Grundlagen der Datenanalyse mit R
(R 1)
Sommersemester 2024

Dr. Gerrit Eichner
Mathematisches Institut der
Justus-Liebig-Universität Gießen
Arndtstr. 2, D-35392 Gießen, Tel.: 0641/99-32104
E-Mail: gerrit.eichner@math.uni-giessen.de
URL: http://www.uni-giessen.de/eichner

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

1.1 Was ist R, woher kommt es und wo gibt es Informationen darüber?

R ist eine nicht-kommerzielle „Umgebung“ für die Bearbeitung, grafische Darstellung und (in der Hauptsache statistische) Analyse von Daten. Es besteht aus einer Programmiersprache (die interpretiert und nicht kompiliert wird) und einer Laufzeitumgebung (unter anderem mit Grafik, einem “Debugger”, Zugriff auf gewisse Systemfunktionen und der Möglichkeit, Programmskripte auszuführen). **R** bietet eine flexible Grafikumgebung für die explorative Datenanalyse und eine Vielzahl klassisch er sowie moderner statistischer und numerischer Verfahren für die Datenanalyse und Modellierung. Viele sind in die “base distribution” (Grundausstattung) von **R**, kurz “base **R**”, integriert, aber zahlreiche weitere stehen in aktuell (April 2024) über 20700 von NutzerInnen beigesteuerten, sogenannten “(add-on-)packages” zur Verfügung, die im Bedarfsfall sehr leicht zusätzlich zu installieren sind. Außerdem können benutzereigene, problemspezifische Funktionen einfach und effektiv programmiert werden.

Die offizielle Homepage des **R**-Projektes ist `https://www.r-project.org`. Sie ist die Quelle aktuellster Informationen sowie zahlreicher weiterführender Links, von denen einige im Folgenden noch genannt werden, wie zum Beispiel zu Manualen, Büchern und den – empfehlenswerten – “Frequently Asked Questions”, kurz FAQs. (Siehe auf jener Homepage in der Rubrik “Documentation” die Punkte “Manuals”, “Books” und “FAQs”.) Die Software **R** selbst wird unter `https://cran.r-project.org` sowohl für verschiedene Linux-„Derivate“, für Microsoft-Windows als auch für Mac OS X bereitgehalten. Außerdem ist dort hinter “Contributed” ein Link zu noch mehr von NutzerInnen beigesteuerten Manualen und Tutorien. Sehr nützlich sind auch die unter dem Link “Task Views” zu großen Themen zusammengestellten so genannten “CRAN Task Views” (aktuell 44 an der Zahl, April 2024). Sie können einen strukturierten und kommentierten, themenspezifischen ersten Überblick verschaffen über einen Teil der oben bereits erwähnten hohen Anzahl an Zusatzpaketen und liefern ein Werkzeug für die automatische Installation aller zu den jeweiligen Themen gehörenden Pakete.

Weitere evtl. sehr interessante Informationsquellen:

- `https://journal.r-project.org` ist die Web-Site der offiziellen, referierten Zeitschrift des **R**-Projektes für “statistical computing”.
- Die E-Mail-Liste **R-help** ist ein gutes Medium, um Diskussionen über und Lösungen für Probleme mitzubekommen bzw. selbst Fragen zu stellen (auch wenn der Umgangston gelegentlich etwas rauh ist). Zugang zu dieser Liste ist am einfachsten über den Link “Mailings Lists” auf der o. g. Homepage des **R**-Projektes möglich.
- Unter `https://en.wikipedia.org/wiki/R_programming_language` sind diverse (Hintergrund-)Daten und Fakten über **R** verfügbar.
- RSeek unter `https://www.rseek.org` offeriert eine äußerst leistungsfähige Suchmaschine, die mehrere Webseiten mit **R**-Bezug gleichzeitig durchsucht.
- Auf der Seite “Stack Overflow” (die zum “Stack Exchange”-Netzwerk von “free, community-driven Q & A sites” gehört und eine solche für “professional and enthusiast programmers” ist) sind unter dem “tag” r derzeit (April 2024) gut 505900 (!) **R**-spezifische Fragen und Antworten gesammelt, zu denen man direkt via `https://stackoverflow.com/questions/tagged/r` gelangt. Sie sind dort noch feiner kategorisierbar. (Auf `https://stackoverflow.com/tags/r/info` erhält man neben Informationen über **R** selbst insbesondere Hinweise darüber, welcher Art Fragen mit dem tag r wie gestellt werden sollten. Außerdem sind dort zahlreiche Links zu weiteren **R**-bezogenen Resourcen zusammengestellt.) (Für statistische Fragen und Antworten ist übrigens die Seite “Cross Validated” auf `https://stats.stackexchange.com` eine mögliche Anlaufstelle.)

- `https://www.r-bloggers.com` ist ein "blog aggregator" für (englischsprachige) Blogs, die von **R**-NutzerInnen zu **R** oder diversesten Themen, die mit **R** bearbeitet werden oder zu tun haben, geschrieben werden. Die Qualität der Beiträge variiert allerdings ...
- Ein (etwas älterer) "Field Guide to the R Ecosystem" steht unter `https://fg2re.sellorm.com/` zur Verfügung, der einen (z. B. ersten) Überblick über die wichtigsten Komponenten des „**R**-Ökosystems" im Großen und Ganzen zu geben versucht.
- Auf `https://www.r-graph-gallery.com` liegt eine umfangreiche "R Graph Gallery", die mehrere Hundert, mit **R** angefertigte, exzellente Grafiken samt ihres reproduzierbaren **R**-Codes präsentiert (wobei für die meisten Grafiken die Zusatzpakete `tidyverse` und `ggplot2` genutzt werden).

Etwas zur Geschichte: Mitte bis Ende der 1970er Jahre wurde in den AT&T Bell Laboratorien (heute Lucent Technologies, Inc.) die „Statistik-Sprache" **S** entwickelt, um eine interaktive Umgebung für die Datenanalyse zu schaffen. 1991 erschien eine Implementation von **S**, für die heute die Bezeichnung "S engine 3" (kurz S3) in Gebrauch ist. 1998 wurde eine völlig neu konzipierte "S engine 4" (kurz S4) veröffentlicht. Für ausführlichere historische Informationen siehe `https://en.wikipedia.org/wiki/S_programming_language`.

Zwischen 1988 und 2008 gab es unter dem Namen S-PLUS eine kommerzielle Version von **S** mit vielen zusätzlichen Funktionen (siehe hierzu `https://en.wikipedia.org/wiki/S-PLUS`).

R ist ebenfalls eine Implementation der Sprache **S** (quasi ein „Dialekt") und entstand ab 1992. Sie ist kostenlose, "open source" Software mit einem GNU-ähnlichen Urheberrecht und offizieller Bestandteil des GNU-Projektes „GNU-S". **R** ähnelt „äußerlich" sehr stark **S** und „innerlich" (= semantisch) der Sprache "Scheme". Faktisch gibt es derzeit also drei Implementationen von **S**: Die alte "S engine 3", die neue "S engine 4" und **R**.

Im Jahr 2000 wurde die **R**-Version 1.0.0 veröffentlicht, im Jahr 2004 Version 2.0.0, 2013 Version 3.0.0 und schließlich im April 2020 Version 4.0.0; inzwischen ist – am 29. Februar 2024 – Version 4.3.3 erschienen. Etwa jedes Frühjahr gibt es (bzw. gab es zumindest bisher) ein "upgrade" von x.y.z auf x.y+1.0 und während des folgenden Jahres ein bis drei kleinere, jeweils von x.y.z auf x.y.z+1.

R-Code wird prinzipiell über eine Kommandozeilenschnittstelle eingegeben und ausgeführt. Allerdings gibt es inzwischen mehrere *exzellente* und empfehlenswerte grafische Benutzerschnittstellen ("graphical user interfaces" = GUIs), Editoren und ganze integrierte Entwicklungsumgebungen ("integrated development environments" = IDEs), die **R** unterstützen (siehe auch am Ende von Abschnitt 1.6). Auf `https://en.wikipedia.org/wiki/R_programming_language` sind, wie schon erwähnt, umfangreiche weitere Daten und Fakten verfügbar.

1.2 Vor dem Start eines Projektes mit R unter Microsoft-Windows

Vorbemerkung: Da wir uns in diesem Abschnitt auf MS-Windows konzentrieren, könnten BenutzerInnen, die mit Mac OS X arbeiten, hilfreiche Erklärungen in den "R Mac OS X FAQ" unter `https://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html` finden.

Soeben (am Ende von 1.1) wurde und am Ende von Abschnitt 1.6 wird auch noch etwas ausführlicher auf diverse GUIs und IDEs für die effiziente Arbeit mit **R** hingewiesen (für die ich z. B. die Nutzung von RStudio wärmstens empfehlen kann). Wir beschränken uns in diesem Vorlesungsskript aber – ganz "old school" – auf die Nutzung des „klassischen" **R**-GUIs, da es für uns bis auf Weiteres ausreichen wird und wir uns hier aus Gründen der Übersichtlichkeit auf **R** selbst konzentrieren wollen. (In den Übungen wird RStudio zum Einsatz kommen.)

Empfehlung: Vor dem Beginn der eigentlichen Arbeit an einem konkreten Projekt sollten Sie sich dafür ein eigenes (Arbeits-)Verzeichnis anlegen und dafür sorgen, dass alle mit dem Projekt

zusammenhängenden Dateien darin gespeichert sind bzw. werden, auch die von **R** automatisch generierten. (Dies hat nichts mit **R** direkt zu tun, dient lediglich der eigenen Übersicht, der Vereinfachung der Arbeit und wird „außerhalb“ von **R** gemacht.) Damit dies geschieht, können Sie sich wie folgt eine Verknüpfung zu **R** in jenes Arbeitsverzeichnis legen und diese geeignet konfigurieren:

Nach der (erfolgreichen) Installation von **R** befindet sich üblicherweise ein **R**-Icon (genauer eine Verknüpfung zu **R**) auf dem Windows-Desktop, für das sich beim Anklicken mit der rechten Maustaste ein Menü öffnet. Darin wird (u. a.) „Verknüpfung erstellen“ angeboten, was Sie auswählen. Eine evtl. auftauchende Frage nach dem Erstellen einer solchen auf dem Desktop bejahen Sie, um genau das zu erreichen. Das sodann auf dem Desktop neu erstellte, zweite **R**-Icon verschieben Sie in das zuvor angelegte Arbeitsverzeichnis. Hier wird dieses Icon mit der rechten Maustaste angeklickt, um in dem erscheinenden Menü unter dem Reiter „Verknüpfung“ das Feld „Ausführen in“ modifizieren zu können, denn dorthinein muss der *vollständige* Pfad zum Arbeitsverzeichnis kopiert werden, den Sie sich (wohl am einfachsten) aus der Adressleiste des Windows-Explorers – durch geschicktes Anklicken derselbigen – holen.

Wie **R** veranlasst wird, in ein gewünschtes Arbeitsverzeichnis zu wechseln, *ohne* dass darin bereits eine Verknüpfung angelegt worden ist, wird auf Seite 11 in Abschnitt 1.6 beschrieben.

1.3 Aufruf des R-GUIs unter Microsoft-Windows

Starten Sie **R** mit Hilfe des jeweiligen **R**-Icons in Ihrem Arbeitverzeichnis (oder finden Sie **R** in Windows' Start-Menü und wechseln Sie wie in Abschnitt 1.6 beschrieben in das gewünschte Arbeitsverzeichnis). Das – Windows-spezifische! – **R**-GUI (= “graphical user interface”) öffnet sich in einem eigenen Fenster, worin in einem Kommandozeilen-(Teil-)Fenster namens “**R** Console” eine Begrüßungsmeldung ausgegeben wird, die *sehr* nützliche Hinweise darüber enthält, wie man an Informationen über und Hilfe für **R** kommt und wie man es zitiert. Darunter wird der **R**-Prompt „>“ dargestellt, der anzeigt, dass **R** ordnungsgemäß gestartet ist und nun **R**-Befehle eingegeben werden können (siehe Abb. 1).

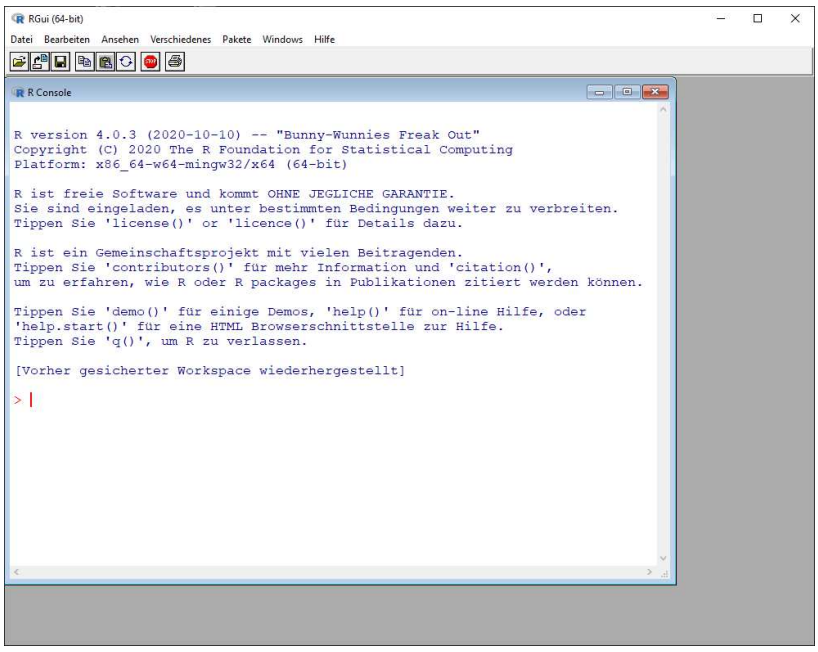

Abbildung 1: Das **R**-GUI (in seiner Voreinstellung) unter Windows mit der **R**-Console und ihrer Begrüßungsmeldung wie es im Wesentlichen auch in der aktuellsten Version von **R** aussieht. (Unter Mac OS X sieht das entsprechende R.app GUI bis auf die Icons und Menues am oberen Rand sehr ähnlich aus.)

1.4 Eingabe und Ausführung von R-Befehlen

S und damit **R** sind funktionale, objektorientierte Sprachen, in denen alles sogenannte *Objekte* sind. Dabei werden Groß- und Kleinschreibung beachtet, d. h., `A` und `a` bezeichnen zwei *verschiedene* Objekte. Jeder Befehl (ist selbst ein Objekt und) besteht – vereinfacht dargestellt – entweder aus einem Ausdruck ("expression") oder einer Zuweisungsanweisung ("assignment"), die jeweils sofort nach deren Eingabe am **R**-Prompt und ihrem Abschluss durch die Return-Taste ausgeführt werden. (Es handelt sich bei **R** um einen Interpreter, d. h., die Eingaben müssen nicht erst kompiliert, sprich in Maschinensprache übersetzt werden; dies geschieht automatisch nach dem Tippen der Return-Taste, falls sie syntaktisch korrekt und vollständig waren.)

Ausdrücke: Ist ein Befehl ein Ausdruck, so wird dieser ausgewertet, das Resultat am Bildschirm ausgedruckt und vergessen. Beispielsweise liefert die Eingabe von `17 + 2` (gefolgt vom Druck der Return-Taste) folgendes:

```
> 17 + 2
[1] 19
```

(Hierbei deutet `[1]` vor der Ausgabe an, dass die Antwort von **R** mit dem ersten Element eines – hier nur einelementigen – Vektors beginnt. Dazu später mehr.)

Zuweisungsanweisungen: Eine Zuweisungsanweisung, gekennzeichnet durch den Zuweisungsoperator `<-`, der aus den zwei Zeichen `<` und `-` ohne Leerzeichen dazwischen (!) besteht, wertet den auf der rechten Seite von `<-` stehenden Ausdruck aus und weist den Resultatwert dem Objekt links von `<-` zu. (Das Resultat wird nicht automatisch ausgegeben.) Zum Beispiel:

```
> x <- 119 + 2
```

Die Eingabe des Objektnamens, der selbst ein Ausdruck ist, veranlasst die Auswertung desselben und liefert als Antwort den „Wert" des Objektes (hier eben eine Zahl):

```
> x
[1] 121
```

Will man also eine Zuweisung durchführen und ihr Ergebnis gleich prüfen, kann das wie eben gezeigt geschehen. Dieses zweischrittige Vorgehen (erst Zuweisung und dann Auswertung des erzeugten Objektes) kann abgekürzt werden, indem die Zuweisungsanweisung in runde Klammern `()` gepackt wird. Sie erzwingen, dass nach der Ausführung des „Inneren" der Klammern das Ergebnis dieser Ausführung, also das entstandene Objekt ausgewertet wird:

```
> (x <- 170 + 2)
[1] 172
```

Mehrere Befehle, Kommentare: Mehrere Befehle können zeilenweise, also durch einen Zeilenumbruch mittels Return-Taste getrennt, eingegeben werden oder gemeinsam in einer Zeile durch einen Strichpunkt (;) getrennt. Befindet sich irgendwo in der Zeile das Zeichen `#` (das Doppelkreuz), so wird jeglicher Text rechts davon bis zum Ende dieser Zeile als Kommentar aufgefasst und ignoriert. Im folgenden Beispiel stehen zwei Zuweisungsanweisungen (worin der arithmetische Divisionsoperator `/` und die **R**-Funktion `sqrt` zur Berechnung der Quadratwurzel verwendet werden) und ein Kommentar in einer Zeile:

```
> (kehrwert <- 1/x);    (wurzel <- sqrt(x))   # Ignorierter Kommentar.
[1] 0.005813953
[1] 13.11488
```

Die Auswertung (von syntaktisch korrekten und vollständigen Ausdrücken) beginnt immer erst nach einer Eingabe von Return und verläuft dann stets sequenziell. Oben wurde also zunächst `kehrwert` erzeugt, dann ausgewertet und schließlich ausgegeben, sodann wurde `wurzel` erzeugt sowie ebenfalls ausgewertet und ausgegeben.

Befehlsfortsetzungsprompt: Ist ein Befehl nach Eingabe von Return oder am Ende der Zeile syntaktisch noch nicht vollständig, so liefert **R** einen „Befehlsfortsetzungsprompt“, nämlich das Zeichen `+`, und erwartet weitere Eingaben in der nächsten Zeile. Dies geschieht so lange bis der Befehl syntaktisch korrekt abgeschlossen ist (oder die ESC-Taste gedrückt und so der Vorgang abgebrochen wird):

```
> sqrt(pi * x^2
+ )                  # Die schliessende Klammer fehlt! Das "+" kommt von R.
[1] 304.8621         # Hier wird sie "nachgeliefert" und ...
```

... der nun syntaktisch korrekte Ausdruck ausgewertet.

Objektnamenvervollständigung: In vielen Systemen ist eine nützliche (halb-)automatische Objektnamenvervollständigung am **R**-Prompt möglich, die mit der Tabulator-Taste erzielt wird und sowohl eingebaute als auch benutzereigene Objekte einbezieht. Beispiel: In unserer laufenden Sitzung nach der Erzeugung des Objektes `kehrwert` (in der Mitte von Seite 4) liefert das Eintippen von

```
> keh
```

gefolgt vom Druck der Tabulator-Taste die Vervollständigung der Eingabe zu `kehrwert`, da `keh` den gesamten Objektnamen schon eindeutig bestimmt. Bei Mehrdeutigkeiten passiert beim einmaligen Druck der Tabulator-Taste nichts, aber ihr *zweimaliger* Druck liefert eine Auswahl der zur Zeichenkette passenden Objektnamen. Beispiel:

```
> ke
```

gefolgt vom *zweimaligen* Druck der Tabulator-Taste bringt (hier)

```
kehrwert   kernapply  kernel
```

als Erwiderung, an der man erkennt, dass die bisherige Zeichenkette für Eindeutigkeit nicht ausreicht. Jetzt kann durch das nahtlose Eintippen weiterer Buchstaben und die erneute Verwendung der Tabulator-Taste gezielt komplettiert werden.

Ausführung von R-Skripten: Umfangreicheren **R**-Code, wie er sich schnell bei etwas aufwändigeren Auswertungen oder für Simulationen ergibt, wird man häufig in einer Textdatei als **R**-Skript (= **R**-Programm) speichern. Seine vollständige Ausführung lässt sich am **R**-Prompt durch die Funktion `source` komfortabel erzielen. Ihr Argument muss der Dateiname des Skripts in Anführungszeichen sein bzw. der Dateiname samt Pfad, wenn sich die Datei nicht im aktuellen Arbeitsverzeichnis befindet. Zum Beispiel versucht

```
> source("Simulation")
```

den Inhalt der Datei „`Simulation`“ aus dem aktuellen Arbeitsverzeichnis (unsichtbar) einzulesen und ihn als (hoffentlich fehlerfreien) **R**-Code auszuführen, während

```
> source("Analysen0815/Versuch_007")
```

dasselbe mit der Datei „`Versuch_007`“ aus dem Unterverzeichnis „`Analysen0815`“ des aktuellen Arbeitsverzeichnisses macht. (Falls Sie nicht wissen, welches das aktuelle Arbeitsverzeichnis ist und welche Dateien sich darin befinden, bekommen Sie es z. B. mit `getwd()` genannt bzw. mit `dir()` oder `list.files()` seinen Inhalt widergegeben. Allerdings könnte auch hier nach der

Eingabe von, z. B., `source("Analysen0815/Ver` durch den *zweimaligen* Druck der Tabulator-Taste eine (halb-)automatische Dateinamenvervollständigung aktiviert werden!)

Dringende Empfehlung zur Lesbarkeit von R-Code: Zur Bewahrung oder Steigerung der Lesbarkeit von umfangreicherem **R**-Code sollten *unbedingt* Leerzeichen und Zeilenumbrüche geeignet verwendet werden! Hier als Beispiel ein Stück **R**-Code, wie man ihn *auf keinen Fall* produzieren sollte:

```
ifelse(x<z,W$Sub[v],A/(tau*(exp((omega-x)/(2*tau))+exp(-(omega-x)/(2*tau)))^2))
```

Und hier derselbe Code, der durch die Verwendung von Leerzeichen, Zeilenumbrüchen und Kommentaren deutlich besser lesbar geworden ist:

```
ifelse(x < z, W$Sub[v],
       A / (tau * (exp((omega - x) / (2*tau)) +
                   exp(-(omega - x) / (2*tau)))^2
           ) # Ende des Nenners von A / (....)
      ) # Ende von ifelse(....)
```

Die Suche nach unvermeidlich auftretenden Programmierfehlern wird dadurch sehr erleichtert.

1.5 Benutzerdefinierte Objekte: Zulässige Namen, speichern und löschen

Alle benutzerdefinierten Objekte (Variablen, Datenstrukturen und Funktionen) werden von **R** während der laufenden Sitzung gespeichert. Ihre Gesamtheit wird "workspace" genannt. Eine Auflistung der Namen der aktuell im workspace vorhandenen Objekte erhalten Sie durch den Befehl

```
> objects()
```

Dasselbe erreicht man mit dem kürzeren Aufruf `ls()`. (Zusätzliche Informationen über die Struktur der Objekte liefert `ls.str()`.)

Zulässige Objektnamen bestehen aus Kombinationen von Klein- und Großbuchstaben, Ziffern und „`.`“ sowie „`_`“, wobei sie mit einem Buchstaben oder „`.`“ beginnen müssen. In letzterem Fall darf das zweite Namenszeichen keine Ziffer sein und diese Objekte werden „unsichtbar“ gespeichert, was heißt, dass sie beim Aufruf von `objects()` oder `ls()` nicht automatisch angezeigt werden. Memo: Groß-/Kleinschreibung wird beachtet! Beispiele: `p1`, `P1`, `P.1`, `Bloodpool.Info.2008`, `IntensitaetsKurven`, `Skalierte_IntensitaetsKurven`

Es empfiehlt sich zur Verbesserung der Code-Lesbarkeit durchaus, lange und aussagefähige Objektnamen zu verwenden, womit übrigens auch das in der folgenden Warnung beschriebene Problem vermieden werden kann.

Warnung vor Maskierung: Objekte im benutzereigenen workspace haben i. d. R. Priorität über **R**-spezifische Objekte mit demselben Namen! Das bedeutet, dass **R** die ursprüngliche Definition möglicherweise nicht mehr zur Verfügung hat und stattdessen die neue, benutzereigene zu verwenden versucht. Man sagt, die benutzereigenen Objekte „maskieren“ die **R**-spezifischen. Dies kann Ursache für (zunächst seltsam erscheinende) Warn- oder Fehlermeldungen oder – schlimmer – augenscheinlich korrektes Verhalten sein, welches aber unerkannt (!) falsche Resultate liefert.

Vermeiden Sie daher Objektnamen wie z. B. `c`, `s`, `t`, `C`, `T`, `F`, `matrix`, `glm`, `lm`, `range`, `tree`, `mean`, `var`, `sin`, `cos`, `log`, `exp` und `names`. Sollten seltsame Fehler(-meldungen) auftreten, kann

es hilfreich sein, sich den workspace mit `objects()` oder `ls()` anzusehen, gegebenenfalls einzelne, verdächtig benannt erscheinende Objekte zu löschen und dann einen neuen Versuch zu starten. Ob ein Objektname bereits vergeben ist und ein Zugriffskonflikt oder eine Maskierung drohen würde, können Sie vor seiner ersten Verwendung z. B. dadurch überprüfen, dass Sie ihn einfach am **R**-Prompt eingeben und auszuwerten versuchen lassen.

Gelöscht werden Objekte durch die Funktion `rm` (wie "remove"). Sie benötigt als Argumente die durch Komma getrennten Namen der zu löschenden Objekte:

```
> rm(a, x, Otto, Werte.neu)
```

löscht die Objekte mit den Namen `a`, `x`, `Otto` und `Werte.neu`, egal ob es Variablen, Datenstrukturen oder Funktionen sind.

Eine Löschung *aller* benutzerdefinierten Objekte auf einen Schlag erzielt man mit dem Befehl

```
> rm(list = objects())     # Radikale Variante: Loescht (fast) alles!
```

der aber natürlich mit Vorsicht anzuwenden ist. Ebenfalls vollständig „vergessen" werden die in der aktuellen Sitzung zum workspace *neu hinzugekommenen* Objekte und die an im workspace bereits existierenden Objekten durchgeführten Änderungen, wenn man beim Verlassen von **R** die Frage "`Save workspace image? [y/n/c]`" mit `n` beantwortet (siehe auch §1.5.4, Seite 9). Also Vorsicht hierbei!

Eine permanente Speicherung der benutzerdefinierten Objekte des workspaces wird beim Verlassen von **R** durch die Antwort `y` auf die obige Frage erreicht. Sie führt dazu, dass alle Objekte des momentanen workspaces in einer Datei namens `.RData` im aktuellen Arbeitsverzeichnis gespeichert werden. Wie man sie in der nächsten **R**-Session „zurückholt" und weitere Details werden in §1.5.4 beschrieben.

1.5.1 Die "command history"

R protokolliert die eingegebenen Befehle mit und speichert sie in einer Datei namens `.RHistory` im aktuellen Arbeitsverzeichnis, wenn beim Verlassen von **R** der workspace gespeichert wird. Am **R**-Prompt kann man mit Hilfe der Cursor-Steuertasten (= Pfeiltasten) in dieser "command history" umherwandern, um so frühere Kommandos „zurückzuholen", sie zu editieren, falls gewünscht, und sie durch Tippen von Return (an jeder beliebigen Stelle im zurückgeholten Kommando) erneut ausführen zu lassen.

1.5.2 Demos

Um sich einen ersten Einblick in die Fähigkeiten von **R** zu verschaffen, steht eine Sammlung von halbautomatisch ablaufenden Beispielen („Demos") zur Verfügung, die mit Hilfe der Funktion `demo` gestartet werden können. Der Aufruf von `demo()` ohne Angabe eines Arguments liefert eine Übersicht über die (in der Basisversion) verfügbaren Demos. Die Angabe des Demo-Namens als Argument von `demo` startet die genannte Demo. Beispiel:

```
> demo(graphics)
```

startet eine Beispielesammlung zur Demonstration von **R**sGrafikfähigkeiten. Beendet wird sie durch Eintippen von `q` (ohne Klammern).

1.5.3 Das Hilfesystem

R hat eine „eingebaute“ Dokumentation. Sie ist in den meisten Installationen per Voreinstellung ein HTML-basiertes Hilfesystem, das bei einer Anfrage den jeweils voreingestellten Web-Browser startet, falls er noch nicht läuft, und die angeforderte Hilfeseite darin in einem neuen „Tab“ darstellt. Die Hilfe wird objektspezifisch mit `help(....)` aufgerufen, wobei anstelle von „`....`“ der Name eines Objektes, sprich einer Funktion, eines Datensatzes oder etwas anderem steht, worüber man Informationen haben will. Zum Beispiel liefert

```
> help(mean)              # = ?mean
```

ausführliche Informationen über die (eingebaute) Funktion `mean`. Die Kurzform `?mean` liefert dasselbe wie `help(mean)`. (Ohne Argument, also durch `help()`, erhält man übrigens Hilfe zur Funktion `help` selbst.)

Durchaus hilfreich in diesem Zusammenhang kann die stichwortbasierte Suche mittels der Funktion `help.search` sein. Sie erlaubt auf verschiedene Arten die Angabe von (auch vagen) „Textmustern“ oder „Schlüsselwörten“, nach denen in **R**s Hilfeseiten – an gewissen Stellen – gesucht werden soll, falls man den exakten Objektnamen nicht (mehr) oder noch nicht weiß.

Sicher *sehr* hilfreich und absolut empfehlenswert, da oft sehr lehrreich, sind die Beispiele, die eine jede Hilfeseite (so man sie gefunden hat) an ihrem Ende im Abschnitt “Examples” üblicherweise bereithält. Sie sind mit der Funktion `example` automatisch ausführbar, wenn man ihr als Argument den Namen des interessierenden Objektes übergibt, also z. B. für die Funktion `mean` durch `example(mean)`.

Allgemein wird die Startseite von **R**s Hilfesystem aufgerufen mit

```
> help.start()
```

Dies sollte den jeweils voreingestellten Web-Browser starten, falls er noch nicht läuft, und – nach wenigen Sekunden – darin die in Abb. 2 zu sehende Seite anzeigen. Wird ein anderer als der voreingestellte Browser bevorzugt, kann seine Verwendung durch

```
> help.start(browser = "....")
```

erzwungen werden (falls er installiert ist), wobei anstelle von `....` der Name des Browser-Programmes stehen muss.

Empfehlungen:

1. Hinter dem Link An Introduction to R (in der Startseite des Hilfesystem in Abb. 2 links unterhalb von `Manuals` zu sehen) steckt eine gute, HTML-basierte, grundlegende Einführung in die Arbeit mit **R** einschließlich einer Beispielsitzung in ihrem Appendix A.

2. Search Engine & Keywords unterhalb von `Reference` führt zu einer nützlichen Suchmaschine für die Hilfeseiten, die die Suche nach “keywords”, Funktionsnamen, Datennamen und Text in den Überschriften jener Hilfeseiten erlaubt. Im Fall einer erfolgreichen Suche wird eine Liste von Links zu den betreffenden Hilfeseiten gezeigt, auf denen umfangreiche Informationen zu finden sind.

3. Unter Frequently Asked Questions (FAQs) unterhalb von `Miscellaneous Material` sind die Antworten auf einige der typischen Fragen, die den neuen “useR” und die neue “useRin” plagen könnten, zu finden und Lösungen für gewisse Probleme angedeutet.

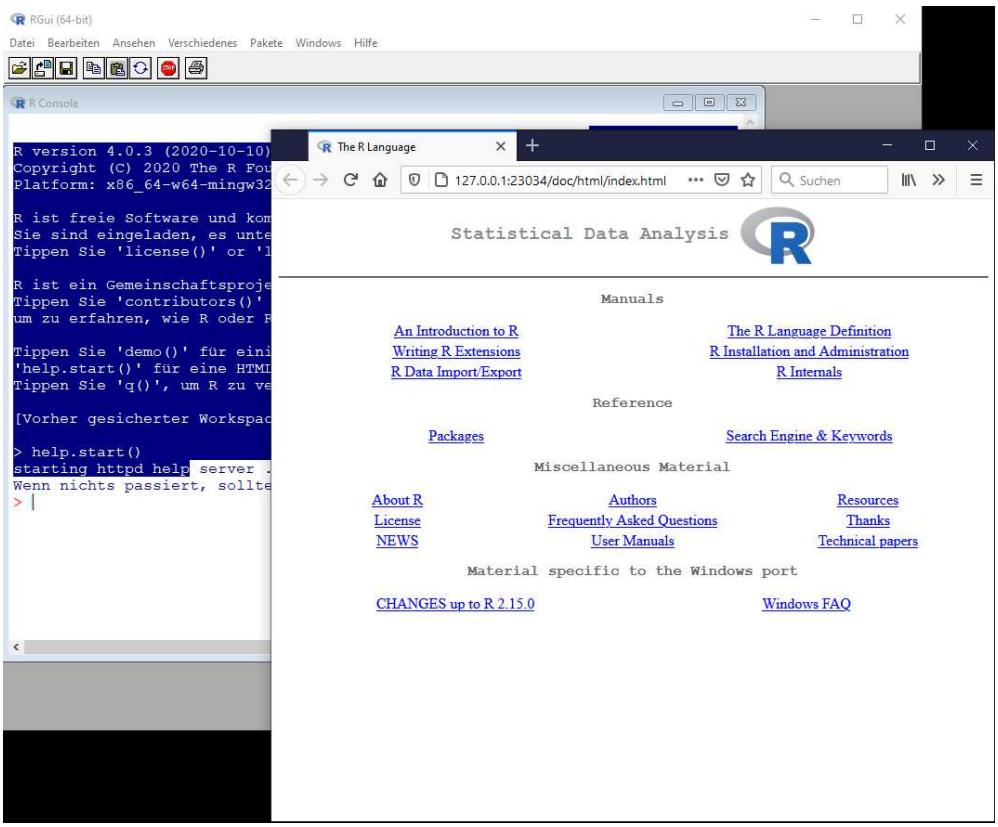

Abbildung 2: Startseite von **R**s Browser-basiertem Hilfesystem.

1.5.4 Beenden von R

Um **R** zu beenden, tippen Sie am Prompt `q()` ein (dabei die leeren Klammern nicht vergessen, denn auch bei `q` handelt es sich um eine Funktion!):

```
> q()
```

Bevor **R** wirklich beendet wird, werden Sie hier *stets* gefragt, ob die Daten, genauer: die Objekte dieser Sitzung gespeichert werden sollen. Sie können `y`(es), `n`(o) oder `c`(ancel) eingeben, um die Daten (vor dem Verlassen von **R**) permanent speichern zu lassen bzw. um **R** zu verlassen, ohne sie zu speichern, bzw. um die Beendigung von **R** abzubrechen (und sofort zu **R** zurückzukehren). Die permanente Speicherung der Objekte des aktuellen workspaces geschieht in der Datei `.RData` des aktuellen Arbeitsverzeichnisses (das Sie im Fall, dass Sie nicht mehr wissen, „wo Sie sind“, mit `getwd()` abfragen können; vgl. Seite 5).

Jener workspace und damit die vormals gespeicherten Objekte können beim nächsten Start von **R** aus dieser Datei `.RData` rekonstruiert und wieder zur Verfügung gestellt werden. Dies geschieht im gewünschten Arbeitsverzeichnis entweder automatisch, indem man **R** durch einen Doppelklick auf das dortige Icon der Datei `.RData` startet (falls `.RData`-Dateien mit **R** verknüpft sind, was bei einer ordnungsgemäßen Windows-Installation automatisch der Fall sein sollte), oder durch durch einen Doppelklick auf die entsprechend angelegte Verknüpfung, oder schließlich „von Hand“, indem man eine „irgendwo und irgendwie“ gestartete **R**-Session veranlasst (z. B. mit Hilfe des **R**-GUIs wie im folgenden Abschnitt 1.6 beschrieben), das gewünschte Verzeichnis als das aktuelle Arbeitsverzeichnis zu wählen und den vormaligen workspace wieder einzulesen.

1.6 Rs "graphical user interface" unter Microsoft-Windows

Es folgt eine rudimentäre Beschreibung einiger Funktionen des Windows-spezifischen und selbst sehr rudimentären **R**-GUIs.

Das **GUI-Menü** (zu sehen am oberen Rand links in Abb. 1 oder Abb. 3) enthält sechs Themen (von „Datei" bis „Hilfe"), von denen die folgenden drei für die neue "use**R**in" und den neuen "use**R**" wichtig oder interessant sind:

1. „Datei": Offeriert einen einfachen (Skript-)Editor, erlaubt etwas Datei-Management und bietet das Speichern und Wiederherstellen ehemaliger **R**-Sitzungen (bestehend aus ihrem workspace und ihrer command history).

2. „Bearbeiten": Bietet Möglichkeiten, **R**-Code, der im Editor eingetippt worden ist, ausführen zu lassen.

3. „Hilfe": Enthält mehrere Punkte zur Hilfe und zu Hintergrundinformationen.

Etwas detailliertere Erläuterungen folgen:

Rs Skripteditor: Zu finden im R-GUI unter „Datei" → „Neues Skript" oder „Öffne Skript ..." (wie in Abb. 3 angedeutet). Ein zweites (Teil-)Fenster mit dem Titel „**R** Editor" öffnet sich (rechts in Abb. 3). Das GUI-Menü-Thema "Windows" bietet übrigens Möglichkeiten, die Teilfenster innerhalb des GUIs schnell anzuordnen.

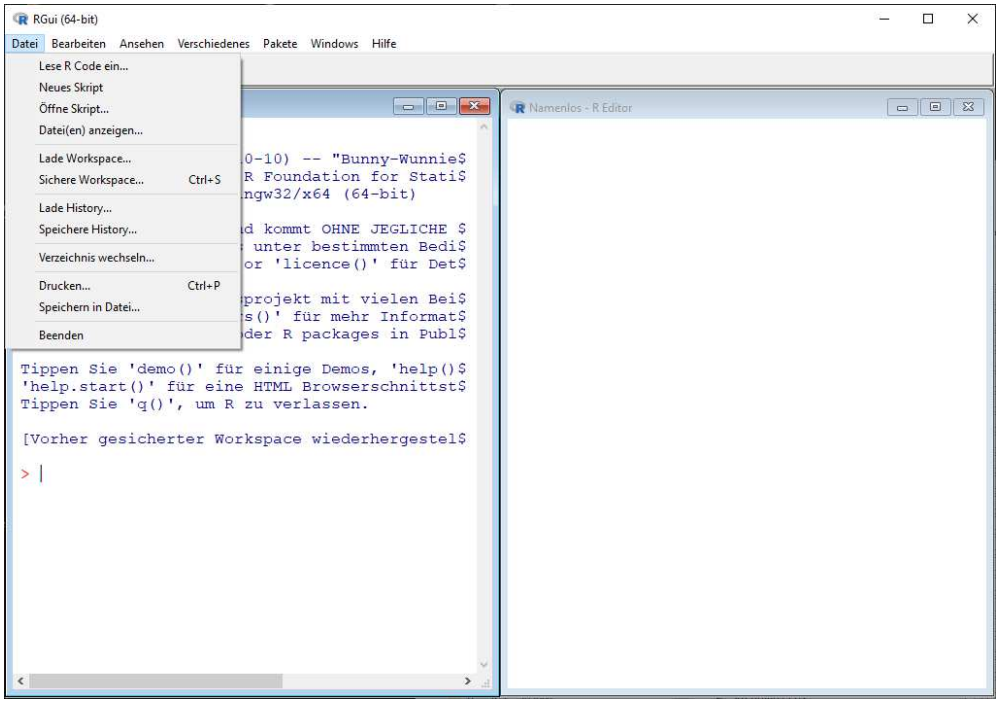

Abbildung 3: Aufruf des **R**-Editors mit einem neuen, also leeren Skript.

Der **R**-Editor kann verwendet werden für die Entwicklung und das Abspeichern von **R**-Code (zusammen mit Kommentaren), der später wieder genutzt oder weiterentwickelt werden soll. Editiert wird darin wie in jedem primitiven Texteditor. Die übliche, einfache Funktionalität steht zur Verfügung: "Copy, cut & paste" mit Hilfe der Tastatur oder der Maus, Ctrl-Z bzw. Strg-Z als Rückgängig-Aktion etc. (Dies und anderes ist auch im GUI-Menü-Thema „Bearbeiten" zu finden.)

Das **Ausführen von R-Code**, der im **R**-Editor steht, kann auf mindestens vier (zum Teil nur geringfügig) verschiedene Methoden erreicht werden (beachte dazu Abb. 4 und siehe auch das GUI-Menü-Thema „Bearbeiten“):

1. Markiere den Code(-Ausschnitt), der ausgeführt werden soll, z. B. mit dem Cursor, und verwende den üblichen “copy & paste”-Mechanismus, um ihn an den Prompt der **R**-Console zu „transportieren“. Dort wird er sofort ausgeführt. (Nicht zu empfehlen, da mühselig!)

2. Wenn eine ganze, aber einzelne Zeile des Codes ausgeführt werden soll, platziere den Cursor in eben jener Zeile des Editors und tippe Ctrl-R bzw. Strg-R oder klicke auf das *dann vorhandene* dritte Icon von links unter dem GUI-Menü (siehe die Icons oben in Abb. 4, worin der **R**-Editor das aktive Fenster ist, und vergleiche sie mit denen oben in Abb. 2, in der die **R**-Console das aktive Fenster ist).

3. Soll ein umfangreicherer Teil an Code ausgeführt werden, markiere ihn im Editor wie in Abb. 4 rechts zu sehen und tippe Ctrl-R bzw. Strg-R oder nutze das in Punkt 2 erwähnte Icon.

4. `source(....)`, wie auf Seite 5 in Abschnitt 1.4 beschrieben, funktioniert natürlich auch hier, wenn der aktuelle Inhalt des **R**-Editors bereits in einer Datei abgespeichert wurde und deren Dateiname (nötigenfalls inklusive vollständiger Pfadangabe) als Argument an `source` übergeben wird. Dies führt i. d. R. sogar zu einer schnelleren Code-Ausführung als die obigen Varianten.

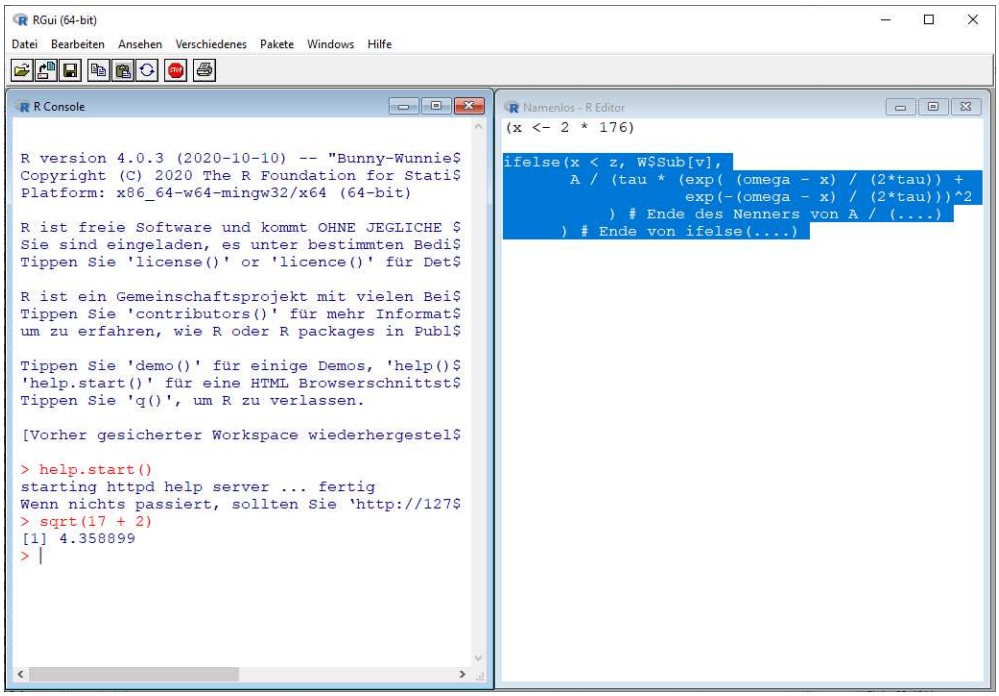

Abbildung 4: Links: direkt am Prompt eingegebener und ausgeführter **R**-Code. Rechts: markierter **R**-Code im **R**-Editor, der ausgeführt werden soll.

Wechsel des Arbeitsverzeichnisses und **Laden eines workspaces:** Im GUI-Menü-Thema „Datei“ sind auch die Punkte zu finden, die für einen Wechsel des Arbeitsverzeichnisses und das Laden eines workspaces nützlich sind, falls **R** nicht durch einen Doppelklick auf die `.RData`-Datei oder auf seiner dortige Verknüpfung im gewünschten Arbeitsverzeichnis gestartet wurde. Es sind dies die Punkte „Verzeichnis wechseln ...“ bzw. „Lade Workspace ...“ (siehe hierfür nochmal Abb. 3).

Rs **Hilfesystem:** Zusätzlich zu der in §1.5.3 beschriebenen Methode, Hilfe zu bekommen, findet sie sich Browser-basiert auch unter „Hilfe“ → „HTML Hilfe“ (zu sehen in Abb. 2). Beachte die Empfehlungen hierzu auf Seite 8 am Ende von §1.5.3. Darüberhinaus ist unter „Hilfe“ → „Manuale (PDF)“ aber auch eine PDF-Version des Manuals “An Introduction to **R**” zu finden.

Bemerkungen:

- Der **R**-eigene Editor braucht nicht verwendet zu werden; **R**-Code kann selbstverständlich jederzeit auch direkt am **R**-Prompt eingetippt werden.

- Der Editor kann natürlich auch dazu genutzt werden, **R**-Ausgaben zu speichern, indem man “copy & paste” von der **R**-Console in den **R**-Editor (oder jeden beliebigen anderen Editor) verwendet. Dies ist natürlich keine effiziente Methode, wiederholt Ausgaben in ein wie auch immer geartetes „Berichtsdokument“ (wie z. B. einen Analysereport) zu transferieren. Hierfür gibt es leistungsfähigere Werkzeuge, von denen wir in Abschnitt 3.4 einige erwähnen (und in den Übungen auch eines vorstellen und nutzen) werden.

- Es existieren einige exzellente, durchweg kostenlose „Alternativen“ zum **R**-GUI, als da wären in alphabetischer Reihenfolge z. B.

  – Eclipse StatET, d. h. die IDE „Eclipse“ (direkt erhältlich über die Web-Site `https://eclipseide.org`) mit dem Plug-in „StatET“ (zu finden beispielsweise unter `https://marketplace.eclipse.org/content/statet-r`). Eine ältere Einführung ist Longhow Lams “Guide to Eclipse and the R plug-in StatET” (der z. B. noch unter `http://epbi-radivot.cwru.edu/EPBI473/files/week1Rbasics/R_Eclipse_StatET.pdf` zu finden ist). Ein ebenfalls älterer, zweiteiliger Blog über die Grundlagen der Nutzung von Eclipse StatET startet hier: `https://www.r-bloggers.com/eclipse-an-alternative-to-rstudio-part-1`;

  – Emacs Speaks Statistics (kurz ESS, `https://ess.r-project.org`);

  – JGR im Paket „JGR“ (sprich “jaguar”, aber in etwa ausgeprochen wie in “Mick Jagger of the Rolling Stones”, `https://www.rforge.net/JGR`);

  – R Commander im Paket „Rcmdr“ (mit näheren Informationen auf seiner Projekt-Homepage `https://socserv.socsci.mcmaster.ca/jfox/Misc/Rcmdr`);

  – RStudio von posit (`https://posit.co/download/rstudio-desktop`) mit umfangreicher Zusatzfunktionalität für die Arbeit an **R**-Projekten, insbes. wenn dabei (wiederholt) Analysereports erstellt werden sollen. (Ein unglaublich vollgepacktes “cheat sheet” (= „Spickzettel“) zu RStudio findet sich unter `https://posit.co/wp-content/uploads/2022/10/rstudio-ide-1.pdf`. Außerdem sind unter `https://posit.co/resources/cheatsheets` noch weitere, nützliche “cheat sheets” zur freien Verfügung gestellt, auf die ich z. T. gelegentlich nochmal hinweisen werde);

  – Tinn-R (`https://sourceforge.net/projects/tinn-r/`) mit einer guten, etwas älteren Einführung in dem kostenlosen e-book, das zum Download unter `https://www.rmetrics.org/ebooks-tinnr` zur Verfügung steht

  und andere. (Auf eine Informationquelle dafür ist am Ende von Abschnitt 1.1 hingewiesen worden.) Ihre Installationen und die Beschreibung Ihrer Funktionalität erfordern allerdings einen gewissen Zusatzaufwand, den ich uns hier im Skript erspare, da uns bis auf Weiteres das **R**-GUI ausreicht. (In den Übungen werden wir, wie erwähnt, jedoch RStudio nutzen.)

1.7 Installation von Zusatzpaketen

Ist ein Zusatzpaket ("add-on package" oder kurz "package") zu installieren, so lässt sich dies, falls man eine funktionierende Internetverbindung hat, i. d. R. völlig problemlos erledigen entweder

- mit Hilfe des **R**-GUIs, indem man im GUI-Menü-Thema „Pakete" den Punkt „Installiere Paket(e)" anklickt (evtl. einen – möglichst nahegelegenen – "Mirror" aussucht) und dann den Namen des gewünschten Paketes auswählt, oder
- am **R**-Prompt mit dem Befehl `install.packages`, z. B. wie in

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

Damit steht der Inhalt des Paketes, also seine Funktionen und evtl. Datensätze in der laufenden **R**-Session aber noch *nicht* zur Nutzung zur Verfügung. Dazu muss das Paket erst noch durch

```
> library("car")
```

in **R**s „Suchpfad" (der **R** durch die in der laufenden Session verfügbare Pakete-Bibliothek leitet) eingetragen werden. Es bleibt dort solange bis es entweder durch `detach()` explizit wieder ausgetragen oder die aktuelle **R**-Session beendet (und es damit automatisch aus dem Suchpfad ausgetragen) wird. D. h. insbesondere, dass das Paket beim nächsten Start von **R** erneut mit Hilfe von `library` einzuladen ist, wenn es wieder benötigt wird.

1.8 Zitieren von R und von Zusatzpaketen

Falls Sie **R** zitieren wollen oder müssen, extrahieren Sie die benötigte Information aus der resultierenden Ausgabe des Befehls

```
> citation()

To cite R in publications use:

  R Core Team (2024). R: A language and environment for statistical
  computing. R Foundation for Statistical Computing, Vienna, Austria.
  URL https://www.R-project.org/.

A BibTeX entry for LaTeX users is

  @Manual{,
    title = {R: A Language and Environment for Statistical Computing},
    author = {{R Core Team}},
    organization = {R Foundation for Statistical Computing},
    address = {Vienna, Austria},
    year = {2024},
    url = {https://www.R-project.org/},
  }

We have invested a lot of time and effort in creating R, please cite it
when using it for data analysis. See also 'citation("pkgname")' for
citing R packages.
```

Und entsprechend, wie Sie den letzten drei Zeilen entnehmen können, erhalten Sie Zitierungsinformation zu einem **R**-Paket, indem Sie `citation` den Namen des betreffenden Paketes in Anführungzeichen übergeben, wie z. B. in `citation("car")`.

1.9 Einführungsliteratur

Einführende Literatur zum Umgang, zur Statistik und zum Programmieren mit **R** (in alphabetischer Reihenfolge der Autorennamen und definitiv keine erschöpfende Aufzählung):

- Braun, W. J., Murdoch, D. J.: *A First Course in Statistical Programming with R.* 3rd ed., Cambridge University Press, 2021.
- Crawley, M. J.: *The R Book.* 2nd ed., John Wiley & Sons, Inc., 2013.
- Dalgaard, P.: *Introductory Statistics with R.* 2nd ed., Springer-Verlag, 2008.
- Everitt, B. S., Hothorn, T.: *A Handbook of Statistical Analyses Using R.* 2nd ed., Chapman & Hall/CRC, Boca Raton, 2010.
- Hatzinger, R., Hornik, K., Nagel, H., Maier, M. J.: *R: Einführung durch angewandte Statistik.* 2., aktualis. Aufl., Pearson Studium, München, 2014
- Ligges, U.: *Programmieren mit R.* 4., aktual. und überarb. Auflage, Springer-Spektrum, Berlin, 2015.
- Maindonald, J., Braun, J.: *Data Analysis and Graphics Using R. An Example-based Approach.* 3rd ed., Cambridge University Press, 2010.
- Verzani, J.: *Using R for Introductory Statistics.* 2nd revised ed. Chapman & Hall/CRC Press, Boca Raton/Florida, 2014.

Weitere, überwiegend auf `https://bookdown.org` zu findende und legal (!) frei (!) zur Verfügung stehende Online-Bücher, die gelegentlich sogar fortentwickelt werden, sind zum Beispiel:

- Grolemund, G.: *Hands-On Programming with R.* `https://rstudio-education.github.io/hopr`
- Gillespie, C., Lovelace, R.: *Efficient R programming.* 2021-03-18, `https://csgillespie.github.io/efficientR`
- Peng, R. D.: *R Programming for Data Science.* 2022-05-31, `https://bookdown.org/rdpeng/rprogdatascience`
- Wickham, H., Çetinkaya-Rundel, M., Grolemund, G.: *R for Data Science (2e).* `https://r4ds.hadley.nz`

Auf `https://bookdown.org` zu stöbern, kann sich auch für speziellere Themen zu Statistik, Data Science, Graphik oder dynamische Reportgenerierung lohnen, wobei die Bücher nicht notwendigerweise, aber überwiegend R-zentriert sind.

2 Datenobjekte: Strukturen, Attribute und elementare Operationen

Eine umfassende und präzise Sprachdefinition für **R** enthält das Dokument "The **R** Language Definition", das über die Startseite von **R**s Hilfesystem zu erreichen ist (vgl. Abb. 2 auf Seite 9). Wir geben hier nur einen kurzen Abriss über die für unsere Zwecke ausreichenden Grundlagen.

Alles in **R** ist ein *Objekt* und jedes Objekt hat eine gewisse Struktur. Alle **R**-Strukturen sind Vektoren (= geordnete, endliche Mengen), die sowohl einen (wohldefinierten) Typ, genannt *Modus* ("*mode*"), als auch eine (wohldefinierte) nicht-negative, endliche Länge haben. Daten, die verarbeitet werden sollen, müssen in Objekten zusammengefasst gespeichert werden. Wir wollen hierbei von *Datenobjekten* sprechen (um sie von anderen Objekten zu unterscheiden, die später noch eingeführt werden). Es folgt, dass auch jedes Datenobjekt einen Modus und eine Länge hat.

2.1 Zu den konzeptionellen Grundlagen

Ein Vektor enthält $n \geq 0$ eindeutig indizierbare *Elemente*. Dabei wird n die Länge des Vektors genannt und die Elemente können sehr allgemeiner Natur sein. (Vektoren der Länge 0 sind möglich und Skalare sind Vektoren der Länge 1; doch dazu später mehr.) Es gibt verschiedene Spezialisierungen für Vektoren, von denen wir die wichtigsten hier in einer Übersicht aufzählen:

2.1.1 Atomare Strukturen/Vektoren

Die einfachste Vektorform umfasst die sogenannten „atomaren Vektoren", die sich dadurch auszeichnen, dass alle Elemente eines solchen Vektors vom selben Modus wie der ganze Vektor sind. Es gibt atomare Vektoren der folgenden sechs Modi:

- `logical`: Mögliche Elemente sind die booleschen Werte TRUE (abgekürzt T) und FALSE (abgekürzt F).
- `numeric`: Hierbei handelt es sich um einen Oberbegriff der zwei **R**-internen Speicher-Modi `integer` und `double` für ganze Zahlen bzw. gewisse „Kommazahlen" (siehe das unten folgende „Beachte"):
  - `integer`: Die Elemente sind ganze Zahlen wie $0, 1, -3$. Sie erhält man nur als Ergebnisse mancher Berechnungen oder durch das Anhängen des Großbuchstabens „L" bei der expliziten Eingabe ihrer Ziffernfolgen: `0L`, `1L`, `-3L`. (Dies wird für unsere hiesigen Belange jedoch nirgends notwendig sein.)
  - `double`: Fließkommazahlen mit der Notation `3.5`, `-6.0`, `8.4e10` ($= 8.4 \times 10^{10}$), `-5e-7` ($= -5 \times 10^{-7}$). Zur Abkürzung können sämtliche folgenden Nachkomma- und führenden Vorkomma-Nullen weggelassen werden, also insbesondere auch bei ganzen Zahlen und Dezimalbrüchen zwischen 0 und 1. Zum Beispiel kann `4.0` durch `4.` abgekürzt werden und `0.35` durch `.35`.

  Beachte: `double`-Werte sind im Allgemeinen keine reellen Zahlen, sondern „nur" gewisse rationale Zahlen mit endlicher und nicht zu umfangreicher Binärdarstellung, denn irrationale Zahlen und Brüche mit zu langer oder sogar unendlicher Binärdarstellung können in einem binärsytembasierten Computer *endlicher* Größe natürlich nicht exakt gespeichert werden. Daher besitzt jedes digitale System grundsätzlich auch nur eine endliche numerische Rechengenauigkeit! Informationen zu den vom jeweiligen Computer-System abhängigen, numerischen Eigenschaften einer **R**-Implementation sind in der Variablen `.Machine` gespeichert und auf ihrer Hilfeseite (zu erreichen via `?.Machine`) erläutert.

- `complex`: Dies repräsentiert komplexe Zahlen $a + b \cdot i$, wobei a und b Zahlen des Modus' `numeric` sind und zwischen b und dem Symbol `i` für die imaginäre Einheit $i = \sqrt{-1}$ kein Leerzeichen stehen darf. Bsp.: `3 + 7i` oder `-1.5 + 0.8i`.
- `character`: Hiermit werden (nahezu) beliebige Zeichenketten gespeichert. Sie werden durch Paare von `"` oder `'` begrenzt, wie z. B. `"Otto"` und `'auto2002'`.
- `raw`: Vektoren dieses Modus enthalten „rohe" Bytes (und werden wir nirgends benötigen).

Der Modus eines Objektes `x` kann mit der Funktion `mode` durch `mode(x)` abgefragt werden.

Sprechweise: Einen Vektor des Modus' `numeric` nennen wir kurz `numeric`-Vektor. Für die anderen Vektormodi gilt Analoges.

2.1.2 Rekursive Strukturen/Vektoren

Eine besondere Form von Vektoren sind die sogenannten „rekursiven" Vektoren: Ihre Elemente sind Objekte beliebiger Modi. Dieser Vektormodus heißt `list`. Ein `list`-Objekt (kurz: eine Liste) ist demnach ein Vektor, dessen Elemente beliebige Objekte verschiedener Modi sein können (also insbesondere selbst wieder Listen). Listen sind mit die wichtigsten (Daten-)Objekte in **R** und werden uns häufig begegnen.

Eine weitere wichtige rekursive Struktur heißt `function`. Sie erlaubt – wie der Name sagt – die Implementation (neuer) benutzerspezifischer Funktionen, die als **R**-Objekte im workspace gespeichert werden können und so **R** quasi erweitern.

2.1.3 Weitere Objekttypen und Attribute

Außer den grundlegenden Eigenschaften Modus und Länge (den sogenannten „intrinsischen Attributen") eines Objektes gibt es noch weitere „Attribute", die Objekten gewisse Struktureigenschaften verleihen. In **R** stehen neben den schon erwähnten Vektoren und Listen viele weitere Objekttypen zur Verfügung, die durch gewisse Attribute „generiert" werden. Beispiele:

- `array` bzw. `matrix`: Sie dienen der Realisierung mehrfach indizierter Variablen und haben ein Attribut „Dimension" (`dim`) und optional ein Attribut „Dimensionsnamen" (`dimnames`).
- `factor` bzw. `ordered` (`factor`): Dies sind intern `integer`-Vektoren, deren Elemente als Ausprägungen einer nominal- bzw. einer ordinal-skalierten Variablen interpretiert und als Zeichenketten (aber ohne Hochkommata) dargestellt werden. Sie haben ein Attribut `levels`, das alle möglichen Ausprägungen der Variablen aufzählt und im Fall `ordered` gleichzeitig die (Rang-)Ordnung dieser Ausprägungen auf der Ordinalskala beschreibt.

2.1.4 Das Attribut „Klasse" ("class")

Ein Attribut der besonderen Art ist die „Klasse" (`class`) eines Objektes. Sie wird für die objektorientierte Programmierung in **R** verwendet. Die Klasse eines Objektes entscheidet häufig darüber, wie gewisse Funktionen mit ihnen „umgehen". Viele **R**-Objekte haben ein `class`-Attribut. Falls ein Objekt kein (explizites) `class`-Attribut hat, so besitzt es stets eine implizite Klasse: Es ist dies `matrix`, `array` oder das Ergebnis von `mode(x)`.

Eine spezielle Klasse, die eine Struktur gewissermaßen *zwischen* `list` und `matrix` implementiert, ist die Klasse `data.frame`. Sie dient der strukturierten Zusammenfassung von, sagen wir, p Vektoren gleicher Länge n, aber verschiedener Modi, in ein $(n \times p)$-matrixförmiges Schema. "Data frames" werden in **R** häufig im Zusammenhang mit dem Anpassen statistischer Modelle verwendet. Dabei werden die Vektoren (= Spalten des Schemas) als Variablen interpretiert und die Zeilen des Schemas als die p-dimensionalen Datenvektoren der n Untersuchungseinheiten.

Auf die oben genannten Datenobjekte und anderen Strukturen (außer `complex`) gehen wir in den folgenden Abschnitten detaillierter ein.

2.2 numeric-Vektoren: Erzeugung und elementare Operationen

Anhand einfacher Beispiele wollen wir Methoden zur Erzeugung von und elementare Operationen für numeric-Vektoren vorstellen. Durch die Zuweisungsanweisung

```
> hoehe <- c(160, 140, 155)
```

wird dem Objekt hoehe ein Vektor mit den Elementen 160, 140 und 155 zugewiesen, indem die Funktion c (vom englischen "concatenation", d. h. „Verkettung") diese Werte zu einem Vektor zusammenfasst. Da es sich bei den Werten auf der rechten Seite von <- nur um integer- und damit um numeric-Größen handelt, wird hoehe (automatisch) zu einem numeric-Vektor. Skalare werden als Vektoren mit nur *einem* Element aufgefasst und können ohne die Funktion c zugewiesen werden:

```
> eine.weitere.hoehe <- 175
```

Mittels der Funktion c lassen sich Vektoren einfach aneinanderhängen:

```
> c(hoehe, eine.weitere.hoehe)
[1] 160 140 155 175
```

Eine Zuweisung, auf deren linker Seite ein Objekt steht, welches auch auf der rechten Seite auftaucht, wie in

```
> (hoehe <- c(hoehe, eine.weitere.hoehe))
[1] 160 140 155 175
```

bewirkt, dass zunächst der Ausdruck rechts vom <- ausgewertet wird und erst dann das Resultat zugewiesen wird. Der ursprüngliche Wert des Objektes wird durch den neuen überschrieben. (Memo: Zur Wirkungsweise der äußeren Klammern siehe Seite 4.)

Die Funktion length liefert die Anzahl der Elemente eines Vektors, also seine Länge:

```
> length(hoehe)
[1] 4
```

Obiges gilt analog für atomaren Vektoren der anderen Modi.

2.2.1 Beispiele regelmäßiger Zahlenfolgen: seq und rep

Gewisse, häufig benötigte Vektoren, deren Elemente spezielle Zahlenfolgen bilden, lassen sich in **R** recht einfach erzeugen. Hierzu stehen die Funktionen seq und rep zur Verfügung.

Die Funktion seq (vom englischen "sequence") bietet die Möglichkeit, regelmäßige Zahlenfolgen zu erzeugen. Sie hat mehrere Argumente, von denen die ersten vier sehr suggestiv from, to, by und length lauten. In einem Aufruf von seq dürfen (verständlicherweise) nicht alle vier Argumente gleichzeitig beliebig spezifiziert werden, sondern höchstens drei von ihnen. Der vierte Wert wird von **R** aus den drei angegebenen passend hergeleitet. Ein paar Beispiele sollen ihre Verwendung demonstrieren:

```
> seq(from = -2, to = 8, by = 1)
[1] -2 -1  0  1  2  3  4  5  6  7  8

> seq(from = -2, to = 8, by = 0.8)
[1] -2.0 -1.2 -0.4  0.4  1.2  2.0  2.8  3.6  4.4  5.2  6.0  6.8  7.6

> seq(from = -2, to = 8)
[1] -2 -1  0  1  2  3  4  5  6  7  8
```

```
> seq(from = 7, to = -1)
[1]  7  6  5  4  3  2  1  0 -1

> seq(from = -5, length = 12, by = 0.2)
[1] -5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 -3.6 -3.4 -3.2 -3.0 -2.8

> seq(from = -5.5, to = 3.5, length = 12)
[1] -5.5000000 -4.6818182 -3.8636364 -3.0454545 -2.2272727 -1.4090909
[7] -0.5909091  0.2272727  1.0454545  1.8636364  2.6818182  3.5000000

> seq(to = -5, length = 12, by = 0.2)
[1] -7.2 -7.0 -6.8 -6.6 -6.4 -6.2 -6.0 -5.8 -5.6 -5.4 -5.2 -5.0
```

Ist die Differenz zwischen Endpunkt `to` und Startwert `from` kein ganzzahliges Vielfaches der Schrittweite `by`, so endet die Folge beim letzten Wert *vor* `to`. Fehlen die Angaben einer Schrittweite `by` und einer Folgenlänge `length`, wird `by` automatisch auf 1 oder -1 gesetzt. Aus drei angegebenen Argumentwerten wird der fehlende vierte automatisch passend bestimmt.

Bemerkung: Argumentwerte können auch über die *Position* im Funktionsaufruf gemäß `seq( from, to, by, length)` übergeben werden, was zahlreiche Abkürzungsmöglichkeiten bietet (aber den/die BenutzerIn für die Korrektheit der Funktionsaufrufe verantwortlich macht und einige Gefahren birgt):

```
> seq(-2, 8)
[1] -2 -1  0  1  2  3  4  5  6  7  8

> seq(-2, 8, 0.8)
[1] -2.0 -1.2 -0.4  0.4  1.2  2.0  2.8  3.6  4.4  5.2  6.0  6.8  7.6
```

Zu den Details der Argumentübergabemöglichkeiten in Funktionsaufrufen gehen wir im Abschnitt 6.5 „Spezifizierung von Funktionsargumenten“ ein.

