technische universität
dortmund

Fakultät Statistik

Programmierung mit R
Teil I

Uwe Ligges

TU Dortmund, WS 2019/20

1.0 Organisation

	Datum / Zeit	Raum
Vorlesung	10:15 – 11:15 Uhr	EF 50/HS 3
Übungen	wird bekanntgegeben (Beginn ab Mo., 21.10.)	CDI / 121

- Webseite der Vorlesung:
  `http://www.statistik.tu-dortmund.de/proginr.html`
- Einteilung der Übungsgruppen nach Anmeldung unter o.g. URL
- Anmeldefrist: Sonntag, 13.10.2019
- Für alle, die etwas Lernen und einen Leistungsnachweis möchten:
  - **Aktive Teilnahme an den Übungen! (Pflicht!)**
  - **Selbständiges Bearbeiten der Übungszettel!**
- **Fragen stellen und mitdiskutieren!**

1.1 Inhalt

Kapitel

1 Einführung

1.1 Inhalt

- Einführung (Hintergrund, Geschichte und Ausblick, Literatur)
- Zuweisungen, Datenstrukturen
- Import und Export (Daten, Programmcode, ...)
- Grafik
- Funktionen, der Workspace
- Konstrukte (Fallunterscheidungen, Schleifen)
- Umgang mit Zeichenketten
- Literatur

1.1 Inhalt des 2. Semesters

- Einführung und Wiederholung
- Funktionen 2 (Wiederholung, Scoping Rules, Workspace, Debugging)
- effiziente Programmierung (hinsichtlich Speicherplatz, Rechenzeit)
- Verteilungen und Stichproben
- Algorithmik und Simulation
- R Pakete verwenden
- Statistische Verfahren und das Formel-Interface
- Objektorientiertes Programmieren (S3, S4)
- Parallele Programme schreiben
- Erweiterbarkeit (Erstellen von Paketen, externer (C) Code, Namespaces)

1.2 Statistiksoftware

Kapitel

1 Einführung

1.2 Statistiksoftware

1.2 Selbst programmieren — Warum?

- Automatisierung sich wiederholender Abläufe
- Anpassung vorhandener Verfahren an neue Problemstellungen
- Implementierung neuer Verfahren
- Simulationen
- ...

1.2 Statistiksoftware: SAS

SAS

- Name:
  - zunächst: „Statistical Analysis Systems“
  - heute: „sas“
- SAS Institute, gegründet 1976
- 2016 mehr als 3 Mrd. US$ Jahresumsatz, 14000 Mitarbeiter
- Lehrveranstaltungen: SAS Kurs, Datenanalyse mit SAS
- Einsatz vor allem in Medizin, Pharmaindustrie, im CRM, beim Scoring

CRM: Customer Relationship Management

1.2 Statistiksoftware – SPSS

SPSS

- Name:
  - zunächst „Statistical Package for the Social Sciences“
  - dann „Superior Performing Software System“
  - heute „SPSS“
- gegründet 1968 als eigenständige Firma
- 2009 an IBM verkauft
- Lehrveranstaltungen: 1-2 tägige Kurse des SBAZ
- Einsatz vor allem in den Sozial- und Geisteswissenschaften, im CRM, beim Scoring

1.2 Statistiksoftware: R

R

- Entwicklung seit 1992
- Inspiriert von S / S-PLUS
- Lehrveranstaltung: Programmierung mit R
- Einsatz an Universitäten und Forschungsinstituten sowie immer mehr in der Industrie
- Lizenz: GPL 2 oder GPL 3 (freie Software, Open Source Software)
- mehr Details folgen

1.2 Statistiksoftware: Weiteres

Weitere Statistiksoftware:

- Gauss (USA, seit 1984)
- Minitab (USA, seit 1972)
- Stata (USA)
- Statistica (StatSoft, Deutschland)
- ...

Statistiksoftware für spezielle Einsatzgebiete:

- Lisrel (für Strukturgleichungsmodelle)
- StatXact (Spezialsoftware für exakte Tests)
- ...

1.2 Anforderungen an Statistiksoftware

Anforderungen an Statistiksoftware:

- Interaktive Arbeit mit Daten für die Datenanalyse
- Erstellung statistischer Grafik
- Hohe numerische Genauigkeit
- Hohe Rechengeschwindigkeit
- Verarbeitung großer Datenmengen
- Automatisierbarkeit von Methoden und sich wiederholender Abläufe
- Einfache Bedienbarkeit / Programmierbarkeit
- Nebenbedingungen: Preis, unterstützte Hardware, Parallelisierung, ...

Diese Anforderungen widersprechen sich teilweise. Je nach Gewichtung der Schwerpunkte erfolgt die Wahl einer entsprechenden Software.

Software kann niemals besser sein als ihr Benutzer!

1.3 Die Geschichte

Kapitel

1 Einführung

1.3 Geschichte

1.3 Die Geschichte

Am Anfang war S, eine 1984 von Richard Becker und John Chambers (vgl. das **braune** Buch) in den *Bell Laboratories* bei *AT&T* (heute bei *Lucent Technologies*) für Statistik, stochastische Simulation und Grafik entwickelte Programmiersprache.

The New S (Version 2) wurde 1988 von Becker, Chambers und Allan Wilks eingeführt (vgl. das **blaue** Buch), und enthält mehr oder weniger die heute bekannte **S** Funktionalität.

Version 3 von **S** wird 1992 in dem **weißen** Buch von Chambers und Trevor Hastie beschrieben, die Möglichkeiten hinzufügten, statistische Modelle einfach zu spezifizieren (in einer Art Formelnotation).

Die neueste Version 4 von **S** beschreibt Chambers im **grünen** Buch 1998. Ansätze der Implementierung dieser Version gibt es im **R** package *methods*.

1.3 Die Geschichte

Die *Association for Computing Machinery* hat John Chambers 1999 mit dem *Software System Award* ausgezeichnet: „**S** has forever altered the way people analyze, visualize, and manipulate data ...“

Man beachte, dass die Ziele von **S** sind:

- Interaktives Rechnen mit Daten
- Den Benutzer einfach zum Programmierer werden zu lassen
- Erstellen von Grafiken für explorative Datenanalyse und Präsentationen
- Einfache Wiederverwendung bereits entwickelter Funktionen

Zwei zu Programmpaketen realisierte Implementationen von **S** haben sich entwickelt: **S–PLUS** und **R** .

1.3 Die Geschichte

R ist eine freie Software (unter der GNU GENERAL PUBLIC LICENSE), die zunächst von Robert Gentleman und Ross Ihaka wegen zu hoher Kosten für **S–PLUS** für Lehrzwecke entwickelt wurde.

Später vereinigte sich das **R** Development Core Team, das z.Zt. aus 20 Personen aus Forschung und Wirtschaft besteht, die rund um den Globus verteilt sind (darunter **S** Erfinder John Chambers).

Die erste zur Sprache **S** in Version 3 vollständig kompatible Version (1.0.0) erschien am 29.02.2000 (Schaltjahr!). Die aktuelle Version ist **R-3.6.1**.

Der Name **R** entstand, da er nahe an **S** liegt und

- Ross' und Robert's Vornamen mit **R** beginnen oder
- man eine „**r**educed version of **S**“ wollte.

1.3 Vor- und Nachteile von R

Vorteile von R

- Open Source
  - Keine „black box“, **alles** kann nachvollzogen werden
  - „Am Puls der Forschung“
  - Erweiterbarkeit, Skalierbarkeit
    (Interfaces zu anderen Programmiersprachen, Datenbanken etc.)
  - Auf (fast) jedem Betriebssystem / jeder Plattform lauffähig
- Support
  - Meist schnelle und kompetente Beantwortung von Fragen auf *R–Help* (eine Mailing–Liste).
  - Fehler werden meist innerhalb weniger Tage behoben.

1.3 Vor- und Nachteile von R

Nachteile von R

- Keine komfortable grafische Benutzeroberfläche (aber R Commander, JGR u.a.)
- Keine interaktive Grafik (aber iplots, rgl u.a.)
- ...

1.3 Was ist R denn jetzt?

- A language and environment for data analysis and graphics
- Open Source
- Mittel zum Technologie- / Methodentransfer durch Packages
- Ein Datenzugriffsmechanismus für
  - text Dateien und **R** Workspaces
  - Datensätze von **S–PLUS, SPSS, Minitab, SAS Xport**, ...
  - Relationale Datenbanken – ODBC, PostgreSQL, MySQL, ...

1.3 Wo kann man R bekommen?

Alles rund um **R** gibt es auf der **R** Homepage `https://www.R-Project.org` und auf dem CRAN (Comprehensive **R** Archive Network): `https://CRAN.R-Project.org`, z.B.:

- **R** sources und binaries für diverse Betriebssysteme
- fast 2^{14} **R** Pakete zu diversen (statistischen) Verfahren
- andere relevante Software (geeignete Editoren, Erweiterungen, ...)
- Bücher, Publikationen, Manuals, FAQs, Dokumentation
- The **R** Journal
- Zugriff auf Mailing Listen (R–help, R–devel)
- Bug–Tracking System und Entwickler–Seiten
- Links zu relevanten anderen Projekten (z.B. Bioconductor, Omegahat)

1.4 Los geht's — R als Taschenrechner

Kapitel

1 Einführung

1.4 R als Taschenrechner

Wie kann man R starten?

Windows: Im Startmenü auf **R** klicken.

andere Betriebssysteme: **R** in die Kommandozeile eintippen.

1.4 Los geht's — R als Taschenrechner

Beispiele:

```
1 + 3
3 + 5 * 2
(4 / pi)^3      # Dies ist ein Kommentar (nach dem Zeichen "#").
0 / Inf
0 / 0           # Nicht definiert -> NaN
Inf - Inf       # Nicht definiert -> NaN
sin(pi / 2)     # Funktionsname: sin ;  Argument: pi / 2
```

Es gilt „Punkt- vor Strichrechnung"!

Es ist aber immer sinnvoll Klammern zu setzen, um die Lesbarkeit eines Programms zu verbessern und Irrtümern vorzubeugen.

1.4 Grundlegendes Rechnen

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

1.4 Funktionen

Gerade ist uns eine Vielzahl verschiedener Funktionen begegnet.

- Jegliches Arbeiten geschieht mit Funktionen.
- Ein Funktionsaufruf hat die Form, dass der Funktionsname angegeben wird, dem in runden Klammern ein oder mehrere durch Kommata getrennte Argumente folgen:

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

- Argumente mit Voreinstellungen müssen beim Funktionsaufruf nicht mit angegeben werden.
- Falls es Voreinstellungen für Argumente gibt, so sind diese in der Regel auf der Hilfeseite der entsprechenden Funktion angegeben.
- Später gibt es noch viel mehr zu Funktionen!

1.4 Hilfe!

Starten des Hilfesystems im Browser	`help.start()`
Hilfe zu einer Funktion	`help("Funktionsname")` `?Funktionsname`

Beispiele:

```
help.start()                  # Der Browser öffnet sich.
options(help_type="html") # Hilfesystem auf html umstellen
?abs                          # Hilfe zur Funktion abs()
```

1.4 Mehr Hilfe!

Sollte die Dokumentation in den Hilfeseiten nicht ausreichen, so sucht man üblicherweise in der folgenden Reihenfolge in weiteren Quellen:

- mitgelieferte FAQs
- mitgelieferte Manuals
- evtl. vorhandene Bücher
- Archive der Mailinglisten R–help und evtl. R–devel
- *zuletzt* eine Frage an die Mailingliste *R–help* richten

Sollte man einen Bug finden, so schreibt man

- an *R–help*, wenn man nur glaubt, dass es ein Bug ist,
- an *R–devel*, wenn man sich nicht ganz sicher ist,
- an *R–bugs*, wenn man sich absolut sicher ist

1.4 Zuweisungen

```
x <- 3.25   # eine Zuweisung

x<-4        # schlecht lesbar
6->x        # veraltet, schlecht lesbar, verwirrend
x=3.25      # geht meist auch, aber nicht überall

neue.Variable1 <- x
neue.Variable2<-neue.Variable1+2#kaum lesbar!
neue.Variable2 <- neue.Variable1 + 2      # jetzt besser lesbar
```

Fazit:

Man sollte sinnvoll Leerzeichen einfügen, um die Lesbarkeit zu erhöhen.

Vorsicht bei `<-` und `->` !

Namen sollten mit einem Buchstaben beginnen, dürfen aber auch Zahlen enthalten.

1.5 Logik

Kapitel

1 Einführung
1.5 Logik

Zunächst einige Überlegungen zur Logik an der Tafel ...

Vergleiche	`==`, `!=`, `>`, `<`, `>=`, `<=`
Konstanten	`TRUE`, `T`, `FALSE`, `F` und `NA` (In R `T`, `F` vermeiden!)
Operatoren	`!` (Negation), `xor()` (ausschließendes oder) `&`, `&&`, `|`, `||` (und, oder)

1.5 Logik

Beispiele:

```
4 < 3                                                    #-> FALSE
(3 + 1) != 3                                             #-> TRUE
(3 >= 2) & (4 == (3 + 1))                                #-> TRUE
TRUE + FALSE + TRUE          # TRUE = 1, FALSE = 0       #-> 2
-3<-2                        # falsch - Zuweisung!       #-> Error
-3 < -2                                                  #-> TRUE
T <- 55                      # In R nicht T / F verwenden!
```

1.5 Logik

Offenbar werden zum Rechnen die logischen Werte `TRUE` und `FALSE` zu den numerischen Werten 1 bzw. 0 konvertiert.

Andererseits werden für logische Vergleiche alle numerischen Werte zu dem logischen Wert `TRUE` konvertiert mit Ausnahme der 0, die zu `FALSE` konvertiert wird.

1.5 Logik

Die Operatoren `&&` und `||` arbeiten *nicht* vektorwertig (später ...), sondern liefern immer einen einzelnen Wahrheitswert (manchmal sehr sinnvoll, aber gefährlich).
Es wird nur soviel der Logik–Verknüpfung ausgewertet, wie zu einer korrekten Aussage benötigt wird (effizient).

Die Operatoren `&` und `|` hingegen arbeiten vektorwertig.

1.5 Logik

Beispiele:

```
FALSE && TRUE       # die rechte Seite wird NICHT ausgewertet
TRUE && TRUE        # die rechte Seite wird ausgewertet
TRUE || FALSE       # die rechte Seite wird NICHT ausgewertet
TRUE || (x <- 3)
FALSE || TRUE       # die rechte Seite wird ausgewertet
FALSE || (x <- 3)
c(TRUE, TRUE) & c(FALSE, TRUE)      # vektorwertig
c(TRUE, TRUE) && c(FALSE, TRUE)     # NICHT vektorwertig
```

1.5 Logik

`any()`: Ist irgendein Element eines Vektors `TRUE`?

