Einführung zu R*

8. Mai 2008

*Ohne Anspruch auf Vollständigkeit und Fehlerfreiheit. Das Skript befindet sich noch im Aufbau und wird laufend verändert. Hinweise an malte.spiess@uni-ulm.de

Inhaltsverzeichnis

1 Einleitung

R ist, ebenso wie S-Plus (siehe A), ein sehr mächtiges Statistikprogramm. Im Grunde ist es eine Programmierumgebung, in der sich sehr viel realisieren lässt. Die UNIX- und WINDOWS-Versionen unterscheiden sich nur unwesentlich. **R** ist auf der Internetseite

www.r-project.org

kostenlos erhältlich. Hier befindet sich ebenfalls eine Dokumentation. Bisher läuft **R** nicht auf allen Rechnern im Mathematik-Netzwerk, aber auf den neueren Rechnern ist es bereits installiert. Allerdings ist es hier leider nicht möglich, Zusatzpakete zu installieren, deshalb muss man hierfür auf die Rechner des Rechenzentrums ausweichen.

Im Folgenden eine kleine Einführung in die Prinzipien von **R**(wobei ↩ für das Betätigen der Return-Taste steht):

Start von **R**: z.B. `thales$ R` ↩.

Nach dem Start erscheint der sogenannte Prompt: `>`. Hier können Befehle eingegeben werden, und mit ↩ bestätigt.

Erscheint statt dem „`>`“ ein “`+`“ als Prompt bedeutet das: Eingabe kann/muss fortgesetzt werden. Beispiel:

```
> 1 +
+ 2
[1] 3
```

Hier erscheint vor der 2 der +-Prompt.

Man kann frühere Kommandos mit den Pfeiltasten (Pfeil nach oben) wieder anzeigen lassen und ausführen.

Das Beenden von **R** ist mit `q()` ↩ möglich
Auf Wunsch speichert **R** die Daten und Kommandozeilenhistorie in „.RData” bzw. „.Rhistory”.

Mehrere Befehle in einer Eingabeziele kann man durch “ ; “ trennen.

Die Hilfefunktion unter **R** kann man mit `> help(`*name*`)` oder `> ?`*name* erhalten. Es gibt außerdem noch ein Browser-basiertes Hilfesystem, das man mit `help.start()` starten kann.

Kommentare werden mit `#` eingeleitet und gehen bis zum Zeilenende.

Variablen wird ihr Wert mit ’`<-`’ zugewiesen.

Wichtig bei Bezeichnungen ist die Groß-/Kleinschreibung. Hier ein kleines Beispiel:

```
> objekt1 <- 1.043   # d.h.: GleitPUNKTzahlen, Zuweisung mit <-
> objekt1 <- 3       # überschreibt die erste Definition
> Objekt1 <- 2       # neues Objekt
```

Bereits belegte Namen sind z.B.: pi, t, f, T, F, mean, var,... (können aber überschrieben werden)

2 Datentypen

In diesem Kapitel wollen wir die wichtigsten Datentypen in **R** vorstellen und erklären, wie man mit diesen Datentypen arbeiten kann. Außerdem sollen die wichtigsten vorgefertigten Funktionen in **R** erwähnt werden, die zu den entsprechenden Datentypen gehören.

2.1 Vektoren

Ein Vektor ist eine geordnete Sammlung mehrerer Objekte **gleicher Art**. Die Objekte werden mit fortlaufender Nummer hintereinander geschrieben und können so auch angesprochen werden. Beispiel:

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

In **R** sind auch einfache Zahlen (z. B. 4) Vektoren und werden entsprechend so behandelt:

```
> x <- 4
> x[1]
[1] 4
```

2.1.1 Erzeugung von Vektoren

Systematisch werden Vektoren durch den Befehl `c(element1,element2,...)` zu einem Vektor verbunden - das c steht für „concatenate“(zusammenfügen).