Für Zahlenfolgen, deren Schrittweite 1 ist, gibt es noch eine weitere Abkürzung, den Doppelpunkt-Operator:

```
> 2:11
[1]  2  3  4  5  6  7  8  9 10 11

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

> -(1:10)
[1]  -1  -2  -3  -4  -5  -6  -7  -8  -9 -10

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

Offenbar liefert er dasselbe wie die Verwendung von lediglich den zwei Argumenten `from` und `to` von `seq`. An Obigem zeigt sich auch schon, dass in Ausdrücken auf die Priorität der verwendeten Operatoren (hier das unäre Minus und der Doppelpunkt-Operator) zu achten ist und nötigenfalls Klammern zu verwenden sind, um Prioritäten zu verändern.

Mit der Funktion `rep` (von “repeat”) lassen sich Vektoren erzeugen, deren Elemente aus Strukturen entstehen, die (möglicherweise auf komplizierte Art und Weise) wiederholt werden. Die Argumente von `rep` lauten `x`, `times`, `length.out` und `each`, wobei `x` den Vektor der zu replizierenden Elemente erhält und die anderen Argumente spezifizieren, wie dies zu geschehen hat. In folgendem Beispiel der einfachsten Form

```
> rep(x = 1:3, times = 4)
[1] 1 2 3 1 2 3 1 2 3 1 2 3
```

werden vier hintereinandergehängte Kopien des Vektors `1:3` erzeugt.

Wird dem `times`-Argument ein Vektor der gleichen Länge wie `x` übergeben, dann erwirkt jedes Element des `times`-Arguments eine entsprechende Vervielfältigung des korrespondierenden Elements von `x`:

```
> rep(x = c(-7, 9), times = c(3, 5))
[1] -7 -7 -7  9  9  9  9  9
```

Das Argument `length.out` gibt an, wie lang der Ergebnisvektor sein soll; die Elemente von `x` werden so oft zyklisch repliziert, bis diese Länge erreicht ist:

```
> rep(x = -1:1, length.out = 11)
[1] -1  0  1 -1  0  1 -1  0  1 -1  0
```

Mit `each` wird angegeben, wie oft *jedes* Element von `x` wiederholt werden soll, was, falls noch andere Argumente angegeben sind, stets *vor* den übrigen Replikationsoperationen gemacht wird:

```
> rep(x = c(-1, 1), each = 3)
[1] -1 -1 -1  1  1  1

> rep(x = c(-1, 0), times = 3, each = 2)
[1] -1 -1  0  0 -1 -1  0  0 -1 -1  0  0

> rep(x = c(-1, 0), length.out = 13, each = 5)
[1] -1 -1 -1 -1 -1  0  0  0  0  0 -1 -1 -1
```

2.2.2 Elementare Vektoroperationen

Wir fassen obige Beispiele in folgender Übersicht zusammen und zählen weitere Funktionen und Operationen auf, welche in **R** für Vektoren zur Verfügung stehen:

R-Befehl und Resultat	Bedeutung/Bemerkung
`> y <- c(4, 1, 4, 8, 5, 3)` `> z <- c(y, 9:3)` `> z <- c(z, -12, rev(z))` *(Überschreibt z ohne nachzufragen!)*	Die Argumente von `c`, egal ob Konstanten, Vektoren oder Ausdrücke, die einen Vektor liefern, werden zu einem Vektor „c“ombiniert und durch `<-` als Objekt unter dem Namen `y` bzw. `z` abgespeichert (vgl. Anfang von Abschnitt 2.2).
`> append(x, values, after)` `....`	Fügt die Elemente des an `values` übergebenen Vektors nach dem durch `after` spezifizierten Index des Vektors `x` ein.
`> length(y)` `[1] 6`	Länge des Vektors `y` (= Anzahl der Vektorelemente); kann 0 sein, wie wir sehen werden.
`> seq(from, to, by, length, along)` `....` `> seq(along = y)` `[1] 1 2 3 4 5 6`	Für Beispiele bzgl. der ersten vier Argumente siehe oben. `seq(along = y)` liefert dasselbe wie `1:length(y)`, außer wenn `y` die Länge 0 hat, dann liefert es einen `integer`-Vektor der Länge 0.
`> rep(x, times, length.out, each)` `....`	Für Beispiele siehe oben.
`> rev(y)` `[1] 3 5 8 4 1 4`	Liefert den Vektor, der die Elemente des Argumentvektors in umgekehrter Reihenfolge enthält.

`> unique(y)` `[1] 4 1 8 5 3`	Liefert die Elemente des Eingabevektors ohne Wiederholungen.
`> sort(y)` `[1] 1 3 4 4 5 8`	Sortiert die Elemente aufsteigend (je nach Modus z. B. numerisch oder lexikografisch) und liefert also die „Ordnungsstatistiken“.
`> rank(y)` `[1] 3.5 1.0 3.5 6.0 5.0 2.0`	Bildet den zum Eingabevektor gehörenden Rangvektor. Bindungen liefern (per Voreinstellung) mittlere Ränge (Engl.: “midranks”).
`> order(y)` `[1] 2 6 1 3 5 4`	Liefert den Vektor der *Indizes* der Eingabedaten für deren aufsteigende Sortierung: Das erste Element von `order(y)` ist der Index des kleinsten Wertes in `y`, das zweite der des zweitkleinsten usw.

Empfehlung: Viel mehr und detailliertere Informationen zu den einzelnen Funktionen liefert jeweils ihre Hilfeseite.
Hinweis: Auf den Zugriff auf einzelne Elemente eines Vektors (ihre Indizierung) gehen wir in Abschnitt 2.6 ein.

2.3 Arithmetik und Funktionen für `numeric`-Vektoren

Die **R**-Arithmetik und viele andere Funktionen operieren für Vektoren *elementweise*, also *vektorisiert*, was für AnwenderInnen anderer Programmiersprachen (und den darin hierfür nötigten Schleifen) gelegentlich etwas gewöhnungsbedürftig ist. Das Konzept ist jedoch suggestiv, komfortabel und leistungsstark. Als (banales) Beispiel eine Flächenberechnung vierer Rechtecke:

```
> breite <- c(25, 17, 34, 6);   breite * hoehe   # mit hoehe von S. 17 (Mitte)
[1] 4000 2380 5270 1050
```

Der „Clou“ ist, dass Vektoren in einem solchen Ausdruck nicht die gleiche Länge zu haben brauchen. Die Elemente der kürzeren Vektoren darin werden dann bis zur Länge des längsten Vektors zyklisch (möglicherweise unvollständig) repliziert. Dies geschieht ohne Warnung, wenn die kleineren Vektorlängen Teiler der größten sind! Bei unterschiedlichen Längen der beteiligten Vektoren hat das Resultat also die Länge des längsten Vektors in diesem Ausdruck. Dies gilt auch für Skalare (= Vektoren der Länge 1). Hierzu als weiteres (banales) Beispiel die Volumenberechnung für vier Quader, die verschiedene Höhen und Breiten, aber nur zwei alternierende Tiefen (2 und 3) haben, und zu denen jeweils ein konstantes „Extravolumen“ (12) hinzukommt:

```
> breite * hoehe * c(2, 3) + 12
[1]  8012  7152 10552  3162
```

In den folgenden Tabellen listen wir verschiedene Funktionen und Operationen auf, welche in **R** für `numeric`-Vektoren zur Verfügung stehen und teilweise speziell für die `integer`-Vektoren die Ganzzahlarithmetik realisieren. Hierzu verwenden wir drei Beispielvektoren:

```
> x <- c(-0.3691, 0.3537, -1.0119, -2.6563, NA, 11.3351)
> y <- c(4, 1, 4, 8, 5, 3);    z <- c(2, 3)
```

`NA` steht dabei für “not available” und bedeutet, dass der Wert dieses Elements schlicht fehlt, also im statistischen Sinn ein “missing value” ist. Im Allgemeinen liefert eine beliebige Operation, in der irgendwo ein `NA` auftaucht, insgesamt den Wert `NA` zurück. Für manche elementweisen Operationen ist es jedoch sinnvoll, wenn sie im Resultatvektor lediglich an denjenigen Stellen ein `NA` ergeben, wo sich in einem der Eingabevektoren ein `NA` befand.

Es gibt noch eine weitere Sorte von “not available”, und zwar im Sinne von „numerisch nicht definiert“, weil “not a number”: `NaN`. Sie ist das Resultat von Operationen wie $0/0$ oder $\infty - \infty$, wobei ∞ (= Unendlich) in **R** durch das Objekt `Inf` implementiert ist. Für mehr Details zu diesen „besonderen“ Werten siehe Abschnitt 2.11, Seite 65.

2.3.1 Elementweise Vektoroperationen: Rechnen, runden, formatieren

Für die Verknüpfung arithmetischer Operationen gelten in der Mathematik gewisse Operatorpräzedenzen á la „Punkt- vor Strichrechnung“ oder bei Gleichwertigkeit „von links nach rechts“, die von **R** (natürlich) berücksichtigt werden. Allerdings treten in **R** arithmetische Operatoren häufig in Kombination mit anderen, „nicht-arithmetischen“ Operatoren auf. Hier ist die Operatorpräzedenz nicht unbedingt offensichtlich. Für Klarheit sorgt hier die Hilfeseite, zu der `?Syntax` führt. Im Übrigen ist die Verwendung von Paaren runder Klammern stets möglich, um eine gewisse, evtl. von der üblichen Ordnung abweichende Auswertungsreihenfolge zu erzwingen (oder einfach nur für sich selbst sichtbar sicherzustellen).

Die Hilfeseite zu den arithmetischen Operatoren erreicht man durch `?Arithmetic`.

`> y + z` `[1] 6 4 6 11 7 6` `> y + x` `[1] 3.6309 .... NA 14.3351`	Elementweise Addition (mit zyklischer Replikation des kürzeren Vektors und mit rein elementweiser Beachtung des NA-Status').
`> y - z` `[1] 2 -2 2 5 3 0`	Elementweise Subtraktion.
`> y * z` `[1] 8 3 8 24 10 9`	Elementweise Multiplikation.
`> y / z` `[1] 2.0000 0.3333 2.0000 ....`	Elementweise Division. (Beachte, was eine Division durch 0 (Null) liefert! Details hierzu in §2.11.2.)
`> y^z` `[1] 16 1 16 512 25 27`	Elementweise Exponentiation (siehe auch das „Beachte“ am Ende dieser Tabelle).
`> y %/% z` `[1] 2 0 2 2 2 1`	Elementweise ganzzahlige Division.
`> y %% z` `[1] 0 1 0 2 1 0`	Elementweise Modulo-Funktion, d. h. Rest bei ganzzahliger Division.
`> pmax(x, y)` `[1] 4.0000 1.0000 4.0000` `[4] 8.0000 NA 11.3351` `> pmin(x, y)` `[1] -0.3691 0.3537 -1.0119` `[4] -2.6563 NA 3.0000`	`pmax` bzw. `pmin` liefert das elementweise (oder auch „parallele“) Maximum bzw. Minimum mehrerer Vektoren (und nicht nur zweier wie hier im Beispiel).
`> pmin(x, y, na.rm = TRUE)` `[1] -0.3691 0.3537 -1.0119` `[4] -2.6563 5.0000 3.0000`	Mit dem Argument `na.rm = TRUE` werden NAs ignoriert. Ausnahme: Ist in allen Vektoren dasselbe Element NA, so ist auch im Resultatvektor dieses Element NA.
`> ceiling(x)` `[1] 0 1 -1 -2 NA 12`	Rundet elementweise zur nächsten ganzen Zahl *auf*.
`> floor(x)` `[1] -1 0 -2 -3 NA 11`	Rundet elementweise zur nächsten ganzen Zahl *ab*, ist also die Gaußklammer.
`> trunc(x)` `[1] 0 0 -1 -2 NA 11`	Rundet elementweise zur nächsten ganzen Zahl *in Richtung Null*, d. h., liefert den ganzzahligen Anteil.
`> round(x)` `[1] 0 0 -1 -3 NA 11` `> round(c(2.5, 3.5))` `[1] 2 4`	Rundet elementweise auf die nächste ganze Zahl, wobei Werte der Art „$k.5$“ auf die nächste gerade ganze Zahl gerundet werden.
`> round(x, digits = 1)` `[1] -0.4 0.4 -1.0 -2.7 NA 11.3` `> round(c(1.25, 1.35), 1)` `[1] 1.2 1.4`	Das zweite (optionale) Argument `digits` gibt, falls positiv, die Zahl der Nachkommastellen, falls negativ, die Zehnerstelle vor dem Komma an, auf die gerundet werden soll. Werte der Art „...$k5$...“ werden an der Stelle k auf die nächste gerade ganze Ziffer gerundet.

`> signif(x, digits = 2)` `[1] -0.37  0.35 -1.00 -2.70` `[5]    NA 11.00`	Rundet auf die für `digits` angegebene Gesamtzahl an signifikanten Stellen (wobei Nullen lediglich für ein einheitliches Druckformat angehängt werden).
`> print(x, digits = 2)` `[1] -0.37  0.35 -1.01 -2.66` `[5]    NA 11.34`	Druckt `x` mit der für `digits` angegebenen Anzahl an Nachkommastellen (gerundet) in einheitlichem Druckformat.
`> format(x, digits = 2)` `[1] "-0.37" " 0.35" "-1.01"` `[4] "-2.66" "   NA" "11.34"`	Erzeugt eine `character`-Darstellung des `numeric`-Vektors `x` in einheitlich langer, rechtsbündiger Formatierung der Elemente von `x` mit der durch `digits` angegebenen Zahl an (gerundeten) Nachkommastellen. Dazu wird mit führenden Leerzeichen aufgefüllt.
`> format(y, nsmall = 2)` `[1] "4.00" "1.00" "4.00" "8.00"` `[5] "5.00" "3.00"`	`nsmall` gibt die Mindestanzahl an Ziffern rechts vom Dezimalpunkt an. Erlaubt: $0 \leq$ `nsmall` ≤ 20. Es wird rechts mit Nullen aufgefüllt.
`> format((-2)^c(3,10,21,32),` `+ scientific = TRUE, digits = 4)` `[1] "-8.000e+00" " 1.024e+03"` `[3] "-2.097e+06" " 4.295e+09"`	`scientific = TRUE` veranlasst die Darstellung in wissenschaftlicher Notation, wobei `digits` wie eben funktioniert. (Viel mehr Informationen liefert die Hilfe. Nützlich ist auch `formatC`.)

Beachte: Die Berechnung der elementweisen Exponentiation x^n für ganzzahlige Werte von n mit $|n| \geq 3$ ist effizienter durch die Verwendung des expliziten Produkts $x * \cdots * x$ zu erreichen. Insbesondere in sehr rechenintensiven Simulationen, z. B. mit sehr langem `x`, sollte man also beispielsweise `x^3` und `x^4` explizit als `x * x * x` bzw. `x * x * x * x` programmieren. Dies *kann* den Laufzeitaufwand reduzieren und die Geduld des/der „Simulanten/in“ ein wenig entlasten. Vorsicht jedoch bei der Verwendung dieser Technik bei großen, „echten“ `integer`-Werten (siehe in §2.1.1 den Spiegelstrich zu `integer`) wie z. B. bei `2000L * 2000L * 2000L`, was aufgrund eines „Ganzzahlüberlaufs“ `NA` liefert (im Gegensatz zu `2000L^3`) und nur eine Warn-, aber keine Fehlermeldung ausgibt!

2.3.2 Zusammenfassende und sequenzielle Vektoroperationen: Summen, Produkte, Extrema

Memo: `x` und `y` sind die Objekte von Seite 20.

`> sum(y)` `[1] 25`	Summe der Vektorelemente.
`> sum(x)` `[1] NA` `> sum(x, na.rm = TRUE)` `[1] 7.6515`	Konsequenz eines NA-Elements hierbei. Aber wenn das logische Argument `na.rm` auf TRUE gesetzt wird, führt dies zum Ausschluss der NA-Elemente *vor* der Summenbildung.
`> prod(y)` `[1] 1920`	Produkt der Vektorelemente. Argument `na.rm` steht auch hier zur Verfügung.
`> cumsum(y)` `[1] 4 5 9 17 22 25`	Kumulative Summen $(y_1, y_1 + y_2, \ldots, y_1 + \cdots + y_n)$,
`> cumprod(y)` `[1] 4 4 16 128 640 1920`	kumulative Produkte (analog),
`> cummax(y)` `[1] 4 4 4 8 8 8`	kumulative Maxima $(y_1, \max\{y_1, y_2\}, \ldots, \max\{y_1, \ldots, y_n\})$
`> cummin(y)` `[1] 4 1 1 1 1 1`	bzw. kumulative Minima (analog) der Vektorelemente. (Das Argument `na.rm` steht **nicht** zur Verfügung.)
`> diff(y)` `[1] -3  3  4 -3 -2`	Sukzessive Differenzen der Vektorelemente: Element i ist der Wert $y_{i+1} - y_i$ (d. h. der Zuwachs von y_i auf y_{i+1}).

`> diff(y, lag = 2)` `[1]  0  7  1 -5`	Sukzessive Differenzen der Vektorelemente, die *lag* Elemente voneinander entfernt sind: Element i ist der Wert $y_{i+lag} - y_i$.

2.3.3 "Summary statistics" (summary etc.)

Statistikspezifische zusammenfassende Vektoroperationen – im Englischen auch "summary statistics" genannt – wollen wir separat aufführen: Die hierbei verwendeten Funktionen zur Bestimmung von Maximum, Minimum, Spannweite, Mittelwert, Median, Varianz und Quantilen eines Datensatzes bzw. der Korrelation zweier Datensätze würden im Fall fehlender Werte (NAs) in den Daten ebenfalls NA zurückliefern oder die Berechnung mit einer Fehlermeldung abbrechen. Dies kann durch das Argument `na.rm` geändert werden.

Memo: `x` und `y` stammen von Seite 20.

`> max(x)` `[1] NA` `> max(x, na.rm = TRUE)` `[1] 11.3351` `> min(x, na.rm = TRUE)` `[1] -2.6563`	Wie erwartet, führt ein NA bei `max` etc. ebenfalls zu NA als Resultat. Das Argument `na.rm = TRUE` erzwingt, dass alle NAs in dem Vektor `x` ignoriert werden, *bevor* die jeweilige Funktion angewendet wird. Die Voreinstellung ist `na.rm = FALSE`.
`> which.max(x)` `[1] 6` `> which.min(x)` `[1] 4`	Liefert den Index des ersten (!) Auftretens des Maximums (bzw. Minimums) in den Elementen eines Vektors. NAs im Vektor werden ignoriert.
`> range(x, na.rm = TRUE)` `[1] -2.6563 11.3351`	Minimum und Maximum der Werte in `x` (ohne NAs).
`> mean(x, na.rm = TRUE)` `[1] 1.5303` `> mean(x, trim = 0.2,` `+ na.rm = TRUE)` `[1] -0.3424333`	Arithmetisches Mittel der Nicht-NA-Werte in `x`. Das Argument `trim` mit Werten in $[0, 0.5]$ spezifiziert, welcher Anteil der sortierten Nicht-NA-Werte in `x` am unteren und oberen Ende *jeweils* weggelassen (getrimmt) werden soll.
`> median(x, na.rm = TRUE)` `[1] -0.3691`	Der Median der Nicht-NA-Werte in `x`. Zu Details seiner Implementation siehe `?median`.
`> quantile(x, na.rm = TRUE)` `     0%     25%     50%     75%` `-2.6563 -1.0119 -0.3691  0.3537` `   100%` `11.3351` `> quantile(x, na.rm = TRUE,` `+ probs = c(0, 0.2, 0.9))` `      0%       20%     90%` `-2.65630 -1.34078  6.94254`	Empirische Quantile der Werte in `x`. Die Voreinstellung liefert Minimum, unteres Quartil, Median, oberes Quartil und Maximum der `x`-Elemente. Das Argument `probs` erlaubt die Bestimmung jedes beliebigen Quantils. Es sind neun (!) verschiedene Quantildefinitionen und die entsprechenden Algorithmen zu ihrer Bestimmung über das nicht gezeigte Argument `type` wählbar; siehe die Hilfeseite. Per Voreinstellung (`type = 7`) wird zwischen den Ordnungsstatistiken von `x` linear interpoliert, wobei $x_{i:n}$ als `(i-1)/(length(x)-1)`-Quantil angenommen wird. Für `type = 1` wird als Quantilfunktion die Inverse der empirischen Verteilungsfunktion F_n genutzt: $F_n^{-1}(p) = \inf\{x : F_n(x) \geq p\}$, $0 \leq p \leq 1$.
`> summary(x)` `    Min. 1st Qu.  Median    Mean` `-2.6560 -1.0120 -0.3691  1.5300` `3rd Qu.    Max.    NA's` `0.3537 11.3400  1.0000`	Das Resultat von `summary` angewendet auf einen `numeric`-Vektor `x` ist in Gestalt einer „6-Zahlenzusammenfassung“ eine Auswahl von Resultaten obiger Funktionen, ggf. nach Zählung sowie Elimination der NA-Elemente.

`> var(x, na.rm = TRUE)` `[1] 31.27915` `> sd(x, na.rm = TRUE)` `[1] 5.592777`	Die Funktionen `var` und `sd` zur Berechnung der empirischen Varianz bzw. Standardabweichung, d. h. der Stichprobenvarianz bzw. -standardabweichung der Elemente eines Vektors besitzen das Argument `na.rm`. Es muss zum Ausschluss von NA-Elementen auf `TRUE` gesetzt werden.
`> cov(x, y, use = "complete.obs")` `[1] -5.75535` `> var(x, y, use = "complete.obs")` *... (Dasselbe wie bei* ***var****)* `> cor(x, y, use = "complete.obs")` `[1] -0.4036338`	`cov` und `var` bzw. `cor` zur Bestimmung der empirischen Kovarianz bzw. empirischen Pearsonschen Korrelation der Elemente zweier Vektoren besitzen das Argument `use`. Es gibt die Behandlung von NAs an. `use = "complete.obs"` sorgt dafür, dass, wenn eines von zwei i-ten Elementen NA ist, beide ausgeschlossen werden. (Für andere Möglichkeiten siehe die Hilfeseite.)

Memo 1: Zu unabhängig und identisch verteilten Zufallsvariablen $X_1, \ldots, X_n$ mit Erwartungswert $\mathbb{E}[X_i] \equiv \mu$ und Varianz $\mathrm{Var}(X_i) = \mathbb{E}[(X_i - \mu)^2] \equiv \sigma^2 > 0$ ist ihre Stichprobenvarianz $\hat{\sigma}^2$ definiert durch $\hat{\sigma}^2 := \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X}_\cdot)^2$. Es gelten zwei mathematisch gelegentlich nützliche, alternative Darstellungen für die Stichprobenvarianz: $\hat{\sigma}^2 = \frac{1}{n-1}\left(\sum_{i=1}^{n} X_i^2 - n\bar{X}_\cdot^2\right) = \frac{1}{n(n-1)}\sum_{1\le i,j\le n}\frac{1}{2}(X_i - X_j)^2$. Sie ist ein *erwartungstreuer* Schätzer für σ^2, d. h. $\mathbb{E}[\hat{\sigma}^2] = \sigma^2$, wobei übrigens $\mathrm{Var}(\hat{\sigma}^2) = \frac{\mu_4}{n} - \frac{\sigma^4(n-3)}{n(n-1)}$ mit $\mu_4 \equiv \mathbb{E}[(X_i - \mu)^4]$.

Memo 2: Wie ist der theoretische Pearsonsche Korrelationskoeffizient für zwei Zufallsvariablen X und Y definiert und was „misst“ er? Wie ist sein empirisches Pendant für zwei (Daten-)Vektoren $(x_1, \ldots, x_n)$ und $(y_1, \ldots, y_n)$ definiert und was „misst“ es?

Beachte im Vorgriff auf Kapitel 2.8 „Matrizen: Erzeugung, Indizierung, Modifikation und Operationen“: `sum`, `prod`, `max`, `min`, `range`, `mean`, `median`, `quantile`, `sd` und `summary` können auch direkt auf Matrizen angewendet werden. Sie wirken dann jedoch nicht spalten- oder zeilenweise, sondern auf die als Vektor aufgefasste Gesamtheit der Matrixeinträge. Die Funktionen `cov` und `var` bzw. `cor` ergeben allerdings die empirische Kovarianz- bzw. Korrelations*matrix* der Spaltenvektoren.

2.3.4 Mathematische Funktionen

Selbstverständlich stehen auch eher „rein mathematische“ Funktionen zur Verfügung, die ebenfalls elementweise auf `numeric`-Vektoren angewendet werden. Eine Auswahl:

Name(n)			Funktion(en)
`sqrt`			Quadratwurzel
`abs`			Absolutbetrag
`sin` `sinpi`	`cos` `cospi`	`tan` `tanpi`	(„Normale“ und spezielle) Trigonometrische Funktionen
`asin`	`acos`	`atan`	Inverse trigonometrische Funktionen
`sinh`	`cosh`	`tanh`	Hyperbolische trigonometrische Funktionen
`asinh`	`acosh`	`atanh`	Inverse hyperbolische trigonometrische Funktionen

`exp`	`log`	Exponentialfunktion und natürlicher Logarithmus
`expm1`	`log1p`	Spezialfunktionen zur präzisen Berechnung von $e^x - 1$ bzw. $\log(1+x)$ jeweils für $\lvert x\rvert \ll 1$.
`log10`	`logb`	Dekadischer Logarithmus und Logarithmus zu freier Basis
`beta`	`lbeta`	Betafunktion und ihr natürlicher Logarithmus
`gamma`	`lgamma`	Gammafunktion Γ und ihr natürlicher Logarithmus
`factorial`	`lfactorial`	Ist $\Gamma(x+1)$ für $x \in \mathbb{R}$ und die Fakultätsfunktion $x!$, falls $x \in \mathbb{N}_0$, bzw. ihr natürlicher Logarithmus
`choose`	`lchoose`	Binomialkoeffizienten bzw. ihr natürlicher Logarithmus

Für weitere Informationen über die aufgeführten Funktionen siehe die Hilfeseiten direkt via ihre Namen wie z. B. `?sqrt` und `?log` oder teilweise auch via `?Trig` oder `?Special`.

Beachte:

- $\Gamma(x) = (x-1)!$, falls x eine natürliche Zahl ist.
- Um die Basis der Logarithmus-Funktion zu wählen, ist für `logb` das Argument `base` da. Beispielsweise liefert `logb(x, base = 2)` den binären Logarithmus. Natürlich ist `logb(x, base = 10)` gerade gleich `log10(x)`.
- π ist unter dem Namen `pi` verfügbar und e ist `exp(1)`.
- Für aufwändigere mathematische Simulationen kann es sich hinsichtlich der Rechenzeit lohnen, die wiederholte Anwendung obiger Funktionen auf denselben oder einen leicht variierten Vektor zu vermeiden. Beispiel: Die Berechnung von `exp(x) + exp(-x)` für einen langen `numeric`-Vektor `x` ist effizienter durch `z <- exp(x); z + 1/z` zu erzielen, weil die Kehrwertbildung eine „einfachere“ Funktion ist als die Exponentialfunktion.
- Zum äußerst beachtenswerten und interessanten Thema Rechengenauigkeit im Sinne von numerischer Präzision finden sich in der Antwort zur R-FAQ 7.31 wichtige Informationen. Das für sich allein schon interessante Thema ist in der Arbeit [44, Goldberg (1991)], die auch auf der Hilfeseite `?Arithmetic` zitiert ist, deutlich weiter ausgeführt.

  Bei Interesse an der digitalen Darstellung von Gleit- oder Fließkommazahlen siehe z. B. `https://www.h-schmidt.net/FloatConverter/IEEE754de.html` und für vielerlei Details `https://de.wikipedia.org/wiki/IEEE_754` zu ihrer Darstellung gemäß IEEE 754.
- Das Paket `Rmpfr` stellt Methoden für arithmetische (einschließlich transzendenter) Funktionen zur Verfügung, die mit “Multiple Precision Floating-Point Reliable”-Zahlen (= “mpfr”-Zahlen) arbeiten, was in etwa bedeutet, dass Berechnungen in beliebiger Genauigkeit möglich sind. `Rmpfr` nutzt das Paket `gmp`, das “Multiple Precision Arithmetic”, also das Entsprechende für große natürliche und rationale Zahlen bietet.
- Seien Sie sich trotz der Implementation von exakten Methoden wie sie z. B. in den vorgenannten Paketen existieren stets gewisser, von **R** völlig unabhängiger, fundamentaler Effekte, wie sie auch in `https://xkcd.com/2295` als “Garbage Math” karikiert werden, bewusst!
- `choose(n, k)` liefert die *Anzahl* aller k-elementigen Teilmengen einer n-elementigen Grundmenge. Die explizite Erzeugung dieser Teilmengen hingegen – und in gewissem Umfang deren Weiterverarbeitung – ermöglicht die Funktion `combn` (siehe ihre Hilfeseite).
- Das “Base R cheat sheet”, zu finden unter `https://posit.co/resources/cheatsheets` oder direkt bei `http://github.com/rstudio/cheatsheets/blob/main/base-r.pdf`, enthält einige Gedächtnisstützen u. a. zu diversen Funktionen.

2.4 `logical`-Vektoren und logische Operatoren

Mit den „reservierten Worten“ `FALSE` und `TRUE` werden in **R** die logischen Konstanten „falsch“ und „wahr“ bezeichnet. Die äquivalent erscheinenden Objekte `F` und `T` hingegen sind globale Variablen, deren Werte per Voreinstellung zwar auf `FALSE` bzw. `TRUE` gesetzt sind, vom Benutzer aber überschrieben werden können! Vorsicht also mit benutzerdefinierten Objekten namens `F` oder `T`, da sie eventuell die eigentlich beabsichtigten Werte `FALSE` oder `TRUE` maskieren. (Siehe hierzu auch den Punkt zum Unterschied zwischen `TRUE` und `T` auf Seite 27.)

Ein logischer Wert ist das Resultat der Auswertung eines booleschen Ausdrucks (d. h. einer Bedingung). Im folgenden Beispiel wird der Vektor `x` (wie üblich) elementweise mit einem Skalar verglichen. Außerdem werden zwei Vektoren `a` und `b` direkt erzeugt. Sowohl der Resultatvektor für `x > 175` als auch `a` und `b` haben den Modus `logical`:

```
> x <- c(160, 145, 195, 173, 181);    x > 175
[1] FALSE FALSE  TRUE FALSE  TRUE

> (a <- c(T, T, F, F));    (b <- c(T, FALSE, TRUE, F))
[1]  TRUE  TRUE FALSE FALSE
[1]  TRUE FALSE  TRUE FALSE
```

In den folgenden beiden Tabellen finden sich (fast) alle logischen Operatoren, die in **R** für logische Vektoren und Ausdrücke zur Verfügung stehen. Zu Hilfeseiten über verschiedene logische Operatoren oder die logischen Konstanten kommen Sie via `?Logic` bzw. `?logical`, oder z. B. durch `?"!"`, also durch das `?` angewendet auf den Operator in Hochkommata.

2.4.1 Elementweise logische Operationen

Operator	Bedeutung und Bemerkung	Anwendungsbeispiel mit `a`, `b`, `x` von oben
$<$	kleiner	`> x < 190` `[1] TRUE TRUE FALSE TRUE TRUE`
$>$	größer	`> x > 173` `[1] TRUE FALSE TRUE FALSE TRUE`
$==$	gleich	`> x == 145` `[1] FALSE TRUE FALSE FALSE FALSE`
$<=$	kleiner oder gleich	`> x <= 190` `[1] TRUE TRUE FALSE TRUE TRUE`
$>=$	größer oder gleich	`> x >= 173` `[1] FALSE FALSE TRUE TRUE TRUE`
!	nicht (= Negation)	`> !a` `[1] FALSE FALSE TRUE TRUE` `> !(x > 173)` `[1] TRUE TRUE FALSE TRUE FALSE`
$!=$	ungleich	`> x != 145` `[1] TRUE FALSE TRUE TRUE TRUE`
&	und	`> a & b` `[1] TRUE FALSE FALSE FALSE`
|	oder	`> a | b` `[1] TRUE TRUE TRUE FALSE`
xor	exclusives oder (entweder–oder)	`> xor(a, b)` `[1] FALSE TRUE TRUE FALSE`

Beachte:

- In arithmetischen Ausdrücken findet eine automatische Konversion (Engl.: "coercing") von `logical` in `numeric` statt, und zwar wird `TRUE` zu 1 und `FALSE` zu 0, was – mit Umsicht – sehr effizient genutzt werden kann:

```
> a * 2                    > (x > 175) * 1          > sum(x <= 190)
[1] 2 2 0 0                [1] 0 0 1 0 1            [1] 4
```

  Frage: Was berechnet wohl `mean(x <= 190)`?

- Diese automatische Konversion findet im Prinzip sogar in beide Richtungen statt, fallweise also auch von `numeric` in `logical`! Finden Sie heraus, was die folgenden Ausdrücke im einzelnen liefern und machen Sie sich die Ursache(n) für die jeweiligen Resultate klar:

```
> !0      > !1      > !2      > !0 + !0      > !0 - !0      > !0 * !0
?         ?         ?         ?              ?              ?

> !0 + !0 == !0 - !0          > !0 * !0 == !0^2
?                             ?
```

  Aber:

```
> (!0) + (!0)          > (!0) - (!0)          > (!0) * (!0)          > (!0)^2
?                      ?                      ?                      ?
```

- Beim Vergleich mathematisch reellwertiger Größen, die in **R** als `numeric` gespeichert sind, ist die begrenzte, digitale Maschinengenauigkeit zu berücksichtigen. Beispiel:

```
> (0.2 - 0.1) == 0.1;    (0.3 - 0.2) == 0.1
[1] TRUE
[1] FALSE
```

  Das ist kein **R**-spezifischer „Fehler", sondern ein fundamentales, computerspezifisches Problem (was z. B. auch unter `https://cran.r-project.org/doc/FAQ/R-FAQ.html` in der R-FAQ 7.31 "Why doesn't R think these numbers are equal?" besprochen wird). Hier können die Funktionen `all.equal` und `identical` weiterhelfen, bezüglich deren Arbeitsweise wir aber auf die Hilfeseiten verweisen.

- Beispiel zum Unterschied zwischen `TRUE` und `T` (bzw. zwischen `FALSE` und `F`):

```
> TRUE <- 5
Fehler in TRUE <- 5 : ungueltige (do_set) linke Seite in Zuweisung
> T <- 5; T
[1] 5
```

2.4.2 Zusammenfassende logische Operationen

Operator	Bedeutung und Bemerkung	Anwendungsbeispiel mit `a`, `b` und `x` von oben
&&	„sequenzielles" und: Ergebnis ist `TRUE`, wenn beide Ausdrücke `TRUE` sind. Ist aber der linke Ausdruck `FALSE`, wird der rechte gar nicht erst ausgewertet.	`> min(x - 150) > 0 &&` `+ max(log(x - 150)) < 1` `[1] FALSE`
||	„sequenzielles" oder: Ergebnis ist `TRUE`, wenn ein Ausdruck `TRUE`. Ist aber der linke Ausdruck schon `TRUE`, wird der rechte nicht mehr ausgewertet.	`> min(x - 150) < 0 ||` `+ max(log(x - 150)) < 1` `[1] TRUE`

all	Sind alle Elemente `TRUE`? (Das Argument `na.rm` steht zur Verfügung.)	`> all(a)` `[1] FALSE`
any	Ist mindestens ein Element `TRUE`? (`na.rm` steht zur Verfügung.)	`> any(a)` `[1] TRUE`
which	Welche Elemente sind `TRUE`?	`> which(b)` `[1] 1 3`
duplicated	Welche Elemente eines Vektors sind Duplikate von Elementen mit kleinerem Index? Liefert einen logischen Vektor zurück, der angibt, welche Elemente Duplikate sind.	`> y <- c(1, 5, 1, 9, 5)` `> duplicated(y)` `[1] FALSE FALSE TRUE` `[4] FALSE TRUE`

Memos:

- Leerzeichen sind als Strukturierungshilfe sehr zu empfehlen, da sie nahezu beliebig zwischen alle **R**-Ausdrücke eingestreut werden können. Gelegentlich sind sie sogar zwingend notwendig: `sum(x<-1)` kann durch Leerzeichen zwei Ausdrücke mit völlig verschiedenen Bedeutungen ergeben, nämlich `sum(x < -1)` und `sum(x <- 1)`. **R** verwendet für `sum(x<-1)` die zweite Interpretation, also muss man für die erste durch die entsprechende Verwendung von Leerzeichen sorgen (oder von Klammern wie in `sum(x<(-1))`, was aber ziemlich unübersichtlich werden kann).

  Tipp: In RStudio liefert die Tastenkombination „Alt+-“ sowohl am Prompt der R-Console als auch im Editor die Ausgabe der vier Zeichen umfassenden Zeichenkette ’ `<-` ’.

- An dieser Stelle erinnern wir noch einmal an die bereits am Anfang von §2.3.1 auf Seite 21 erwähnten Regeln zur Operatorpräzedenz und an die Hilfeseite `?Syntax` zur Syntax.

2.5 character-Vektoren und elementare Operationen

Vektoren des Modus `character` (kurz: `character`-Vektoren) bestehen aus Elementen, die Zeichenketten ("strings") sind. Zeichenketten stehen in paarweisen doppelten Hochkommata, wie z. B. `"x-Werte"`, `"Peter, Paul und Mary"`, `"NEUEDATEN2002"` und `"17"`. Paarweise einfache Hochkommata können auch verwendet werden, allerdings nicht gemischt mit doppelten innerhalb einer Zeichenkette. `character`-Vektoren werden wie andere Vektoren mit der Funktion `c` zusammengesetzt:

```
> (namen <- c("Peter", "Paul", "Mary"))
[1] "Peter" "Paul"  "Mary"
> weitere.namen <- c('Tic', 'Tac', 'Toe')
> (alle.namen <- c(namen, weitere.namen))
[1] "Peter" "Paul"  "Mary"  "Tic"   "Tac"   "Toe"
```

Werden `character`-Vektoren mit Vektoren anderer Modi, wie z. B. `numeric` oder `logical` verknüpft, so werden alle Elemente in den Modus `character` konvertiert:

```
> c(1.3, namen, TRUE, F)
[1] "1.3"   "Peter" "Paul"  "Mary"  "TRUE"  "FALSE"
```

2.5.1 Zusammensetzen von Zeichenketten: paste

Das „nahtlose" Zusammensetzen von Zeichenketten aus korrespondierenden Elementen mehrerer `character`-Vektoren kann mit der Funktion `paste` ("to paste" = kleben) bewerkstelligt werden (siehe Beispiele unter der folgenden Auflistung):

- Eine beliebige Anzahl an Zeichenketten (d. h. an einelementigen `character`-Vektoren) wird zu einer einzelnen Zeichenkette (in einem einelementigen `character`-Vektor) zusammengesetzt.
- Vorher werden dabei automatisch logische Werte in `"TRUE"` bzw. `"FALSE"` und Zahlen in Zeichenketten, die ihren Ziffernfolgen entsprechen, konvertiert.
- Zwischen die zusammenzusetzenden Zeichenketten wird mit Hilfe des Arguments `sep` ein (frei wählbares) Trennzeichen eingebaut; Voreinstellung ist das Leerzeichen ("blank"), also `sep = " "`.
- `character`-*Vektoren* werden elementweise (unter zyklischer Wiederholung der Elemente der kürzeren Vektoren) zu `character`-Vektoren zusammengesetzt, derart dass die Zeichenketten korrespondierender Elemente zusammengesetzt werden.

Beispiele:

```
> paste("Peter", "ist", "doof!")                    # Aus drei Zeichenketten
[1] "Peter ist doof!"                               # wird eine.

> paste("Tic", "Tac", "Toe", sep = ",")             # Dito, aber mit "," als
[1] "Tic,Tac,Toe"                                   # Trennzeichen.

> paste("Hoehe", hoehe, sep = "=")                  # Aus numeric wird cha-
[1] "Hoehe=160" "Hoehe=140" "Hoehe=155" "Hoehe=175"  # racter und Vektoren ...

> paste("Data", 1:4, ".txt", sep = "")              # werden zyklisch (teil-)
[1] "Data1.txt" "Data2.txt" "Data3.txt" "Data4.txt"  # repliziert bis ...

> paste(c("x", "y"), rep(1:5, each = 2), sep = "")    # alle Laengen zueinan-
[1] "x1" "y1" "x2" "y2" "x3" "y3" "x4" "y4" "x5" "y5" # der passen.
```

2.5.2 Benennung & „Entnennung“ von Vektorelementen: `names`, `setNames` & `unname`

Eine besondere und – wie sich in Punkt 4 von §2.6.1 zeigen wird – besonders nützliche Anwendung von Zeichenketten ist die Benennung der Elemente eines Vektors mit Hilfe der Funktion `names` (wobei diese auf der linken Seite des Zuweisungsoperators `<-` verwendet wird). Dies ändert nichts an den Werten der Elemente des Vektors:

```
> alter <- c(12, 14, 13, 21, 19, 23)
> names(alter) <- alle.namen;   alter
 Peter Paul Mary Tic Tac Toe
    12   14   13  21  19  23
```

Ein nützliche Funktion, die Tipparbeit bei der Benennung von Vektorelementen spart, da sie das benannte Objekt zurückgibt, ist `setNames`. Probieren Sie aus, was anstelle von obiger `names`-Zuweisung der Befehl `setNames(alter, alle.namen)` liefert!

Damit ist der Vektor `alter` nach wie vor ein `numeric`-Vektor, hat aber nun zusätzlich das Attribut `names`, wie die Funktion `attributes` zeigt.

```
> attributes(alter)
$names
[1] "Peter" "Paul"  "Mary"  "Tic"   "Tac"   "Toe"
```

Es ist auch möglich, die Elemente eines Vektors schon bei seiner Erzeugung zu benennen:

```
> c(Volumen = 3.21, Masse = 15.8)
Volumen   Masse
   3.21   15.80
```

Sollte die Benennung von Vektorelementen stören (z. B. weil sie bei langen Vektoren mit außerdem langen Elementenamen viel Speicherplatz in Anspruch nimmt), kann sie mit `unnname` entfernt werden (was in der Regel auch den Speicherplatzbedarf des Vektors stark reduziert):

```
> object.size(alter)   # liefert ca. die Groesse seines Argumentes in Bytes.
[1] 576
> unname(alter);     object.size(unname(alter))
[1] 12 14 13 21 19 23
[1] 88
```

Bemerkung: Der Speicherplatzbedarf von `numeric`-Vektoren mit unbenannten Elementen ist ca. acht Bytes pro Element plus ein konstanter “Overhead” von ca. 40 Bytes (rote Linie in Abb. 5). Pro Elementname kommen (bei kurzen Elementnamen) ebenfalls *in etwa* acht Bytes hinzu, was die Größe eines solchen Vektors also ungefähr verdoppelt (blaue Linie in Abb. 5). Faktisch spielt der Speicherplatzbedarf, den Objekte haben, für viele der Anwendungen *in diesem Kurs* i. d. R. eine untergeordnete Rolle.

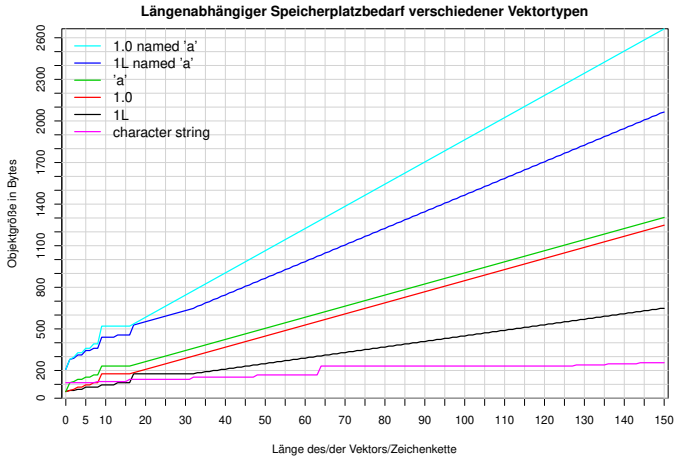

Abbildung 5: Speicherplatzbedarf verschiedener Vektortypen abhängig von der Vektor- bzw. Zeichenkettenlänge.

2.5.3 Weitere Operationen: strsplit, nchar, substring, abbreviate & Co.

Ein paar Funktionen, die für die Arbeit mit character-Vektoren von Nutzen sein können, sind in der folgenden Tabelle aufgeführt:

`> paste(..., sep, collapse)`	Für verschiedene Beispiele ohne Verwendung des Argumentes `collapse` siehe §2.5.1.
`> paste(weitere.namen,` `+ collapse = " und ")` `[1] "Tic und Tac und Toe"` `> paste(weitere.namen,` `+ sep = " und ")` `[1] "Tic" "Tac" "Toe"`	Die Angabe des Argumentes `collapse` führt zum „Kollaps“ aller Eingabezeichenketten zu *einer* Zeichenkette, wobei der Wert von `collapse` eingefügt wird. Beachte den Unterschied zur Wirkung von `sep = " und "`!
`> paste0(....)` `....`	Ist im Ergebnis äquivalent zu `paste(..., sep = "", collapse)`, also zu `paste` mit dem leeren Zeichen als Separator, aber geringfügig effizienter und eine recht häufig nützliche Version.
`> strsplit(....)` `....`	Die sehr nützliche „Umkehrung“ von `paste`, für deren – etwas komplexere – Funktionsweise wir auf die Hilfeseite verweisen.
`> nchar(alle.namen)` `[1] 5 4 4 3 3 3`	Liefert die Längen der in seinem Argument befindlichen Zeichenketten (falls nötig, nach Konversion des Arguments in einen `character`-Vektor).
`> substring(text = namen,` `+ first = 2, last = 3)` `[1] "et" "au" "ar"` `> substring(namen, 2)` `[1] "eter" "aul"  "ary"` `> substring(namen, 2, 3) <- "X"` `> namen` `[1] "PXer" "PXl"  "MXy"`	Produziert einen Vektor von Teil-Zeichenketten der Eingabe an `text`. Das Argument `first` bestimmt die Startposition und `last` die Endposition der Teil-Zeichenketten in den Eingabezeichenketten. Ohne Wert für `last` gehen die Teil-Zeichenketten bis zum Ende der Eingabezeichenketten. Mit einer Zuweisungsanweisung werden im Eingabevektor die jeweiligen Teile durch die rechte Seite von `<-` ersetzt. (Die Werte für `first` und `last` können Vektoren sein; dann wird auf dem Eingabevektor entsprechend elementweise operiert.)
`> abbreviate(names = alle.namen,` `+ minlength = 2)` `Peter Paul Mary  Tic  Tac  Toe` `"Pt" "Pl" "Mr" "Tic" "Tac" "To"`	Generiert (mit Hilfe eines fürs Englische maßgeschneiderten Algorithmus') aus den Zeichenketten des Eingabevektors eindeutige Abkürzungen der Mindestlänge `minlength`. Der Resultatvektor hat benannte Elemente, wobei die Elementnamen die Zeichenketten des Eingabevektors sind.

Hinweise:

- Für weitere, sehr leistungsfähige, aber in ihrer Umsetzung etwas kompliziertere Zeichenkettenbearbeitungen verweisen wir auf die Hilfeseite der Funktion `gsub` und ihre dort genannten „Verwandten“.
- Das Zusatzpaket `stringr` bietet eine Sammlung an sehr leistungsfähigen Zeichenkettenfunktionen, die u. a. in ihrer Nomenklatur konsistenter und z. T. einfacher zu verwenden sind als die in “base **R**” bereits zur Verfügung stehenden. Das zugehörige “cheat sheet”

ist wieder unter `https://posit.co/resources/cheatsheets` zu finden oder direkt bei `https://raw.githubusercontent.com/rstudio/cheatsheets/main/strings.pdf`.

- Mit `trimws`, `tolower`, `toupper` und `chartr` in "base **R**" lassen sich ein paar spezielle, einfachere Umwandlungen elegant bewerkstelligen (siehe deren Hilfeseiten und dort insbes. die Beispiele für ein paar trickreiche, nicht ganz so einfache Umwandlungen).

- Gelegentlich ganz nützlich sind die in "base **R**" schon vorhandenen `character`-Vektoren `letters`, `LETTERS`, `month.name` und `month.abb`. Sie enthalten als Elemente das Alphabet in Klein- bzw. Großbuchstaben sowie die Monatsnamen bzw. deren Abkürzungen:

```
> letters
 [1] "a" "b" "c" .... "z"
> LETTERS
 [1] "A" "B" "C" .... "Z"
> month.name
 [1] "January"   "February"  "March" .... "December"
> month.abb
 [1] "Jan" "Feb" "Mar" .... "Dec"
```

  Ein Objekt mit den Wochentagsnamen ist in "base **R**" hingegen bisher offenbar nicht vorhanden ....

2.6 Indizierung und Modifikation von Vektorelementen: [ ]

Der Zugriff auf einzelne Elemente eines Vektors wird durch seine Indizierung mit der Nummer des gewünschten Elements erreicht. Dies geschieht durch das Anhängen der Nummer in eckigen Klammern `[ ]` an den Namen des Vektors: `x[`i`]` für $i > 0$ liefert das i-te Element des Vektors `x`, sofern dieser mindestens i Elemente besitzt; andernfalls erhält man den Wert `NA` zurück. Die Indizierung der Elemente beginnt mit 1 (im Gegensatz zur Sprache C, wo sie mit 0 beginnt). Der Index 0 (Null), also `x[0]` liefert einen leeren Vektor zuück.

2.6.1 Indexvektoren

Durch die Angabe eines *Indexvektors* kann auf ganze Teilmengen von Vektorelementen zugegriffen werden. Für die Konstruktion eines Indexvektors gibt es vier Methoden:

1. **Indexvektor aus positiven Integer-Werten:** Die Elemente des Indexvektors müssen aus der Menge $\{1, \ldots,$ `length(x)`$\}$ sein. Diese Integer-Werte indizieren die Vektorelemente und liefern sie *in der Reihenfolge*, wie sie im Indexvektor auftreten, zu einem Vektor zusammengesetzt zurück:

```
> alter[5]
 Tac
  19
> alter[c(4:2, 13)]
 Tic Mary Paul <NA>
  21   13   14   NA
> c("Sie liebt mich.", "Sie liebt mich nicht.")[c(1,1,2,1,1,2)]
[1] "Sie liebt mich."   "Sie liebt mich."   "Sie liebt mich nicht."
[4] "Sie liebt mich."   "Sie liebt mich."   "Sie liebt mich nicht."
```

2. **Indexvektor aus negativen Integer-Werten:** Die Menge der Elemente des Indexvektors (d. h. duplizierte Negativindizes werden ignoriert) spezifiziert in diesem Fall die Elemente, die *ausgeschlossen* werden:

```
> alle.namen[-(1:3)]
[1] "Tic" "Tac" "Toe"
> alter[-length(alter)]
 Peter Paul Mary Tic Tac
    12   14   13  21  19
```

3. **Logischer Indexvektor:** Ein Indexvektor mit logischen Elementen wählt die Vektorelemente aus, an deren Position im Indexvektor ein `TRUE`-Wert steht; `FALSE`-Werten entsprechende Elemente werden ausgelassen. Ein zu kurzer Indexvektor wird (nötigenfalls unvollständig) zyklisch repliziert, ein zu langer liefert NA für die überzähligen Indizes zurück:

```
> alter[c(TRUE, TRUE, FALSE, FALSE, TRUE)]
Peter  Paul   Tac   Toe
   12    14    19    23
> x[x > 180]              # x von Seite 26
[1] 195 181
> alle.namen[alter >= 21]
[1] "Tic" "Toe"
> alle.namen[alter >= 14 & alter < 18]
[1] "Paul"

> letters[c(FALSE, FALSE, TRUE)]   # Jeder dritte Buchstabe (von 26)
[1] "c" "f" "i" "l" "o" "r" "u" "x"
```

4. **Indexvektor aus Zeichenketten:** Diese Möglichkeit besteht nur, wenn der Vektor, dessen Elemente indiziert werden sollen, benannte Elemente besitzt (also das Attribut `names` hat, wie z. B. nach Anwendung der Funktion `names`; vgl. §2.5.2). In diesem Fall können die Elementenamen zur Indizierung der Elemente verwendet werden. Ein unzutreffender Elementename liefert ein NA zurück:

```
> alter["Tic"]
 Tic
  21
> alter[c("Peter", "Paul", "Mary", "Heini")]
Peter  Paul  Mary  <NA>
   12    14    13    NA
> alter[weitere.namen]
 Tic Tac Toe
  21  19  23
```

   Beachte: Ein großer Vorteil der Verwendung von Indexvektoren aus Zeichenketten liegt darin, dass **R** die gewünschten Elemente „findet“, ohne dass Sie ihre Positionsnummern im Vektor kennen müssen. D. h. Ihr **R**-Code wäre gegenüber einer z. B. versehentlichen Permutation der Elemente unempfindlich.

Memo: Auch hierzu enthält das “Base R cheat sheet” (via hier oder direkt hier) einige Gedächtnisstützen.

2.6.2 Zwei spezielle Indizierungsfunktionen: `head` und `tail`

Gelegentlich möchte man auf die ersten oder letzten k Elemente eines Vektors `x` zugreifen, was in ersterem Fall recht leicht durch `x[1:k]` bewältigt wird. Im zweiten Fall bedarf es jedoch der Bestimmung und etwas unübersichtlichen Verwendung der Vektorlänge, z. B. wie in `x[(length(x)-k+1):length(x)]`. Die Funktionen `head` und `tail` erleichtern diese Zugriffe auf kompakte Weise (zumindest im Fall von `tail`):

```
> head(alter, n = 2)
Peter  Paul
   12    14
> tail(alter, n = 1)
Toe
 23
```

Bei positivem `n` vorderer bzw. hinterer Teil der Länge `n` eines Vektors. Voreinstellung für `n`: Jeweils sechs Elemente. (Weiterer Vorteil dieser Funktionen: Beide sind z. B. auch auf Matrizen und Data Frames anwendbar. Siehe die entsprechenden Abschnitte 2.8, Seite 42 und 2.10, Seite 55.)

```
> head(alter, n = -2)
Peter  Paul  Mary   Tic
   12    14    13    21
> tail(alter, n = -1)
Paul Mary  Tic  Tac  Toe
  14   13   21   19   23
```

Bei einem negativen Wert für `n` liefert `head` alle Elemente *ohne die letzten* |`n`| Stück und `tail` alle Elemente *ohne die ersten* |`n`| Stück.

2.6.3 Indizierte Zuweisungen

Zuweisungsanweisungen dürfen auf der linken Seite von `<-` ebenfalls einen indizierten Vektor der Form `vektor[indexvektor]` enthalten. Dabei wird die Zuweisung nur auf die indizierten Elemente dieses Vektors angewendet. Hat das Auswertungsresultat der rechten Seite nicht dieselbe Länge wie der Indexvektor, so wird bei „zu kurzer rechter Seite“ wieder zyklisch aufgefüllt und bei „zu langer rechter Seite“ die rechte Seite abgeschnitten sowie in den Fällen, wo dies nicht „aufgeht“, eine Warnung ausgegeben. Beispiele (mit `x` von Seite 26):

```
> x;    x[3] <- 188;   x
[1] 160 145 195 173 181
[1] 160 145 188 173 181
> x[x > 180] <- c(-1, -2, -3)
Warnmeldung:
In x[x > 180] <- c(-1, -2, -3) :
  Anzahl der zu ersetzenden Elemente ist kein Vielfaches der Ersetzungslaenge
> x
[1]  160  145   -1  173   -2
```

Natürlich funktioniert das bei jeder Art von Vektor:

```
> (Farben <- c("rot", "gruen", "blau")[c(1, 1, 2, 1, 3, 3, 1, 2, 2)])
[1] "rot"   "rot"   "gruen" "rot"   "blau"  "blau"  "rot"  "gruen" "gruen"

> Farben[Farben == "rot"] <- "rosa";    Farben
[1] "rosa"  "rosa"  "gruen" "rosa"  "blau"  "blau"  "rosa" "gruen" "gruen"
```

Beachte die Bedeutung leerer eckiger Klammern (`[ ]`): Sie dienen zur Indizierung *aller* Vektorelemente gleichzeitig, im Gegensatz zur Wirkung einer Zuweisung ganz ohne eckige Klammern, die ein Überschreiben des ganzen Objektes zur Folge hat:

```
> x[ ] <- 99;   x
[1] 99 99 99 99 99
> x <- 2;   x
[1] 2
```

Frage: Welche Wirkung hat `x[x < 0] <- -x[x < 0]` auf einen `numeric`-Vektor `x`?

2.7 Faktoren und geordnete Faktoren: Definition und Verwendung

In **R** ist es möglich, nominalskalierte und ordinalskalierte Daten zu charakterisieren. Nominale Daten sind Werte aus einer Menge von sogenannten "Levels" *ohne* Ordnungsrelation, wie sie beispielsweise bei einem Merkmal wie dem Geschlecht, der Blutgruppe, der Automarke oder dem Beruf auftreten. Solche Merkmale werden in **R** als „Faktoren" bezeichnet. Bei ordinalen Daten sind die Levels *mit* einer Ordnungsrelation versehen, wie z. B. bei den Merkmalen Schulnote, Schmerzempfinden, Ausbildungsabschluss, soziale Schicht, Altersklasse etc. Diese Merkmale werden in **R** „geordnete Faktoren" genannt. In beiden Fällen sind nur endliche Mengen von Levels zugelassen. Der Sinn dieses Konzepts in **R** liegt im Wesentlichen in der adäquaten Interpretation und Behandlung derartiger Variablen in statistischen Modellen und Funktionen.

Die k möglichen Levels eines (geordneten oder ungeordneten) Faktors werden in **R** durch die natürlichen Zahlen von 1 bis k codiert, sind aber mit Zeichenketten assoziierbar, sodass die Bedeutung der Levels (für den Menschen) besser erkennbar bleibt. Die Daten werden in **R** als `numeric`-Vektoren von Levelcodes gespeichert. Diese Vektoren besitzen zwei Attribute: Das Attribut `levels`, das die Menge der k Zeichenketten enthält, welche alle *zulässigen* Levels des Faktor beschreiben, und das Attribut `class`.

Im Fall eines (ungeordneten) Faktors hat das `class`-Attribut den Wert `"factor"` und macht dadurch kenntlich, dass es sich um einen Vektor mit Werten eines (ungeordneten) Faktors handelt. Das `class`-Attribut eines geordneten Faktors hingegen hat den Wert `c("ordered", "factor")` und die *Reihenfolge* der Levels im `levels`-Attribut spiegelt die jeweilige Ordnungsrelation der Levels wider. Wir wollen solche Vektoren fortan kurz `factor`- bzw. `ordered`-Vektoren nennen.

Beachte, dass der Modus eines `factor`- oder `ordered`-Vektors `numeric` ist!

Anhand von Beispieldaten sollen die Erzeugung und die Arbeit mit Vektoren der beiden Klassen (Faktor bzw. geordneter Faktor) erläutert werden. Angenommen, wir erzeugen die folgenden Vektoren:

```
> alter <- c(35, 39, 53, 14, 26, 68, 40, 56, 68, 52, 19, 23, 27, 67, 43)
> geschlecht <- c("m", "m", "w", "w", "m", "w", "w", "m", "m", "w", "m", "m",
+  "w", "w", "w")
> blutgruppe <- c("A", "B", "B", "0", "A", "AB", "0", "AB", "B", "AB", "A",
+  "A", "AB", "0", "B")
> gewicht <- c(82, 78, 57, 43, 65, 66, 55, 58, 91, 72, 82, 83, 56, 51, 61)
> groesse <- c(181, 179, 153, 132, 166, 155, 168, 158, 188, 176, 189, 179,
+  167, 158, 174)
> rauchend <- c("L", "G", "X", "S", "G", "G", "X", "L", "S", "X", "X", "L",
+  "X", "X", "S")
```

Die Vektoren `alter`, `gewicht` und `groesse` enthalten nun metrische Daten und haben den Modus `numeric`. Die Daten in den `character`-Vektoren `geschlecht` und `blutgruppe` sind in unserer *Interpretation* von nominalem Typ, während hinter den Daten in `rauchend` eine Ordnung steckt, es sich also um ordinale Daten handelt („X“ $\hat{=}$ Nichtraucher/in, „G“ $\hat{=}$ Gelegenheitsraucher/in, „L“ $\hat{=}$ Leichte/r Raucher/in, „S“ $\hat{=}$ Starke/r Raucher/in). Tatsächlich wird die Eigenschaft, Ausprägungen einer nominal bzw. ordinal skalierten Variablen und damit Werte aus einer jeweils endlichen Menge ohne bzw. mit Ordnungsrelation zu enthalten, nicht in den `character`-Vektoren repräsentiert. Das ändern wir in den folgenden Paragraphen.

Wir werden die obigen nominalen Datenvektoren des Modus' `character` in solche der Klasse `factor` umwandeln und den ordinalen Datenvektor in einen der Klasse `ordered`. Ferner wird das Alter der Personen (in `alter`) in vier Intervalle gruppiert und diese Gruppierung dann in einem `ordered`-Vektor gespeichert werden.

2.7.1 Erzeugung von Faktoren (`factor`, `gl`) und Levelabfrage (`levels`)

```
> (ge <- factor(geschlecht))
[1] m m w w m w w m m w m m w w w
Levels: m w
> (blut <- factor(blutgruppe))
[1] A  B  B  0  A  AB 0  AB ....
Levels: 0 A AB B
```

Erzeugung von (ungeordneten) Faktoren aus den `character`-Vektoren `geschlecht` und `blutgruppe`. Die Ausgabe von Faktoren erfolgt ohne Hochkommata, um zu zeigen, dass es keine `character`-Vektoren sind. Außerdem werden die Levels dokumentiert.

```
> levels(ge)
[1] "m" "w"
> levels(blut)
[1] "0"  "A"  "AB" "B"
```

Die Menge der Levels ist per Voreinstellung automatisch alphanumerisch sortiert worden. Dies impliziert *per se* noch keine Ordnung, muss aber beachtet werden! (Siehe unten bei der Vergabe von Levelnamen durch das Argument `labels` und bei der Erzeugung geordneter Faktoren.)

```
> blut2 <- factor(blutgruppe,
+ levels = c("A", "B", "AB", "0"))
> levels(blut2)
[1] "A"  "B"  "AB" "0"
```

Es ist bei der Faktorerzeugung möglich, die Menge der zulässigen Levels sowie eine Levelsreihenfolge/-sortierung vorzugeben, indem das Argument `levels` verwendet wird. Für jeden Wert im Ausgangsvektor (hier `blutgruppe`), der *nicht* im `levels`-Argument auftaucht, würde `NA` vergeben.

```
> (ge2 <- factor(geschlecht,
+ levels = c("m", "w"),
+ labels = c("Mann", "Frau")))
[1] Mann Mann Frau Frau Mann ....
Levels: Mann Frau
```

Das Argument `labels` erlaubt die freie Umbenennung der Faktorlevels gleichzeitig mit der Faktorerzeugung.

Bemerkung: Eine sehr nützliche, da effiziente Funktion zur Erzeugung von Faktorvektoren, in deren Elementen die Faktorlevels in gewissen Mustern aufeinanderfolgen, ist `gl`. Für Details verweisen wir auf ihre Hilfeseite und auf die Ausgabe von `example(gl)`.

2.7.2 Änderung der Levelsortierung (`relevel`, `reorder`), Zusammenfassung von Levels (`levels`), Löschen unnötiger Levels (`droplevels`)

```
> (blut <- factor(blut, levels =
+ c("A", "B", "AB", "0"))
[1] A  B  B  0  A  AB 0  AB ....
Levels: A B AB 0
```

Die Levelsortierung kann geändert werden, indem der Faktor quasi neu erzeugt wird und dabei dem `levels`-Argument die Levels in der gewünschten Reihenfolge übergeben werden.

```
> relevel(blut, ref = "0")
[1] A  B  B  0  A  AB 0  AB ....
Levels: 0 A B AB
```

Soll nur eines der Levels zum „ersten“ gemacht und die anderen „nach hinten verschoben“ werden, reicht es, `relevel` das neue „Referenzlevel“ über sein Argument `ref` anzugeben.

```
> reorder(....)
```

Sind Faktorlevels in Abhängigkeit von Werten einer anderen (i. d. R. `numeric`-)Variablen zu sortieren, kann `reorder` die Lösung sein; siehe ihre Hilfeseite.

```
> levels(blut2) <- c("non0",
+ "non0", "non0", "0")
> blut2
[1] non0 non0 non0    0 ....
Levels: non0 0
```

Levels werden zusammengefasst (und auch umbenannt) durch Zuweisung eines entsprechenden `character`-Vektors passender Länge an das `levels`-Attribut. Die Zuordnung der alten Levels zu den neuen geschieht über ihre Elementepositionen.

```
> blut[1:5]  # = head(blut, 5)
[1] A B B 0 A
Levels: A B AB 0
```

Die Levels eines Faktors sind gegenüber (jeglicher Form von) Indizierung invariant, d. h., nicht mehr im Faktorvektor auftretende Levels werden nicht gelöscht, ...

```
> droplevels(blut[1:5])
[1] A B B 0 A
Levels: A B 0
> blut[1:5, drop = TRUE]
[1] A B B 0 A
Levels: A B 0
```

sondern müssen, sofern dies gewünscht oder nötig ist, entweder explizit mit `droplevels` oder durch Verwendung der Indizierungsoption `drop = TRUE` entfernt werden.

2.7.3 Erzeugung von geordneten Faktoren: `ordered`, `gl`

```
> (rauch <- ordered(rauchend))
[1] L G X S G G X L S X X L X X S
Levels: G < L < S < X
```

Erzeugung eines geordneten Faktors aus dem `character`-Vektor `rauchend`. Dabei wird für die Levelordnung die alphabetische Levelsortierung verwendet.

`> (rauch <- ordered(rauchend,` `+ levels = c("X", "G", "L", "S")))` `[1] L G X S G G X L S X X L X X S` `Levels: X < G < L < S`	Die Vorgabe einer Levelordnung bei Erzeugung des geordneten Faktors geschieht durch das Argument `levels`. Dabei bestimmt die Levelreihenfolge die Ordnung. Jeder Wert im Ausgangsvektor, der nicht im `levels`-Argument auftaucht, liefert `NA`.
`> (rauch2 <- ordered(rauchend,` `+ levels = c("X", "G", "L", "S"),` `+ labels = c("NR", "gel.", "leicht",` `+ "stark")))` `[1] leicht gel.  NR  stark ....` `Levels: NR < gel. < leicht < stark`	Die direkte Erzeugung eines geordneten Faktors mit vorgegebener Levelordnung *und* freier Namensgebung für die Levels ist durch die kombinierte Verwendung der Argumente `levels` und `labels` möglich.
`> ordered(blut)` `[1] B  0  0  A  B  AB A  AB 0 ....` `Levels: A < B < AB < 0`	Die Anwendung von `ordered` auf einen (ungeordneten) `factor`-Vektor liefert einen geordneten Faktor, dessen Levelordnung und -bezeichnungen vom `factor`-Vektor übernommen werden. (Hier ein unsinniges Beispiel.)

Bemerkung: Die in §2.7.1 bereits erwähnten Funktionen `factor` und `gl` besitzen jeweils ein logisches Argument namens `ordered`, das auch mit ihnen die Generierung geordneter Faktorvektoren (im Fall von `gl` mit Levelmustern) erlaubt; siehe ihre Hilfeseite.

2.7.4 Änderung der Levelordnung, Zusammenfassung von Levels, Löschen unnötiger Levels bei geordneten Faktoren

`> (rauch <- ordered(rauch,` `+ levels = c("S", "L", "G", "X")))` `[1] L G X S G G X L S X X L X X S` `Levels: S < L < G < X`	Die Levelordnung kann geändert werden, indem der geordnete Faktor mit der gewünschten Levels-Reihenfolge mittels `ordered` erneut erzeugt wird. (Beachte: `relevel` funktioniert bei einem geordneten Faktor nicht!)
`> levels(rauch2) <- c("NR", "R",` `+ "R", "R");  rauch2` `[1] R  R  NR R  R  R  NR R  R ....` `Levels: NR < R`	Das Zusammenfassen (und Umbenennen) von Levels geschieht wie bei ungeordneten Faktoren (siehe §2.7.2). Hierbei bleibt die Eigenschaft, ein geordneter Faktor zu sein, erhalten!
`> tail(rauch, 7)` `[1] S X X L X X S` `Levels: X < G < L < S` `> droplevels(tail(rauch, 7))` `[1] S X X L X X S` `Levels: X < L < S`	Für das Entfernen von nicht mehr im Faktorvektor auftretenden Levels gilt dasselbe wie bei ungeordneten Faktoren (siehe §2.7.2), wobei die Eigenschaft, ein geordneter Faktor zu sein, erhalten bleibt.

Als beispielhafte Bestätigung dessen, was am Anfang von Abschnitt 2.7 gesagt wurde:

```
> mode(blut)
[1] "numeric"
> class(blut)
[1] "factor"
> attributes(blut)
$levels
[1] "A"  "B"  "AB" "0"
$class
[1] "factor"
```

```
> mode(rauch)
[1] "numeric"
> class(rauch)
[1] "ordered" "factor"
> attributes(rauch)
$levels
[1] "S" "L" "G" "X"
$class
[1] "ordered" "factor"
```

2.7.5 Klassierung numerischer Werte und Erzeugung geordneter Faktoren: `cut`

```
> (AKlasse <- cut(alter, breaks =
+ c(0, 25, 45, 60, Inf)))
 [1] (25,45]  (25,45]  (45,60]
 [4] (0,25]   (25,45]  (60,Inf]
 [7] (25,45]  (45,60]  (60,Inf]
[10] (45,60]  (0,25]   (0,25]
[13] (25,45]  (60,Inf] (25,45]
Levels:
(0,25] (25,45] (45,60] (60,Inf]
```

Aus dem `numeric`-Vektor `alter` wird durch die Funktion `cut` ein `factor`-Vektor (hier `AKlasse`) erzeugt, indem sie gemäß der im Argument `breaks` angegebenen Intervallgrenzen (per Voreinstellung) links offene und rechts abgeschlossene Klassen $(b_i, b_{i+1}]$ bildet und daraus (per Voreinstellung) entsprechende Faktorlevels konstruiert (`Inf` $= +\infty$). Jedes Element des Ausgangsvektors wird im Faktor durch das Intervall/Level codiert, in das es fällt. (Levels können durch das Argument `labels` beliebig benannt werden.)

```
> (AKlasse <- ordered(AKlasse))
[1] (25,45] (25,45] (45,60] ....
Levels:
(0,25] < (25,45] < (45,60] < (60,Inf]
```

Die Anwendung von `ordered` auf den resultierten `factor`-Vektor liefert den geordneten Faktor, dessen Levelordnung und -bezeichnungen aus dem `factor`-Vektor übernommen werden.

2.7.6 Tabellierung von Faktoren und Faktorkombinationen: `table`

```
> table(AKlasse)
AKlasse
 (0,25]  (25,45]  (45,60] (60,Inf]
      3        6        3        3
```

`table` erstellt ein `table`-Objekt mit der Tabelle der absoluten Häufigkeiten jedes Levels in `AKlasse`.

```
> table(ge, AKlasse)
   AKlasse
ge  (0,25] (25,45] (45,60] (60,Inf]
  m      2       3       1        1
  w      1       3       2        2
```

Für zwei Argumente wird die Kontingenztafel aller Levelkombinationen erstellt.

```
> table(AKlasse, ge, rauch)
,,rauch = X   ,,rauch = G   ,,rauch = L   ,,rauch = S
          ge            ge            ge            ge
AKlasse   m w AKlasse  m w  AKlasse  m w  AKlasse  m w
 (0,25]   1 0 (0,25]   0 0  (0,25]   1 0  (0,25]   0 1
 (25,45]  0 2 (25,45]  2 0  (25,45]  1 0  (25,45]  0 1
 (45,60]  0 2 (45,60]  0 0  (45,60]  1 0  (45,60]  0 0
 (60,Inf] 0 1 (60,Inf] 0 1  (60,Inf] 0 0  (60,Inf] 1 0
```

Für mehr als zwei Argumente besteht die Tabellierung aller Levelkombinationen – etwas unübersichtlicher – aus mehreren Kontingenztafeln (und ist hier wegen Platzmangel etwas umformatiert).

Hinweise:

- Siehe auch die Hilfeseiten zu `proportions`, `margin.table` und `addmargins`, mit deren Hilfe eine bestehende Tabelle absoluter Häufigkeiten in eine solche mit relativen Häufigkeiten überführt werden kann bzw. die Marginal- sprich Randhäufigkeiten (also Summen über einen Index) bestimmt werden können bzw. eine Tabelle um nützliche Marginalhäufigkeiten ergänzt werden kann.

- Beachte, dass `NA`s und `NaN`s – per Voreinstellung – durch `table(x)` *nicht* mittabelliert werden! Dazu müssen `NA` und `NaN` wie (z. B.) in `table(factor(x, exclude = NULL))` für den zu tabellierenden Faktor `x` temporär explizit zu einem Level gemacht oder `table(x, useNA = "ifany")` oder `table(x, useNA = "always")` verwendet werden (siehe die Hil-

feseite). `summary(x)` kann auch verwendet werden, liefert aber – per Voreinstellung – möglicherweise nicht die gesamte Tabelle.

- In §?? zu Unabhängigkeits- bzw. Homogenitätstests sowie in §?? zu Kontingenztafeln für $k \geq 2$ Faktoren wird eine weitere Funktion namens `xtabs` zur Erstellung von Häufigkeits-/Kreuz- oder Kontingenztabellen sowie eine namens `ftable` zu deren „flachen" Darstellung, falls $k \geq 3$ ist, vorgestellt.

2.7.7 Aufteilung gemäß Faktor(en)gruppen und faktor(en)gruppierte Funktionsanwendungen: `split`, `tapply` & `ave`

```
> split(gewicht, AKlasse)
$`(0,25]`
[1] 43 82 83

$`(25,45]`
[1] 82 78 65 55 56 61

$`(45,60]`
[1] 57 58 72

$`(60,Inf]`
[1] 66 91 51
```

Ist `g` ein Faktorvektor derselben Länge des Vektors `x`, so teilt `split(x, g)` die Elemente von `x` in Gruppen ein, die durch wertgleiche Elemente in `g` definiert sind. Rückgabewert von `split` ist eine Liste (siehe hierzu Abschnitt 2.9, Seite 50) von Vektoren der gruppierten `x`-Elemente. Die Komponenten der Liste sind benannt durch die Levels des gruppierenden Faktors. Hier geschieht eine Gruppierung der `gewicht`-Elemente gemäß der durch `AKlasse` indizierten Altersgruppen.

```
> split(gewicht, list(ge, AKlasse))
$`m.(0,25]`
[1] 82 83

$`w.(0,25]`
[1] 43

$`m.(25,45]`
[1] 82 78 65

$`w.(25,45]`
[1] 55 56 61

$`m.(45,60]`
[1] 58

$`w.(45,60]`
[1] 57 72

$`m.(60,Inf]`
[1] 91

$`w.(60,Inf]`
[1] 66 51
```

Ist `g` (in `split(x, g)`) eine Liste von Faktorvektoren (alle derselben Länge von `x`), werden die Elemente von `x` in Gruppen eingeteilt, die durch die Levelkombinationen der Faktoren, die in `g` zusammengefasst sind, definiert werden. Rückgabewert ist eine Liste von Vektoren der gruppierten `x`-Elemente, deren Komponenten durch die aufgetretenen Levelkombinationen der gruppierenden Faktoren (getrennt durch einen Punkt „.") benannt sind.

Die Funktion `tapply` ist eine Erweiterung von `split`, indem sie die Anwendung einer im Prinzip frei wählbaren Funktion auf die gruppierten Elemente ermöglicht: Falls `g` ein Faktor derselben Länge des Vektors `x` ist, wendet `tapply(x, g, FUN)` die Funktion `FUN` auf Gruppen der Elemente von `x` an, wobei diese Gruppen durch gleiche Elemente von `g` definiert sind.

Ist `g` eine Liste (vgl. Abschnitt 2.9, Seite 50) von Faktoren (alle derselben Länge von `x`), wird `FUN` auf Gruppen der Elemente von `x` angewendet, die durch die Levelkombinationen der Faktoren, die in `g` genannt sind, definiert werden:

```
> tapply(gewicht, AKlasse, mean)
   (0,25]   (25,45]   (45,60] (60,Inf]
 69.33333 66.16667 62.33333 69.33333
> tapply(gewicht, AKlasse, sd)
   (0,25]   (25,45]   (45,60] (60,Inf]
 22.81082 11.37395 8.386497 20.20726
```

Faktorgruppierte Anwendung der Funktion `mean` zur Bestimmung der mittleren Gewichte für jede durch `AKlasse` indizierte Altersgruppe. Dito mittels `sd` zur Berechnung der empirischen Standardabweichungen.

```
> tapply(gewicht, list(ge, AKlasse),
+ mean)
  (0,25]  (25,45] (45,60] (60,Inf]
m   82.5 75.00000    58.0     91.0
w   43.0 57.33333    64.5     58.5
```

Faktorengruppierte Anwendung von `mean` auf die durch jede Levelkombination von `ge` mit `AKlasse` indizierten Einträge in `gewicht`. (`list` wird in Abschnitt 2.9, Seite 50 erläutert.)

Ein Funktionsaufruf der Art `ave(x, g1, ...., gk)` für einen `numeric`-Vektor `x` und Faktor-Vektoren `g1` bis `gk` macht etwas ähnliches wie `tapply`, ist aber per Voreinstellung auf die Berechnung von Mittelwerten für Gruppen von `x`-Elementen eingestellt, wobei diese Gruppen durch gleiche Kombinationen von Elementen von `g1` bis `gk` definiert sind. Außerdem ist das Ergebnis von `ave(x, g1, ...., gk)` ein Vektor derselben Länge wie `x`, dessen Elemente, die zur selben Gruppe gehören, alle auch denselben Wert, nämlich den betreffenden Gruppenmittelwert haben. Beispiele:

```
> ave(gewicht, AKlasse)
 [1] 66.16667 66.16667 62.33333 69.33333 66.16667 69.33333 66.16667 62.33333
 [9] 69.33333 62.33333 69.33333 69.33333 66.16667 69.33333 66.16667

> ave(gewicht, ge, AKlasse)
 [1] 75.00000 75.00000 64.50000 43.00000 75.00000 58.50000 57.33333 58.00000
 [9] 91.00000 64.50000 82.50000 82.50000 57.33333 58.50000 57.33333
```

Warnung vor dem „Nebeneffekt“ der ab- oder unabsichtlichen Anwendung von `c` auf einen Faktor: Vergleichen Sie das Ergebnis von `c(faktorvektor)` mit den Ergebnissen von

`> as.vector(faktorvektor)`,

`> as.numeric(faktorvektor)`

und

`> as.numeric(levels(faktorvektor)[faktorvektor])`

wenn `faktorvektor` ein Faktorvektor ist mit Elementen, die natürliche Zahlen, Dezimalbrüche oder Zeichenketten darstellen (s. u.)! Zu den Funktionen `as.vector` und `as.numeric` wird in Abschnitt 2.11 noch etwas mehr mitgeteilt.