`all()`: Sind alle Elemente eines Vektors `TRUE`?

Beispiele:

```
a1 <- c(FALSE, FALSE)
a2 <- c(FALSE, TRUE)
a3 <- c(TRUE, TRUE)

any(a1)      # FALSE
any(a2)      # TRUE
all(a2)      # FALSE
all(a3)      # TRUE
all(!a1)     # TRUE
```

1.5 Logik – fehlende Werte

Fehlende Werte werden durch `NA` dargestellt

- Testen auf `NA`s mit `is.na()`, aber *nicht* mit `x == NA`.
- `NA`s entfernen mit `na.omit()`.
- Der Wert `NaN` (Not a Number, nicht definiert) wird wie `NA` behandelt (Aussage `is.na(NaN)` ist wahr).

Beispiele:

```
x <- c(2, NA, 4)
x == NA           # [1] NA NA NA
is.na(x)          # [1] FALSE  TRUE FALSE
```

1.6 Sprache und Editoren

Kapitel

1 Einführung

1.6 Sprache und Editoren

1.6 Alles Objekte und Vektoren!

- *Alles* ist ein Objekt (jegliche Art Daten und Funktionen)!
- **R** ist eine *objektorientierte Sprache*!
  (Das heben wir uns für das Ende des Kurses auf!)
- Es gibt *nur* Daten und Funktionen!
- **R** ist eine *funktionale Sprache*!
  D.h. egal was man durch gültige Eingabe bewirkt, wird durch Funktionen getan (auch die Ausgabe von Werten auf der Konsole!).
  Jegliches Arbeiten mit Daten geschieht über Funktionen.
- **R** ist eine *vektorbasierte Sprache*!
  Jedes Objekt in **R** wird *intern* durch einen Vektor repräsentiert, selbst Funktionen!
- Die sehr nützliche Funktion `str()` zeigt die Struktur eines Objekts an.

1.6 Der Workspace — und wie man R beendet

- **R** hält alle Objekte in einem *Workspace*.
- Die Funktion `ls()` zeigt alle Objekte im aktuellen Workspace an.
- Mit `rm(Objektname)` kann ein Objekt aus dem Workspace gelöscht werden.
- Das Beenden von **R** geschieht mit der Funktion `quit()`.
  Beim Beenden wird auch die Frage gestellt, ob man den aktuellen Workspace speichern möchte. Bei Beantwortung der Frage mit `JA`:
  - Die *history* der letzten Befehle wird in der Datei `.Rhistory` gespeichert.
  - Die aktuellen Objekte im Workspace werden in der Datei `.Rdata` gespeichert.
- Den Workspace mit `save.image()` speichern und mit `load()` wieder laden.
- Als Standard–Speicherort für die Datei des Workspace wird das Verzeichnis gewählt, aus dem R aufgerufen wird, bzw. das unter Windows als Arbeitsverzeichnis in der Verknüpfung gesetzt ist.

1.6 Komfortable Editoren für R

In der Kommandozeile von **R** kann man zwar sehr schön einfache Berechnungen durchführen, das Erstellen von Funktionen und Programmen fällt jedoch schwer.
Es empfiehlt sich daher, einen Editor zu verwenden.
Eine Funktion kann dann in einer Textdatei auf der Festplatte gespeichert werden und mit `source("Dateiname")` geladen werden. Einfachere Funktionen oder Programmteile können auch mittels Copy&Paste übertragen werden.

Unter `http://cran.r-project.org/other-software.html` gibt es u.a. die Erweiterung *ESS* (Emacs Speaks Statistics) für den unter allen Betriebssystemen laufenden, mächtigen und freien Editor *Emacs* bzw. *XEmacs*.

1.6 Komfortable Editoren für R

Mit dieser Erweiterung können vom *(X)Emacs* aus Statistikpakete (z.B. **S–PLUS**, **SAS**, **R** usw.) komfortabel gesteuert werden. An der *ESS* Entwicklung sind auch Mitglieder des **R** Development Core Teams beteiligt.

Auf den Linux Rechnern der Fakultät ist *XEmacs* mit *ESS* installiert.

R wird im *(X)Emacs* gestartet durch die Tastenkombination:

`ESC-x, Shift-R, Enter`

1.6 Komfortable Editoren für R

Sonst sehr empfehlenswert (insbesondere auch für Windows):

- Die Entwicklungsumgebung *RStudio*: `http://www.rstudio.com/`
- Der freie Editor *Tinn-R*:
  `https://sourceforge.net/projects/tinn-r/`

Wie auch *ESS* bieten die hier genannten Werkzeuge neben Syntax–Highlighting Kommunikationsmöglichkeiten mit **R** über Shortcuts und die „Klickwelt“.

2.0 Datenstrukturen

Unter Datenstrukturen versteht man eine Beschreibung dessen, wie die Daten dargestellt werden und angeordnet sind.

- Je nach Problemstellung und Art der Daten gibt es verschiedene geeignete Darstellungsarten.
- In der Informatik benutzt man Datenstrukturen um Daten so anzuordnen, dass effizient damit zu rechnen, z.B. Bäume für Sortierverfahren.
- In der Statistik werden Datenstrukturen benutzt, die die Natur der Daten berücksichtigen, so dass sie angemessen repräsentiert werden und in Modellen spezifiziert werden können.

Wir werden u.a. folgende Datenstrukturen kennenlernen:
Vektoren, Matrizen, Arrays, Data Frames, Listen, ...

2.1 Vektoren

Kapitel

2 Datenstrukturen

2.1 Vektoren

2.1 Vektoren

Mit der Funktion `c()` (combine) kann man Vektoren erzeugen, z.B.:

```
x <- c(4.1, 5.0, 6.35)
x <- c(7.9, x, 2)
y <- c("Hallo", "Welt")
y <- c("Statistik", TRUE, x)    # alles wird zu "character"
```

Die Daten innerhalb eines Vektors müssen konsistent sein. U.a. sind folgende Typen möglich:

`logical`	Wahr (`TRUE`) oder Falsch (`FALSE`)
`numeric [integer]`	Ganzzahlen
`numeric [double]`	reelle Zahlen in doppelter Maschinengenauigkeit
`complex`	Komplexe Zahlen
`character`	Buchstaben und Zeichenfolgen (strings)

2.1 Vektoren

Bei nicht konsistenter Eingabe wird das „niedrigste“ Niveau gewählt, wie etwa im letzten Beispiel:

```
> y
[1] "Statistik"   "TRUE"  "7.9"   "4.1"   "5"      "6.35"  "2"
```

Alle Typen sind als character darstellbar, aber es ist nicht eindeutig möglich, das Wort „Statistik“ als Zahl oder gar als logischen Wert darzustellen.

Analog wird im folgenden Beispiel die reelle Zahl als komplexe darstellbar sein, aber nicht umgekehrt:

```
c(5, 2+1i)      # 5+0i 2+1i
```

- Vektoren *transponieren*: `t(x)`
- Vektoren *multiplizieren*: `x %*% y`

2.1 Folgen und Wiederholungen

Folgen	`seq(Anfang, Ende, by = Abstand)`
	`seq(along = Objekt)`
	`Anfang:Ende`
Wiederholungen	`rep(Objekt, Anzahl)`
	`rep(Objekt, each = Anzahl)`

Beispiele:

```
1:10                    #-> 1 2 3 4 5 6 7 8 9 10
x <- seq(4, 14, 2)      #-> 4 6 8 10 12 14
rep(3, 12)              #-> 3 3 3 3 .....
rep(x, 2)               #-> 4 6 8 ... 14 4 6 8 ... 14
rep(x, each = 2)        #-> 4 4 6 6 8 8 ... 14 14
rep(3:5, 1:3)           #-> 3 4 4 5 5 5
rep(TRUE, 3)            #-> TRUE TRUE TRUE
seq(along = x)          #-> 1 2 3 4 5 6
```

2.1 Rechnen mit Vektoren

Das Rechnen mit Vektoren geschieht komponentenweise, wie z.B. in:

```
2:4 * 1:3                          #-> 2 6 12
x <- c(4.1, 5.0, 6.35) * 2         #-> 8.2 10 12.7
x + 5:7                            #-> 13.2 16 19.7
3:5 - 1:6                          #-> 2 2 2 -1 -1 -1
t(2:4) %*% 1:3                     #-> 20
```

Stimmen die **Längen** (`length(x)`, auch bei anderen Objekten) von Vektoren nicht überein (Längen $m < n$), so

- wird der kürzere Vektor (oder Skalar) so oft wiederholt, wie nötig (`c(1, 2, 3) * 2 = c(1, 2, 3) * c(2, 2, 2)`)
- oder er wird durch Wiederholung der ersten $n - m + 1$ Elemente verlängert, bis er passt (mit Warnung)

2.1 Indizierung von Vektoren

Es können einzelne Elemente eines Vektors mit Hilfe des zugehörigen Index angesprochen werden. Dazu wird dem Namen des Vektors der entsprechende Index in eckigen Klammern nachgestellt, z.B.: `x[3]`.

- Mehrere Indices können als Vektor angegeben werden.
- Voranstellen eines Minus–Zeichens ermöglicht eine inverse Auswahl (z.B. zum Ausschließen eines oder mehrerer Elemente).
- Auch logische Indizierung ist möglich
  (`TRUE` für „gute“, `FALSE` für „schlechte“ Elemente)
- Benannte Elemente können über ihren Namen angesprochen werden.
- Man kann auch einer Indizierung einzelne Elemente eines Vektors zuweisen, um Elemente eines Vektors zu ersetzen.

2.1 Indizierung von Vektoren

Beispiele:

```
x <- c(3, 6, 9, 8, 4, 1, 2)
length(x)                  # 7
x[3]                       # 9
x[c(4, 2)]                 # 8 6

x[-3]                      # 3 6 8 4 1 2
x[- c(2, 3, 7)]            # 3 8 4 1

x < 4                      # TRUE FALSE FALSE FALSE FALSE TRUE TRUE
logik.vektor <- x < 4
x[logik.vektor]
x[x < 4]                   # 3 1 2
```

2.1 Indizierung von Vektoren

Beispiele:

```
x <- c(besser = 3, schlechter = 5, egal = 2)
x["besser"]          #-> 3

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

2.1 Zusammenstellung nützlicher Funktionen

`sort()`	Sortieren
`rev()`	Umkehren eines Vektors: rev(sort()) für abst. Sort.
`rank()`	Vektor der Ränge (ohne/mit Entfernen von Bindungen)
`order()`	Vektor mit Indexzahlen, der zum Sortieren verwendet wird.
`identical()`	Überprüfung von exakter Gleichheit zweier Objekte
`all.equal()`	Überprüfung auf numerische Gleichh. bzgl. Maschinen-Rechengenauigkeit
`unique()`	Entfernt mehrfach vorkommende Werte aus einem Vektor
`duplicated()`	Liefert `TRUE` bei mehrfach vorhandenen Werten
`which()`	Welche Elemente erfüllen eine gegebene Bedingung?
`cumsum()`	Erzeugt für jedes i die Summe von x_1 bis x_i
`choose()`	Effiziente Berechnung des Binomialkoeffizienten
`gamma(), lgamma()`	Gammafunktion bzw. deren ln

2.2 Matrizen und Arrays

Kapitel

2 Datenstrukturen

2.2 Matrizen und Arrays

2.2 Matrizen

Matrizen werden mit der Funktion

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE)
```

erzeugt, wobei die *Argumente* folgende Bedeutung haben:

`data`	Ein Vektor mit den Daten, sortiert nach Spalten oder Zeilen.
`nrow`	Zeilenanzahl
`ncol`	Spaltenanzahl (unnötig, wenn `data` und `nrow` angegeben ist)
`byrow`	Falls `TRUE`, wird die Matrix zeilenweise aufgebaut, sonst spaltenweise

Beispiele:

```
X <- matrix(c(4, 7, 3, 8, 9, 2), nrow = 3) # spaltenweise
Y <- matrix(c(4, 7, 3, 8, 9, 2), nrow = 3, byrow = TRUE)
Z <- matrix(c(4, 7, 3, 8, 9, 2), ncol = 3)
t(Z) == Y                                  # transponieren - OK!
```

2.2 Indizierung von Matrizen

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

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

Beispiele:

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

2.2 Funktionen für Matrizen

Wichtige und nützliche Funktionen zum Rechnen mit Matrizen:

`%*%`	Matrixmultiplikation
`chol()`	Choleski–Zerlegung einer Matrix
`diag()`	Abfragen und Setzen der Hauptdiagonalen
`dimnames()`	Spalten- und Zeilennamen
`eigen()`	Eigenwerte und -vektoren
`kappa()`	Konditionszahl einer Matrix
`ncol(), nrow()`	Anzahl Spalten bzw. Zeilen einer Matrix
`qr()`	QR–Zerlegung
`solve()`	Invertierung einer Matrix
`svd()`	Singulärwertzerlegung
`t()`	Transponieren einer Matrix

2.2 Funktionen für Matrizen

Für Experten:

Wir erinnern uns: *Eigentlich ist alles ein Vektor.* Daher ist die resultierende Matrix eigentlich „nur“ der Vektor `data` mit entsprechenden Dimensionsattributen.

Beispiele:

```
X <- matrix(c(4, 7, 9, 2), nrow = 2)
str(X)                          #->  num [1:2, 1:2] 4 7 9 2
t(X)                            # Transponiert die Matrix
solve(X)                        # Invertiert die Matrix
diag(X)                         # Diagonale extrahieren
```

2.2 Arrays

Bisher wurden die Datenstrukturen vom Skalar über den Vektor bis zur Matrix erweitert. Eine weitere Verallgemeinerung ist das *Array*.

Arrays können beliebig viele Dimensionen besitzen und werden mit dem Befehl

```
array(data, dim = length(x))
```

erzeugt, wobei als Argument mit `dim` ein Vektor mit der Anzahl der Elemente jeder Dimension angegeben werden kann.

Die Indizierung erfolgt analog zur Indizierung von Vektoren und Matrizen.

Beispiele:

```
X <- array(1:60, dim = c(3, 5, 4))
X[2, 2, 2]                                  #-> 20
X <- array(1:60, dim = c(3, 5, 2, 2))
X[2, 3, 1, 2]                               #-> 38
```

2.3 Listen

Kapitel

2 Datenstrukturen

2.3 Listen

2.3 Listen

Eine sehr flexible Datenstruktur sind *Listen*. Jedes Element einer Liste kann einen unterschiedlichen Datentyp besitzen, so kann eine Liste z.B.

