]][1,2]
[1] 0
```

Eine Möglichkeit, die Indizierung von Listen etwas zu erleichtern, besteht darin, den einzelnen Einträgen Namen zuzuweisen. Hierfür wird der Befehl names() in Kombination mit einer Zuweisung verwendet:

```
> names(a)
NULL
> names(a) <- c("x","y","z")
```

Der Befehl zeigt zunächst die Namen der einzelnen Einträge an. Das Ergebnis NULL zeigt, dass noch keine vergeben sind. Durch die Zuweisung auf names(a) wird dem ersten Eintrag der Name x gegeben, dem zweiten Eintrag der Name y und dem dritten Eintrag der Name z. Wird nun hinter dem Namen der Liste ein Dollarzeichen gesetzt, gefolgt vom Namen des Eintrages, wird dieser angezeigt:

```
> a$x
     [,1] [,2]
[1,]    0    0
[2,]    0    0
> a$z
[1] 5
```

Die Indizierung innerhalb der Einträge verläuft dann über einfache rechteckige Klammern, also beispielsweise a$x[1,2].

Nun ist wahrscheinlich bereits ersichtlich geworden, wie sich das ursprüngliche Problem aus dem vorausgegangenen Abschnitt lösen lässt:

```
> beispiel <- function(x) {
+   y <- 3*x+2
+   z <- x*x
+   k <- list(y,z)
+   return(k)
+ }
```

Zunächst werden innerhalb der Funktion die beiden temporären Objekte y und z erzeugt. Diese werden in eine Liste k geschrieben, welche mittels des Befehls return() ausgegeben wird. Nun liefert die Funktion folgendes Resultat:

```
> beispiel(1)
[[1]]
[1] 5

[[2]]
[1] 1

> class(beispiel(1))
[1] "list"
```

Man hat also nun die Ergebnisse beider Berechnungen.

15.3.3 Klassen und Methoden

Das Ergebnis der Beispielfunktion beispiel aus dem letzten Abschnitt hat die Klasse list. Möchte man die Klasse ändern, verwendet man den Befehl class() mit dem entsprechenden Objekt als Argument und nimmt eine Zuweisung hierauf vor, wobei die rechte Seite prinzipiell beliebigen Text enthalten kann:

```
> a <- beispiel(1)
> class(a)
[1] "list"
> class(a) <- "beispiel"
> class(a)
[1] "beispiel"
```

Das Objekt a, das ein Ergebnis unserer Beispielfunktion enthält, hat zunächst die Klasse list. Dies wird in eine Klasse beispiel geändert. Die Verwendung spezieller Klassen ermöglicht es, auf vorgefertigte Methoden zur Auswertung zurückzugreifen. Beispielsweise könnte man einem Ergebnisvektor die Klasse ts zuweisen, wobei ts für *time series*, also eine Zeitreihe, steht. Für solche Objekte gibt es Auswertungs- und Visualisierungsmöglichkeiten, die den Arbeitsaufwand teils erheblich reduzieren können. Zudem kann man auch eigene Methoden definieren.

Für unsere Klasse beispiel wollen wir nun eine spezielle Methode für den Befehl summary() definieren. Allgemein liefert der Befehl summary() Zusammenfassungen von Ergebnissen, wobei für viele Objektklassen bereits spezifische Methoden definiert sind. Diese kann man sich wie folgt anzeigen lassen:

```
> methods(summary)
 [1] summary.aov              summary.aovlist
 [3] summary.connection       summary.data.frame
 [5] summary.Date             summary.default
 [7] summary.ecdf*            summary.factor
 [9] summary.glm              summary.infl
[11] summary.lm               summary.loess*
[13] summary.manova           summary.matrix
[15] summary.mlm              summary.nls*
[17] summary.packageStatus*   summary.POSIXct
[19] summary.POSIXlt          summary.ppr*
[21] summary.prcomp*          summary.princomp*
[23] summary.stepfun          summary.stl*
[25] summary.table            summary.tukeysmooth*
```

Beispielsweise gibt es für Objekte der Klasse table eine spezielle Methode für den Befehl summary(). Zur Erstellung einer Methode für Objekte der Klasse beispiel könnte man wie folgt vorgehen:

```
> summary.beispiel <- function(x) {
+    print(x[[1]]+x[[2]])
+    }
```

Man erstellt eine Funktion, wobei der Name dieser Funktion aus zwei Teilen besteht. Der erste Teil gibt den Namen der eigentlichen Funktion wieder – in diesem Fall summary. Der zweite Teil folgt auf einen Punkt und gibt den Namen der Klasse wieder, für die die Methode gelten soll. Innerhalb der Definition der Funktion – also zwischen den geschweiften Klammern – lassen sich im Prinzip beliebige Auswertungen angeben. In diesem Beispiel wird die Summe der beiden Einträge der Liste gebildet:

```
> summary(a)
[1] 6
```

Wie hier ersichtlich wird, muss man den zweiten Teil des Funktionsnamens nicht angeben. Die Methode wird von R entsprechend der Klasse des angegebenen Objektes automatisch ausgewählt.

Die ursprüngliche Funktion beispiel() erzeugt allerdings noch nicht standardmäßig Objekte der Klasse beispiel. Um dies zu ändern, ergänzen wir eine Zeile in der Definition dieser Funktion:

```
> beispiel <- function(x) {
+   y <- 3*x+2
+   z <- x*x
+   k <- list(y,z)
+   class(k) <- "beispiel"
+   return(k)
+ }
```

Hier wurde nach der vierten Zeile eine Zuweisung eingefügt, die dafür sorgt, dass alle mit der Funktion beispiel() erzeugten Objekte die Klasse beispiel haben, so dass sie mit der speziellen summary() Methode ausgewertet werden können:

```
> b <- beispiel(2)
> summary(b)
[1] 12
> c <- beispiel(3)
> summary(c)
[1] 20
```

Der Vorteil der Verwendung von speziellen Methoden für existierende Befehle liegt darin, dass man im Idealfall nur eine kleine Zahl von Funktionen im Kopf behalten muss, diese aber gleichzeitig an die eigenen Bedürfnisse anpassen kann.

15.4 Ein einfaches Beispiel

Das bisher Beschriebene wollen wir nun nutzen, um ein sehr einfaches Beispiel zu erstellen. Hierbei werden wir einen zweifachen Würfelwurf simulieren. Mit „zweifacher Würfelwurf“ ist gemeint, dass ein Würfel zwei mal geworfen wird und wir beide Ergebnisse festhalten wollen. Zudem wollen wir dies n-mal wiederholen.

Hierfür kann folgende Funktion verwendet werden:

```
> wurf <- function(n=1) {
+   zahl <- 1
+   x <- matrix(0,nrow=n,ncol=2)
+   while (zahl<=n) {
+     x[zahl,] <- sample(1:6,2,replace=T)
+     zahl <- zahl+1
+   }
+   class(x) <- "wurf"
+   return(x)
+ }
```

Die Funktion hat den Namen wurf und nimmt als einziges Argument die Zahl der durchzuführenden zweifachen Würfe, wobei diese als Standard auf den Wert 1 gesetzt ist. In der zweiten Zeile wird eine Variable zahl definiert, die den Wert 1 aufweist. Über diese werden wir die Zahl der durchgeführten simulierten Würfe zählen – für jeden simulierten Wurf wird der Wert weiter unten um 1 erhöht. Anschließend erzeugen wir eine Matrix x. Diese hat n Zeilen, also der Zahl der Durchführungen entsprechend, und 2 Spalten, was der Zahl der Resultate pro Durchführung entspricht.

Hierauf folgt eine while()-Schleife, welche solange wiederholt durchgeführt wird, wie der Wert der Variablen zahl kleiner oder gleich dem Wert der Option n ist – solange also, bis die Zahl der durchgeführten Würfe der gewünschten Zahl entspricht. In Zeile 5 werden mittels des sample() Befehls zwei Werte aus den Zahlen 1 bis 6 mit Zurücklegen gezogen und entsprechend der Nummer des aktuellen Wurfs in die Ergebnismatrix geschrieben. Darauf wird die Zählvariable für die durchgeführten Würfe um den Wert 1 erhöht.

Auf die Schleife folgen noch zwei weitere Befehle. Zunächst wird der Matrix x die Klasse wurf zugewiesen. Und abschließend wird mittels des Befehls return() festgelegt, dass die Funktion wurf() die Ergebnismatrix x als Resultat ausgeben soll.

Eine alternative Definition der Funktion, die ohne while()-Schleife auskommt, sieht wie folgt aus:

```
> wurf <- function(n=1) {
+   x <- cbind(sample(1:6,n,replace=T),sample(1:6,n,replace=T))
+   class(x) <- "wurf"
+   return(x)
+  }
```

Die Generierung der Würfelwürfe erfolgt gebündelt in Zeile 2. Mittels des Befehls sample() werden zwei Vektoren an Zufallswerten erzeugt, die den ersten und den zweiten Wurf repräsentieren. Diese werden mittels des Befehls cbind() zu einer Matrix mit n Zeilen und 2 Spalten kombiniert. Diese Definition der Funktion ist deutlich sparsamer (und durch die Vermeidung von Schleifen auch schneller) als die erste, wenn auch vielleicht nicht so intuitiv zugänglich.

Unabhängig davon, welche der beiden Funktionsdefinitionen wir verwenden, ein Aufruf liefert nun Ergebnisse folgender Art:

```
> a <- wurf(5)
> a
```

```
     [,1] [,2]
[1,]    4    2
[2,]    6    2
[3,]    5    2
[4,]    2    6
[5,]    3    1
attr(,"class")
[1] "wurf"
```

In diesem Fall erhalten wir das Ergebnis von insgesamt fünf zweifachen Würfelwürfen. Anstelle des Wertes 5 kann man natürlich auch andere Werte verwenden. Wenn man allerdings 10000 Durchführungen simulieren möchte, ergibt sich bei der Anzeige ein Problem. Hier können wir nun eine einfache Methode für den Befehl summary() definieren, die uns auch bei einer großen Zahl an Würfen eine effektive Auswertung erlaubt:

```
> summary.wurf <- function(x) {
+   table(x[,1],x[,2])/dim(x)[1]
+   }
```

Zur Auswertung wird der Befehl table() ausgeführt. Wird diesem ein Vektor als Argument übergeben, werden die in diesem Vektor vorkommenden Werte und die Häufigkeit ihres Auftretens ausgegeben:

```
> z <- c(0,1,0,1,0,1)
> table(z)
z
0 1
3 3
```

Es wird also eine einfache Tabelle gebildet. Werden zwei Vektoren angegeben - wie bei unserer speziellen Methode zwei Spaltenvektoren – wird aus diesen eine Kreuztabelle erstellt. Um anstelle der absoluten Häufigkeiten relative Häufigkeiten zu erhalten, wird zudem durch die Zahl der Zeilen der Matrix dividiert, also durch die Zahl der Würfe. Das Ergebnis sieht nun so aus:

```
> a <- wurf(10000)
> summary(a)

        1      2      3      4      5      6
1 0.0278 0.0271 0.0292 0.0262 0.0274 0.0283
2 0.0256 0.0250 0.0303 0.0287 0.0254 0.0292
3 0.0289 0.0292 0.0289 0.0285 0.0250 0.0286
4 0.0301 0.0297 0.0275 0.0281 0.0260 0.0252
5 0.0277 0.0274 0.0270 0.0286 0.0270 0.0266
6 0.0306 0.0285 0.0285 0.0271 0.0285 0.0266
```

Insgesamt werden 10000 zweifache Würfelwürfe ausgeführt. In 2.78% aller Fälle wird zweimal die 1 gewürfelt. In 2.56% aller Fälle wird erst eine 2 und anschließend eine 1 gewürfelt und so fort. An diesem Beispiel wird schnell ersichtlich, dass die Definition einer speziellen Methode selbst im Falle einer großen Anzahl an zweifachen Würfelwürfen noch immer einfache Auswertungen erlaubt.

Literaturverzeichnis

Behr, A. and U. Pötter: 2010, *Einführung in die Statistik mit R*. München: Vahlen.

Bortz, J.: 2005, *Statistik für Human- und Sozialwissenschaftler*. Heidelberg: Springer, 6. edition.

Bortz, J. and G. A. Lienert: 2008, *Kurzgefasste Statistik für die klinische Forschung*. Heidelberg: Springer, 3. edition.

Bortz, J., G. A. Lienert, and K. Boehnke: 2008, *Verteilungsfreie Methoden in der Biostatistik*. Heidelberg: Springer, 3. edition.

Cleveland, W. S. and R. McGill: 1984, 'Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods'. *Journal of the American Statistical Association* 79, 531–554.

Crawley, M. J.: 2007, *The R Book*. Chichester: Wiley & Sons.

Dalgaard, P.: 2002, *Introductory Statistics with R*. New York: Springer.

Dolic, D.: 2003, *Statistik mit R. Einführung für Wirtschafts- und Sozialwissenschaftler*. München: Oldenbourg.

Kähler, W.-M.: 2002, *Statistische Datenanalyse. Verfahren verstehen und mit SPSS gekonnt einsetzen*. Braunschweig: Vieweg, 2. edition.

Ligges, U.: 2009, *Programmieren mit R*. Berlin: Springer.

Murrel, P.: 2009, 'R Graphics Devices'. http://www.stat.auckland.ac.nz/~paul/R/devices.html.

Sauerbier, T. and W. Voß: 2002, *Kleine Formelsammlung Statistik*. Fachbuchverlag Leipzig.

Venables, W. N., D. M. Smith, and R Core Development Team: 2009, 'An Introduction to R'. Version 2.10.0, abrufbar unter www.r-project.org.

Index