```
> u <- c(1, -7, 4, NA)
> (uf <- factor(u))
[1] 1    -7   4    <NA>
Levels: -7 1 4

> c(uf)
[1]  2  1  3 NA

> as.vector(uf)
[1] "1"  "-7" "4"  NA

> as.numeric(uf)
[1]  2  1  3 NA

> as.numeric(levels(uf))[uf]
[1]  1 -7  4 NA
```

```
> v <- c(1.3, -7.5, 4.8, NA)
> (vf <- factor(v))
[1] 1.3  -7.5 4.8  <NA>
Levels: -7.5 1.3 4.8

> c(vf)
[1]  2  1  3 NA

> as.vector(vf)
[1] "1.3"  "-7.5" "4.8"  NA

> as.numeric(vf)
[1]  2  1  3 NA

> as.numeric(levels(vf))[vf]
[1]  1.3 -7.5  4.8   NA
```

```
> w <- c("1.3", "-7", "a", NA)
> (wf <- factor(w))
[1] 1.3  -7   a    <NA>
Levels: -7 1.3 a

> c(wf)
[1]  2  1  3 NA

> as.vector(wf)
[1] "1.3" "-7"  "a"   NA

> as.numeric(wf)
[1]  2  1  3 NA

> as.numeric(levels(wf))[wf]
[1]  1.3 -7.0   NA   NA
Warnmeldung:
NAs durch Umwandlung erzeugt
```

2.8 Matrizen: Erzeugung, Indizierung, Modifikation und Operationen

In **R** stehen mehrdimensional indizierte Felder (Englisch: "arrays") zur Verfügung, für die wir den Terminus „Array" übernehmen. Dies sind Vektoren mit einem Attribut `dim` (und evtl. zusätzlich mit einem Attribut `dimnames`); sie bilden die Klasse `array`. Ein Spezialfall hierin sind wiederum die *zwei*dimensionalen Arrays, genannt Matrizen; diese haben die Klasse `matrix`. Sowohl Arrays als auch Matrizen sind intern als Vektoren gespeichert und unterscheiden sich von „echten" Vektoren nur durch die zwei schon genannten Attribute `dim` und `dimnames`.

2.8.1 Grundlegendes zu Arrays

Arrays werden mit der Funktion `array` erzeugt. Sie benötigt als Argumente einen Datenvektor, dessen Elemente in das Array eingetragen werden sollen, sowie einen Dimensionsvektor, der das `dim`-Attribut wird und dessen Länge k die Dimension des Arrays bestimmt. Die Elemente des Dimensionsvektors sind positive `integer`-Werte und die Obergrenzen eines jeden der k Indizes, mit denen die Elemente des Arrays indiziert werden. Es sei zum Beispiel `z` ein Vektor mit 1500 Elementen. Dann erzeugt die Anweisung

```
> a <- array(z, c(3, 5, 100))
```

ein offenbar dreidimensionales $(3 \times 5 \times 100)$-Array, dessen Elemente die Elemente von `z` (in einer gewissen Reihenfolge) sind und dessen Element a_{ijk} mit `a[i, j, k]` indiziert wird. Dabei muss $i \in \{1, 2, 3\}, j \in \{1, 2, 3, 4, 5\}$ und $k \in \{1, \ldots, 100\}$ sein.
Die Elemente des Datenvektors `z` werden in das Array `a` eingetragen, indem sukzessive die Elemente `a[i, j, k]` gefüllt werden, wobei der erste Index (`i`) seinen Wertebereich am schnellsten und der letzte Index (`k`) seinen Wertebereich am langsamsten durchläuft. D. h., für das dreidimensionale Array `a` sind die Elemente `z[1], ..., z[1500]` von `z` sukzessive in die Elemente `a[1,1,1]`, `a[2,1,1]`, `a[3,1,1]`, `a[1,2,1]`, `a[2,2,1]`, ..., `a[2,5,100]` und `a[3,5,100]` „eingelaufen".
Unsere häufigsten Anwendungen werden jedoch nicht mehrdimensionale Arrays benötigen, sondern Matrizen, auf die wir uns i. F. konzentrieren werden. Nichtsdestotrotz sind Arrays kennenswerte, da äußerst nützliche Datenstrukturen, für die auch sehr leistungsfähige Funktionen existieren (die wir zum Teil kennenlernen werden); `?array` führt zu weiteren Informationen.

2.8.2 Erzeugung von Matrizen: `matrix`

Eine Matrix wird mit der Funktion `matrix` erzeugt. (Dies ist auch als zweidimensionales Array mit `array` möglich, aber `matrix` ist „maßgeschneidert" für Matrizen.) Sie erwartet als erstes Argument einen Datenvektor, dessen Elemente *spaltenweise* in die Matrix eingetragen werden, und in mindestens einem weiteren Argument eine Angabe, wie viele Zeilen (bzw. Spalten) die Matrix haben soll. Folgende Beispiele sollen die Funktionsweise von `matrix` erläutern:

```
> z <- c(130, 26, 110, 24, 118, 25, 112, 25)
> (Werte <- matrix(z, nrow = 4))
     [,1] [,2]
[1,]  130  118
[2,]   26   25
[3,]  110  112
[4,]   24   25
```

Hier wird aus dem Datenvektor `z` eine Matrix mit `nrow = 4` Zeilen (Englisch: "rows") erzeugt. Die Spaltenzahl wird automatisch aus der Länge des Datenvektors ermittelt. Der Datenvektor füllt die Matrix dabei spaltenweise auf. (Dies ist die Voreinstellung.)

Ist die Länge des Datenvektors kein ganzzahliges Vielfaches der für `nrow` angegebenen Zeilenzahl, werden (höchstens) so viele Spalten angelegt, bis der Datenvektor in der Matrix vollständig enthalten ist. Die dann noch leeren Elemente der Matrix werden durch zyklische Wiederholung

der Datenvektorelemente aufgefüllt (und es wird dann eine Warnung ausgegeben). In folgendem Beispiel wird der achtelementige Datenvektor `z` in eine dreizeilige Matrix eingetragen, was durch eine Warnung quittiert wird:

```
> matrix(z, nrow = 3)
     [,1] [,2] [,3]
[1,]  130   24  112
[2,]   26  118   25
[3,]  110   25  130
Warnmeldung:
In matrix(z, nrow = 3) :
  Datenlaenge [8] ist kein Teiler oder Vielfaches der Anzahl der Zeilen [3]
```

Die hier und in den nächsten Abschnitten folgenden Tabellen enthalten Anweisungen und Operationen, die zur Erzeugung, Spalten- und Zeilenbenennung, Indizierung, Erweiterung von und Rechnung mit Matrizen zur Verfügung stehen.

`> Werte <- matrix(z, nrow = 4)`	(Siehe obiges Beispiel.)
`> matrix(z, ncol = 4)` `     [,1] [,2] [,3] [,4]` `[1,]  130  110  118  112` `[2,]   26   24   25   25`	Der Datenvektor `z` wird in eine Matrix mit `ncol = 4` Spalten ("columns") geschrieben. Die Zeilenzahl wird automatisch aus der Länge von `z` ermittelt.
`> matrix(z, nrow = 4, ncol = 5)` `....`	`ncol` und `nrow` können gleichzeitig angegeben werden; `z` wird dann zyklisch verwendet, wenn zu kurz, und abgeschnitten, wenn zu lang. Mit dem Argument `byrow = TRUE` wird der Datenvektor *zeilenweise* in die Matrix eingelesen. Voreinstellung ist `byrow = FALSE`, also spaltenweises Einlesen. (Ohne jegliche Angabe von `ncol` und `nrow` wird eine einspaltige Matrix erzeugt; ohne Angabe eines Datenvektors eine Matrix mit `NA`-Einträgen.)
`> (Werte <- matrix(z, ncol = 2,` `+ byrow = TRUE))` `     [,1] [,2]` `[1,]  130   26` `[2,]  110   24` `[3,]  118   25` `[4,]  112   25`	

2.8.3 Be- & „Entnennung" von Spalten und Zeilen: `dimnames`, `colnames`, `rownames`, `unname`

`> dimnames(Werte) <- list(c("Alice", "Bob",` `+ "Carol", "Deborah"),` `+ c("Gewicht", "Alter"));     Werte` `        Gewicht Alter` `  Alice     130    26` `    Bob     110    24` `  Carol     118    25` `Deborah     112    25`	Mit der Funktion `dimnames` werden den Zeilen einer Matrix die Namen zugewiesen, die sich im ersten `character`-Vektor der Liste (dazu mehr in Abschnitt 2.9, Seite 50) auf der rechten Seite befinden. Die Spalten bekommen die Namen, die im zweiten Vektor sind.
`> Werte <- matrix(z, ncol = 2, byrow = TRUE,` `+ dimnames = list(c("Alice", "Bob", "Carol",` `+ "Deborah"), c("Gewicht", "Alter")))`	Eine Zeilen- und Spaltenbenennung kann auch schon bei der Erstellung der Matrix über das Argument `dimnames` erfolgen.
`> dimnames(Werte)` `[[1]]` `[1] "Alice"  "Bob"  "Carol"  "Deborah"` `[[2]]` `[1] "Gewicht" "Alter"`	Der Zugriff auf die Dimensionsnamen durch `dimnames` liefert eine Liste, deren zwei Komponenten die Vektoren der Zeilen- bzw. Spaltennamen enthalten, falls vorhanden; ansonsten jeweils das NULL-Objekt.

```
> dimnames(Werte) <- list(NULL,
+ c("Gewicht", "Alter"));   Werte
     Gewicht Alter
[1,]     130    26
[2,]     110    24
[3,]     118    25
[4,]     112    25
```

Die Zuweisung des NULL-Objekts an Stelle eines Zeilennamenvektors führt zur Löschung vorhandener Zeilennamen. An der Stelle des zweiten Vektors würde es zur Löschung der Spaltennamen führen.

```
> colnames(Werte)
....
> rownames(Werte) <- c("A", "B",
+ "C", "D")
```

Zugriff auf oder Zuweisung an Spalten- bzw. Zeilennamen allein ist hiermit kompakter möglich.

```
> unname(Werte)
     [,1] [,2]
[1,]  130   26
[2,]  110   24
[3,]  118   25
[4,]  112   25
```

Spalten- und Zeilennamen einer Matrix können mit `unnname` entfernt werden (z. B. wenn sie störend zu lang sind). Hier geschieht dies nur „temporär“ für die Ausgabe; damit sie permanent entfernt würden, wäre das Ergebnis von `unname(Werte)` natürlich an `Werte` zuzuweisen.

Beachte: Bei der Verwendung von Dimensionsnamen für Matrizen entsteht ein erhöhter Speicherplatzbedarf, der bei großen Matrizen erheblich sein kann und insbesondere bei aufwändigeren Berechnungen (vor allem in Simulationen) die Geschwindigkeit von `R` negativ beeinflusst. Daher kann es sinnvoll sein, solche Dimensionsnamen vorher zu entfernen. (Siehe hierzu auch die Bemerkung zum Speicherplatzbedarf von Elementenamen bei Vektoren in §2.5.2.)

2.8.4 Erweiterung um Spalten oder Zeilen: `cbind`, `rbind`

```
> groesse <- c(140, 155, 142, 175)
> (Werte <- cbind(Werte, groesse))
  Gewicht Alter groesse
A     130    26     140
B     110    24     155
C     118    25     142
D     112    25     175
```

Eine bestehende Matrix lässt sich um zusätzliche Spalten und Zeilen erweitern, indem an sie ein passender Vektor „angebunden“ wird. `cbind` erledigt dies spaltenweise, `rbind` zeilenweise. Die Länge des Vektors muss – je nach Art des „Anbindens“ – mit der Zeilen- bzw. Spaltenzahl der Matrix übereinstimmen.

```
> (Werte <- rbind(Werte,
+ E = c(128, 26, 170)))
  Gewicht Alter groesse
A     130    26     140
B     110    24     155
C     118    25     142
D     112    25     175
E     128    26     170
```

Ist der anzubindende Vektor in einem Objekt gespeichert, so wird dessen Name als Spalten- bzw. Zeilenname übernommen. Ansonsten kann durch *name = wert* ein Spalten- bzw. Zeilenname angegeben werden.
Matrizen mit gleicher Zeilenzahl werden durch `cbind` spaltenweise aneinandergehängt, solche mit gleicher Spaltenzahl mit `rbind` zeilenweise.

2.8.5 Matrixdimensionen und Indizierung von Elemente: `dim`, `[]`, `head` & `tail`

```
> dim(Werte)
[1] 5 3
> nrow(Werte); ncol(Werte)
[1] 5
[1] 3
```

`dim` liefert die Zeilen- und Spaltenzahl zusammen als zweielementigen Vektor.
`nrow` bzw. `ncol` liefern sie einzeln. (`Werte` stammt aus §2.8.4.)

Die Indizierung von Matrizen funktioniert analog zu der von Vektoren (vgl. §2.6.1):

```
> Werte[3,2]
[1] 25
```

Matrixelemente werden durch Doppelindizes indiziert: `[i,j]` liefert das Element in der i-ten Zeile und j-ten Spalte. Ein Spalten- oder Zeilenname wird nicht mitgeliefert.

```
> Werte[2,]
Gewicht Alter groesse
    110    24     155
> Werte[,1]
  A   B   C   D   E
130 110 118 112 128
```

Spalten (bzw. Zeilen) werden als Ganzes ausgelesen, wenn der jeweils andere Index unspezifiziert bleibt. Ein Komma muss verwendet werden, um die gewünschte Dimension zu spezifizieren. Das Resultat ist (per Voreinstellung) stets ein Vektor und Namen werden übernommen.

```
> Werte[c(1,2), 2]
 A  B
26 24
> Werte[, c(1,3)]
  Gewicht groesse
A     130     140
B     110     155
C     118     142
D     112     175
E     128     170
```

Durch Angabe eines Index*vektors* für den Spalten- bzw. Zeilenindex erhält man die angegebenen Spalten bzw. Zeilen.

Das Resultat ist ein Vektor, wenn einer der Indizes eine einzelne Zahl ist; es ist eine Matrix, wenn beide Angaben Indexvektoren sind. Namen werden übernommen.

```
> Werte[-2, -3]
  Gewicht Alter
A     130    26
C     118    25
D     112    25
E     128    26
```

Negative Integer-Indizes wirken hier genauso wie bei Vektoren, nämlich ausschließend.

```
> Werte[c(F,F,T,F,F),]
Gewicht  Alter groesse
    118      25     142
> Werte[c(T,T,F,F,F), c(T,F,T)]
  Gewicht groesse
A     130     140
B     110     155
```

Die Auswahl von Elementen und ganzen Spalten sowie Zeilen kann auch durch logische Indexvektoren geschehen. Sie wirken analog wie bei Vektoren.

```
> Werte[c(F,T), c(T,F,T)]
  Gewicht groesse
B     110     155
D     112     175
```

Sind logische Indexvektoren nicht lang genug, werden sie zyklisch repliziert.

```
> Werte[1, "Gewicht"]
 Gewicht
     130
> Werte[, c("Gewicht", "Alter")]
  Gewicht Alter
A     130    26
B     110    24
C     118    25
D     112    25
E     128    26
```

Die Auswahl von Elementen und ganzen Spalten sowie Zeilen kann, wenn eine Benennung vorliegt, auch mit den Namen der Spalten und Zeilen geschehen.

```
> Werte[Werte[, "groesse"] > 160,
+ "Gewicht"]
  D   E
112 128
```

Verschiedene Indizierungsmethoden sind mit gewissen Einschränkungen kombinierbar.

```
> head(Werte, n = 2)
  Gewicht Alter groesse
A     130    26     140
B     110    24     155
> tail(Werte, n = 2)
  Gewicht Alter groesse
D     112    25     175
E     128    26     170
```

Analog zur Anwendung auf Vektoren (vgl. §2.6.2) liefern `head` und `tail` die oberen bzw. unteren `n` *Zeilen* einer Matrix. Voreinstellung für `n` ist 6. Negative Werte für `n` liefern den "head" bis auf die letzten bzw. den "tail" bis auf die ersten |`n`| Zeilen der Matrix.

2.8.6 Indizierte Zuweisungen zu Elementen, Spalten oder Zeilen

Indizierte Zuweisungen funktionieren für Matrizen analog zu denen für Vektoren (vgl. §2.6.3):

```
> Werte[i,j] <- v1

> Werte[,j] <- v2

> Werte[i,] <- v3
```

Hier wird in das Matrixelement (`i,j`) der Wert von `v1` geschrieben, falls er ein Skalar ist, dann in die `j`-te Matrixspalte der Inhalt von `v2`, falls er ein Vektor ist, dessen Länge zur Zeilenzahl der Matrix passt, und schließlich in die `i`-te Matrixzeile der Inhalt von `v3`, falls er ein Vektor mit zur Spaltenzahl der Matrix passender Länge ist.

Memo: Evtl. das "Base R cheat sheet" nutzen (via hier oder direkt hier).

2.8.7 Einige spezielle Matrizen: `diag`, `col` & `row`, `lower.tri` & `upper.tri`

```
> (y <- diag(1:3))
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
> diag(y)
[1] 1 2 3
```

Erhält `diag` einen Vektor als Argument, wird eine Diagonalmatrix mit den Elementen dieses Vektors auf der Hauptdiagonalen erzeugt.

Ist `diag`s Argument eine Matrix, wird ihre Hauptdiagonale als Vektor extrahiert. Die Zuweisungsform `diag(y) <- 4:6` ersetzt die Hauptdiagonale von `y` durch den zugewiesenen Vektor.

```
> col(y)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
[3,]    1    2    3
> row(y)
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
[3,]    3    3    3
```

`col` erwartet eine Matrix als Argument und generiert dazu eine gleich große Matrix, deren Einträge die Spaltenindizes ihrer Elemente sind. `row` macht entsprechendes mit Zeilenindizes.

```
> lower.tri(y)
      [,1]  [,2]  [,3]
[1,] FALSE FALSE FALSE
[2,]  TRUE FALSE FALSE
[3,]  TRUE  TRUE FALSE
> upper.tri(y, diag = TRUE)
      [,1]  [,2] [,3]
[1,]  TRUE  TRUE TRUE
[2,] FALSE  TRUE TRUE
[3,] FALSE FALSE TRUE
```

Ein schönes „Anwendungsbeispiel" für die beiden obigen Funktionen sind die Implementationen von `lower.tri` und `upper.tri` zur Erzeugung von logischen unteren oder oberen Dreiecksmatizen (je nach Wahl des Argumentes `diag` mit oder ohne der Diagonalen):

```
> lower.tri
function(x, diag = FALSE) {
 x <- as.matrix(x)
 if(diag) row(x) >= col(x) else row(x) > col(x)
 }
```

2.8.8 Ein paar wichtige Operationen der Matrixalgebra

In der folgenden Auflistung sei mindestens eines der Objekte `A` und `B` eine `numeric`-Matrix und wenn beide Matrizen sind, dann mit Dimensionen, die zu den jeweils betrachteten Operationen „passen“. D. h., `A` oder `B` können in manchen Situationen auch Vektoren oder Skalare sein, die dann von **R** in der Regel automatisch entweder als für die betrachtete Operation geeignete Zeilen- oder Spaltenvektoren interpretiert werden oder (z. B. im Fall von Skalaren) hinreichend oft repliziert werden, damit die Anzahlen der Elemente der an der Operation beteiligten Objekte zueinander passen:

`A + B`, `A - B`, `A * B`, `A / B`, `A^B`, `A %/% B`, `A %% B`	Die elementaren arithmetischen Operationen arbeiten – wie bei Vektoren (siehe §2.3.1) – *elementweise*, falls die Dimensionen von `A` und `B` „zueinanderpassen“. Sind die Dimensionen von `A` und `B` verschieden, gibt es eine Fehlermeldung. (Ist jedoch `A` oder `B` ein Vektor oder Skalar, werden seine Elemente zyklisch repliziert und mit den Matrixelementen in der Reihenfolge verknüpft, in der jene als aus den Matrixspalten zusammengesetzter Vektor gespeichert sind.) Beachte, dass der Hinweis auf Seite 22 zur Effizienz beim Potenzieren mit natürlichen Exponenten hier ganz besonders nützlich sein kann, da `A^n` alle Elemente von `A` potenziert.
`t(A)`	Die Transponierte der Matrix `A`, also A'.
`A %*% B`	Matrixprodukt der Matrizen `A` und `B`.
`crossprod(A, B)`	Kreuzprodukt von `A` mit `B`, d. h. $A'B$, aber i. d. R. etwas schneller als `t(A) %*% B`. Dabei ist `crossprod(A)` dasselbe wie `crossprod(A, A)`, aber **R**-intern effizienter als die Kombination von `t` und `%*%`.
`outer(A, B)` *Operatorform:* `A %o% B` *(Dabei ist* `o` *das kleine „O“ und nicht die Null!)*	Äußeres Produkt von `A` mit `B`, d. h., jedes Element von `A` wird mit jedem Element von `B` multipliziert, was $(a_{ij}B)_{ij}$ liefert. Für Spaltenvektoren ist es dasselbe wie `A %*% t(B)` und ergibt dann auch eine Matrix. Für die Verknüpfung der Elemente kann durch das hier nicht gezeigte Argument `FUN` jede binäre Operation angegeben werden; daher heißt `outer` auch *generalisiertes* äußeres Produkt. Voreinstellung ist `FUN = "*"` (Multiplikation), was für die Operatorform `%o%` die feste Einstellung ist. (Für Beispiele mit anderen binären Operationen siehe S. 49.)
`solve(A, B)`	Lösung des linearen Gleichungssystems $AX = B$.
`solve(A)`	Dies liefert die zu `A` inverse Matrix A^{-1}, falls sie existiert. Beachte: Die Berechnung von $AB^{-1}C$ durch `A %*% solve(B, C)` ist effizienter als durch `A %*% solve(B) %*% C`.
`chol(A)`	Choleski-Zerlegung von `A`.
`eigen(A)`	Eigenwerte und Eigenvektoren von `A`, d. h. Skalare λ_i und Vektoren v_i mit der Eigenschaft $Av_i = \lambda_i v_i$.
`kappa(A)`	Eine Konditionszahl von `A`.
`qr(A)`	QR-Zerlegung von `A` und „nebenbei“ Bestimmung des Rangs von `A`.
`svd(A)`	Singulärwertzerlegung von `A`.

Bemerkungen: Wichtige Informationen über weitere optionale Argumente, numerische Implementationen, gelegentliche „Verwandte“ etc. der obigen Funktionen sind auf ihren Hilfeseiten zu finden. Zu obigem ist (z. B.) in [86, Schott (1997)] diesbezügliche Mathematik zusammengestellt. Für speziell strukturierte Matrizen, wie z. B. Dreiecksmatrizen, symmetrische, dünn oder dicht besetzte, die typischerweise auch noch – sehr – groß sind, gibt es ein Paket namens `Matrix`, das **R**'s `matrix`-Konzept um Funktionen für den Zugriff auf sehr effiziente Algorithmen erweitert.

2.8.9 Effiziente Berechnung von Zeilen- bzw. Spaltensummen oder -mittelwerten (auch gruppiert): `colSums` & Verwandte sowie `rowsum`

`colSums(x)` `rowSums(x)` `colMeans(x)` `rowMeans(x)`	Spalten- bzw. zeilenweise Summen und arithmetische Mittel für ein `numeric`-Array `x` (also auch eine `numeric`-Matrix). Falls `x` eine Matrix ist, entsprechen die Funktionen (in dieser Reihenfolge) `apply(x, 2, sum)`, `apply(x, 1, sum)`, `apply(x, 2, mean)` bzw. `apply(x, 1, mean)` von §2.8.10, allerdings sind die `apply`-Versionen *erheblich* langsamer. Allen diesen Funktionen steht das Argument `na.rm` zur Verfügung. (Für die Varianz u. a. existieren in "base **R**" keine derartigen optimierten Funktionen. Jedoch scheint das Paket `matrixStats` dieses „Defizit" zu verkleinern.)
`rowsum(x, g)`	Die Zeilen der `numeric`-Matrix `x` werden für jedes Level des (Faktor-)Vektors `g` in Gruppen zusammengefasst. Dann wird für jede dieser Zeilengruppen die Summe einer jeden Spalte berechnet. Das Ergebnis ist eine Matrix, die so viele Zeilen hat wie verschiedene Werte in `g` sind und so viele Spalten wie `x` hat. (Beachte, dass offenbar zeilengruppierte *Spalten*summen gebildet werden, was – im Vergleich mit den obigen Funktionen für Zeilen- oder Spaltensummen – mit dem Namen dieser Funktion wenig assoziierbar ist.)

2.8.10 Zeilen- und spaltenweise iterative Anwendung von (nahezu) beliebigen Operationen: `apply`, `sweep` & `scale`

Das Kommando `apply(matrix, dim, FUN)` wendet die Funktion `FUN` iterativ auf die in `dim` spezifizierte Dimension von `matrix` an. (Die Argumente von `apply` haben eigentlich andere, aber weniger suggestive Namen.) Die Funktion `scale` (i. F. ohne Beispiel) erledigt für eine Matrix wahlweise entweder das spaltenweise Zentrieren (Subtraktion des Spaltenmittelwertes) oder das spaltenweise Standardisieren (Division durch die Spaltenstandardabweichung) oder beides.

```
> apply(Werte, 2, mean)
 Gewicht   Alter  groesse
   119.6    25.2    156.4

> apply(Werte, 2, sd)
 Gewicht   Alter  groesse
 9.09945 0.83666 15.88395

> apply(Werte, 2, sort)
     Gewicht Alter groesse
[1,]     110    24     140
[2,]     112    25     142
[3,]     118    25     155
[4,]     128    26     170
[5,]     130    26     175
```

Hier wird die Funktion `mean` durch `apply` wegen deren zweiten Arguments `2` auf die zweite Dimension, also die Spalten der (aus §2.8.4 stammenden) Matrix `Werte` angewendet, was die spaltenweisen Mittelwerte liefert. (Beachte aber §2.8.9!) Das analoge Beispiel mit `sd` liefert die empirischen Standardabweichungen der Spalten. (Memo: Die Zeilen sind die erste Dimension einer Matrix.)

Im dritten Beispiel wird jede Spalte von `Werte` sortiert und ...

```
> t(apply(Werte, 1, sort))
  Alter Gewicht groesse
A    26     130     140
B    24     110     155
C    25     118     142
D    25     112     175
E    26     128     170
> apply(Werte, 2, mean,
+ na.rm = TRUE)
....
```

... im vierten (zugegebenermaßen ziemlich unsinnigen) Beispiel wegen des Argumentes `1` jede Zeile. Beachte die Verwendung von `t`. Begründung: Das Resultat von `apply` ist stets eine Matrix von *Spalten*vektoren. Daher muss das Ergebnis bedarfsweise transponiert werden.

Benötigt die anzuwendende Funktion `FUN` weitere Argumente, so können diese in benannter Form an `apply` übergeben werden; sie werden sozusagen „durchgereicht". Beispiel: Das `na.rm`-Argument der Funktion `mean`.

```
> sweep(Werte, 2, 1:3 * 10)
  Gewicht Alter groesse
A     120     6     110
B     100     4     125
C     108     5     112
D     102     5     145
E     118     6     140
```

Der Befehl `sweep(matrix, dim, vector, FUN = "-")` „kehrt“ (“to sweep” = kehren) den Vektor `vector` elementweise entlang der Dimension `dim` aus der Matrix `matrix` entsprechend der Funktion `FUN` aus. Im Beispiel wird der Vektor $(10, 20, 30)'$ elementweise von jeder Spalte von `Werte` subtrahiert (= Voreinstellung). (Siehe auch obige Bemerkung zu `scale`.)

2.8.11 Kovarianz und Korrelation zwischen Spalten: `cov` und `cor`

```
> cov(Werte)   # = var(Werte)
         Gewicht Alter groesse
Gewicht    82.8  7.10  -40.30
  Alter     7.1  0.70   -0.35
groesse   -40.3 -0.35  252.30

> cor(Werte)
          Gewicht Alter  groesse
Gewicht  1.0000  0.9326 -0.2788
  Alter  0.9326  1.0000 -0.0263
groesse -0.2788 -0.0263  1.0000
```

`cov` und `var` bzw. `cor` liefern für eine Matrix die empirische Kovarianz- bzw. Korrelationsmatrix ihrer Spalten. Zur Behandlung von `NA`s steht beiden das Argument `use` zur Verfügung. Wird ihm der Wert `"complete.obs"` übergeben, werden die Zeilen, in denen ein `NA` auftritt, ganz eliminiert. Der Wert `"pairwise.complete.obs"` erzwingt die maximal mögliche Nutzung aller verfügbaren Elemente pro Variable (= Spalte). Details hierzu stehen auf der Hilfeseite. (Für weitere auf Matrizen anwendbare Funktionen siehe auch „Beachte“ auf S. 24 oben.)

Zur Erinnerung: Die empirische Kovarianz zweier (Daten-)Vektoren $\mathbf{x} \equiv (x_1, \ldots, x_n)'$, $\mathbf{y} \equiv (y_1, \ldots, y_n)' \in \mathbb{R}^n$ mit $n \geq 2$ ist $\hat{\sigma}(\mathbf{x}, \mathbf{y}) := \frac{1}{n-1} \sum_{r=1}^{n} (x_r - \bar{x}.)\,(y_r - \bar{y}.)$, wobei $\bar{x}. \equiv \frac{1}{n} \sum_{r=1}^{n} x_r$ und $\bar{y}.$ analog definiert ist.

Zu $p \geq 2$ Vektoren $\mathbf{x}_1, \ldots, \mathbf{x}_p \in \mathbb{R}^n$, welche oft spaltenweise als Matrix $\mathbf{M} \equiv (\mathbf{x}_1 | \cdots | \mathbf{x}_p)$ zusammengefasst werden, ist ihre empirische Kovarianzmatrix gegeben durch die $(p \times p)$-Matrix $\widehat{\mathrm{Cov}}(\mathbf{M}) := (\hat{\sigma}(\mathbf{x}_i, \mathbf{x}_j))_{1 \leq i,j, \leq p}$. Analog ist die empirische Korrelationsmatrix $\widehat{\mathrm{Cor}}(\mathbf{M})$ definiert.

2.8.12 Erzeugung spezieller Matrizen mit Hilfe von `outer`

```
> outer(1:3, 1:5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
> outer(1:3, 1:5, FUN = "+")
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    3    4    5    6
[2,]    3    4    5    6    7
[3,]    4    5    6    7    8
> outer(1:3, 1:5, paste, sep = ":")
      [,1]  [,2]  [,3]  [,4]  [,5]
[1,] "1:1" "1:2" "1:3" "1:4" "1:5"
[2,] "2:1" "2:2" "2:3" "2:4" "2:5"
[3,] "3:1" "3:2" "3:3" "3:4" "3:5"
> x <- c(-1, 0, -2)
> y <- c(-1, 2, 3, -2, 0)
> 1 * (outer(x, y, FUN = "<="))
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    0    1
[2,]    0    1    1    0    1
[3,]    1    1    1    1    1
```

Das äußere Produkt liefert für zwei Vektoren die Matrix der Produkte aller möglichen paarweisen Kombinationen der Vektorelemente, d. h. das Produkt aus einem Spaltenvektor mit einem Zeilenvektor. (Hier ein Teil des kleinen „1×1“.)

Das generalisierte äußere Produkt erlaubt an Stelle der Multiplikation die Verwendung einer binären Operation oder einer Funktion, die mindestens zwei Vektoren als Argumente verwendet. Im zweiten Beispiel ist dies die Addition und im dritten die Funktion `paste`, der außerdem noch das optionale Argument `sep = ":"` mit übergeben wird. (Die Ergebnisse sind „selbsterklärend“.)

Die Matrix der Indikatoren $1_{\{x_i \leq y_j\}}$ für alle $1 \leq i \leq 3$ und $1 \leq j \leq 5$ für die Beispielvektoren `x` und `y`.

2.9 Listen: Konstruktion, Indizierung und Verwendung

Die bisher kennengelernten Datenobjekte Vektor und Matrix (bzw. Array) enthalten jeweils (atomare) Elemente desselben Modus' (`numeric`, `logical` oder `character`). Es aber kann nötig und sinnvoll sein, auch (nicht-atomare) Elemente unterschiedlicher Modi in einem neuen Objekt – einer „Liste" – zusammenzufassen. Dies wird durch den rekursiven Vektormodus `list` ermöglicht, der durch die Funktion `list` erzeugt wird. Insbesondere für Funktionen (eingebaut oder selbstdefiniert) sind Listen der bevorzugte Objekttyp, um komplexere Berechnungsergebnisse an die aufrufende Stelle zurückzugeben. Eine andere, bereits gesehene Anwendung für Listen ist die Vergabe von Zeilen- und Spaltennamen an Matrizen (siehe §2.8.3, Seite 43).

Die Länge einer Liste ist die Anzahl ihrer Elemente. Die Elemente einer Liste sind nummeriert und unter Verwendung eckiger Klammern (wie Vektoren) indizierbar. Dabei liefern einfache eckige Klammern `[ ]` die indizierten Elemente wieder als Liste zurück. Falls nur ein Listenelement (z. B. durch `[2]`) indiziert wurde, erhält man also eine Liste der Länge 1. Um auf ein einzelnes Listenelement zuzugreifen, ohne es in eine Listenstruktur eingebettet zu bekommen, muss der Index des gewünschten Elements in *doppelte* eckige Klammern `[[ ]]` gesetzt werden. Eine Benennung der Listenelemente ist ebenfalls möglich; sie heißen dann Komponenten. Der Zugriff auf sie geschieht durch den jeweiligen Komponentennamen in einfachen oder doppelten eckigen Klammern oder mit Hilfe des Komponentenoperators `$`.

Beispiele mögen das Konzept der Listen und die Indizierungsmethoden erläutern, wobei die folgenden Objekte verwendet werden:

```
> personen
[1] "Peter" "Paul"  "Mary"  "Tic"

> alter
[1] 12 14 13 21
```

```
> m
     [,1] [,2] [,3] [,4]
[1,]  130  110  118  112
[2,]   26   24   25   25
```

2.9.1 Erzeugung und Indizierung: `list`, `[[ ]]`, `head` bzw. `tail`

```
> (daten <- list(personen, alter, m))
[[1]]:
[1] "Peter" "Paul"  "Mary"  "Tic"

[[2]]:
[1] 12 14 13 21

[[3]]:
     [,1] [,2] [,3] [,4]
[1,]  130  110  118  112
[2,]   26   24   25   25
```

Die Funktion `list` fasst ihre Argumente zu einer Liste zusammen. Hier eine dreielementige Liste, deren Elemente ein `character`-Vektor, ein `numeric`-Vektor und eine `numeric`-Matrix sind.
In der Ausgabe der Liste sind in den doppelten eckigen Klammern die Indices der Listenelemente zu sehen. (Das nachgestellte „`:`" hat keine Bedeutung.)

```
> length(daten)
[1] 3
```

Die Länge einer Liste (= Anzahl ihrer Elemente) bestimmt `length`.

```
> daten[2]
[[1]]:
[1] 12 14 13 21
```

`[i]` ist die *Liste*, die das i-te Element der Ausgangsliste enthält.

```
> daten[[1]]
[1] "Peter" "Paul"  "Mary"  "Tic"
```

`[[i]]` liefert das i-te Element der Ausgangsliste (hier also den Vektor `personen`).

```
> daten[[2]][2:3]
[1] 14 13
```

`[[i]][j]` ist das j-te Element des i-ten Elements der Ausgangsliste (hier also das zweite und dritte Element des ursprünglichen Vektors `alter`).

`> daten[[3]][2, 4]` `[1] 25`	Wenn das i-te Element der Ausgangsliste eine Doppelindizierung zulässt, ist `[[i]][j,k]` das (j,k)-Element jenes i-ten Elements (hier also das $(2,4)$-Element der ursprünglichen Matrix `m`).
`> head(....)` `> tail(....)`	`head` und `tail` liefern bei Listen die ersten bzw. letzten `n` Listenelemente zurück; vgl. §2.6.2.

Bemerkung: Eine *sehr* nützliche Funktion, um sich die Struktur von umfangreichen Listen (oder anderen Objekten) übersichtlich in abgekürzter Form anzusehen, ist `str` (siehe §§2.9.3 und 2.10.5).

Memo: "Base R cheat sheet" (via hier oder direkt von hier).

2.9.2 Benennung von Listenelementen und ihre Indizierung: `names` und `$`

`> names(daten) <- c("Personen", "Alter",` `+ "Werte");  daten` `$Personen:` `[1] "Peter" "Paul"  "Mary"  "Tic"` `$Alter:` `[1] 12 14 13 21` `$Werte:` `     [,1] [,2] [,3] [,4]` `[1,]  130  110  118  112` `[2,]   26   24   25   25`	Die Benennung der Listenelemente erfolgt mittels der Funktion `names`. Die benannten Listenelemente werden Komponenten genannt In der Ausgabe der Liste sind die mit doppelten eckigen Klammern versehenen Indices verschwunden und durch die Komponentennamen samt einem vorangestellten `$`-Zeichen und einem nachgestellten „`:`“ ersetzt. Beachte, dass die Komponentennamen nicht in Hochkommata stehen!
`> names(daten)` `[1] "Personen" "Alter"  "Werte"`	Die Anwendung von `names` liefert die Komponentennamen.
`> daten$Personen` `[1] "Peter" "Paul"  "Mary"  "Tic"`	Zugriff auf die `"Personen"` benannte Komponente von `daten` mittels des Komponentenoperators `$`.
`> daten[["Personen"]]` `[1] "Peter" "Paul"  "Mary"  "Tic"`	Analoge Wirkung der Indizierung durch doppelte eckige Klammern und Komponentennamen.
`> daten["Personen"]` `$Personen` `[1] "Peter" "Paul"  "Mary"  "Tic"`	Beachte den Unterschied zur Indizierung mit einfachen eckigen Klammern und Komponentennamen!
`> daten$Werte[2,2]` `[1] 24`	Kombination der Indizierungsmethoden: `$Werte[2,2]` ist das $(2,2)$-Element der Komponente `"Werte"`.
`> (D <- list(Land = c("Baden-Wuerttemberg",` `+ "Bayern", "Berlin", ....),` `+ "Bevoelkerung in Mio." = c(10.7, 12.4,` `+ 3.4, ....)))` `$Land:` ` [1] "Baden-Wuerttemberg"  "Bayern" ....` `$‘Bevoelkerung in Mio.‘:` ` [1] 10.7  12.4 ....`	Erzeugung einer Liste mit zwei Komponenten, die bereits bei der Listenerzeugung die Komponentennamen `Land` und `Bevoelkerung in Mio.` erhält (also ohne Verwendung der Funktion `names`). (Beachte die Notwendigkeit von Hochkommas bei Komponentennamen mit Leerzeichen.)

Beachte:

- Benannte Listenelemente bieten denselben, bereits in Punkt 4 in §2.6.1 für benannte Vektorelemente erwähnten Vorteil bei der Verwendung von Indexvektoren aus Zeichenketten.
- Indizierung von Komponenten unter Verwendung von `character`-Variablen, die die Komponentennamen enthalten, geht nur mit `[[ ]]` und nicht mit `$`. Beispiel:

```
> idx <- "Personen";   daten[[idx]]     # Funkioniert wie gewuenscht.
[1] "Peter" "Paul"  "Mary"  "Tic"
> daten$idx   # Liefert eine korrekte Antwort, aber wohl nicht das Gewuenschte.
NULL
```

2.9.3 Indizierte Zuweisungen, Elementelöschung sowie Konkatenation von Listen

Nach der Benennung ihrer Komponenten (im vorherigen Paragraph) hat `daten` nun die folgende Struktur:

```
> str(daten)
List of 3
 $ Personen: chr [1:4] "Peter" "Paul" "Mary" "Tic"
 $ Alter   : num [1:4] 12 14 13 21
 $ Werte   : num [1:2, 1:4] 130 26 110 24 118 25 112 25
```

- Es ist möglich, weitere Komponenten hinzuzufügen, indem indizierte Zuweisungen an noch nicht existente Komponenten vorgenommen wird, egal ob mit dem `$`-Operator oder mittels der doppelten eckigen Klammern `[[ ]]`:

```
> daten$Unsinn <- "Blah, blah, blah";     daten[["WeitererUnsinn"]] <- "Oje"
> str(daten)
List of 5
 $ Personen      : chr [1:4] "Peter" "Paul" "Mary" "Tic"
 $ Alter         : num [1:4] 12 14 13 21
 $ Werte         : num [1:2, 1:4] 130 26 110 24 118 25 112 25
 $ Unsinn        : chr "Blah, blah, blah"
 $ WeitererUnsinn: chr "Oje"
```

- Gelöscht werden können einzelne Listenkomponenten (z. B. auch) durch die indizierte Zuweisung des leeren Objektes `NULL`:

```
> daten$Unsinn <- NULL;    daten[["WeitererUnsinn"]] <- NULL;    str(daten)
List of 3
 $ Personen: chr [1:4] "Peter" "Paul" "Mary" "Tic"
 $ Alter   : num [1:4] 12 14 13 21
 $ Werte   : num [1:2, 1:4] 130 26 110 24 118 25 112 25
```

  Das hätte auch mit `daten[[4]] <- daten[[5]] <- NULL` funktioniert, aber natürlich vorausgesetzt, dass man die Elementenummern weiß!

- Auch durch Konkatenation mit einer weiteren Liste kann eine Liste verlängert werden:

```
> woher <- list(Erfasser = "Eichner",
+               Datum = c(Jahr = 2014, Monat = 5, Tag = 16),
+               Erfunden = TRUE)
> str(daten2 <- c(daten, woher))
List of 6
 $ Personen: chr [1:4] "Peter" "Paul" "Mary" "Tic"
 $ Alter   : num [1:4] 12 14 13 21
```

```
 $ Werte   : num [1:2, 1:4] 130 26 110 24 118 25 112 25
 $ Erfasser: chr "Eichner"
 $ Datum   : Named num [1:3] 2014 5 16
  ..- attr(*, "names")= chr [1:3] "Jahr" "Monat" "Tag"
 $ Erfunden: logi TRUE
```

Soeben sind die Komponenten `Personen`, `Alter` und `Werte` der Liste `daten` und die Komponenten `Erfasser`, `Datum` und `Erfunden` der Liste `woher` sozusagen auf derselben Ebene in der Liste `daten2` zusammengebaut worden.

Beachten Sie den Unterschied zu:

```
> str(daten3 <- c(daten, Info = list(woher)))
List of 4
 $ Personen: chr [1:4] "Peter" "Paul" "Mary" "Tic"
 $ Alter   : num [1:4] 12 14 13 21
 $ Werte   : num [1:2, 1:4] 130 26 110 24 118 25 112 25
 $ Info    :List of 3
  ..$ Erfasser: chr "Eichner"
  ..$ Datum   : Named num [1:3] 2014 5 16
  .. ..- attr(*, "names")= chr [1:3] "Jahr" "Monat" "Tag"
  ..$ Erfunden: logi TRUE
```

Hier wurde an die Liste `daten` als weitere Komponente namens `Info` die Liste `woher` angehängt (und das Ergebnis unter `daten3` abgelegt). D. h., im Gegensatz zum vorherigen Beispiel sind die Komponenten `Erfasser`, `Datum` und `Erfunden` nun nicht auf derselben Ebene wie `Personen`, `Alter` und `Werte`, sondern quasi eine Ebene darunter, genauer „unter“ der Komponente `Info`.

Die Arbeit mit der rekursiven Struktur `list` kann insbesondere bei stark geschachtelten Listen anfangs recht unübersichtlich und gewöhnungsbedürftig sein. In dieser Hinsicht eine eventuell sehr nützliche Funktion zur (auch rekursiven!) Modifikation von Listen, sprich ihren Komponenten ist `modifyList`, für deren Leistungsumfang wir jedoch auf die Hilfeseite verweisen.

Beachten Sie, dass Listenkomponenten selbst auch Listen sein können, wie gerade in `daten3` gesehen (daher der Namen rekursive Struktur). Dafür ist eine sogar rekursive Indizierung erlaubt/möglich, die aber mit Vorsicht zu verwenden (bzw. zu genießen) ist:

```
> daten3[[c(4, 2, 3)]]          # Welches Element ist das wohl?
[1] 16
```

2.9.4 Elementweise iterative Anwendung von Operationen: `lapply`, `sapply` & Co.

Ähnlich zur Funktion `apply`, die eine zeilen- und spaltenweise iterative Anwendung von Funktionen auf Matrizen ermöglicht, existieren zwei Funktionen, die dies für die Elemente einer Liste realisieren. Es handelt sich hierbei um `lapply` und `sapply`.

```
> lapply(daten, class)
$Personen:
[1] "character"

$Alter:
[1] "numeric"

$Werte:
[1] "matrix"
```

`lapply` erwartet als erstes Argument eine Liste, auf deren jedes Element sie das zweite Argument (hier die Funktion `class`) anwendet. Das Ergebnis ist eine Liste der Resultate der Funktionsanwendungen (hier die Klassen der Elemente von `daten`). Die Ergebnisliste erhält die Komponentennamen der Eingabeliste, falls jene eine benannte Liste ist.

`> sapply(daten, class)` `   Personen     Alter    Werte` `"character" "numeric" "matrix"`	`sapply` funktioniert wie `lapply`, simplifiziert aber die Ergebnisstruktur, wenn möglich; daher ist hier das Ergebnis ein *Vektor*.
`> x <- list(A = 1:10, B = exp(-3:3),` `+ C = c(TRUE, FALSE, FALSE, TRUE))` `> lapply(x, mean)` `$A` `[1] 5.5` `$B` `[1] 4.535125` `$C` `[1] 0.5`	Weitere Beispiele zur Wirkung von `sapply` und `lapply`: Hier wird die Funktion `mean` auf jede Komponente der Liste `x` angewendet.
`> lapply(x, quantile, probs = 1:3/4)` `$A` ` 25%  50%  75%` `3.25 5.50 7.75` `$B` `      25%       50%       75%` `0.2516074 1.0000000 5.0536690` `$C` `25% 50% 75%` `0.0 0.5 1.0`	Sollen der elementweise anzuwendenden Funktion (hier `quantile`, vgl. §2.3.3) weitere Argumente (hier `probs = 1:3/4`) mitgeliefert werden, so sind diese in der Form *Name = Wert* durch Komma getrennt hinter dem Namen jener Funktion aufzuführen.
`> sapply(x, quantile, probs = 1:3/4)` `         A           B    C` `25%   3.25  0.25160736  0.0` `50%   5.50  1.00000000  0.5` `75%   7.75  5.05366896  1.0`	Dito, aber `sapply` simplifiziert die Ergebnisstruktur zu einer *Matrix*, da die jeweiligen Ergebnisse (hier `numeric`-)Vektoren derselben Länge sind. Beachte wie die Ergebnisvektoren *spaltenweise* zu einer Matrix zusammengefasst werden und wie Zeilen- und Spaltennamen der Matrix zustandekommen.

Bemerkungen: Eine spezialisierte Version von `sapply` namens `vapply` erlaubt die Angabe des Typs des Rückgabewertes, was sicherer und gelegentlich auch etwas schneller sein kann.
Es gibt eine rekursive Version namens `rapply` und eine „multivariate“ Variante von `lapply` und `sapply` namens `mapply`, für deren mächtige Funktionalitäten wir auf die Hilfeseite verweisen.

2.10 Data Frames: Eine Klasse „zwischen“ Matrizen und Listen

Ein Data Frame ist eine Liste der Klasse `data.frame`, die benannte Elemente, also Komponenten hat. Diese Komponenten müssen atomare Vektoren der Modi `numeric`, `logical` bzw. `character` oder der Klassen `factor` bzw. `ordered` (also Faktoren) sein, die alle die gleiche Länge haben. Eine andere Interpretation ist die einer „verallgemeinerten“ Matrix mit benannten Spalten (und Zeilen): Innerhalb einer jeden Spalte stehen zwar (atomare) Elemente des gleichen Modus bzw. derselben Klasse, aber von Spalte zu Spalte können Modus bzw. Klasse variieren.

Data Frames sind das „Arbeitspferd“ vieler statistischer (Modellierungs-)Verfahren in **R**, da sie hervorragend geeignet sind, die typische Struktur p-dimensionaler Datensätze vom Stichprobenumfang n widerzuspiegeln: Jede der n Zeilen enthält einen p-dimensionalen Beobachtungsvektor, der von einer der n Untersuchungseinheiten (auch Fälle oder Englisch “cases” genannt) stammt. Jede der p Komponenten (= Spalten) enthält die n Werte, die für eine der p Variablen (= Merkmale) registriert wurden.

Der im **R**-Paket `rpart` enthaltene Datensatz `cu.summary` (Autodaten aus der 1990er Ausgabe des “Consumer Report”) ist ein Beispiel eines Data Frames (falls `rpart` noch nicht installiert, siehe Abschnitt 1.7):

```
> data(cu.summary,       # Stellt eine Kopie (!) des Objektes "cu.summary" aus
+ package = "rpart")     # dem Paket "rpart" im workspace zur Verfuegung.
> head(cu.summary)
                Price   Country Reliability Mileage  Type
Acura Integra 4 11950     Japan Much better      NA Small
Dodge Colt 4     6851     Japan        <NA>      NA Small
Dodge Omni 4     6995       USA  Much worse      NA Small
Eagle Summit 4   8895       USA      better      33 Small
Ford Escort   4  7402       USA       worse      33 Small
Ford Festiva 4   6319     Korea      better      37 Small
```

Hinweise: Alle bereits in “base **R**” zur Verfügung stehenden Datensätze werden durch `library( help = "datasets")` aufgelistet. Mit `?`*datensatzname* wird die Hilfeseite zum jeweiligen Datensatz gezeigt.

2.10.1 Indizierung: `[ ]`, `$`, `head` und `tail` sowie `subset`

Der Zugriff auf einzelne Elemente, ganze Zeilen oder Komponenten eines Data Frames kann wie bei Matrizen mittels der Zeile-Spalte-Indizierung `[i, j]`, `[i, ]` oder `[, j]` geschehen bzw. über Zeilen- und Komponentennamen gemäß `["zname", "sname"]`, `["zname", ]` oder `[, "sname"]`. Die Spalten (und *nur* sie), da sie Komponenten einer Liste sind, können mit Hilfe des Komponentenoperators `$` über ihre Namen indiziert werden (via *dataframe*`$`*name*). Beispiele:

```
> cu.summary[21,1]
[1] 8695
> cu.summary["Eagle Summit 4", "Mileage"]
[1] 33
```

Zugriff auf einzelne Elemente durch matrixtypische Zeile-Spalte-Indizierung bzw. -Benennung.

```
> cu.summary["GEO Metro  3", ]
            Price Country Reliability ....
GEO Metro  3  6695   Japan        <NA> ....
```

Matrixtypische Auswahl einer ganzen Zeile durch Zeilenbenennung. Beachte, dass die Spaltennamen übernommen werden.

```
> cu.summary$Country
  [1] Japan     Japan     USA     ....
 ....
[115] Japan     Japan     Germany
10 Levels: Brazil England France .... USA
```

Listentypische Auswahl einer Komponente über ihren Namen. Beachte, dass beim `$`-Operator keine Hochkommata verwendet werden.

`> head(....)` `> tail(....)`	Sie liefern bei Data Frames (wie bei Matrizen; vgl. §2.8.5) die ersten bzw. letzten `n` Zeilen zurück.
`> subset(cu.summary, subset = Price < 6500,` `+ select = c(Price, Country, Reliability))` `Price Country Reliability` `Ford Festiva 4 6319 Korea better` `Hyundai Excel 4 5899 Korea worse` `Mitsubishi Precis 4 5899 Korea worse` `Subaru Justy 3 5866 Japan <NA>` `Toyota Tercel 4 6488 Japan Much better`	`subset` wählt über ihr Argument `subset`, das (im Ergebnis) einen `logical`-Vektor erwartet, die Zeilen des Data Frames aus und über ihr Argument `select`, das (im Ergebnis) einen die Spalten des Data Frames indizierenden Vektor erwartet, eben jene aus. (Das Argument `select` ist mächtiger als hier gezeigt; siehe Hilfeseite.)

Beachte: `subset` wird nur für den interaktiven Gebrauch empfohlen; für die Programmierung die Verwendung von `[`. Wie erreichen Sie den oben gezeigten Auswahleffekt ohne `subset`?

Hinweis: Bei der Extraktion eines Teils eines Data Frames, der einen oder mehrere Faktor/en enthält, werden Levels des/der Faktors/en, die im extrahierten Teil nicht mehr „benötigt“ würden, da sie darin nicht mehr auftreten, im Allgemeinen *nicht* entfernt. Hier hilft die Funktion `droplevels`, da sie aus jedem Faktor des Teil-Data Frames die unbenutzen Levels eliminiert.

Memo: Auch zu Data Frames enthält das “Base R cheat sheet” (via hier oder direkt hier) einige Gedächtnisstützen.

2.10.2 Erzeugung: `data.frame`, `expand.grid`

Data Frames können zusammengesetzt werden aus bereits existierenden Vektoren, Matrizen, Listen und anderen Data Frames, und zwar standardmäßig mit der Funktion `data.frame`. Dabei müssen die „Dimensionen“ der zusammenzusetzenden Objekte i. d. R. zueinanderpassen. Ausnahmen sind auf der Hilfeseite beschrieben.

Beim „Einbau“ von Matrizen, Listen und anderen Data Frames liefert jede Spalte bzw. Komponente eine eigene Komponente des neuen Data Frames. Dazu müssen Matrizen und Data Frames dieselbe Zeilenzahl haben und die Komponenten von Listen dieselbe Länge (die wiederum alle mit der Länge eventuell hinzugefügter Vektoren übereinstimmen müssen). `character`-Vektoren werden beim Einbau in einen Data Frame (per Voreinstellung) belassen (und nicht mehr zu Faktoren konvertiert wie es in **R**-Versionen unter 4.0.0 der Fall war).

Im Fall von Vektoren können für die Komponenten (= Spalten) explizit Namen angegeben werden, gemäß der Form *Name = Wert*. Wird dies nicht getan, übernehmen die Data-Frame-Komponenten den Objektnamen des jeweiligen Vektors oder werden von **R** automatisch mit einem syntaktisch zulässigen und eineindeutigen (aber häufig furchtbaren) Namen versehen, der gelegentlich mit einem `X` beginnt.

Als Zeilennamen des Data Frames werden die Elementenamen (eines Vektors) oder Zeilennamen der ersten Komponente, die so etwas besitzt, übernommen, ansonsten werden die Zeilen durchnummeriert.

In der nächsten Tabelle werden folgende Beispielobjekte verwendet:

```
> hoehe
[1] 181 186 178 175
> Gewicht
[1] 130 110  63  59
> Diaet
[1]  TRUE FALSE FALSE  TRUE
```

```
> Werte
     Gewicht Alter groesse
[1,]     105    26     171
[2,]      83    24     176
[3,]      64    25     168
[4,]      58    25     165
```

```
> (W.df <- data.frame(Groesse = hoehe,
+ Gewicht, Diaet, BlutG = c("A", "A", "AB",
+ "0"), Rh = c("R+", "R-", "R-", "R-")))
  Groesse Gewicht Diaet BlutG Rh
1     181     130  TRUE     A R+
2     186     110 FALSE     A R-
3     178      63 FALSE    AB R-
4     175      59  TRUE     0 R-
```

Erzeugung eines Data Frames mit den explizit benannten Komponenten `Groesse`, `BlutG` und `Rh` sowie den Komponenten `Gewicht` und `Diaet`, die als Namen den Objektnamen des entsprechenden Vektors erhalten.

```
> (W2.df <- data.frame(Werte))
  Gewicht Alter groesse
1     105    26     171
2      83    24     176
3      64    25     168
4      58    25     165
```

Erzeugung eines Data Frames aus einer bereits existierenden Matrix. Die Namen von Spalten werden von der Matrix übernommen, falls sie existieren.

Beachte: Bei Betrachtung des obigen Beispiels mit dem `character`-Vektor der Blutgruppen `c("A", "A", "AB", "0")` fällt auf, dass in dem Data Frame `W.df` die Hochkommata des Modus' `character` verloren gegangen zu sein scheinen. Es ist also auf diese Weise kein Unterschied zwischen einer Faktor- und einer `character`-Komponente zu erkennen. (Siehe hierzu aber §2.10.5!)

In manchen Situationen ist es notwendig, zu k (Faktor-)Vektoren mit je $n_1, n_2, \ldots, n_k$ verschiedenen Elementen (bzw. Levels) alle möglichen $n_1 \cdot n_2 \cdots n_k$ Kombinationen zu generieren, d. h. ihr kartesisches Produkt zu bilden – und zwar nützlicherweise, wie sich noch herausstellen wird, als Zeilen eines Data Frames. Dies könnte z. B. der Fall sein für die Erfassung von Daten aus Experimenten, in denen eine Messgröße von Interesse unter verschiedenen Bedingungen erhoben wurde, welche durch die Kombinationen von verschiedenen Levels mehrerer (Einfluss-)Faktoren charakterisiert werden. Oder wenn man eine Funktion, die von mindestens zwei Variablen abhängt, auf einem vollständigen endlichen *Gitter* aller Kombinationen ausgewählter Werte dieser Variablen auswerten will, aber die dabei zu durchlaufenden Werte der Variablen als separate Vektoren gespeichert hat.

Hier ist die Funktion `expand.grid` das Werkzeug der Wahl, denn sie erzeugt aus jenen separaten Vektoren das entsprechende Gitter (Engl.: “grid”).

Beispiel: Soll eine Funktion mit drei Argumenten für alle Kombinationen der drei Strahlungswerte in $S = \{0, 50, 100\}$ mit den fünf Temperaturwerten in $T = \{0, 10, 20, 30, 40\}$ und den zwei Wind-„Werten“ in $W = \{$„ja“, „nein“$\}$ ausgewertet werden, und sind S, T und W als separate Vektoren à la

```
> Strahl <- seq(0, 100, by = 50);     Temp <- seq(0, 40, by = 10)
> Wind <- c("ja", "nein")
```

gespeichert, dann erhalten wir das gewünschte Gitter, sprich das kartesisches Produkt $S \times T \times W$ mit seinen $3 \cdot 5 \cdot 2 = 30$ Elementen als Data Frame mit (frei benennbaren Komponenten) durch

```
> expand.grid(Strahlung = Strahl, Temperatur = Temp, Wind = Wind)
   Strahlung Temperatur Wind
1          0          0   ja
2         50          0   ja
3        100          0   ja
4          0         10   ja
5         50         10   ja
6        100         10   ja
....
```

```
15       100         40   ja
16         0          0 nein
....
29        50         40 nein
30       100         40 nein
```

Offenbar durchläuft die erste Komponente des resultierenden Data Frames ihren Wertebereich vollständig, bevor die zweite Komponente auf ihren nächsten Wert „springt“. Analog ändert sich der Wert einer j-ten Komponente, falls vorhanden, erst auf den nächsten, wenn sämtliche Kombinationen in den ersten $j-1$ Komponenten durchlaufen sind, usw.

Hinweis: Zu den verwandten Problemen der Erzeugung aller möglichen m-elementigen Kombinationen, sprich Teilmengen bzw. ungeordneten Auswahlen aus *einer* n-elementigen (Grund-) Menge und der Berechnung ihrer Anzahl mit Hilfe der Funktionen `combn` bzw. `choose` siehe §2.3.4.

2.10.3 Indizierte Zuweisungen

Die indizierte Zuweisung zu einzelnen Elementen funktioniert für Data Frames analog zu der für Matrizen (vgl. §2.8.6). Die indizierte Zuweisung zu Spalten kann einerseits wie für Matrizen erfolgen (da Data Frames quasi „verallgemeinerte“ Matrizen sind) und andererseits auch wie für Listen, da Data Frames spezielle Listen sind (vgl. §2.9.3). Allerdings ist dabei darauf zu achten, dass die Zuweisung von „neuen“ Inhalten zu den im Data Frame spaltenweise herrschenden Modi passen!

2.10.4 Zeilen- & Spaltennamen: `dimnames`, `row.names` & `case.names`, `colnames` & `names`

Auf Zeilen- und Spaltennamen von Data Frames kann analog zu denen von Matrizen mit `dimnames`, `rownames` bzw. `colnames` zugegriffen werden. Allerdings gibt es ein paar auf Data Frames spezialisierte Versionen, nämlich `row.names` oder `case.names` für die Zeilen-, sprich Fallnamen bzw. (einfach) `names` für die Spalten-, sprich Variablennamen. Diese Funktionsnamen orientieren sich an der am Anfang von Abschnitt 2.10 kurz erwähnten Interpretation der Struktur von Data Frames.

2.10.5 “Summary statistics” und Struktur eines Data Frames: `summary` und `str`

Die Funktion `summary` hat eine spezielle Methode für Data Frames, sodass sie „direkt“ auf einen Data Frame anwendbar ist und geeignete “summary statistics” einer jeden Komponente des Data Frames in recht übersichtlicher Form liefert (siehe auch Seite 23 bzgl. `summary`):

Für `numeric`-Komponenten werden als arithmetische “summary statistics” ihre „6-Zahlenzusammenfassung“ bestimmt. Für Faktoren und geordnete Faktoren werden die Häufigkeitstabellen ihrer Levels bzw. Ausschnitte daraus angegeben, falls die Tabellen zu groß sind, weil mehr als sechs Levels auftreten. (Die Anzahl der maximal anzuzeigenden Faktorlevels lässt sich aber mit dem `summary`-Argument `maxsum` ändern.) Die Reihenfolge der Levels in einer Häufigkeitstabelle entspricht dabei der Levelssortierung des Faktors, wenn er so viele Levels hat wie angezeigt werden; ansonsten werden sie in der Reihenfolge ihrer absteigenden Häufigkeiten gezeigt. (Für Komponenten der Klasse `AsIs` (hier nicht gezeigt) wird die Länge dieser Komponente, ihre Klasse und ihr Modus, also `character` ausgegeben; ebenso für `character`-Komponenten.) Beispiel:

```
> summary(cu.summary)
    Price            Country        Reliability    Mileage          Type
Min.   : 5866   USA      :49   Much worse :18   Min.   :18.00   Compact:22
1st Qu.:10125   Japan    :31   worse      :12   1st Qu.:21.00   Large  : 7
Median :13150   Germany  :11   average    :26   Median :23.00   Medium :30
Mean   :15743   Japan/USA: 9   better     : 8   Mean   :24.58   Small  :22
3rd Qu.:18900   Korea    : 5   Much better:21   3rd Qu.:27.00   Sporty :26
Max.   :41990   Sweden   : 5   NA's       :32   Max.   :37.00   Van    :10
                (Other)  : 7                    NA's   :57.00
```

Die bereits im Zusammenhang mit Listen (auf Seite 51) erwähnte Funktion `str`, wie Struktur, ist auch für größere Data Frames eine *hervorragende* Möglichkeit, Informationen über ihre Struktur und ihren Inhalt in kompakter, abgekürzter Form darzustellen:

```
> str(cu.summary)
‘data.frame’:   117 obs. of  5 variables:
 $ Price      : num  11950  6851  6995  8895  7402 ...
 $ Country    : Factor w/ 10 levels "Brazil","England",..: 5 5 10 10 10 ...
 $ Reliability: Ord.factor w/ 5 levels "Much worse"<"worse"<..: 5 NA 1 4 2 ...
 $ Mileage    : num  NA NA NA 33 33 37 NA NA 32 NA ...
 $ Type       : Factor w/ 6 levels "Compact","Large",..: 4 4 4 4 4 4 4 4 ...
```

2.10.6 Komponentenweise Anwendung von Funktionen: `lapply`, `sapply`

Analog zur Anwendung einer Funktion auf die Elemente einer Liste durch `lapply` oder `sapply` (siehe Seite 53) lässt sich dies auch (*nur*) für die Komponenten eines Data Frames mit `lapply` oder `sapply` realisieren, da sie als Listenelemente auffassbar sind:

```
> lapply(cu.summary, class)
$Price:
[1] "numeric"

$Country:
[1] "factor"

$Reliability:
[1] "ordered" "factor"

$Mileage:
[1] "numeric"

$Type:
[1] "factor"
```

Hier wendet `lapply` auf die Komponenten (d. h. die Listenelemente) des Data Frames `cu.summary` die Funktion `class` an, was als Resultat eine Liste mit den Klassen der Komponenten (= Listenelemente) liefert.

(Memo: Zu Klasse und Modus von Faktoren siehe bei Bedarf nochmal am Anfang von Abschnitt 2.7.)

```
> sapply(cu.summary, mode)
     Price    Country Reliability    Mileage
 "numeric" "numeric"   "numeric" "numeric"
      Type
 "numeric"
```

`sapply` macht das Analoge zu `lapply` (hier mit `mode`), vereinfacht aber nach Möglichkeit die Struktur des Resultats; hier zu einem Vektor.

Natürlich ist auch die komponentenweise Anwendung von `summary` auf einen Data Frame möglich, wie z. B. durch `lapply(cu.summary, summary)`, was eine Liste mit den “summary statistics” einer jeden Komponente des Data Frames liefert. Allerdings ist diese nicht so übersichtlich wie das Resultat der direkten Anwendung von `summary` auf einen Data Frame.

2.10.7 Anwendung von Funktionen auf Faktor(en)gruppierte Zeilen: `by`

Die Funktion `by` bewerkstelligt die zeilenweise Aufteilung eines Data Frames gemäß der Werte eines Faktors – oder mehrerer Faktoren – in (wiederum) Data Frames und die nachfolgende Anwendung jeweils einer (und derselben) Funktion auf diese Teil-Data Frames. (Vgl. auch §2.7.7 zur Arbeitsweise von `tapply`, welches die interne Grundlage für `by` ist.)

Im folgenden Beispiel teilt `by` den Data Frame `cu.summary` zeilenweise gemäß der verschiedenen Werte des Faktors `cu.summary$Type` zunächst in Teil-Data Frames auf und wendet dann jeweils die Funktion `summary` auf diese an. Das Resultat wird hier nur teilweise gezeigt:

```
> by(cu.summary, cu.summary$Type, summary)
cu.summary$Type: Compact
     Price            Country       Reliability    Mileage          Type
 Min.   : 8620   USA      :7   Much worse :2    Min.   :21.00   Compact:22
 1st Qu.:10660   Germany  :4   worse      :5    1st Qu.:23.00   Large  : 0
 Median :11898   Japan/USA:4   average    :3    Median :24.00   Medium : 0
 Mean   :15202   Japan    :3   better     :4    Mean   :24.13   Small  : 0
 3rd Qu.:18307   Sweden   :3   Much better:5    3rd Qu.:25.50   Sporty : 0
 Max.   :31600   France   :1   NA's       :3    Max.   :27.00   Van    : 0
                 (Other)  :0                    NA's   : 7.00
------------------------------------------------------------
cu.summary$Type: Large
     Price          Country       Reliability    Mileage          Type
 Min.   :14525   USA    :7   Much worse :2    Min.   :18.00   Compact:0
 1st Qu.:16701   Brazil :0   worse      :0    1st Qu.:19.00   Large  :7
 Median :20225   England:0   average    :5    Median :20.00   Medium :0
 Mean   :21500   France :0   better     :0    Mean   :20.33   Small  :0
 3rd Qu.:27180   Germany:0   Much better:0    3rd Qu.:21.50   Sporty :0
 Max.   :27986   Japan  :0                    Max.   :23.00   Van    :0
                 (Other):0                    NA's   : 4.00
------------------------------------------------------------
....
```

Bemerkungen: Die bloße Aufteilung eines Data Frames in Teil-Data Frames gemäß der Werte eines Faktors oder mehrerer Faktoren (ohne die direkte Anwendung einer sonstigen Operation) ist mithilfe der Funktion `split` möglich (vgl. auch §2.7.7). In obiger Ausgabe ist gut zu erkennen, dass unbenutzte Faktorlevels nicht automatisch aus Teil-Data Frames verschwinden (wie im Hinweis auf S. 56 beschrieben) und dass hier evtl. eine Gelegenheit für den geeigneten (!) Einsatz von `droplevels` wäre. Eine weitere Funktion zur Berechnung von "summary statistics" für Teilmengen – typischwerweise – eines Data Frames ist `aggregate`. (Für beide Funktionen verweisen wir auf ihre Hilfeseiten.)

2.10.8 Bemerkung zur „Arithmetik" mit Data Frames

Mit Data Frames, deren Komponenten alle numerisch oder logisch sind, kann „als Ganzes" Arithmetik ähnlich (aber nicht identisch!) wie mit Matrizen betrieben werden, da sie bei der Auswertung von arithmetischen Ausdrücken intern temporär in Matrizen konvertiert werden. Ein Unterschied liegt in der Behandlung nicht zueinander passender Dimensionen von sowie für Data Frames und Matrizen, die nur ein Element enthalten, sprich eine Zeile und eine Spalte haben. Vergleichen Sie hierzu die Resultate der Operationen in den folgenden drei Beispielpaaren, deren Fazit lauten muss: Vorsicht, wenn Data Frames wie Matrizen behandelt werden sollen!