- verschieden lange Vektoren / Matrizen unterschiedlichen Typs enthalten,
- selbst wieder Element einer Liste sein.

Es ist manchmal auch sinnvoll, den Elementen einer Liste Namen zuzuweisen.

Listen werden mit `list(Element1, Element2, ...)` bzw. benannte Listen mit

`list(E1 = Element1, E2 = Element2, ...)` erzeugt.

Der Zugriff auf Elemente einer Liste erfolgt mittels des `[[ ]]` Operators.

2.3 Listen

- Im Gegensatz zu Vektoren kann hier nicht vektorwertig indiziert werden.
- Namen statt eines Index sind möglich.
- Auf benannte Elemente kann man einfacher mit dem `$` Operator zugreifen:

```
Liste$Elementname
```

2.3 Listen

Beispiele:

```
L1 <- list(c(1, 2, 5, 4), matrix(1:4, 2), c("A", "Be"))
L1[[2]]                                  # Das zweite Element von L1
L1[[2]][2, 1]                            # davon das Element [2, 1]
L1[[1]][3]                               #-> 5
L2 <- list(Info = "Unsinn", Liste = L1)
L2$Info                                  # Das Element "Info"
L2[["Info"]]                             # Das Element "Info"
L2[[2]][[1]][3]                          # doppelter Listenzugriff
L2$Liste[[1]][3]                         #-> 5 (das gleiche in grün)
str(L2)  # liefert wieder detaillierte Einsicht in die Struktur
```

Experten: Intern ist alles ein Vektor! Was ist aber der Unterschied zwischen `L2[1]` und `L2[[1]]` in obigem Beispiel?

Tipp: Vektoren dürfen nur Elemente gleichen Typs mit Länge 1 enthalten.

2.4 Dataframes

Kapitel

2 Datenstrukturen
2.4 Dataframes

Bei *Dataframes* (Datentabellen ?), die mit `data.frame()` erzeugt werden können, handelt es sich um spezielle *Listen*, die die Einschränkung haben, dass die einzelnen Elemente nur Vektoren gleicher Länge sein dürfen.

- *Dataframes* sind *die* typische Datenstruktur für Datensätze!
- Sehr viele Funktionen erwarten *Dataframes* als Argument(e).
- *Dataframes* können wie *Listen und* wie *Matrizen* indiziert werden.

2.4 Dataframes

Beispiele:

```
Name <- c("Olaf", "Sven", "Anja", "Martina", "Eike")
MN <- c(89045, 43678, 88475, 69781, 88766)
HF <- c(rep("Statistik", 2),
        rep("Datenanalyse", 2),
        "Mathematik")
Erstsemester <- data.frame(Vorname = Name,
                           Matrikel.Nr = MN, Hauptfach = HF)
Erstsemester$Matrikel.Nr[3]
Erstsemester[2:3, 2]
```

2.4 subset() und %in%

Häufig möchte man bestimmte Zeilen eines *Dataframes* extrahieren, abhängig von Werten gewisser Variablen.

- Prinzipiell geht das per Indizierung, wie etwa:

```
Erstsemester[Erstsemester$Hauptfach == "Statistik", ]
```

- Eine mächtige(re), einfach zu bedienende Funktion zu diesem Zweck ist `subset()`:

```
subset(Erstsemester, Hauptfach == "Statistik")
```

- Wenn man dazu noch den Operator `%in%` verwendet, hat man sehr komplexe Auswahlmöglichkeiten in *Dataframes*:

```
subset(Erstsemester,
       Hauptfach %in% c("Statistik", "Datenanalyse"))
```

2.4 split() und merge()

- Um einen *Dataframe* nach den Werten einer Variable aufzuteilen, benutzt man z.B.:

  `split(Erstsemester, Erstsemester$Hauptfach)`.

- `merge()` vereinigt zwei verschiedene *Dataframes*.

Beispiel anhand des Erstsemester Dataframes:

```
NNamen <- data.frame(
    Matrikel.Nr = c(43678, 88475,  88766, 89045),
    Nachname = c("Troschke", "Busse", "Michaelis", "Schoffer"))

merge(Erstsemester, NNamen, all = TRUE)
```

Ein detailliertes Beispiel findet man unter `?merge`.

2.5 Verschiedenes

Kapitel

2 Datenstrukturen

2.5 Verschiedenes

2.5 factor() und ordered()

Zur Darstellung diskreter und qualitativer Merkmale bieten sich Faktoren an. Zur Erzeugung von Objekten für Merkmale ohne natürliche Anordnung, z.B. verschiedene Farben, verwendet man die Funktion `factor()`.

Für solche Merkmale, die angeordnet werden können, gibt es `ordered()`.

Wichtig werden Faktoren im Bereich linearer Modelle und zur Aufteilung eines Datensatzes gemäß solch eines Faktors, insbesondere auch zur Grafikerstellung.

2.5 factor() und ordered()

Beispiele:

```
trt <- factor(rep(c("Control", "Treated"), c(3, 4)))
trt <- factor(rep(c("Statistik", "Datenanalyse"), c(3, 4)))
str(trt)                    # interne Kodierung des Faktors
mode(trt)                   # wird als Zahl gespeichert
length(trt)                 # Anzahl Einträge
table(trt)                  # Tabelle der Levels
```

2.5 Klassen nach S 4

Ein Abstecher zu **S** 4. Bei (Objekt-) *Klassen*, die nach den „neuen“ Konventionen erzeugt sind, handelt es sich um eine weitere Datenstruktur.

Klassen können in sogenannten *Slots* beliebige andere Objekte enthalten, meist also Vektoren, Matrizen, Dataframes, Listen, etc. Auch wenn man zunächst noch keine eigenen Klassen erzeugen will, kann man doch auf ein Objekt in einem Slot eines existierenden Klassenobjekts zugreifen wollen, nämlich mit dem `@` – Operator, z.B.:

```
meinKlassenObjekt@derInteressanteSlot
```

2.5 Klassen nach S 4

Beispiele:

```
setClass("neu", representation(x = "numeric", y = "numeric"))
n1 <- new("neu", x = rnorm(10), y = 1:10)

n1[1]; n1[[1]]; n1$x              # Taugt alles nicht!
n1@x                              # Die Lösung!!!
```

2.5 Allgemeines zu Datenstrukturen

- Wenn man ein Objekt indiziert, kann ein niedriger dimensionales Objekt entstehen, ohne dass man es will (Probleme in komplexen Programmen).
  Die Lösung: `drop = FALSE` spezifizieren, wie in `X[1, 1, drop = FALSE]`.
- Neben den Funktionen `str()` und `length()`, kann auch `mode()` nützlich sein.
- Es gibt natürlich auch komplexe Zahlen, z.B.: `2+3i`.
- `View(x)` zeigt das Objekt `x` im Spreadsheet, falls möglich.

2.5 Allgemeines zu Datenstrukturen

Beispiele:

```
X <- matrix(1:4, 2)
X[1, 1]
X[1, 1, drop=FALSE]                    # ohne Verlust von Dimensionen

sqrt(-1:1)                             # OK.
sqrt(-1:1 + 0i)                        # So geht's auch komplex!

View(iris)                             # ansehen im Spreadsheet
```

3.1 Daten einlesen und exportieren

Kapitel

3 Daten IO

3.1 Daten in Textdateien

Wer Daten analysieren will, muss diese zunächst einmal in das jeweilige Statistik–Paket einlesen / importieren (und später evtl. wieder schreiben / exportieren).

Am einfachsten ist das für Textdateien mit den Funktionen `read.table()`, `scan()` und `read.fwf()`, aber auch das Lesen von komplizierteren Strukturen, aus Datenbanken (*data warehouses*, z.B. der Formate *ODBC, PostgreSQL, MySQL,* ...), Binärdateien, oder Formaten anderen Statistikpakete z.B. (S–PLUS, SPSS, Minitab, SAS Xport, ...) ist möglich.

3.1 Daten einlesen – `read.table()`

Da es zur Funktion `read.table()`, die für das Einlesen eines Datensatzes als *Dataframe* gedacht ist, eine große Menge verschiedener Argumente gibt, werden hier nur einige aufgelistet (weitere siehe `?read.table`):

`file`	(Pfad und) Dateiname
`header = FALSE`	Spaltennamen vorhanden ?
`sep = ""`	Spaltenseparator
`dec = "."`	Dezimalzeichen

Beispiele:

```
X <- data.frame(A = c(2, 5, 7), B = c("a", "h", "i"))
write.table(X, "c:/temp/X_Daten.txt")
Y <- read.table("c:/temp/X_Daten.txt")
```

3.1 Daten einlesen – `scan()`, `readLines()`

- Eine wesentlich mächtigere, aber auch komplizierter zu handhabende Funktion zum Lesen von `ASCII`–Dateien ist `scan()`. Damit können auch Dateien mit sehr verschiedenen und komplizierten Datenstrukturen eingelesen werden.
  Ein Blick in die Hilfe (`?scan`) lohnt sich!
- Zum Lesen von `ASCII`–Dateien unterschiedlich langer Zeilen eignet sich `readLines()`.

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

3.1 Daten schreiben – `write.table()`, ...

Analog zu den eben kennengelernten Funktionen zum Einlesen von `ASCII`–Daten, gibt es auch solche zum Schreiben:

- `write.table()` ist das Analogon zu `read.table()` (siehe das Beispiel dort), mit sehr ähnlichen Argumenten.
- `write()` kann als „Gegenteil“ von `scan()` angesehen werden, wobei insbesondere das Argument der Spaltenzahl, `ncolumns`, dem Voreiligen Streiche spielen kann.

3.2 Datenbanken

Kapitel

3 Daten IO

3.2 Datenbanken

Wer mit wirklich großen Datenbeständen arbeitet, wie etwa bei

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

hat keine Daten der Größe Kilo- oder MegaByte, sondern kommt in Bereiche von Giga- oder TeraByte (10^{12}).

3.2 Datenbanken

Man kann also nicht mehr den vollständigen Datensatz auf den Rechner laden und Auswertungen machen, sondern man muss Teilmengen der Daten von der Datenbank anfordern und evtl. schon elementare Rechnungen vom Datenbanksystem durchführen lassen.

Zur Kommunikation mit Datenbanken (hier wird meist die Sprache *SQL* verwendet: „**S**tructured **Q**uery **L**anguage“) bieten sich u.a. die folgenden **R** packages an (Details in der jeweiligen Dokumentation):

- *PostgreSQL*, *RmSQL*, *RMySQL*, *RSQLite* und *RPgSQL* für verschiedene *SQL* Systeme (meist unter Unix).
- *RODBC* für Microsofts Standard der *ODBC* Datenbanken (z.B. in *Access, Excel*), aber auch mit der Möglichkeit z.B. über MyODBC Treiber auf MySQL Datenbanken zuzugreifen.

3.2 Datenbanken — RODBC

Beispiel 1: Um unter Windows mit Hilfe des packages *RODBC* auf eine MySQL Datenbank zuzugreifen, muss die Datenquelle zunächst unter Windows als *DSN* (Data Source Name) bekannt gemacht werden.

Unter Windows 7 findet man unter *Start – Systemsteuerung – Verwaltung – Datenquellen (ODBC)* die Möglichkeit, eine Benutzer-DSN einzutragen. Hier wählt man den *MySQL ODBC* Treiber, der zunächst evtl. noch installiert werden muss, und trägt dann z.B. ein:

```
Data Source Name: mineur          (Name, den man in R angibt)
Host (oder IP):   129.217.206.14  (Name des Servers)
Database Name:    mineur          (Name auf dem Server)
User:             Student         (Benutzername auf dem Server)
Password:         Student         (zugehöriges Passwort)
```

3.2 Datenbanken — RODBC

Danach kann man dann Befehle aus dem Paket benutzen (siehe die Online-Hilfe) und z.B. sagen:

```
library("RODBC")                          # das Paket laden
channel <- odbcConnect("mineur", case="nochange")
   # Verbindung öffnen zum Eintrag "mineur"
sqlTables(channel)
   # Zeigt die Namen der vorhandenen Tabellen an
sqlColumns(channel, "iristab")
   # Zeigt die Spaltennamen und zugehörige Datentypen an
sqlQuery(channel, "select * from iristab")
   # Gibt Tabelle "iristab" vollständig aus.
sqlQuery(channel, "select * from iristab where Species = 'setosa'")
   # Gibt Zeilen der Tabelle "iristab" aus,
   # bei denen die Variable "Species" den Eintrag "setosa" hat.
close(channel)
```

3.2 Datenbanken — RODBC

Die Tabelle ForestCover enthält Daten für die Forstwirtschaft. Es gibt insgesamt 581.012 Beobachtungen und 55 Variablen, also insgesamt 31.955.660 Einträge.

Um diese Daten aus einer ASCII Datei mit den intelligenten **R** Funktionen einzulesen, braucht man eine erhebliche Zeit und etwa 1GB RAM.

Wir sehen uns die Geländesteigungen an, wo Pappeln (CoverType $= 4$) bzw. Espen (CoverType $= 5$) wachsen und eine Straße nicht weiter als 100 Meter entfernt ist.

3.2 Datenbanken — RODBC

```
sqlTables(channel)
sqlColumns(channel, "ForestCover")$COLUMN_NAME
PappelSt <- sqlQuery(channel, "select Slope from ForestCover
                     where CoverType = 4 and HorDistRoad < 100")
EspenSt <- sqlQuery(channel, "select Slope from ForestCover
                     where CoverType = 5 and HorDistRoad < 100")
summary(PappelSt)
summary(EspenSt)
close(channel)          # Verbindung wieder schließen
```

3.2 Excel Daten einlesen

Excel Dateien können über kleine Umwege gelesen werden:

- Die Funktion `read.xlsx()` im Paket *openxlsx*.
- *RODBC*, ein Interface zu Mircosofts Datenbanksprache *ODBC*.
- Export von *Excel* zu `ASCII` (einfach für 1-2 Datensätze).
- Die Funktion `read.xls()` im Paket *gdata* (benötigt Perl).
- Die Funktion `read.xlsx()` im Paket *xlsx* (benötigt Java).