```
> matrix(1:6, ncol = 2) + 1:6
     [,1] [,2]
[1,]    2    8
[2,]    4   10
[3,]    6   12
```

```
> data.frame(1:3, 4:6) + 1:6
  X1.3 X4.6
1    2    8
2    4   10
3    6   12
```

```
> matrix(1:6, ncol = 2) + 1:9
Fehler: Dimensionen [Produkt 6] passen
nicht zur Laenge des Objektes [9]
Zusaetzlich: Warnmeldung:
In matrix(1:6, ncol = 2) + 1:9 : Laenge
des laengeren Objektes ist kein Vielfaches
der Laenge des kuerzeren Objektes
```

```
> data.frame(1:3, 4:6) + 1:9
  X1.3 X4.6
1    2    8
2    4   10
3    6   12
```

```
> data.frame(1) + 1:3
  X1
1  2
```

```
> matrix(1) + 1:3
[1] 2 3 4
```

2.10.9 „Organisatorisches“ zu Data Frames und dem Objektesuchpfad: `attach`, `detach` und `search`, `with`, `within` und `transform`

Die Komponenten eines Data Frames sind, wie auf den vorherigen Seiten vorgestellt, mittels des Komponentenoperators `$` über ihre Namen indizierbar. Sie sind Bestandteile des Data Frames und keine eigenständigen Objekte. D. h., jedes Mal, wenn z. B. auf die Komponente `Price` in `cu.summary` zugegriffen werden soll, ist stets auch der Name des Data Frames `cu.summary` anzugeben, damit **R** weiß, wo es die Komponente (sprich Variable) `Price` herholen soll.

Gelegentlich jedoch ist der Zugriff auf eine Komponente wiederholt oder auf mehrere Komponenten gleichzeitig notwendig (oder beides). Dann wird die Verwendung von *dataframe*`$`*komponentenname* schnell sehr aufwändig und unübersichtlich. Um hier Erleichterung zu schaffen, kann man den Data Frame, auf dessen Komponenten zugegriffen werden soll, vorher „öffnen“, sodass die Angabe seines Namens und die Verwendung des Operators `$` nicht mehr nötig ist, sondern die Komponenten des Data Frames gewissermaßen zu eigenständigen Objekten werden.

- Das „Öffnen“ eines Data Frames bewerkstelligt die Funktion `attach` und das „Schließen“ die Funktion `detach`. Durch `attach(DF)` wird der Data Frame `DF` (falls er existiert) in der zweiten Position des Objektesuchpfads von **R** hinzugefügt (Engl.: “attached”). An der ersten Stelle dieses Suchpfades steht der aktuelle “workspace” (namens `.GlobalEnv`) und beim Aufruf eines Objektes über seinen Namen sucht **R** zuerst in diesem workspace. Wenn **R** ein solchermaßen benanntes Objekt dort nicht findet, wird im nächsten Eintrag des Suchpfades, also hier im Data Frame `DF` weitergesucht. Führt dies zum Erfolg, wird das Objekt zur Verfügung gestellt; anderenfalls wird eine entsprechende Fehlermeldung ausgegeben. Soll der Data Frame `DF` aus dem Suchpfad wieder entfernt werden, so geschieht dies durch `detach(DF)`.

  Wie der aktuelle Suchpfad aussieht, zeigt die Funktion `search`: Es werden alle geladenen Pakete und anderen “attach”-ten Datenbanken oder Objekte aufgezählt.

  Achtung, die obige Methodik birgt Gefahren: Angenommen, es soll auf eine Komponente des Data Frames `DF` zugegriffen werden, deren Name genauso lautet, wie der eines Objektes im workspace. Dann wird **R** bereits im workspace fündig, beendet (!) seine Suche und stellt das gefundene Objekt zur Verfügung (und zwar *ohne* dem/der BenutzerIn mitzuteilen, dass es noch ein weiteres Objekt des gleichen Namens gibt). Fazit: Es obliegt dem/der BenutzerIn zu kontrollieren, dass das „richtige“ Objekt verwendet wird. (Dies ist ein weiteres Beispiel für die Notwendigkeit, den eigenen workspace öfter einmal aufzuräumen. Siehe hierzu Abschnitt 1.5, Seite 6.)

Des weiteren sind nach der Ausführung von `attach` Änderungen an Variablen des attach-ten Data Frames in der Regel *nicht* permanent, sondern „verflüchtigen“ sich durch ein `detach`!

Am Beispiel eines weiteren, bereits in “base **R**” zur Verfügung stehenden Data Frames namens `airquality` (mit Messwerten zur Luftqualität in New York von Mai bis September 1973) soll die oben beschriebene Funktionsweise von `attach` und Co. dargestellt werden:

```
> head(airquality, 5)                  # Fuer Infos siehe ?airquality
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
```

`> Ozone[41:42]` `Fehler: Objekt 'Ozone' nicht gefunden`	Offenbar gibt es kein Objekt namens `Ozone` im **R**-Suchpfad.
`> attach(airquality)` `> Ozone[41:42]` `[1] 39 NA` `> summary(Solar.R)` `Min. 1st Qu. Median Mean 3rd Qu. Max. NA's` ` 7.0 115.8 205.0 185.9 258.8 334.0 7.0`	Nachdem der Data Frame `airquality` dem **R**-Suchpfad hinzugefügt worden ist, sind `Ozone` und `Solar.R` sowie alle anderen Komponenten von `airquality` als eigenständige Objekte unter ihrem Komponentennamen verfügbar, denn ...
`> search()` `[1] ".GlobalEnv" "airquality"` `[3] "package:methods" "package:stats"` `[5] "package:graphics" "package:grDevices"` `[7] "package:utils" "package:datasets"` `[9] "Autoloads" "package:base"`	... wie der aktuelle Suchpfad zeigt, befindet sich der Data Frame `airquality` im Augenblick darin an zweiter Position, aber ...
`> detach(airquality)` `> summary(Solar.R)` `Fehler in summary(Solar.R): Objekt 'Solar.R' nicht gefunden`	... nach dem Entfernen des Data Frames aus dem Suchpfad eben nicht mehr (wie ein erneutes `search` explizit bestätigen würde).

- Weitere, sehr nützliche Funktionen zur Arbeit mit und Modifikation von Data Frames sind auch `with`, `within` und `transform`, für deren detaillierte Funktionsweise wir nachdrücklich auf ihre Hilfeseiten verweisen und hier nur kurze Beschreibungen sowie jeweils ein kleines Anwendungsbeispiel liefern:

  ▷ `with(data, expr)` wertet den Ausdruck `expr` in der „lokalen“ Umgebung `data` aus, indem `data` quasi vorübergehend in den Suchpfad eingehängt wird. Dabei ist `data` typischerweise ein Data Frame, kann aber auch eine Liste sein. Das Resultat des `with`-Aufrufs ist der Wert von `expr`, aber jegliche Zuweisung innerhalb von `expr` ist nur lokal gültig, d. h., findet nicht im benutzereigenen workspace statt und ist daher außerhalb des `with`-Aufrufs vergessen. `with` ist m. E. gegenüber der Kombination aus `attach` und `detach` i. d. R. klar zu bevorzugen! Beispiel:

```
> with(airquality, {
+  TempC <- (Temp - 32) * 5/9
+  summary(TempC)
+  summary(Ozone)
+  })

  Min. 1st Qu. Median  Mean
  1.00   18.00  31.50 42.13
  3rd Qu.   Max.    NA's
    63.25 168.00   37.00
```

Die in `{....}` stehenden Anweisungen sind der Ausdruck, der in der durch `airquality` lokal definierten Umgebung auszuwerten ist und worin der `Temp` und `Ozone` bekannt sind. Der Wert dieser `with`-Anweisung ist der Wert des letzten Ausdrucks in `{....}`, also `summary(Ozone)`, wohingegen `TempC` nach Abarbeitung der `with`-Anweisung genauso wieder verschwunden ist wie das – gar nicht ausgegebene – Ergebnis von `summary(TempC)`.

▷ `within(data, expr)` arbeitet ähnlich zu `with`, versucht allerdings, die in `expr` ausgeführten Zuweisungen und anderen Modifikationen in der Umgebung `data` permanent zu machen. Der Wert des `within`-Aufrufs ist der Wert des Objekts in `data` mit den durch `expr` resultierenden Modifikationen. Beispiel:

```
> air2 <- within(airquality, {
+  TempC <- (Temp - 32) * 5/9
+  Month <- factor(month.abb[Month])
+  rm(Day)
+  })

> head(air2, 2)
  Ozone Solar.R Wind Temp Month  TempC
1    41     190  7.4   67   May 19.444
2    36     118  8.0   72   May 22.222
```

Die Anweisungen in `{....}` werden in der durch `airquality` lokal definierten Umgebung ausgewertet, worin `Temp`, `Month` und `Day` bekannt sind. Das Ergebnis dieses `within`s ist der Wert von `airquality` nach Ausführung der Modifikationen, also mit neu angehängter Komponente `TempC`, neuem Wert für die bereits existierende Komponente `Month` und ohne die Komponente `Day`. Beachte: `airquality` selbst bleibt unverändert.

▷ `transform('_data', ...)` arbeitet ähnlich zu `within`, allerdings dürfen in `...` nur Ausdrücke der Art `name = expr` auftreten. Jede `expr` wird in der durch den Data Frame `_data` definierten Umgebung ausgewertet und ihr Wert nach Möglichkeit der jeweils zu `name` passenden Komponente des Data Frames `_data` zugewiesen. Wenn keine passende Komponente in `_data` existiert, wird der erhalten Wert an `_data` als weitere Komponente angehängt. Der Wert des `transform`-Aufrufs ist der modifizierte Wert von `_data`. Beispiel:

```
> air3 <- transform(airquality,
+  NegOzone = -Ozone,
+  Temp = (Temp - 32) * 5/9
+  )

> head(air3, 2)
  Ozone Solar.R Wind   Temp Month Day
1    41     190  7.4 19.444     5   1
2    36     118  8.0 22.222     5   2
  NegOzone
1      -41
2      -36
```

Die Ausdrücke `name = expr` werden in der durch `airquality` lokal definierten Umgebung ausgewertet, worin `Ozone` und `Temp` bekannt sind. Das Ergebnis dieses `transform`s ist der Wert von `airquality` nach Ausführung der Modifikationen, also mit neu angehängter Komponente `NegOzone` und neuem Wert für die bereits existierende Komponente `Temp`. Beachte: `airquality` selbst bleibt unverändert.

2.10.10 Nützliche Transformationshilfen: `stack`, `reshape`, `merge` sowie die Pakete `reshape`, (`d`)`plyr` und `tidyr`

Die Arbeit mit einem komplexeren Datensatz, der in der Regel aus Gründen der Datenerfassung und -verwaltung zunächst in Form einer Tabelle arrangiert wurde wie sie nicht für die direkte Verarbeitung mit **R** geeignet ist, bedingt oft das – wiederholte – Umformen dieser Datenstruktur, die in **R** typischerweise als Data Frames gespeichert wird. Zu diesem Zweck stehen in "base **R**" und in dem einen oder anderen **R**-Package mehrere, unterschiedlich leistungsfähige und komplizierte Funktionen zur Verfügung, deren Beschreibung hier zu weit führte. Stattdessen zählen wir nur einige auf und verweisen auf ihre Hilfeseiten bzw. auf sonstige Beschreibungen:

- `stack` stapelt Spalten oder vektorielle Komponenten eines Objektes zu einem neuen Vektor übereinander und generiert einen Faktor, der die Herkunftsspalte oder -komponente eines jeden Elementes des neuen Vektors angibt. `unstack` kehrt die Operation um und „entstapelt" einen solchen Vektor entsprechend (siehe die Hilfeseite).

- `reshape` ermöglicht die Transformation von (auch komplexeren) Data Frames aus dem sogenannten "wide"-Format (bei dem – an derselben Untersuchungseinheit erhobene – Messwiederholungen in separaten Spalten ein und derselben Zeile gespeichert sind) ins "long"-Format (bei dem jene Messwiederholungen in einer Spalte, aber auf verschiedene Zeilen verteilt sind) und umgekehrt. Ist z. B. für die Umformung von Data Frames mit Longitudinaldaten geeignet. (Umfangreiche Details sind auf der Hilfeseite zu finden.)

- `merge` erlaubt das Zusammensetzen von Data Frames mithilfe von Vereinigungsoperationen wie sie für Datenbanken verwendet werden (siehe die Hilfeseite).

- Das Paket `reshape` (beachte, dass hiermit nicht die Funktion `reshape` gemeint ist!) stellt zwei Funktionen namens `melt` und `cast` zur Verfügung, durch die die Funktionalität von `tapply` (vgl. §2.7.7), `by`, `aggregate` (für beide vgl. §2.10.7), `reshape` und anderen in ein vereinheitlichtes Konzept zusammengefasst worden ist/sei. Siehe hierzu die ausführliche und beispielbewehrte Darstellung in [101, Wickham (2007)].

- Im Paket `plyr` (vom selben Autor wie das Paket `reshape`) wird das Konzept „Teilen-Anwenden-Zusammenführen" für verschiedene Datenstrukturen in äußerst kompakter und kohärenter Weise umgesetzt, was insbesondere auch für Data Frames extrem hilfreich sein kann. In [103, Wickham (2011)] wird dies ausführlich beschrieben und anhand von zwei umfangreichen Fallstudien vorgeführt.

- `dplyr` (siehe [105, Wickham & Romain (2016)]) enthält gewissermaßen das "next-level"-Konzept von `plyr` konzentriert auf Data Frames und deren Manipulationen mittels einfacher Syntax sowie zugleich extrem effizienter Implementation. Hierzu existiert auch ein hervorragendes "cheat sheet" (siehe hier oder direkt `https://raw.githubusercontent.com/rstudio/cheatsheets/main/data-transformation.pdf`).

- Ein hinsichtlich Design-Philosophie, Programmiergrammatik und Datenstrukturen besonders konsistentes „Universum" für "Data Science" ist das "tidyverse". Es ist eine Sammlung an extrem leistungsfähigen **R**-Paketen, über die es unter `https://www.tidyverse.org` mehr Infomationen einschließlich eines exzellenten Online-Buches (`https://r4ds.had.co.nz`) zu finden gibt. Unter anderem ist `dplyr` ein Teil des tidyverse und auch ein Paket namens `tidyr`, das ebenfalls leistungsfähige Funktionen für Datentransformationen u. a. anbietet. Sein zugehöriges "cheat sheet" ist ebenfalls hier oder direkt unter `https://raw.githubusercontent.com/rstudio/cheatsheets/main/tidyr.pdf` verfügbar.

2.11 Abfrage und Konversion der Objektklasse sowie Abfrage von und Tipps zu „pathologischen“ Elementen oder Objekten

2.11.1 `class`, `is` und `as`

Es gibt die Möglichkeit, Informationen über die Klasse eines Datenobjekts abzufragen. Dies wird durch die Funktion `class` erreicht (wie in §2.10.6 schon gesehen). Sie liefert die entsprechende Information für ihr Argument. Die Überprüfung, ob ein Objekt einer speziellen Klasse angehört, geht mit der Funktion `is(object, class)`. Ihr Argument `object` bekommt das zu prüfende Objekt und ihr Argument `class` den Namen (als Zeichenkette) der Klasse. Das Resultat ist `TRUE`, wenn das Objekt der genannten Klasse angehört, `FALSE` anderenfalls. Die Funktion `as(object, class)` konvertiert das Objekt `object` in eins der Klasse `class`.

Es folgt eine (unvollständige) Auswahl an Möglichkeiten für das Argument `class`, wie sie in `is` und `as` Anwendung finden können:

`"array"`	Arrays	`"list"`	Listen
`"character"`	Zeichenkettenobjekte	`"logical"`	Logische Objekte
`"complex"`	Komplexwertige Objekte	`"matrix"`	Matrixobjekte
`"function"`	Funktionen	`"name"`	Benannte Objekte
`"integer"`	Ganzzahlige Objekte	`"numeric"`	Numerische Objekte
		`"vector"`	Vektoren

Ausnahmen sind die Klassen `data.frame`, `factor` und `ordered`: Für eine Konversion dorthinein gibt es die Funktionen `as.data.frame`, `as.factor` und `as.ordered`.
Außerdem existieren „Kurzformen“ wie `as.array`, `as.character` usw., auf denen `as` basiert. Sie können i. d. R. genauso verwendet werden (wie in untem folgenden Beispiel).

Ein paar Anwendungsbeispiele für obige Konzepte und für Falltüren in obigen Konzepten:

```
> a <- c("1", "2", "3", "4");     class(a)
[1] "character"

> a + 10
Error in a + 10 : non-numeric argument to binary operator
> is(a, "numeric")
[1] FALSE

> a <- as(a, "numeric");     class(a);     a
[1] "numeric"
[1] 1 2 3 4

> as.numeric(factor(1:5)) + 1
[1] 2 3 4 5 6
> as.numeric(factor(11:15)) + 1   # Memo: Faktoren sind vom Modus numeric!
[1] 2 3 4 5 6

> class(cu.summary)
[1] "data.frame"
> is(cu.summary, "data.frame");     is(cu.summary, "list")
[1] TRUE
[1] FALSE
```

2.11.2 Fehlende Werte (NA, NaN), Unendlich (Inf) und das leere Objekt (NULL)

Die symbolischen Konstanten NA, NaN, Inf und NULL sind sogenannte reservierte Worte in **R** und nehmen Sonderrollen ein:

- NA (Abkürzung für "not available") als Element eines Objektes repräsentiert einen „fehlenden" Wert an der Stelle dieses Elements.
- NaN (steht für "not a number") und repräsentiert Elemente die „numerisch nicht definiert" sind, weil sie das Resultat von Operationen wie $0/0$ oder $\infty - \infty$ sind.
- Inf implementiert ∞ (= Unendlich) in **R**.
- NULL ist sozusagen das leere Objekt (Länge = 0).

Zu jeder der obigen Konstanten liefert ihre Hilfeseite zahlreiche weitere technische Details (siehe z. B. ?NA).

Der Test, ob ein Objekt NA oder NaN enthält, wird mit der Funktion is.na geklärt. Soll konkret nur auf NaN geprüft werden, ist is.nan zu verwenden. Wo sich in einem Vektor die Einträge Inf oder -Inf befinden, zeigt is.infinite; mit is.finite wird Endlichkeit geprüft. Ob ein Objekt das NULL-Objekt ist, klärt die Funktion is.null.

Beachte: In einem character-Vektor ist der "missing value" NA verschieden von der Zeichenkette "NA", die aus den zwei Buchstaben „N" und „A" besteht.

Hier ein paar Anwendungsbeispiele für obige Konzepte und für Falltüren in obigen Konzepten:

```
> x <- c(2, -9, NA, 0);     class(x);     is.na(x)
[1] "numeric"
[1] FALSE FALSE  TRUE FALSE

> (y <- x/c(2,0,2,0))
[1]    1 -Inf   NA  NaN

> is.na(y);     is.nan(y)
[1] FALSE FALSE  TRUE  TRUE
[1] FALSE FALSE FALSE  TRUE

> is.finite(y);     is.infinite(y)
[1]  TRUE FALSE FALSE FALSE
[1] FALSE  TRUE FALSE FALSE

> is.na(c("A", NA, "NA"))
[1] FALSE  TRUE FALSE

> (xx <- as(x, "character"));     is.na(xx)
[1] "2"  "-9" NA   "0"
[1] FALSE FALSE  TRUE FALSE
```

Hinweise:

- Nützlich ist die Kombination der Funktionen is.na und which (siehe hierfür nochmal §2.4.2), um z. B. herauszufinden, an welchen Positionen eines Vektores oder einer Matrix ein Eintrag NA steht. Für die Anwendung auf Matrizen ist das which-Argument arr.ind sinnvollerweise auf TRUE zu setzen (siehe auch die Hilfeseite).

- Wenn in einem `numeric`-Vektor `NA`-Einträge „aufgefüllt“ werden sollen, indem die einzutragenden Werte aus den benachbarten Nicht-`NA`-Einträgen abgeleitet werden sollen, z. B. indem zwischen linkem und rechtem Nachbarn interpoliert wird, kann die Funktion `na.approx` des Paketes `zoo` von großem Nutzen sein. Bei nicht notwendigerweise numerischen Vektoren kann mit der Funktion `na.locf` jedes `NA` durch den letzten vorhergehenden Nicht-`NA`-Eintrag ersetzt werden. Diese – statistisch oft *nicht* sinnvolle – Vorgehensweise wird auch “last observation carried forward” (daher locf) genannt.
- Sollen in einem Vektor, einer Matrix oder einem Data Frame (oder sogar jeweils mehreren davon) alle Elemente bzw. Zeilen, auch Fälle genannt, *identifiziert* werden, die vollständig sind, also keine fehlenden Werte (sprich `NA`s oder `NaN`s) enthalten, so kann `complete.cases` die Funktion der Wahl sein.
- Sollen aus einem Vektor, einer Matrix oder einem Data Frame alle Elemente bzw. Zeilen, auch Fälle genannt, *entfernt* werden, die unvollständig sind, also (mindestens) einen fehlenden Wert (sprich `NA` oder `NaN`) enthalten, so liefert `na.omit` das um die unvollständigen Fälle „bereinigte“ Objekt zurück.
- Sollen Elemente eines Data Frames (oder auch einer `character`-Matrix), die den leeren `character`-String `""` enthalten, durch `NA` ersetzt werden, kann dies erzielt werden durch

```
> is.na(Objektname) <- Objektname == ""
```

3 Import und Export von Daten bzw. ihre Ausgabe am Bildschirm

Üblicherweise wird man große Datensätze, die es darzustellen und zu analysieren gilt, aus externen Dateien in **R** einlesen, was Datenimport genannt wird. Umgekehrt ist es gelegentlich notwendig, **R**-Datenobjekte in eine Datei auszulagern oder sogar in einen speziellen Dateityp zu exportieren, um sie für andere Programme lesbar zu machen.

3.1 Datenimport aus einer Datei: `scan`, `read.table` und Co.

Zum Zweck des Datenimports aus einer Datei stehen verschieden leistungsfähige Funktionen zur Verfügung. Wir konzentrieren uns hier auf den Datenimport aus ASCII-Textdateien mit Hilfe der "base **R**"-Funktionen `scan` sowie `read.table` und ihrer Verwandten. Hinsichtlich des Imports anderer Dateiformate verweisen wir auf das **R**-Manual "R Data Import/Export" (siehe z. B. die **R**-Hilfe und das Paket `foreign`) und auf das Paket `readr`, zu dem auch ein gutes "cheat sheet" existiert, zu finden via hier oder direkt bei `https://raw.githubusercontent.com/rstudio/cheatsheets/main/data-import.pdf`.

3.1.1 Die Funktion `scan`

Mit Hilfe der Funktion `scan` kann man aus einer ASCII-Datei Daten einlesen, die dort in einem Tabellenformat (= Zeilen-Spalten-Schema) vorliegen. Dabei stehen Optionen zur Verfügung, die steuern, wie die Datei zu lesen ist und wie die resultierende Datenstruktur in **R** auszusehen hat. Die Funktion `scan` ist recht schnell, allerdings muss der Benutzer/die Benutzerin für eine korrekte Felder- und Datenmodi-Zuordnung sorgen. `scan`'s unvollständige Syntax mit ihren Voreinstellungen lautet

```
scan(file = "", what = double(0), nmax = -1, sep = "", dec = ".", skip = 0,
     flush = FALSE)
```

Zur Bedeutung der aufgeführten Argumente:

- `file`: Erwartet in Hochkommata den Namen (nötigenfalls mit Pfadangabe) der Quelldatei, aus der die Daten gelesen werden sollen. (Es kann sogar eine vollständige URL sein!) Die Voreinstellung `""` liest die Daten von der Tastatur ein, wobei dieser Vorgang durch die Eingabe einer leeren Zeile oder von <Ctrl-D> (bzw. <Strg-D>) unter Unix oder <Ctrl-Z> (bzw. <Strg-Z>) unter Windows abgeschlossen wird. Eine Menge Tipparbeit bei der Angabe eines Dateinamens kann die Funktion `file.choose` ersparen, für die wir auf die Hilfeseite verweisen.
- `what` dient als Muster, gemäß dessen die Daten zeilenweise einzulesen sind, wobei die Voreinstellung `double(0)` bedeutet, dass versucht wird, *alle* Daten als `numeric` (in „doppelter Genauigkeit“) zu interpretieren. An Stelle von `double(0)` sind (u. a.) `logical(0)`, `numeric(0)`, `complex(0)` oder `character(0)` möglich, bzw. Beispiele vom jeweiligen Modus, also `TRUE`, `0`, `0i` oder `""`, was praktikabler, weil kürzer ist. `scan` versucht dann, alle Daten in dem jeweiligen Modus einzulesen. Das Resultat von `scan` ist ein Vektor des Modus' der eingelesenen Daten. Bekommt `what` eine Liste zugewiesen, wird angenommen, dass jede Zeile der Quelldatei ein Datensatz ist, wobei die Länge der `what`-Liste als die Zahl der Felder eines einzelnen Datensatzes interpretiert wird und jedes Feld von dem Modus ist, den die entsprechende Komponente in `what` hat. Das Resultat von `scan` ist dann eine Liste, deren Komponenten Vektoren sind, und zwar jeweils des Modus' der entsprechenden Komponenten in `what`.
- `nmax` ist die Maximalzahl einzulesender Werte oder, falls `what` eine Liste ist, die Maximalzahl der einzulesenden Datensätze. Bei Weglassung oder negativem Wert wird bis zum Ende der Quelldatei gelesen.

- sep gibt in Hochkommata das Trennzeichen der Felder der Quelldatei an: "\t" für den Tabulator oder "\n" für *newline* (= Zeilenumbruch). Bei der Voreinstellung "" trennt jede beliebige Menge an "white space" (= Leerraum) die Felder.
- dec bestimmt das in der Quelldatei für den Dezimalpunkt verwendete Zeichen und ist auf "." voreingestellt.
- skip gibt die Zahl der Zeilen an, die am Beginn der Quelldatei vor dem Einlesen übersprungen werden sollen (wie beispielsweise Titel- oder Kommentarzeilen). Voreinstellung ist skip = 0, sodass ab der ersten Zeile gelesen wird.
- flush = TRUE veranlasst, dass jedes Mal nach dem Lesen des letzten in der what-Liste geforderten Feldes in einem Datensatz der Rest der Zeile ignoriert wird. Man kann so hinter diesem letzten Feld in der Quelldatei z. B. Kommentare erlauben, die von scan nicht gelesen werden. Die Voreinstellung flush = FALSE sorgt dafür, dass jede Zeile vollständig gelesen wird.

(Für weitere Details siehe scans Hilfeseite.)

Die folgenden Beispiele sollen verdeutlichen, was mit scan möglich ist. Dazu verwenden wir zwei ASCII-Textdateien namens SMSA0 und SMSAID0, von denen hier jeweils ein Ausschnitt zu sehen ist (und mit denen wir uns noch öfter beschäftigen werden). Die Bedeutung der darin enthaltenen Daten ist auf Seite 71 etwas näher erläutert.

Zunächst ein Ausschnitt aus der Datei SMSA0. Darin sind die ersten vier Zeilen Titel- bzw. Leerzeilen und erst ab Zeile fünf stehen die eigentlichen Daten, deren Leerräume aus verschieden vielen Leerzeichen ("blanks") bestehen:

```
Datensatz "SMSA" (Standardized Metropolitan Settlement Areas) aus
Neter, Wasserman, Kuttner: "Applied Linear Models".

ID   2    3    4     5    6     7     8    9      10    11     12
  1  1384 9387  78.1 12.3 25627 69678 50.1 4083.9 72100 709234 1
  2  4069 7031  44.0 10.0 15389 39699 62.0 3353.6 52737 499813 4
....
141   654  231  28.8  3.9   140  1296 55.1   66.9  1148  15884 3
```

Es folgt ein Ausschnitt aus der Datei SMSAID0. Darin beinhalten die Daten selbst Leerräume (nämlich die "blanks" in den Städtenamen und vor den Staatenabkürzungen) und die Datenfelder sind durch jeweils genau einen (unsichtbaren) Tabulator \t getrennt:

```
Identifikationscodierung (Spalte ID) des Datensatzes "SMSA" aus
Neter, Wasserman, Kuttner: "Applied Linear Models".
1    NEW YORK, NY         48  NASHVILLE, TN     95   NEWPORT NEWS, VA
2    LOS ANGELES, CA      49  HONOLULU, HI      96   PEORIA, IL
....
47   OKLAHOMA CITY, OK    94  LAS VEGAS, NV     141  FAYETTEVILLE, NC
```

Wir gehen im Folgenden davon aus, dass sich die Dateien SMSA0 und SMSAID0 in dem Verzeichnis befinden, in dem **R** gestartet wurde, sodass sich eine Pfadangabe erübrigt.

Daten einlesen mit scan

```
> (smsa <- scan("SMSA0", skip = 4))
Read 1692 items
  [1]     1.0   1384.0   9387.0
  [3]    78.1     12.3  25627.0
 ....
[1690] 1148.0  15884.0      3.0
```

In der Datei SMSA0 werden wegen skip = 4 die ersten vier Zeilen ignoriert und der gesamte danach folgende Inhalt zeilenweise als numeric-Vektor (Voreinstellung) eingelesen, wobei jeglicher Leerraum (per Voreinstellung) als Trennzeichen fungiert und das Resultat (hier) als smsa gespeichert wird.

```
> scan("SMSA0", nmax = 11, skip = 4)
Read 11 items
 [1]     1.0   1384.0    9387.0  78.1
 [5]    12.3  25627.0   69678.0  50.1
 [9]  4083.9  72100.0  709234.0
```

Wegen des Arguments `nmax = 11` werden (wg. `skip = 4` ab der fünften Zeile) lediglich die ersten 11 Felder eingelesen. (Ohne Zuweisung werden die Daten nicht gespeichert, sondern nur am **R**-Prompt ausgegeben.)

```
> scan("SMSA0", what = list(ID = "",0,0,0,0,
+ 0,0,0,0,0,0,Code = ""), nmax = 2, skip = 4)
Read 2 records
$ID:                    .
[1] "1" "2"             .
                        [[7]]:
[[2]]:                  [1] 69678 39699
[1] 1384 4069
                        [[8]]:
[[3]]:                  [1] 50.1 62.0
[1] 9387 7031
                        [[9]]:
[[4]]:                  [1] 4083.9 3353.6
[1] 78.1 44.0
                        [[10]]:
[[5]]:                  [1] 72100 52737
[1] 12.3 10.0
                        [[11]]:
[[6]]:                  [1] 709234 499813
[1] 25627 15389
.                       $Code:
.                       [1] "1" "4"
```

Die Liste, die hier `what` zugewiesen wird, spezifiziert, dass jeder Datensatz aus 12 Feldern besteht und das erste Feld jeweils den Modus `character` hat, gefolgt von 10 `numeric`-Feldern und wiederum einem `character`-Feld. Die (wg. `skip = 4` ab der fünften Zeile) einzulesenden Felder der `nmax = 2` Datensätze (!) werden demnach so interpretiert, wie die Komponenten des `what`-Arguments beim zyklischen Durchlauf. Das Ergebnis ist eine Liste von Vektoren des Modus' der entsprechenden `what`-Komponente. Ihre Komponenten haben dabei die Namen, die ihnen im `what`-Argument (evtl.) gegeben wurden.

```
> scan("SMSAID0", what = list(Nr1 = 0,
+ City1 = "", Nr2 = 0, City2 = "", Nr3 = 0,
+ City3 = ""), nmax = 2, sep = "\t", skip = 2)
Read 2 records
$Nr1
[1] 1 2

$City1
[1] "NEW YORK, NY"    "LOS ANGELES, CA"

$Nr2
[1] 48 49

$City2
[1] "NASHVILLE, TN" "HONOLULU, HI"

$Nr3
[1] 95 96

$City3:
[1] "NEWPORT NEWS, VA" "PEORIA, IL"
```

Hier werden in der Datei `SMSAID0` (wg. `skip = 2` ab der dritten Zeile) sechs Felder pro Datensatz erwartet, die abwechselnd den Modus `numeric` und `character` haben sowie durch jeweils genau einen Tabulator als Trennzeichen separiert sind (`sep = "\t"`). Es sollen `nmax = 2` Datensätze eingelesen werden und die Komponentennamen der Ergebnisliste lauten `Nr1`, `City1`, `Nr2`, `City2`, `Nr3` und `City3`. (Auch hier werden keine Daten gespeichert, sondern nur ausgegeben.)

3.1.2 Die Beispieldaten "SMSA"

Die von uns verwendeten SMSA-Daten stammen aus [80, Neter, Wasserman und Kuttner (1990)]. Hier die dort zu findenden Hintergrundinformationen und Beschreibungen. Zitat:

This data set provides information for 141 large Standard Metropolitan Statistical Areas (**SMSA**s) in the United States. A standard metropolitan statistical area includes a city (or cities) of specified population size which constitutes the central city and the county (or counties) in which it is located, as well as contiguous counties when the economic and social relationships between the central and contiguous counties meet specified criteria of metropolitan character and integration. An SMSA may have up to three central cities and may cross state lines.

Each line of the data set has an identification number and provides information on 11 other variables for a single SMSA. The information generally pertains to the years 1976 and 1977. The 12 variables are:

Variable Number	*Variable Name*	*Description*
1	Identification number	1 – 141
2	Land area	In square miles
3	Total population	Estimated 1977 population (in thousands)
4	Percent of population in central cities	Percent of 1976 SMSA population in central city or cities
5	Percent of population 65 or older	Percent of 1976 SMSA population 65 years old or older
6	Number of active physicians	Number of professionally active nonfederal physicians as of December 31, 1977
7	Number of hospital beds	Total number of beds, cribs, and bassinettes during 1977
8	Percent high school	Percent of adult population (persons 25 years old or older) who completed 12 or more years of school, according to the 1970 Census of the Population
9	Civilian labor force	Total number of persons in civilian labor force (person 16 years old or older classified as employed or unemployed) in 1977 (in thousands)
10	Total personal income	Total current income received in 1976 by residents of the SMSA from all sources, before deduction of income and other personal taxes but after deduction of personal contributions to social security and other social insurance programs (in millions of dollars)
11	Total serious crimes	Total number of serious crimes in 1977, including murder, rape, robbery, aggravated assault, burglary, larceny-theft, and motor vehicle theft, as reported by law enforcement agencies
12	Geographic region	Geographic region classification is that used by the U.S. Bureau of the Census, where: 1 = NE, 2 = NC, 3 = S, 4 = W

Data obtained from: U.S. Bureau of the Census, *State and Metropolitan Area Data Book, 1979* (a Statistical Abstract Supplement).

Zitat Ende.

3.1.3 Die Funktion `read.table` und ihre Verwandten

Eine leistungsfähigere (aber etwas langsamere) Funktion, die eine automatische Datenmodi-Erkennung versucht und als Resultat einen Data Frame liefert, ist `read.table`. Sie liest Daten aus einer ASCII-Textdatei, falls diese darin im Tabellenformat angelegt sind. Die unvollständige Syntax von `read.table` mit ihren Voreinstellungen lautet:

```
read.table(file, header = FALSE, sep = "", dec = ".", row.names, col.names,
           as.is = !stringsAsFactors, nrows = -1, skip = 0, stringsAsFactors)
```

Zur Bedeutung der aufgeführten Argumente:

- `file`: Erwartet in Hochkommata den Namen (nötigenfalls mit Pfadangabe) der Quelldatei, aus der die Daten gelesen werden sollen. (Es kann sogar eine vollständige URL sein!) Jede Zeile der Quelldatei ergibt eine Zeile im resultierenden Data Frame.

- Ist `header = TRUE`, so wird versucht, die erste Zeile der Quelldatei für die Variablennamen des resultierenden Data Frames zu verwenden. Voreinstellung ist `FALSE`, aber wenn sich in der Kopfzeile (= erste Zeile) der Datei ein Feld weniger befindet als die restliche Datei Spalten hat (und somit die Einträge in der ersten Dateizeile als Variablennamen interpretiert werden können), wird `header` automatisch auf `TRUE` gesetzt.

- `sep` erhält das Zeichen, durch das die Datenfelder in der Quelldatei getrennt sind (siehe bei `scan`, Seite 69; Voreinstellung: Jede beliebige Menge Leerraum).

- `dec` bestimmt das in der Quelldatei für den Dezimalpunkt verwendete Zeichen und ist auf `"."` voreingestellt.

- `row.names` ist eine optionale Angabe für Zeilennamen. Fehlt sie und hat die Kopfzeile ein Feld weniger als die Spaltenzahl der restlichen Datei, so wird die erste Tabellenspalte für die Zeilennamen verwendet (sofern sie keine Duplikate in ihren Elementen enthält, denn ansonsten gibt es eine Fehlermeldung). Diese Spalte wird dann als Datenspalte entfernt. Anderenfalls, wenn `row.names` fehlt, werden Zeilennummern als Zeilennamen vergeben.
  Durch die Angabe eines `character`-Vektors (dessen Länge gleich der eingelesenen Zeilenzahl ist) für `row.names` können die Zeilen explizit benannt werden. Die Angabe einer einzelnen Zahl wird als Spaltennummer bzw. die einer Zeichenkette als Spaltenname interpretiert und die betreffende Spalte der Tabelle wird als Quelle für die Zeilennamen festgelet. Zeilennamen, egal wo sie herkommen, müssen eindeutig sein!

- `col.names` ist eine optionale Angabe für Variablennamen. Fehlt sie, so wird versucht, die Information in der Kopfzeile der Quelldatei für die Variablennamen zu nutzen (so, wie bei `header` beschrieben). Wenn dies nicht funktioniert, werden die Variablen mit den Namen `V1`, `V2` usw. (bis zur Nummer des letzten Feldes) versehen.
  Durch die Angabe eines `character`-Vektors (passender Länge) für `col.names` können die Variablen explizit benannt werden. Variablennamen, egal wo sie herkommen, müssen eindeutig sein!

- `stringsAsFactors` (üblicherweise mit der Voreinstellung `FALSE`) gibt an, ob `character`- in `factor`-Vektoren konvertiert werden sollen. (Hinweis: Die Voreinstellung war vor der Version 4.0.0 anders.)

- `as.is = !stringsAsFactors` bedeutet in der üblichen Voreinstellung, dass *alle* Spalten, die nicht in `logical`, `numeric` oder `complex` konvertierbar sind, zu `character`-Vektoren werden. Wird – aus irgendeinem vielleicht guten Grund – `stringsAsFactors = TRUE` gesetzt, kann hier mit einem `logical`-, `numeric`- oder `character`-Indexvektor explizit angegeben werden, welche Spalten doch `character` bleiben sollen, falls sie nicht in `logical`,

numeric oder complex konvertiert werden können. Beachte, dass sich dieser Indexvektor dann auf *alle* Spalten der Quelldatei bezieht, also auch auf die der Zeilennamen, falls vorhanden. (Hinweis: Die Voreinstellung war vor Version 4.0.0 anders.)

- nrows ist die Maximalzahl einzulesender Zeilen. Negative Werte führen zum Einlesen der gesamten Quelldatei. (Die Angabe eines Wertes, auch wenn er etwas größer ist als die tatsächliche Zeilenzahl der Quelldatei, kann den Speicherplatzbedarf des Lesevorgangs reduzieren.)

- skip ist die am Beginn der Quelldatei zu überspringende Zeilenzahl (siehe bei scan, Seite 69; Voreinstellung: skip = 0, sodass ab der ersten Zeile gelesen wird).

Für viele weitere Details siehe read.tables Hilfeseite.

Für Anwendungsbeispiele mögen die Dateien SMSA1, SMSA2, SMSA3 und SMSAID1 dienen, von denen unten jeweils ein Ausschnitt gezeigt ist. SMSA1 bis SMSA3 enthalten im Wesentlichen dieselben Daten; lediglich das Vorhandensein einer Überschriftszeile bzw. die Einträge in der letzten Spalte unterscheidet/n die Dateien. Ihre Spalten sind durch den (unsichtbaren) Tabulator getrennt. Die Datei SMSAID1 enthält bis auf die einleitenden Kommentarzeilen dieselben Informationen wie SMSAID0 von Seite 69; die „Spalten“ sind auch hier durch einzelne (unsichtbare) Tabulatoren getrennt.

Die Datei SMSA1:

```
1       1384    9387    78.1    12.3    ....    72100   709234  1
2       4069    7031    44.0    10.0    ....    52737   499813  4
3       3719    7017    43.9    9.4     ....    54542   393162  2
....
141     654     231     28.8    3.9     ....    1148    15884   3
```

Die Datei SMSA2:

```
        LArea   TPop    CPop    OPop    ....    PIncome SCrimes GReg
1       1384    9387    78.1    12.3    ....    72100   709234  1
2       4069    7031    44.0    10.0    ....    52737   499813  4
3       3719    7017    43.9    9.4     ....    54542   393162  2
....
```

Die Datei SMSA3:

```
ID      LArea   TPop    CPop    OPop    ....    PIncome SCrimes GReg
1       1384    9387    78.1    12.3    ....    72100   709234  NE
2       4069    7031    44.0    10.0    ....    52737   499813  W
3       3719    7017    43.9    9.4     ....    54542   393162  NC
....
```

Die Datei SMSAID1:

```
1    NEW YORK, NY         48  NASHVILLE, TN     95   NEWPORT NEWS, VA
2    LOS ANGELES, CA      49  HONOLULU, HI      96   PEORIA, IL
....
47   OKLAHOMA CITY, OK    94  LAS VEGAS, NV     141  FAYETTEVILLE, NC
```

Es folgen verschiedene Möglichkeiten, die Dateien SMSA1, SMSA2, SMSA3 sowie SMSAID1 einzulesen. Dabei gehen wir davon aus, dass diese in dem Verzeichnis liegen, in dem **R** gestartet wurde, sodass sich eine Pfadangabe erübrigt.

ASCII-Dateien einlesen mit `read.table`

```
> read.table("SMSA1")
     V1    V2   V3    V4 .... V12
1     1  1384 9387  78.1 ....   1
2     2  4069 7031  44.0 ....   4
....
```

`SMSA1` enthält Daten in Form einer Tabelle ohne Kopfzeile. Das Resultat von `read.table` ist ein Data Frame mit Nummern als Zeilenbenennung und `V1` bis `V12` als Variablennamen. (Ohne Zuweisung werden die eingelesenen Daten nicht gespeichert, sondern nur am **R**-Prompt ausgegeben.)

```
> read.table("SMSA1", row.names = 1)
       V2   V3    V4 .... V12
1    1384 9387  78.1 ....   1
2    4069 7031  44.0 ....   4
....
```

Durch `row.names =` n wird die n-te Spalte der Quelldatei zur Zeilenbenennung ausgewählt und (offenbar *nach* dem Einlesen) als Datenspalte entfernt (beachte die Variablennamen).

```
> read.table("SMSA1", row.names = 1,
+ col.names = LETTERS[1:12])
        B    C     D .... L
1    1384 9387  78.1 .... 1
2    4069 7031  44.0 .... 4
....
```

Mittels `col.names` werden die Spalten *vor* dem Einlesen explizit benannt. Zeilennamen werden dann der `row.names`-Spalte der Datei entnommen.

```
> read.table("SMSA2")
    LArea TPop  CPop .... GReg
1    1384 9387  78.1 ....    1
2    4069 7031  44.0 ....    4
....
```

`SMSA2` hat eine Kopfzeile mit einem Eintrag weniger als die Zeilen der restlichen Tabelle, weswegen ihre Einträge zu Variablennamen werden. Automatisch wird die erste Spalte der Quelldatei zur Zeilenbenennung ausgewählt und als Datenspalte entfernt.

```
> (SMSA <- read.table("SMSA3",
+ header = TRUE, row.names = "ID"))
    LArea TPop  CPop .... GReg
1    1384 9387  78.1 ....   NE
2    4069 7031  44.0 ....    W
....

> sapply(SMSA, class)
    LArea      TPop      CPop      OPop
"integer" "integer" "numeric" "numeric"
    APhys     HBeds    HSGrad   CLForce
"integer" "integer" "numeric" "numeric"
  PIncome  SCrimes          GReg
"integer" "integer"  "character"
```

`SMSA3` enthält in allen Zeilen gleich viele Einträge, weswegen durch `header = TRUE` die Verwendung der Kopfzeileneinträge als Variablennamen „erzwungen“ werden muss. Wegen `row.names = "ID"` ist die Spalte „ID“ der Quelldatei zur Zeilenbenennung ausgewählt und als Datenspalte entfernt geworden. Außerdem enthält die letzte Dateispalte weder logische, numerische noch komplexwertige Einträge, sodass sie ein `character`-Vektor wird, wie die Anwendung von `class` zeigt (oder `str` auch zeigen würde).

Im folgenden Beispiel wird die ASCII-Datei `SMSAID1` eingelesen, deren „Datenspalten“ durch den Tabulator getrennt sind (`sep = "\t"`). Dabei werden die Variablen des resultierenden Data Frames mittels `col.names` explizit benannt:

```
> (SMSAID <- read.table("SMSAID1", sep = "\t", col.names = c("Nr1", "City1",
+ "Nr2", "City2", "Nr3", "City3")))
   Nr1              City1 Nr2             City2 Nr3               City3
1    1       NEW YORK, NY  48     NASHVILLE, TN  95     NEWPORT NEWS, VA
2    2    LOS ANGELES, CA  49      HONOLULU, HI  96           PEORIA, IL
```

```
3    3        CHICAGO, IL  50  JACKSONVILLE, FL  97       SHREVEPORT, LA
....
47  47  OKLAHOMA CITY, OK  94     LAS VEGAS, NV 141     FAYETTEVILLE, NC

> sapply(SMSAID, class)
      Nr1       City1         Nr2       City2         Nr3       City3
"integer" "character"   "integer" "character"   "integer" "character"
```

Hinweise: (Für Details siehe die jeweiligen Hilfeseiten.)

- Zu `read.table` gibt es die vier „spezialisierte“ Varianten `read.csv`, `read.csv2`, `read. delim` und `read.delim2`, die genau wie `read.table` funktionieren und lediglich andere Voreinstellungen haben:
  `read.csv` ist für das Einlesen von “comma separated value”-Dateien (CSV-Dateien) voreingestellt und `read.csv2` ebenso, wenn in Zahlen das Komma anstelle des Punkts als Dezimalzeichen verwendet werden und gleichzeitig das Semikolon als Feldtrenner fungiert.
  Völlig analog wirken `read.delim` und `read.delim2`, außer dass sie als Feldtrenner den Tabulator erwarten.

  Beachte, dass in allen vier Fällen `header = TRUE` gesetzt ist (sowie `fill = TRUE` gilt und das Kommentarzeichen deaktiviert ist; zur Bedeutung dieses Sachverhalts siehe die Hilfeseite).

- Für den Import sogenannter “flat tables”, die flache (i. S. v. zweidimenionale) Darstellungen faktisch höherdimensionaler (Häufigkeits-)Tabellen enthalten, existiert eine praktische Variante namens `read.ftable` (worauf wir am Ende von §**??** „Kontingenztafeln für $k \geq 2$ Faktoren: `xtabs` & `ftable`“ einmal kurz eingehen werden).

- `read.fwf` erlaubt das Einlesen von ASCII-Dateien, die im tabellarischen “**f**ixed **w**idth **f**ormat” arrangiert sind.

- Für den vergleichsweise schnellen und effizienten Import besonders großer Dateien ist das Paket `data.table` mit insbes. seiner Funktion `fread` sehr zu empfehlen.

- Beachten Sie auch das bereits am Anfang dieses Kapitels erwähnte Paket `readr` mit seinem “cheat sheet”.

3.2 Datenausgabe am Bildschirm und ihre Formatierung: `print`, `cat` & Helferinnen

Für die Datenausgabe am Bildschirm stehen z. B. die Funktionen `print` und `cat` zur Verfügung.

- `print`: Bisher bekannt ist, dass die Eingabe eines Objektnamens am **R**-Prompt den „Inhalt“ bzw. Wert des Objektes als Antwort liefert. Intern wird dazu automatisch die Funktion `print` – genauer eine geeignete „Methode“ von `print` – aufgerufen, die „weiß“, wie der Objektinhalt bzw. -wert für eine Bildschirmdarstellung zu formatieren ist, und zwar je nachdem, ob es ein Vektor oder eine Matrix welchen Modus' auch immer ist, eine Liste oder ein Data Frame oder – was wir noch kennenlernen werden – z. B. eine Funktion oder das Ergebnis einer Funktion zur Durchführung eines Hypothesentests.

  Der explizite Aufruf von `print` erlaubt i. d. R. die Modifikation gewisser Voreinstellungen der Ausgabe, wie z. B. die minimale Anzahl der gezeigten signifikanten Ziffern durch das `integer`-Argument `digits` (1 - 22) oder die *Nicht*-Verwendung von Anführungszeichen bei Zeichenketten mittels des `logical`-Arguments `quote = FALSE`.

  Notwendig ist die explizite Verwendung von `print`, wenn aus dem Rumpf einer (benutzereigenen) Funktion oder aus einer `for`-Schleife heraus z. B. zu „Protokollzwecken“ eine Ausgabe an den Bildschirm erfolgen soll. (Details hierzu folgen in Kapitel 6.)

- `cat` liefert eine viel rudimentärere und weniger formatierte Ausgabe der Werte der an sie übergebenen (auch mehreren) Objekte. Sie erlaubt dem/der Benutzer/in aber die eigenständigere Formatierung und außerdem die Ausgabe der Objektwerte in eine Datei, falls an `cat`'s `character`-Argument `file` ein Dateiname übergeben wird. Existiert die Datei schon, wird sie entweder *ohne Warnung* überschrieben oder die aktuelle Ausgabe an ihren bisherigen Inhalt angehängt, wozu jedoch `cat`'s `logical`-Argument `append` auf `TRUE` zu setzen ist.

  Per Voreinstellung fügt `cat` an jedes ausgegebene Element ein Leerzeichen an, was durch ihr `character`-Argument `sep` beliebig geändert werden kann.

  Beachte: Sehr nützliche Sonderzeichen ("escape character"), die `cat` „versteht", sind (z. B.) der Tabulator `\t` und der Zeilenumbruch `\n`. Den "backslash" \ erhält man durch `\\`.

Beachte: Die Funktionen `format`, `formatC` und `prettyNum` können sehr nützlich sein, wenn (hauptsächlich `numeric`-)Objekte für die Ausgabe „schön" formatiert werden sollen.
Zu allen oben aufgeführten Funktionen ist – wie immer – eine (auch wiederholte) Konsultation der Hilfeseiten zu empfehlen.

3.3 Datenexport in eine Datei: `sink`, `write` und `write.table`

Mit `sink` kann man die Bildschirmausgabe „ein zu eins" solange in eine anzugebende Datei „umleiten" – also auch die Ausgabe von `print` – bis ein erneuter `sink`-Aufruf (ohne Argument) die Umleitung wieder aufhebt.

Für den Datenexport in eine Datei stehen ebenfalls verschiedene Funktionen zur Verfügung: `cat` unter Verwendung ihres Argumentes `file`, wie bereits in Abschnitt 3.2 erwähnt, und `write` als „Umkehrungen" von `scan` sowie `write.table` als „Umkehrung" der Funktion `read.table`. (Für nähere Informationen zu all diesen Funktionen siehe deren Hilfeseiten.)

Nur ein paar kurze Beispiele. Angenommen, wir haben

```
> alter
 [1] 35 39 53 14 26 68 40 56 68 52 19 23 27 67 43
> gewicht
 [1] 82 78 57 43 65 66 55 58 91 72 82 83 56 51 61
> rauchend
 [1] "L" "G" "X" "S" "G" "G" "X" "L" "S" "X" "X" "L" "X" "X" "S"
> m <- cbind(alter, gewicht)     # also eine (15x2)-Matrix
```

Dann kann der Export der obigen Vektoren und der Matrix `m` in Dateien wie folgt geschehen:

Umleiten der R-Bildschirmausgabe in eine Datei mit `sink`

```
> sink("RauchTab");   table(rauchend);   sink
```

Erzeugt, falls sie noch nicht existiert, die Datei `RauchTab` (in dem Verzeichnis, in dem **R** gestartet wurde) und öffnet sie zum Schreiben. Die Ausgaben *aller* folgenden Befehle – hier nur `table(rauchend)` – werden „eins zu eins" in diese Datei geschrieben, und zwar bis zum nächsten `sink`-Befehl mit leerer Argumenteliste. Ab da geschieht wieder normale Bilschirmausgabe. (Voreinstellung von `sink` ist, dass der ursprüngliche Inhalt der Zieldatei überschrieben wird. Mit dem Argument `append = TRUE` erreicht man, dass die Ausgaben am Zieldateiende angehängt werden.) Der Inhalt der (vorher leeren) Datei `RauchTab` ist nun:

```
 G L S X
 3 3 3 6
```

Daten mit `write` in eine Datei ausgeben

`> write(alter,` `+ file = "Alter",` `+ ncolumns = 5)`	Schreibt den Vektor `alter` zeilenweise in die ASCII-Datei `Alter` (in dem Verzeichnis, worin **R** gestartet wurde), und zwar in der 5-spaltigen Form `35 39 53 14 26` `68 40 56 68 52` `19 23 27 67 43.`
`> write(rauchend,` `+ file = "Rauch")`	Der `character`-Vektor `rauchend` wird als eine Spalte in die ASCII-Datei `Rauch` geschrieben.
`> write(t(m),` `+ file = "Matrix",` `+ ncolumns = ncol(m))`	Die (15×2)-Matrix `m` wird (nur so) als (15×2)-Matrix in die Datei `Matrix` geschrieben. Beachte die Notwendigkeit des Transponierens von `m`, um *zeilen*weise in die Zieldatei geschrieben zu werden, denn intern werden Matrizen *spalten*weise gespeichert und demzufolge auch so ausgelesen!

Daten mit `write.table` in eine ASCII-Datei ausgeben

Aus

```
> cu.summary[1:3,]
                Price Country Reliability Mileage  Type
Acura Integra 4 11950   Japan Much better      NA Small
Dodge Colt 4     6851   Japan        <NA>      NA Small
Dodge Omni 4     6995     USA  Much worse      NA Small
```

wird durch

```
> write.table(cu.summary[1:3,], "cu.txt")
```

die ASCII-Datei `cu.txt` folgenden Inhalts im aktuellen Arbeitsverzeichnis erzeugt.

```
"Price" "Country" "Reliability" "Mileage" "Type"
"Acura Integra 4" 11950 "Japan" "Much better" NA "Small"
"Dodge Colt 4" 6851 "Japan" "NA" NA "Small"
"Dodge Omni 4" 6995 "USA" "Much worse" NA "Small"
```

`write.table` wandelt ihr erstes Argument zunächst in einen Data Frame um (falls es noch keiner ist) und schreibt diesen dann zeilenweise in die angegebene Zieldatei. Dabei werden die Einträge einer jeden Zeile per Voreinstellung durch ein Leerzeichen (`" "`) getrennt. Beachte die vom Variablentyp abhängige, unterschiedliche Verwendung von Anführungszeichen im Zusammenhang mit NAs, wie z. B. beim Faktor `Reliability` und der `numeric`-Variablen `Mileage`. Mit Hilfe des Arguments `sep` kann das Trennzeichen für die Einträge einer jeden Zeile beliebig festgelegt werden, wie z. B. bei `sep = "\t"` der Tabulator verwendet wird.

Bemerkung: Für sehr große Data Frames mit sehr vielen Variablen kann `write.table` recht langsam sein. Die Funktion `write.matrix` im **R**-Paket `MASS` ist für große `numeric`-Matrizen oder Data Frames, die als solche darstellbar sind, effizienter; siehe ihre Hilfeseite. Auch das Paket `data.table` (und hierin seine Funktion `fwrite`) sollte in Betracht gezogen werden.

3.4 Ausgabe in Dateien mit Formatierung in TEX, HTML, im "Open-Document-Format" (ODF) oder mit R Markdown

Auf den ersten Blick – und auch auf den zweiten – sind die von **R** gelieferten Ausgaben, seien es Datenstrukturen wie Data Frames oder Resultatelisten von Funktionen wie wir sie noch zuhauf kennenlernen werden, layouttechnisch wenig berauschend, was der diesbzgl. mangelnden

Leistungsfähigkeit der **R**-Console geschuldet ist.

Allerdings gibt es sehr leistungsfähige Werkzeuge, **R**-Strukturen in LaTeX- oder HTML-Code „übersetzen“ zu lassen, um diesen danach in entsprechende Dokumente einzubinden. Speziell für Objektdarstellungen in LaTeX steht z. B. im **R**-Paket `Hmisc` die Funktion `latex` zur Verfügung. Soll hingegen HTML-Code erzeugt werden, ist z. B. die Funktion `HTML` des Pakets `R2HTML` verwendbar. Geht es im Wesentlichen „nur“ um die Darstellung von tabellen-ähnlichen **R**-Objekten, kann z. B. `xtable` aus dem gleichnamigen Paket Ausgaben wahlweise in *beiden* Formaten liefern.

Für weitere Informationen zu obigem und zu dem mit diesem Thema verwandten, sehr interessanten Aspekt der (halb-)automatischen Reportgenerierung in TeX (mit Hilfe der Funktion `Sweave`, die bereits in “base **R**” vorhanden ist; siehe Hilfeseite) oder im ODF (mit Hilfe von `odfWeave` des genauso benannten **R**-Pakets) siehe auch den Task View “Reproducible Research” auf CRAN (`http://cran.r-project.org/` → Task Views → Reproducible Research). Diesbzgl. höchst empfehlenswert sind auch [41, Gandrud (2014)] und [107, Xie (2015)].

Ein weiteres, in diesem Zusammenhang höchst attraktives Konzept ist R Markdown. Wir werden es in Kombination mit der Nutzung von RStudio in den Übungen kennenlernen. Eine essenzielle und empfehlenswerte Quelle ist [108, Xie et al. (2019)] und seine ständig aktualisierte Online-Version unter `https://bookdown.org/yihui/rmarkdown` auf bookdown.org. Ein hilfreiches “cheat sheet” ist wieder hier oder direkt bei `https://raw.githubusercontent.com/rstudio/cheatsheets/main/rmarkdown.pdf`.

Alles in allem ist aber darauf hinzuweisen, dass das Einarbeiten in die oben erwähnten Werkzeuge eine gewisse Zeit in Anspruch nehmen kann, die sich mittelfristig aber definitiv auszahlt!

4 Elementare explorative Grafiken

Wir geben einen Überblick über einige in **R** zur Verfügung stehende grafische Möglichkeiten, univariate oder bi- bzw. multivariate Datensätze gewissermaßen „zusammenfassend“ darzustellen. Die hierbei zum Einsatz kommenden Verfahren hängen nicht nur von der Dimensionalität, sondern auch vom Skalenniveau der Daten ab.

Die im Folgenden vorgestellten Funktionen erlauben dank ihrer Voreinstellungen in den meisten Anwendungsfällen ohne großen Aufwand die schnelle Erzeugung relativ guter und aussagefähiger Grafiken. Andererseits bieten alle Funktionen der Benutzerin und dem Benutzer viele weitere, zum Teil sehr spezielle Einstellungsmöglichkeiten für das Layout der Grafiken. Die Nutzung dieser Optionen kann dann jedoch zu ziemlich komplizierten Aufrufen führen, deren Diskussion hier zu weit ginge. In Kapitel 7 „Weiteres zur elementaren Grafik“ gehen wir diesbezüglich auf einige Aspekte näher ein, aber detaillierte Informationen zur Syntax der Funktionsaufrufe und genauen Bedeutung der im Einzelnen zur Verfügung stehenden Funktionsargumente sollten den jeweiligen Hilfeseiten entnommen werden.

4.1 Grafikausgabe am Bildschirm und in Dateien

Um eine grafische Ausgabe am Bildschirm oder in eine Datei zu ermöglichen, ist für den ersten Fall zunächst ein Grafikfenster und für den zweiten eine Grafikdatei zu öffnen. Jedes grafische Ausgabemedium, egal ob Bildschirm-Grafikfenster oder Grafikdatei, wird “device” genannt. Für die Verwendung dieser Devices dienen (unter anderem) die folgenden Befehle:

Öffnen und Schließen von Grafik-Devices:	
`> X11()` `> windows()` `> quartz()`	`x11()`, `X11()`, `windows()` (nicht unter Unix) oder `quartz()` (nur für Mac) öffnet bei jedem Aufruf ein neues Grafikfenster. Das zuletzt geöffnete Device ist das jeweils aktuell *aktive*, in welches die Grafikausgabe erfolgt.
`> dev.list()`	Listet Nummern und Typen aller geöffneten Grafik-Devices auf.
`> dev.off()`	Schließt das zuletzt geöffnete Grafik-Device *ordnungsgemäß*.
`> graphics.off()`	Schließt alle geöffneten Grafik-Devices auf einen Schlag.
`> postscript(file)` .... (Grafikbefehle)	Öffnet bei jedem Aufruf eine neue (EPS- (= “Encapsulated PostScript”-) kompatible) PostScript-Datei mit dem als Zeichenkette an `file` übergebenen Namen. Das zuletzt geöffnete Device ist das jeweils aktuelle, in welches die Grafikausgabe erfolgt.
`> dev.off()`	Schließt das zuletzt geöffnete Grafik-Device und *muss* bei einer PostScript-Datei unbedingt zur Fertigstellung verwendet werden, denn erst danach ist sie vollständig und korrekt interpretierbar.
`> pdf(file)` .... (Grafikbefehle) .... `> dev.off()`	Völlig analog zu `postscript`, aber eben für PDF-Grafikdateien.

Hinweise: Der Aufruf einer grafikproduzierenden Funktion, *ohne* dass vorher ein Grafik-Device explizit geöffnet wurde, aktiviert automatisch ein Grafik*fenster*, sodass ein `x11()`, `X11()` oder `windows()` für ein erstes Grafik*fenster* nicht nötig ist.

Die Funktionen `postscript` und `pdf` haben eine Vielzahl weiterer Argumente, die insbesondere dann Verwendung finden (können), wenn beabsichtigt ist, die Grafikdateien später z. B. in LaTeX-Dokumente einzubinden; wir verweisen hierfür auf die Hilfeseiten. Für einen Überblick über alle derzeit in **R** verfügbaren Formate grafischer Ausgabe ist `?Devices` hilfreich.

4.2 Explorative Grafiken für univariate Daten

In diesem Abschnitt listen wir einige Funktionen auf, die zur Darstellung und explorativen Analyse *univariater* Datensätze verwendet werden können. Zunächst (in §4.2.1) geschieht dies für (endlich) diskrete oder nominal- bzw. ordinalskalierte Daten, wo es im Wesentlichen um die Darstellung der beobachteten (absoluten oder relativen) Häufigkeiten der Werte geht. Dabei verwenden wir die Daten aus dem folgenden ...

Beispiel: In einer Studie zum Mundhöhlenkarzinom wurden von allen Patienten Daten über den bei ihnen aufgetretenen maximalen Tumordurchmesser in Millimeter (mm) und ihren „ECOG-Score“ erhoben. Letzterer ist ein qualitatives Maß auf der Ordinalskala 0, 1, 2, 3 und 4 für den allgemeinen physischen Leistungszustand eines Tumorpatienten (der von 0 bis 4 schlechter wird). Die Skala der Tumordurchmesser wurde in die vier Intervalle $(0, 20]$, $(20, 40]$, $(40, 60]$ und $(60, 140)$ eingeteilt.

Die rechts gezeigte `numeric`-Matrix `HKmat` enthält die gemeinsamen absoluten Häufigkeiten der gruppierten, maximalen Tumordurchmesser (pro Spalte) und der Realisierungen 0 bis 4 des ECOG-Scores (pro Zeile). Offenbar hat `HKmat` (durch sein `dimnames`-Attribut) selbsterklärend benannte Zeilen und Spalten.

```
> HKmat
      (0,20] (20,40] (40,60] (60,140)
ECOG0   1301    1976     699       124
ECOG1    173     348     189        59
ECOG2     69     157     104        34
ECOG3     16      27      18         7
ECOG4      5       6       4         2
```

Die Resultate der im Folgenden beschriebenen Funktionen zur grafischen Darstellung dieser Daten sind ab Seite 81 anhand von Beispiel-Plots zu bestaunen (hier “plot” (Engl.) = Diagramm).

Hinweise:

- Die Verwendung deutscher Umlaute in der Textausgabe für Grafiken kann etwas komplizierter sein, z. B. wenn sie nicht auf der Tastatur vorkommen (was bei uns natürlich eher selten der Fall ist). Im ersten der folgenden Beispiele wird aus Demonstrationszwecken daher einmalig Gebrauch von sogenannten „Escape-Sequenzen“ für die ASCII-Nummer (in Oktalsystemschreibweise) gemacht, um deutsche Umlaute in der Grafikausgabe zu generieren. Der Vollständigkeit halber listen wir die Escape-Sequenzen der deutschen Sonderzeichen in der folgenden Tabelle auf:

Sonderzeichen	ä	Ä	ö	Ö	ü	Ü	ß
Oktale Escape-Sequenz	`\344`	`\304`	`\366`	`\326`	`\374`	`\334`	`\337`

- Dateien aus einer anderen „Locale“ (d. h. regional- und sprachspezifischen Einstellung) als der eigenen können auf einem „fremden“ Zeichensatz basieren, also eine andere Enkodierung (= “encoding”) für `character`-Vektoren besitzen. Für die Konversion von Enkodierungen kann die Information unter `?Encodings` und die Funktion `iconv` hilfreich und nützlich sein.

4.2.1 Die Häufigkeitsverteilung diskreter Daten: Säulen-, Flächen- und Kreisdiagramme sowie Dot Charts

```
> barplot(HKmat[, 1],
+ main = "S\344ulendiagramm
+ f\374r gMAXTD =(0,20]")
```

(Memo: `\344` $\widehat{=}$ ä, `\374` $\widehat{=}$ ü. *Hier einmalig zu Demonstrationszwecken verwendet.)*

`barplot` mit einem Vektor als erstem Argument erzeugt ein **Säulen-** (oder **Balken-**)**Diagramm** mit Säulenhöhen, die diesem Vektor (hier: `HKmat[, 1]`) entnommen werden. Die Säulenbeschriftung geschieht automatisch über das `names`-Attribut des Vektors, falls vorhanden (oder explizit mit dem nicht gezeigten Argument `names.arg`). Die Plot-Überschrift wird durch `main` festgelegt. (Siehe oben links in Abb. 6.)

`> barplot(HKmat[, 1], col =` `+  rainbow(nrow(HKmat)),` `+ border = NA, ....)`	`col` erlaubt die Vergabe von Farben für die Säulen (wobei hier durch `rainbow(nrow(HKmat))` fünf Farben aus dem Spektrum des Regenbogens generiert werden; siehe Hilfeseite), `border = NA` schaltet die Säulenumrandung aus. (In Abb. 6 oben rechts. Details zu derartigen und weiteren, allgemeinen Grafik-Argumenten folgen in Kapitel 7.)
`> barplot(HKmat,` `+ angle = c(45, 135),` `+ density =` `+  10 * seq(nrow(HKmat)),` `+ legend.text = TRUE,` `+ main = "Flächendiagramm` `+ für HKmat")`	Eine Matrix als erstes Argument liefert für *jede Spalte* ein **Flächen-Diagramm**: Darin ist jede Säule gemäß der Werte in der jeweiligen Matrixspalte vertikal in Segmente unterteilt. Deren Schraffur-Winkel und -Dichten werden durch `angle` bzw. `density` bestimmt. Säulenbeschriftung: Matrix*spalten*namen (oder `names.arg`). Wegen `legend.text = TRUE` wird aus den Matrix*zeilen*namen eine entsprechende Legende erzeugt (auch explizit angebbar). Überschrift: `main`. (Siehe Abb. 6 unten links.)
`> barplot(HKmat,` `+ beside = TRUE, col =` `+  heat.colors(nrow(HKmat)),` `+ legend.text = TRUE, ....)`	`beside = TRUE` lässt für eine Matrix nebeneinander stehende Säulen entstehen, `col` steuert (zyklisch) die Farben innerhalb dieser Säulengruppen, `legend.text = TRUE` führt zur Legende. (Abb. 6 unten rechts.)

Hinweis: Die Hilfeseite zu `barplot` zählt weitere ihrer Argumente auf, u. a. das für Aussehen und Platzierung der Legende.

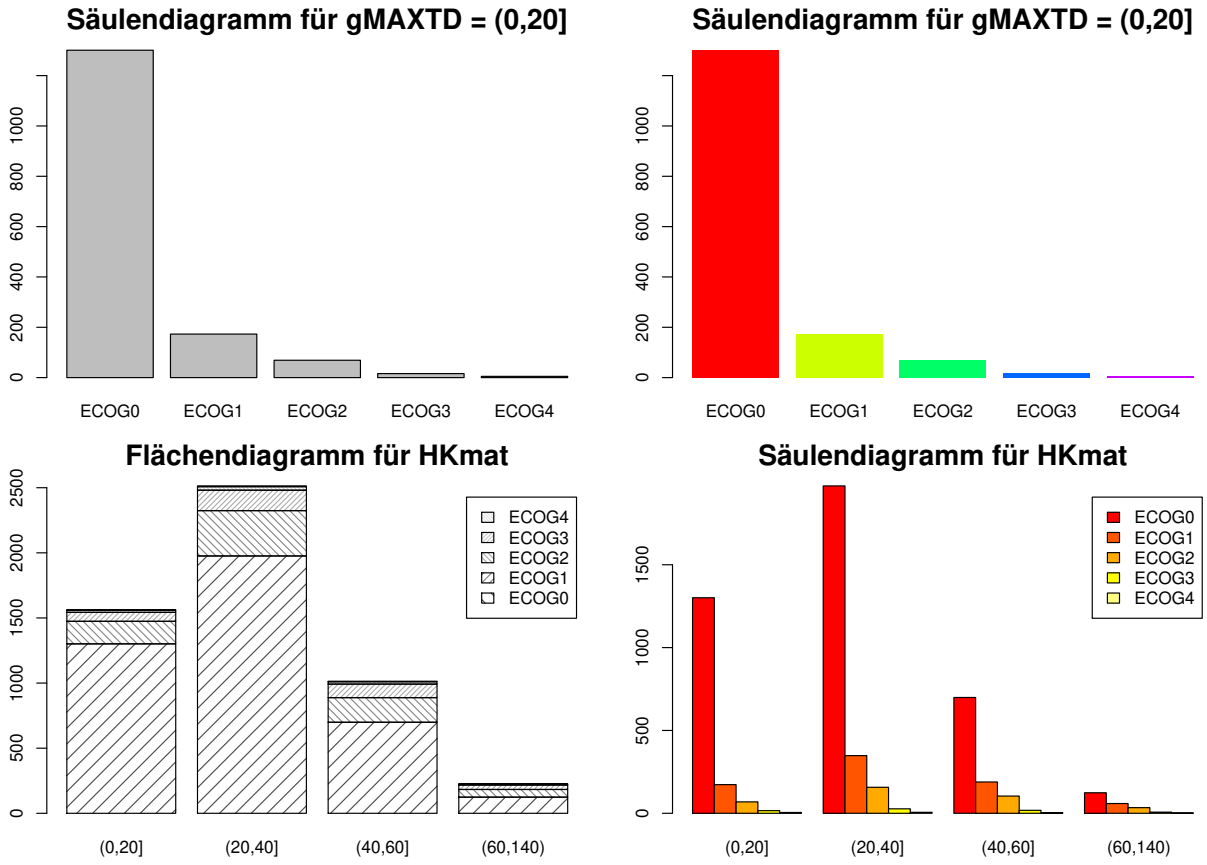

Abbildung 6: Säulen-, Flächen- und Kreisdiagramme.

`> dotchart(HKmat[, 1],` `+ main = "Dot Chart für` `+ gMAXTD = (0,20]")`	`dotchart` erzeugt zu einem Vektor für die Werte seiner Elemente einen **"Dot Chart"**. (Dies ist eine im Uhrzeigersinn um 90 Grad gedrehte und auf das absolut Nötigste reduzierte Art von Säulendiagramm.) Die „zeilenweise" Chart-Beschriftung geschieht automatisch über das `names`-Attribut des Vektors (oder explizit mit dem Argument `labels`). Überschrift: `main`. (Abb. 7 unten links.)
`> dotchart(HKmat,` `+ main = "Dot Chart für` `+ HKmat")`	Aus einer Matrix wird für jede Spalte ein gruppierter Dot Chart angefertigt. Dessen gruppeninterne „Zeilen"-Beschriftung geschieht einheitlich durch die Matrix*zeilen*namen. Die Charts werden in ein gemeinsames Koordinatensystem eingezeichnet und durch die Matrix*spalten*namen beschriftet. Überschrift: `main`. (Abb. 7 unten rechts.)

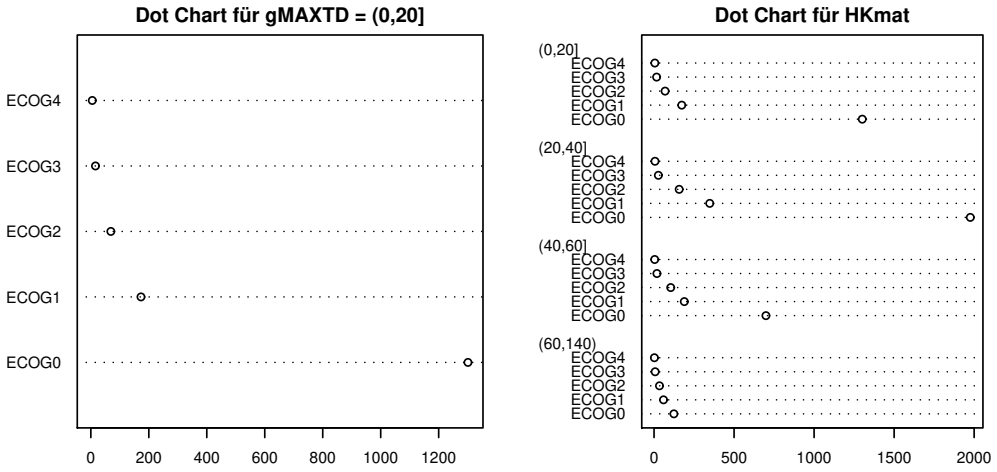

Abbildung 7: Dot Charts; links: ungruppiert; rechts: gruppiert.

Hinweise:

- Für einen Vektor erlaubt das einen Faktor erwartende `dotchart`-Argument `groups` die beliebige Gruppierung seiner Elemente gemäß des Faktors. Die Faktorlevels beschriften die Gruppen und mit dem weiteren `dotchart`-Argument `labels` ließen sich die gruppeninternen Zeilen benennen. (Nicht gezeigt; siehe Hilfeseite.)
- Die – eigentlich naheliegende – Übergabe des Resultates von `table` (siehe §2.7.6) oder von `xtabs` (siehe Hilfeseite oder später §**??** zu Unabhängigkeits- bzw. Homogenitätstests bzw. in §**??** zu Kontingenztafeln für $k \geq 2$ Faktoren) führt im Falle einer *ein*dimensionalen Häufigkeitstabelle zu einer Warnmeldung, weil `dotchart` einen numerischen Vektor (oder eine solche Matrix) erwartet, in den (bzw. die) ein `table`-Objekt erst noch konvertiert wird.

`> pie(HKmat[, 1],` `+ sub = paste("gMAXTD =",` `+  colnames(HKmat)[1]))`	`pie` erstellt zu einem Vektor (hier `HKmat[, 1]`) ein **Kreisdiagramm**, dessen Sektorenwinkel proportional zu den Werten der Vektorelemente sind. Die Einfärbung der Sektoren geschieht automatisch (oder explizit durch das Argument `col`) und ihre Benennung über das `names`-Attribut des Vektors (oder explizit mit dem Argument `labels`). (Mithilfe der Argumente `angle` und `density` können die Sektoren auch unterschiedlich schraffiert werden.) Einen Untertitel für den Plot liefert `sub` und `main` die Überschrift. (Siehe in Abb. 8 links oben. Die übrigen drei Diagramme wurden analog hergestellt, indem nacheinander die anderen Spalten von `HKmat` indiziert wurden.)

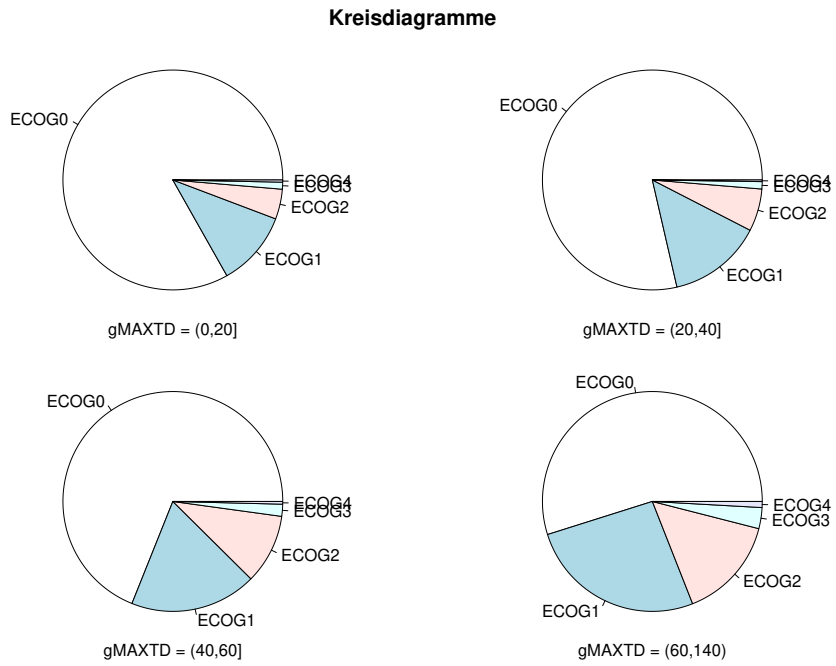

Abbildung 8: Kreisdiagramme.

Bemerkung: Kreisdiagramme sind zur Informationsdarstellung oft weniger gut geeignet als Säulendiagramme oder Dot Charts, weswegen letztere vorzuziehen sind ([22, Cleveland (1985)]).

4.2.2 Die Verteilung metrischer Daten: Histogramme, Kern-Dichteschätzer, "stem-and-leaf"-Diagramme, Boxplots, Strip Charts und Q-Q-Plots

Für die im Folgenden verwendeten, metrisch skalierten Beispieldaten

```
> X1 <- c(298, 345, 183, 340, 350, 380, 190, 351, 443, 290, 160, 298, 185,
+         370, 245, 377,  92, 380, 195, 265, 232, 358, 290, 307, 433, 255,
+         320, 237, 472, 438, 250, 340, 204, 282, 195, 251,  90, 200, 350, 620)
```

findet sich in §4.3.2 eine Kurzbeschreibung.

`> brks <- c(seq(80, 480,` `+ length = 7),` `+ 630),` `> hist(X1, freq = FALSE,` `+ breaks = brks,` `+ main = "Histogramm",` `+ xlab = "mm",` `+ ylab = "Dichte")`	`hist` erzeugt zu einem Vektor (hier `X1`) ein wegen `freq = FALSE` auf 1 flächennormiertes **Histogramm** für die Werte in diesem Vektor. (Bei `freq = TRUE` sind die Säulenhöhen absolute Klassenhäufigkeiten.) Die Klasseneinteilung der x-Achse erfolgt gemäß der Angabe von Klassengrenzen für `breaks`; wenn sie fehlt, werden aus den Daten automatisch Grenzen bestimmt. Überschrift: `main`; Achsenbeschriftung: `xlab` und `ylab`. (Siehe Abb. 9 links oben; zu weiteren Layout-Details siehe den darunter folgenden Hinweis.)
`> plot(density(X1),` `+ main =` `+ "Kern-Dichteschätzer",` `+ xlab = "mm",` `+ ylab = "Dichte")`	`density` berechnet einen sogenannten **(nicht-parametrischen) Kern-Schätzer** für die Dichte der den Daten zugrundeliegenden Verteilung (und ist quasi ein „geglättetes" Histogramm, wobei der Grad der Glättung anhand eines gewissen (Optimalitäts-)Kriteriums datenabhängig gewählt wird). Die Zeichnung des Graphen der geschätzten Dichte muss explizit durch `plot` angefordert werden. (Siehe Abb. 9 rechts oben; zu weiteren Layout-Details siehe den darunter folgenden Hinweis und zu einigen wenigen technischen Details Abschnitt 8.3.)

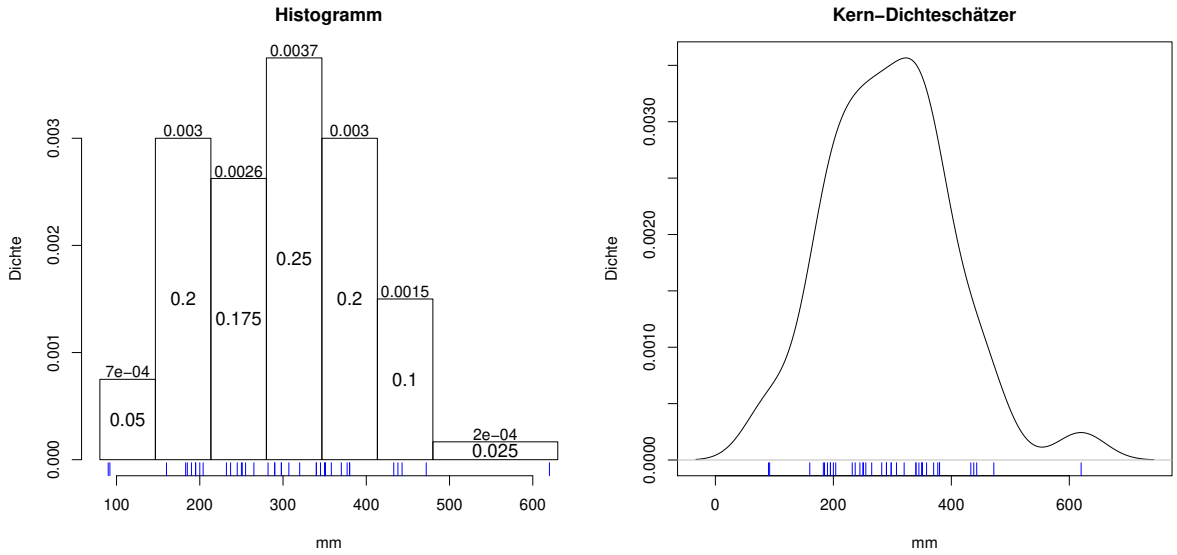

Abbildung 9: Links: Histogramm. Rechts: Kern-Dichteschätzer. Beide für dieselben Daten.

Hinweis: Das links gezeigte Histogramm wurde durch folgende Befehle mit den zusätzlichen (blauen) Markierungen der Rohdatenwerte sowie den Zusatzinformationen zu Säulenhöhen und -flächen versehen. (Der Plot des Kern-Dichteschätzer (rechts) wurde nur durch `rug` um die blauen Markierungen der Rohdaten auf der waagrechten Achse ergänzt.)