3.2 Excel Daten einlesen — RODBC

Um unter Windows mit Hilfe des Pakets *RODBC* auf Daten in einer Excel (oder Access) Tabelle zuzugreifen, kann man analog vorgehen.
Die Funktionen `odbcConnectExcel()` (für altes Excel, 32-bit R) und `odbcConnectExcel2007()` erledigen das wie folgt:

```
library("RODBC")   # das Paket laden und Verbindung herstellen:
channel <- odbcConnectExcel2007("C:/RKursDaten.xls")
sqlTables(channel) # Zeigt Namen der vorhandenen Tabellen an
sqlColumns(channel, "iris")
sqlQuery(channel, "select PetalLength from \"iris$\"")
# Gibt die Spalte "PetalLength" der Tabelle "iris$" aus.
sqlQuery(channel, "select * from \"iris$\" where
                Species = 'setosa' and SepalLength < 5")
# Gibt diejenigen Zeilen der Tabelle "iris$" aus,
# bei denen die Variable "Species" den Eintrag "setosa" hat und
# der Wert von "SepalLength" kleiner als 5 ist.
close(channel)          # Verbindung wieder schließen
```

3.3 Binärdaten

Kapitel

3 Daten IO

3.3 diverse Datenformate

Natürlich gibt es nicht nur `ASCII`–Dateien und Datenbanken, sondern auch die verschiedensten, z.T. proprietären Binärformate unterschiedlicher Programme und Tools (darunter auch Statistik-Programme).

3.3 Binärdaten

- Man kann Daten in einem platzsparenden Binärformat mit Hilfe der Funktion `save()` speichern. Wir haben schon `save.image()` kennengelernt, womit man nicht nur ein Objekt, sondern gleich den ganzen *Workspace* in einer Binärdatei speichert.
- Das Lesen einer mit `save()` oder `save.image()` gespeicherten Datei geschieht mit `load()`.
- Für Binärformate einiger Statistik-Programme gibt es, zumindest für den Import, Funktionen (Filter) im package *foreign*, nämlich für **Epi, Minitab, SAS Xport, S–PLUS, SPSS, Stata, ...**.
- Für einige andere Binärdateien gibt es auch Importfilter. Eine Suche auf *CRAN* und im WWW hilft manchmal weiter.

3.3 Connections

Es gibt auch Binärdaten–Formate, für die kein Filter erhältlich ist. Man kann recht leicht eigene Import-Funktionen schreiben, indem man *Connections* (vgl. `?connections`) benutzt.

Bei *Connections* handelt es sich um Verbindungen zum Lesen bzw. Schreiben von Informationen zu einem Gerät:

`file()`	Datei auf dem lokalen Rechner oder im lokalen Netzwerk
`pipe()`	Direkte (streaming) Eingabe aus einem Programm
`fifo()`	First In First Out, Ein- / Ausgabe aus einem anderen laufenden Prozess (nicht unter Windows)
`url()`	Zugriff auf Daten im Internet, z.B. `http`, `ftp`, ...
`gzfile()`, `bzfile()`	Direktes Lesen und Schreiben komprimierter Daten
`socketConnection()`	Allgemeinste Zugriff auf ein „Gerät“ im Sinne des Betriebssystems.

3.3 Connections

Das wirkliche Lesen und Schreiben der Daten aus bzw. zu *Connections* geschieht dann mit

- `readBin()` und `writeBin()` für den komfortablen Zugriff einzelner Bytes einer Binärdatei und
- `readChar()` und `writeChar()` für den Zugriff auf Character in Binärdateien.

Eine interessante Möglichkeit Connections zu nutzen ist, eine Verbindung zur Zwischenablage herzustellen, z.B. zum Einlesen (bzw. Schreiben) kleiner Datensätze aus dem E-Mail Programm oder aus Excel. Die Funktion `file()` unterstützt eine entsprechende `"clipboard"` Connection (bzw. `"X11_clipboard"`).

3.3 Connections

Beispiele:

```
read.table(file("clipboard"), header = TRUE)
x <- scan(file("clipboard"))
write(x, file = file("clipboard", open = "w"), ncol = 2)
```

3.3 Objekte (..., Funktionen!) speichern und laden

Eine weitere und sehr wichtige Möglichkeit in **R** Objekte zu speichern ist `dump()`. Es wird nämlich eine Text–Repräsentation des jeweiligen Objekts in eine Datei gespeichert, die wieder in **R** mit `source()` eingelesen werden kann.

Dieses Vorgehen eignet sich dazu

- Objekte zwischen **S–PLUS** und **R** untereinander oder auf verschiedenen Betriebssystemen auszutauschen (unabhängig von Binärdarstellungen) und
- Objekte außerhalb von **R** zu editieren.

Letzterer Punkt findet insbesondere bei Funktionen Verwendung, da es sehr lästig ist, längere Funktionen in der Kommandozeile zu bearbeiten, und daher i.A. auf externe Editoren zurückgegriffen wird.
Hier kommt `source()` zum Wiedereinladen der Funktion zum Einsatz.

3.4 Datum und Zeit

Kapitel

3 Daten IO

3.4 Datum und Zeit

Oftmals möchte und muss man mit Datum oder Zeitangaben rechnen. Beispiele sind:

- Anmeldezeiten von Studierenden in einer Vorlesung
- Alter als Differenz aus aktuellem Datum und Geburtsdatum
- Abstand zwischen Temperaturmessungen
- Überlebensdauer von Bauteilen oder Menschen

3.4 Datum und Zeit konvertieren

Beim Importieren von Daten werden Zeitformate aus Datenbanksysteme meist richtig übernommen. Kommen die Daten aber aus anderen Datenquellen, so muss man meist Zeichenketten, die Datum und Zeit repräsentieren, in ein Zeitobjekt umwandeln.

Es gibt drei Arten von Zeitobjekten:

Objekt	Konvertierung	Sinn
`Date`	`as.Date()`	Rechnen mit Datumsangaben
`POSIXlt`	`strptime()`	Rechnen mit Datum, Uhrzeit, inkl. Zeitzonen und Sommer-/Winterzeit
`POSIXct`	`as.POSIXct()`	konvertiert nach `POSIXct`, das besser in data.frames funktioniert als `POSIXlt`

Solange man nur ein Datum ohne Uhrzeit benötigt, kann man mit `as.Date()` arbeiten.

3.4 Datum und Zeit konvertieren

Es werden dazu Formalismen benötigt, die sagen, an welcher Stelle einer Zeichenkette z.B. die Information des Tages, des Monats oder der Minute stehen. Es wird symbolisiert (siehe `?strptime`):

- %b abgekürzter Monatsname in aktueller Ländereinstellung
- %d Tag (01–31)
- %H Stunde (00–23)
- %I Stunde (01–12)
- %m Monat (01–12)
- %M Minute (00–59)
- %p AM/PM (mit %I verwendet)
- %S Sekunde (00–61) (wegen Schaltsekunden nicht bis nur 59)
- %y Jahr ohne Jahrhundert (00–99; 00–68 bekommen eine 20 davor, sonst eine 19)
- %Y Jahreszahl mit Jahrhundert (z.B. 2019)

3.4 Datum und Zeit konvertieren

Beispiele:

```
R_1.0.0 <- "Feb 29, 2000"
R_1.0.0Date <- as.Date(R_1.0.0, "%b %d, %Y")

R_3.6.1 <- "2019-07-05"
R_3.6.1Date <- as.Date(R_3.6.1, "%Y-%m-%d")

Vorlesung <- "11.10.19, 10:15 Uhr"
VorlesungPOSIXlt <- strptime(Vorlesung, "%d.%m.%y, %H:%M")
```

3.4 Mit Datum und Zeit rechnen

Wichtige Funktionen zum Rechnen mit Datums- und Zeitdaten:

Funktion	Bedeutung
`+, -, ...`	Rechnen mit Zeiten
`difftime()`	Zeitdifferenz in verschiedenen Einheiten berechnen
`format()`	Formatierte Ausgabe von Daten, mit einer Formatspezifikation wie auf der vorigen Folie
`weekdays()`	Wochentage zu Daten
`Sys.time()`	Das aktuelle Datum

3.4 Mit Datum und Zeit rechnen

Beispiele:

```
R_3.6.1Date - R_1.0.0Date
Sys.time()
difftime(Sys.time(), VorlesungPOSIXlt, unit = "week")

format(VorlesungPOSIXlt, "%d.%m.%Y %H:%M")
format(VorlesungPOSIXlt, "%d. %b '%y")
weekdays(VorlesungPOSIXlt)

as.POSIXct(VorlesungPOSIXlt)
## 6 Woche x 7 Tage x 24 Stunden x 60 Minuten x 60 Sekunden
## ab Beginn der 1. Vorlesung:
VorlesungPOSIXlt + 6 * 7 * 24 * 60 * 60
```

4.1 Grafik

Kapitel

4 Grafik
4.1 Einführung

S, an Interactive Environment for Data Analysis and Graphics
R: A Language for Data Analysis and Graphics,
so die Titel des Buches von R. Becker und J. Chambers, bzw. des Artikels von R. Gentleman und R. Ihaka.

Demnach liegt eine der besonderen Stärken der **S** language, und insbesondere von **R**, im Grafikbereich.

Zu unterscheiden gilt es die „konventionellen“ Grafikfunktionen und die in Cleveland (1993) eingeführten „Trellis“ Grafiken (in **R**: „Lattice“ im gleichnamigen *package*). Siehe auch Murrell (2005).

4.1 Grafik

Beginnen wir mit den „konventionellen“ Grafikfunktionen:

- Wer eine Grafik macht, kann diese mit vielen Software–Produkten (z.B. **S–PLUS, Excel**) „Zusammenklicken“.
- Wer viele Grafiken desselben Typs produzieren muss, möchte zur Automatisierung die Grafik programmieren können.

Jetzt ein wenig mehr Struktur:

- Einfache explorative Grafiken können sehr leicht erzeugt werden.
- Hochwertige Grafiken für Publikationen und Präsentationen können erstellt werden.

4.2 Grafik: Devices

Kapitel

4 Grafik

4.2 Devices

Die Liste der möglichen *Devices* umfasst u.a. Bildschirmgrafiken für das interaktive Arbeiten:

- *windows*, *x11* (Unix), *quartz* (Mac).

4.2 Grafik: Devices

Vektor-Grafik Formate:

- *postscript*: PostScript Grafiken, z.B. zum Einbinden in LaTeXDokumente,
- *pdf*: Adobe Portable Document Format, z.B. zur Präsentation, zum Verteilen, zum Einbinden in PDFLaTeX Dokumente,
- *svg*: Scalable Vector Graphics (modernes Vektor-Grafik Format),
- *win.metafile*: Windows meta–file, damit es auch unter Word klappt.

Bitmap-Grafik Formate:

- *png*: PNG Grafiken, ähnlich GIF,
- *jpeg*: JPEG kennt jeder: Qualitativ schlecht, aber enorm kleine Dateigröße – gut für das Internet,
- *bmp*: Bitmap — kann jeder lesen, aber enorm groß und nicht skalierbar.

4.2 Grafik: Devices

Vorteile von Vektor-Grafik Formaten:

- Objekte (Punkte, Linien, Text, . . .) werden einzeln kodiert.
- Grafiken sind skalierbar ohne Qualitätsverlust (werden nicht pixelig).
- Meist sehr kleine Dateigröße, außer man zeichnet extrem viele einzelne Objekte (z.B. Datenpunkte).
- Oft sind die Grafiken nachträglich editierbar.

Vorteile von Bitmap-Grafik Formaten:

- Sehr einfach in andere Software importierbar.
- Konstante Größe bei gegebener Auflösung, das wird aber nur bei extrem vielen Datenpunkten interessant.

Insgesamt überwiegen i.d.R. die Vorteile der Vektor-Grafik Formate.

4.2 Grafik: Devices und Funktionsarten

Wenn man einfach mit der Grafikerzeugung im interaktiven Modus beginnt, so wird als Device automatisch die Bildschirmgrafik gestartet. Im nicht interaktiven Modus ist es das pdf Device.

Die übliche Reihenfolge zum Produzieren einer Grafik ist:

- *Device* starten, z.B. `postscript("c:/testgrafik.ps")`
- Grafik erzeugen, z.B. `plot(1:10)`
- *Device* schließen mit `dev.off()`

Abgesehen von *Trellis (Lattice)* Grafikfunktionen gibt es zwei Arten:

- *High-level* Funktionen erzeugen bzw. initialisieren eine Grafik.
- *Low-level* Funktionen fügen Elemente zu einer (per *High-level* Funktion) erzeugten Grafik hinzu.

4.3 Grafik: High-level

Kapitel

4 Grafik
4.3 High-level Grafik

Einige wichtige und nützliche *High-level* Grafikfunktionen:

`plot()`	kontextabhängig	`hist()`	Histogramm
`barplot()`	Stabdiagramm	`image()`	Bilder (3. Dim. als Farbe)
`boxplot()`	Boxplot	`mosaicplot()`	Mosaikplots (kateg. Daten)
`coplot ()`	Conditioning Plots	`pairs()`	Streudiagramm–Matrix
`curve()`	Funktionen plotten	`persp()`	perspektivische Flächen
`dotchart()`	Dot Plots (Cleveland)	`qqplot()`	QQ–Plot

4.3 Grafik: High-level

Sehr viele Funktionen können auf verschiedene Objekte angewendet werden und reagieren „intelligent“, so dass für jedes Objekt eine sinnvolle Grafik erzeugt wird.

`plot()` erzeugt z.B. einen Scatterplot bei Eingabe von zwei Vektoren, Residualplots bei Eingabe eines *lm*-Objektes (lineares Modell), eine per Linie verbundene Zeitreihe bei Zeitreihenobjekten, etc.
Hier wird die Objektorientiertheit von **R** ausgenutzt (später mehr dazu).

4.3 Grafik: High-level

Es ist also schnell mal ein Streudiagramm, Histogramm, Boxplot etc. erstellt, so dass man sich die Daten sehr schnell anschauen kann.

Beispiele:

```
data(trees)                              # Lade Beispieldatensatz
plot(trees)                              # Malt Streudiagramm-Matrix

plot(trees$Girth, trees$Volume)  # ein bestimmtes Streudiagramm
tree.lm <- lm(trees$Volume ~ trees$Girth)   # Regression
abline(tree.lm)                          # Regressionsgerade
plot(tree.lm)                            # Plots zur Residualanalyse

boxplot(trees)                           # Boxplots
hist(trees$Volume)                       # Histogramm einer Variablen
qqnorm(trees$Volume)                     # Normalverteilung ???
```

4.4 Grafik: Parameter

Kapitel

4 Grafik

4.4 Parameter

Aber natürlich reicht es nicht immer, die Daten schnell anzuschauen, sondern die Grafiken müssen angepasst, d.h. z.B. mit Überschriften versehen werden.

4.4 Grafik: Parameter

Zunächst einmal gibt es eine Vielzahl von Parametern, die man an die meisten Grafikfunktionen in Form von Argumenten übergeben kann. Dazu gehören vor allem diejenigen, die in den Hilfen `?plot` und `?plot.default` aufgeführt sind, aber zum größten Teil auch die in `?par`.

Mit der Funktion `par()` werden die wichtigsten Voreinstellungen im Grafikbereich durchgeführt. Die Hilfe dieser mächtigen Funktion sollte man sich wirklich ansehen.

Insgesamt empfiehlt es sich, einmalige Einstellungen in der Grafikfunktion anzugeben, eine Änderung für mehrere Grafiken aber mit `par()` vorzunehmen. Viele Einstellungen ändert man ausschließlich in `par()`.

4.4 Grafik: Parameter

Zu den am häufigsten gebrauchten Argumenten gehören:

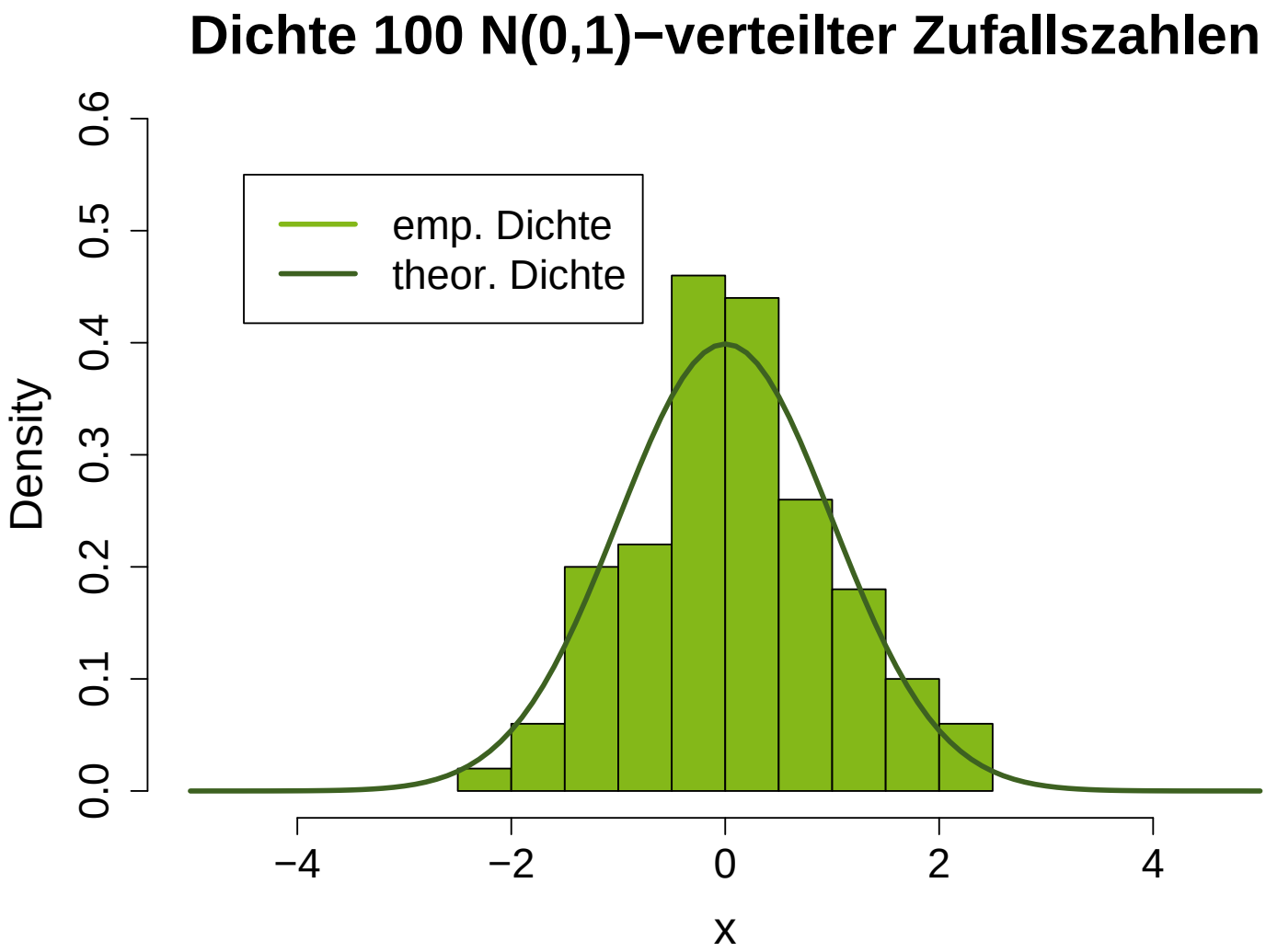
`type`	Type (l=Linie, p=Punkt, b=Beides, n=Nichts)
`xlim, ylim`	Zu plottender Bereich in x-/y- Richtung
`log`	Logarithmierte Darstellung
`main, sub`	Überschrift und „Unterschrift“
`xlab, ylab`	x-/y-Achsenbeschriftung
`axes`	Sollen Achsen geplottet werden
`col`	Farben
`bg`	Hintergrundfarbe
`pch`	Symbol für einen Punkt
`cex`	Größe eines Punktes bzw. Buchstaben
`lty, lwd`	Linientyp (gestrichelt, ...) und Linienbreite
`mfrow`	Mehrere Plots in ein *Device* in Zeilen und Spalten

4.4 Grafik: Parameter

Beispiele:

```
greend <- rgb(0.24,0.38,0.13)
greenh <- rgb(0.477,0.754,0.258)
x <- rnorm(100)              # 100 N(0,1)-verteilte Zufallszahlen
par(bg = "yellow")           # Hintergrund gelb
##### Beschriftetes und manuell skaliertes Histogramm:
hist(x, main = "Dichte 100 N(0,1)-verteilter Zufallszahlen",
  freq = FALSE, col = greenh, xlim = c(-5, 5), ylim = c(0, 0.6))
#### Hinzufügen der theor. Dichtefunktion in blau:
curve(dnorm, from = -5, to = 5, add = TRUE, col = greend, lwd=3)
#### Nur das Histogramm in eine PDF Datei schreiben:
pdf("c:/test.pdf")
hist(x, main = "Dichte 100 N(0,1)-verteilter Zufallszahlen",
  freq = FALSE, col = greenh, xlim = c(-5, 5), ylim = c(0, 0.6))
dev.off()
```

4.4 Grafik: Parameter

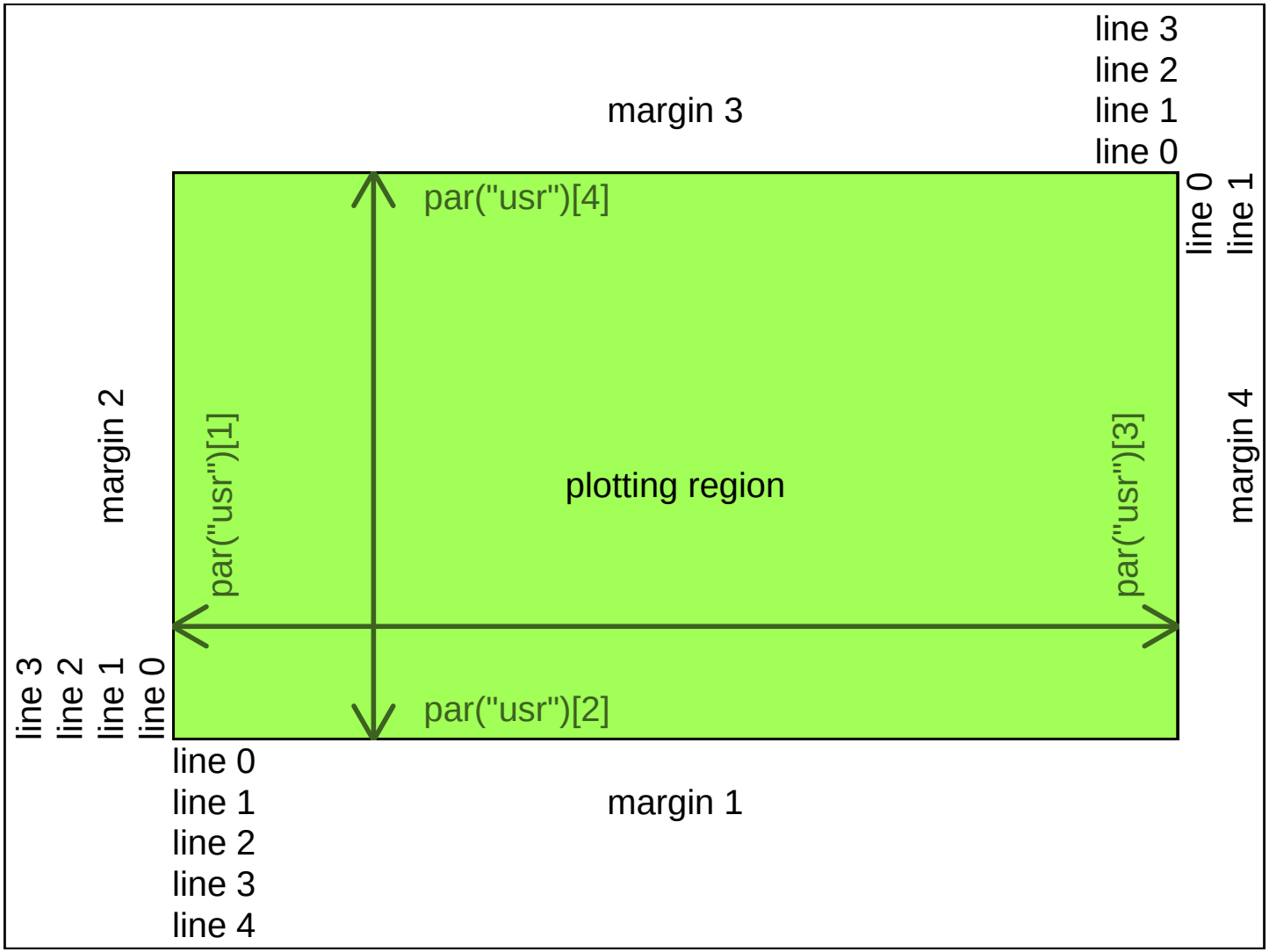


4.4 Grafik: Parameter

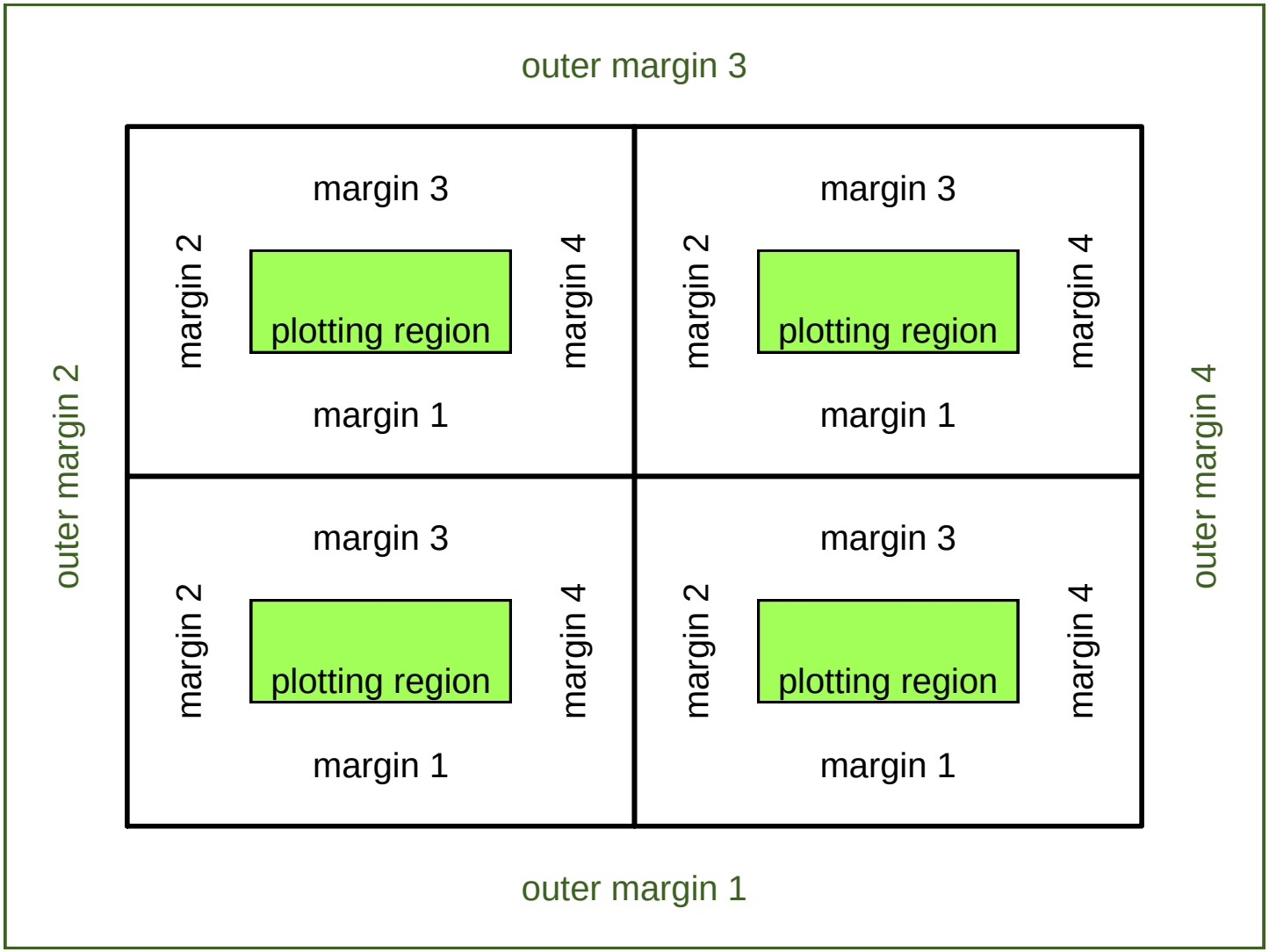
Beispiele:

```
## 4 Plots in ein Device (2 Zeilen, 2 Spalten):
par(mfrow = c(2, 2))

# Grafiken von vorher (mit ein paar geänderten Parametern):
plot(trees$Girth, trees$Volume, pch = 7)
boxplot(trees, col = "blue")
hist(trees$Volume, las = 1)
qqnorm(trees$Volume, cex.axis = 2)

par(mfrow = c(1, 1))
```

4.4 Grafik: Ränder



4.4 Grafik: Ränder



4.5 Grafik: Low-level

Kapitel

4 Grafik
4.5 Low-level Grafik

Da man häufig auch Elemente zu einem Plot hinzufügen will, etwa zusätzliche Punkte, Linien für Konfidenzintervalle, Beschriftungen, Legenden etc., gibt es auch sogenannte *Low-level* Funktionen. Solche Funktionen helfen z.B. bei der Berechnung von geeigneten Achsenbeschriftungen oder können einzelne Elemente zu einer bestehenden Grafik hinzufügen.