```
> histwerte <- hist(X1, ....(wie oben))
> rug(X1, col = "blue")   # Markierungen fuer die Rohdaten
> with(histwerte, {
+  text(x = mids, y = density/2, cex = 1.2,
+       labels = round(density * diff(brks), 4))
+  text(x = mids, y = density, pos = 3, offset = 0.1,
+       labels = round(density, 4))
+  })
```

Zum Rückgabewert von `hist` siehe ihre Hilfeseite. Die Funktion `with` wurde kurz schon in §2.10.9 auf S. 62 angesprochen und auf `text` gehen wir kurz in Abschnitt 7.2 ein bzw. verweisen auf ihre Hilfeseite. Die Funktion `hist` hat zwar auch ein Argument namens `labels`, dass, wenn man es auf `TRUE` setzt, dafür sorgt, dass die Säulenhöhen als Zahlenwerte auf die obere Kante einer jeden Säule geschrieben werden, allerdings werden diese Werte (fest eingestellt) nur auf drei Nachkommastellen gerundet ausgegeben.

```
> stem(X1)
 The decimal point is 2 digit(s)
 to the right of the |
  0 | 99
  1 | 6899
  2 | 00003455567899
  3 | 001244555567888
  4 | 3447
  5 |
  6 | 2

> stem(X1, scale = 2)
 The decimal point is 2 digit(s)
 to the right of the |
```

Ein **“stem-and-leaf”-Diagramm** ist eine halbgrafische Darstellung klassierter Häufigkeiten der Werte in einem Vektor, ähnlich der Art eines Histogramms. Allerdings gehen dabei die Werte selbst nicht „verloren“ (im Gegensatz zum Histogramm), sondern werden so dargestellt, dass sie bis auf Rundung rekonstruierbar sind. (Siehe z. B. [93, Tukey (1977)].) Dazu werden die Daten aufsteigend sortiert und die führenden Stellen der Werte zur Klasseneinteilung verwendet. Diese Einteilung liefert den Stamm (“stem”) des Diagramms, der durch den senkrechten Strich „`|`“ angedeutet wird. Hier sind es Hunderterklassen, was aus “`The decimal point is 2 digit(s) to the right of the |`” folgt. Die

`0 | 99` `1 |` `1 | 6899` `2 | 000034` `2 | 55567899` `3 | 001244` `3 | 555567888` `4 | 344` `4 | 7` `5 |` `5 |` `6 | 2`	nächste Stelle (in diesem Fall also die – gerundete – Zehnerstelle) der Daten bildet die Blätter ("leafs"), die rechts am Stamm „angeheftet" werden. Der Strich markiert also die Grenze zwischen Stamm und Blättern (hier zwischen Hundertern und Zehnern). Das Argument `scale` erlaubt, die „Auflösung" des Wertebereichs einzustellen: Je größer sein Wert, desto mehr Klassen werden gebildet. (Dadurch wird auch die „Höhe" des Stamms gesteuert.)

Boxplots und **Strip Charts** (= eindimensionale Streudiagramme) dienen nicht nur der – mehr oder minder übersichtlich zusammenfassenden – grafischen Darstellung metrischer Daten, sondern auch der qualitativen Beurteilung von Verteilungsannahmen über deren „Generierungsmechanismus". Das bezieht sich z. B. auf Verteilungseigenschaften wie Lage und Streuung, Symmetrie bzw. Asymmetrie (= Schiefe) sowie die Existenz von „Ausreißern". Beachten Sie hierzu auch die prototypischen Verteilungsformen in Abb. 12 am Ende dieses Paragraphen, die Skriptergänzung "Beware of Dynamite" sowie unbedingt die Erläuterungen zu Boxplots in §4.2.3!

`> x1 <- rnorm(50, 3, 3)` `> x2 <- rexp(200, 1/3)`	(`x1` und `x2` erhalten 50 bzw. 200 pseudo-zufällige $\mathcal{N}(3, 3^2)$- bzw. $Exp(1/3)$-verteilte, künstliche Beispieldaten. Für Details hierzu siehe Kapitel 5.)
`> boxplot(x1)`	Ein `numeric`-Vektor als erstes Argument liefert den **Boxplot** nur für dessen Inhalt. (Keine Grafik gezeigt.)
`> boxplot(list(x1, x2),` `+ names = c("N(3,9)-Daten",` `+ "Exp(1/3)-Daten"),` `+ main = "Boxplots",` `+ boxwex = 0.3)`	`boxplot` erzeugt für jede Komponente von `list(x1, x2)` einen Boxplot, in einem *gemeinsamen* Koordinatensystem nebeneinander angeordnet und durch die Zeichenketten in `names` benannt. Überschrift: `main`. Steuerung der relativen Breite der Boxen: `boxwex`. (Siehe Abb. 10 links und zur Theorie §4.2.3.) Anstelle einer Liste als erstes Argument von `boxplot` ist auch ein Data Frame möglich, da er als Liste aufgefasst werden kann (nicht gezeigt).
`> boxplot(split(` `+ SMSA$SCrimes, SMSA$GReg))`	`split` erzeugt eine für `boxplot` geeignete Liste, indem sie ihr erstes Argument (`numeric`) gemäß des zweiten (`factor`) auf Listenkomponenten aufteilt. (Ohne Bild.)
`> boxplot(SCrimes ~ GReg,` `+ data = SMSA)`	Das erste Argument von `boxplot` darf auch eine „Formel" sein. Der Formeloperator `~` (Tilde) bedeutet, dass seine linke (hier: `numeric`-)Variable auf der Ordinate abgetragen wird, und zwar „aufgeteilt" entlang der Abszisse gemäß seiner rechten (hier: `factor`-)Variablen. Quelle der Variablen: Der Data Frame für `data`. (Kein Bild gezeigt.)
`> stripchart(list(x1, x2),` `+ vertical = TRUE,` `+ method = "jitter",` `+ group.names = c("N(3,9)-` `+ Daten", "Exp(1/3)-Daten"),` `+ ....)`	`stripchart` erzeugt für jede Komponente von `list(x1, x2)` ein hier wg. `vertical = TRUE` senkrechtes, eindimensionales Streudiagramm (**Strip Chart**) der Daten, die hier infolge von `method = "jitter"` zur besseren Unterscheidbarkeit horizontal „verwackelt" sind (In Abb. 10 in der Mitte). Das erste Argument kann auch ein Vektor oder eine Formel sein (siehe die Hilfeseite).

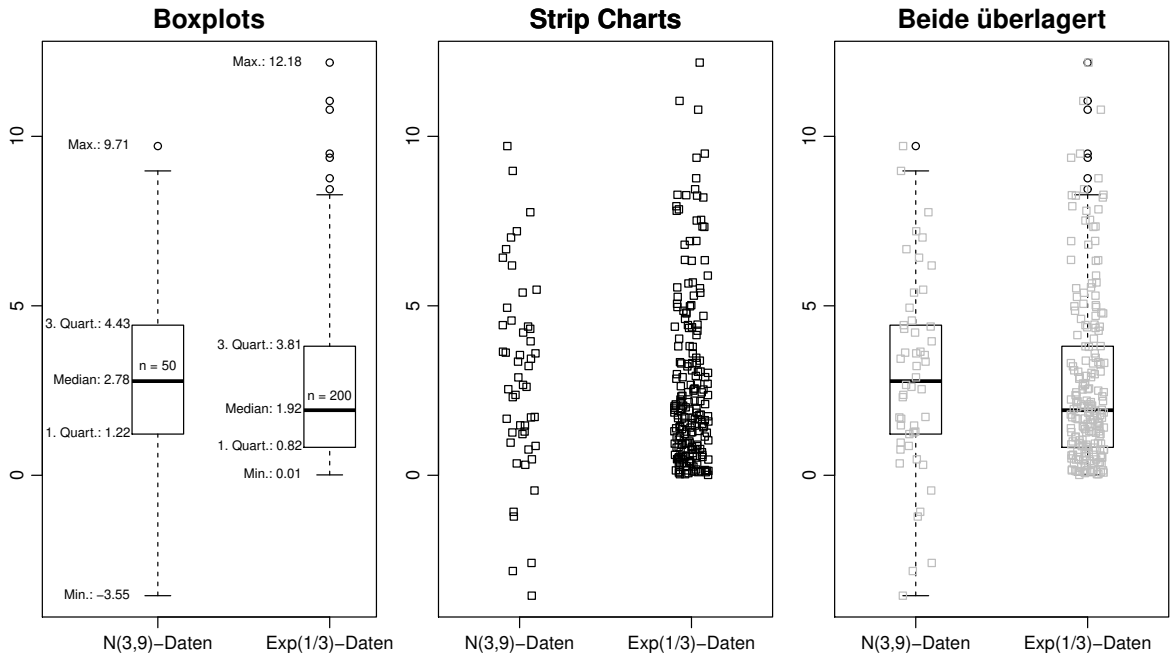

Abbildung 10: Jeweils für dieselben Daten: Links: Boxplots. Mitte: Strip Charts. Rechts: Boxplots und Strip Charts überlagert. Nicht gut ist die doppelte Darstellung der potenziellen Ausreißer; siehe hierzu den unten folgenden ersten Hinweis!

Hinweise:

1. Die Überlagerung von Boxplot und Strip Chart ist mittels `add = TRUE` (und evtl. `at`) in `stripchart` möglich (im Bild rechts). Dies ist empfehlenswert gerade bei wenigen Daten. Dabei muss kompensiert werden, dass `boxplot` und `stripchart` in ihren Voreinstellungen unterschiedlich ausgerichtete Grafiken liefern. Außerdem sollte man verhindern, dass `boxplot` Ausreißer separat markiert (siehe sein Argument `outline`), da diese von `stripchart` ja auch gezeichnet werden und dann in der Grafik (wie oben rechts) doppelt auftreten.

2. Die oben gezeigte linke Boxplots-Grafik wurde zur Erläuterung zum Teil mit Zusatzinfomationen versehen, und zwar (in etwa) durch die folgenden Befehle bzw. -ergänzungen:

```
> xlist <- list("N(3,9)-Daten" = x1, "Exp(1/3)-Daten" = x2)
> boxplot(xlist, main = "Boxplots", boxwex = 0.3, xlim = c(0.4, 2.2))
> bp <- boxplot(xlist, plot = FALSE, range = 0)
> shift <- 0.15  # = boxwex/2
> with(bp, {
+      text(x = stats[3,], labels = paste("n =", n), pos = 3, cex = 0.7)
+      for(i in 1:ncol(stats)) {
+       text(x = rep(i - shift, nrow(stats)), y = stats[, i],
+            labels = paste(c("Min.", "1. Quart.", "Median",
+                             "3. Quart.", "Max."),
+                           round(stats[, i], 2), sep = ": "),
+            pos = 2, offset = 0.1, cex = 0.7)
+       } } )
```

   Zu den Argumenten und dem Rückgabewert von `boxplot` siehe ihre Hilfeseite. Auf `with` wurde kurz schon auf Seite 62 in §2.10.9 hingewiesen. Auf `text` gehen wir kurz in Abschnitt 7.2 ein bzw. verweisen auf die Hilfeseite und der Funktion `for` widmen wir uns in Abschnitt 6.6 bzw. empfehlen die Hilfeseite via `?"for"` (beachte die hier am **R**-Prompt notwendigen Hochkommata!).

3. Eine interessante Erweiterung des Boxplots ist der Box-Percentile-Plot wie er z. B. im Paket `Hmisc` durch die Funktion `bpplot` realisiert wird. Eine „geglättete“ Version dazu ist der sogenannte Violinplot, für den z. B. in den Paketen `vioplot` und `ggplot2` entsprechende Funktionen zur Verfügung stehen. Zu Strip Charts mit zufälligem „Verwackeln“ sind als sehenswerte, deterministische Alternativen sogenannte “bee swarm”-Plots im Gebrauch; sie erhält man z. B. durch die Funktion `beeswarm` des gleichnamigen Paketes. Wir werden uns mit diese Alternativen kurz in den Übungen befassen.

Normal-Q-Q-Plots wurden speziell dafür entwickelt, eine qualitative Beurteilung der Normalverteilungsannahme vorzunehmen:

`> x <- rnorm(100, 3, 3)`	(`x` erhält 100 pseudo-zufällige $\mathcal{N}(3, 3^2)$-verteilte, künstliche Beispieldaten; Details folgen in Kapitel 5.)
`> qqnorm(x, main = "Normal-` `+ Q-Q-Plot mit Soll-Linie",` `+ sub = "100 Realisierungen` `+ der N(3, 9)-Verteilung")`	`qqnorm(x, ....)` liefert einen **Normal-Q-Q-Plot**, d. h. einen Plot der `x`-Ordnungsstatistiken gegen die entsprechenden Quantile der Standardnormalverteilung zur visuellen Prüfung der Normalverteiltheit der Werte in `x` (zur Theorie siehe §4.2.3). Überschrift und Untertitel: `main` und `sub`. (Siehe Abb. 11 links.) Mit dem (nicht gezeigten) Argument `datax = TRUE` lässt sich erreichen, dass die Daten nicht auf der vertikalen, sondern auf der horizontalen Achse abgetragen werden.
`> qqline(x)`	`qqline` zeichnet die „Soll-Linie“ ein, in deren Nähe die Punkte im Fall normalverteilter Daten theoretisch zu erwarten sind.

Abb. 11 links zeigt einen Normal-Q-Q-Plot für 100 Realisierungen der $\mathcal{N}(3, 3^2)$-Verteilung und rechts einen solchen für 100 Realisierungen der t-Verteilung mit 3 Freiheitsgraden, die im Vergleich zur Normalverteilung bekanntermaßen mehr Wahrscheinlichkeitsmasse in beiden Rändern hat. Dies schlägt sich im Normal-Q-Q-Plot nieder durch ein stärkeres „Ausfransen“ der Punktekette in beide Richtungen der “Sample Quantiles”-Achse, d. h. hier am linken Rand des Koordinatensystems nach unten und am rechten nach oben. Zur Theorie siehe Punkt 2 in §4.2.3!

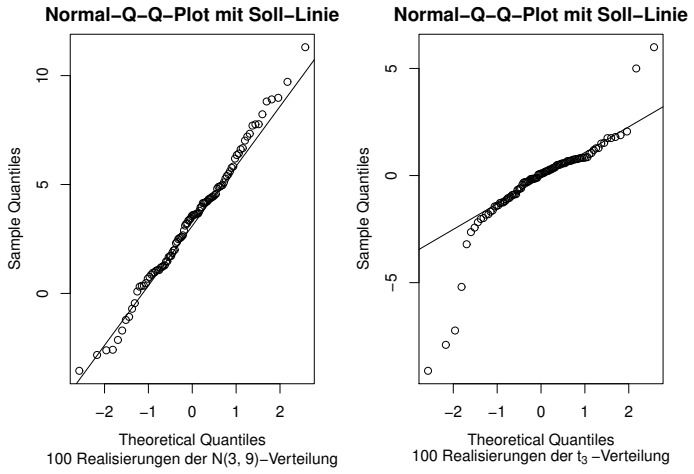

Abbildung 11: Normal Q-Q-Plots: links für normalverteilte, rechts für t_3-verteilte Daten.

Hinweise:

- Im Package `car` wird eine Funktion `qqPlot` zur Verfügung gestellt, die die Funktionalität von `qqnorm` und `qqline` erweitert durch ein Band punktweiser Konfidenzintervalle um die Soll-Linie, wodurch die Beurteilung der Zulässigkeit der Normalverteilungsannahme etwas

unterstützt wird. Zur Theorie dieser Konfidenzintervalle siehe [36, Fox (1997)], §3.1.3 und evtl. sogar auch den **R**-Code in `car:::qqPlot.default`.

- Um einen Eindruck vom möglichen Aussehen von Q-Q-Plots zu bekommen, werden wir entsprechende Simulationen in den Übungen durchführen.
- Abb. 12 zeigt einige prototypische Verteilungsformen in Gestalt ihrer Dichten und dazugehörigen theoretischen Boxplots sowie beispielhafter Normal-Q-Q-Plots für einige aus der jeweiligen Verteilung generierte Daten, um zu veranschaulichen, wie und inwieweit die Formen der drei Darstellungen idealtypisch zusammenhängen: Symmetrie bzw. Asymmetrie lässt sich offenbar leicht beurteilen, wohingegen die Modalität anhand von Boxplots in der Regel nicht erkennbar ist und anhand von Q-Q-Plots nur, wenn sie stark ausgeprägt ist.
- Für die qualitative Charakterisierung einer Verteilung(sform) können die folgenden Eigenschaften als wesentlich bezeichnet werden:
  - Stetig oder diskret, oder etwa eine Kombination davon?
  - Art des Trägers: Nominal, ordinal, metrisch? Endlich oder unendlich? Beschränkt oder unbeschränkt?
    Bspe.: $\{A, B, AB, 0\}, \{1, 2 \ldots, k\}, \mathbb{N}_0, [0, 1], \mathbb{R}^+, \mathbb{R}, \ldots$
  - Symmetrisch oder asymmetrisch, und für Letzteres evtl. genauer: links- oder rechtsschief?
  - Uni-, bi- oder multimodal?

4.2.3 Zur Theorie und Interpretation von Boxplots und Q-Q-Plots

1. Boxplots: Sie sind eine kompakte Darstellung der Verteilung eines metrischen Datensatzes mit Hervorhebung der wichtigen Kenngrößen Minimum, 1. Quartil, Median, 3. Quartil und Maximum. Sie erlauben die schnelle Beurteilung von Lage, Streuung und Symmetrie der Daten. Darüber hinaus werden potenzielle *Ausreißer* in den Daten markiert. Unter Ausreißern werden dabei isolierte oder isoliert wirkende Punkte verstanden, i. S. v. abseits vom „Hauptteil“ der Daten liegend wirkende *einzelne*(!) Werte. Ihre formale Identifikation geschieht hierbei („relativ“ zur Normalverteilung) gemäß des folgenden Kriteriums: Ist

$$\begin{aligned} X_i \;&>\; \text{3. Quartil} + 1.5 \cdot \text{Quartilsabstand} = X_{\frac{3n}{4}:n} + 1.5 \cdot \left(X_{\frac{3n}{4}:n} - X_{\frac{n}{4}:n} \right) \quad \text{oder} \\ X_i \;&<\; \text{1. Quartil} - 1.5 \cdot \text{Quartilsabstand} = \; X_{\frac{n}{4}:n} - 1.5 \cdot \left(X_{\frac{3n}{4}:n} - X_{\frac{n}{4}:n} \right), \end{aligned}$$

so gilt X_i als potenzieller Ausreißer und wird durch ein isoliertes Symbol (hier ein Kreis) markiert. Die (hier) gestrichelten – “whiskers” genannten – Linien, die von den Enden der zentralen Box ausgehen, erstrecken sich nach oben bis zum größten noch nicht als Ausreißer geltenden Datum und nach unten bis zum kleinsten noch nicht als Ausreißer geltenden Datum, das jeweils durch eine kleine „Kappe“ des whiskers markiert ist.

Begründung: Die obige Ausreißer-Identifikation ist durch die folgende, im Normalverteilungsfall gültige Approximation motivierbar (vgl. (1) und (3) unten):

$$X_{\frac{3n}{4}:n} + 1.5 \cdot \left(X_{\frac{3n}{4}:n} - X_{\frac{n}{4}:n} \right) \;\approx\; 4\sigma \cdot \Phi^{-1}(3/4) + \mu \;\approx\; 2.7 \cdot \sigma + \mu$$

Da $\mathbb{P}(|\mathcal{N}(\mu, \sigma^2) - \mu| > 2.7 \cdot \sigma) \approx 0.007$, sollten also im Fall $X_i \sim \mathcal{N}(\mu, \sigma^2)$ weniger als 1 % der Daten außerhalb der obigen Schranken liegen. Zur Erinnerung hier auch die „$k\sigma$-Regel“ der Normalverteilung:

$$\begin{array}{lll} k = \frac{2}{3}\colon & \mathbb{P}\left(\mu - \frac{2}{3}\sigma < X \leq \mu + \frac{2}{3}\sigma\right) = 0.4950 \approx 0.5 & \\ k = 1\colon & \mathbb{P}\left(\mu - \;\sigma < X \leq \mu + \;\sigma\right) = 0.6826 \approx \frac{2}{3} & \text{„}\sigma\text{-Intervall“} \\ k = 2\colon & \mathbb{P}(\mu - 2\sigma < X \leq \mu + 2\sigma) = 0.9544 \approx 0.95 & \text{„}2\sigma\text{-Intervall“} \\ k = 3\colon & \mathbb{P}(\mu - 3\sigma < X \leq \mu + 3\sigma) = 0.9973 \approx 0.997 & \text{„}3\sigma\text{-Intervall“} \end{array}$$

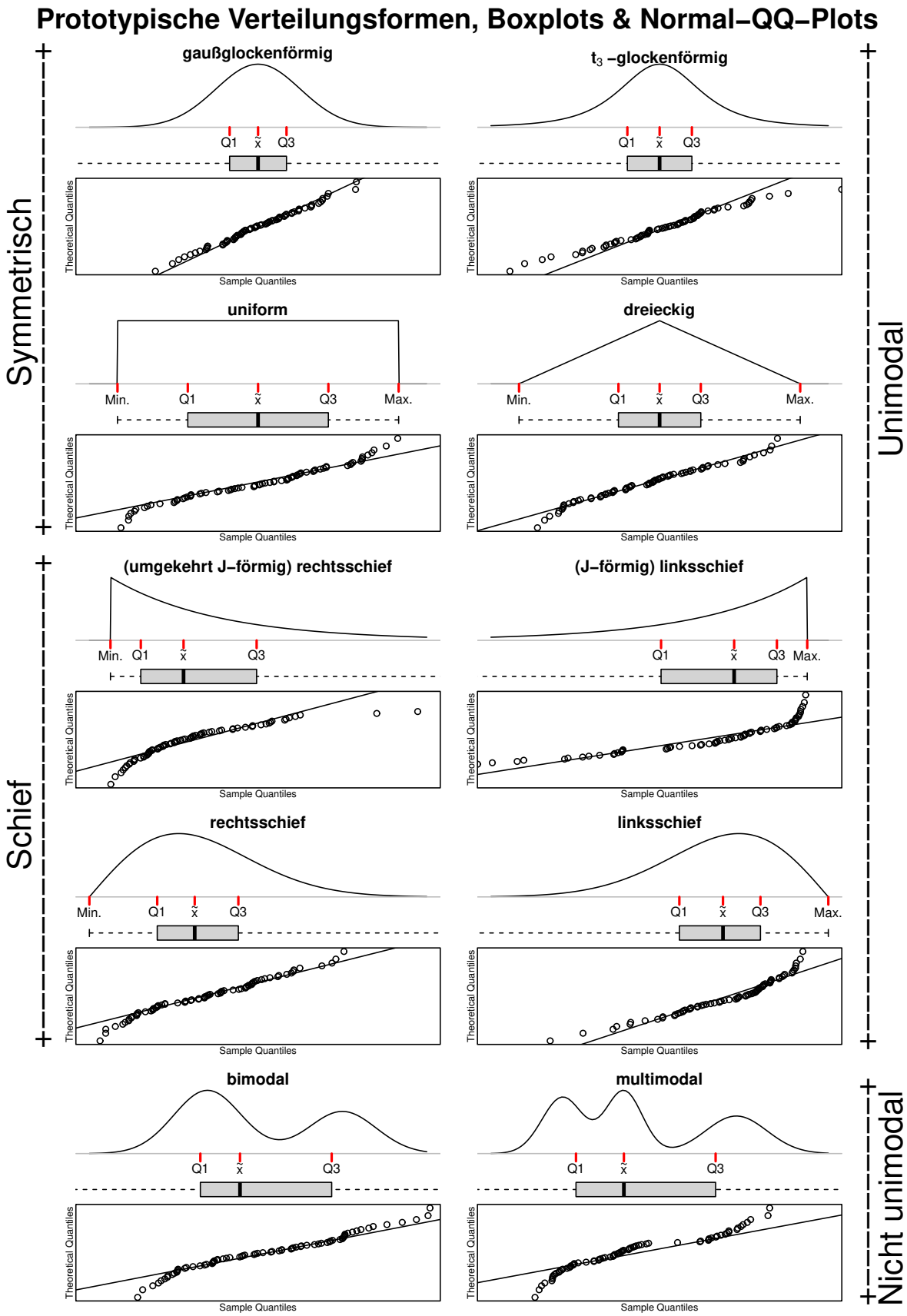

Abbildung 12: Prototypische Verteilungsformen, ihre Kategorisierung und deren Terminologie anhand von beispielhaften „Dreierpacks“ aus jeweils einer Dichte, dem dazugehörigen theoretischen Boxplot und dem Normal-Q-Q-Plot eines Beispieldatensatzes ($n = 75$) aus der zur Dichte gehörenden Verteilung.

2. Q-Q-Plots: Der Satz von Glivenko-Cantelli besagt für unabhängige Zufallsvariablen $X_1, \ldots, X_n \sim F$ mit beliebiger Verteilungsfunktion F, dass ihre empirische Verteilungsfunktion F_n mit Wahrscheinlickeit 1 (also „fast sicher“) gleichmäßig gegen F konvergiert. Kurz:

$$\sup_{q \in \mathbb{R}} |F_n(q) - F(q)| \longrightarrow 0 \text{ fast sicher für } n \to \infty$$

Daraus ist eine Aussage für die Konvergenz der empirischen Quantilfunktion F_n^{-1} herleitbar:

$$F_n^{-1}(p) \longrightarrow F^{-1}(p) \text{ fast sicher für } n \to \infty \text{ an jeder Stetigkeitsstelle } 0 < p < 1 \text{ von } F^{-1}$$

D. h., für hinreichend großes n ist $F_n^{-1}(p) \approx F^{-1}(p)$ an einem jeden solchen p. Und alldieweil wir für jede Ordnungsstatistik $X_{i:n}$ die Identität $F_n^{-1}(i/n) = X_{i:n}$ haben, muss gelten:

$$X_{i:n} \approx F^{-1}\left(i/n\right) \quad \text{für } i = 1, \ldots, n \text{ mit hinreichend großem } n \tag{1}$$

Die Approximation (1) verbessert sich, wenn man die Quantilfunktion etwas „shiftet“:

$$\begin{aligned} X_{i:n} &\approx F^{-1}\left((i-1/2)/n\right) \quad \text{für } n > 10 \qquad \text{bzw.} \\ X_{i:n} &\approx F^{-1}\left((i-3/8)/(n+1/4)\right) \quad \text{für } n \leq 10 \end{aligned} \tag{2}$$

Aufgrund dieser Approximationen sollten sich in einem Q-Q-Plot genannten Streudiagramm der – auch empirische Quantile heißenden – Ordnungsstatistiken $X_{i:n}$ gegen die theoretischen Quantile $F^{-1}((i-1/2)/n)$ bzw. $F^{-1}((i-3/8)/(n+1/4))$ die Punkte in etwa entlang der Identitätslinie $y = x$ aufreihen. Damit haben wir ein Vehikel, um die Verteilungsannahme für die X_i zu beurteilen: Sollte die „Punktekette“ des Q-Q-Plots *nicht* in etwa an der Identitätslinie entlang verlaufen, so können die Daten nicht aus der Verteilung zu F stammen.

Im **Fall der Normalverteilung** ist $F = \Phi_{\mu,\sigma^2} \equiv \Phi\left(\frac{\cdot-\mu}{\sigma}\right)$, wobei Φ die Verteilungsfunktion der Standardnormalverteilung ist, μ der Erwartungswert und σ^2 die Varianz. Aus $\Phi_{\mu,\sigma^2}(x) = \Phi\left(\frac{x-\mu}{\sigma}\right)$ folgt

$$\Phi_{\mu,\sigma^2}^{-1}(u) = \sigma\ \Phi^{-1}(u) + \mu, \tag{3}$$

sodass unter X_i i.i.d. $\sim \mathcal{N}(\mu, \sigma^2)$ für die Ordnungsstatistiken $X_{1:n}, \ldots, X_{n:n}$ gelten muss:

$$\begin{aligned} X_{i:n} &\approx \sigma\ \Phi^{-1}\left((i-0.5)/n\right) + \mu \quad \text{für } n > 10 \qquad \text{bzw.} \\ X_{i:n} &\approx \sigma\ \Phi^{-1}\left((i-3/8)/(n+1/4)\right) + \mu \quad \text{für } n \leq 10 \end{aligned} \tag{4}$$

Offenbar stehen die empirischen Quantile $X_{i:n}$ aus einer beliebigen Normalverteilung in einer approximativ *linearen* Beziehung zu den theoretischen Quantilen $\Phi^{-1}((i-0.5)/n)$ bzw. $\Phi^{-1}((i-3/8)/(n+1/4))$ der Standardnormalverteilung, wobei Steigung und „y-Achsenabschnitt“ dieser Beziehung gerade σ bzw. μ sind. In einem Streudiagramm dieser Quantile, das Normal-Q-Q-Plot genannt und durch die Funktion `qqnorm` geliefert wird, sollte die resultierende Punktekette demnach einen approximativ linearen Verlauf zeigen. (Dabei ist irrelevant, wo und wie steil diese Punktekette verläuft, denn die Zulässigkeit statistischer Verfahren hängt oft nur davon ab, dass die Daten überhaupt aus einer Normalverteilung stammen.)

Fazit: Zeigt sich im Normal-Q-Q-Plot *kein* approximativ linearer Verlauf, so ist die Normalverteilungsannahme für die X_i *nicht* zulässig.

Ergänzung: Der Erwartungswert μ und die Standardabweichung σ sind in der Praxis – auch unter der Normalverteilungsannahme – in der Regel unbekannt, lassen sich aber durch das arithmetische Mittel $\hat{\mu}$ und die Stichprobenstandardabweichung $\hat{\sigma}$ konsistent schätzen. Die Soll-Linie $y(x) = \sigma\, x + \mu$ für den Normal-Q-Q-Plot der $X_{i:n}$ gegen $\Phi^{-1}((i-0.5)/n)$ oder $\Phi^{-1}((i-3/8)/(n+1/4))$ *könnte* also durch $y(x) = \hat{\sigma}\, x + \hat{\mu}$ approximiert und zur Beurteilung des linearen Verlaufs der Punktekette als Referenz eingezeichnet werden, was jedoch *nicht* geschieht. Stattdessen wird aus Gründen der Robustheit (durch die Funktion `qqline`) diejenige Gerade eingezeichnet, die

durch die ersten und dritten empirischen und theoretischen Quartile verläuft, also durch die Punkte $(\Phi^{-1}(1/4), X_{\frac{n}{4}:n})$ und $(\Phi^{-1}(3/4), X_{\frac{3n}{4}:n})$. Sie hat die Gleichung

$$y(x) = \frac{X_{\frac{3n}{4}:n} - X_{\frac{n}{4}:n}}{\Phi^{-1}(3/4) - \Phi^{-1}(1/4)}\ x + \frac{X_{\frac{3n}{4}:n} + X_{\frac{n}{4}:n}}{2}$$

Was hat diese robuste Gerade mit der eigentlichen, linearen Beziehung zu tun? Antwort(en):

1. Für symmetrische Verteilungen ist das arithmetische Mittel von erstem und drittem empirischen Quartil ein guter Schätzer des Medians, der im Normalverteilungsfall gleich dem Erwartungswert μ ist. Also schätzt der y-Achsenabschnitt dieser robusten Geraden das μ.
2. Der empirische Quartilsabstand $X_{\frac{3n}{4}:n} - X_{\frac{n}{4}:n}$ ist ein Schätzer für $F^{-1}(3/4) - F^{-1}(1/4)$, wofür im Normalverteilungsfall gemäß (3) gilt: $F^{-1}(3/4) - F^{-1}(1/4) = \sigma\ (\Phi^{-1}(3/4) - \Phi^{-1}(1/4))$. Damit schätzt hier die Steigung dieser robusten Geraden das σ.

Bemerkungen:

- Als die „Bibel“ der explorativen Datenanalyse gilt [93, Tukey (1977)]. Eine Einführung in die grafische Datenanalyse geben auch [18, Chambers et al. (1983)]. In beiden Referenzen werden auch die obigen Verfahren beschrieben.
- Zum praktischen Sinn oder Unsinn eines statistischen Tests der Hypothese, dass Daten aus einer exakten Normalverteilung kommen, ist die Funktion `SnowsPenultimateNormalityTest` des **R**-Paketes `TeachingDemos` und ihre Hilfeseite eine humorvoll gehaltene Ermahnung.
  Auch lesenswert zu diesem Thema sind `https://stackoverflow.com/questions/7781798/seeing-if-data-is-normally-distributed-in-r` und `https://stats.stackexchange.com/questions/2492/is-normality-testing-essentially-useless`.

4.3 Explorative Grafiken für multivariate Daten

Für **multivariate Datensätze** ist die Darstellung aufwändiger und schwieriger (bis unmöglich). Die Häufigkeitstabelle eines zweidimensionalen, endlich-diskreten oder nominal- bzw. ordinalskalierten Datensatzes lässt sich noch durch ein multiples Balkendiagramm oder durch Mosaikplots veranschaulichen und die Verteilung von zwei- bzw. dreidimensionalen metrischskalierten Daten durch ein zwei- bzw. dreidimensionales Streudiagramm. Ab vier Dimensionen jedoch erlauben im metrischen Fall nur noch paarweise Streudiagramme aller bivariaten Kombinationen der Dimensionen der multivariaten Beobachtungsvektoren eine grafisch einigermaßen anschauliche (aber faktisch für höherdimensionale Strukturen unzulängliche) Betrachtung; Häufigkeitstabellen multivariat nominal- bzw. ordinalskalierter Daten werden schnell völlig unübersichtlich.

4.3.1 Die Häufigkeitsverteilung bivariat diskreter Daten: Mosaikplots

Für bivariate endlich-diskrete oder nominal- bzw. ordinalskalierte Daten sind Mosaikplots eine mögliche Darstellung der Häufigkeitstabelle der Wertepaare (auch Kontingenztafel genannt). Als Beispiel verwenden wir wieder die bereits als absolute Häufigkeiten in der `numeric`-Matrix `HKmat` vorliegenden Mundhöhlenkarzinomdaten (vgl. S. 80):

```
> HKtab <- as.table(HKmat)
> rownames(HKtab) <- 0:4
> dimnames(HKtab)
[[1]]
[1] "0" "1" "2" "3" "4"
[[2]]
[1] "(0,20]"   "(20,40]"
[3] "(40,60]"  "(60,140)"
```

Zunächst wandelt `as.table` das `matrix`-Objekt `HKmat` in ein `table`-Objekt um. (Sinnvollerweise erzeugt man solche Häufigkeitstabellen von vorneherein mit `table`, wie wir es später in Abschnitt **??** über einen Test im Multinomialmodell öfter tun werden. Außerdem verkürzen wir die Zeilennamen, damit die grafische Ausgabe übersichtlicher wird.)

```
> mosaicplot(HKtab,
+ xlab = "ECOG-Score",
+ ylab = "Gruppierter
+ maximaler
+ Tumordurchmesser")
```

`mosaicplot` erstellt für ein `table`-Objekt als erstem Argument (hier `HKtab`) den **Mosaikplot** in Abb. 13 links. Darin ist die Fläche von „Fliese“ (i, j) proportional zu H_{ij}/n, der relativen Häufigkeit in Tabellenzelle (i, j). Um dies zu erreichen, ist die Fliesenbreite proportional zu $H_{\cdot j}/n$, der relativen Spaltenhäufigkeit, und die Fliesenhöhe proportional zu $H_{ij}/H_{\cdot j}$, der Zellenhäufigkeit relativ zur Spaltenhäufigkeit.

```
> mosaicplot(t(HKtab)),
+ xlab = "Gruppierter
+ maximaler
+ Tumordurchmesser",
+ ylab = "ECOG-Score")
```

Daher ist die Darstellung nicht „transponierinvariant“, wie der Vergleich des Ergebnisses für `t(HKtab)` in Abb. 13 rechts mit der Grafik links daneben schnell klar macht.
Die „Zeilen“- und „Spalten“-Beschriftung wird dem `dimnames`-Attribut des `table`-Objekts entnommen, die „Achsen“-Beschriftung steuern `xlab` und `ylab` und die Überschrift ist mit `main` möglich. (Die gezeigten Grafiken haben eine eigene, nicht durch `mosaicplot` generierte Zusatzbeschriftung durch Zellenhäufigkeiten.)

Bemerkung: Durch Mosaikplots lassen sich auch höher- als zweidimensionale Häufigkeitsverteilungen endlich-diskreter oder nominal- bzw. ordinalskalierter Datensätze darstellen (siehe z. B. [38, Friendly, M. (1994)]). Im Vorgriff auf Abschnitt **??** zu Kontingenztafeln sei erwähnt, dass u. a. im Package `vcd` ([76, Meyer et al. (2006)]), welches durch “Visualizing Categorical Data” ([39, Friendly (2000)]) inspiriert wurde, weitere grafische Darstellungsmethoden für die Verteilung diskreter Daten zur Verfügung stehen. Ein in §**??** zum Cohen-Friendly Assoziationsplot auf S. **??** untergebrachter Hinweis liefert diesbzgl. weitere Informationen.

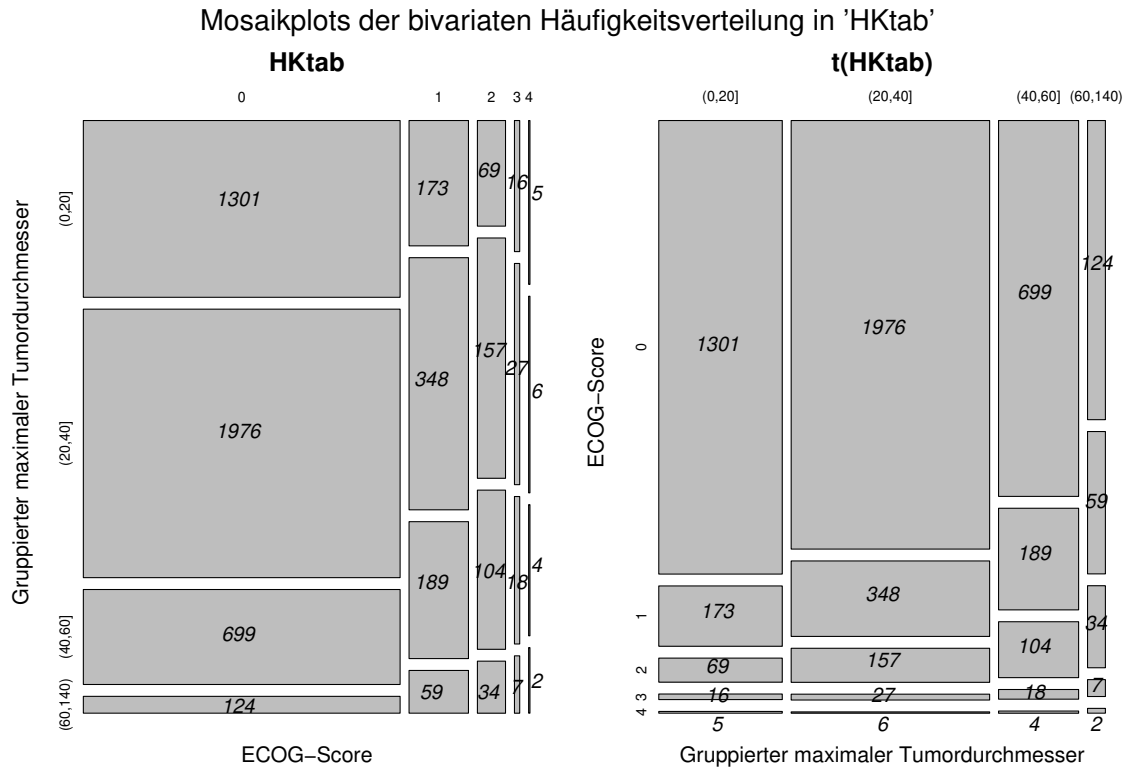

Abbildung 13: Mosaikplots.

4.3.2 Die Verteilung multivariat metrischer Daten: Streudiagramme

Als ein Beispiel für multivariate metrisch skalierte Daten dienen uns sechsdimensionale Messungen $(X_1, \ldots, X_6)$ an 40 Exemplaren des „Brillenschötchens“ (biscutella laevigata) (aus [91, Timischl (1990)], Seite 4), die in den jeweils 40-elementigen Vektoren `X1`, ..., `X6` gespeichert seien. Diese Variablen enthalten die folgenden (namensgleichen) Merkmale, gefolgt von einem Ausschnitt aus der Tabelle der Rohdaten:

X_1 : Sprosshöhe in mm
X_2 : Länge des größten Grundblattes in mm
X_3 : Anzahl der Zähne des größten Grundblattes an einem Blattrand
X_4 : Anzahl der Stengelblätter am Hauptspross
X_5 : Länge des untersten Stengelblattes in mm
X_6 : Spaltöffnungslänge in μm
X_7 : Chromosomensatz (d = diploid, t = tetraploid)
X_8 : Entwicklungsstand (1 = blühend, 2 = blühend und fruchtend,
3 = fruchtend mit grünen Schötchen,
4 = fruchtend mit gelben Schötchen)

i	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
1	298	50	1	6	39	27	d	4
2	345	65	2	7	47	25	d	1
3	183	32	0	5	18	23	d	3
⋮				...				
20	265	63	4	6	52	23	d	4

i	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
21	232	75	3	6	70	26	d	2
22	358	64	2	11	39	28	t	4
23	290	48	0	12	39	30	t	1
⋮				...				
40	620	48	4	10	40	26	d	2

Ein (zweidimensionales) Streudiagramm (Engl.: “scatter plot”) wie in Abb. 14 für die Realisierungen der zwei Variablen X_1 und X_5 zu sehen (genauer: der Elemente in `X5` gegen die Elemente in `X1`) kann wie folgt mit Hilfe der Funktion `plot` erzeugt werden:

```
> plot(X1, X5, xlab = "Sprosshöhe in mm",          # Produziert Abb. 14
+ ylab = "Länge unterstes Stengelblatt in mm",
+ main = "Streudiagramm")
```

Das erste Argument von `plot` ist der Vektor der waagrechten Koordinaten, das zweite der Vektor der senkrechten Koordinaten der zu zeichnenden Punkte. `xlab` und `ylab` liefern die Achsenbeschriftungen; `main` die Überschrift. (Mit dem hier nicht verwendeten Argument `sub` wäre noch ein Untertitel möglich.)

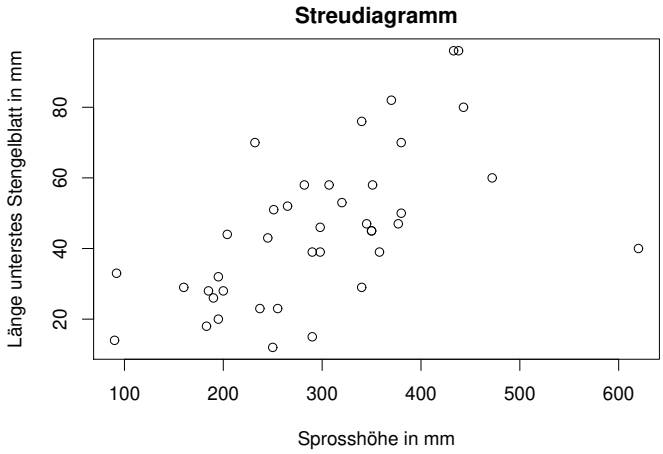

Abbildung 14: Streudiagramm.

Hinweis: Sollte es – anders als hier – in einem Streudiagramm z. B. aufgrund gerundeter metrischer Koordinaten zu starken Überlagerungen der Punkte kommen, kann ein "sunflower plot" hilfreich sein. Siehe `example(sunflowerplot)` und die Hilfeseite zu `sunflowerplot`. Für *sehr* umfangreiche und dicht liegende zweidimensionale Datensätze lohnt sich ein Blick auf die Darstellungen, die `example(smoothScatter)` präsentiert, oder einer auf `example(hexbinplot)` des **R**-Paketes `hexbin`.

Zusätzlich kann in ein Streudiagramm – sagen wir zur Unterstützung des optischen Eindrucks eines möglichen Zusammenhangs der dargestellten Variablen – eine sogenannte (nicht-parametrische) Glättungskurve (Engl.: "smooth curve") eingezeichnet werden (siehe Abb. 15). Dies ermöglicht die Funktion `scatter.smooth`. Sie bestimmt mit der sogenannten „loess"-Methode eine lokal-lineare oder lokal-quadratische Regressionskurve und zeichnet den dazugehörigen Plot. („loess" könnte von dem deutschen Begriff „Löss" (= kalkhaltiges Sediment des Pleistozäns, das sich oft sanft wellig zeigt) kommen, oder eine Abkürzung für "locally estimated scatter smoother" sein. Auf die technischen Details gehen wir hier nicht ein, sondern verweisen auf [20, Cleveland (1979)] als Primärquelle und für weitere Informationen z. B. auf[23, Cleveland et al. (1992)] oder [34, Fahrmeir et al. (2007), §7.1.7, S. 339] oder [98, Wand & Jones (1995)] unter "local polynomial regression estimates" oder auch auf `https://de.wikipedia.org/wiki/Kernel-Regression` unter lokal lineare Kernel-Regression. Der folgende **R**-Code produziert Abb. 15:

```
> scatter.smooth(X1, X5, span = 2/3, degree = 1, lpars = list(col = "red"),
+ xlab = "Sprosshöhe in mm", ylab = "Länge unterstes Stengelblatt in mm",
+ main = "Streudiagramm mit Glättungskurve")
```

Die ersten zwei Argumente enthalten die x- und y-Koordinaten der Datenpunkte. Die Argumente `span` und `degree` steuern Grad und Art der Glättung: `span` ist der Anteil der Daten, der in die lokale Glättung eingehen soll. Je größer `span`, umso „glatter" die Kurve. `degree = 1` legt eine lokal-lineare und `degree = 2` eine lokal-quadratische Glättung fest. Durch `lpars` können Grafikparameter für die Glättungskurve übergeben werden. `xlab`, `ylab` und `main` (und `sub`) funktionieren wie bei `plot`.

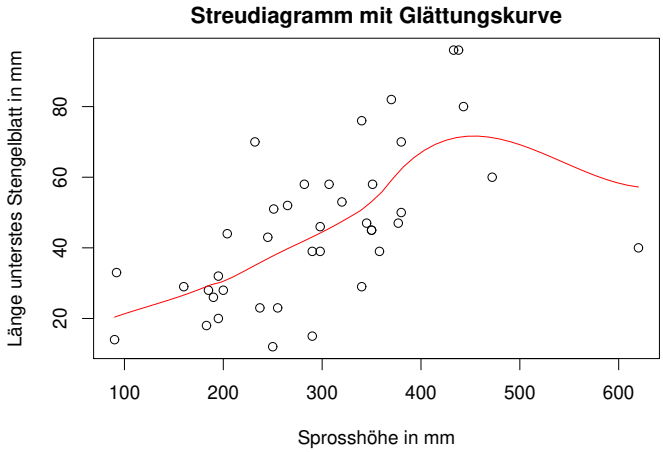

Abbildung 15: Streudiagramm mit nicht-parametrische Glättungskurve.

Ein Plot der Streudiagramme *aller* bivariaten Kombinationen der Komponenten multivariater Beobachtungsvektoren (hier 6-dimensional) befindet sich in Abb. 16. Es ist eine sogenannte Scatterplot-Matrix oder ein sogenannter Pairs-Plot für die Matrix `M <- cbind(X1, ...., X6)`, deren Zeilen als die multivariaten Beobachtungsvektoren aufgefasst werden. Er wird durch die Funktion `pairs` erzeugt, die außerdem durch ihr Argument `labels` eine explizite Beschriftung der Variablen im Plot ermöglicht (dabei ist „`\n`“ das Steuerzeichen für den Zeilenumbruch).

Auch in einen Pairs-Plot kann in jedes der Streudiagramme eine (nicht-parametrische) lokale Glättungskurve eingezeichnet werden. Dazu muss dem Argument `panel` der Funktion `pairs` die Funktion `panel.smooth` übergeben werden. Im folgenden Beispiel geschieht dies gleich in Form einer “in-line”-definierten Funktion mit einem speziell voreingestellten Argument für den Glättungsgrad (`span = 2/3`), nur um anzudeuten, wie dieser (und andere) Parameter hier variiert werden könnte(n). (Der gewählte Wert 2/3 ist sowieso auch der **R**-Voreinstellungswert.)

Die folgenden Befehle führten zu der in Abb. 16 gezeigten Grafik:

```
> var.labels <- c("Sprosshöhe\n(mm)", "Länge größ-\ntes Grund-\nblatt (mm)",
+ "Zähne\ngrößtes\nGrundblatt", "Stengelblätter\nHauptspross",
+ "Länge unters-\ntes Stengel-\nblatt (mm)",
+ "Spaltöff-\nnungslänge\n(Mikrom.)")

> pairs(cbind(X1, X2, X3, X4, X5, X6), labels = var.labels,
+ panel = function(x, y) {
+   panel.smooth(x, y, span = 2/3, col.smooth = "blue")
+   })
```

Bemerkungen:

- Die in `panel.smooth` zum Einsatz kommende lokale Glättungsmethode ist allerdings *nicht* die loess-Methode, sondern die sogenannte „lowess“-Methode (= “locally weighted scatter smoother”), die eine robuste, lokal gewichtete Glättung durchführt. (Siehe z. B. [21, Cleveland (1981)] oder auch auf `https://de.wikipedia.org/wiki/Kernel-Regression` unter lokal lineare Kernel-Regression.)
- An der „Kreuzung“ von erster Spalte und fünfter Zeile des Arrangements in Abb. 16 findet sich das Streudiagramm aus Abb. 15 wieder. Offensichtlich sind die jeweils darin eingezeichneten Glättungskurven verschieden.
- Für weitere, sehr leistungsfähige Argumente der Funktion `pairs` sowie ihre äußerst nützliche Formelvariante verweisen wir (wieder einmal) auf die Hilfeseite.

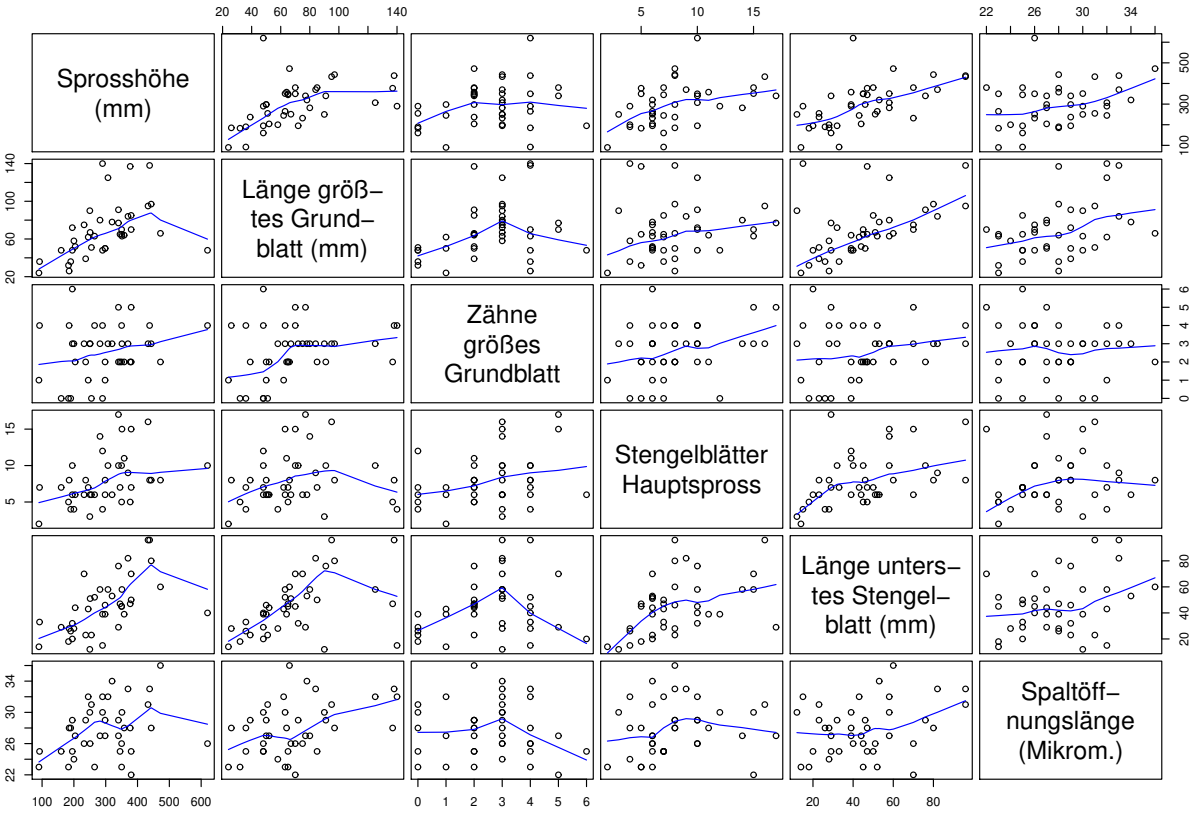

Abbildung 16: Pairs-Plot (= Scatterplot-Matrix) eines Teils der Brillenschötchendaten.

4.3.3 Die Verteilung trivariat metrischer Daten: Bedingte Streudiagramme ("co-plots")

Eine weitere Möglichkeit, eventuelle Zusammenhänge in dreidimensionalen Daten grafisch zweidimensional zu veranschaulichen, bieten die sogenannten "conditioning plots" (kurz: "co-plots") oder bedingten Plots. Hier werden paarweise Streudiagramme zweier Variablen (zusammen mit nicht-parametrischen Glättungskurven) unter der Bedingung geplottet, dass eine dritte Variable in ausgewählten Wertebereichen liegt. Realisiert wird dies durch die Funktion `coplot` und anhand eines Beispiels soll das Verfahren erläutert werden:

Im – i. d. R. mit "base **R**" mit-installierten – **R**-Paket `lattice` ist ein Datensatz namens `ethanol` eingebaut, der Daten aus einem Experiment mit einem Ein-Zylinder-Testmotor zur Abhängigkeit der NO_x-Emmission (`NOx`) von verschiedenen Kompressionswerten (`C`) und Gas-Luft-Gemischverhältnissen (`E`) enthält. Er wird durch `data` zur Verfügung gestellt und der folgende Pairs-Plot liefert einen ersten Eindruck der Daten:

```
> data(ethanol, package = "lattice")
> pairs(ethanol)
```

ergibt Abb. 17. Eine deutliche (evtl. quadratische) Abhängigkeit des `NOx` von `E` wird sofort offenkundig, was sich für `NOx` und `C` nicht sagen lässt. Auch ein Einfluss von `C` *auf die Abhängigkeit* des `NOx` von `E`, also eine *Wechselwirkung* zwischen `C` und `E`, ist so nicht zu entdecken.

Eine Möglichkeit hierzu bieten die nun beschriebenen co-plots, da sie eine detailliertere Betrachtung erlauben.
Zunächst betrachten wir den Fall, dass die **Variable** `C` (die nur fünf verschiedene Werte angenommen hat und somit als diskret aufgefasst werden kann) **als bedingende Größe** verwendet

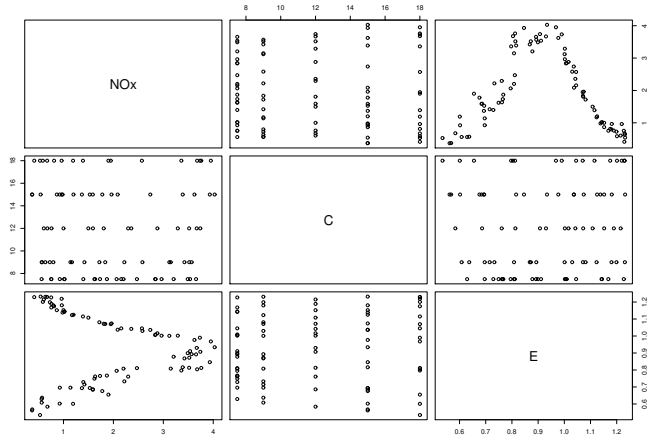

Abbildung 17: Pairs-Plot (= Scatterplot-Matrix) der Ethanol-Daten.

wird: Erst werden die `C`-Werte bestimmt, nach denen die Abhängigkeit von `NOx` und `E` bedingt werden soll (siehe `C.points` unten). Dann wird in `coplot` „`NOx` an `E` modelliert, gegeben `C`“ (mittels der Modellformel `NOx ~ E | C`), wozu die gegebenen `C`-Werte dem Argument `given.values` zugewiesen werden. Das Argument `data` erhält den Data Frame mit den in der Modellformel verwendeten Variablen und das Argument `panel` die Funktion `panel.smooth` für die lowess-Glättungskurve (vgl. hierzu Seite 95 im vorherigen Paragraphen):

```
> C.points <- sort(unique(ethanol$C))
> coplot(NOx ~ E | C, given.values = C.points, data = ethanol,
+ panel = panel.smooth)
```

Das Resultat in Abb. 18 ist wie folgt zu „lesen“: Die Werte(bereiche) der Bedingungsvariablen `C` sind in dem oberen „Panel“ (mit der Überschrift `Given : C`) als graue Balken dargestellt, denen die darunter gezeigten `NOx`-`E`-Streudiagramme so zugeordnet sind, dass diese (beginnend links unten) von links nach rechts und von unten nach oben zu den Balken im `Given`-Panel von link nach rechts gehören.

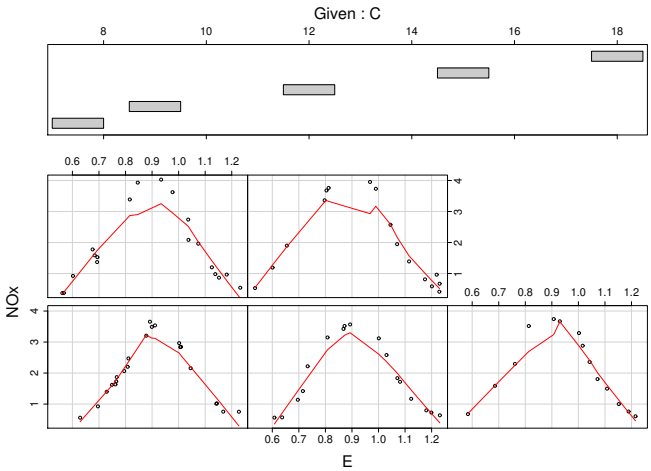

Abbildung 18: Conditioning plot der Ethanol-Daten.

Beobachtung: Über den gesamten Wertebereich von `C` hinweg scheint die Abhängigkeit von `NOx` von `E` vom selben (quadratischen?) Typ zu sein und dies auch gleichermaßen stark.

Nun wollen wir **nach der stetigen Variablen `E` bedingen**. Dazu wird mit der Funktion `co.intervals` der `E`-Wertebereich in `number` Intervalle aufgeteilt, die sich zu `overlap` überlappen (siehe `E.ints` unten) und nach denen die Abhängigkeit von `NOx` und `C` bedingt werden soll. Dann wird „`NOx` an `C` modelliert, gegeben `E`“, wozu die gegebenen `E`-Intervalle dem Argument `given.values` zugewiesen werden. Die Argumente `data` und `panel` fungieren wie eben, wobei hier an `panel` eine in-line-definierte Funktion mit spezieller Einstellung für den Glättungsparameter `span` in `panel.smooth` übergeben wird. Da hier `span = 1` ist, wird stärker geglättet als bei der Voreinstellung 2/3. Das Ergebnis ist in Abb. 19 zu sehen.

```
> E.ints <- co.intervals(ethanol$E, number = 9, overlap = 0.25)
> coplot(NOx ~ C | E, given.values = E.ints, data = ethanol,
+ panel = function(x, y, ...) { panel.smooth(x, y, span = 1, ...) } )
```

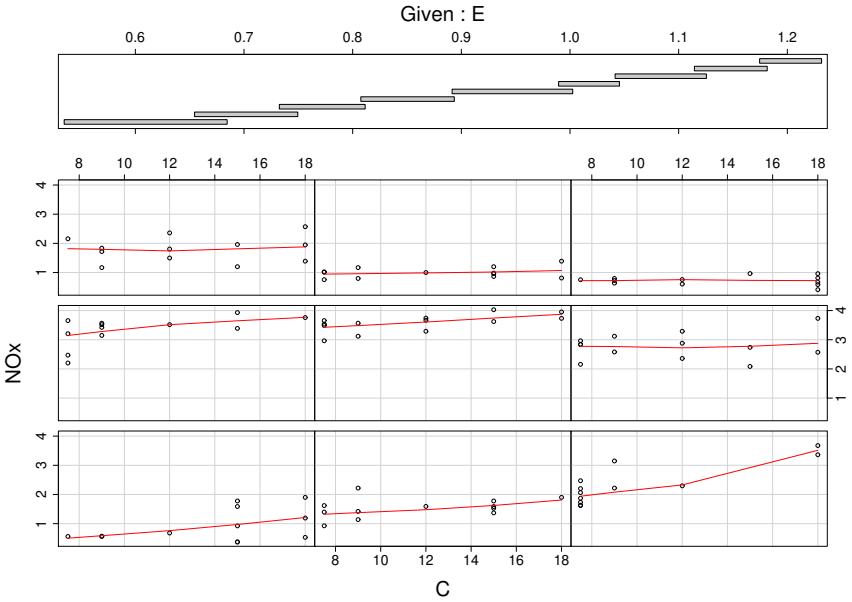

Abbildung 19: Conditioning plot der Ethanol-Daten.

Beobachtung: Für niedrige Werte von `E` nimmt `NOx` mit steigendem `C` zu, während es für mittlere und hohe `E`-Werte als Funktion von `C` (!) konstant zu sein scheint.

Hinweise: Weitere und zum Teil etwas komplexere Beispiele zeigt Ihnen `example(coplot)`. Eine andere, ebenfalls beachtenswerte Darstellung der Abhängigkeit der Verteilung einer Variablen von zwei anderen liefert die sog. Streuungsfächerkarte (Engl. “fan chart”, siehe z. B. `https://de.wikipedia.org/wiki/Streuungsf%C3%A4cherkarte`).

4.3.4 Weitere Möglichkeiten und Hilfsmittel für multivariate Darstellungen: `stars`, `symbols`

Eine weitere Möglichkeit, jede einzelne multivariate Beobachtung zu veranschaulichen, und zwar als „Sternplot“, bietet `stars` (manchmal auch „Spinnen-“ oder „Radarplot“ genannt); siehe Abb. 20. Anhand eines Ausschnitts aus dem Datensatz `UScereal` im **R**-Paket `MASS` wird die Methode kurz vorgestellt, aber näher wollen wir hier nicht darauf eingehen. Zu Details siehe `?stars` und außerdem `help(UScereal, package = "MASS")`.

```
> data(UScereal, package = "MASS")
> stars(UScereal[-c(1, 9)][1:15,], nrow = 3, ncol = 5, cex = 1.1,
+ main = "Nährwertinformationen verschiedener Cerealiensorten")
```

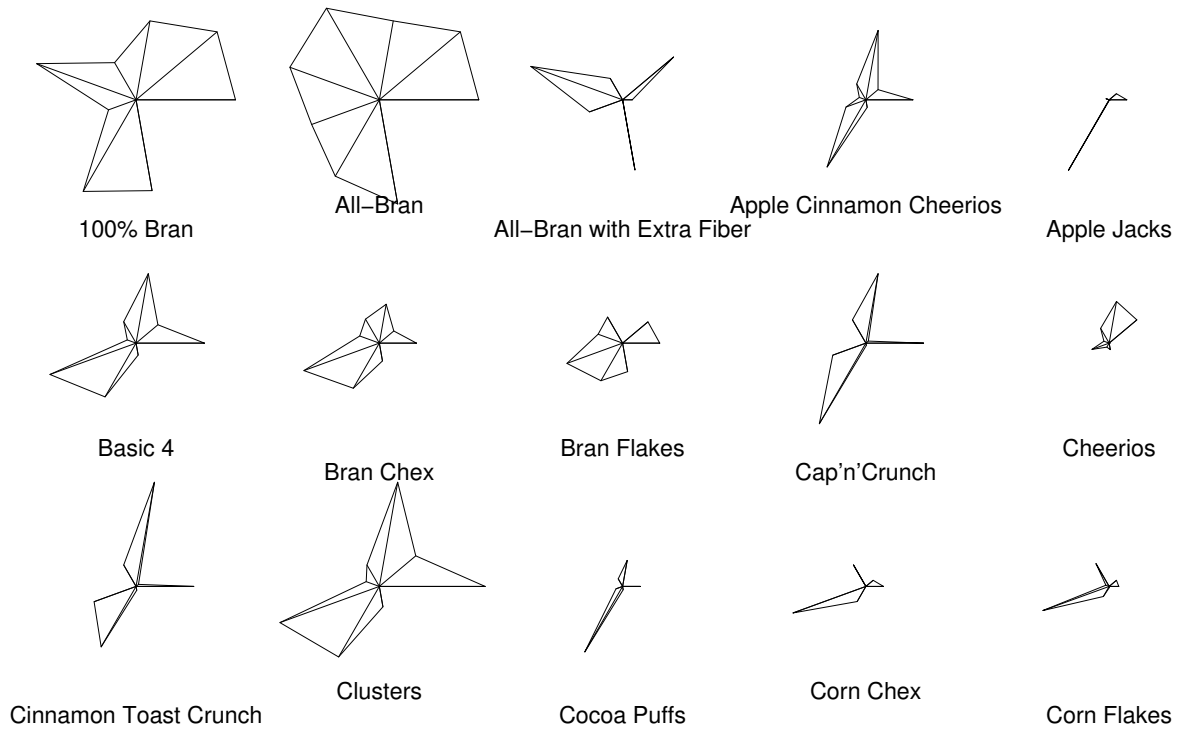

Abbildung 20: Sternplots der Cerialien-Daten.

Hinweise:

- Erläuterungen zur Interpretation der Sternplots finden Sie in den Details der Hilfeseite von `stars`. Die (sehr empfehlenswerten) Beispiele auf jener Hilfeseite liefern Ihnen außerdem andere oder in der Ausgestaltung ausgefeiltere sowie komplexere Varianten; siehe also `example(stars)`.
- Über die Fähigkeiten und die Anwendung der Funktion `symbols` sollten Sie sich durch ihre via `example(symbols)` erhältliche Beispielekollektion sowie die zugehörige Hilfeseite informieren.
- Amüsant und interessant, auch weil 100 % ernst gemeint, ist das Darstellungskonzept der "Chernoff faces" für multivariate Daten, wozu Sie z. B. unter `https://de.wikipedia.org/wiki/Chernoff_faces` mehr finden.
- Ein interessantes sowie weit- und tiefgehendes Buch zum Thema Grafik in **R** ist "R Graphics" ([78, Murrell (2005)] bzw. seine zweite Auflage [79, Murrell (2011)]). Dasselbe gilt für das exzellente Buch "Lattice. Multivariate Data Visualization with R" ([85, Sarkar (2008)]), das die **R**-Implementation und - Erweiterung des hervorragenden "Trellis-Grafik"-Systems im **R**-Paket `lattice` vorstellt. Hierin ist für multivariate Daten insbes. die Darstellung mit Hilfe sog. "parallel coordinate plots" (durch die Funktion `parallelplot`) zu erwähnen. Ebenfalls höchst interessant und leistungsfähig ist das **R**-Paket `ggplot2`, das in ([102, Wickham (2009)]) beschrieben wird.

5 Wahrscheinlichkeitsverteilungen und Pseudo-Zufallszahlen

R hat bereits in der "base distribution" für viele Wahrscheinlichkeitsverteilungen sowohl die jeweilige Dichtefunktion (im Fall einer stetigen Verteilung) bzw. Wahrscheinlichkeitsfunktion (im diskreten Fall) als auch Verteilungs- und Quantilfunktion implementiert; fallweise natürlich nur approximativ. Ebenso sind Generatoren für Pseudo-Zufallszahlen implementiert, mit denen Stichproben aus diesen Verteilungen simuliert werden können. Wenn wir im Folgenden (zur Abkürzung) von der Erzeugung von Zufallszahlen sprechen, meinen wir stets *Pseudo*-Zufallszahlen. (Beachte: In der **R**-Terminologie wird nicht zwischen Dichtefunktion einer stetigen Verteilung und Wahrscheinlichkeitsfunktion einer diskreten Verteilung unterschieden, sondern beides als "density function" bezeichnet.)

5.1 Die „eingebauten" Verteilungen

Die **R**-Namen der obigen vier Funktions*typen* setzen sich wie folgt zusammen: Ihr erster Buchstabe gibt den Funktionstyp an, der Rest des Namens identifiziert die Verteilung, für die dieser Typ zu realisieren ist. Der Aufruf eines Funktionstyps einer Verteilung benötigt (neben eventuell anzugebenden spezifischen Verteilungsparametern) die Angabe der Stelle, an der eine Funktion ausgewertet werden soll, bzw. die Anzahl der zu generierenden Zufallszahlen. Etwas formaler gilt allgemein für eine beliebige, implementierte Verteilung *dist* (mit möglichen Parametern „....") mit Verteilungsfunktion (VF) $F_{....}$ und Dichte- bzw. Wahrscheinlichkeitsfunktion $f_{....}$:

$$\left.\begin{array}{ll} \texttt{d}\mathit{dist}\texttt{(x, ...)} & = f_{....}(x) \;:\; \text{Dichte-/Wahrscheinlichkeitsfkt.} \\ \texttt{p}\mathit{dist}\texttt{(x, ...)} & = F_{....}(x) \;:\; \text{VF (also } \mathbb{P}(X \le x) \text{ für } X \sim F_{....}) \\ \texttt{q}\mathit{dist}\texttt{(y, ...)} & = F^{-1}_{....}(y) : \text{Quantilfunktion} \\ \texttt{r}\mathit{dist}\texttt{(n, ...)} & \text{liefert } n \text{ Zufallszahlen aus} \end{array}\right\} \text{der Verteilung } \mathit{dist}$$

Beispiele: Für den stetigen Fall betrachten wir die (Standard-)Normalverteilung:

$$\left.\begin{array}{ll} \texttt{dnorm(x)} & = \phi(x) \quad : \text{Dichte} \\ \texttt{pnorm(x)} & = \Phi(x) \quad : \text{VF} \\ \texttt{qnorm(y)} & = \Phi^{-1}(y) : \text{Quantilfunktion} \\ \texttt{rnorm(n)} & \text{liefert } n \text{ Zufallszahlen aus} \end{array}\right\} \text{der Standardnormalverteilung}$$

Gemäß der Voreinstellung liefern Aufrufe der Funktionstypen aus der Familie der Normalverteilung immer Resultate für die *Standard*normalverteilung, wenn die Parameter `mean` und `sd` nicht explizit mit Werten versehen werden:

Die Normalverteilung	
`> dnorm(c(7, 8), mean = 10)` `[1] 0.004431848 0.053990967`	Werte der Dichte der $\mathcal{N}(10, 1)$-Verteilung an den Stellen 7 und 8.
`> pnorm(c(1.5, 1.96))` `[1] 0.9331928 0.9750021`	Werte der Standardnormalverteilungsfunktion Φ an den Stellen 1.5 und 1.96.
`> qnorm(c(0.05, 0.975))` `[1] -1.644854  1.959964`	Das 0.05- und das 0.975-Quantil der Standardnormalverteilung, also $\Phi^{-1}(0.05)$ und $\Phi^{-1}(0.975)$.
`> rnorm(6, mean = 5, sd = 2)` `[1] 5.512648 8.121941 5.672748` `[4] 7.665314 4.081352 4.632989`	Sechs Zufallszahlen aus der $\mathcal{N}(5, 2^2)$-Verteilung.

Für den diskreten Fall betrachten wir die Binomialverteilung:

$$\left.\begin{array}{lll} \texttt{dbinom(k, size =}\ m\texttt{, prob =}\ p\texttt{)} & = \mathbb{P}(X = k) & : \text{ W.-Funktion} \\ \texttt{pbinom(k, size =}\ m\texttt{, prob =}\ p\texttt{)} & = F(k) := \mathbb{P}(X \le k) & : \text{ VF} \\ \texttt{qbinom(y, size =}\ m\texttt{, prob =}\ p\texttt{)} & = F^{-1}(y) & : \text{ Quantilfunktion} \\ \texttt{rbinom(n, size =}\ m\texttt{, prob =}\ p\texttt{)} & \text{liefert } n \text{ Zufallszahlen aus} & \end{array}\right\} \ldots$$

... der Binomial(m, p)-Verteilung (d. h. für $X \sim \text{Bin}(m, p)$).

Hier zwei Tabellen vieler der in **R** zur Verfügung stehenden Verteilungen, ihrer jeweiligen Funktionsnamen und Parameter (samt Voreinstellungen, sofern gegeben; beachte auch die Ergänzung auf Seite 102 oben):

Diskrete Verteilungen		
... (-)Verteilung	**R**-Name	Verteilungsparameter
Binomial	`binom`	`size, prob`
Geometrische	`geom`	`prob`
Hypergeometrische	`hyper`	`m, n, k`
Multinomial	`multinom`	`size, prob` (nur `r....` und `d....`)
Negative Binomial	`nbinom`	`size, prob`
Poisson	`pois`	`lambda`
Wilcoxons Vorzeichen-Rangsummen	`signrank`	`n`
Wilcoxons Rangsummen	`wilcox`	`m, n`

Stetige Verteilungen		
... (-)Verteilung	**R**-Name	Verteilungsparameter
Beta	`beta`	`shape1, shape2, ncp = 0`
Cauchy	`cauchy`	`location = 0, scale = 1`
χ^2	`chisq`	`df, ncp = 0`
Exponential	`exp`	`rate = 1`
F	`f`	`df1, df2 (ncp = 0)`
Gamma	`gamma`	`shape, rate = 1`
Log-Normal	`lnorm`	`meanlog = 0, sdlog = 1`
Logistische	`logis`	`location = 0, scale = 1`
Multivariate Normal (im package `mvtnorm`)	`mvnorm`	`mean = rep(0,`*`d`*`), sigma = diag(`*`d`*`)` (mit *d* = Dimension)
Multivariate t (im package `mvtnorm`)	`mvt`	(etwas komplizierter; siehe seine Online-Hilfe)
Normal	`norm`	`mean = 0, sd = 1`
Students t	`t`	`df, ncp = 0`
Uniforme	`unif`	`min = 0, max = 1`
Weibull	`weibull`	`shape, scale = 1`

Hinweise: Mehr Informationen über die einzelnen Verteilungen – wie z. B. die Beziehung zwischen den obigen **R**-Funktionsargumenten und der mathematischen Parametrisierung der Dichten – liefert die Hilfeseite, die etwa mit dem Kommando `?d`*dist* konsultiert werden kann, wenn etwas über die Verteilung namens *dist* (vgl. obige Tabelle) in Erfahrung gebracht werden soll. Beachte: `?`*dist* funktioniert *nicht* oder liefert nicht das Gewünschte! Aber `?distribution` liefert eine Übersicht über alle Verteilungen im Paket `stats` mit Links auf deren Hilfeseiten. (Zusätzliche Möglichkeit: Mittels `help.search("distribution")` erhält man u. A. Hinweise auf alle Hilfedateien, in denen etwas zum Stichwort “distribution” steht.) Ist man an einem umfangreichen

Überblick über die in "base **R**" und anderen **R**-Paketen zur Verfügung gestellten Verteilungen interessiert, lohnt sich ein Blick auf den Task View "Probability Distributions" auf CRAN unter `http://cran.r-project.org` → Task Views → Distributions.

Ergänzung: Urnenmodelle sind in **R** ebenfalls realisierbar, und zwar mithilfe der Funktion `sample`. Mit ihr kann das Ziehen von Elementen aus einer (endlichen) Grundmenge mit oder ohne Zurücklegen simuliert werden. Es kann auch festgelegt werden, mit welcher Wahrscheinlichkeit die einzelnen Elemente der Grundmenge jeweils gezogen werden sollen, wobei die Bedeutung dieser Festlegung beim Ziehen mit Zurücklegen eine andere ist als beim Ziehen ohne Zurücklegen. Die Voreinstellung von `sample` produziert eine zufällige Permutation der Elemente der Grundmenge. In Abschnitt **??** „Bernoulli-Experimente mit `sample`" gehen wir etwas näher darauf ein; vorerst verweisen wir für Details auf die Hilfeseite.

5.2 Bemerkungen zu Pseudo-Zufallszahlen in R

Die Generierung von Pseudo-Zufallszahlen aus den verschiedenen Verteilungen basiert auf uniformen Pseudo-Zufallszahlen, welche mit einem Zufallszahlengenerator (Engl.: "random number generator" = RNG) erzeugt werden. In **R** stehen im Prinzip mehrere verschiedene RNGs zur Verfügung. Per Voreinstellung ist es der „Mersenne-Twister", ein uniformer RNG, der eine sehr lange, aber letztendlich doch periodische Zahlensequenz (der Länge $2^{19937} - 1$) im offenen Intervall $(0, 1)$ erzeugt. Diesbzgl. wird in der Hilfeseite die Publikation [74, Matsumoto und Nishimura (1998)] zitiert. Weitere Details und zahlreiche Literaturverweise liefert `?RNG`. (Außerdem interessant in diesem Zusammenhang könnten die folgenden Websites sein: `https://en.wikipedia.org/wiki/Random_number_generation`, `https://www.random.org/`, `https://www.fourmilab.ch/hotbits/`.)

Der Zustand des RNGs wird von **R** in dem Objekt `.Random.seed` im workspace (gewissermaßen unsichtbar) gespeichert. Vor dem **aller**ersten Aufruf des RNGs existiert `.Random.seed` jedoch noch nicht, z. B. wenn **R** in einem neuen Verzeichnis zum ersten Mal gestartet wird. Den Startzustand des RNGs leitet **R** dann aus der aktuellen Uhrzeit ab, zu der der RNG zum ersten Mal aufgerufen wird. Bei der Erzeugung von Zufallszahlen ändert sich der Zustand des RNGs und der jeweils aktuelle wird in `.Random.seed` dokumentiert. Daher führen wiederholte Aufrufe des RNGs, insbesondere in verschiedenen **R**-Sitzungen, zu verschiedenen Zufallszahlen(folgen).

Für die Überprüfung und den Vergleich von Simulationen ist es jedoch notwendig, Folgen von Zufallszahlen reproduzieren (!) zu können. Um dies zu erreichen, kann der Zustand des RNGs von der Benutzerin oder dem Benutzer gewählt werden, wozu die Funktion `set.seed` dient. Wird sie vor dem Aufruf einer der obigen `r....`-Funktionen (und unter Verwendung desselben RNGs) jedesmal mit demselben Argument (einem `integer`-Wert) aufgerufen, so erhält man stets die gleiche Folge an Zufallszahlen. Beispiel:

```
> runif(3)                          # Drei uniforme Zufallszahlen (aus dem
[1] 0.1380366 0.8974895 0.6577632   # RNG mit unbekanntem Startzustand).