4.5 Grafik: Low-level

Eine Auswahl solcher Funktionen:

`abline()`	„intelligente“ Linie	`mtext()`	Text in den Rändern
`arrows()`	Pfeile	`points()`	Punkte
`axis()`	Achsen	`polygon()`	(ausgefüllte) Polygone
`grid()`	Gitternetz	`segments()`	Linien (vektorwertig)
`legend()`	Legende	`text()`	Text
`lines()`	Linien (schrittweise)	`title()`	Beschriftung

4.5 Grafik: Low-level

Beispiele:

```
## Fortsetzung des "Histogramm" Beispiels!
## Eine Legende für das Histogramm:
legend(-4.5, 0.55, legend = c("emp. Dichte", "theor. Dichte"),
   col = c(greenh, greend), lwd = 3)
## oder "interaktiv" mit
legend(locator(1), legend = c("emp. Dichte", "theor. Dichte"),
   col = c(greenh, greend), lwd = 3)

## Beispiel: Eigene Achsen! Erzeuge zunächst Daten:
gruen <- rnorm(10); rot <- rnorm(10, mean = 1)
blau <- rnorm(10, sd = 5)
plot(rep(1:3, each = 10), c(gruen, rot, blau), xlab = "Farben",
   col = rep(c("green", "red", "blue"), each = 10), xaxt = "n")
axis(1, at = 1:3, labels = c("grün", "rot", "blau"))
```

4.6 Grafik: Mathematische Beschriftung

Kapitel

4 Grafik
4.6 Mathematische Beschriftung

R kann Grafiken mit mathematischen Symbolen und math. Notation beschriften (Murrell und Ihaka, 2000), z.B.: $f(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}$.
Gerade für Publikationen und Präsentationen muss so nicht mit diversen Programmen getrickst oder gar darauf verzichtet werden.

Die Eingabe einer Formel erfolgt mit Hilfe eines **R** Ausdrucks (Kontrollwörter sind an LaTeX angelehnt), d.h. man schreibt die Formel einfach in **R**, lässt sie aber nicht auswerten, sondern übergibt sie direkt, z.B. mit `expression()`.

4.6 Grafik: Mathematische Beschriftung

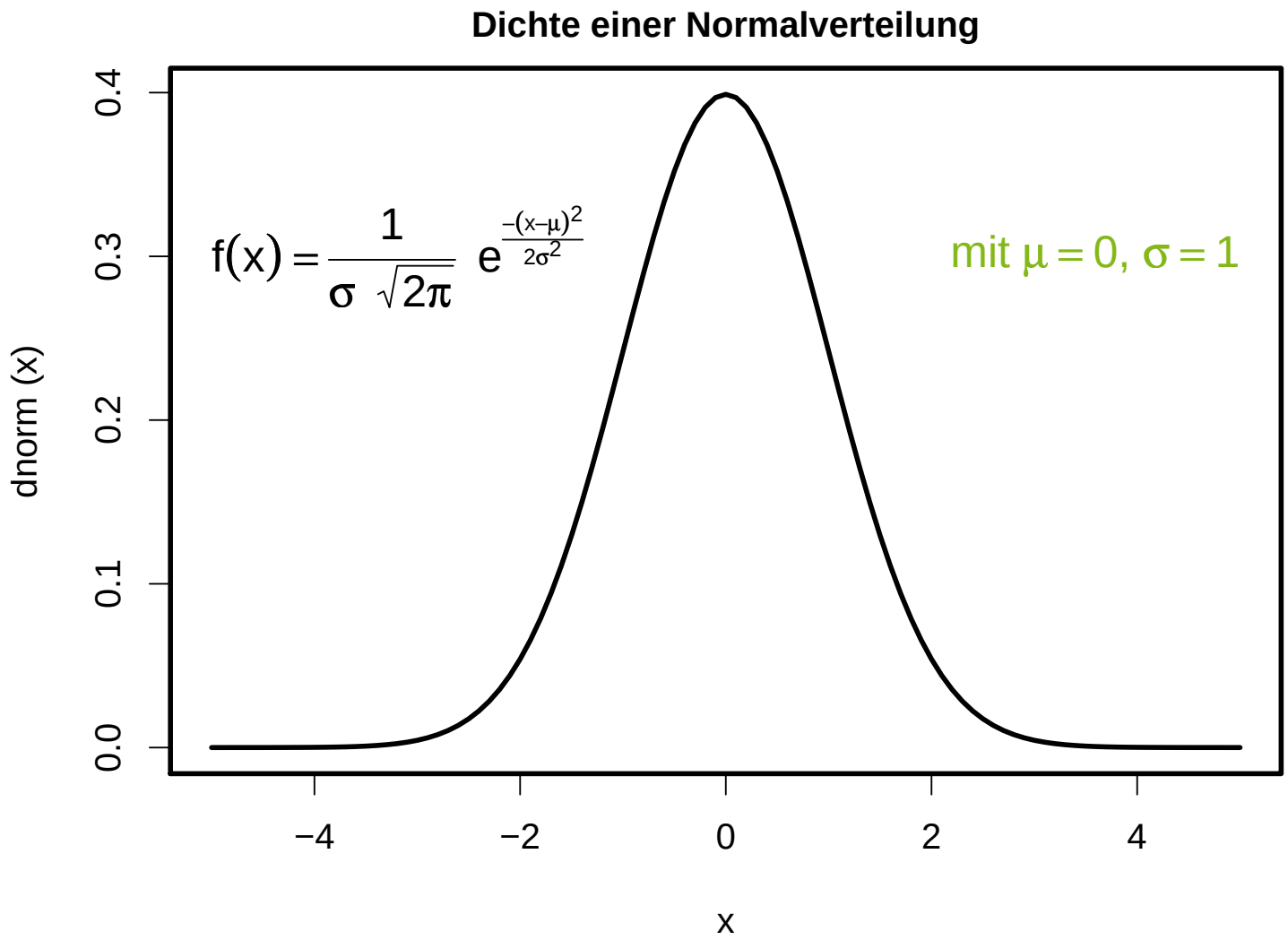
Für Details empfiehlt sich dringend ein Blick auf die Hilfeseite `?plotmath`.

Zusätzlich könnte man sogar noch Variablen einfügen und automatisch ersetzen lassen. Details können im **R** Newsletter 2/3 nachgelesen werden.

Beispiele:

```
## Dichtefunktion einer N(0,1)-Verteilung:
curve(dnorm, from = -5, to = 5, col = greenh, lwd = 3,
    main = "Dichte einer Normalverteilung")
text(-5, 0.3, adj = 0, cex = 1.3,
    expression(f(x) == frac(1, sigma ~~ sqrt(2*pi)) ~~
        e^{frac(-(x - mu)^2, 2 * sigma^2)}))
text(5, 0.3, adj = 1, cex = 1.3, col = greenh,
    expression("mit " * {mu == 0} * ", " * {sigma == 1}))
```

4.6 Grafik: Mathematische Beschriftung



4.7 Grafik: Transparenz

Kapitel

4 Grafik
4.7 Transparenz

Der Einsatz von Transparenz ist nützlich zur Visualisierung großer Datenmengen oder überlagerten Strukturen. Transparenz ist nicht für alle Devices verfügbar.

```
x <- c(rnorm(10000), (a<- rnorm(1000, sd=0.5)))
y <- c(rnorm(10000), a)
plot(x, y, pch=16)
plot(x, y, col = rgb(0, 0, 0, alpha = 0.05), pch = 16)
```

4.8 Grafik: Interaktion

Kapitel

4 Grafik

4.8 Interaktion

R selbst stellt keine wirklich interaktiven Grafiken zur Verfügung. Minimale Interaktion ist allerdings gegeben durch die Funktionen

- `locator()`, die ausgibt, wohin man in einem Plot klickt, und
- `identify()` zur Ausgabe des Indexes derjenigen Datenpunkte, in deren Nähe man klickt, zum **Beispiel**:

```
plot(trees$Girth, trees$Volume)
identify(trees$Girth, trees$Volume)
```

4.8 Grafik: Interaktion mit rgl

Auswege sind (Beispiele werden vorgeführt):

- Eine Kommunikationsmöglichkeit mit *ggobi* (ein interaktives Visualisierungssystem),
- das Paket *rgl* von Daniel Adler,
- das *iPlots* Paket von Simon Urbanek.

4.8 Grafik: Interaktion mit rgl

Beispiele:

```
library("rgl")
plot3d(iris, size = 3, col = as.numeric(iris$Species))

x <- y <- seq(-10, 10, length= 30)
f <- function(x,y) {r <- sqrt(x^2+y^2); 10 * sin(r)/r }
z <- outer(x, y, f)

rgl.clear()
persp3d(x, y, z, color = "red", zlim = c(-1.2, 12),
        back = "fill", front = "fill", alpha = 0.5)
persp3d(x, y, z+2, color = greend, zlim = c(-1.2, 12),
        add = TRUE, back = "fill", front = "fill", alpha = 0.5)
```

4.8 Grafik: Interaktion mit iplots

Beispiele:

```
## iplot
library("iplots")
attach(trees)
ihist(Girth)
ihist(Height)
ihist(Volume)
iplot(Girth, Volume)
mylm <- lm(Volume ~ Girth)
iabline(mylm)
iplot(Height, Volume)
detach(trees)
```

5.1 Funktionen

Kapitel

5 Funktionen
5.1 Einführung

Bisher sind uns schon einige Funktionen begegnet, darunter einfache wie `sin()`, aber auch komplexere wie `read.table()`, die verschiedene Argumente akzeptieren.

- Jegliches Arbeiten geschieht mit Funktionen!

5.1 Funktionen

- Ein Funktionsaufruf hat die Form

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

  dabei kann die Benennung der Argumente u.U. weggelassen werden.
  - Es gibt spezielle Funktionen mit Kurzformen, z.B. +.

    Der Ausdruck `3 + 5` würde in voller Form lauten: `"+"(3, 5)`.

    Der Name steht hier in Anführungszeichen, da es kein regulärer Name ist (solche beginnen mit Buchstaben!).
  - Auch die Zuweisung ist eine Funktion: `"<-"(x, 3)`.
- Es gibt Argumente, die Voreinstellungen (defaults) haben.
  - Ein Argument ohne default *muss* beim Funktionsaufruf angegeben werden.
  - Ein Argument mit Voreinstellung *kann* beim Funktionsaufruf geändert werden.
  - Funktionsaufrufe müssen nicht immer Argumente besitzen, wie z.B. `ls()`.

5.1 Funktionen

Eigene Funktionen sind immer dann sinnvoll, wenn eine Folge von anderen Funktionsaufrufen (unter einem Namen) zusammengefasst werden soll, z.B. für mehrmaliges Ausführen mit verschiedenen Parametern.

Die allgemeine Form einer Funktionsdefinition ist:

`Meine.Funktion <- function(Argumente){ Befehlsfolge }`,

wobei die `Argumente` mit oder ohne Voreinstellung angegeben werden. Beim Aufruf der Funktion werden die `Argumente` an die `Befehlsfolge` weitergereicht.

Nicht nur bei der Definition von Funktionen, sondern auch bei allen anderen Konstruktionen (`for()`, `if()`), können Befehlsfolgen, solange sie in geschweiften Klammern stehen, aus mehreren Zeilen bestehen.

5.1 Funktionen

Eine typische Definition einer Funktion könnte wie folgt aussehen:

```
median <- function(x, na.rm = FALSE)
{
     # ... viel code! ...
    sort(x, partial = half)[half])
}
```

- Diese Funktion hat zwei Argumente: `x, na.rm`.
- Nur das zweite hat einen default, nämlich `FALSE`.
- Die letzte Zeile der Funktion gibt den Wert der Funktion an. Sollen mehrere Werte (als Liste) zurückgegeben werden, benutzt man `return()`.

5.1 Funktionen

- Sei `a` ein Vektor, dann sind z.B. folgende Aufrufe sinnvoll:
  - `median(a)` (`na.rm` muss nicht angegeben werden – default)
  - `median(a, TRUE)`
    (In richtiger Reihenfolge müssen Argumente nicht benannt werden.)
  - `median(na.rm = TRUE, x = a)`
    (Benannte Argumente dürfen in beliebiger Reihenfolge stehen.)

5.1 Funktionen

Man muss also zwischen den *formal* in der Funktion definierten Argumenten und den *tatsächlich* beim Funktionsaufruf gegebenen Argumenten unterscheiden. Die Regeln dazu werden in der folgenden Reihenfolge angewandt (Beispiel `median()`):

- Alle Argumente mit vollständigem Namen werden zugeordnet (`x = 1:10`).
- Argumente mit teilweise passendem Namen werden den übrigen *formalen* Argumenten zugeordnet (`na = TRUE`).
- Alle unbenannten Argumente werden der Reihe nach den übrigen *formalen* Argumenten zugeordnet.
- Übrige Argumente werden dem evtl. vorhandenen formalen Argument `...` zugeordnet (s.u.).

Auf das Fehlen eines formalen Arguments kann man innerhalb einer Funktion mit `missing()` testen.

5.1 Funktionen

Beispiele:

```
a <- c(1, NA, 5, 3)

## zunächst unbenannte Argumente:
median(a)
median(a, TRUE)
median(TRUE, a)                    # das macht keinen Sinn!

## nun zumindest teilweise benannte Argumente:
median(x = a, na.rm = TRUE)
median(na.rm = TRUE, a)
median(n = TRUE, a)
```

5.1 Funktionen

In der Definition einer Funktion ist der Einsatz des formalen „Drei–Punkte Arguments“ `...` möglich. Alle nicht zugeordneten *tatsächlich* gegebenen Argumente werden durch `...` aufgenommen und können innerhalb der Funktion weiterverarbeitet oder an andere Funktionen durch den Aufruf mittels `...` weitergegeben werden.
Es dient also meist zum Durchreichen von Argumenten.