> set.seed(42)                      # Wahl eines Startzustandes des RNGs.
> runif(3)                          # Drei weitere uniforme Zufallszahlen.
[1] 0.9148060 0.9370754 0.2861395

> set.seed(42)                      # Wahl *desselben* RNG-Startzustandes
> runif(3)                          # wie eben => Replizierung der drei
[1] 0.9148060 0.9370754 0.2861395   # uniformen Zufallszahlen von eben.

> runif(3)                          # Ausgehend vom akt. Zustand liefert der
[1] 0.8304476 0.6417455 0.5190959   # RNG nun 3 andere uniforme Zufallszahlen.
```

6 Programmieren in R

Schon in "base **R**" (also **R**s „Grundausstattung") gibt es eine Unmenge eingebauter Funktionen, von denen wir bisher nur einen Bruchteil kennengelernt haben. Viele der Funktionen sind durch eine Reihe von Optionen über Argumente im Detail ihrer Arbeitsweise modifizierbar, aber sie sind nichtsdestotrotz vorgefertigte „Konserven". Häufig ist es gewünscht oder nötig, eine solche Funktion mit einer langen Liste von speziellen Optionen oder eine ganze Gruppe von Funktionen in immer derselben Abfolge wiederholt aufzurufen. Auch sind gelegentlich (beispielsweise bei der Simulation statistischer Verfahren) problemspezifische Algorithmen umzusetzen, was keine der eingebauten Funktionen allein kann, sondern nur eine maßgeschneiderte Implementation. Diese Ziele lassen sich durch das Programmieren neuer Funktionen in **R** erreichen.

6.1 Definition neuer Funktionen: Ein Beispiel

Anhand einer Beispielfunktion für die Explorative Datenanalyse (EDA) sollen die Grundlagen der Definition neuer, problemspezifischer Funktionen erläutert werden: Viele inferenzstatistische Verfahren hängen stark von Verteilungsannahmen über die Daten ab, auf die sie angewendet werden sollen. Für die Beurteilung, ob diese Annahmen gültig sind, ist die EDA hilfreich. Sie verwendet grafische Methoden, die wir zum Teil bereits kennengelernt haben, und es kann sinnvoll sein, diese in einer neuen Funktion zusammenzufassen.

Als Beispieldaten verwenden wir Messungen des amerikanischen Physikers A. A. Michelson aus dem Jahr 1879 zur Bestimmung der Lichtgeschwindigkeit in km/s (die folgenden Werte resultieren aus den eigentlichen Geschwindigkeitswerten durch Subtraktion von 299000 km/s):

```
> v <- c( 850, 740, 900, 1070, 930, 850,  950,  980, 980, 880,
+        1000, 980, 930,  650, 760, 810, 1000, 1000, 960, 960)
```

Eine neu definierte Funktion für die explorative Datenanalyse

```
> eda <- function(x) {
+ par(mfrow = c(2,2),
+     cex = 1.2)
+ hist(x)
+ boxplot(x)
+ qa <- diff(quantile(x,
+            c(1/4, 3/4)))
+ dest <- density(x,
+                 width = qa)
+ plot(dest)
+ qqnorm(x)
+ qqline(x)
+ summary(x)
+ }

> eda(v)
 Min. 1st Qu. Median Mean
  650     850    940  909

 3rd Qu. Max.
    980 1070
```

Die Zuweisung `eda <- function(x) {....}` definiert eine Funktion namens `eda`, die bei ihrem Aufruf ein funktionsintern mit `x` bezeichnetes Argument erwartet. Die in `{....}` stehenden Befehle werden beim Aufruf von `eda` sequenziell abgearbeitet:
Erst „formatiert" `par` ein Grafikfenster passend (und öffnet es, wenn noch keins offen war; Details hierzu in Kapitel 7). Als nächstes wird ein Histogramm für die Werte in `x` geplottet (womit klar ist, dass der an `x` übergebene Wert ein `numeric`-Vektor sein muss). Als drittes entsteht ein Boxplot. Dann wird der Quartilsabstand der Daten berechnet und funktionsintern in `qa` gespeichert. Hernach bestimmt `density` einen Kern-Dichteschätzer (per Voreinstellung mit Gaußkern), wobei `qa` für die Bandbreite verwendet wird; das Resultat wird `dest` zugewiesen und dann geplottet. Des Weiteren wird ein Normal-Q-Q-Plot mit Soll-Linie angefertigt. Zuletzt werden arithmetische "summary statistics" berechnet und da dies der letzte Befehl in `eda` ist, wird dessen Ergebnis als Resultatwert von `eda` an die Stelle des Funktionsaufrufes zurückgegeben (wie `eda(v)` zeigt).

Das Resultat der Anwendung der Funktion `eda` auf `v` ist die Ausgabe der "summary statistics" für `v` und als „Nebenwirkung" in einem Grafik-Device die Grafiken in Abb. 21.

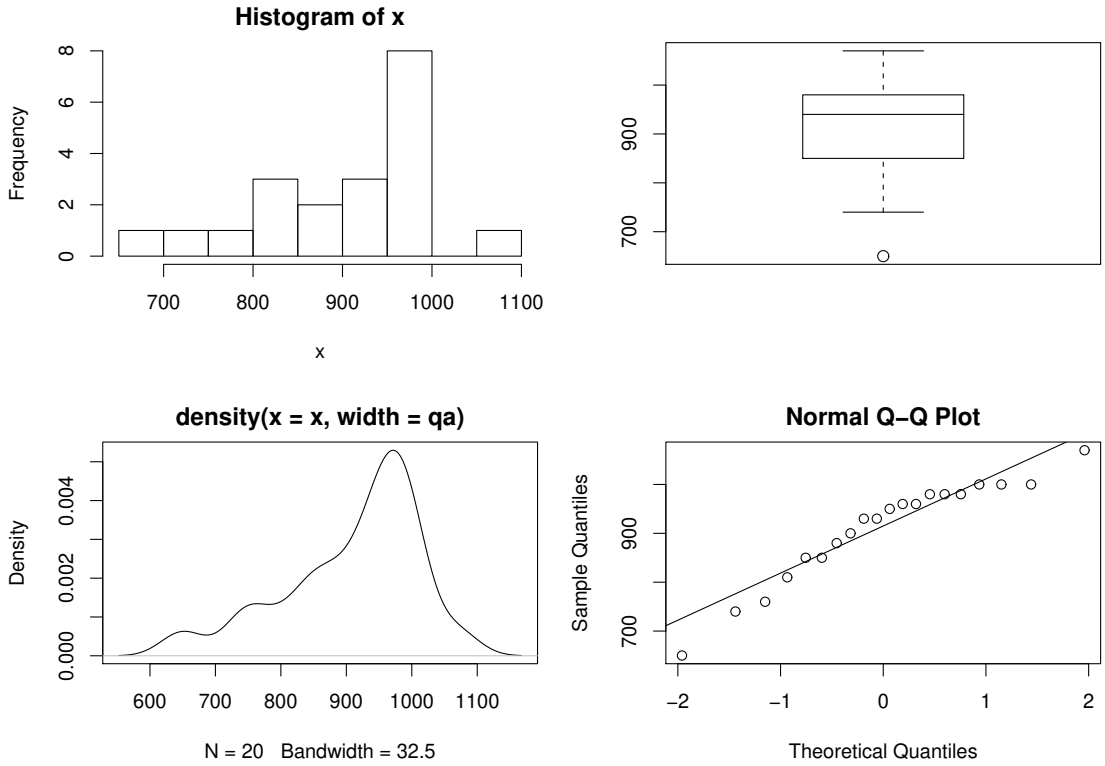

Abbildung 21: Grafisches Ergebnis der Anwendung der Funktion `eda` auf `v`.

6.2 Syntax der Funktionsdefinition

Allgemein lautet die Syntax der Definition einer neuen **R**-Funktion

neuefunktion `<- function(`*argumenteliste*`) {` *funktionsrumpf* `}`

Dabei ist

- *neuefunktion* der (im Prinzip frei wählbare) Name des Objektes, als das die neue Funktion gespeichert wird und das zur Klasse `function` zählt;
- `function` ein reservierter **R**-Ausdruck;
- *argumenteliste* die stets in runde Klammern `()` zu packende, möglicherweise leere Argumenteliste von durch Kommata getrennten Argumenten, die beim Aufruf der Funktion mit Werten versehen und innerhalb der neuen Funktion verwendet werden (können);
- *funktionsrumpf* der Funktionsrumpf, welcher aus einem zulässigen **R**-Ausdruck oder einer Sequenz von zulässigen **R**-Ausdrücken besteht, die durch Semikola oder Zeilenumbrüche getrennt sind. Er ist, wenn er aus mehr als einem Ausdruck besteht, notwendig in geschweifte Klammern `{ }` zu packen; bei nur einem Ausdruck ist die Verwendung von `{ }` zwar nicht nötig, aber dennoch empfehlenswert.

Seit der **R**-Version 4.1.0 existiert in "base **R**" eine Abkürzung für den Ausdruck `function`. Ihr Symbol ist '\'. Der Ausdruck `\(`*argumenteliste*`) {`*funktionsrumpf*`}` wird dabei intern in `function(`*argumenteliste*`) {`*funktionsrumpf*`}` überführt.
Beispiel: `\(x) {x + 1}` bedeutet also `function(x) {x + 1}`.
Diese Abkürzung *kann* helfen, **R**-Code übersichtlicher zu gestalten, insbes. in Fällen, in denen (sehr!) einfache Funktionen zum Einsatz kommen – wie typischerweise bei temporär genutzten, anonymen (= namenlosen) Funktionen. Ich empfehle, sie nur *sehr* sparsam zu verwenden.

6.3 Verfügbarkeit einer Funktion und ihrer lokalen Objekte

Durch die Ausführung obiger **R**-Anweisung wird unter dem Namen *neuefunktion* die *argumenteliste* und die Gesamtheit der in *funktionsrumpf* stehenden **R**-Ausdrücke als **ein** neues Objekt der Klasse `function` zusammengefasst und im aktuellen workspace gespeichert. In unserem Eingangsbeispiel ist dies das `function`-Objekt mit dem Namen `eda`.
Die in *funktionsrumpf* stehenden **R**-Ausdrücke dürfen insbesondere auch Zuweisungsanweisungen sein. Diese Zuweisungsanweisungen haben i. d. R. jedoch nur „lokalen“ Charakter, d. h., sie sind außerhalb der Funktion ohne Effekt und werden nach dem Abarbeiten der Funktion im Allgemeinen wieder vergessen. Mit anderen Worten, jedes innerhalb des Funktionsrumpfs *erzeugte* Objekt (auch wenn sein Name mit dem eines außerhalb der Funktion bereits existierenden Objekts übereinstimmt) hat eine rein temporäre und völlig auf das „Innere“ der Funktion eingeschränkte, eigenständige Existenz. Diese Objekte werden daher auch lokale Variablen oder Objekte genannt. In unserem `eda`-Beispiel sind `qa` und `dest` solche Objekte, die nur während der Abarbeitung der Funktion existieren und auch nur innerhalb des Rumpfes der Funktion bekannt sind.

6.4 Rückgabewert einer Funktion

Jede **R**-Funktion liefert einen Rückgabewert. Er ist das Resultat, d. h. der Wert des im Funktionsrumpf als letztem ausgeführten Ausdrucks, wenn nicht vorher ein Aufruf der Funktion `return` erreicht wird, der die Abarbeitung des Funktionsrumpfes sofort beendet und deren Argument dann der Rückgabewert ist. Sind mehrere Werte/Objekte zurückzugeben, können sie in eine Liste (z. B. mit benannten Komponenten) zusammengefasst werden.

War die Aufrufstelle der Funktion der Prompt der **R**-Console und ist keine andere Vorkehrung getroffen, wird der Rückgabewert der Funktion einfach in der Console angezeigt und vergessen. Soll er *nicht* angezeigt werden, so ist im Funktionsrumpf der Ausdruck, dessen Wert zurückzugeben ist, in die Funktion `invisible` einzupacken. Nichtsdestotrotz wird der Rückgabewert der Funktion an die Aufrufstelle übergeben, nur eben „unsichtbar“, d. h., er kann nach wie vor an ein Objekt zugewiesen und so gespeichert werden.

Beispiel: Im Fall von `eda` in Abschnitt 6.1 ist der Rückgabewert das Ergebnis von `summary(x)`. Obiges bedeutet, dass der Aufruf `eda(v)` bzw. einer jeden neu definierten Funktion auch auf der rechten Seite einer Zuweisungsanweisung stehen darf: `michelson.summary <- eda(v)` ist also eine zulässige **R**-Anweisung, die als Haupteffekt dem Objekt `michelson.summary` den Rückgabewert des Funktionsaufrufs `eda(v)`, also die “summary statistics” für `v` zuweist und als Nebeneffekt die jeweiligen Grafiken erzeugt. Sollte das Ergebnis von `eda`, also `summary(x)`, unsichtbar zurückgegeben werden, müsste der letzte Ausdruck in `eda`s Rumpf `return(invisible(summary(x)))` oder kürzer `invisible(summary(x))` lauten. In diesem Fall hätte die Zuweisungsanweisung `michelson.summary <- eda(v)` also dieselbe Wirkung wie zuvor, aber `eda(v)` allein würde den Rückgabewert einfach „auf Nimmerwiedersehen“ verschwinden lassen.

6.5 Spezifizierung von Funktionsargumenten

Es ist guter und sicherer Programmierstil, möglichst alle Informationen, die innerhalb des Funktionsrumpfs verarbeitet werden sollen, durch die Argumenteliste des Funktionsaufrufs an die Funktion zu übergeben (und nicht von innen auf „außerhalb“ der Funktion vorhandene Objekte zuzugreifen). Dadurch wird die Argumenteliste jedoch schnell zu einem recht langen „Flaschenhals“, sodass eine gut durchdachte Argumenteliste entscheidend zur Flexibilität und Praktikabilität einer Funktion beitragen kann.

Die in der Argumenteliste einer Funktions*definition* auftretenden Argumente sind ihre Formalparameter und die beim Funktions*aufruf* tatsächlich an die Argumenteliste übergebenen Ob-

jekte heißen Aktualparameter. Entsprechend haben Formalparameter Formalnamen und die Aktualparameter Aktualnamen.

6.5.1 Argumente mit default-Werten

Um eine Funktion großer Flexibilität zu konstruieren, ist in ihrer Definition häufig eine umfangreiche Argumenteliste vonnöten, was ihren Aufruf rasch recht aufwändig werden lässt. Oft sind jedoch für mehrere Argumente gewisse Werte als dauerhafte „Voreinstellungen“ wünschenswert, die nur in gelegentlichen Aufrufen durch andere Werte zu ersetzen sind. Um diesem Aspekt Rechnung zu tragen, ist es schon in der Funktionsdefinition möglich, Argumenten Voreinstellungswerte (= “default values”) zuzuweisen, sodass diese bei einem konkreten Aufruf der Funktion nicht in der Argumenteliste des Aufrufs angegeben zu werden brauchen (wenn die voreingestellten Werte verwendet werden sollen). Dazu sind den jeweiligen Formalparametern in der Argumenteliste die gewünschten Voreinstellungswerte mittels Ausdrücken der Art *formalparameter* = *wert* zuzuweisen.

In §4.3.2 hatten wir auf Seite 94 in Abb. 14 das Beispiel eines Streudiagramms für die Werte in zwei Vektoren `X1` und `X5`, in dem die x- und y-Achsenbeschriftung durch die Argumente `xlab` und `ylab` der Funktion `plot` festgelegt wurden. Angenommen, wir sind hauptsächlich an Streudiagrammen im Zusammenhang mit der einfachen linearen Regression interessiert und wollen häufig die Beschriftung „Regressor“ und „Response“ für die x- bzw. y-Achse verwenden. Dann könnten wir uns eine Funktion `rplot` wie folgt definieren:

```
> rplot <- function(x, y, xlabel = "Regressor", ylabel = "Response") {
+ plot(x, y, xlab = xlabel, ylab = ylabel)
+ }
```

In der Definition von `rplot` sind die zwei Argumente `xlabel` und `ylabel` auf die default-Werte „Regressor“ bzw. „Response“ voreingestellt und brauchen deshalb beim Aufruf von `rplot` in seiner Argumenteliste nicht mit Werten versehen zu werden, wenn diese Achsenbeschriftung verwendet werden soll. Wenn jedoch `xlabel` und `ylabel` im Aufruf von `rplot` mit Werten versehen werden, „überschreibt“ dies (temporär!) deren default-Werte. Vergleiche die beiden folgenden Aufrufe und die resultierenden Plots darunter in Abb. 22:

```
> rplot(X1, X5)
```

```
> rplot(X1, X5,
+ xlabel = "Sprosshöhe [mm]",
+ ylabel = "Unterstes Blatt [mm]")
```

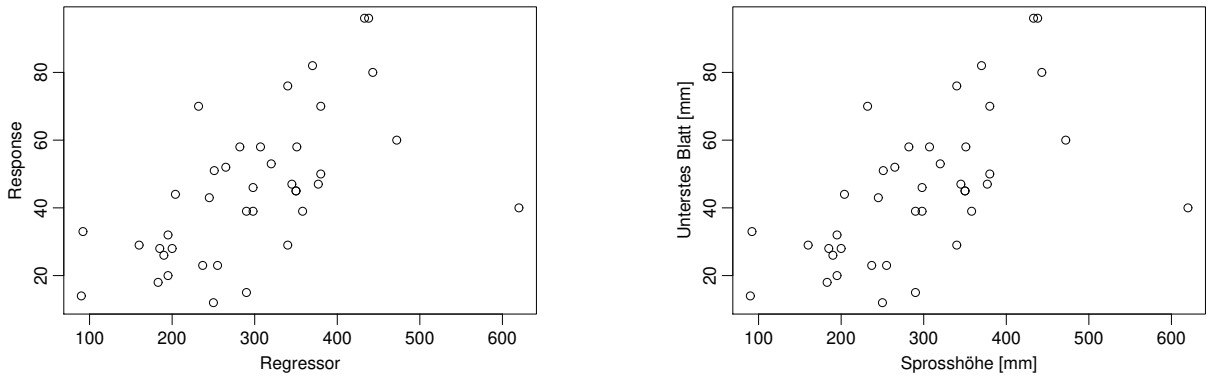

Abbildung 22: Zwei Aufrufe von `rplot`: links mit default-Werten, rechts mit explizit angegebenen.

6.5.2 Variable Argumentezahl: Das „Dreipunkteargument“

Gelegentlich ist es nötig, innerhalb einer neu definierten Funktion wie `rplot` eine weitere Funktion (oder mehrere Funktionen) aufzurufen, an die gewisse Argumente von der Aufrufstelle von `rplot` einfach nur „durchgereicht“ werden sollen, ohne dass sie durch `rplot` verändert oder benutzt werden. Insbesondere kann auch die Anzahl der durchzureichenden Argumente variieren. Um dies zu ermöglichen, muss in der Argumenteliste der Funktionsdefinition von `rplot` der spezielle, „Dreipunkteargument“ (oder “dot-dot-dot”) genannte Formalparameter „`...`“ stehen. Er spezifiziert eine variable Anzahl beliebiger Argumente für diese Funktion (und zwar zusätzlich zu den in der Argumenteliste bereits angegebenen). In einem solchen Fall können dieser Funktion unter Verwendung der Syntax *formalparameter* = *wert* beliebige und beliebig viele Argumente übergeben werden. Innerhalb des Funktionsrumpfes wird der Formalparameter „`...`“ typischerweise nur in den Argumentelisten weiterer Funktionsaufrufe verwendet (siehe aber auch den knappen Hinweis zum Dreipunkteargument auf Seite 109 oben).

In unserem Beispiel `rplot` könnten durch eine solche Erweiterung beispielsweise Argumente an die Funktion `plot` durchgereicht werden, die das Layout des zu zeichnenden Koordinatensystems, die zu verwendenden Linientypen, Plot-Zeichen usw. beeinflussen (vgl. Abb.23):

```
> rplot <- function(x, y, xlabel = "Regressor", ylabel = "Response", ...) {
+ plot(x, y, xlab = xlabel, ylab = ylabel, ...)
+ }
```

```
> rplot(X1, X5,
+ xlabel = "Höhe [mm]",
+ ylabel = "Unterstes Blatt [mm]",
+ xlim = c(0, max(X1)),
+ ylim = c(0, max(X5)),
+ las = 1, bty = "l", pch = 4)
```

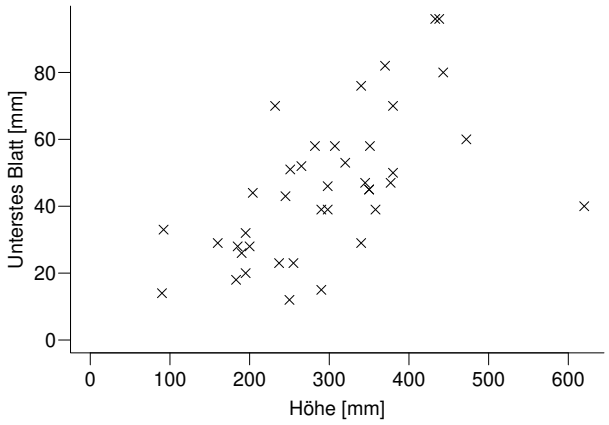

Abbildung 23: Aufruf von `rplot` unter Verwendung des Dreipunkte-Arguments.

In dem obigen Aufruf von `rplot` werden die Aktualparameter `X1`, `X5`, `xlabel` und `ylabel` von `rplot` „abgefangen“, weil die ersten beiden über ihre Position im Funktionsaufruf den beiden ersten Formalparametern `x` und `y` von `rplot` zugewiesen werden und die beiden anderen mit Formalnamen in der Argumenteliste von `rplot` in Übereinstimmung gebracht werden können. Die Argumente `xlim`, `ylim`, `las`, `bty` und `pch` sind in der Argumenteliste von `rplot` nicht aufgeführt und werden daher vom Dreipunkteargument „`...`“ übernommen und im Rumpf von `rplot` unverändert an `plot` weitergereicht. (Zur Bedeutung der Layout-Parameter `xlim` bis `pch` siehe Kapitel 7 und zu den Details der Parameterübergabe den folgenden Abschnitt.)

6.5.3 Zuordung von Aktual- zu Formalparametern beim Funktionsaufruf

Hier zählen wir **R**s wesentliche Regeln und ihre Anwendungsreihenfolge für die Zuordnung von Aktual- zu Formalparametern beim Funktionsaufruf auf, wobei die vier Schritte anhand der Funktion `mean(x, trim = 0, na.rm = FALSE, ...)` und eines Beispielvektors `z <- c(9, 14, NA, 10, 8, 7, 5, 11, 1, 3, 2)` erläutert werden:

1. **Zuordnung über vollständige Formalnamen**: Zunächst wird für jedes Argument der Bauart *argumentname* = *wert*, bei dem *argumentname* vollständig mit einem Formalnamen in der Argumenteliste der Funktion übereinstimmt, *wert* diesem Formalparameter zugewiesen. Die Reihenfolge der Argumente mit vollständigen Formalnamen ist dabei im Funktionsaufruf irrelevant:

```
> mean(na.rm = TRUE, x = z)
[1] 7
```

2. **Zuordnung über unvollständige Formalnamen**: Sind nach Schritt 1 noch Argumente der Art *argumentname* = *wert* übrig, so wird für jedes verbliebene Argument, dessen *argumentname* mit den Anfangszeichen eines noch „unversorgten“ Formalnamens übereinstimmt, *wert* dem entsprechenden Formalparameter zugewiesen. Auch hier spielt die Reihenfolge der Argumente im Funktionsaufruf keine Rolle:

```
> mean(na.rm = TRUE, x = z, tr = 0.1)      > mean(x = z, na = TRUE, tr = 0.1)
[1] 6.875                                  [1] 6.875
```

3. **Zuordnung über Argumentepositionen**: Sind auch nach Schritt 2 noch Argumente übrig, werden die Werte **unbenannter** Argumente von links beginnend, der Reihe nach den noch übrigen „unversorgten“ Formalparametern zugewiesen:

```
> mean(na.rm = TRUE, tr = 0.1, z)          > mean(z, 0.1, TRUE)
[1] 6.875                                  [1] 6.875
```

4. **Zuordnung über das Dreipunkteargument**: Sind nach dem 3. Schritt immer noch Argumente übrig, werden sie dem Dreipunkteargument zugewiesen, falls es in der Funktionsdefinition vorhanden ist, ansonsten gibt es eine Fehlermeldung:

```
> mean(z, na = TRUE, nochwas = "Unsinn")           # Offenbar besitzt mean
[1] 7                                              # das Dreipunkteargument,
> median(z, na = TRUE, nochwas = "Unsinn")         # aber median nicht.
Error in median(z, na = TRUE, nochwas = "Unsinn") :
        unused argument(s) (nochwas ...)
```

   Beachte: Für benannte Argumente *hinter* dem Dreipunkteargument klappt eine Zuordnung über unvollständige Formalnamen nicht, d. h. deren Argumentnamen sind beim Funktionsaufruf nicht abkürzbar.

6.5.4 Nützliches für den Zugriff auf Argumentelisten und Argumente sowie auf den Quellcode existierender Funktionen

Es folgen einige **knappe Hinweise** zur Existenz von Funktionen, die für die „direkte“ Arbeit mit Argumentelisten und Argumenten einer Funktion im Rumpf derselben nützlich sein können. Details sind unbedingt der jeweiligen Hilfeseite zu entnehmen.

- Argumente können einen Vektor als default-Wert haben, dessen Elemente mit Hilfe von `match.arg` auf „Passgenauigkeit“ mit dem beim Funktionsaufruf übergebenen Aktualparameter untersucht werden können, was einerseits sehr flexible und gleichzeitig äußerst knappe Funktionsaufrufe ermöglicht und andererseits die Wertemenge zulässiger Aktualparameter beschränkt und eine „integrierte Eingabefehlerkontrolle“ implementiert:

```
> rplot <- function(x, y, xlabel = c("Regressor", "Unabhängige Variable"),
+                   ylabel = c("Response", "Abhängige Variable"), ...) {
+ xlabel <- match.arg(xlabel)
```

```
+ ylabel <- match.arg(ylabel)
+ plot(x, y, xlab = xlabel, ylab = ylabel, ...)
+ }
```

`match.arg` prüft, ob und wenn ja, zu welchem der default-Werte der tatsächlich übergebene Aktualparameter passt und liefert den (nötigenfalls vervollständigten) Wert zurück.

- `hasArg` und `missing` dienen im Rumpf einer Funktion der Prüfung, ob Formalparameter beim Aufruf jener Funktion mit Aktualparametern versorgt wurden bzw. ob diese fehlten.

  `formals` und `args` liefern die gesamte Argumenteliste einer Funktion in zwei verschiedenen Formaten.

- Zugriff auf den „Inhalt“ des Dreipunktearguments erhält man mittels `list(...)`. Das Ergebnis ist eine Liste mit den in `...` auftauchenden Paaren *argumentname* = *wert* als Komponenten.

  Evtl. recht instruktiv für die Funktionsweise des Dreipunktearguments könnte auch die kleine Aufgabe in einer E-Mail von Bert Gunter vom 30. 1. 2013 auf **R**-help sein, zu finden z. B. unter `https://stat.ethz.ch/pipermail/r-help/2013-January/346460.html`.

  Die bereits in §2.9.3 auf Seite 53 erwähnte Funktion `modifyList` könnte bei der Bearbeitung, sprich Modifikation der Argumenteliste einer selbstgeschriebenen Funktion inkl. ihres Dreipunkteargumentes hilfreich sein. Siehe z. B. die E-Mail von Arun K. am 5. 2. 2014 auf **R**-help, archiviert u. a. unter `https://stat.ethz.ch/pipermail/r-help/2014-February/370837.html`.

- `deparse` und `substitute` in geeigneter Kombination im Rumpf einer Funktion erlauben, an die Aktualnamen der an die Formalparameter übergebenen Aktualparameter heranzukommen, was z. B. genutzt werden kann für die flexible Beschriftung von Grafiken, die „innerhalb“ von benutzereigenen Funktionen angefertigt werden:

  ```
  > myplot <- function(x, y) {
  + plot(x, y, xlab = deparse(substitute(x)), ylab = deparse(substitute(y))) }
  ```

  `deparse(substitute(y))` ermittelt in obigem Fall z. B. die Zeichenkette, die den Objektnamen des an den Formalparameter `y` übergebenen Aktualparameter darstellt.

- Will man sich den Quellcode einer *selbstdefinierten* Funktion ansehen, so erreicht man dies durch Eingabe des Funktionsnamens – ohne das Klammernpaar `()` – am **R**-Prompt (da der Code ja der Wert des `function`-Objektes mit dem betreffenden Namen ist). Für bereits in “base **R**” oder in **R**-Paketen existierende Funktionen ist es gelegentlich jedoch geringfügigst aufwändiger, an ihren Quellcode zu kommen, da sie aus technischen Gründen „versteckt“ sein können. In [68, Ligges (2015)] wird genauer beschrieben, was dahinter steckt und warum dann wie vorzugehen ist. Hier nur ein paar Beispiele:

  – Für die Funktion `sd` zur Berechnung der Standardabweichung ist es ganz einfach. Die Eingabe ihres Namens liefert ihre Definition und eine Zusatzinformation (die hier aus Platzgründen nur unvollständig und leicht editiert abgedruckt werden):

  ```
  > sd
  function (x, na.rm = FALSE) {
     if (is.matrix(x)) {
         msg <- "sd(<matrix>) is deprecated.\n Use apply(*, 2, sd) instead."
         warning(paste(msg, collapse = ""), call. = FALSE, domain = NA)
         apply(x, 2, sd, na.rm = na.rm)
     } else
  ....
  }
  ```

– Bei `mean` geschieht dies eigentlich auch, aber das Ergebnis ist – zunächst – wenig erhellend, denn es sagt uns letztendlich nur, dass es mehrere sogenannte Methoden für die „generische“ Funktion `mean` gibt und eine adäquate aufgerufen wird:

```
> mean
function (x, ...)
UseMethod("mean")
<bytecode: 0x0555f5e0>
<environment: namespace:base>
```

Welche Methoden es gibt, bekommt man durch die Funktion `methods` aufgelistet:

```
> methods(mean)
[1] mean.data.frame   mean.Date    mean.default   mean.difftime
[5] mean.POSIXct      mean.POSIXlt
```

Und die konkrete Abfrage der interessierenden Methode liefert schließlich ihre Definition (die aus Platzgründen hier ebenfalls nicht vollständig abgedruckt wird):

```
> mean.default
function (x, trim = 0, na.rm = FALSE, ...) {
    if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
        warning("argument is not numeric or logical: returning NA")
        return(NA_real_)
    }
....
}
```

– Schwieriger wird's wenn einige der Methoden “invisible” sind, wie z. B. die `ecdf`-Methode der generischen Funktion `summary`, nämlich `summary.ecdf`, denn an sie kommt man nicht durch lediglich die Eingabe ihres Namens:

```
> methods(summary)
 [1] summary.aov             summary.aovlist         summary.aspell*
 [4] summary.connection      summary.data.frame      summary.Date
 [7] summary.default         summary.ecdf*           summary.factor
....
   Non-visible functions are asterisked

> summary.ecdf
Fehler: Objekt 'summary.ecdf' nicht gefunden
```

– Die Online-Hilfe zu `summary.ecdf` zeigt uns jedoch (in der linken oberen Ecke ihrer Hilfeseite), dass diese Methode zum Paket `stats` gehört, sodass das Voranstellen des Paketnamens und zweier oder dreier Doppelpunkte den Zugriff auf den Funktionscode erlaubt:

```
> stats:::summary.ecdf
function (object, ...) {
    header <- paste("Empirical CDF:\t ", environment(object)$n,
        "unique values with summary\n")
....
}
```

6.6 Kontrollstrukturen: Bedingte Anweisungen, Schleifen, Wiederholungen

Die Abarbeitungsreihenfolge von **R**-Ausdrücken in einer Funktion ist sequenziell. Kontrollstrukturen wie bedingte Anweisungen und Schleifen erlauben Abweichungen davon:

Bedingte Anweisungen: `if`, `ifelse`, `switch`	
`> if(`*bedingung* `) {` `+  `*ausdruck1* `+ } else {` `+  `*ausdruck2* `+ }`	Die `if`-Anweisung erlaubt die alternative Auswertung zweier Ausdrücke abhängig vom Wert des Ausdrucks *bedingung*, der ein skalares `logical`-Resultat liefern muss: ist es `TRUE`, wird *ausdruck1* ausgewertet, anderenfalls *ausdruck2*. Der „`else`-Zweig“ (`else{....}`) kann weggelassen werden, wenn für den Fall *bedingung* = `FALSE` keine Aktion benötigt wird. Der Resultatwert der `if`-Anweisung ist der des tatsächlich ausgeführten Ausdrucks.
`> a <- if(n > 10) {` `+  print("Shift 1")` `+  (1:n - 1/2)/n` `+ } else {` `+  print("Shift 2")` `+  (1:n - 3/8)/` `+  (n + 1/4)` `+ }`	Die Bestandteile der gesamten „`if{ }else{ }`“-Struktur und insbesondere ihre vier geschweiften Klammern sollten stets so auf verschiedene Zeilen verteilt werden, wie links gezeigt, um Lesbarkeit und Eindeutigkeit der `if`-Struktur zu gewährleisten. (Handelt es sich bei *ausdruck1* oder *ausdruck2* um einen einzelnen Ausdruck, so können die geschweiften Klammern `{ }` jeweils weggelassen werden, was jedoch fehleranfällig ist.)
`> ifelse(`*test*`,` `+  `*ausdr1*`,` *ausdr2*`)` `> ifelse(` `+  runif(5) > 1/2,` `+  letters[1:5],` `+  LETTERS[1:5])` `[1] "A" "b" "C" "D" "e"`	Die Funktion `ifelse` ist eine vektorisierte Version von `if`, wobei *test* ein `logical`-Objekt ist oder ergibt; dieses gibt auch die Struktur des Resultates vor. *ausdr1* und *ausdr2* müssen vektorwertige Resultate derselben Länge wie *test* liefern (können also schon Vektoren sein). Für jedes Element von *test*, das `TRUE` ist, wird das korrespondierende Element von *ausdr1* in das entsprechende Element des Resultates eingetragen, für jedes Element, das `FALSE` ist, das korrespondierende Element von *ausdr2*.
`> switch(`*ausdr*`,` `+ `*argname.1* `=` *ausdr.1*`,` `+  ....,` `+ `*argname.N* `=` *ausdr.N* `+ )` `> Verteilung <- "...."` `> switch(Verteilung,` `+  normal = rnorm(1),` `+  cauchy = rcauchy(1),` `+  uniform = runif(1),` `+  stop("Unbekannt")` `+  )`	Die Funktion `switch` ist, wie ihr Name nahelegt, ein Schalter, der abhängig vom Wert des (Steuer-)Ausdrucks *ausdr* die Ausführung von höchstens einem von mehreren, alternativen Ausdrücken veranlasst. Der Resultatwert von `switch` ist der Wert des tatsächlich ausgeführten Ausdrucks: *ausdr* muss sich zu `numeric` oder `character` ergeben. Ist das Ergebnis `numeric` mit dem Wert i, wird *ausdr.i* ausgewertet; ist es `character`, so wird derjenige Ausdruck ausgeführt, dessen Formalname (*argname.1* bis *argname.N*) mit dem Inhalt von *ausdr* exakt übereinstimmt. Falls der Wert von *ausdr* nicht zwischen 1 und N liegt oder mit keinem Formalnamen übereinstimmt, wird `NULL` zurückgegeben oder, falls ein letzter unbenannter Ausdruck existiert, dessen Wert: In `switch(`*ausdr*`,` *f1* `=` *a1*`, ....,` *fN* `=` *aN*`, aX)` wäre es dann also `aX`. Beachte: Jeder Ausdruck *ausdr.i* kann ein durch ein Paar geschweifter Klammern (`{....}`) gebildeter Block an Ausdrücken sein (die darin evtl. durch Semikola getrennt sind).

Die `for`-Schleife	
`> for(`*variable* `in` *werte*`) {` `+` *ausdruck* `+ }` `> X <- numeric(100)` `> e <- rnorm(99)` `> alpha <- 1.01` `> for(i in 1:99) {` `+ X[i+1] <- alpha *` `+ X[i] + e[i]` `+ }`	Die `for`-Schleife weist *variable* sukzessive die Elemente in *werte* zu und führt dabei *ausdruck* jedes Mal genau einmal aus. *variable* muss ein zulässiger Variablenname sein, `in` ist ein notwendiges Schlüsselwort, *werte* ein Ausdruck, der ein `vector`-Objekt liefert, und *ausdruck* ein beliebiger Ausdruck (oder eine Sequenz von Ausdrücken). Innerhalb von *ausdruck* steht *variable* zur Verfügung, braucht aber nicht verwendet zu werden. Der Rückgabewert der gesamten `for`-Schleife ist der Wert von *ausdruck* in ihrem letzten Durchlauf. Hier wird als intendierter Nebeneffekt der vorbereitete Vektor `X` elementweise mit neuen Werten „gefüllt“.
Vielfach wiederholte Ausdruckauswertung mit `replicate`	
`> replicate(n,` *ausdruck*`)` `> replicate(10, {` `+ x <- rnorm(30)` `+ y <- rnorm(30)` `+ plot(x, y)` `+ } )`	Für $\texttt{n} \in \mathbb{N}$ wird *ausdruck* `n`-mal ausgewertet und die `n` Ergebnisse in einer möglichst einfachen Struktur (z. B. als Vektor) zusammengefasst zurückgegeben. `replicate` ist faktisch nur eine “wrapper”-Funktion für eine typische Anwendung von `sapply` (vgl. S. 53) und insbesondere geeignet für Simulationen, in denen wiederholt Stichproben von Zufallszahlen zu generieren und jedes Mal auf dieselbe Art und Weise zu verarbeiten sind.

Hinweise: Für detailliertere Informationen und weitere Anwendungsbeispiele zu obigen Kontrollstrukturen bzw. auch zu anderen Schleifentypen (`repeat` und `while`) sowie Kontrollbefehlen für Schleifen (`next` und `break`) siehe die Hilfeseiten. Beachte: Um vom **R**-Prompt aus die Hilfeseite zu `if` oder zur `for`-Schleife zu bekommen, müssen Anführungszeichen verwendet werden: `?"if"` bzw. `?"for"`.
Weitere nützliche Funktionen, die die Effizienz der Nutzung von Kontrollstrukturen noch erheblich steigern können, sind z. B. `stopifnot`, `all.equal` und `identical` sowie `stop` und `warning`. Für ihre Eigenschaften verweisen wir auf die zugehörigen Hilfeseiten.

Warnungen, Tipps & weitere – knappe – Hinweise:

- **Warnung:** `for`-Schleifen können in **R** *ziemlich ineffizient* und daher relativ langsam sein. Für gewisse, insbesondere iterative Berechnungen sind sie zwar nicht zu umgehen, sollten aber, wann immer möglich, vermieden und durch eine vektorisierte Version des auszuführenden Algorithmus ersetzt werden!

  Für unvermeidbar hoch-iterative (oder auch rekursive) Algorithmen ist in Erwägung zu ziehen, sie in C oder Fortran zu programmieren und die C- bzw. Fortran-Schnittstelle von **R** zu nutzen. Siehe hierzu die unten empfohlenen Bücher sowie das Paket `Rcpp` oder im Hilfesystem in der Dokumentation “Writing R Extensions” den Abschnitt 6 “System and foreign language interfaces” (vgl. §1.5.3, Bild 2).

  Was Sie *auf jeden Fall* beachten sollten, weil es anderenfalls zu extrem ineffizientem Code führen kann und schlicht als schlechter **R**-Programmierstil gilt, ist das Folgende:

  – Lassen Sie nie Berechnungen in einer Schleife `for(i in ....) {y[i] <- ....}` durchführen, falls diese Berechnungen vektorisierbar sind!

    Es lohnt sich oft, etwas länger über eine Vektorisierung nachzudenken als einfach mit einer `for`-Schleife „drauflos zu hacken“.

  - Lassen Sie nie Objekte (Vektoren, Matrizen, Listen etc.) durch oder in eine/r `for`-Schleife „wachsen“, d. h. dynamisch entstehen wie z. B. in `for(i in ....) {y <- c(y, ....)}`! Dasselbe gilt hierbei für `cbind` oder `rbind` anstelle von `c`.

    Selbst wenn man nicht weiß, wie groß das resultierende Objekt schließlich genau sein wird, man aber immerhin eine obere Schranke kennt, so lohnt es sich oft, diese Schranke zu nutzen, indem man ein zu großes Objekt kreiert, durch geeignete Indizierung (nur teilweise) füllt und das erhaltene Objekt schließlich auf die endgültig benötigte Dimension verkleinert.

- Um aus dem Inneren einer Schleife während ihrer Iterationen etwas auszudrucken (wie z. B. den jeweiligen Wert der Laufvariable oder Zwischenergebnisse), bedarf es der expliziten Verwendung des `print`- oder `cat`-Befehles. Bsp.:

```
> set.seed(2022)
> X <- numeric(100);    e <- rnorm(99);   alpha <- 1.01
> for(i in 1:99) {
+  cat("X[", i+1, "] = ", sep = "")
+  X[i+1] <- alpha * X[i] + e[i]
+  cat(X[i+1], "\n")
+  }
X[2] = 0.900142
X[3] = -0.2642024
X[4] = -1.16433
....
```

- Einige sehr nützliche Hilfsfunktionen, um Eigenschaften von Programmcode wie Schnelligkeit und Objektgrößen zu kontrollieren bzw. um die Fehlersuche zu erleichtern, sind die folgenden:

  - `system.time` ermittelt die CPU-Zeit (und andere Zeiten), die zum Abarbeiten eines an sie übergebenen **R**-Ausdrucks benötigt wurde.
  - `object.size` ergibt den ungefähren Speicherplatzbedarf in Bytes des an sie übergebenen Objektes.
  - `browse` erleichtert die Fehlersuche z. B. im Rumpf einer Funktion, indem sie erlaubt, die Auswertungsumgebung der Stelle, an der sie aufgerufen wurde, zu inspizieren.

    `debug` ermöglicht noch mehr, z. B. das „schrittweise“ Abarbeiten der Ausdrücke in einem Funktionsrumpf und (ebenfalls) das zwischenzeitliche Inspizieren sowie nötigenfalls sogar das Modifizieren der dabei verwendeten oder erzeugten Objekte.

    Für die Funktionsweise und Nutzung von `browse` und `debug` siehe jeweils ihre Hilfeseite.
  - `Rprof` erlaubt das zeitlich diskrete, aber relativ engmaschige “profiling” (eine Art Protokollieren) der Abarbeitung von **R**-Ausdrücken. Siehe die zugehörige Hilfeseite.
  - Das Paket `microbenchmark` stellt sowohl eine präzise Zeitmessung (für kleinere Code-Stücke) zur Verfügung als auch einfache Möglichkeiten des tabellarischen und grafischen Vergleichs der Laufzeiten mehrerer **R**-Ausdrücke.

- Gute und nützliche, einführende, aber auch schon tiefgehende Bücher zur Programmierung in **R** bzw. **S** sind „Programmieren mit R“ ([68, Ligges (2015)]) und “S Programming” ([96, Venables & Ripley (2004)]).

  Noch tiefer- und weitergehend sind “Software for Data Analysis: Programming with R” ([17, Chambers (2008)]), das exzellente “The Art of R Programming” ([75, Matloff (2011)]) und das ebensolche “Advanced R” ([104, Wickham (2019)] mit seiner stets aktualisierten Online-Version unter `https://adv-r.hadley.nz` auf `https://bookdown.org`).

  Eine Schnittstelle für die (nahezu) nahtlose Integration von C++-Funktionen in R-Code bietet das Paket `Rcpp`, exzellent und ausführlichst beschrieben in [30, Eddelbuettel (2013)].

6.7 Verkettung von Funktionen und der "simple forward pipe operator"

Unter einer Verkettung (oder Komposition) zweier Funktionen f und g wird ihre Hintereinanderausführung in dem Sinne verstanden, dass das Ergebnis der Ausführung von g als Eingabe für f dient. Diese Verknüpfung ließe sich in der Form $y := g(x)$ gefolgt von $f(y)$ umsetzen, wird aber i. d. R. durch $f(g(x))$ (oder gelegentlich $(f \circ g)(x)$) dargestellt.

Das Konzept lässt sich zum einen in offensichtlicher Form auf eine beliebige endliche Anzahl an Funktionen $f_1, f_2, f_3, \ldots, f_n$ erweitern und lautet dann $f_n(\ldots f_3(f_2(f_1(x)))\ldots)$ bzw. $y_1 := f_1(x), y_2 := f_2(y_1), y_3 := f_3(y_2), \ldots, y_n := f_n(y_{n-1})$. Zum anderen kann es aber sogar sein, dass die Funktionen nicht nur das Ergebnis ihrer jeweiligen direkten Vorgängerin als (einzige) Eingabe benötigen, sondern zusätzliche Argumente besitzen, die mit Eingaben „gefüttert" werden können oder sogar müssen. All dies führt schnell zu umfangreichen und unübersichtlichen Ausdrücken, was sich natürlich bei ihrer Programmierung genauso auswirkt, insbes. wenn die verwendeten Funktionen auch noch längere Namen haben.

Seit der **R**-Version 4.1.0 existiert in "base **R**" ein sogenannter "simple forward pipe operator", kurz ein "(forward) pipe", um derartige „Lindwürmer" von Funktionsverkettungen optisch zu „entzerren". Sein Symbol ist `|>` (also das Paar aus dem senkrechten Strich `|` und dem „Größerzeichen" `>`). Er dient der syntaktischen Abkürzung einer gewissen Komposition von Funktionen, indem er seine linke Seite zum ersten (und auch nur zum ersten!) Argument des Ausdrucks auf seiner rechten Seite transferiert. (Für den Ausdruck auf der rechten Seite gelten gewisse Einschränkungen.) Dadurch erlaubt er, vor allem tiefer verschachtelte Verkettungen von Funktionsaufrufen übersichtlicher als sequenzielle Folge von Verarbeitungsschritten zu schreiben und so auch leichter nachzuvollziehen.

Beispiele: Für ein bereits existierendes `Z` (wie z. B. nach `Z <- 1:10`) liefert

```
> apply(matrix(rep(Z, each = length(Z) - 1), nrow = length(Z)), 2, rev)
```

dasselbe wie

```
> Z |> rep(each = length(Z) - 1) |>
+  matrix(nrow = length(Z)) |>
+  apply(2, rev)
```

Oder:

```
> smsa <- matrix(scan(file = "SMSA", skip = 4), ncol = 12, byrow = TRUE)
```

hat denselben Effekt wie

```
> smsa <- scan(file = "SMSA", skip = 4) |> matrix(ncol = 12, byrow = TRUE)
```

Die Hilfeseite des pipe-Operators (in RStudio über den Suchbegriff "pipeOP" zu finden; am **R**-Prompt durch `?"|>"` – beachte die Hochkommata!) liefert klärende technische Details und in ihrem "Examples"-Abschnitt auch weitere anschauliche Anwendungsbeispiele.

7 Weiteres zur elementaren Grafik

Die leistungsfähige Grafik von **R** zählt zu seinen wichtigsten Eigenschaften. Um sie zu nutzen, bedarf es eines speziellen Fensters bzw. einer speziellen Ausgabedatei zur Darstellung der grafischen Ausgabe. Wie bereits gesehen, gibt es viele Grafikfunktionen, die gewisse Voreinstellungen besitzen, welche es erlauben, ohne großen Aufwand aufschlussreiche Plots zu erzeugen. Grundlegende Grafikfunktionen, wie sie im Folgenden beschrieben werden, ermöglichen dem/der BenutzerIn die Anfertigung sehr spezieller Plots unter Verwendung verschiedenster Layoutfunktionen und Grafikparameter. Für die *interaktive* Nutzung der Grafikausgabe stehen ebenfalls (mindestens) zwei Funktionen zur Verfügung.

7.1 Grafikausgabe

Zur Vollständigkeit sind hier nochmal die in 4.1 aufgeführten Funktionen wiederholt. (Beachte auch die dortigen Hinweise.)

Öffnen und Schließen von Grafik-Devices:	
`> X11` `> windows` `> quartz` .... (Grafikbefehle)	`X11` öffnet unter Unix, `windows` nur unter Windows oder `quartz` nur beim Mac bei jedem Aufruf ein neues Grafikfenster. Das zuletzt geöffnete Device ist das jeweils aktuell *aktive*, in welches die Grafikausgabe erfolgt.
`> dev.list`	Listet Nummern und Typen aller geöffneten Grafik-Devices auf.
`> dev.off`	Schließt das zuletzt geöffnete Grafik-Device *ordnungsgemäß*.
`> graphics.off`	Schließt alle geöffneten Grafik-Devices auf einen Schlag.
`> postscript(file)` .... (Grafikbefehle)	Öffnet bei jedem Aufruf eine neue (EPS- (= "Encapsulated PostScript"-) kompatible) PostScript-Datei mit dem als Zeichenkette an `file` übergebenen Namen. Das zuletzt geöffnete Device ist das jeweils aktuelle, in welches die Grafikausgabe erfolgt.
`> dev.off`	Schließt das zuletzt geöffnete Grafik-Device und *muss* bei einer PostScript-Datei unbedingt zur Fertigstellung verwendet werden, denn erst danach ist sie vollständig und korrekt interpretierbar.
`> pdf(file)` .... (Grafikbefehle) .... `> dev.off`	Völlig analog zu `postscript`, aber eben für PDF-Grafikdateien.

7.2 Elementare Zeichenfunktionen: `plot`, `points`, `lines` & Co.

Eine Funktion zur „Erzeugung" eines Streudiagramms samt Koordinatensystem (= „Kosy"):

Koordinatensysteme & Streudiagramme	
`> x <- runif(50)` `> y <- 2*x + rnorm(50)` `> z <- runif(50)` `> X11`	(Synthetische Beispieldaten einer einfachen linearen Regression $y_i = 2x_i + \varepsilon_i$ mit ε_i i.i.d. $\sim \mathcal{N}(0,1)$, weiterer z_i i.i.d. $\sim \mathcal{U}(0,1)$ und Öffnen eines neuen Grafikfensters.)
`> plot(x, y)` `> plist <- list(x = x, y = y)` `> plot(plist)` `> pmat <- cbind(x, y)` `> plot(pmat)`	Erstellt ein Kosy und zeichnet („plottet") zu den `numeric`-Vektoren `x` und `y` das Streudiagramm der Punkte (`x[i], y[i]`) für i=1, ..., `length(x)`. Statt `x` und `y` kann eine Liste übergeben werden, deren Komponenten `x` und `y` heißen, oder eine zweispaltige Matrix, deren erste Spalte die x- und deren zweite Spalte die y-Koordinaten enthalten.

`> DF <- data.frame(y, x, z)` `> plot(y ~ x + z, data = DF)`	`plot` akzeptiert auch eine Formel (vgl. `boxplot` und `stripchart` in §4.2.2 auf S. 86): `y ~ x1 + .... + xk` bedeutet, dass die „linke“ `numeric`-Variable auf der Ordinate abgetragen wird und die „rechte(n)“ `numeric`-Variable(n) entlang der Abszisse eines jeweils eigenen Kosys. Quelle der Variablen ist der Data Frame für `data`.
`> plot(y)` `> plot(y ~ 1, data = DF)`	Für einen Vektor allein oder die Formel `y ~ 1` wird das Streudiagramm der Punkte (`i, y[i]`) gezeichnet.

Funktionen, mit denen einfache grafische Elemente in ein *bestehendes* Koordinatensystem eingezeichnet werden können:

Punkte, Linien, Pfeile, Text & Gitter in ein Koordinatensystem einzeichnen	
`> points(x, y)` `> points(plist)` `> points(pmat)` `> points(x)`	In ein *bestehendes* (durch `plot` erzeugtes) Kosy werden für die `numeric`-Vektoren `x` und `y` an den Koordinaten (`x[i], y[i]`) Punkte eingetragen. Ebenso können (wie bei `plot`) eine Liste (mit Komponenten namens `x` und `y`) oder eine zweispaltige Matrix verwendet werden. Für alleiniges `x` werden die Punkte an (`i, x[i]`) eingezeichnet.
`> lines(x, y)` `> lines(plist)` `> lines(pmat)` `> lines(x)`	Zeichnet einen Polygonzug durch die Punkte (`x[i], y[i]`) in ein *existierendes* Kosy. Eine *x*-*y*-Liste, eine zweispaltige Matrix oder ein alleiniges `x` funktionieren analog zu `points`.
`> abline(a, b)` `> abline(a)` `> abline(h = const)` `> abline(v = const)`	Zeichnet eine Gerade mit y-Achsenabschnitt `a` und Steigung `b` in ein *bestehendes* Kosy ein. Bei alleiniger Angabe eines *zwei*elementigen Vektors `a` wird `a[1]` der y-Achsenabschnitt und `a[2]` die Steigung. Alleinige Angabe von `h = const` liefert horizontale Geraden mit den Elementen des `numeric`-Vektors `const` als Ordinaten; `v = const` führt zu vertikalen Geraden mit den Abszissen aus `const`.
`> segments(x0, y0,` `+ x1, y1)` `> arrows(x0, y0,` `+ x1, y1)`	Fügt für die gleich langen `numeric`-Vektoren `x0`, `y0`, `x1` und `y1` in ein *vorhandenes* Kosy für jedes i=1, ..., `length(x0)` eine Strecke von (`x0[i], y0[i]`) bis (`x1[i], y1[i]`) ein. `arrows` wirkt wie `segments`; zusätzlich wird an jedem Endpunkt (`x1[i], y1[i]`) eine Pfeilspitze gezeichnet. Durch `length` wird die Größe (in Zoll) des bzw. der zu zeichnenden Pfeilspitzen gesteuert; die Voreinstellung ist mit 0.25 Zoll ziemlich groß. (Eine Angabe `code = 3` liefert Pfeilspitzen an beiden Enden.)
`> text(x, y, labels,` `+ adj, pos)`	Für die `numeric`-Vektoren `x` und `y` und den `character`-Vektor `labels` wird der Text `labels[i]` (per Voreinstellung horizontal und vertikal zentriert) an die Stelle (`x[i], y[i]`) in ein *bestehendes* Kosy geschrieben. Mit `adj` oder `pos` lässt sich die Ausrichtung der Textes relativ zur Stelle (`x[i], y[i]`) variieren. Voreinstellung für `labels` ist `1:length(x)`, d. h., an den Punkt (`x[i], y[i]`) wird `i` geschrieben.
`> grid(nx, ny)`	Zeichnet ein (`nx` × `ny`)-Koordinatengitter in einen bestehenden Plot ein, das sich an seiner Achseneinteilung orientiert.

Bemerkung: Alle obigen Funktionen habe zahlreiche weitere Argumente. Eine große Auswahl davon wird im folgenden Abschnitt beschrieben, aber nicht alle; speziell interessant für `points` ist `pch` und für `lines` sind es `lty` und `col` auf Seite 118. Die Hilfeseite der einzelnen Funktionen ist – wie immer – eine wertvolle Informationsquelle.

7.3 Die Layoutfunktion par und Grafikparameter für plot, par et al.

Viele Grafikfunktionen, die einen Plot erzeugen, besitzen optionale Argumente, die bereits beim Funktionsaufruf die Spezifizierung einiger Layoutelemente (wie Überschrift, Untertitel, Legende) erlauben. Darüber hinaus stehen bei Plotfunktionen, die ein Koordinatensystem erzeugen, Argumente zur Verfügung, mit denen man dessen Layout spezifisch zu steuern vermag (Achsentypen, -beschriftung und -skalen, Plottyp). Weitere Grafikparameter werden durch die Funktion `par` gesetzt und steuern grundlegende Charakteristika aller nachfolgenden Plots. Sie behalten so lange ihre (neue) Einstellung bis sie explizit mit `par` wieder geändert oder beim Aufruf eines neuen Grafik-Devices (Grafikfenster oder -datei) automatisch für diese Grafikausgabe auf ihre Voreinstellung zurückgesetzt werden. Zusätzlich gibt es einige „eigenständige“ Layoutfunktionen (wie `title`, `mtext`, `legend`), die zu einem bestehenden Plot (bereits erwähnte) Layoutelemente auch nachträglich hinzuzufügen erlauben. Die folgende Auflistung ist *nicht* vollständig!

Seitenlayoutparameter der Funktion `par`	
`mfrow = c(m, n)`	`mfrow = c(m,n)` teilt eine Grafikseite in $m \cdot n$ Plotrahmen ein, die in m Zeilen und n Spalten angeordnet sind. Dieser Mehrfachplotrahmen wird links oben beginnend *zeilenweise* mit Plots gefüllt.
`mfcol = c(m, n)`	Wie `mfrow`, nur wird der Mehrfachplotrahmen *spaltenweise* gefüllt.
`oma = c(u, l, o, r)`	Spezifiziert die maximale Textzeilenzahl für den unteren (`u`), linken (`l`), oberen (`o`) und rechten (`r`) „äußeren“ Seitenrand eines Mehrfachplotrahmens. Voreinstellung: `oma = c(0,0,0,0)`. Für die Beschriftung dieser Ränder siehe `mtext` und `title` in Abschnitt 7.4.
`mar = c(u, l, o, r)`	Bestimmt die Breite der Seitenränder (unten, links, oben und rechts) eines einzelnen Plotrahmens in Zeilen ausgehend von der Box um den Plot. Voreinstellung: `mar = c(5,4,4,2)+0.1`, was etwas „verschwenderisch“ ist.
`pty = "c"`	Setzt den aktuellen Plotrahmentyp fest. Mögliche Werte für `c` sind `s` für einen quadratischen Plotrahmen und `m` für einen maximalen, typischerweise rechteckigen Plotrahmen (Voreinstellung).

Hinweise:

- Das “cheat sheet” “How big is your graph?”, auch hier zu finden oder direkt bei `https://raw.githubusercontent.com/rstudio/cheatsheets/main/how-big-is-your-graph.pdf`, hilft, den Überblick und weitere Einblicke in die Steuerung des klassischen Seitenlayouts zu bekommen.
- Flexiblere und komplexere Einteilungen von Grafik-Devices erlaubt u. a. die Funktion `layout`, die jedoch inkompatibel mit vielen anderen Layoutfunktionen ist; siehe ihre Hilfeseite und die dortigen Beispiele.

Koordinatensystemparameter der Funktionen `plot` **oder** `par`	
`axes = L`	Logischer Wert; `axes = FALSE` unterdrückt jegliche Achsenkonstruktion und -beschriftung. Voreinstellung: `TRUE`. (Siehe auch `axis` in Abschnitt 7.4.)
`xlim = c(x1, x2)` `ylim = c(y1, y2)`	Erlaubt die Wahl der Achsenskalen: `x1, x2, y1, y2` sind approximative Minima und Maxima der Skalen, die für eine „schöne“ Achseneinteilung automatisch gerundet werden.
`log = "c"`	Definiert eine logarithmische Achsenskaleneinteilung: `log = "xy"`: doppelt-logarithmisches Achsensystem; `log = "x"`: logarithmierte x- und „normale“ y-Achse; `log = "y"`: logarithmierte y- und „normale“ x-Achse.

`lab = c(x, y, len)`	Kontrolliert die Anzahl der Achseneinteilungsstriche ("ticks") und ihre Länge (jeweils approximativ): `x`: Anzahl der tick-Intervalle der x-Achse; `y`: Anzahl der tick-Intervalle der y-Achse; (`len`: tick-Beschriftungsgröße beider Achsen. *Funktionslos!*) Voreinstellung: `lab = c(5,5,7)`.
`tcl = x`	Länge der ticks als Bruchteil der Höhe einer Textzeile. Ist `x` negativ (positiv), werden die ticks außerhalb (innerhalb) des Kosys gezeichnet; Voreinstellung: -0.5.
`las = n`	Orientierung der Achsenskalenbeschriftung: `las = 0`: parallel zur jeweiligen Achse (Voreinstellung); `las = 1`: waagrecht; `las = 2`: senkrecht zur jeweiligen Achse; `las = 3`: senkrecht.
`mgp = c(x1, x2, x3)`	Abstand der Achsenbeschriftung (`x1`), der -skalenbeschriftung (`x2`) und der -linie (`x3`) vom Zeichenbereich in relativen Einheiten der Schriftgröße im Plotrand. Voreinstellung: `mgp = c(3,1,0)`. Größere Werte führen zur Positionierung weiter vom Zeichenbereich entfernt, negative zu einer innerhalb des Zeichenbereichs.

Linien-, Symbol-, Textparameter für `par`, `plot` & Co.	
`type = "c"`	Spezifiziert als `plot`-Argument den Plottyp. Mögliche Werte für `c` sind `p`: Punkte allein; `l`: Linien allein; `b`: beides, wobei die Linien die Punkte aussparen; `o`: beides, aber überlagert; `h`: vertikale "high-density"-Linien; `s`, `S`: Stufenplots (zwei Arten Treppenfunktionen); `n`: „nichts“ außer ein leeres Koordinatensystem.
`pch = "c"` `pch = n`	Verwendetes Zeichen `c` bzw. verwendeter Zeichencode `n` für zu plottende Punkte. Voreinstellung: `o`. Andere Symbole können der Hilfeseite zu `points` entnommen werden (am besten durch `example(points)`).
`lty = n` `lty = "text"`	Linientyp (geräteabhängig) als `integer`-Code `n` oder als `character`-Wert. Normalerweise ergibt `lty = 1` (Voreinstellung) eine durchgezogene Linie und für höhere Werte erhält man gestrichelte und/oder gepunktete Linien. Für `text` sind englische Ausdrücke zugelassen, die den Linientyp beschreiben wie z. B. `solid` oder `dotted`. Details: Hilfeseite zu `par`.
`col = n` `col = "text"` `col.axis` `col.lab` `col.main` `col.sub`	Linienfarbe als `integer`-Code oder `character`: `col = 1` ist die Voreinstellung (i. d. R. schwarz); 0 ist stets die Hintergrundfarbe. Hier sind englische Farbnamen zugelassen, z. B. `red` oder `blue`. Die Parameter `col.axis`, `col.lab`, `col.main`, `col.sub` beziehen sich speziell auf die Achsenskalenbeschriftung, die Achsenbeschriftung, die Überschrift bzw. den Untertitel, wie durch `title` erzeugt werden (siehe Abschnitt 7.4).
`cex = x` `cex.axis` `cex.lab` `cex.main` `cex.sub`	Zeichen-„Expansion“ relativ zur Standardschriftgröße des verwendeten Gerätes: Falls $x > 1$, ein Vergrößerungs-, falls $x < 1$, ein Verkleinerungsfaktor. Für `cex.axis`, `cex.lab`, `cex.main` und `cex.sub` gilt dasselbe wie für `col.axis` usw. Details zu `col` und `cex`: Hilfeseite zu `par`.
`adj = x`	Ausrichtung von (z. B. durch `text`, `title` oder `mtext`, siehe Abschnitt 7.4) geplottetem Text: `adj = 0`: linksbündig, `adj = 0.5`: horizontal zentriert (Voreinstellung), `adj = 1`: rechtsbündig. Zwischenwerte führen zur entsprechenden Ausrichtung zwischen den Extremen.

Beschriftungsparameter der Funktion `plot`	
`main = "text"`	Plotüberschrift `text`; 1.5-fach gegenüber der aktuellen Standardschriftgröße vergrößert. Voreinstellung: `NULL`.
`main = "text1\ntext2"`	`\n` erzwingt einen Zeilenumbruch und erzeugt (hier) eine zweizeilige Plotüberschrift. (Analog für das Folgende.)
`sub = "text"`	Untertitel `text`, unterhalb der x-Achse. Voreinstellung: `NULL`.
`xlab = "text"` `ylab = "text"`	x- bzw. y-Achsenbeschriftung (`text`). Voreinstellung ist jeweils der Name der im Aufruf von `plot` verwendeten Variablen.

7.4 Achsen, Überschriften, Untertitel und Legenden

Wurde eine Grafik angefertigt, ohne schon beim Aufruf von `plot` Achsen, eine Überschrift oder einen Untertitel zu erstellen (z. B. weil `plot`'s Argumenteliste `axes = FALSE, ylab = "", xlab = ""` enthielt und weder `main` noch `sub` mit einem Wert versehen wurden), kann dies mit den folgenden Funktionen nachgeholt und bedarfsweise auch noch genauer gesteuert werden, als es via `plot` möglich ist. Darüberhinaus lässt sich eine Grafik um eine Legende ergänzen.

„Eigenständige" Plotbeschriftungsfunktionen	
`> axis(side,` `+ at = NULL,` `+ labels = TRUE,` `+ tick = TRUE,` `+ ....)`	Fügt zu einem bestehenden Plot eine Achse an der durch `side` spezifizierten Seite (1 = unten, 2 = links, 3 = oben, 4 = rechts) hinzu, die ticks an den durch `at` angebbaren Stellen hat (`Inf`, `NaN`, `NA` werden dabei ignoriert); in der Voreinstellung `at = NULL` werden die tick-Positionen intern berechnet. `labels = TRUE` (Voreinstellung) liefert die übliche numerische Beschriftung der ticks (auch ohne eine Angabe für `at`). Für `labels` kann aber auch ein `character`- oder `expression`-Vektor mit an den ticks zu platzierenden Labels angegeben werden; dann muss jedoch `at` mit einem Vektor derselben Länge versorgt sein. `tick` gibt an, ob überhaupt ticks und eine Achsenlinie gezeichnet werden; in der Voreinstellung geschieht dies.
`> title(main,` `+ sub,` `+ xlab,` `+ ylab,` `+ line,` `+ outer = FALSE)`	Fügt zu einem bestehenden Plot (zentriert in den Rändern des aktuellen Plotrahmens) die Überschrift `main` (1.5-mal größer als die aktuelle Standardschriftgröße) und den Untertitel `sub` unterhalb des Kosys sowie die Achsenbeschriftungen `xlab` und `ylab` hinzu. Der voreingestellte Zeilenabstand ist durch `line` änderbar, wobei positive Werte den Text weiter vom Plot entfernt positionieren und negative Werte näher am Plot. Falls `outer = TRUE`, landen die Texte in den Seiten-Außenrändern.
`> mtext(text,` `+ side = 3,` `+ line = 0,` `+ outer = FALSE)`	Schreibt den Text `text` in einen der Seitenränder des aktuellen Plotrahmens. Dies geschieht in den Seitenrand `side`: 1 = unten, 2 = links, 3 = oben (Voreinstellung), 4 = rechts. Dort wird in die Zeile `line` (Bedeutung wie bei `title`) geschrieben. Soll der Text in den (äußeren) Rand eines *Mehrfach*plotrahmens geplottet werden (also beispielsweise als Überschrift für einen Mehrfachplotrahmen), muss `outer = TRUE` angegeben werden.
`> legend(x = 0,` `+ y = NULL,` `+ legend,` `+ lty, pch,` `+ ncol = 1)`	In den bestehenden Plot wird eine Legende eingezeichnet, deren linke obere Ecke sich an einer durch `x` und `y` definierten Position befindet und worin die Elemente des `character`-Vektors `legend` auftauchen, gegebenenfalls kombiniert mit den durch `lty` und/oder `pch` spezifizierten Linientypen bzw. Plotsymbolen. (Die nächste Seite und Abschnitt 7.5 zeigen Beispiele.) Mit `ncol` wird die (auf 1 voreingestellte) Spaltenzahl variiert. Weitere grafische Parameter sind angebbar; siehe die Hilfeseite.

`> legend("top",` `+ legend, ....)`	Anstelle numerischer Werte für `x` und `y` kann an `x` eines der Worte `"bottomright"`, `"bottom"`, `"bottomleft"`, `"left"`, `"topleft"`, `"top"`, `"topright"`, `"right"` oder `"center"` als Ort der Legende (im Inneren des Kosys) übergeben werden.

Die Grafik in Abb. 24 zeigt das Beispiel einer Seite mit einem (2×2)-Mehrfachplotrahmen und verschiedensten Layouteinstellungen, die durch die darunter folgenden **R**-Kommandos erzeugt wurde.

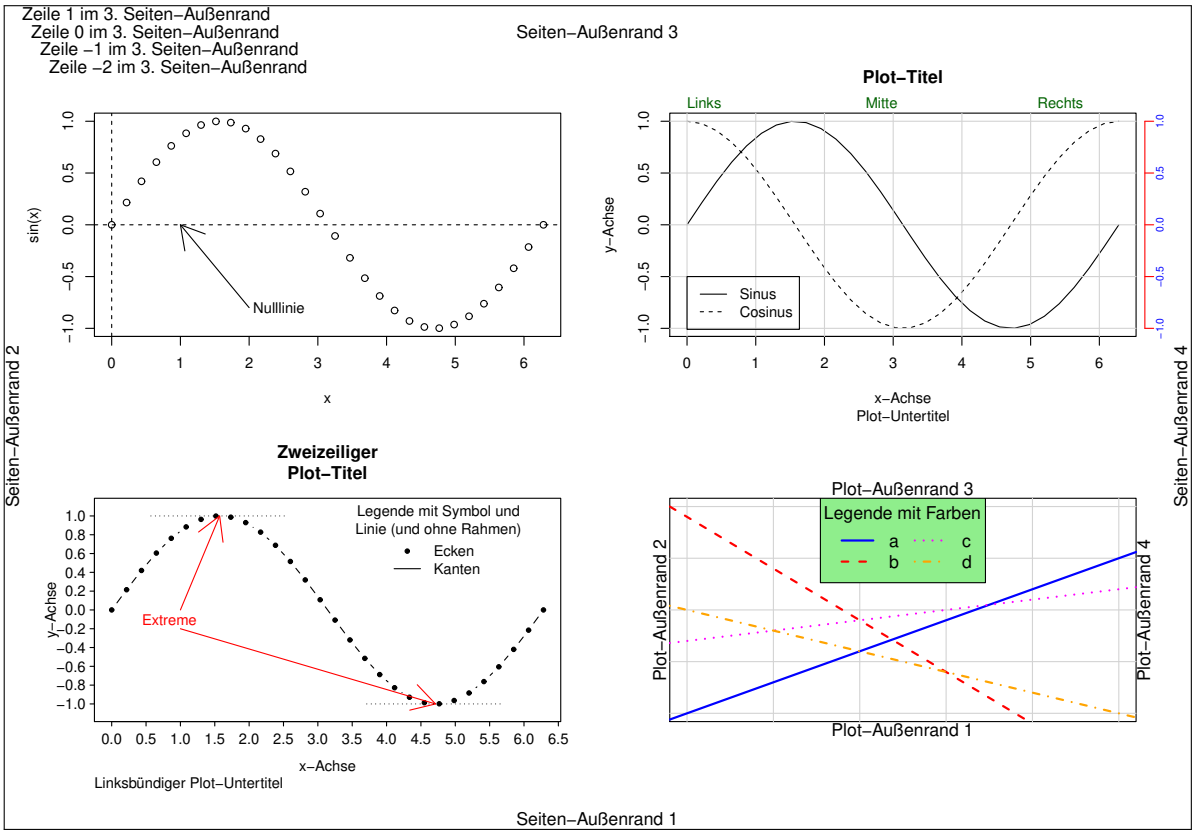

Abbildung 24: Verschiedene Layout-Effekte in einem (2×2)-Mehrfachplotrahmen.

```
# 30 äquidistante Stützstellen von 0 bis 2*pi als Beispieldaten und
# Formatieren eines Grafikfensters als (2 x 2)-Mehrfachplotrahmen:
#*********************************************************************
x <- seq(0, 2*pi, length = 30)
par(mfrow = c(2, 2), oma = c(1, 1, 2, 1), mar = c(5, 4, 4, 2) + 0.1)

# Plot links oben:
#*****************
plot(x, sin(x));        abline(v = 0, lty = 2);     abline(h = 0, lty = "dashed")
arrows(2, -0.8, 1, 0);   text(2, -0.8, "Nulllinie", pos = 4, offset = 0.2)

# Plot rechts oben:
#******************
plot(x, sin(x), type = "l", main = "Plot-Titel", sub = "Plot-Untertitel",
     xlab = "x-Achse", ylab = "y-Achse")
lines(x, cos(x), lty = "dashed");     grid(lty = "solid")
legend(0, -0.5, legend = c("Sinus", "Cosinus"), lty = c("solid", "dashed"))

axis(side = 4, line = 0.5, cex.axis = 0.75, col = "red", col.axis = "blue",
     mgp = c(3, 0.25, 0))
```

```
axis(side = 3, at = c(0, 2.6, 5.1), labels = c("Links", "Mitte", "Rechts"),
     tick = FALSE, mgp = c(3, 0.1, 0), hadj = 0, col.axis = "darkgreen")

# Plot links unten:
#******************
par(mgp = c(2, 0.5, 0))
plot(x, sin(x), type = "b", pch = 20, las = 1, lab = c(10, 10, 7), tcl = -0.25,
     ylim = c(-1.1, 1.1), xlab = "x-Achse", ylab = "y-Achse")

text(1, -0.1, "Extreme", adj = 0.7, col = "red")
arrows(c(1, 1), c(0, -0.2), c(pi/2, 3*pi/2), c(1, -1), col = "red")
segments(c(pi/2, 3*pi/2) - 1, c(1, -1),
         c(pi/2, 3*pi/2) + 1, c(1, -1), lty = "dotted")
legend(3.5, 1, legend = c("Ecken", "Kanten"), lty = c(-1, 1), pch = c(20, -1),
       bty = "n", title = "Legende mit Symbol und\nLinie (und ohne Rahmen)")

title(main = "Zweizeiliger\nPlot-Titel", cex = 0.75)
title(sub = "Linksbündiger Plot-Untertitel", adj = 0, cex = 0.6)

# Plot rechts unten:
#*******************
par(mgp = c(3, 1, 0))
plot(c(0, 5), c(-1, 1), type = "n", axes = FALSE, xlab = "", ylab = "");    box()
grid(lty = "solid")

abline(-1,    0.3, lty = 1, lwd = 2, col = "blue")
abline( 0.9, -1/2, lty = 2, lwd = 2, col = "red")
abline(-0.3,  0.1, lty = 3, lwd = 2, col = "magenta")
abline( 0,   -0.2, lty = 4, lwd = 2, col = "orange")

legend("top", legend = letters[ 1:4], lty = 1:4, lwd = 2,
       col = c("blue", "red", "magenta", "orange"), cex = 1.2, ncol = 2,
       bg = "lightgreen", title = "Legende mit Farben")

mtext("Plot-Außenrand 1", side = 1);   mtext("Plot-Außenrand 2", side = 2)
mtext("Plot-Außenrand 3", side = 3);   mtext("Plot-Außenrand 4", side = 4)

# Text im äußeren Rahmen der Seite:
#**********************************
mtext(paste("Seiten-Außenrand", 1:4), side = 1:4, outer = TRUE)   # mtext mit
mtext(paste("Zeile", 1:-2, "im 3. Seiten-Außenrand"),             # vektoriellen
      outer = TRUE, line = 1:-2, adj = 0:3/100)                   # Argumenten.

# Schließen der Grafikausgabe:
#*****************************
dev.off()
```

7.5 Einige (auch mathematisch) nützliche Plotfunktionen

In mathematisch-statistischen Anwendungen sind mit gewisser Häufigkeit die Graphen stetiger Funktionen, einseitig stetiger Treppenfunktionen oder geschlossener Polygonzüge (mit gefärbtem Inneren) zu zeichnen. Hierfür präsentieren wir einige **R**-Plotfunktionen im Rahmen von Anwendungsbeispielen (und verweisen für Details auf deren Hilfeseiten):

7.5.1 Stetige Funktionen: `curve`

```
> curve(sin(x)/x, from = -20, to = 20, n = 500)
> curve(abs(1/x), add = TRUE, col = "red", lty = 2)
```

(Plot links oben in Abb. 25.) Das erste Argument von `curve` ist der Ausdruck eines Aufrufs einer (eingebauten oder selbstdefinierten) Funktion. Die Argumente `from` und `to` spezifizieren den Bereich, in dem die Funktion ausgewertet und gezeichnet wird. `n` (Voreinstellung: 101) gibt an, wie viele äquidistante Stützstellen dabei zu nutzen sind. Ist `add = TRUE`, wird in ein *bestehendes* Koordinatensystem gezeichnet, wobei dessen x-Achsenausschnitt verwendet wird (und dies, wenn nicht anders spezifiziert, mit `n = 101` Stützstellen). Wie stets, bestimmen `col`, `lty` und `lwd` Linienfarbe, -typ bzw. -dicke.

7.5.2 Geschlossener Polygonzug: `polygon`

```
> x <- c(1:7, 6:1);   y <- c(1, 10, 8, 5, 1, -1, 3:9)  # Koordinaten der Ecken.
> plot(x, y, type = "n")                              # Generiert das (leere) Kosy.
> polygon(x, y, col = "orange", border = "red")              # Der Polygonzug.
> text(x, y)                                       # Nur zur Beschriftung der Ecken.
```

(Plot rechts oben in Abb. 25.) `polygon` zeichnet ein Polygon in ein *bestehendes* Koordinatensystem. Seine ersten beiden Argumente `x` und `y` müssen die Koordinaten (`x[i], y[i]`) der Polygonecken enthalten, die durch Kanten zu verbinden sind, wobei die letzte Ecke automatisch mit der ersten (`x[1], y[1]`) verbunden wird. Innen- und Randfarbe werden durch `col` bzw. `border` festgelegt.

7.5.3 Beliebige Treppenfunktionen: `plot` in Verbindung mit `stepfun`

```
> x <- c(0, 1, 3, 9)
> y <- c(1, -1, 2, 4, 3)
> plot(stepfun(x, y), verticals = FALSE)
```

(Linker Plot in zweiter „Zeile“ in Abb. 25.) Für einen aufsteigend sortierten Vektor `x` von Sprungstellen und einen *um ein Element längeren* Vektor `y` von Funktionswerten *zwischen* den Sprungstellen generiert `stepfun(x, y)` ein `stepfun`-Objekt, das eine Treppenfunktion repräsentiert, die links von der ersten Sprungstelle `x[1]` auf dem Niveau `y[1]` startet und an `x[i]` auf `y[i+1]` springt und die per Voreinstellung eine sogenannte cadlag-Funktion ist, d. h. eine rechtsseitig stetige Funktion mit Grenzenwerten von links (franz.: „continue à droite, limite à gauche“, kurz cadlag). Sie wird von `plot` „mathematisch korrekt“ gezeichnet, falls das `plot`-Argument `verticals = FALSE` ist.

7.5.4 Die empirische Verteilungsfunktion: `plot` in Verbindung mit `ecdf`

```
> plot(ecdf(rnorm(12)))    # Empirische Verteilungsfunktion zu 12 standard-
                           # normalverteilten Pseudozufallsvariablen.
```

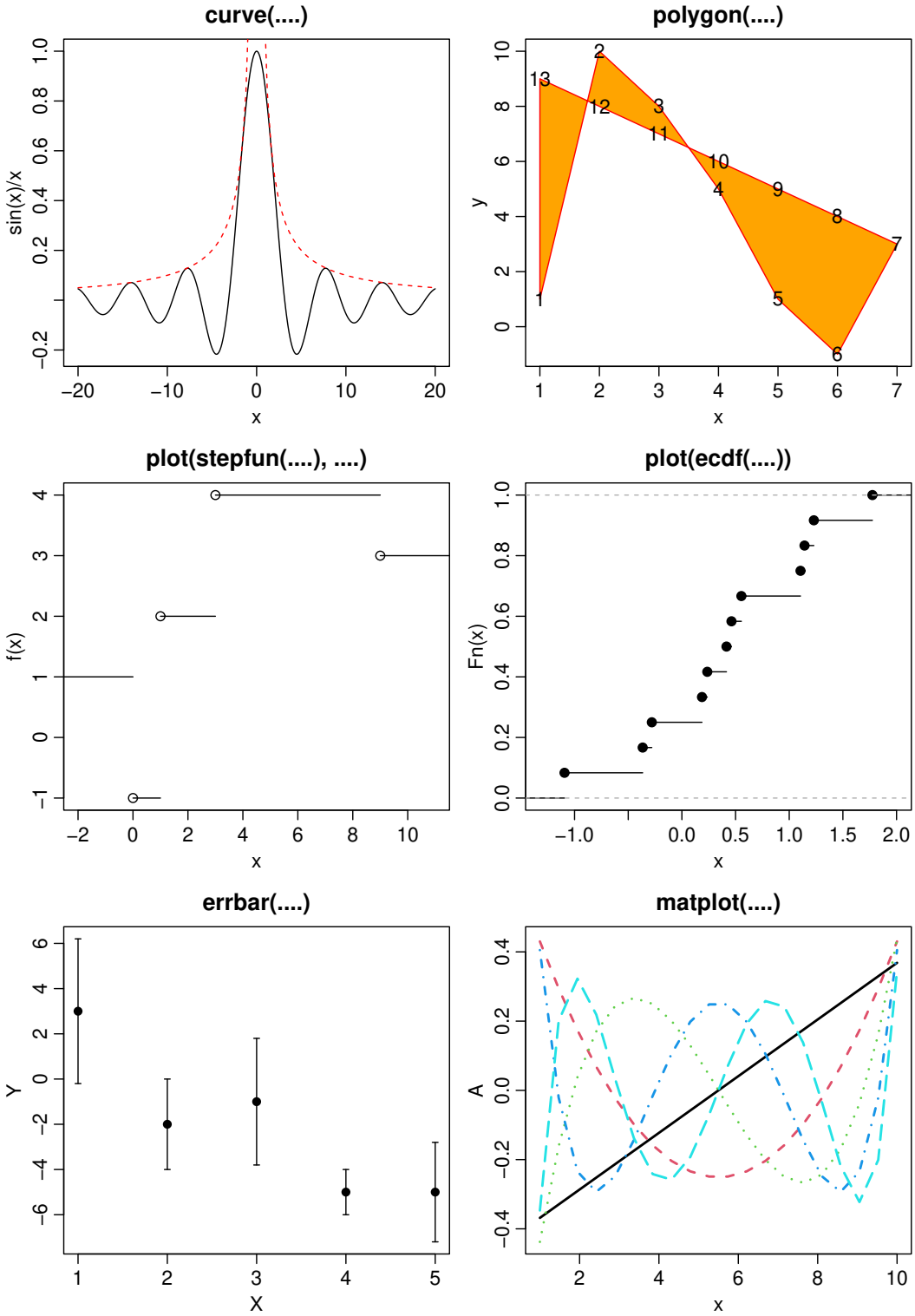

Abbildung 25: Einige auch mathematisch nützliche Plotfunktionen.

(Rechter Plot in zweiter „Zeile“ in Abb. 25.) Die empirische Verteilungsfunktion F_n zu n Werten ist eine (cadlag-)Treppenfunktion, die auf dem Niveau 0 startet und an der i-ten OS auf den Funktionswert i/n springt. Zu einem beliebigen `numeric`-Vektor generiert `ecdf` ein `stepfun`-Objekt, das diese speziellen Eigenschaften hat und von `plot` „mathematisch korrekt“ gezeichnet wird, da hierzu automatisch die `plot`-Methode `plot.stepfun` verwendet wird.

7.5.5 „Fehlerbalken“: `errbar` im Package `Hmisc`

```
> X <- 1:5;   Y <- c(3, -2, -1, -5, -5)
> delta <- c(3.2, 2, 2.8, 1, 2.2)
> library(Hmisc)
> errbar(x = X, y = Y, yplus = Y + delta, yminus = Y - delta)
> detach(package:Hmisc)
```

(Plot ganz links unten in Abb. 25.) Mit `library(Hmisc)` wird das Package `Hmisc` in den Suchpfad eingefügt, damit die Funktion `errbar` zur Verfügung steht. `errbar` plottet an den für `x` angegebenen Stellen vertikale „Fehlerbalken“. Die Werte für `y` (hier also die Werte in `Y`) werden markiert und die (absoluten) Unter- und Obergrenzen der vertikalen Balken werden von `yminus` bzw. `yplus` erwartet.

Beachte: Die Argumente `main` und `sub` sind in `errbar` entgegen der Aussage auf ihrer Hilfeseite-Seite dysfunktional.

7.5.6 Mehrere Polygonzüge „auf einmal“: `matplot`

`matplot(A, B)` trägt jede Spalte einer n-zeiligen Matrix `B` gegen jede Spalte einer anderen, ebenfalls n-zeiligen Matrix `A` als Polygonzug in ein gemeinsames Koordinatensystem ab, wobei jede Spaltenkombination einen eigenen Linientyp erhält. Bei ungleicher Spaltenzahl wird (spalten-)zyklisch repliziert; also kann jede der Matrizen auch ein n-elementiger Vektor sein.

Beispiel: `B` sei eine $(n \times k)$-Matrix, die in ihren $j = 1, \ldots, k$ Spalten die Funktionswerte $p_j(x_i)$ gewisser Orthonormalpolynome $p_1, \ldots, p_k$ an Stützstellen x_i für $i = 1, \ldots, n$ enthält:

```
> x <- seq(1, 10, length = 20)
> (B <- poly(x, 5))     # Gewisse Orthonormalpolynome bis zum Grad 5

                1           2           3           4           5
 [1,] -0.36839420  0.43019174 -0.43760939  0.40514757 -0.34694765
 [2,] -0.32961586  0.29434172 -0.16122451 -0.02132356  0.20086443
 [3,] -0.29083753  0.17358614  0.03838679 -0.23455912  0.32260045
....
[20,]  0.36839420  0.43019174  0.43760939  0.40514757  0.34694765
```

Durch

```
> matplot(x, B, type = "l")
```

wird eine Grafik erzeugt, in der für jede Spalte $j = 1, \ldots, k$ ein (farbiger) Polygonzug durch die Punkte (`x[i]`, `B[i, j]`), $i = 1, \ldots, n$, gezeichnet ist, wobei die Polygonzüge automatisch verschiedene Linientypen bekommen. (Plot in Abb. 25 ganz unten rechts.)

7.6 Interaktion mit Plots

R erlaubt dem/der BenutzerIn unter Verwendung der Maus mit einem in einem Grafikfenster bestehenden Plot zu interagieren: Durch „Anklicken“ mit der Maus können geplottete Punkte in einem Koordinatensystem (Kosy) identifiziert oder Informationen an beliebigen, „angeklickten“ Positionen in einen Plot eingefügt werden.

Interaktion mit Plots	
`> identify(x, y)` `[1]  4  7 13 19 24`	In den in einem Grafikfenster *bestehenden* Plot von `y` gegen `x` wird nach jedem Klick mit der linken Maustaste die *Indexnummer* des dem Mauszeiger nächstliegenden Punktes in dessen Nähe in das bestehende Kosy platziert. Ein Klick der mittleren Maustaste beendet die Funktion `identify`. Rückgabewert: Vektor der Indizes der „angeklickten“ Punkte (hier 5 Stück).
`> identify(x, y,` `+ labels = c("a",` `+ "b", ....))`	Wird `labels` ein `character`-Vektor (derselben Länge wie `x`) zugewiesen, so wird das dem „angeklickten“ Punkt entsprechende Element von `labels` in dessen Nähe gezeichnet.
`> identify(x, y,` `+ plot = FALSE)`	Für `plot = FALSE` entfällt das Zeichnen. (`identify` gibt aber stets die Indices der angeklickten Punkte zurück.)
`> locator()` `$x` `[1] 0.349  4.541` `$y` `[1] -0.291 0.630`	Für einen in einem Grafikfenster bestehenden Plot wird eine Liste von x-y-Koordinaten der mit der linken Maustaste „angeklickten“ Stellen geliefert (hier 2 Stück; Voreinstellung: Maximal 500). Mit einem Klick der mittleren Maustaste wird die Funktion beendet. (Kann z. B. zur Positionierung einer Legende genutzt werden; siehe unten.)
`> locator(n = 10,` `+ type = "c")`	Das Argument `n` bestimmt die Höchstzahl der identifizierbaren Punkte. `type` gibt an, ob bzw. was an den „angeklickten“ Koordinaten geplottet werden soll. Mögliche Werte für `c` sind `p`: Punkte allein; `l`: Die Punkte verbindende Linien (also ein Polygonzug); `o`: Punkte mit verbindenden Linien überlagert; `n`: „Nichts“ (Voreinstellung).

Anwendungsbeispiele für die Interaktion mit Plots	
`> pos <- locator(1)` `> legend(pos,` `+ legend, ....)`	`pos` speichert die in einem bestehenden Plot in einem Grafikfenster angeklickte Stelle. Dann wird dort die durch `legend` spezifizierte Legende positioniert.
`> text(locator,` `+ labels)`	An den durch `locator` in einem bestehenden Plot in einem Grafikfenster spezifizierten Positionen werden die in `labels` angegebenen Elemente (zyklisch durchlaufend) gezeichnet. Allerdings geschieht das Zeichnen erst, *nachdem* durch einen Klick der mittleren Maustaste die Funktion `locator` beendet worden ist.
`> mies <- identify(x,` `+ y, plot = FALSE)` `> xgut <- x[-mies]` `> ygut <- y[-mies]`	Die Indexnummern der unter den (`x[i], y[i]`) durch Anklicken mit der Maus identifizierten Punkte werden in `mies` gespeichert, damit diese „miesen“ Punkte danach aus `x` und `y` entfernt werden können.

Hinweise:

- Wörtliche Wiederholung der Bemerkungen von Seite 98 unten: Ein interessantes sowie weit- und tiefgehendes Buch zum Thema Grafik in **R** ist “R Graphics” ([78, Murrell (2005)] bzw. seine zweite Auflage [79, Murrell (2011)]). Dasselbe gilt für das exzellente Buch “Lattice. Multivariate Data Visualization with R” ([85, Sarkar (2008)]), das die **R**-Implementation und -Erweiterung des hervorragenden “Trellis-Grafik”-Systems im **R**-Paket `lattice` vorstellt. Hierin ist für multivariate Daten insbes. die Darstellung mit

Hilfe sog. “parallel coordinate plots” (durch die Funktion `parallelplot`) zu erwähnen. Ebenfalls interessant und höchst leistungsfähig ist das **R**-Paket `ggplot2`, das in ([102, Wickham (2009)]) beschrieben wird.

- In der bereits in Abschnitt 1.1 erwähnten “R Graph Gallery” sind zahlreiche, mit **R** angefertigte, exzellente Grafiken (samt des sie produzierenden **R**-Codes) zu finden; siehe `https://www.r-graph-gallery.com`.

- **R** stellt auch eine LaTeX-ähnliche Funktionalität für die Erstellung mathematischer Beschriftung von Grafiken zur Verfügung. Die Hilfeseite dazu erhält man via `?plotmath`. `example(plotmath)` und `demo(plotmath)` liefern einen Überblick über das, was damit möglich ist, und wie es erreicht werden kann. Warnung: Die Syntax ist etwas „gewöhnungsbedürftig“.

- Für erheblich leistungsfähigere sowie vor allem interaktive und dynamische Grafikfunktionalität als die, die wir hier kennengelernt haben, stehen z. B. die **R**-Packages `rgl` und `rggobi` zur Verfügung. Sie besitzen auch eigene Websites, zu denen man z. B. aus der Package-Sammlung von **R** kommt: `www://cran.r-project.org/` → “Packages” → `rgl` oder `rggobi` und dort dann der angegebenen URL folgen.

  Als Beispiel für eine interaktive (mit Hilfe der Maus drehbare) Grafik mit der Funktion `plot3d` des Paketes `rgl` führen Sie das Folgende aus (nachdem Sie, falls noch nicht geschehen, die Pakete `lattice` (für die Beispieldaten) und `rgl` installiert haben):

```
> data(ethanol, package = "lattice")
> library(rgl)
> plot3d(ethanol, col = "red", type = "s", size = 1)
```

  Bewegen Sie dann den Maus-Cursor in das neu geöffnete RGL-Grafikfenster, halten Sie die linke Maustaste geklickt und bewegen Sie die Maus, um die 3D-Grafik beliebig zu rotieren.

8 Zur para- und nicht-parametrischen Inferenzstatistik in Ein- und Zweistichprobenproblemen für metrische Daten

8.1 Auffrischung des Konzepts statistischer Tests

Die Inferenzstatistik dient ganz allgemein dem Ziel, Aussagen über unbekannte Parameter einer Grundgesamtheit (= Population) auf der Basis von Daten, d. h. einer Stichprobe zu machen. Hin und wieder liegen über die Werte von Parametern Vermutungen oder Annahmen vor; man spricht von „Hypothesen“. Über die Richtigkeit solcher Hypothesen entscheiden zu helfen ist der Zweck statistischer Tests. Die statistische Testtheorie stellt eine Verbindung zwischen Stichprobe und Population, also zwischen dem Experiment als kleinem Ausschnitt aus der Wirklichkeit und der „Gesamtwirklichkeit“ her: Sie beschreibt, wie sich Stichproben wahrscheinlichkeitstheoretisch „verhalten“, wenn eine gewisse Hypothese über die jeweilige Population zutrifft, und liefert Kriterien, die zu entscheiden erlauben, ob die Hypothese angesichts der tatsächlich beobachteten Stichprobe beizubehalten oder zu verwerfen ist.

8.1.1 Motivation anhand eines Beispiels

Aufgrund langjähriger Erfahrung sei bekannt, dass die Körpergewichte (in Gramm) von ausgewachsenen, männlichen, weißen Labormäusen (vgl. z. B. `de.wikipedia.org/wiki/Hausmaus` → „Äußere Merkmale“) einer gewissen Zuchtlinie unter natürlichen Bedingungen normalverteilt sind mit dem Erwartungswert $\mu_0 = 49$ g und der Standardabweichung $\sigma_0 = 4.2$ g. Für das Körpergewicht X einer zufällig ausgewählten solchen Maus gilt also: $X \sim \mathcal{N}(\mu_0, \sigma_0^2)$.

Es wird vermutet, dass ein gewisser Umweltreiz das Körpergewicht beeinflusst. Allerdings sei bekannt, dass auch unter dem Einfluss dieses Umweltreizes das Körpergewicht eine normalverteilte Zufallsvariable – sagen wir Y – mit Varianz σ_0^2 ist. Jedoch ist ihr Erwartungswert μ unbekannt und möglicherweise verschieden von μ_0. Kurz: $Y \sim \mathcal{N}(\mu, \sigma_0^2)$.

Es soll geklärt werden, ob $\mu = \mu_0$ ist oder nicht. Zu diesem Zweck werde in einem Experiment eine Gruppe von n zufällig ausgewählten Mäusen ähnlicher physischer Konstitution dem besagten Umweltreiz unter sonst gleichen Bedingungen ausgesetzt. (Die Mäuse seien nicht aus demselben Wurf, um auszuschließen, dass „aus Versehen“ z. B. eine außergewöhnlich schwache oder starke Familie untersucht wird.) Die nach einer vorher festgelegten Zeitspanne beobachtbaren zufälligen Körpergewichte $Y_1, \ldots, Y_n$ der Mäuse sind also $\mathcal{N}(\mu, \sigma_0^2)$-verteilt mit einem unbekannten μ. Aber gemäß der Erwartungstreue des arithmetischen Mittels gilt $\mathbb{E}[\bar{Y}_n] = \mu$, sodass für konkrete Realisierungen $y_1, \ldots, y_n$ der zufälligen Körpergewichte erwartet werden kann, dass $\bar{y}_n \approx \mu$ ist (egal welchen Wert μ hat). Für den Fall, dass der Umweltreiz *keinen* Einfluss auf das mittlere Körpergewicht hat, ist $\mu = \mu_0$ und demzufolge auch $\bar{y}_n \approx \mu_0$.

Es werden nun $n = 50$ Realisierungen $y_1, \ldots, y_n$ mit einem Mittelwert $\bar{y}_n = 47.46$ ermittelt und es stellt sich die Frage, ob der beobachtete Unterschied zwischen $\bar{y}_n = 47.46$ und $\mu_0 = 49$, also die Abweichung $|\bar{y}_n - \mu_0| = 1.54$

- eine im Rahmen des Szenarios $Y_1, \ldots, Y_n \sim \mathcal{N}(\mu_0, \sigma_0^2)$ gewöhnlich große, auf reinen *Zufalls*schwankungen beruhende Abweichung ist, oder ob sie
- so groß ist, dass sie eine ungewöhnliche Abweichung darstellt, also gegen $\mu = \mu_0$ spricht und vielmehr das Szenario $\mu \neq \mu_0$ stützt.

8.1.2 Null- & Alternativhypothese, Fehler 1. & 2. Art

Zur Beantwortung der eben gestellten Frage kann ein statistischer Test verwendet werden. Dazu formulieren wir das Problem mittels zweier (sich gegenseitig ausschließender) Hypothesen:

Die Nullhypothese $H_0 : \mu = \mu_0$ und die Alternative $H_1 : \mu \neq \mu_0$

stehen für die möglichen Verteilungsszenarien der Y_i in obigem Beispiel. Dabei wird hier durch die Alternative H_1 nicht spezifiziert, in welche Richtung μ von μ_0 abweicht, geschweige denn, welchen Wert μ stattdessen hat.

Auf der Basis einer Stichprobe soll nach einem (noch zu entwickelnden) Verfahren entschieden werden, ob H_0 abzulehnen ist oder nicht. (Der Name Nullhypothese stammt aus dem Englischen: "to nullify" = entkräften, widerlegen.) Dabei müssen wir uns bewusst sein, dass statistische Tests, also Tests auf der Basis zufälliger Daten, nur *statistische* Aussagen über den „Wahrheitsgehalt" von Hypothesen machen können. Testentscheidungen bergen also immer die Gefahr von Fehlern und wir unterscheiden hierbei zwei Arten:

- **Der Fehler 1. Art:**

  Bei einem Test von H_0 gegen H_1 kann es passieren, dass eine „unglückliche" Stichprobenzusammenstellung, die allein durch natürliche Zufallseinflüsse entstand, uns veranlasst, die Nullhypothese *fälschlicherweise zu verwerfen* (d. h., H_0 zu verwerfen, obwohl H_0 in Wirklichkeit richtig ist). Ein solcher Irrtum heißt Fehler 1. Art. Die Testtheorie stellt Mittel zur Verfügung, die Wahrscheinlichkeit für einen solchen Fehler zu kontrollieren – auszuschließen ist er so gut wie nie! Jedoch kann man sich die (Irrtums-)Wahrscheinlichkeit $\alpha \in (0, 1)$ für den Fehler 1. Art beliebig (aber üblicherweise klein) vorgeben. Sie wird das Signifikanzniveau des Tests genannt.

  Häufig verwendete Werte für α sind 0.1, 0.05 und 0.01, aber an und für sich ist α frei wählbar; es quantifiziert die subjektive Bereitschaft, im vorliegenden Sachverhalt eine Fehlentscheidung 1. Art zu treffen. Also sind auch andere als die genannten Niveaus zulässig, wie z. B. 0.025 oder 0.2, aber je kleiner α ist, umso „zuverlässiger" ist eine Ablehnung von H_0, *wenn* es dazu kommt.

- **Der Fehler 2. Art:**

  Er wird gemacht, wenn man die Nullhypothese *fälschlicherweise beibehält* (d. h., H_0 beibehält, obwohl H_0 in Wirklichkeit falsch ist). Die Wahrscheinlichkeit für diesen Fehler 2. Art wird mit β bezeichnet. Im Gegensatz zu α, das man sich vorgibt, ist β *nicht* so ohne Weiteres kontrollierbar, sondern hängt von α, der Alternative, der den Daten zugrunde liegenden Varianz und dem Stichprobenumfang n ab. (Näheres hierzu in Abschnitt **??** „Testgüte und Fallzahlschätzung für Lokationsprobleme im Normalverteilungsmodell".)

Wir geben eine tabellarische Übersicht über die vier möglichen Kombinationen von unbekannter Hypothesenwirklichkeit und bekannter Testentscheidung:

		Unbekannte Wirklichkeit	
		H_0 wahr	H_0 falsch
Datenabhängige Testentscheidung	H_0 verwerfen	Fehler 1. Art$^{*)}$	Richtige Entscheidung
	H_0 beibehalten	Richtige Entscheidung	Fehler 2. Art$^{**)}$

$*$): Auftrittswahrscheinlichkeit α ist unter Kontrolle.
$**$): Auftrittswahrscheinlichkeit β ist *nicht* unter Kontrolle.

Offenbar werden H_0 und H_1 nicht symmetrisch behandelt. Allerdings kommt in der Praxis dem Fehler 1. Art auch häufig ein größeres Gewicht zu, da H_0 oft eine Modellannahme ist, die sich bisher bewährt hat. Deshalb will man sich durch Vorgabe eines kleinen αs dagegen schützen, H_0 ungerechtfertigterweise zu verwerfen:

- Die Entscheidung „H_0 verwerfen" ist verlässlich, weil die Wahrscheinlichkeit für eine Fehlentscheidung hierbei unter Kontrolle und üblicherweise klein ist. Man sagt, die Entscheidung „H_0 verwerfen" ist *statistisch signifikant* (auf dem (Signifikanz-)Niveau α).

- Die Entscheidung „H_0 beibehalten“ ist *nicht* verlässlich, denn selbst im Fall, dass H_0 falsch ist, besitzt die (Fehl-)Entscheidung H_0 beizubehalten die möglicherweise hohe Wahrscheinlichkeit β. Man sagt, die Entscheidung „H_0 beibehalten“ ist *nicht signifikant* (auf dem (Signifikanz-)Niveau α).

Fazit: Statistische Tests sind *Instrumente zum Widerlegen von Nullhypothesen*, nicht zu deren Bestätigung. (M. a. W., sie sind Werkzeuge für den Nachweis der jeweiligen Alternative.)

Wie kommt man jedoch überhaupt zu einer Entscheidung gegen oder für H_0?
Das generelle Konstruktionsprinzip statistischer Tests ist, eine geeignete (Test-)Statistik zu finden, deren wahrscheinlichkeitstheoretisches Verhalten, d. h. deren Verteilung im Fall der Gültigkeit von H_0 (kurz: „unter H_0“) bestimmt werden kann und unter H_1 in bekannter Weise davon abweicht, um darauf basierend eine Entscheidungsregel anzugeben, die für konkrete Daten die Frage zu beantworten erlaubt, ob H_0 verworfen werden kann/muss/darf. (Formal und ausführlich für ein diskretes Verteilungsmodell finden Sie dies z. B. in Kap. 30 von [51, Henze (2021)] dargestellt.)

Die Bestimmung des wahrscheinlichkeitstheoretischen Verhaltens einer Teststatistik ist (in der Regel) nur im Rahmen eines wahrscheinlichkeitstheoretischen Modells für den dem Gesamtproblem zugrunde liegenden „Datengenerierungsmechanismus“ möglich. Wir betrachten im folgenden Paragraph beispielhaft das (stetige) Normalverteilungsmodell.

8.1.3 Konstruktion eines Hypothesentests im Normalverteilungsmodell

Wir wollen zunächst einen **zweiseitigen Test** samt Entscheidungsregel **für eine Hypothese über den Erwartungswert in einem einzelnen Normalverteilungsmodell** herleiten. Dazu seien $X_1, \ldots, X_n$ unabhängige und $\mathcal{N}(\mu, \sigma_0^2)$-verteilte Zufallsvariablen, wobei μ unbekannt und σ_0^2 **bekannt** ist. Zu testen sei für ein gegebenes μ_0

$$H_0 : \mu = \mu_0 \quad \text{gegen} \quad H_1 : \mu \neq \mu_0 \qquad \text{zum Signifikanzniveau } \alpha.$$

Dieses Verfahren (siehe z. B. auch [51, Henze (2021)], Abschnitt 34.9) heißt zweiseitiger Gaußtest. Wie in §8.1.1 schon motiviert, ist die Verteilung des zufälligen Differenzbetrages $|\bar{X}_n - \mu_0|$ unter H_0 wichtig, um eine Entscheidung gegen oder für H_0 zu fällen, denn er misst den Unterschied zwischen dem beobachteten Mittelwert und dem unter H_0 erwarteten. Aus diesem Grund machen wir nun die folgende ...

Annahme: **Die Nullhypothese $H_0 : \mu = \mu_0$ sei richtig.**
Dann sind $X_1, \ldots, X_n$ unabhängig $\mathcal{N}(\mu_0, \sigma_0^2)$-verteilt und für $\bar{X}_n = \frac{1}{n}\sum_{i=1}^n X_i$ gilt $\bar{X}_n \sim \mathcal{N}(\mu_0, \sigma_0^2/n)$ und somit:

$$\frac{\bar{X}_n - \mu_0}{\sigma_0/\sqrt{n}} \sim \mathcal{N}(0,1) \qquad \text{für alle } n \geq 1$$

D. h. insbesondere, dass $\bar{X}_n$ in der Nähe von μ_0 erwartet wird, aber sogar auch, dass die Verteilung von $|\bar{X}_n - \mu_0|$ vollständig bekannt ist und für jedes $x > 0$ gilt:

$$\mathbb{P}_{H_0}\left(\left|\underbrace{\frac{\bar{X}_n - \mu_0}{\sigma_0/\sqrt{n}}}_{\sim\mathcal{N}(0,1)}\right| \leq x\right) = \Phi(x) - \Phi(-x) = 2\Phi(x) - 1 \quad \text{(wg. } \Phi\text{s Symmetrie)}$$

Daraus leitet sich sofort auch eine Wahrscheinlichkeitsaussage über das Ereignis ab, dass die betragliche Differenz $|\bar{X}_n - \mu_0|$ einen Wert *über*schreitet, bzw. (aufgrund der Stetigkeit der Normalverteilung) mindestens so groß ist wie dieser Wert:

$$\mathbb{P}_{H_0}\left(\frac{|\bar{X}_n - \mu_0|}{\sigma_0/\sqrt{n}} \geq x\right) = 2(1 - \Phi(x))$$

Unser Ziel ist nun, zu bestimmen, welche Differenzbeträge unter H_1 zu erwarten sind und also unter H_0 „ungewöhnlich groß“ sind: Man *setzt fest (!)*, dass dies der Fall ist, wenn sie auf oder oberhalb einer Schwelle liegen, die unter H_0 eine *kleine* Überschreitungwahrscheinlichkeit besitzt. Geben wir uns eine kleine Überschreitungswahrscheinlichkeit α vor (mit $0 < \alpha \leq 1/2$) und setzen wir für x das $(1-\alpha/2)$-Quantil der Standardnormalverteilung, also $u_{1-\alpha/2} \equiv \Phi^{-1}(1-\alpha/2)$ ein, so folgt

$$\mathbb{P}_{H_0}\left(\frac{|\bar{X}_n - \mu_0|}{\sigma_0/\sqrt{n}} \geq u_{1-\alpha/2}\right) = 2(1-\Phi(u_{1-\alpha/2})) = \alpha$$

Wir haben also hiermit den Bereich ungewöhnlich großer Differenzbeträge $|\bar{X}_n - \mu_0|$ gefunden: Es sind all diejenigen Werte, für welche $|\bar{X}_n - \mu_0|/(\sigma_0/\sqrt{n})$ mindestens so groß wie der kritische Wert $u_{1-\alpha/2}$ ist.

Mit anderen Worten formuliert bedeutet das obige Resultat:

- Unter $H_0 : \mu = \mu_0$ fällt $\bar{X}_n$ tendenziell **in** das Intervall $\left(\mu_0 - \frac{u_{1-\alpha/2}\,\sigma_0}{\sqrt{n}},\ \mu_0 + \frac{u_{1-\alpha/2}\,\sigma_0}{\sqrt{n}}\right)$, nämlich mit der großen Wahrscheinlichkeit $1-\alpha$.

Andererseits tritt $\bar{X}_n$ (weil stets „in der Nähe“ von μ) unter $H_1 : \mu \neq \mu_0$ tendenziell entfernt von μ_0 auf, realisiert sich dann also tendenziell außerhalb des obigen Intervalls. Das *Komplement* von $\left(\mu_0 - \frac{u_{1-\alpha/2}\,\sigma_0}{\sqrt{n}},\ \mu_0 + \frac{u_{1-\alpha/2}\,\sigma_0}{\sqrt{n}}\right)$ wird daher (für H_0) kritischer Bereich genannt (i. S. v. „gefährlich“ für H_0). Es liegt also insgesamt folgendes Fazit nahe:

- Realisiert sich $\bar{X}_n$ in dem kritischen Bereich, worin es unter H_0 nur mit der kleinen Wahrscheinlichkeit α zu erwarten ist, so spricht dies gegen $H_0 : \mu = \mu_0$ und für $H_1 : \mu \neq \mu_0$.

Das führt für eine konkrete Stichprobe $x_1, \ldots, x_n$ zu der folgenden

Entscheidungsregel für den zweiseitigen Test auf Basis des kritischen Wertes:

$$H_0 \text{ ist zum Niveau } \alpha \text{ zu verwerfen} \quad \Longleftrightarrow \quad \frac{|\bar{x}_n - \mu_0|}{\sigma_0/\sqrt{n}} \geq u_{1-\alpha/2}$$

Zur Erinnerung: Dieses Verfahren heißt *zweiseitiger* Gaußtest, weil die Alternative $H_1 : \mu \neq \mu_0$ für μ eine Abweichung von μ_0 in *zwei* Richtungen zulässt. Die Größe $(\bar{X}_n - \mu_0)/(\sigma_0/\sqrt{n})$ ist die Teststatistik und $u_{1-\alpha/2}$ ist, wie oben schon gesagt, der kritische Wert für ihren Betrag. Die Entscheidungsregel garantiert, dass der Fehler 1. Art eine Auftrittswahrscheinlichkeit von (höchstens) α und somit der Test das Signifikanzniveau α hat.

Das **Beispiel** der Körpergewichte von Labormäusen (von Seite 127): Angenommen, das (vorab festgelegte) Signifikanzniveau α des Tests ist 0.05. Dann lautet die Entscheidungsregel wegen $n = 50$, $\sigma_0 = 4.2$ und $u_{1-\alpha/2} = u_{0.975} \doteq 1.96$ für die konkrete Stichprobe $y_1, \ldots, y_n$: „Verwirf H_0, falls $|\bar{y}_n - \mu_0|/(4.2/\sqrt{50}) \geq 1.96$“. Wir hatten $|\bar{y}_n - \mu_0| = 1.54$ und somit $1.54/(4.2/\sqrt{50}) \doteq 2.593$, sodass H_0 hier in der Tat verworfen werden kann.

Im Fall des **einseitigen Gaußtests**

$$H_0' : \mu \overset{(\leq)}{\geq} \mu_0 \qquad \text{gegen} \qquad H_1' : \mu \overset{(>)}{<} \mu_0 \qquad \text{zum Signifikanzniveau } \alpha$$

erfolgt die obige Argumentation analog (mit einer kleinen Zusatzüberlegung für den Fall, dass H_0 zutrifft, weil echte Ungleichheit gilt) und führt schließlich zur

Entscheidungsregel für den einseitigen Test auf Basis des kritischen Wertes:

$$H_0' \text{ ist zum Niveau } \alpha \text{ zu verwerfen} \quad \Longleftrightarrow \quad \frac{\bar{x}_n - \mu_0}{\sigma_0/\sqrt{n}} \overset{(\geq +)}{\leq -} u_{1-\alpha}$$

Beachte, dass hier das $(1-\alpha)$- und nicht das $(1-\alpha/2)$-Quantil der Standardnormalverteilung zum Einsatz kommt!

8.1.4 Der p-Wert

Gleich nochmal zum **Beispiel** der Körpergewichte von Labormäusen: Offenbar und auf gewünschte Weise beeinflusst das Niveau α die Testentscheidung. Wir wollen die Wirkung von α in diesem Zahlenbeispiel mit $n = 50$ und $\sigma_0 = 4.2$ etwas näher untersuchen: Für verschiedene Werte von α erhalten wir verschiedene kritische Werte und abhängig davon entsprechende Testentscheidungen:

α	$u_{1-\alpha/2}$	Testentscheidung
0.1	1.6448	H_0 verwerfen!
0.05	1.9600	H_0 verwerfen!
0.01	2.5758	H_0 verwerfen!
0.005	2.8070	H_0 beibehalten!
0.001	3.2905	H_0 beibehalten!

Generell wächst der kritische Wert $u_{1-\alpha/2}$ monoton, wenn α kleiner wird (hier sogar streng monoton als eine direkte Konsequenz aus dem streng monotonen Wachstum der Quantilfunktion der Standardnormalverteilung). Dieser Sachverhalt gilt nicht nur für die kritischen Werte in zweiseitigen Gaußtests, sondern allgemein für jeden Test:

> *Bei vorliegenden Daten* gibt es ein kleinstes Niveau, auf dem H_0 gerade noch verworfen werden kann. Dieses kleinste Niveau heißt p-Wert.

Demnach ist die Beziehung zwischen dem p-Wert und dem von der Anwenderin bzw. dem Anwender (**vor** der Testdurchführung) fest gewählten Niveau α wie folgt:

> Test-Entscheidungsregel auf Basis des p-Wertes:
>
> H_0 ist zum Niveau α zu verwerfen $\iff$ p-Wert $\leq \alpha$

Software-Pakete berechnen in der Regel den p-Wert zu den eingegebenen Daten und überlassen somit die letztendliche Testentscheidung ihrem/r Anwender/in.

Wir wollen die **Berechnung des p-Wertes beim zweiseitigen Gaußtest** aus §8.1.3 exemplarisch durchführen. Laut obiger Definition des p-Wertes gilt:

$$\text{Der (zweiseitige) } p\text{-Wert ist das kleinste } \alpha \text{ mit } \frac{|\bar{x}_n - \mu_0|}{\sigma_0/\sqrt{n}} \geq u_{1-\alpha/2}.$$

Da $u_{1-\alpha/2}$ streng monoton fallend und stetig in α ist, gleicht der p-Wert gerade demjenigen α, für welches in der vorstehenden Ungleichung Gleichheit gilt. Daraus erhalten wir:

$$\text{Zweiseitiger } p\text{-Wert} = 2\left(1 - \Phi\left(\frac{|\bar{x}_n - \mu_0|}{\sigma_0/\sqrt{n}}\right)\right)$$

Die Formel für diesen p-Wert deckt sich mit seiner folgenden, sehr suggestiven Interpretation (die ebenfalls allgemein für jeden Test gilt):

> Der p-Wert ist die Wahrscheinlichkeit, dass die (zufällige!) Teststatistik unter H_0 einen Wert annimmt, der mindestens so weit „in Richtung H_1“ liegt wie derjenige, der sich für die konkret vorliegenden Daten ergeben hat.

Begründung: Zur Abkürzung sei $T_n := (\bar{X}_n - \mu_0)/(\sigma_0/\sqrt{n})$ die Teststatistik und $t_n^* := (\bar{x}_n - \mu_0)/(\sigma_0/\sqrt{n})$ der von ihr für die konkret vorliegenden Daten angenommene Wert. Dann gilt hier beim zweiseitigen Gaußtest:

$$\mathbb{P}_{H_0}\left(\begin{array}{l}\text{Teststatistik } T_n \text{ realisiert sich min-}\\ \text{destens so weit „in Richtung } H_1\text{“ wie}\\ \text{der konkret vorliegende Wert } t_n^*.\end{array}\right) = \mathbb{P}_{H_0}\left(|T_n| \geq |t_n^*|\right)$$

$$= 1 - \mathbb{P}_{H_0}\left(|T_n| < |t_n^*|\right) = 1 - \mathbb{P}_{H_0}\left(-|t_n^*| < T_n < |t_n^*|\right)$$

$$= 1 - \{\Phi(|t_n^*|) - \Phi(-|t_n^*|)\} = 2(1 - \Phi(|t_n^*|)) = \text{Zweiseitiger } p\text{-Wert}$$

Bemerkungen:

1. Als Heuristik kann man demnach sagen: Ein kleiner p-Wert bedeutet, dass ein „mindestens so extremer“ Wert wie der für die Teststatistik beobachtete unter H_0 unwahrscheinlich gewesen ist und daher gegen H_0 spricht. Als „Eselsbrücke“ zur Interpretation diene der Buchstabe „p“: Der p-Wert quantifiziert gewissermaßen die **P**lausibilität der Nullhypothese derart, dass H_0 umso weniger plausibel erscheint, je kleiner er ist. Anders, aber auch *stark* heuristisch formuliert:

   > Je kleiner der p-Wert, umso mehr „Vertrauen“ kann man in die Richtigkeit der Entscheidung haben, H_0 zu verwerfen.

   Ist ein p-Wert „sehr klein“, d. h., unterschreitet er das vorgegebene Signifikanzniveau α, so wird er – stellvertretend für die Testentscheidung H_0 zu verwerfen – oft kurz „signifikant“ (auf dem Niveau α) genannt.

2. Offenbar hängt der p-Wert von der Art der Alternative H_1 ab! Oben haben wir den zweiseitigen Fall betrachtet; den einseitigen diskutieren wir unten, wobei dort noch unterschieden werden muss, ob die Alternative $\mu < \mu_0$ oder $\mu > \mu_0$ lautet.

3. In der Praxis ist es sinnvoll, bei der Angabe eines Testergebnisses den p-Wert mitzuliefern, da er mehr Information enthält als die Testentscheidung allein, wie wir an unserem **Beispiel** illustrieren wollen: Dort hatten wir $n = 50$, $|\bar{x}_n - \mu_0| = 1.54$ und $\sigma_0 = 4.2$. Einsetzen dieser Werte in obige Formel liefert einen p-Wert von 0.009522, sodass ein Verwerfen von H_0 auf dem Niveau 0.01 möglich ist. Allerdings ist es doch ein etwas knapper Unterschied, auf dem diese Entscheidung beruht. Wäre $\bar{x}_n = 46.75$ g und somit $|\bar{x}_n - \mu_0| = 2.25$ beobachtet worden, ergäbe sich ein p-Wert von 0.00015 und das Niveau 0.01 würde *deutlich* unterschritten, sodass ein Verwerfen von H_0 wesentlich besser abgesichert wäre.

4. In der Praxis ist es aber nicht sinnvoll, *nur* den p-Wert anzugeben, denn aus seiner Kenntnis allein lässt sich im Allgemeinen nicht ableiten, warum die Daten zu einem signifikanten oder nicht signifikanten Ergebnis geführt haben: Ist der p-Wert signifikant, kann die Ursache ein wahrhaft großer Unterschied $|\mu - \mu_0|$ sein oder nur ein großes n relativ zur Streuung σ_0; ist er nicht signifikant, kann die Ursache ein wahrhaft kleines $|\mu - \mu_0|$ sein oder nur ein relativ zu σ_0 zu kleines n.

5. Des weiteren ist es sinnvoll bis gefordert, auf Basis der beobachteten Daten ein Konfidenzintervall für den Parameter anzugeben, über den eine Hypothese getestet wurde, da man in der Regel an der Größenordnung des Effektes der betrachteten Behandlung interessiert ist (und nicht nur daran, ob es ein signifikanter Effekt ist). (Vgl. z. B. [42, Gardner & Altman (1986)] und [62, ICH-E9 (1998), §5.5, p. 25 – 26].)

Wir zeigen nun die **p-Wert-Berechnung im Fall des einseitigen Gaußtests** von

$$H_0' : \mu \overset{(\leq)}{\geq} \mu_0 \qquad \text{gegen} \qquad H_1' : \mu \overset{(>)}{<} \mu_0$$

Gemäß der Entscheidungsregel auf Basis des kritischen Wertes (Seite 130, oben) folgt:

$$\text{Der (einseitige) } p\text{-Wert ist das kleinste } \alpha \text{ mit } \frac{\bar{x}_n - \mu_0}{\sigma_0/\sqrt{n}} \overset{(\geq +)}{\leq -} u_{1-\alpha},$$

sodass sich mit einer dem zweiseitigen Fall analogen Argumentation zu Stetigkeit und Monotonie von $u_{1-\alpha}$ in α ergibt:

$$\text{Einseitiger } p\text{-Wert} = 1 - \Phi\left(\overset{(+)}{-} \frac{\bar{x}_n - \mu_0}{\sigma_0/\sqrt{n}}\right) = \begin{cases} \Phi\left(\frac{\bar{x}_n - \mu_0}{\sigma_0/\sqrt{n}}\right) & \text{für} \quad H_1' : \mu < \mu_0 \\ 1 - \Phi\left(\frac{\bar{x}_n - \mu_0}{\sigma_0/\sqrt{n}}\right) & \text{"} \quad H_1' : \mu > \mu_0 \end{cases}$$

Zusammenfassend formulieren wir das allgemein gültige, **prinzipielle Vorgehen bei statistischen Tests**:

1. Formulierung von Hypothese H_0 und Alternative H_1 (insbes. auch Ein- oder Zweiseitigkeit), um den „Datengenerierungsmechanismus" in zwei disjunkte Verteilungsfamilien zu zerlegen, sowie Festlegung des Signifikanzniveaus α.
2. Wahl des Tests.
3. Erhebung der Daten und explorative Analyse, um die verteilungstheoretischen Voraussetzungen des Tests zu prüfen. (Falls zweifelhaft, zurück zu Punkt 2 oder sogar 1.)
4. Berechnung der Teststatistik und des p-Wertes.
5. Entscheidung, ob H_0 verworfen werden kann/muss, unter Angabe von H_0 und H_1, des Tests und des p-Wertes (sowie eventuell relevanter Informationen zur Teststatistik wie z. B. ihre Freiheitsgrade).

8.2 Konfidenzintervalle für die Parameter der Normalverteilung

Häufig ist es zusätzlich zu einem Punktschätzer oder einem Hypothesentest für einen unbekannten Parameter – wie dem Erwartungswert oder der Varianz – gewünscht oder sogar gefordert, einen Bereichsschätzer mit einer festgelegten Überdeckungswahrscheinlichkeit $1-\alpha$, also ein Konfidenzintervall (KI) zum (Konfidenz-)Niveau $1-\alpha$ anzugeben. Dies ist sehr empfehlenswert, da der p-Wert allein (wie vorseitig in der 4. Bemerkung bereits erläutert) keine Aussage über die Größenordnung eines Effektes $\mu - \mu_0$ (oder $\mu_1 - \mu_2$ in noch folgenden Paragraphen) machen kann.

8.2.1 Der Erwartungswert μ

Im Modell unabhängiger Zufallsvariablen $X_1, \ldots, X_n$ mit der identischen Verteilung $\mathcal{N}(\mu, \sigma^2)$ ist bekanntermaßen $\sqrt{n}(\bar{X}_n - \mu)/\sigma \sim \mathcal{N}(0,1)$ für alle $n \geq 1$ und somit

$$\mathbb{P}\left(\bar{X}_n - \frac{u_{1-\alpha/2}\,\sigma}{\sqrt{n}} \leq \mu \leq \bar{X}_n + \frac{u_{1-\alpha/2}\,\sigma}{\sqrt{n}}\right) = 2\Phi(u_{1-\alpha/2}) - 1 = 1 - \alpha \quad \forall \mu \in \mathbb{R}, \sigma \in \mathbb{R}^+$$

Bei bekanntem σ^2 ist ein Konfidenzintervall für den unbekannten Erwartungswert μ also wie folgt angebbar:

$X_1, \ldots, X_n$ seien unabhängig und identisch $\mathcal{N}(\mu, \sigma^2)$-verteilt. Dann ist

$$\left[\bar{X}_n - \frac{u_{1-\alpha/2}\,\sigma}{\sqrt{n}}, \bar{X}_n + \frac{u_{1-\alpha/2}\,\sigma}{\sqrt{n}}\right] \text{ ein exaktes KI für } \mu \text{ zum Niveau } 1-\alpha.$$

Beachte:

- Die nicht zufällige Länge dieses sog. Gaußschen Konfidenzintervalls hängt vom Niveau $1-\alpha$, der Varianz σ^2 und vom Stichprobenumfang n ab: Je höher das Niveau bzw. je größer die Varianz, umso länger das Konfidenzintervall; je größer der Stichprobenumfang, umso kürzer das Konfidenzintervall.

- Das obige Konfidenzintervall ist symmetrisch um $\bar{X}_n$.

In Anwendungen dürfte es allerdings der Regelfall sein, dass σ^2 unbekannt ist (und somit ein Störparameter, Englisch: nuisance parameter). Das obige Konfidenzintervall ist dann nutzlos, da seine Grenzen σ explizit enthalten. Die Lösung des Problems ist, das unbekannte σ durch seinen Schätzer $\hat{\sigma}_n$ zu ersetzen, denn im betrachteten Normalverteilungsmodell ist $\sqrt{n}(\bar{X}_n - \mu)/\hat{\sigma}_n \sim t_{n-1}$ für alle $n \geq 2$, wobei t_{n-1} die (Studentsche) t-Verteilung mit $n-1$ Freiheitsgraden (kurz: t_{n-1}-Verteilung) ist, deren Verteilungsfunktion bekannt ist und wir mit $F_{t_{n-1}}$ bezeichnen werden.

Daraus ergibt sich (analog zum Fall des bekannten σ^2) für jedes $n \geq 2$

$$\mathbb{P}\left(\bar{X}_n - \frac{t_{n-1;1-\alpha/2}\,\hat{\sigma}_n}{\sqrt{n}} \leq \mu \leq \bar{X}_n + \frac{t_{n-1;1-\alpha/2}\,\hat{\sigma}_n}{\sqrt{n}}\right) = 2F_{t_{n-1}}(t_{n-1;1-\alpha/2}) - 1 = 1-\alpha,$$

wobei $t_{n-1;1-\alpha/2}$ das $(1-\alpha/2)$-Quantil der t_{n-1}-Verteilung ist (siehe z. B. [51, Henze (2021)], Abschnitt 34.13).

Bei unbekanntem σ^2 folgt also:

$X_1, \ldots, X_n$ seien unabhängig und identisch $\mathcal{N}(\mu, \sigma^2)$-verteilt. Dann ist

$$\left[\bar{X}_n - \frac{t_{n-1;1-\alpha/2}\,\hat{\sigma}_n}{\sqrt{n}}, \bar{X}_n + \frac{t_{n-1;1-\alpha/2}\,\hat{\sigma}_n}{\sqrt{n}}\right] \text{ ein exaktes KI für } \mu \text{ zum Niveau } 1-\alpha.$$

Beachte:

- Die zufällige (!) Länge dieses sog. Studentschen Konfidenzintervalls hängt vom Niveau $1-\alpha$, der (zufälligen!) Stichprobenvarianz $\hat{\sigma}_n^2$ und vom Stichprobenumfang n ab: Je höher das Niveau bzw. je größer die Stichprobenvarianz, umso länger das Konfidenzintervall; je größer der Stichprobenumfang, umso kürzer das Konfidenzintervall.

- Aufgrund der Tatsache, dass die $(1-\alpha)$-Quantile der t-Verteilungen stets betraglich größer als die der Standardnormalverteilung sind, ist dieses Konfidenzintervall grundsätzlich etwas länger als ein Konfidenzintervall bei bekannter Varianz, selbst wenn (zufällig) $\sigma = \hat{\sigma}_n$ sein sollte. Dies ist gewissermaßen der „Preis“, den man für die Unkenntnis der Varianz zu zahlen hat.

Bemerkungen:

- Am Beispiel des obigen Konfidenzintervalls rekapitulieren wir die generell gültige **Häufigkeitsinterpretation** von Konfidenzintervallen:

Das Ereignis $\left\{\bar{X}_n - \frac{t_{n-1;1-\alpha/2}\,\hat{\sigma}_n}{\sqrt{n}} \le \mu \le \bar{X}_n + \frac{t_{n-1;1-\alpha/2}\,\hat{\sigma}_n}{\sqrt{n}}\right\}$ hat die Wahrscheinlichkeit $1-\alpha$. D. h., für $(1-\alpha)\cdot 100$ % *aller* Stichproben $x_1, \ldots, x_n$ trifft die Aussage

$$\bar{x}_n - \frac{t_{n-1;1-\alpha/2}\,s_n}{\sqrt{n}} \le \mu \le \bar{x}_n + \frac{t_{n-1;1-\alpha/2}\,s_n}{\sqrt{n}}$$

zu, für die verbleibenden $\alpha \cdot 100$ % *aller* Stichproben jedoch nicht.

- Eine häufig anzutreffende Formulierung (hier mit erfundenen Zahlen) lautet

  „Das wahre μ liegt mit einer Wahrscheinlichkeit von 95 % zwischen 0.507 und 0.547." **DAS IST UNSINN!**

  Nach unserer Modellierung hat das unbekannte μ einen *festen* Wert, also liegt es entweder zwischen 0.507 und 0.547 oder nicht. Nicht μ ist zufällig, sondern die Schranken des Konfidenzintervalls, und zwar natürlich nur *bevor* die Daten erhoben worden sind. Neue Stichproben ergeben also im Allgemeinen *andere* Schranken bei *gleichem* μ. Die **korrekte Formulierung** lautet schlicht „$[0.507, 0.547]$ ist ein 95 %-Konfidenzintervall für μ" und ihre (Häufigkeits-) Interpretation steht oben.

- Sollen die Grenzen eines Konfidenzintervalls gerundet werden, so ist die Untergrenze stets *ab*- und die Obergrenze stets *auf*zurunden, um zu vermeiden, dass das Konfidenzintervall durch Rundung zu kurz wird und deshalb das nominelle Niveau nicht mehr einhält. Das so erhaltene Konfidenzintervall hat dann *mindestens* das angestrebte Niveau. (Eine Vorgehensweise, die dazu führt, dass Niveaus auf jeden Fall eingehalten und möglicherweise „übererfüllt" werden, nennt man auch konservativ.)

- Gelegentlich ist aus sachlichen Gründen tatsächlich nur eine der beiden Konfidenzintervallgrenzen von Interesse, weil man nur eine obere oder untere Schranke für den untersuchten Effekt benötigt. Das sollte ausgenutzt werden, denn es existieren auch *einseitige* Konfidenzintervalle, bei denen eine der beiden Intervallgrenzen nach $-\infty$ oder ∞ „verschoben" ist, sodass nur die jeweils andere Grenze „übrig" bleibt. Der Vorteil liegt darin, dass bei gleichem Konfidenzniveau die untere (obere) Konfidenzschranke größer (kleiner) ist als die Untergrenze (Obergrenze) des entsprechenden zweiseitigen Konfidenzintervalls.

  Die exakten einseitigen Studentschen Konfidenzintervalle zum Niveau $1-\alpha$ lauten (wobei Sie als entscheidenden Unterschied beachten sollten, dass die t-Quantile hier für $1-\alpha$ und nicht $1-\alpha/2$ verwendet werden):

$$\left[\bar{X}_n - \frac{t_{n-1;1-\alpha}\,\hat{\sigma}_n}{\sqrt{n}}, +\infty\right) \qquad \text{und} \qquad \left(-\infty, \bar{X}_n + \frac{t_{n-1;1-\alpha}\,\hat{\sigma}_n}{\sqrt{n}}\right]$$

8.2.2 Die Varianz σ^2

Für unabhängig und identisch $\mathcal{N}(\mu, \sigma^2)$-verteilte Zufallsvariablen $X_1, \ldots, X_n$ ist $(n-1)\hat{\sigma}_n^2/\sigma^2 \sim \chi^2_{n-1}$ für alle $n \ge 2$. Daraus leitet sich (bei unbekanntem Erwartungswert μ) das folgende Konfidenzintervall für die unbekannte Varianz σ^2 ab:

$X_1, \ldots, X_n$ seien unabhängig und identisch $\mathcal{N}(\mu, \sigma^2)$-verteilt. Dann ist

$$\left[\frac{(n-1)\hat{\sigma}_n^2}{\chi^2_{n-1;1-\alpha/2}}, \frac{(n-1)\hat{\sigma}_n^2}{\chi^2_{n-1;\alpha/2}}\right]$$

ein exaktes KI für σ^2 zum Niveau $1-\alpha$.

Beachte: Die zufällige (!) Länge dieses – um $\hat{\sigma}_n^2$ *asymmetrischen* – Konfidenzintervalls hängt vom Niveau $1-\alpha$, der (zufälligen!) Stichprobenvarianz $\hat{\sigma}_n^2$ und vom Stichprobenumfang n ab: Je

höher das Niveau bzw. je größer die Stichprobenvarianz, umso länger das Konfidenzintervall; je größer der Stichprobenumfang, umso kürzer das Konfidenzintervall (was hier nicht offensichtlich ist, aber durch die Beziehung $F_{k,\infty} \stackrel{\mathcal{L}}{=} \chi^2_k/k$ und das monotone Wachstum des Quantils $F_{k,\infty;\gamma}$ in k nachvollzogen werden kann).

Bemerkung: Auch für die Varianz existieren exakte einseitige Konfidenzintervalle. Sie lauten zum Niveau $1-\alpha$ (wobei Sie als entscheidenden Unterschied beachten sollten, dass die χ-Quantile hier für $1-\alpha$ und nicht $1-\alpha/2$ verwendet werden):

$$\left[\frac{(n-1)\hat{\sigma}_n^2}{\chi^2_{n-1;1-\alpha}}, +\infty\right) \qquad \text{und} \qquad \left(0, \frac{(n-1)\hat{\sigma}_n^2}{\chi^2_{n-1;\alpha}}\right]$$

Eine z. B. auf der Basis von Daten aus einer sog. Pilot- oder Vorstudie ermittelten Konfidenzoberschranke für die Varianz kann in der Fallzahlschätzung bei der Planung einer (klinischen) Hauptstudie Anwendung finden.

8.2.3 Zur Fallzahlschätzung für Konfidenzintervalle mit vorgegebener „Präzision“

Soll bei der Schätzung des Erwartungswertes μ einer Normalverteilung durch ein Konfidenzintervall eine gewisse „Präzision“ erzielt werden, so ist zunächst zu klären, was unter „Präzision“ zu verstehen ist. Hier gibt es mindestens zwei verschiedene Konzepte, wie z. B. dass der *Erwartungswert* der zufälligen (!) Länge des Konfidenzintervalls eine vorgegebene Schranke nicht überschreiten soll (was natürlich nicht bedeutet, dass das konkret berechnete Intervall diese Längenbeschränkung erfüllt) oder dass das zufällige (!) Konfidenzintervall mit vorgegebener großer Wahrscheinlichkeit (auch Power oder Güte genannt und mit $1-\beta$ bezeichnet) komplett innerhalb eines Intervalls $[\mu-\Delta, \mu+\Delta]$ mit ebenfalls vorgegebenem Δ zu liegen kommt (und dies jeweils weiterhin bei festgelegter Überdeckungswahrscheinlichkeit $1-\alpha$ für μ). Ausführlich sind die Problematik und Lösungen dafür z. B. in [6, Bock (1998)] in Abschnitt 3.2 beschrieben.

8.3 Eine Hilfsfunktion für die explorative Datenanalyse

Viele klassische inferenzstatistische Verfahren für metrische Daten hängen stark von Verteilungsannahmen über die Daten ab, auf die sie angewendet werden sollen. In den in diesem Kapitel vorgestellten *parametrischen* Verfahren sind dies die Normalverteilungsannahme und die Unabhängigkeit der Daten. Die Beurteilung der Gültigkeit dieser Annahmen kann durch die explorative Datenanalyse (EDA) unterstützt werden, wofür mehrere grafische Methoden zur Verfügung stehen, die wir zum Teil bereits kennengelernt haben.

In Abschnitt 6.1 hatten wir schon einige dieser Methoden in einer neuen Funktion zusammengefasst, die wir im Folgenden noch etwas erweitern:

Eine (erweiterte) Funktion für die explorative Datenanalyse

```
> eda <- function(x) {
+ oldp <- par(mfrow = c(3, 2),
+   mar = c(3.5, 3, 2.6, 0.5) + 0.1,
+   mgp = c(1.7, 0.6,0), cex = 0.8)
+
+ hist(x, freq = FALSE,
+      ylab = "Dichte", xlab = "",
+      main = "Histogramm")
+ rug(x, col = "blue")
+ boxplot(x, horizontal = TRUE,
+         main = "Boxplot")
+ set.seed(42)
+ stripchart(x, add = TRUE,
+            col = "blue",
+            method = "jitter")
+
+ qa <- diff(quantile(x, c(1, 3)/4))
+ dest <- density(x, width = qa)
+ plot(dest, ylab = "Dichte",
+      main = "Dichteschätzer")
+ rug(x, col = "blue")
+
+ qqnorm(x, datax = TRUE,
+    ylab = "Stichprobenquantile",
+    xlab = "Theoretische Quantile")
+ qqline(x, datax = TRUE)
+ ts.plot(x, ylab = "",
+         xlab = "Zeit bzw. Index",
+         main = "Zeitreihe")
+
+ acf(x, main = "Autokorrelation")
+
+ par(oldp)
+
+ d <- max(3, getOption("digits")-3)
+ c(summary(x),
+   "St.Dev." = signif(sd(x),
+                      digits = d))
+ }
```

Die hiesige Funktion `eda` ist gegenüber der in Abschnitt 6.1 definierten insofern erweitert, als dass durch `par(mfrow = c(3,2), ....)` zunächst ein (3×2)-Mehrfachplotrahmen mit gewissen Modifikationen – vor allem von Abständen – des Standardlayouts angelegt und die *vorherige* `par`-Einstellung in `oldp` gespeichert wird. Sodann werden in die ersten vier der sechs Plotrahmen – ebenfalls leicht modifiziert im Vergleich zu Abschnitt 6.1 – ein flächennormiertes Histogramm (durch `rug` ergänzt um die blauen Markierungen der Rohdaten auf der waagrechten Achse), ein horizontaler Boxplot mit leicht „verwackelt“ überlagerten Rohdaten (in blau), ein Kern-Dichteschätzer (mit Gaußkern und dem Quartilsabstand der Daten als Fensterbreite; ebenfalls ergänzt um die Rohdatenmarkierungen) sowie ein Normal-Q-Q-Plot (wegen `datax = TRUE` mit den Daten auf der x-Achse und mit vertauschter „Zuständigkeit“ von `xlab` und `ylab` für die Achsenlabels) samt Soll-Linie geplottet.

Zusätzlich werden durch `ts.plot` ein „Zeitreihenplot“ (`ts` wie “time series”) der Elemente von `x` gegen ihre Indices produziert und mittels `acf` die Autokorrelationsfunktion der Daten für mehrere “lags” (zusammen mit den approximativen 95 %-Konfidenzgrenzen) gezeichnet; diese beiden letzten Plots können eine mögliche serielle Korrelation der Daten zutage fördern. Als vorvorletztes wird die *vorherige* `par`-Einstellung wieder reaktiviert und als Rückgabewert liefert die Funktion dann den Vektor der “summary statistics” des Arguments `x` ergänzt um die Standardabweichung der Daten mit zur Ausgabe von `summary` passender Anzahl signifikanter Ziffern.

Zur Demonstration verwenden wir wieder den Datensatz von Michelson zur Bestimmung der Lichtgeschwindigkeit (vgl. Seite 103):

```
> v <- c( 850,  740,  900, 1070,  930,  850,  950,  980,  980,  880,
+        1000,  980,  930,  650,  760,  810, 1000, 1000,  960,  960)
> eda(v)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. St.Dev.
  650.0   850.0   940.0   909.0   980.0  1070.0   104.9
```

Das Resultat der Anwendung der Funktion `eda` auf `v` ist auch hier die Ausgabe der "summary statistics" für `v` und die Grafiken in Abb. 26.

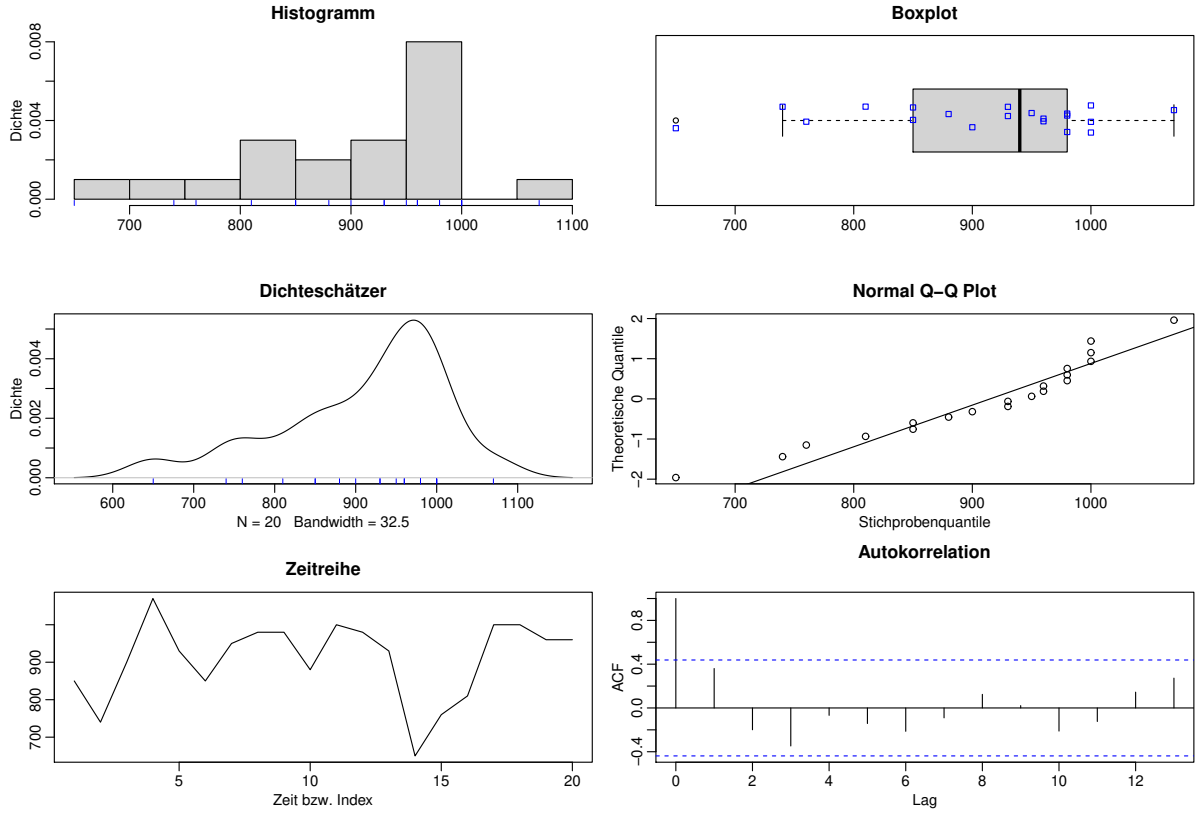

Abbildung 26: Grafisches Ergebnis der Anwendung der erweiterten Funktion `eda` auf `v`.

Bemerkungen:

- Zum theoretischen Hintergrund des Plots der Autokorrelationsfunktion (rechts unten in Abb. 26) verweisen wir z. B. auf [95, Venables & Ripley (2003)] oder [89, Shumway & Stoffer (2011)].
- Bevor wir uns i. F. den angekündigten Ein- und Zweistichprobenproblemen zuwenden sei an die Bemerkung zum Sinn oder Unsinn eines statistischen Tests der Normalverteilungshypothese auf Seite 91 in §4.2.3 erinnert.
- Außerdem merken wir an, dass z. B. [111, Zuur et al. (2009)] einen Beleg dafür liefert, dass sich – hier in Ökologie und Evolutionsbiologie tätige – angewandte StatistikerInnen ausführliche Gedanken darüber machen, dass und wie eine adäquate explorative Datenanalyse zentraler Bestandteil der Auswertung von Daten sein muss (obwohl es sich auf den ersten Blick um Trivialitäten zu handeln scheint).

8.4 Ein Einstichproben-Lokationsproblem

Lokations- (oder eben Lage-)Parameter einer Verteilung charakterisieren ihre „zentrale Lage“. Die bekanntesten Lokationsparameter sind der Erwartungswert (sofern existent) und der (stets existente, aber nicht notwendig eindeutig bestimmte) Median einer Verteilung.

Wir betrachten in diesem Abschnitt Tests für Lokationshypothesen über *eine* zugrunde liegende Population. Unter der Normalverteilungsannahme wird dazu Students t-Test (mit verschiedenen Alternativen) für Hypothesen über den Erwartungswert verwendet. In Fällen, in denen die Daten *nicht* normalverteilt sind – wie es die explorative Datenanalyse im vorliegenden Beispiel der Michelson-Daten auf der voherigen Seite übrigens nahelegt – sollten eher nicht-parametrische (Rang-)Verfahren, wie z. B. Wilcoxons Vorzeichen-Rangsummentest für Hypothesen über den Median einer immerhin noch stetigen und symmetrischen Verteilung, verwendet werden.

Natürlich untersuchen Tests für Erwartungswerthypothesen nicht dasselbe wie solche für Medianhypothesen! Dies muss bei der Interpretation der Ergebnisse beachtet werden (auch wenn das in der praktischen Anwendung selten geschieht ...). Zu Illustrationszwecken werden wir *beide* Verfahren am Michelson-Datensatz aus dem vorherigen Abschnitt vorführen.

8.4.1 Der Einstichproben-t-Test

Als Referenz und zur Erinnerung hier zunächst eine Wiederholung von etwas Theorie (siehe z. B. [51, Henze (2021)], Abschnitt 34.11):

Annahmen: $X_1, \ldots, X_n$ sind unabhängig $\mathcal{N}(\mu, \sigma^2)$-verteilt mit unbekannten μ und σ^2.

Zu testen:

$$H_0 : \mu = \mu_0 \qquad \text{gegen} \qquad H_1 : \mu \neq \mu_0 \qquad \text{zum Signifikanzniveau } \alpha$$

bzw.

$$H_0' : \mu \overset{(\leq)}{\geq} \mu_0 \qquad \text{gegen} \qquad H_1' : \mu \overset{(>)}{<} \mu_0 \qquad \text{zum Signifikanzniveau } \alpha.$$

Teststatistik:

$$\frac{\bar{X}_n - \mu_0}{\hat{\sigma}_n/\sqrt{n}} \sim t_{n-1} \qquad \begin{array}{l}\text{unter } H_0 \text{ bzw. unter } \mu = \mu_0, \\ \text{was der „Rand“ von } H_0' \text{ ist,}\end{array}$$

wobei $\bar{X}_n = \dfrac{1}{n}\displaystyle\sum_{i=1}^{n} X_i$ ist und $\hat{\sigma}_n^2 = \dfrac{1}{n-1}\displaystyle\sum_{i=1}^{n}(X_i - \bar{X}_n)^2$ der Schätzer für σ^2.

Entscheidungsregel für konkrete Daten $x_1, \ldots, x_n$ auf Basis des kritischen Wertes:

$$\text{Verwirf } H_0 \text{ zum Niveau } \alpha \quad \Longleftrightarrow \quad \frac{|\bar{x}_n - \mu_0|}{s_n/\sqrt{n}} \geq t_{n-1;1-\alpha/2}$$

bzw.

$$\text{verwirf } H_0' \text{ zum Niveau } \alpha \quad \Longleftrightarrow \quad \frac{\bar{x}_n - \mu_0}{s_n/\sqrt{n}} \overset{(\geq +)}{\leq} - \, t_{n-1;1-\alpha},$$

wobei $\bar{x}_n$ das arithmetische Mittel und s_n^2 die Stichprobenvarianz der Daten sowie $t_{k;\gamma}$ das γ-Quantil der t_k-Verteilung ist.

Alternativ: Entscheidungsregel für konkrete Daten $x_1, \ldots, x_n$ auf Basis des p-Wertes:

$$\text{Verwirf } H_0^{(\prime)} \quad \Longleftrightarrow \quad p\text{-Wert} \leq \alpha,$$

wobei

$$p\text{-Wert} = \begin{cases} 2\left(1 - F_{t_{n-1}}\left(\frac{|\bar{x}_n - \mu_0|}{s_n/\sqrt{n}}\right)\right) & \text{für} \quad H_1 : \mu \neq \mu_0 \\ F_{t_{n-1}}\left(\frac{\bar{x}_n - \mu_0}{s_n/\sqrt{n}}\right) & \text{"} \quad H_1' : \mu < \mu_0 \\ 1 - F_{t_{n-1}}\left(\frac{\bar{x}_n - \mu_0}{s_n/\sqrt{n}}\right) & \text{"} \quad H_1' : \mu > \mu_0 \end{cases}$$

und F_{t_k} die (um 0 symmetrische) Verteilungsfunktion der t_k-Verteilung ist. (Die Herleitung dieser p-Wert-Formel erfolgt völlig analog zu der des Gaußtests auf den Seiten 131 und 133.)

Die **R**-Funktion `t.test` berechnet den p-Wert zu den eingegebenen Daten und überlässt die Testentscheidung dem/der Anwender/in (wie die meisten Software-Pakete). Dies geschieht auch bei allen anderen implementierten Tests. Aus diesem Grund werden wir im Folgenden nur noch die Entscheidungsregeln auf Basis des p-Wertes formulieren. Nun zur Durchführung der t-Tests in **R** für die Michelson-Daten:

Students Einstichproben-t-Test

```
> t.test(v, mu = 990)

   One-sample t-Test

data: v
t = -3.4524, df = 19,
p-value = 0.002669
alternative hypothesis: true mean
is not equal to 990
95 percent confidence interval:
 859.8931 958.1069
sample estimates:
 mean of x
       909
```

Die Funktion `t.test` führt Students t-Test über den Erwartungswert μ einer als normalverteilt angenommenen Population durch. Ihr erstes Argument erwartet die Daten (hier in `v`) aus jener Population. Dem Argument `mu` wird der hypothetisierte Erwartungswert μ_0 übergeben, zu lesen als H_0 : `mu` $= \mu_0$, hier mit $\mu_0 = 990$. Als Resultat liefert die Funktion den Namen der Datenquelle (`data`), den Wert der Teststatistik (`t`), die Anzahl ihrer Freiheitsgrade (`df`) und den p-Wert (`p-value`) des – hier – zweiseitigen Tests. Außerdem werden die Alternative (Voreinstellung: Zweiseitige Alternative H_1 : `mu` $\neq \mu_0$), das 95 %-Konfidenzintervall für μ (vgl. 134) und das arithmetische Mittel der Daten angegeben. Ohne Angabe für `mu` wird die Voreinstellung `mu` $= 0$ verwendet.