Beispiele:

```
Punkte <- function(x, ...){
    x <- x - 2
    median(x, ...)
}
x <- log(-1:100)
Punkte(x)
Punkte(x, na.rm = TRUE)
```

6.1 Bedingte Anweisungen

Kapitel

6 Konstrukte
6.1 Bedingte Anweisungen

In vielen Fällen wird bei der Programmierung eine Fallunterscheidung benötigt, etwa

- zur Überprüfung von Argumenten und zur Ausgabe von Fehlermeldungen,
- zum Abbruch eines iterativen Verfahrens bei Konvergenz und
- zum Programmieren einer mathematisch / methodischen begründeten Fallunterscheidung.

6.1 Bedingte Anweisungen

Für bedingte Anweisungen gibt es die Konstrukte

(1) `if(Bedingung){Ausdruck1} else{Ausdruck2}`

(2) `ifelse(Bedingung, Vektor1, Vektor2)`.

Mit (1) steht eine bedingte Anweisung für recht komplexe Ausdrücke zur Verfügung, wobei die *Bedingung* aber nicht vektorwertig sein darf, oder vielmehr nur das erste Element im Fall einer vektorwertigen *Bedingung* verwendet wird.
Der Teil `else{Ausdruck2}` darf auch weggelassen werden.
Zunächst wird die *Bedingung* ausgewertet. Ist diese wahr, so wird *Ausdruck1* ausgewertet, sonst *Ausdruck2*.

Funktion (2) zeichnet sich durch vektorwertige Operationen aus.
Je nach logischem Wert der vektorwertigen *Bedingung* wird das zugehörige Element aus *Vektor1* (bei wahr) oder *Vektor2* (bei unwahr) gewählt.

6.1 Bedingte Anweisungen

Beispiele:

```
x <- 5
if(x == 5){           # falls x = 5 ist:
    x <- x + 1        # x um eins erhöhen und
    y <- 3            # y auf drei setzen
} else                # sonst:
    y <- 7            # y auf sieben setzen

if(x < 99) cat("x ist kleiner als 99\n")

ifelse(x == c(5, 6), c("A1", "A2"), c("A3", "A4"))

# Absolutbetrag ausrechnen:
x <- c(-2:2)
ifelse(x < 0, -x, x)
```

6.2 Schleifen

Kapitel

6 Konstrukte
6.2 Schleifen

Schleifen sind unverzichtbar, um eine größere Anzahl sich wiederholender Befehle (mit unterschiedlichem Parameter) aufzurufen. Etwa bei Simulationen kommen solche Schleifen sehr häufig zum Einsatz, weil beispielsweise gewisse Befehlsfolgen und Funktionen immer wieder mit unterschiedlichen Zufallszahlen gestartet werden müssen.

6.2 Schleifen

Die einfachste Form einer „Schleife“ ist `replicate()`.

Das folgende Beispiel veranschaulicht den zentralen Grenzwertsatz. 1000 mal wird der Mittelwert von 100 rechteckverteilten Zufallszahlen berechnet:

```
zahlen <- replicate(1000, mean(runif(100)))
hist(zahlen)  # Histogramm der erzeugten Mittelwerte
```

6.2 Schleifen

Es gibt drei weitere Varianten von Schleifen, sowie zwei wesentliche Kontrollbefehle:

```
(1)  repeat{ Befehle }
(2)  while(Bedingung){ Befehle }
(3)  for(Wert in Objekt){ Befehle }
```

(i) `break`: Der Befehl `break` beendet die Schleife vollständig.
(ii) `next`: Mit den Befehl `next` wird innerhalb einer Schleife, ohne die folgenden Befehle auszuführen, in den nächsten Durchlauf gesprungen.

6.2 Schleifen

Zu (1): Mit `repeat{ Befehle }` werden die `Befehle` immer wieder wiederholt. Die Schleife kann nur mit `break` beendet werden, das meist in einer bedingten Anweisung (`if()`) auftaucht.

Zu (2): Mit `while(Bedingung){ Befehle }` werden die `Befehle` solange wiederholt, wie die `Bedingung` erfüllt ist.

Zu (3): Mit der Schleife `for(Wert in Objekt){ Befehle }` wird von `Wert` zunächst der erste Wert von `Objekt` angenommen und alle `Befehle` mit diesem Wert ausgeführt. Danach werden alle `Befehle` ausgeführt, wobei `Wert` den zweiten Wert des `Objekt` annimmt usw. `Wert` muss nicht notwendigerweise in der Befehlsfolge benutzt werden.

6.2 Schleifen

Da Schleifen in **R** *langsam* sind, sollten sie möglichst vermieden werden, vor allem dann, wenn stattdessen vektorwertig gearbeitet werden kann. Grund: Alle Funktionen müssen innerhalb einer Schleife mehrfach statt einfach ausgewertet werden!

6.2 Schleifen

Beispiele (Schlechter Stil, nur zur Demonstration!!!):

```
i <- 0
repeat{
    i <- i + 1          # Addiere 1 zu i.
    if(i == 3) break    # Stoppe, falls i=3 ist.
}

while(i > 1) {
    i <- i - 1          # Solange i > 1 ist, erniedrige i um 1.
}
```

6.2 Schleifen

```
x <- c(3, 6, 4, 8, 0)    # Vektor der Länge 5 (=length(x))
for(i in x)
    print(sqrt(i))

for(i in seq(along=x))       # dasselbe übr Indizes
    print(sqrt(x[i]))

for(i in seq(along = x)){ # Für alle i im Vektor seq(along=x)
    x[i] <- x[i]^2        # Quadriere das i-te Element von x
    print(x[i])           # und gib es auf dem Bildschirm aus.
}

x^2                       # BESSER (schneller, übersichtlicher)!
```

6.2 Schleifen

Beispiele (Schlechter Stil, nur zur Demonstration!!!):

```
x <- NULL
for(i in 1:length(x))      # Hier kann was schief gehen!
    print(x[i])

for(i in seq(along = x))   # So ist es sicherer!
    print(x[i])
```

6.3 apply!

Kapitel

6 Konstrukte

6.3 apply!

Wir wissen, dass vektorwertiges Arbeiten in **R** meist *schneller* ist als die Benutzung von Schleifen. Auf Vektoren und Matrizen ist nach kurzem Überlegen oft das Vorgehen leicht ersichtlich, etwa durch

- Vektor– oder Matrixmultiplikation (%*%),
- äußeres Produkt (`outer()` oder %o%),
- Kronecker–Produkt (%x%) oder
- die üblichen komponentenweisen Rechenoperationen.

Es soll aber auch anders quasi-vektorwertiges Arbeiten möglich sein, nämlich mit `apply()`, `lapply()`, `sapply()` und `tapply()`.

6.3 apply()

Eine wichtige Funktion für vektorwertiges Arbeiten auf Matrizen bzw. Arrays ist die Funktion `apply()`. Mit ihr können geeignete Funktionen auf jede Spalte oder Zeile einer Matrix angewendet werden, ohne dass komplizierte und langsame Schleifen eingesetzt werden müssen. Sie hat die Form:

`apply(X, margin, function)` , wobei

- `X` das Array (Matrix),
- `margin` die beizubehaltende Dimensionsnummer und
- `function` die komponentenweise (in ihrem ersten Argument) anzuwendende Funktion ist.

6.3 apply()

Beispiele:

```
X <- matrix(c(4, 7, 3, 8, 9, 2), nrow = 3)

# Zeilenmaxima berechnen:
erg <- numeric(nrow(X))
for(i in 1:nrow(X)){
    erg[i] <- max(X[i,])
}

# oder einfacher:
apply(X, 1, max)

apply(X, 2, range)     # Spalten-Extrema
```

6.3 apply()

Beispiele:

```
# 4 Wege, wie man die Zeilenquadratsumme berechnen kann:
QS <- function(x){
 sum(x^2)
}
apply(X, 1, QS)

apply(X, 1, function(x) sum(x^2))

rowSums(X^2)

diag(X %*% t(X))
```

6.3 lapply(), sapply()

Für Listen, Dataframes und Vektoren lässt sich besser die Funktion `lapply()` verwenden. Mit ihr wird das eingegebene Objekt komponentenweise bearbeitet:

`lapply(object, function, ...)` , wobei weitere Argumente zur Funktion `function` als zusätzliche Argumente angegeben werden können. Das Ergebnis ist eine Liste (ein Listen-Element je Komponente des eingegebenen Objekts).

`sapply()` funktioniert analog zu `lapply()`, versucht aber nach Möglichkeit die Ergebnisse zu einem Vektor bzw. einer Matrix zu vereinfachen.

Beispiele:

```
L <- list(x = 1:10, y = 1:5 + 0i)
lapply(L, mean)      # Liste wird beibehalten, mit Datentyp
sapply(L, mean)      # Vektor wird erzeugt - gleicher Datentyp!
```

6.3 tapply()

`tapply()` wendet Funktionen (typischerweise) auf Elemente eines Vektors an, wobei dieser Vektor jedoch durch weitere Argumente (Faktoren) gruppiert wird.

Durch diese Gruppierung entstehen Tafeln, in die pro Gruppe das Ergebnis der angewandten Funktion eingetragen ist.

Beispiele:

```
data(warpbreaks)
warpbreaks
tapply(warpbreaks$breaks, warpbreaks[, 3], sum)
tapply(warpbreaks$breaks, warpbreaks[,-1], mean)
```

7.0 Zeichenketten

Kapitel

7 Zeichenketten

Ein weiterer Punkt in der Programmierung ist die Verarbeitung von Zeichenketten.

Wie Zeichenketten (mit `cat()` und `print()`) auf die Konsole ausgegeben werden, haben wir bereits gesehen. Manchmal möchte man aber auch Zeichenketten anders verarbeiten, etwa zusammensetzen, zerlegen, darin suchen oder diese für die Ausgabe angemessen formatieren.

7.0 Zeichenketten

Typische Anwendungen von Zeichenkettenoperationen sind:

- Zusammensetzen von Buchstabenkombinationen und Nummern, etwa um eine Liste durchnummerierter Dateien nacheinander automatisch bearbeiten zu können
- Das Zerlegen von Strings in einzelne Teile, die durch Sonderzeichen voneinander getrennt sind
- Suchen, ob unter einer Liste von Zeichenfolgen bestimmte Zeichenkombinationen vorhanden sind
- benutzerfreundliche Ausgabe von Informationen in tabellarischer Form

7.0 Zeichenketten

Eine Zusammenstellung der wichtigsten Funktionen für Operationen auf Zeichenketten:

`paste()`	Zusammensetzen von Zeichenketten
`strsplit()`	Zerlegen von Zeichenketten
`grep()`	sucht Zeichenfolgen in Vektoren
`gsub()`	zum Ersetzen gefundener Teil–Zeichenfolgen
`match()`, `pmatch()` `charmatch()`	Suchen von (Teil)–Zeichenketten in anderen und und Ausgabe der Positionen der Übereinstimmungen
`substring()`	Ausgabe und Ersetzung von Teil–Zeichenfolgen
`nchar()`	Anzahl Zeichen in einer Zeichenkette
`toupper()`, `tolower()`	Umwandlung in Groß- bzw. Kleinbuchstaben
`formatC()`	erlaubt sehr allgemeine Formatierung
`parse()` `deparse()`	Konvertierung einer Zeichenfolge in eine *expression* und umgekehrt

7.0 Zeichenketten

Beispiele:

```
x <- 8.25
cat(paste("Die Variable x hat den Wert", x, "\n"))
x <- "Hermann Müller"; y <- "Hans_Meier"
strsplit(x, " ")                    # Vorname u. Nachname einzeln
strsplit(y, "_")                    # Vorname u. Nachname einzeln
grep("Hans", c(x, y))               # Im 2.
gsub("ü", "ue", c(x, y))            # ü -> ue
nchar(x)
toupper(x)                          # HERMANN MÜLLER
ep <- parse(text = "z <- 5")        # jetzt eine Expression,
eval(ep)                            #  die ausgewertet werden kann
```

7.0 Zeichenketten

In den Funktionen `grep()`, `gsub()` und einigen anderen werden s.g. *regular expressions* verwendet, siehe die Hilfeseite `?regexp`. Solche *regular expressions* sind aus anderen Sprachen wie Perl bekannt und nicht besonders R spezifisch. Sie sind sehr mächtig bei der Text(vor)verarbeitung.

7.0 Zeichenketten

Beispiele:

```
dateiname <- "1135-1947-11-03.txt"
# Extrahieren aus Dateinamen der Form: ID-Jahr-Monat-Tag.txt

id <- gsub("^([[:digit:]]+)-.*", "\\1", dateiname)
datum <- gsub("^[[:digit:]]+-(.*)\\.txt$", "\\1", dateiname)

# oder:
ohneEndung <- strsplit(dateiname, "\\.")[[1]][1]
Teile <- strsplit(ohneEndung, "-")[[1]]
id <- Teile[1]
datum <- paste(Teile[2:4], collapse="-")
```

8.0 Grafik: Trellis / Lattice

Kapitel

8 Lattice Grafik

Hier soll auf die Idee der *Trellis* (Cleveland, 1993) bzw. *lattice* (Sarkar, 2002) Grafiken eingegangen werden. Für das Paket *lattice* hat Deepayan Sarkar 2004 den Chambers' Award der ASA gewonnen hat.

8.0 Grafik: Trellis / Lattice

- Das Paket *lattice* basiert auf dem Grafiksystem im Paket *grid* (Murrell, 2001). Beide Pakete sind in einer Standard R Installation enthalten.
- Grid Grafiken und Standard R Grafiken sind i.A. inkompatibel.
- Das CRAN Paket *gridBase* behebt einige dieser Inkompatibilitäten und ermöglicht die Integration *grid*- und Standard-Grafiken.
- Die Wörter *trellis*, *lattice* und *grid* lassen sich alle mit „Gitter“ übersetzen: Es werden hier mehrere Grafiken gleichen Typs in einem Raster direkt nebeneinander dargestellt, wobei in jedem nur eine Teilgruppe der Daten zu sehen ist, z.B. nach einer anderen (Faktor-) Variable aufgeteilt.

8.0 Grafik: Trellis / Lattice

- Das „einzige“ zur Verfügung stehende *Device* ist `trellis.device()`, welches aber sowohl für Bildschirmdarstellung, als auch externe Grafiken, z.B. in Form von PostScript oder PDF, benutzt werden kann:

  ```
  trellis.device(device = pdf, file = "Rtestgrafik2.pdf")
  ```

  Es handelt sich um ein Meta Device, das neben einigen Konfigurationen intern die entsprechenden anderen bekannten Devices startet.

- Anders als bei den Standard–Grafiken wird vor der Grafikerstellung ein Objekt der Klasse *trellis* generiert und bearbeitet. Erst, wenn alle Elemente enthalten sind, wird es mit `print()` (bzw. der entsprechenden Methode `print.trellis()`) gezeichnet.
  Gerade im nicht-interaktiven Modus muss man unbedingt an das `print()` denken, da sonst keine Grafik erzeugt wird.

8.0 Grafik: Trellis / Lattice

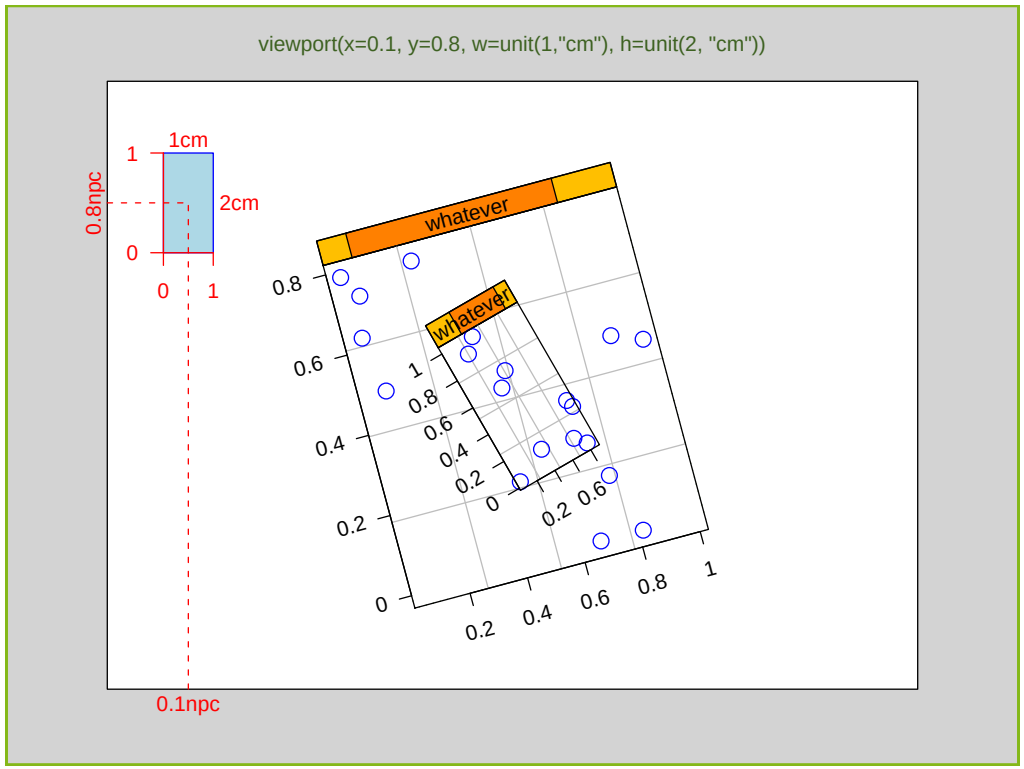
- Das Interface zu *Trellis* Grafikfunktionen ist so angelegt, dass die Daten als Formel (mit „~“) eingegeben werden, um die Abhängigkeitsstruktur in den Daten für die Anordnung der nebeneinanderliegenden Plots ausdrücken zu können.
- Die Trellis-Implementation in `lattice` geht über die ursprüngliche Trellis-Implementation in S hinaus. Neben dem großen Engagement des Autors liegt dies vor allem auch daran, dass in R mit Paul Murrell's `grid` Paket eine enorm mächtige Grundlage für Lattice zur Verfügung gestellt wurde:

8.0 Grafik: Trellis / Lattice

- Es ist ein Grundproblem der Standard-Grafik *allgemein* Grafiken innerhalb einer Grafik zu realisieren: `layout()` oder das simple `par(mfrow=.)` können nicht geschachtelt, d.h. mehrstufig verwendet werden, so dass z.B. unmöglich ist, mehrere `filled.contour()`, `coplot()` oder `pairs()` Grafiken zu einer neuen Grafik zusammenzustellen.
- Auch die scheinbar sehr flexible Verwendung von `par()` stößt an Grenzen oder wird hässlich, nicht umsonst enthält der Source-code von `par.c` den Kommentar
  „The horror, the horror ...“, Marlon Brando in Apocalypse Now

8.0 Grafik: Einschub zu grid

Grid selber arbeitet mit GROBs, **Gr**aphical **Ob**jects, *viewports* und *units* und ermöglicht damit ein klares und extrem flexibles System von Grafiken in diversen Koordinatensystemen, daher z.B. auch editierbare grafische Elemente, oder im Prinzip beliebige *Grafiken innerhalb einer Grafik*.



8.0 Grafik: Einschub zu grid

Beispiele:

```
library("grid")
vp <- viewport(x = 0.1, y = 0.8,
               w = unit(1, "cm"), h = unit(2, "cm"))
grid.show.viewport(vp)
grid.rect(gp = gpar(col = "dark blue", lwd=3))
grid.text('viewport(x = 0.1, y = 0.8, w = unit(1, "cm"),
  h = unit(2, "cm"))', y = 0.95, gp = gpar(col = "orange3"))
pushViewport(vp.schraeg <- viewport(h=0.5, w=0.3, angle=15))
grid.rect(gp = gpar(col = "lightblue", lty = 3))
grid.panel()
grid.panel(vp = vp.schraeg)
popViewport()
```

8.0 Grafik: Trellis / Lattice

Hier eine Liste von häufig verwendeten *Trellis* Funktionen:

`barchart()`	Balkendiagramm
`bwplot()`	Boxplot
`cloud()`	3D Punktewolken
`densityplot()`	Dichten
`dotplot()`	Punkteplots
`histogram()`	Histogramm
`levelplot()`	Levelplots
`panel.....()`	Funktionen zum Hinzufügen von Elementen
`piechart()`	Kuchendiagramm
`print.trellis()`	Trellis Objekt plotten
`qq()`	QQ–Plots
`wireframe()`	persp. 3D Flächen
`xyplot()`	Scatterplot

8.0 Grafik: Trellis / Lattice

Die folgenden Beispiele stammen von den *lattice* Hilfeseiten. Sie zeigen direkt, wie mit Lattice Grafiken und dem Formelinterface umgegangen werden kann.

Beispiele:

```
library("lattice")
bwplot(~ breaks | wool + tension, data = warpbreaks)
```

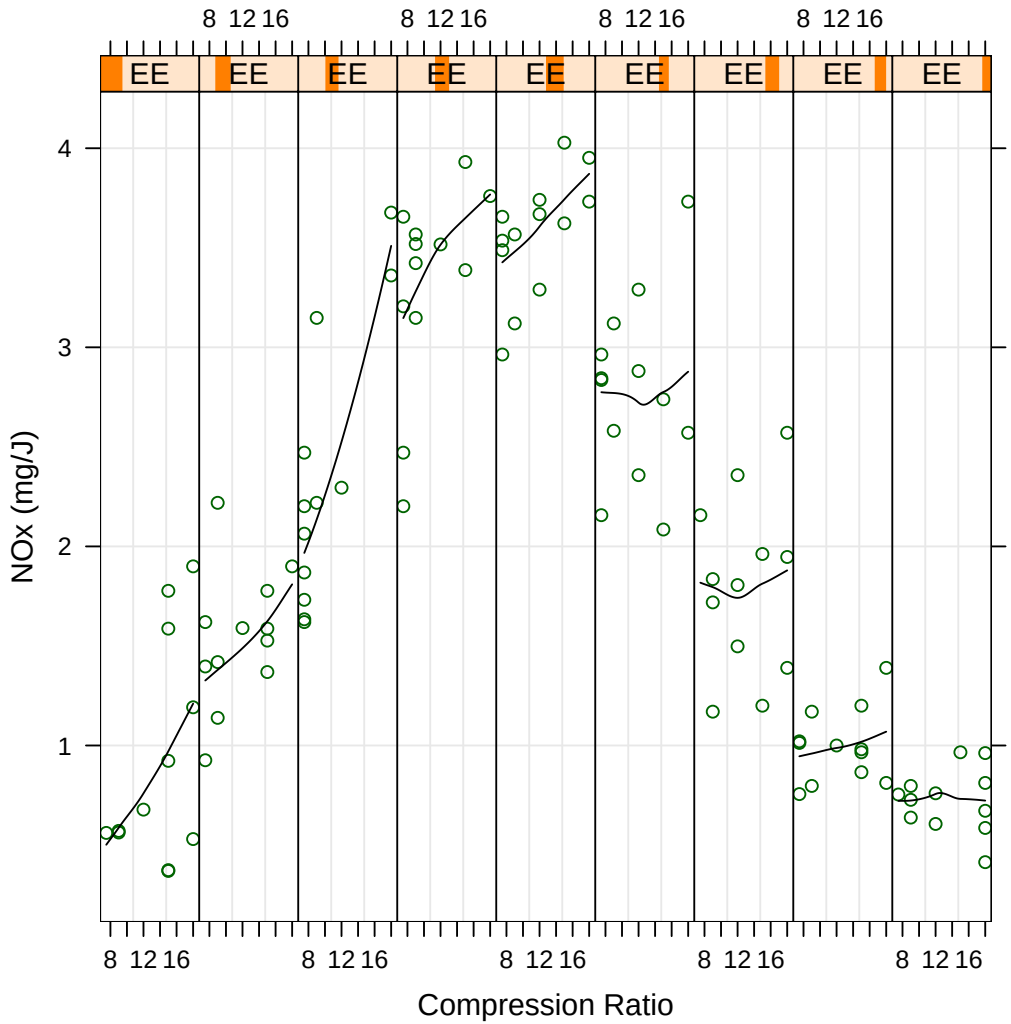
```
# Ethanol Daten
xyplot(NOx ~ C, data = ethanol)
```

```
EE <- equal.count(ethanol$E, number = 9, overlap = 1/4)
xyplot(NOx ~ C | EE, data = ethanol)
```

8.0 Grafik: Trellis / Lattice

```
EE <- equal.count(ethanol$E, number = 9, overlap = 1/4)
abgas <- xyplot(NOx ~ C | EE, data = ethanol,
  prepanel = function(x, y) prepanel.loess(x, y, span = 1),
  xlab = "Compression Ratio", ylab = "NOx (mg/J)",
    panel = function(x, y) {
      panel.grid(h = -1, v = 2)
      panel.xyplot(x, y)
      panel.loess(x, y, span=1)})
print(abgas)
print(update(abgas, aspect = "xy"))
```

8.0 Grafik: Trellis / Lattice



8.0 Grafik: Trellis / Lattice

```
print(histogram( ~ height | voice.part, data = singer,
  xlab = "Height (inches)", type = "density",
  panel = function(x, ...) {
    panel.histogram(x, ...)
    panel.mathdensity(dmath = dnorm, col = "red",
      args = list(mean = mean(x), sd = sd(x)))
  }))
```

```
densityplot( ~ height | voice.part, data = singer,
  layout = c(2, 4), xlab = "Height (inches)", bw = 5)
```

9.0 Literatur — S Definitionen

- Becker, R.A. and Chambers, J.M. (1984): **S**, *an Interactive Environment for Data Analysis and Graphics*, Wadsworth, Belmont. (brown!)
- Becker, R.A., Chambers, J.M. and Wilks, A.R. (1988): *The New* **S** *Language*, Wadsworth & Brooks/Cole, Pacific Grove. (blue!)
- Chambers, J.M. and Hastie, T.J. (1992): *Statistical Models in* **S**, Wadsworth & Brooks/Cole, Pacific Grove. (white!)
- Chambers, J.M. (1998): *Programming with Data – A Guide to the* **S** *Language*. Springer, New York. (green!)

9.0 Literatur — R Definitionen

- Gentleman, R. and Ihaka, R. (2000): Lexical Scope and Statistical Computing. *Journal of Computational and Graphical Statistics 9*, 491–508.
- Ihaka, R. and Gentleman, R. (1996): **R**: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics 5*, 299–314.
- Murrell, P. and Ihaka, R. (2000): An Approach to Providing Mathematical Annotation in Plots. *Journal of Computational and Graphical Statistics 9*, 582–599.

9.0 Literatur — R – Standardwerke I

- Braun, W.J. and Murdoch, D.J. (2007): *A First Course in Statistical Programming with R*, Cambridge University Press, Cambridge.
- Chambers, J.M. (2008): *Software for Data Analysis: Programming with R*, Springer, New York.
- Dalgaard, P. (2008): *Introductory Statistics with R*, 2. Auflage, Springer, New York.
- Everitt, B. and Hothorn, T. (2009): *A Handbook of Statistical Analysis Using R*, 2. Auflage, Chapman & Hall/CRC, Boca Raton.
- Fox, J. (2011): *An R and S–PLUS Companion to Applied Regression*, 2. Auflage, Sage Publications, Thousand Oaks.
- Murrell, P. (2005): *R Graphics*, Chapman & Hall/CRC, Boca Raton.
- Pinheiro, J.C. and Bates, D.M. (2001): *Mixed effects models in S and S–PLUS*, Springer, New York.

9.0 Literatur — R – Standardwerke II

- Venables, W.N. and Ripley, B.D. (2002): *Modern Applied Statistics with S*, 4th ed., Springer, New York. (yellow!)
- Venables, W.N. and Ripley, B.D. (2000): *S Programming*, Springer, New York.

Diesem Kurs liegt das folgende deutsche Buch zu Grunde:

- Ligges, U. (2009): *Programmieren mit R*, 3. Auflage, Springer, Heidelberg.

9.0 Literatur — Online

Unter anderem ist die folgende Literatur im PDF Format frei auf CRAN unter `http://cran.r-project.org/other-docs.html` erhältlich oder zumindest gelinkt (alle über 100 Seiten und als „brauchbar“ befunden):

- Burns, P.J. (1998): *S Poetry*, `http://www.burns-stat.com/pages/spoetry.html`.
- Maindonald, J. (2001): *Using R for Data Analysis and Graphics*.

The R Journal ist Online zu lesen unter `http://journal.r-project.org/`.

9.0 Literatur — Online (R Core Team)

Die folgenden Online–Manuals werden vom R–Project auf CRAN unter `http://CRAN.R-Project.org/manuals.html` publiziert und sind in jeder **R** Version enthalten:

- Venables, W.N., Smith, D.M., and the R Core Team (2017): *An Introduction to R*.
- R Core Team (2019): *R Data Import / Export*.
- Hornik, K. (2019): *R FAQ*.
- R Core Team (2019): *R Installation and Administration*.
- R Core Team (2019): *R Language Definition*.
- R Core Team (2019): *R: A Language and Environment for Statistical Computing*.
- R Core Team (2019): *Writing R Extensions*.