```
> t.test(v, mu = 990,
+ alternative = "greater")
....
95 percent confidence interval:
 868.4308      Inf
....

> t.test(v, mu = 990,
+ alter = "less")
....

> t.test(v, mu = 990,
+ conf.level = 0.90)
....
```

Über das Argument `alternative` lässt sich die Alternative spezifizieren: `alternative = "greater"` bedeutet H_1 : `mu` $> \mu_0$ und `alter = "less"` bedeutet H_1 : `mu` $< \mu_0$. (In diesen Fällen sind die angegebenen, hier aber zum Teil nicht gezeigten Konfidenzintervalle *einseitig*.) Die Argumentnamen dürfen – wie immer – so weit verkürzt werden, wie eindeutige Identifizierbarkeit gewährleistet ist. Dasselbe gilt für die zugewiesenen Werte `"less"` (max. Abk.: `"l"`) und `"greater"` (max. Abk.: `"g"`).)

Mit `conf.level` ist das Konfidenzniveau beliebig wählbar (Voreinstellung: 95 %).

8.4.2 Wilcoxons Vorzeichen-Rangsummentest

Wenn die Normalverteilungsannahme *nicht* gerechtfertigt erscheint, dann stehen verschiedene nicht-parametrische (Rang-)Verfahren für das Einstichproben-Lokationsproblem zur Verfügung, wie Wilcoxons Vorzeichentest, der lediglich eine stetige (aber sonst beliebige) Verteilung mit eindeutig bestimmtem Median voraussetzt, und Wilcoxons Vorzeichen-Rangsummentest, der eine stetige und um einen eindeutig bestimmten Median symmetrische Verteilung benötigt. Beispiele hierfür sind jede t-, Cauchy-, Doppelexponential-, $U(-a,a)$- oder $\Delta(-a,a)$-Verteilung. (Beispiele für stetige und symmetrische Verteilungen *ohne* eindeutig bestimmten Median lassen sich z. B. aus zwei Verteilungen mit kompakten Trägern leicht zusammensetzen. Memo: Falls der Erwartungswert einer symmetrischen Verteilung existiert, ist er gleich ihrem Median.)

Empfehlenswerte Referenzen zu dem Thema sind [16, Büning & Trenkler (1994)], [53, Hettmansperger (1984)] und [57, Hollander & Wolfe (1973)]. Bzgl. Tests für beliebige, also auch nicht stetige Verteilungen ist [15, Brunner & Munzel (2013)] mit einer hervorragenden Darstellung der Thematik sowie einer guten Mischung aus Theorie und durchgerechneten Beispielen sehr zu empfehlen.

Wir stellen hier Wilcoxons Vorzeichen-Rangsummentest näher vor (siehe z. B. [16], §4.4.3 oder [57], ch. 3, §1),welcher in **R** durch die Funktion `wilcox.test` realisiert ist. Auf Wilcoxons Vorzeichentest gehen wir in §8.4.3 kurz ein.

Annahmen: $X_1, \ldots, X_n$ sind unabhängig und nach einer stetigen Verteilungsfunktion $F(\cdot - \theta)$ verteilt, wobei F **symmetrisch** um 0 ist und 0 ihr eindeutig bestimmter Median ist (d. h., $F(\cdot - \theta)$ ist symmetrisch um θ und θ ist ihr eindeutig bestimmter Median sowie Erwartungswert, falls letzterer existiert).

Zu testen:

$$H_0 : \theta = \theta_0 \qquad \text{gegen} \qquad H_1 : \theta \neq \theta_0 \qquad \text{zum Signifikanzniveau } \alpha$$

bzw.

$$H_0' : \theta \overset{(\leq)}{\geq} \theta_0 \qquad \text{gegen} \qquad H_1' : \theta \overset{(>)}{<} \theta_0 \qquad \text{zum Signifikanzniveau } \alpha.$$

Teststatistik:

$$W_n^+ := \sum_{i=1}^{n} 1_{\{X_i - \theta_0 > 0\}} R(|X_i - \theta_0|),$$

wobei $R(z_i)$ der *Rang* von z_i unter $z_1, \ldots, z_n$ ist (d. h., $R(z_i) = j$, falls $z_i = z_{j:n}$). Die exakte Verteilung von Wilcoxons Vorzeichen-Rangsummenstatistik W_n^+ (auch Wilcoxons signierte Rangsummenstatistik genannt) ist unter $\theta = \theta_0$ bekannt, aber für großes n sehr aufwändig explizit zu berechnen. Einfache, kombinatorische Überlegungen zeigen, dass

$$\mathbb{E}[W_n^+] = \frac{n(n+1)}{4} \qquad \text{und} \qquad \mathrm{Var}(W_n^+) = \frac{n(n+1)(2n+1)}{24} \qquad \text{unter } \theta = \theta_0$$

sowie dass W_n^+ symmetrisch um ihren Erwartungswert verteilt ist. Des Weiteren ist nachweisbar, dass W_n^+ unter $\theta = \theta_0$ asymptotisch normalverteilt ist:

$$Z_n := \frac{W_n^+ - \mathbb{E}[W_n^+]}{\sqrt{\mathrm{Var}(W_n^+)}} \overset{n\to\infty}{\longrightarrow} \mathcal{N}(0,1) \quad \text{in Verteilung unter } \theta = \theta_0$$

Treten unter den $|X_i - \theta_0|$ exakte Nullwerte auf, so werden diese eliminiert und n wird auf die Anzahl der von 0 verschiedenen Werte gesetzt. Treten (z. B. aufgrund von Messungenauigkeiten in den Daten) Bindungen unter den (von 0 verschiedenen) $|X_i - \theta_0|$ auf, so wird für die Bestimmung ihrer Ränge die Methode der Durchschnittsränge verwendet, was den Erwartungswert von W_n^+ nicht, wohl aber ihre Varianz beeinflusst. In beiden Fällen wird in `wilcox.test` per Voreinstellung die Normalapproximation verwendet, aber mit einer im Nenner leicht modifizierten Statistik Z_n^{mod} anstelle von Z_n (worauf wir jedoch hier nicht näher eingehen). Z_n^{mod} hat dieselbe Asymptotik wie Z_n.

Entscheidungsregel für konkrete Daten $x_1, \ldots, x_n$ auf Basis des p-Wertes:

$$\text{Verwirf } H_0^{(\prime)} \text{ zum (evtl. approximat.) Niveau } \alpha \quad \Longleftrightarrow \quad p\text{-Wert} \leq \alpha,$$

wobei für die Berechnung des p-Wertes die folgende Fallunterscheidung zur Anwendung kommt, in welcher w_n der zu $x_1, \ldots, x_n$ realisierte Wert von W_n^+ sei:

1. Keine Bindungen, keine exakten Nullwerte und n „klein“ (was in `wilcox.test` $n < 50$ bedeutet):

$$p\text{-Wert} = \begin{cases} 2F_n^+(w_n) & \text{für} \quad H_1 : \theta \neq \theta_0, \text{wenn } w_n \leq n(n+1)/4 \\ 2(1 - F_n^+(w_n - 1)) & \text{“} \quad H_1 : \theta \neq \theta_0, \text{wenn } w_n > n(n+1)/4 \\ F_n^+(w_n) & \text{“} \quad H_1' : \theta < \theta_0 \\ 1 - F_n^+(w_n - 1) & \text{“} \quad H_1' : \theta > \theta_0 \end{cases}$$

   wobei F_n^+ die um $n(n+1)/4$ symmetrische Verteilungsfunktion der diskreten (!) Verteilung von W_n^+ unter $\theta = \theta_0$ ist.

2. Bindungen oder exakte Nullwerte oder n „groß“ (in `wilcox.test` $n \geq 50$):

   Hier wird Z_n^{mod} anstelle von Z_n verwendet und außerdem, um die Approximation an die asymptotische Normalverteilung für endliches n zu verbessern, im Zähler von Z_n^{mod} eine sogenannte Stetigkeitskorrektur vorgenommen. Der Zähler lautet dann

$$W_n^+ - \mathbb{E}[W_n^+] - c, \quad \text{wobei} \quad c = \begin{cases} \text{sign}(W_n^+ - \mathbb{E}[W_n^+]) \cdot 0.5 & \text{für} \quad H_1 : \theta \neq \theta_0 \\ 0.5 & \text{“} \quad H_1' : \theta > \theta_0 \\ -0.5 & \text{“} \quad H_1' : \theta < \theta_0 \end{cases}$$

   Wenn nun z_n^{mod} die konkrete Realisierung von Z_n^{mod} bezeichne, dann ist

$$p\text{-Wert} = \begin{cases} 2\left(1 - \Phi\left(\left|z_n^{mod}\right|\right)\right) & \text{für} \quad H_1 : \theta \neq \theta_0 \\ \Phi\left(z_n^{mod}\right) & \text{“} \quad H_1' : \theta < \theta_0 \\ 1 - \Phi\left(z_n^{mod}\right) & \text{“} \quad H_1' : \theta > \theta_0 \end{cases}$$

Beachte: Die Verteilung von W_n^+ unter H_0 ist sehr empfindlich gegenüber Abweichungen von der Symmetrieannahme (siehe [14, Brunner und Langer (1999)], §7.1.2).

Zur heuristischen Motivation der Funktionsweise von Wilcoxons Vorzeichen-Rangsummenstatistik sind in Abb. 27 zwei „prototypische“ Szenarien skizziert und danach ihre Wirkung auf W_n^+ beschrieben.

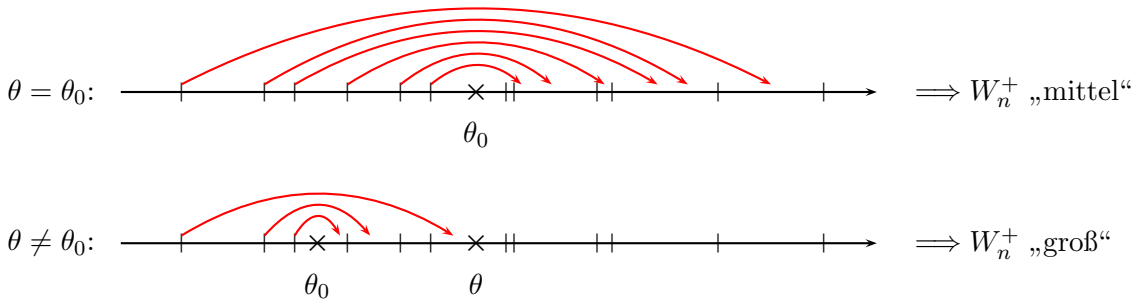

Abbildung 27: Zur Heuristik von Wilcoxons Vorzeichen-Rangsummenstatistik.

Durch die Betragsbildung $|X_i - \theta_0|$ werden die links von θ_0 liegenden X_i quasi auf die rechte Seite von θ_0 „gespiegelt“ (im Bild symbolisiert durch die Pfeile). Danach werden Rangzahlen für die (gespiegelten und ungespiegelten) Werte $|X_i - \theta_0|$ vergeben. W_n^+ „wertet“ jedoch nur die Rangzahlen derjenigen $|X_i - \theta_0|$, deren X_i auch schon *vor* der Spiegelung rechts von θ_0 gelegen haben, und quantifiziert somit den Grad der „Durchmischung“ der $|X_i - \theta_0|$ dergestalt, dass W_n^+ große Werte oder kleine Werte annimmt, wenn der Durchmischungsgrad gering ist (wie im unteren Szenario), was unter H_1 zu erwarten ist.

Wilcoxons Vorzeichen-Rangsummentest

```
> wilcox.test(v, mu = 990)

  Wilcoxon signed-rank test
  with continuity correction

data:  v
V = 22.5, p-value = 0.00213
alternative hypothesis: true
location is not equal to 990

Warning message:
In wilcox.test.default(v,
  mu = 990) : cannot compute
  exact p-value with ties
```

`wilcox.test` führt in dieser Form Wilcoxons Vorzeichen-Rangsummentest für den Median θ einer als stetig und symmetrisch um θ verteilt angenommenen Population durch, wozu ihr erstes Argument die Daten (hier in `v`) aus jener Population erwartet. An `mu` wird der hypothetisierte Median θ_0 übergeben, zu lesen als H_0 : `mu` $= \theta_0$. Zurückgeliefert werden der Name der Datenquelle, der Wert der Teststatistik W_n^+ (`V`), der p-Wert (`p-value`) des zweiseitigen Tests und die Alternative (Voreinstellung: Zweiseitige Alternative H_1 : `mu` $\neq \theta_0$; “`true location`” meint θ). Der p-Wert wurde – ohne Hinweis! – über die approximativ normalverteilte Teststatistik Z_n^{mod} berechnet, da in den Michelson-Daten Bindungen auftreten, und dies (per Voreinstellung) `with continuity correction`. Immerhin wird eine entsprechende `Warning message` ausgegeben.
Ohne Angabe für `mu` würde die Voreinstellung `mu` $= 0$ verwendet. Durch das nicht gezeigte Argument `alternative` von `wilcox.test` ließe sich (wie bei `t.test`) die Alternative spezifizieren: `alternative = "greater"` ist H_1 : `mu` $> \theta_0$ und `alter = "less"` ist H_1 : `mu` $< \theta_0$.

```
> set.seed(2021)
> wilcox.test(jitter(v),
+ mu = 990)

  Wilcoxon signed-rank test

data:  jitter(v)
V = 23, p-value = 0.001209
alternative hypothesis:
true mu is not equal to 990
```

Brechen wir (hier lediglich zur Veranschaulichung!) mithilfe von `jitter` die Bindungen in den Daten künstlich zufällig auf, erhalten wir die Ausgabe der „nicht-approximativen“ Version von Wilcoxons Vorzeichen-Rangsummentest: Den Wert der Teststatistik W_n^+ (`V`) und den exakten p-Wert des zweiseitigen Tests sowie die Alternative. (Hinweise: `jitter` ist sehr „zurückhaltend“ beim Aufbrechen von Bindungen; siehe ihre Hilfeseite. `set.seed` garantiert die Reproduzierbarkeit.)

Hinweise:

- Mit dem nicht gezeigten Argument `conf.int = TRUE` erhielte man den Hodges–Lehmann–Schätzer für den „Pseudomedian“, der im hier vorliegenden Fall einer symmetrischen Verteilung gleich ihrem Median θ ist, ergänzt um ein Konfidenzintervall (das ein exaktes ist, wenn auch der p-Wert exakt ist, und ansonsten ein approximatives). (Siehe die Hilfeseite und für mathematische Details siehe z. B. [53, Hettmansperger (1984), Abschnitt 2.3] oder [57, Hollander & Wolfe (1973), Abschnitte 3.2 und 3.3].)
- Das **R**-Paket `exactRankTests`, das allerdings “no longer under development” ist, bietet durch seine Funktion `wilcox.exact` auch für Stichproben *mit* Bindungen die Berechnung exakter p-Werte des Wilcoxon-Rangsummentests. Stattdessen und auch unabhängig davon gibt es das **R**-Paket `coin`, dessen Funktion `wilcox_test` exakte und approximative *bedingte* p-Werte der Wilcoxon-Tests liefert.

8.4.3 Wilcoxons Vorzeichentest

Wilcoxons Vorzeichentest setzt, wie zu Beginn des vorherigen Paragraphen bereits erwähnt, lediglich eine stetige (aber sonst beliebige) Verteilung mit eindeutig bestimmtem Median voraus. Er ist in **R** nicht durch eine eigene Funktion realisiert, da er auf einen einfachen Binomialtest hinausläuft, den wir im Abschnitt **??** „Einstichprobenprobleme im Binomialmodell“ ausführlich behandeln. Daher fassen wir uns hier kurz:

Annahmen: $X_1, \ldots, X_n$ sind unabhängig und nach einer stetigen Verteilungsfunktion $F(\cdot - \theta)$ verteilt, wobei 0 der eindeutig bestimmte Median von F ist (und damit θ der eindeutig bestimmte Median von $F(\cdot - \theta)$).

Zu testen:

$$H_0 : \theta = \theta_0 \qquad \text{gegen} \qquad H_1 : \theta \neq \theta_0 \qquad \text{zum Signifikanzniveau } \alpha$$

bzw.

$$H_0' : \theta \overset{(\leq)}{\geq} \theta_0 \qquad \text{gegen} \qquad H_1' : \theta \overset{(>)}{<} \theta_0 \qquad \text{zum Signifikanzniveau } \alpha.$$

Teststatistik:

$$W_n := \sum_{i=1}^{n} 1_{\{X_i - \theta_0 > 0\}}$$

Offenbar sind die Indikatoren $1_{\{X_i - \theta_0 > 0\}}$ unter $\theta = \theta_0$ unabhängig und identisch verteilte Bernoulli-Variablen zum Parameter $p \equiv \mathbb{P}\left(1_{\{X_i - \theta_0 > 0\}}\right) = \mathbb{P}(X_i > \theta_0) = 1/2$ (weil ja θ_0 der Median ist), sodass W_n unter $\theta = \theta_0$ eine Binomial-Verteilung zu den Parametern n und $p = 1/2$ besitzt. Für alles Weitere verweisen wir auf den schon erwähnten Abschnitt **??** „Einstichprobenprobleme im Binomialmodell“.

8.5 Zweistichproben-Lokations- und Skalenprobleme

Hier werden Hypothesen über Lokations- bzw. Skalen- (also Streuungs-) *Unterschiede* zwischen zwei zugrunde liegenden Populationen getestet. Unter der Normalverteilungsannahme und der Annahme, dass die (in der Regel unbekannten!) Varianzen in den beiden Populationen gleich sind, wird für den Test auf Erwartungswertegleichheit Students Zweistichproben-t-Test verwendet. Bei verschiedenen Varianzen ist Welchs Modifikation des t-Tests angezeigt. Um Varianzengleichheit zu testen, gibt es einen F-Test. In Fällen *nicht* normal-, aber stetig verteilter Daten sollten wiederum nicht-parametrische (Rang-)Verfahren verwendet werden, namentlich Wilcoxons Rangsummentest auf Lokationsunterschied zwischen den beiden Populationen (und z. B. die – hier nicht vorgestellten – Tests von Levene, von Ansari und von Mood auf Skalenunterschied). Für einen allgemeineren, nicht-parametrischen Vergleich zweier Populationen siehe die Bemerkung am Ende von §8.5.4 auf Seite 150.

8.5.1 Der Zweistichproben-F-Test für der Vergleich zweier Varianzen

Er ist implementiert in der Funktion `var.test`, aber auch hier zur Erinnerung zunächst wieder etwas Theorie:

Annahmen: $X_1, \ldots, X_n$ sind unabhängig $\mathcal{N}(\mu_X, \sigma_X^2)$-verteilt und unabhängig davon sind $Y_1, \ldots, Y_m$ unabhängig $\mathcal{N}(\mu_Y, \sigma_Y^2)$-verteilt mit unbekannten Varianzen σ_X^2 und σ_Y^2.

Zu testen:

$$H_0 : \frac{\sigma_X^2}{\sigma_Y^2} = r_0 \qquad \text{gegen} \qquad H_1 : \frac{\sigma_X^2}{\sigma_Y^2} \neq r_0 \qquad \text{zum Signifikanzniveau } \alpha$$

bzw.

$$H_0' : \frac{\sigma_X^2}{\sigma_Y^2} \overset{(\leq)}{\geq} r_0 \qquad \text{gegen} \qquad H_1' : \frac{\sigma_X^2}{\sigma_Y^2} \overset{(>)}{<} r_0 \qquad \text{zum Signifikanzniveau } \alpha.$$

Teststatistik:

$$\frac{1}{r_0}\frac{\hat\sigma^2_{Xn}}{\hat\sigma^2_{Ym}} \sim F_{n-1,m-1} \qquad \begin{array}{l}\text{unter } H_0 \text{ bzw. unter } \sigma^2_X/\sigma^2_Y = r_0,\\ \text{dem „Rand“ von } H_0',\end{array}$$

wobei $\hat\sigma^2_{Xn} = (n-1)^{-1}\sum_{i=1}^n (X_i - \bar X_n)^2$ der Schätzer für σ^2_X ist und für σ^2_Y analoges gilt.

Entscheidungsregel für konkrete Daten $x_1,\ldots,x_n$ und $y_1,\ldots,y_m$ auf Basis des p-Wertes:

$$\text{Verwirf } H_0^{(')} \text{ zum Niveau } \alpha \quad\Longleftrightarrow\quad p\text{-Wert} \le \alpha,$$

wobei mit dem mit $1/r_0$ skalierten Quotienten der Stichprobenvarianzen der x- bzw. y-Daten $q^2_{n,m} := 1/r_0 \cdot s^2_{Xn}/s^2_{Ym}$ gilt:

$$p\text{-Wert} = \begin{cases} 2\cdot\min\left\{F_{n-1,m-1}\left(q^2_{n,m}\right), 1-F_{n-1,m-1}\left(q^2_{n,m}\right)\right\} & \text{für} \quad H_1 : \sigma^2_X/\sigma^2_Y \neq r_0 \\ F_{n-1,m-1}\left(q^2_{n,m}\right) & \text{“} \quad H_1' : \sigma^2_X/\sigma^2_Y < r_0 \\ 1-F_{n-1,m-1}\left(q^2_{n,m}\right) & \text{“} \quad H_1' : \sigma^2_X/\sigma^2_Y > r_0 \end{cases}$$

worin $F_{s,t}$ die (übrigens *nicht* um 1 symmetrische) Verteilungsfunktion der F-Verteilung zu s Zähler- und t Nenner-Freiheitsgraden ist. (Memos: $(n-1)\hat\sigma^2_n/\sigma^2 \sim \chi^2_{n-1}$ (s. §8.2.2) und für $\chi^2_r \perp_{\mathbb{P}} \chi^2_s$ gilt $(\chi^2_r/r)/(\chi^2_s/s) \sim F_{r,s}$. Außerdem ist $\mathbb{E}[F_{s,t}] = t/(t-2)$ für $t > 2$, also hat obige Teststatistik unter H_0 den Erwartungswert $(m-1)/(m-3)$ für $m > 3$.)

Als **Beispiel** verwenden wir Daten von [90, Snedecor & Cochran (1980)] über Gewichtszuwächse in zwei verschiedenen Würfen von Ratten, die unterschiedlich proteinhaltige Nahrung erhielten:

```
> protreich <- c(134, 146, 104, 119, 124, 161, 107, 83, 113, 129, 97,123)
> protarm <- c(70, 118, 101, 85, 107, 132, 94)
```

Eine explorative Datenanalyse (nicht gezeigt) lässt an der Normalverteiltheit der Daten nicht zweifeln, sodass der F-Test für der Vergleich der zwei Varianzen zulässig erscheint:

F-Test für der Vergleich zweier Varianzen

```
> var.test(protreich, protarm,
+ ratio = 1)

  F test to compare two variances

data:  protreich and protarm
F = 1.0755, num df = 11,
denom df = 6, p-value = 0.9788
alternative hypothesis: true ratio
of variances is not equal to 1
95 percent confidence interval:
 0.198811 4.173718
sample estimates:
ratio of variances
           1.07552
```

`var.test` führt so den F-Test für $H_0 : \sigma^2_x/\sigma^2_y = r_0$ für zwei als normalverteilt angenommene Populationen aus, wobei ihre zwei ersten Argumente die Datenvektoren erwarten. An `ratio` wird der für σ^2_x/σ^2_y hypothetisierte Quotientenwert r_0 übergeben, zu lesen als H_0 : `ratio` $= r_0$. Die (hier explizit gezeigte) Voreinstellung lautet `ratio = 1` und testet damit also auf Gleichheit der Varianzen. Resultate: Namen der Datenquellen, Wert der Teststatistik (`F`), Freiheitsgrade ihres Zählers und Nenners (`num df`, `denom df`), p-Wert (`p-value`). Außerdem: Alternative (Voreinstellung: Zweiseitig, H_1 : `ratio` $\neq r_0$), 95 %-Konfidenzintervall und Schätzwert für σ^2_x/σ^2_y.
Mit `conf.level` und `alternative` lassen sich Konfidenzniveau bzw. Alternative spezifizieren: `alternative = "greater"` ist H_1 : `ratio` $> r_0$; `alter = "less"` ist H_1 : `ratio` $< r_0$.

Ergebnis: Die Varianzen unterscheiden sich nicht signifikant; somit kann der (im Folgenden vorgestellte) Zweistichproben-t-Test auf obige Daten angewendet werden, wobei aber bedacht werden sollte, dass in obigem F-Test natürlich ein Fehler 2. Art aufgetreten sein könnte ... und wir ja nur feststellen können, dass wir kein stichhaltiges Argument gegen H_0 gefunden haben (nicht, dass wir H_0 „bewiesen“ haben).

8.5.2 Students Zweistichproben-t-Test bei unbekannten, aber gleichen Varianzen

Annahmen: $X_1, \ldots, X_n$ sind unabhängig $\mathcal{N}(\mu_X, \sigma^2)$-verteilt und unabhängig davon sind $Y_1, \ldots, Y_m$ unabhängig $\mathcal{N}(\mu_Y, \sigma^2)$-verteilt mit unbekanntem, aber gleichem σ^2.

Zu testen:

$$H_0 : \mu_X - \mu_Y = \Delta_0 \qquad \text{gegen} \qquad H_1 : \mu_X - \mu_Y \neq \Delta_0 \qquad \text{zum Signifikanzniveau } \alpha$$

bzw.

$$H_0' : \mu_X - \mu_Y \overset{(\leq)}{\geq} \Delta_0 \qquad \text{gegen} \qquad H_1' : \mu_X - \mu_Y \overset{(>)}{<} \Delta_0 \qquad \text{zum Signifikanzniveau } \alpha.$$

Teststatistik:

$$\frac{\bar{X}_n - \bar{Y}_m - \Delta_0}{\sqrt{\left(\frac{1}{n} + \frac{1}{m}\right) \frac{(n-1)\hat{\sigma}_{Xn}^2 + (m-1)\hat{\sigma}_{Ym}^2}{n+m-2}}} \sim t_{n+m-2}$$

unter H_0 bzw. unter $\mu_X - \mu_Y = \Delta_0$, dem „Rand“ von H_0'.

Entscheidungsregel für konkrete Daten $x_1, \ldots, x_n$ und $y_1, \ldots, y_m$ auf Basis des p-Wertes:

$$\text{Verwirf } H_0^{(')} \text{ zum Niveau } \alpha \quad \Longleftrightarrow \quad p\text{-Wert} \leq \alpha,$$

wobei mit s_{Xn}^2 und s_{Ym}^2 als Stichprobenvarianzen der x- bzw. y-Daten und

$$t_{n,m}^* := \frac{\bar{x}_n - \bar{y}_m - \Delta_0}{\sqrt{\left(\frac{1}{n} + \frac{1}{m}\right) \frac{(n-1)s_{Xn}^2 + (m-1)s_{Ym}^2}{n+m-2}}}$$

als Realisierung der obigen Teststatistik gilt:

$$p\text{-Wert} = \begin{cases} 2\left(1 - F_{t_{n+m-2}}\left(|t_{n,m}^*|\right)\right) & \text{für} \quad H_1 : \mu_X - \mu_Y \neq \Delta_0 \\ F_{t_{n+m-2}}\left(t_{n,m}^*\right) & \text{“} \quad H_1' : \mu_X - \mu_Y < \Delta_0 \\ 1 - F_{t_{n+m-2}}\left(t_{n,m}^*\right) & \text{“} \quad H_1' : \mu_X - \mu_Y > \Delta_0 \end{cases}$$

wobei F_{t_k} die (um 0 symmetrische) Verteilungsfunktion der t_k-Verteilung ist. (Siehe z. B. [51], Abschnitt 34.20 in Verbindung mit [52], S. 262 f, für mathematische Details.)

Doch nun die praktische Durchführung von Students Zweistichproben-t-Test (ebenfalls) mit Hilfe der Funktion `t.test` für die Daten von Snedecor und Cochran:

Students Zweistichproben-t-Test

```
> t.test(protreich, protarm,
+ mu = 0, var.equal = TRUE)

  Two-Sample t-Test

data:  protreich and protarm
t = 1.8914, df = 17,
p-value = 0.07573
alternative hypothesis:
true difference in means is not
equal to 0
95 percent confidence interval:
  -2.193679  40.193679
sample estimates:
 mean of x mean of y
       120       101
```

In dieser Form führt `t.test` Students t-Test auf Erwartungswertegleichheit für zwei mit gleichen Varianzen als normalverteilt angenommene Populationen aus, wobei ihre zwei ersten Argumente die Datenvektoren erwarten. `mu` erhält den für $\mu_X - \mu_Y$ hypothetisierten Differenzwert Δ_0, zu lesen als H_0 : `mu` $= \Delta_0$ (hier explizit gezeigt mit der Voreinstellung `mu = 0`, also für $H_0 : \mu_X - \mu_Y = 0$). Beachte: `var.equal = TRUE` ist *nicht* die Voreinstellung. Resultate: Namen der Datenquellen, Wert der Teststatistik (`t`), ihre Freiheitsgrade (`df`), p-Wert (`p-value`). Außerdem: Alternative (Voreinstellung: Zweiseitig, H_1 : `mu` $\neq \Delta_0$), 95 %-Konfidenzintervall für die Differenz $\mu_X - \mu_Y$ und die Schätzer für beide Erwartungswerte. Konfidenzniveau und Alternative wären mit `conf.level` bzw. `alternative` spezifizierbar: `alternative = "greater"` ist H_1 : `mu` $> \Delta_0$; `alter = "less"` ist H_1 : `mu` $< \Delta_0$.

8.5.3 Die Welch-Modifikation des Zweistichproben-t-Tests

Die bisher im Normalverteilungsmodell betrachteten Ein- und Zweistichprobentests halten das Niveau α exakt ein. Für das Zweistichproben-Lokationsproblem mit unbekannten und möglicherweise *verschiedenen* Varianzen ist kein solcher exakter Test bekannt. Es handelt sich dabei um das sogenannte Behrens-Fisher-Problem, wofür es immerhin einen approximativen Niveau-α-Test gibt: Welchs Modifikation des Zweistichproben-t-Tests. Auch dieser Test ist durch `t.test` verfügbar und ist sogar die Voreinstellung. (Für Hintergrundinfos und Referenzen zu mathematischen Details siehe z. B. [49], §7.4.4.2.)

Annahmen: $X_1, \ldots, X_n$ sind unabhängig $\mathcal{N}(\mu_X, \sigma_X^2)$-verteilt und unabhängig davon sind $Y_1, \ldots, Y_m$ unabhängig $\mathcal{N}(\mu_Y, \sigma_Y^2)$-verteilt mit unbekannten μ_X, μ_Y, σ_X^2 und σ_Y^2.

Zu testen:

$$H_0 : \mu_X - \mu_Y = \Delta_0 \qquad \text{gegen} \qquad H_1 : \mu_X - \mu_Y \neq \Delta_0 \qquad \text{zum Signifikanzniveau } \alpha$$

bzw.

$$H_0' : \mu_X - \mu_Y \overset{(\leq)}{\geq} \Delta_0 \qquad \text{gegen} \qquad H_1' : \mu_X - \mu_Y \overset{(>)}{<} \Delta_0 \qquad \text{zum Signifikanzniveau } \alpha.$$

Teststatistik:

$$\frac{\bar{X}_n - \bar{Y}_m - \Delta_0}{\sqrt{\frac{\hat{\sigma}_{Xn}^2}{n} + \frac{\hat{\sigma}_{Ym}^2}{m}}} \overset{approx.}{\sim} t_\nu \qquad \text{mit} \quad \nu = \left\lfloor \frac{\left(\frac{\hat{\sigma}_{Xn}^2}{n} + \frac{\hat{\sigma}_{Ym}^2}{m}\right)^2}{\frac{1}{n-1}\left(\frac{\hat{\sigma}_{Xn}^2}{n}\right)^2 + \frac{1}{m-1}\left(\frac{\hat{\sigma}_{Ym}^2}{m}\right)^2} \right\rfloor$$

unter H_0 bzw. unter $\mu_X - \mu_Y = \Delta_0$, dem „Rand“ von H_0'.

Entscheidungsregel für konkrete Daten $x_1, \ldots, x_n$ und $y_1, \ldots, y_m$ auf Basis des p-Wertes:

$$\text{Verwirf } H_0^{(\prime)} \text{ zum approximat. Niveau } \alpha \quad \Longleftrightarrow \quad p\text{-Wert} \leq \alpha,$$

wobei mit s_{Xn}^2 und s_{Ym}^2 als Stichprobenvarianzen der x- bzw. y-Daten sowie mit $t_{n,m}^*$ und ν^* als Realisierungen der obigen Teststatistik bzw. der (ganzzahligen) Freiheitsgrade nach obiger Formel gilt:

$$p\text{-Wert} = \begin{cases} 2\left(1 - F_{t_{\nu^*}}\left(|t_{n,m}^*|\right)\right) & \text{für} \quad H_1 : \mu_X - \mu_Y \neq \Delta_0 \\ F_{t_{\nu^*}}\left(t_{n,m}^*\right) & \text{“} \quad H_1' : \mu_X - \mu_Y < \Delta_0 \\ 1 - F_{t_{\nu^*}}\left(t_{n,m}^*\right) & \text{“} \quad H_1' : \mu_X - \mu_Y > \Delta_0 \end{cases}$$

wobei F_{t_k} die (um 0 symmetrische) Verteilungsfunktion der t_k-Verteilung ist.

Welch-Modifikation des Zweistichproben-t-Tests

```
> t.test(protreich, protarm)

  Welch Two-Sample t-Test

data: protreich and protarm
t= 1.9107, df= 13.082, p-value= 0.0782
alternative hypothesis: true dif-
ference in means is not equal to 0
95 percent confidence interval:
  -2.469073  40.469073
sample estimates:
 mean of x mean of y
       120       101
```

Aufgrund der (nicht gezeigten) Voreinstellung `var.equal = FALSE` und `mu = 0` führt `t.test` so Welchs t-Test auf Erwartungswertegleichheit für zwei mit möglicherweise ungleichen Varianzen als normalverteilt angenommenen Populationen durch (d. h., den Test für $H_0 : \texttt{mu} = 0$, also $H_0 : \mu_X - \mu_Y = \Delta_0$ mit $\Delta_0 = 0$).
Alles Weitere inklusive der Resultate: Analog zum Zweistichproben-t-Test für gleiche Varianzen des vorherigen §8.5.2. Allerdings wird die Anzahl der Freiheitsgrade (`df`) *nicht* abgerundet angegeben, d. h., der Wert ist das Argument von $\lfloor \cdot \rfloor$ in der obigen Definition von ν.

8.5.4 Wilcoxons Rangsummentest (Mann-Whitney U-Test)

Wäre die Normalverteilungsannahme für obige Daten nicht gerechtfertigt, wohl aber die Annahme zweier stetiger und „typgleicher“, aber sonst beliebiger Verteilungen, käme als nichtparametrisches Pendant Wilcoxons Rangsummentest infrage (der äquivalent zum Mann-Whitney U-Test ist; siehe auch die zweite Bemerkung am Ende dieses Abschnitts, Seite 149). Dieser Test wird durch `wilcox.test` zur Verfügung gestellt. (Für mathematische Details siehe z. B. [16, Büning & Trenkler (1994), §5.4.2, S. 132] oder [57, Hollander & Wolfe (1973), Abschnitt 4.1].)

Annahmen: $X_1, \ldots, X_n$ sind unabhängig und nach einer stetigen Verteilungsfunktion F verteilt und unabhängig davon sind $Y_1, \ldots, Y_m$ unabhängig nach der (gleichartigen/formgleichen!) Verteilungsfunktion $F(\cdot - \theta)$ verteilt (d. h., die Y_j können sich als aus F stammende $\tilde{X}_j$s vorgestellt werden, die um θ zu $Y_j := \tilde{X}_j + \theta$ geshiftet wurden; θ ist also nicht notwendigerweise ein Median o. Ä., sondern „lediglich“ die Verschiebung der Y-Verteilung gegenüber der X-Verteilung).

Zu testen:

$$H_0 : \theta = \theta_0 \qquad \text{gegen} \qquad H_1 : \theta \neq \theta_0 \qquad \text{zum Signifikanzniveau } \alpha$$

bzw.

$$H_0' : \theta \overset{(\leq)}{\geq} \theta_0 \qquad \text{gegen} \qquad H_1' : \theta \overset{(>)}{<} \theta_0 \qquad \text{zum Signifikanzniveau } \alpha.$$

Teststatistik:

$$W_{n,m} := \sum_{i=1}^{n} R(X_i + \theta_0) - \frac{n(n+1)}{2},$$

wobei $R(X_i + \theta_0)$ der *Rang* von $X_i + \theta_0$ unter den $N := n + m$ „gepoolten“ Werten $X_1 + \theta_0, \ldots, X_n + \theta_0, Y_1, \ldots, Y_m$ ist. Die Verteilung von $W_{n,m}$ ist unter $\theta = \theta_0$ bekannt. Kombinatorische Überlegungen zeigen, dass

$$\mathbb{E}[W_{n,m}] = \frac{nm}{2} \qquad \text{und} \qquad \mathrm{Var}(W_{n,m}) = \frac{nm(N+1)}{12} \qquad \text{unter } \theta = \theta_0$$

ist und dass für $W'_{n,m} := \sum_{j=1}^{m} R(Y_j) - m(m+1)/2$ gilt $W_{n,m} + W'_{n,m} = nm$. Des Weiteren ist $W_{n,m}$ unter $\theta = \theta_0$ asymptotisch normalverteilt:

$$Z_{n,m} := \frac{W_{n,m} - \mathbb{E}[W_{n,m}]}{\sqrt{\mathrm{Var}(W_{n,m})}} \overset{n,m\to\infty}{\longrightarrow} \mathcal{N}(0,1) \quad \text{unter } \theta = \theta_0, \text{ falls } \frac{n}{m} \to \lambda \notin \{0, \infty\}$$

Treten Bindungen zwischen den $X_i + \theta_0$ und Y_j auf, so wird die Methode der Durchschnittsränge verwendet, was den Erwartungswert von $W_{n,m}$ nicht, wohl aber ihre Varianz beeinflusst. In diesem Fall wird in `wilcox.test` per Voreinstellung die Normalapproximation verwendet, aber mit einer im Nenner leicht modifizierten Statistik $Z_{n,m}^{mod}$ anstelle von $Z_{n,m}$ (worauf wir hier jedoch nicht näher eingehen). $Z_{n,m}^{mod}$ hat dieselbe Asymptotik wie $Z_{n,m}$.

Entscheidungsregel für konkrete Daten $x_1, \ldots, x_n$ und $y_1, \ldots, y_m$ auf Basis des p-Wertes:

$$\text{Verwirf } H_0^{(')} \text{ zum (evtl. approximat.) Niveau } \alpha \quad \Longleftrightarrow \quad p\text{-Wert} \leq \alpha,$$

wobei für die Berechnung des p-Wertes die folgende Fallunterscheidung zur Anwendung kommt, in welcher $w_{n,m}$ der realisierte Wert von $W_{n,m}$ sei:

1. Keine Bindungen und „kleine“ Stichprobenumfänge (in `wilcox.test` für $n < 50$ und $m < 50$):

$$p\text{-Wert} = \begin{cases} 2F_{n,m}(w_{n,m}) & \text{für} \quad H_1 : \theta \neq \theta_0, \text{wenn } w_{n,m} \leq nm/2 \\ 2(1 - F_{n,m}(w_{n,m} - 1)) & \text{“} \quad H_1 : \theta \neq \theta_0, \text{wenn } w_{n,m} > nm/2 \\ F_{n,m}(w_{n,m}) & \text{“} \quad H_1' : \theta < \theta_0 \\ 1 - F_{n,m}(w_{n,m} - 1) & \text{“} \quad H_1' : \theta > \theta_0 \end{cases}$$

   wobei $F_{n,m}$ die um $nm/2$ symmetrische Verteilungsfunktion der diskreten (!) Verteilung von $W_{n,m}$ unter $\theta = \theta_0$ zu den Stichprobenumfängen n und m ist.

2. Bindungen oder „große“ Stichprobenumfänge (in `wilcox.test` für $n \geq 50$ oder $m \geq 50$):

   Hier wird $Z_{n,m}^{mod}$ anstelle von $Z_{n,m}$ verwendet und außerdem eine Stetigkeitskorrektur (so wie bei Wilcoxons Vorzeichen-Rangsummentest auf Seite 142, nur mit dem hiesigen $W_{n,m}$ an Stelle des dortigen W_n^+), um die Asymptotik von $Z_{n,m}^{mod}$ zu verbessern. Wenn nun $z_{n,m}^{mod}$ die konkrete Realisierung von $Z_{n,m}^{mod}$ bezeichne, dann ist

$$p\text{-Wert} = \begin{cases} 2\left(1 - \Phi\left(\left|z_{n,m}^{mod}\right|\right)\right) & \text{für} \quad H_1 : \theta \neq \theta_0 \\ \Phi\left(z_{n,m}^{mod}\right) & \text{“} \quad H_1' : \theta < \theta_0 \\ 1 - \Phi\left(z_{n,m}^{mod}\right) & \text{“} \quad H_1' : \theta > \theta_0 \end{cases}$$

Bemerkung: Ist $\theta > 0$, so ist die Verteilungsfunktion $F(\cdot - \theta)$ von Y gegenüber der Verteilungsfunktion F von X nach rechts verschoben, denn es ist $F(x - \theta) \leq F(x)$ für alle $x \in \mathbb{R}$. Man sagt dann, Y ist stochastisch größer als X (kurz: $Y \geq_{\mathbb{P}} X$). Analog ist im Fall $\theta < 0$ die Verteilungsfunktion $F(\cdot - \theta)$ gegenüber F nach links verschoben und man nennt Y stochastisch kleiner als X (kurz: $Y \leq_{\mathbb{P}} X$). Schließlich bedeutet $\theta \neq 0$, dass $F(\cdot - \theta)$ gegenüber F entweder nach rechts oder nach links verschoben ist, d. h., dass entweder $Y \geq_{\mathbb{P}} X$ oder $Y \leq_{\mathbb{P}} X$ gilt.

Zur heuristischen Motivation der Funktionsweise von Wilcoxons Rangsummenstatistik sind in Abb. 28 für den Fall $\theta_0 = 0$ zwei „prototypische“ Datenszenarien dargestellt, wobei X_i u.i.v. $\sim F$ (markiert durch „$\times$“) und unabhängig davon Y_j u.i.v. $\sim F(\cdot - \theta)$ (markiert durch „$\Diamond$“).

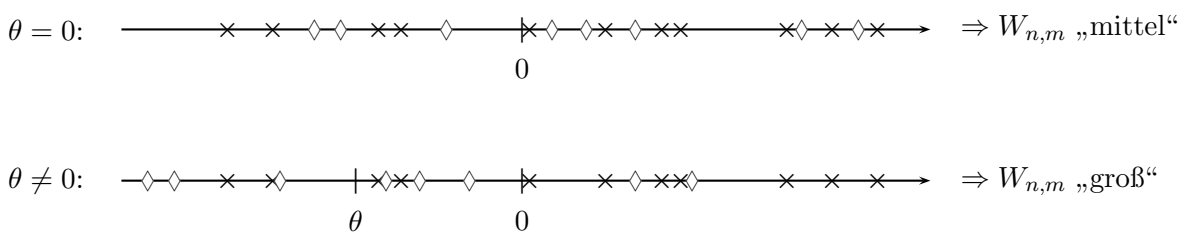

Abbildung 28: Zur Heuristik von Wilcoxons Rangsummenstatistik.

Begründung: $\theta < 0 \;\Rightarrow\; F(x - \theta) \geq F(x) \;\Rightarrow\; Y \leq_{\mathbb{P}} X \;\Rightarrow$ die Y_j haben tendenziell kleinere Ränge als die $X_i \;\Rightarrow\; W_{n,m}$ tendenziell „groß“. Für $\theta \geq 0$ analog umgekehrt.

$W_{n,m}$ misst, wie gut die $(X_i + \theta_0)$- und die Y_j-Daten durchmischt sind: Je schlechter die Durchmischung, desto kleiner oder größer ist $W_{n,m}$, wobei kleine Werte auftreten, wenn die $(X_i + \theta_0)$-Gruppe gegenüber der Y_j-Gruppe tendenziell nach links verschoben ist (und somit die $(X_i + \theta_0)$-Ränge tendenziell kleiner als die Y_j-Ränge sind). Große Werte treten im umgekehrten Fall auf (wie im unteren Szenario der Skizze in Abb. 28; Memo: Dort ist $\theta_0 = 0$).

Auf der nächsten Seite wird die Umsetzung des Tests durch die Funktion `wilcox.test` an einem Beispiel erläutert. Aus Platzgründen ziehen wir weitere Bemerkungen hierher vor:

Weitere Bemerkungen:

- Es spielt keine Rolle, welche Stichprobe das erste Argument von `wilcox.test` ist (sofern man selbst auf das korrekte Vorzeichen beim hypothetisierten θ_0 achtet!), da für $W_{n,m}$ und $W_{n,m}'$ (die Summenstatistik für die Ränge der $Y_1, \ldots, Y_m$) stets $W_{n,m} + W_{n,m}' = nm$ gilt, sodass Wilcoxons Rangsummentest für $W_{n,m}$ und für $W_{n,m}'$ äquivalent ist.
- Faktisch ist in `wilcox.test` der zu Wilcoxons Rangsummentest äquivalente U-Test von Mann und Whitney realisiert. Tatsächlich ist die Mann-Whitney-Teststatistik U genau so definiert wie das hiesige $W_{n,m}$. Bei der eigentlichen Wilcoxon Rangsummenstatistik $W := \sum_{i=1}^{n} R(X_i +$

θ_0) wird nämlich nicht $n(n+1)/2$ abgezogen. Aber damit unterscheiden sich W und U (= $W_{n,m}$) nur durch die Konstante $n(n+1)/2$, sodass beide Verfahren äquivalent sind.

- Mit dem nicht gezeigten Argument `conf.int = TRUE` erhielte man den Hodges–Lehmann–Schätzer für den Shift θ, ergänzt um ein Konfidenzintervall (das ein exaktes ist, wenn auch der p-Wert exakt ist, und ansonsten ein approximatives). (Siehe die Hilfeseite und für mathematische Details siehe z. B. [57, Hollander & Wolfe (1973), Abschnitte 4.2 und 4.3].)
- (Wörtl. Wiederholung des 2. Hinweises auf S. 143 unten.) Das **R**-Paket `exactRankTests`, das allerdings "no longer under development" ist, bietet durch seine Funktion `wilcox.exact` auch für Stichproben *mit* Bindungen die Berechnung exakter p-Werte des Wilcoxon-Rangsummentests. Stattdessen und auch unabhängig davon gibt es das **R**-Paket `coin`, dessen Funktion `wilcox_test` exakte und approximative *bedingte* p-Werte der Wilcoxon-Tests liefert.

Wilcoxons Rangsummentest

```
> wilcox.test(protreich, protarm)

  Wilcoxon rank sum test with
  continuity correction

data:  protreich and protarm
W = 62.5, p-value = 0.09083
alternative hypothesis: true
location shift is not equal to 0

Warning message:
In wilcox.test.default(protreich,
protarm) : cannot compute exact
           p-value with ties
```

Achtung: Mit "`true location shift`" ist hier die Verschiebung der X- ggü. der Y-Verteilung gemeint, wohingegen das θ auf Seite 148 die Verschiebung der Y- ggü. der X-Verteilung ist. "`true location shift`" bezieht sich also auf $-\theta$ und somit ist (das hier nicht gezeigte Argument) `mu` $= -\theta$!

So führt `wilcox.test` Wilcoxons Rangsummentest auf einen Lokations-Shift zwischen den als stetig und gleichartig angenommenen Verteilungen zweier Populationen durch, wobei das erste Argument den X- und das zweite den Y-Datenvektor erhält. Das nicht gezeigte Argument `mu` erwartet den hypothetisierten Shiftwert der X- ggü. der Y-Verteilung (also $-\theta_0$ gemäß des „Achtung" links). Voreinstellung ist `mu = 0`, also wird hier H_0 : `mu` $= 0$ getestet. Resultate: Die Datenquellen, der errechnete Wert der Teststatistik $W_{n,m}$ (`W`) und sein p-Wert (`p-value`) sowie die Alternative (Voreinstellung: Zweiseitig, H_1 : `mu` $\neq -\theta_0$). Jedoch wurde aufgrund einer Bindung zwischen den beiden Stichproben für die p-Wert-Berechnung die approximativ normalverteilte Teststatistik $Z_{n,m}^{mod}$ (`with continuity correction`) verwendet und eine entsprechende `Warning message` ausgegeben.
Die Alternative lässt sich auch hier mit `alternative` spezifizieren: `alternative = "greater"` bedeutet H_1 : `mu` $> -\theta_0$, d. h., X-Verteilung tendiert zu um *mehr* als θ_0 größeren Werten als Y-Verteilung. `alter = "less"` ist entsprechend H_1 : `mu` $< -\theta_0$.

```
> protarm[5] <- 107.5
> wilcox.test(protreich, protarm)

  Wilcoxon rank sum test

data:  protreich and protarm
W = 62, p-value = 0.1003
alternative hypothesis: true
location is not equal to 0
```

Nach dem künstlichen Aufbrechen der Bindung (zu Demonstrationszwecken) erhalten wir die Ausgabe des exakten Wilcoxon-Rangsummentests: Wert von $W_{n,m}$ (`W`) und exakter p-Wert (`p-value`). Alles Weitere ist wie oben.

Bemerkung: Für den Fall, dass es ausreicht, einen unspezifizierten Unterschied zwischen zwei stetigen Verteilungen nachzuweisen, steht als nicht-parametrisches Verfahren z. B. auch der Kolmogorov-Smirnov Test zur Verfügung, der in **R** in `ks.test` implementiert ist. Wir verweisen auf die diesbzgl. Hilfeseite einschließlich der dort ausgesprochenen Warnungen und angegebenen Referenzen oder auch auf [16, Büning & Trenkler (1994)].

Literatur

[1] Agresti, A.: *Categorical Data Analysis.* John Wiley, New York, 1990. (2nd ed., 2002)

[2] Agresti, A.: *An Introduction to Categorical Data Analysis.* John Wiley, New York, 1996. (2nd ed., 2007)

[3] Akaike, H.: *A new look at statistical model identification.* IEEE Transactions on Automatic Control, Vol. AC-19, No. 6, 1974, pp. 716 - 723.

[4] Bates, D. M. : *lme4: Mixed-effects modeling with R.* Unveröffentlicher und unfertiger Entwurf, 2010. URL http://lme4.r-forge.r-project.org/book/

[5] Belsley, D. A., Kuh, E., Welsch, R. E.: *Regression Diagnostics.* John Wiley, New York, 1980.

[6] Bock, J.: *Bestimmung des Stichprobenumfangs für biologische Experimente und kontrollierte klinische Studien.* R. Oldenbourg Verlag, München, 1998. (Evtl. vergriffen.)

[7] Bortz, J., Lienert, G. A., Boehnke, K.: *Verteilungsfreie Methoden in der Biostatistik.* 2., korrig. und aktualis. Auflage, Springer-Verlag, Berlin, 2000. (oder 3. Aufl. 2008).

[8] Box, G. E. P., Cox, D. R.: *An analysis of transformations.* Journal of the Royal Statistical Society, Series B, Vol. 26, 1964, pp. 211 - 252.

[9] Box, G. E. P., Hunter, W. G., Hunter, J. S.: *Statistics for Experimenters: An Introduction to Design, Data Analysis and Model Building.* John Wiley, New York, 1978. (2nd ed., 2005)

[10] Braun, W. J., Murdoch, D. J.: *A First Course in Statistical Programming with R.* 3rd ed., Cambridge University Press, 2021.

[11] Bretz, F., Hothorn, T., Westfall, P.: *Multiple Comparisons Using R.* Chapman & Hall/CRC Press, Boca Raton/Florida, 2010.

[12] Brown, L. D., Cai, T. T., DasGupta, A.: *Interval Estimation for a Binomial Proportion.* Statistical Sciences, 2001, Vol. 16, No. 2, pp. 101 - 133.

[13] Brunner, E., Domhof, S., Langer, F.: *Nonparametric Analysis of Longitudinal Data in Factorial Experiments.* John Wiley, 2002. (Vermutlich vergriffen.)

[14] Brunner, E., Langer, F.: *Nichtparametrische Analyse longitudinaler Daten.* Oldenbourg-Verlag, 1999.

[15] Brunner, E., Munzel, U.: *Nicht-parametrische Datenanalyse. Unverbundene Stichproben.* 2. Auflage, Springer-Verlag, Berlin, 2013.

[16] Büning, H., Trenkler, G.: *Nichtparametrische statistische Methoden.* 2., völlig neu überarb. Ausg., Walter-de-Gruyter, Berlin, 1994.

[17] Chambers, J.: *Software for Data Analysis: Programming with R.* Corr. 2nd printing, Springer-Verlag, Berlin, 2008. (Taschenbuchausgabe aus 2010)

[18] Chambers, J. M., Cleveland, W. S., Kleiner, B., Tukey, P. A.: *Graphical Methods for Data Analysis.* Wadsworth Pub. Co., Belmont/Californien, 1983.

[19] Chen, D.-G., Peace, K. E.: *Clinical Trial Data Analysis Using R.* Chapman & Hall/CRC, Taylor & Francis Group, Boca Raton/Florida, 2011.

[20] Cleveland, W. S.: *Robust Locally Weighted Regression and Smoothing Scatterplots.* In: Journal of the American Statistical Association. Vol. 74, Nr. 368, Dec. 1979, pp. 829 - 836.

[21] Cleveland, W. S.: *LOWESS: A program for smoothing scatterplots by robust locally weighted regression.* The American Statistician, 35, p. 54.

[22] Cleveland, W. S.: *The Elements of Graphing Data.* Wadsworth Advanced Books and Software, Monterey/Californien, 1985. (Hobart Pr., revised ed., 1994)

[23] Cleveland, W. S., Grosse, E., Shyu, W. M.: *Local regression models.* Chapter 8 of *Statistical Models in S*, eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole, 1992.

[24] Cohen, J.: *Statistical power analysis for the behavioral sciences.* 2nd ed., Hillsdale, NJ: Lawrence Erlbaum, 1988.

[25] Cook, R. D.: *Detection of influential observations in linear regression.* Technometrics, 19 (1), pp. 15-18, 1977.

[26] Cook, R. D., Weisberg, S.: *Residuals and Influence in Regression.* Chapman & Hall, New York, 1982.

[27] Cox, D. R., Hinkley, D. V.: *Theoretical Statistics.* Chapman & Hall, London, 1974.

[28] Dalgaard, P.: *Introductory Statistics with R.* 2nd ed., Springer-Verlag, New York, 2008.

[29] Dilba, G., Schaarschmidt, F., Hothorn,L. A.: *Inferences for Ratios of Normal Means.* In: R News Vol. 7(1), April 2007, pp. 20 - 23. URL `https://cran.r-project.org/doc/Rnews/Rnews_2007-1.pdf`

[30] Eddelbuettel, D.: *Seamless R and C++ integration with Rcpp.* Springer-Verlag, New York, 2013.

[31] Everitt, B. S., Hothorn, T.: *A Handbook of Statistical Analyses Using R.* 2nd ed., Chapman & Hall/CRC, Boca Raton, 2010.

[32] Faraway, J. J.: *Linear Models with R.* Chapman & Hall/CRC, Boca Raton, 2nd ed., 2015.

[33] Faraway, J. J.: *Extending the Linear Model with R.*, 2nd ed., Chapman & Hall/CRC, Boca Raton, 2016.

[34] Fahrmeir, L., Kneib, T., Lang, S.: *Regression. Modelle, Methoden und Anwendungen.* 2. Aufl., Springer-Verlag, Berlin, 2009.

[35] Fleiss, J. L., Levin, B., Paik, M. C.: *Statistical Methods for Rates and Proportions.* 3rd ed., John Wiley, New York, 2003.

[36] Fox, J.: *Applied Regression Analysis, Linear Models, and Related Methods.* Sage Publications, Thousand Oaks, 1997.

[37] Fox, J., Weisberg, H. S.: *An R Companion to Applied Regression.* 3rd ed., Sage Publications, Thousand Oaks, 2019.

[38] Friendly, M.: *Mosaic displays for multi-way contingency tables.* Journal of the American Statistical Association, 89, pp. 190 - 200.

[39] Friendly, M.: *Visualizing Categorical Data.* SAS Insitute, Carey, NC, 2000. URL `http://www.math.yorku.ca/SCS/vcd`

[40] Friendly, M., Meyer, D.: *Discrete Data Analysis with R. Visualization and Modeling Techniques for Categorical and Count Data* Chapman & Hall/CRC, Boca Raton/Florida, 2016.

[41] Gandrud, C.: *Reproducible Research with R and RStudio.* Chapman & Hall/CRC: The R Series, Boca Raton/Florida, 2014.

[42] Gardner, M. J., Altman, D. G.: *Confidence intervals rather than P values: estimation rather than hypothesis testing.* British Medical Journal 1986, Vol. 292, pp. 746 - 750.

[43] Ghosh, B. K.: *Some monotonicity theorems for χ^2, F and t distributions with appplications.* Journal of the Royal Statistical Society, Series B, 1973, Vol. 35, pp. 480 - 492.

[44] Goldberg, D.: *What Every Computer Scientist Should Know about Floating-Point Arithmetic.* ACM Computing Surveys, Vol. 23(1), 1991. Available at `http://www.validlab.com/goldberg/paper.ps`, extended version at `http://www.validlab.com/goldberg/paper.pdf`

[45] Graybill, F. A.: *Theory and Application of the Linear Model.* Duxbury Press, Wadworth Publishing Comp., Inc., North Scituate, 1976. (Taschenbuch aus 2000))

[46] Hagemann, S.: *Geschachtelte Hypothesensequenzen in linearen Modellen.* Bachelorthesis, Math. Inst., JLU Giessen, 2012.

[47] Harrell, Jr., F. E.: *Regression Modeling Strategies. With Applications to Linear Models, Logistic Regression, and Survival Analysis.* Corr. 2nd Printing, Springer-Verlag, New York, 2002.

[48] Hatzinger, R., Hornik, K., Nagel, H., Maier, M. J.: *R: Einführung durch angewandte Statistik.* 2., aktualis. Aufl., Pearson Studium, München, 2014.

[49] Hedderich, J., Sachs, L.: *Angewandte Statistik. Methodensammlung mit R.* 17., überarb. Aufl., Springer-Verlag, 2020.

[50] Heiberger, R. M., Holland, B.: *Statistical Analysis and Data Display: An Intermediate Course with Examples in R.* 2nd ed., Springer-Verlag, New York, 2015. URL `http://www.springer.com/us/book/9781493921218`

[51] Henze, N.: *Stochastik für Einsteiger. Eine Einführung in die faszinierende Welt des Zufalls.* 13., überarb. und aktualis. Aufl., Springer Spektrum, 2021, `https://doi.org/10.1007/978-3-662-63840-8`.

[52] Henze, N.: *Stochastik: Eine Einführung mit Grundzügen der Maßtheorie: Inkl. zahlreicher Erklärvideos.* 1. Aufl., Springer Spektrum, 2019, `https://doi.org/10.1007/978-3-662-59563-3`.

[53] Hettmansperger, T. P.: *Statistical inference based on ranks.* John Wiley, New York, 1984.

[54] Hochberg, Y., Tamhane, A. C.: *Multiple Comparison Procedures.* John Wiley, New York, 1987.

[55] Hocking, R. R.: *Methods and Applications of Linear Models: Regression and the Analysis of Variance.* John Wiley, New York, 1996.

[56] Hocking, R. R.: *Methods and Applications of Linear Models: Regression and the Analysis of Variance.* 2nd ed., John Wiley, New York, 2003.

[57] Hollander, M., Wolfe, D. G.: *Nonparametric statistical methods.* John Wiley, New York, 1974. (2nd ed., 1999)

[58] Hollander, M., Wolfe, D. A., Chicken, E.: *Nonparametric statistical methods.* 3rd ed., John Wiley, Hoboken/New Jersey, 2014.

[59] Horn, M., Vollandt, R.: *Multiple Tests und Auswahlverfahren.* Spektrum Akademischer Verlag, 1995. (Ehemals im Gustav Fischer Verlag, Stuttgart erschienen.)

[60] Horthorn, T., Everitt, B. S.: *A Handbook of Statistical Analyses Using R.* 3rd ed. Chapman & Hall/CRC Press, Taylor & Francis Group, Boca Raton/Florida, 2014.

[61] Hsu, J. C.: *Multiple Comparisons.* Chapman & Hall/CRC, London, 1996.

[62] ICH E9: *Statistical Principles for Clinical Trials.* London, UK: International Conference on Harmonisation, 1998. Adopted by CPMP March 1998 (CPMP/ICH/363/96). URL `http://www.ich.org`

[63] Jurečková, J., Picek, J.: *Robust Statistical Methods with R.* Chapman & Hall/CRC, Boca Raton, 2006.

[64] Kendall, M. G., Stuart, A.: *Advanced Theory of Statistics: 2. Inference and relationship*, 2nd ed., Griffin, London, 1961.

[65] Kendall, M. G., Stuart, A.: *Advanced Theory of Statistics: 1. Distribution theory*, 2nd ed., Griffin, London, 1963.

[66] Kloke, J., McKean, J. W.: *Nonparametric statistical methods using R.* Chapman & Hall/CRC, Boca Raton, 2015.

[67] Ligges, U.: *R Help Desk: Accessing the sources.* In: R News Vol. 6(4), October 2006, pp. 43 - 45. URL `https://cran.r-project.org/doc/Rnews/Rnews_2006-4.pdf`

[68] Ligges, U.: *Programmieren mit R.* 4., aktual. und überarb. Auflage, Springer-Spektrum, Berlin, 2015.

[69] Linhart, H., Zucchini, W.: *Model Selection.* John Wiley, New York, 1986.

[70] Ludbrook, J.: *Linear regression analysis for comparing two measurers or methods of measurement: But which regression?* Clinical & Experimental Pharmacology & Physiology, 2010; 37(7): 692 - 699.

[71] Maindonald, J., Braun, J.: *Data Analysis and Graphics Using R. An Example-based Approach.* 3rd ed., Cambridge University Press, 2010.

[72] Mardia, K. V., Kent, J. T., Bibby, J. M.: *Multivariate Analysis.* Academic Press, London, 1979. (Evtl. vergriffen)

[73] Mathai, A. M., Provost, S. B.: *Quadratic Forms in Random Variables: Theory and Applications.* Marcel Dekker, Inc., New York, 1992. (Scheint vergriffen.)

[74] Matsumoto, M., Nishimura, T.: *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator.* ACM Transactions on Modeling and Computer Simulation, Vol. 8, 1998, pp. 3 - 30.

[75] Matloff, N.: *The Art of R Programming. A Tour of Statistical Software Design.* No Starch Press, Inc., San Francisco/California, 2011.

[76] Meyer, D., Zeileis, A., Hornik, K.: *The Strucplot Framework: Visualizing Multi-way Contingency Tables with vcd.* Journal of Statistical Software, Vol. 17 (3), 2006, pp. 1 - 48. URL `https://www.jstatsoft.org/v17/i03`

[77] Monto, A. S.: *Francis Field Trial of Inactivated Poliomyelitis Vaccine: Background and Lessons for Today.* Epidemiological Reviews, Vol. 21 (1), 1999, pp. 7 - 23, via URL `http://epirev.oxfordjournals.org/content/21/1`

[78] Murrell, P.: *R Graphics.* Chapman & Hall/CRC Press, Boca Raton/Florida, 2005.

[79] Murrell, P.: *R Graphics.* 2nd ed. Chapman & Hall/CRC Press, Boca Raton/Florida, 2011.

[80] Neter, J., Wasserman, W., Kutner, M. H.: *Applied Linear Statistical Models: Regression, Analysis of Variance, and Experimental Designs.* 3rd ed., Richard D. Irwin, Inc., 1990. (Gebraucht ab 24 €) ODER: Kutner, M. H., Neter, J., Nachtsheim, C. J., Wasserman, W.: *Applied Linear Statistical Models.* 5th revised ed., McGraw Hill Higher Education, 2004.

[81] Passing H., Bablok, W.: *A new biometrical procedure for testing the equality of measurements from two different analytical methods. Application of linear regression procedures for method comparison studies in Clinical Chemistry, Part I.* Journal of Clinical Chemistry and Clinical Biochemistry, Vol. 21 (11), pp. 709 – 720. `https://doi:10.1515/cclm.1983.21.11.709`.

[82] Rodriguez, R. N.: *Correlation.* In: Encyclopedia of Statistical Sciences, Vol. 2, ed. Kotz & Johnson, John Wiley, New York, 1982.

[83] Rodriguez, R. N.: *Correlation.* In: Encyclopedia of Statistical Sciences, Vol. 2, ed. Kotz & Johnson, John Wiley, New York, 2004. URL `http://dx.doi.org/10.1002/0471667196.ess0423.pub2`

[84] Sachs, L., Hedderich, J.: *Angewandte Statistik. Methodensammlung mit R.* 12., vollst. neu bearb. Aufl., Springer-Verlag, 2006. (Für die 17., überarb. Aufl., 2020 siehe [49].)

[85] Sarkar, D.: *Lattice: Multivariate Data Visualization with R.* Springer-Verlag, New York, 2008.

[86] Schott, James R.: *Matrix analysis for statistics.* Wiley series in probability and statistics, New York, 1997.

[87] Searle, S. R.: *Linear Models.* John Wiley & Sons, New York, 1971.

[88] Seber, G. A. F.: *Linear Regression Analysis.* John Wiley & Sons, New York, 1977.

[89] Shumway, R. H., Stoffer, D. S.: *Time series analysis and its applications. With R examples.* 3. ed., Springer-Verlag, New York/NY, 2011.

[90] Snedecor, G. W., Cochran, W. G.: *Statistical Methods.* 8th ed., Iowa State University Press, Blackwell Publishers, Ames/Iowa, 1989.

[91] Timischl, W.: *Biostatistik. Eine Einführung für Biologen.* Springer-Verlag, Wien, New York, 1990. (Nachfolger: *Biostatistik. Eine Einführung für Biologen und Mediziner.* 2., neu bearb. Aufl., 2000.)

[92] Tong, Y. L.: *The multivariate normal distribution.* Springer-Verlag, New York, 1990. (Scheint vergriffen.)

[93] Tukey, J. W.: *Exploratory Data Analysis* Addison-Wesley, Reading/Massachusetts, 1977.

[94] Venables, W. N.: *Exegeses on linear models.* A paper presented to the S-PLUS User's Conference in Washington, DC/USA, 8 - 9th Oct. 1998. Version of May 13, 2000. URL `http://www.stats.ox.ac.uk/pub/MASS3/Exegeses.pdf`

[95] Venables, W. N., Ripley, B. D.: *Modern Applied Statistics with S.* 4th ed., Corr. 2nd printing, Springer-Verlag, New York, 2003.

[96] Venables, W. N., Ripley, B. D.: *S Programming.* Corr. 3rd printing, Springer-Verlag, New York, 2004. (∼ 97 €) ODER: Corr. 2nd printing, 2001.

[97] Verzani, J.: *Using R for Introductory Statistics.* 2nd revised ed. Chapman & Hall/CRC Press, Boca Raton/Florida, 2014.

[98] Wand, M. P., Jones, M. C.: *Kernel Smoothing.* Chapman & Hall/CRC Press, Boca Raton/Florida, 1995.

[99] Weisberg, S.: *Applied Linear Regression.* 3rd ed., John Wiley, New York, 2005.

[100] Wichura, M. J.: *The Coordinate-Free Approach to Linear Models.* Cambridge Series in Statistical and Probabilistic Mathematics, Band 19, Cambridge University-Press, Cambridge, 2006.

[101] Wickham, H.: *Reshaping data with the reshape Package.* Journal of Statistical Software, November 2007, Vol. 21, No. 12, pp. 1 - 20. URL `https://www.jstatsoft.org/v21/i12`

[102] Wickham, H.: *ggplot2. Elegant Graphics for Data Analysis.* Springer-Verlag, New York, 2009.

[103] Wickham, H.: *The Split-Apply-Combine Strategy for Data Analysis.* Journal of Statistical Software, April 2011, Vol. 40, No. 1, pp. 1 - 29. URL `https://www.jstatsoft.org/v40/i01`

[104] Wickham, H.: *Advanced R.* 2nd ed., Chapman & Hall/CRC, Boca Raton, 2019.

[105] Wickham, H., Francois, R.: *dplyr: A Grammar of Data Manipulation.* R package version 0.5.0, 2016. `https://CRAN.R-project.org/package=dplyr`

[106] Witting, H., Müller-Funk, U.: *Mathematische Statistik II. Asymptotische Statistik: Parametrische Modelle und nichtparametrische Funktionale.* Teubner, Stuttgart, 1995. (Scheint vergriffen.)

[107] Xie, Y.: *Dynamic Documents with R and knitr.* 2nd rev. ed. Chapman & Hall/CRC: The R Series, Boca Raton/Florida, 2015. Online: `https://yihui.org/knitr`

[108] Xie, Y., Allaire, J.J., Grolemund, G.: *R Markdown: The Definitive Guide.* Chapman & Hall/CRC: The R Series, Boca Raton/Florida, 2019. Online: `https://bookdown.org/yihui/rmarkdown/`

[109] Zeileis, A., Leisch, F., Hornik, K., Kleiber, C.: *strucchange: An R package for testing for structural change in linear regression models.* Journal of Statistical Software, 7(2):1 - 38, 2002. URL `https://www.jstatsoft.org/v07/i02/`.

[110] Zhao, Y. D., Rahardja, D., Qu, Y.: *Sample size calculation for the Wilcoxon-Mann-Whitney test adjusting for ties.* Statistics in Medicine 2008, Vol. 27, pp. 462 - 468.

[111] Zuur, A. F., Ieno, E. N., Elphick, C. S.: *A protocol for data exploration to avoid common statistical problems.* Methods in Ecology & Evolution 2009, pp. 1 - 12.