Weitere Möglichkeiten zur Generierung von Vektoren sind:

- `rep()`: Syntax: `rep(x, times=n)` wiederholt Objekt x n mal
  Beispiel: rep(1,4) ist identisch mit c(1,1,1,1)
- `seq()`: Syntax: `seq(from=, to=, by=, length=)`
  Beispiel: > seq(1,3, by=0.1) ergibt 1.0, 1.1, 1.2,..., 2.9, 3.0
- `from:to` # entspricht `seq(from,to,by=1.0)`,
  Beispiel: 1:3 ergibt 1,2,3
- `scan()` # zeilenweise Einlesen von Std-Eingabe oder aus einer Datei
  Beispiele:

```
> daten <- scan() # Liest folgende Zeilen ein
1: 3              # Wir tragen eine 3 als erstes Element ein
2: 5              # Wir tragen eine 5 als zweites Element ein
3:                # keine Eingabe führt zum Beenden des Einlesens
Read 2 items      # Zeigt Leseende an
> print(daten)
[1] 3 5
> daten <- scan("input.data")  #liest alles aus der Datei
                               # (es sollten nur Zahlen in der Datei stehen)
```

2.1.2 Rechnen mit Vektoren

Rechnungen mit Vektoren werden von **R** immer komponentenweise durchgeführt. Bei der Addition ergibt sich so die normale Vektoraddition, aber auch bei der Multiplikation wird komponentenweise vorgegangen. Auch einfache Funkionen (z. B. `sqrt()` - die Wurzel) werden auf jeden Eintrag einzeln ausgeführt. Beispiel:

```
> x <- 1:3
> y <- rep(2, 3)
> x + y
[1] 3 4 5
> x * y
[1] 2 4 6
> sqrt(x)
[1] 1.000000 1.414214 1.732051
```

Ist bei einer solchen Operation ein Vektor kleiner als der andere, wird der kleinere so oft wiederholt, dass er so lang ist wie der größere. Beispiel:

```
> x <- 1:3
> x + 5
[1] 6 7 8
> y <- c(2,4)
> x + y
[1] 3 6 5
Warning message:
In x + y : Länge der längeren Objekts
   ist kein Vielfaches der Länge der kürzeren Objektes
```

Ist die Länge des kleineren Vektors (hier `y` - Länge 2) kein Teiler der Länge des größeren Vektors (hier `x` - Länge 3), kommt zwar eine Warnung, aber die Operation wird dennoch durchgeführt.

2.1.3 Funktionen auf Vektoren

- Bestimmung der Länge eines Vektors, d.h. die Anzahl der Einträge: `length()`.
- Ein Vektor wird mit dem Befehl `t()` transponiert.

Beispiel:

```
> x<-c(1.4,3.7,2.0,4.6,5.1)
> length(x)  # Die Länge von x
[1] 5
> t(x)       # Transponieren von x
     [,1] [,2] [,3] [,4] [,5]
[1,]  1.4  3.7    2  4.6  5.1
```

Beachte

- Der erste Eintrag in einem Vektor ist an Position 1.
- Das Ergebnis des Transponierens ist eine Matrix, siehe Abschnitt 2.3.

2.1.4 Auswählen von Teilvektoren

In **R** gibt es viele Möglichkeiten, um Teilvektoren von gegebenen Vektoren auszuwählen. Diese Methoden lassen sich in der Regel auch auf andere Objekttypen übertragen, wobei die Vorgehensweise mit `subset()` am leichtesten auf alle Objekte anwendbar ist.

Am leichtesten lassen sich die Methoden an einem Beispiel demonstrieren. Im folgenden sei deshalb immer `x` folgendermaßen definiert:

```
x <- c(1,3,2,1,5,-1)
```

Als erstes wollen wir die ersten 3 Elemente auswählen:

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

Im allgemeinen lässt sich durch `x[v]`, wobei `v` ein Vektor ist, ein beliebiger Teilvektor von `x` auswählen:

```
> x[c(3,2,5)]
[1] 2 3 5
```

Es gibt aber noch weitere geschickte Methoden, z. B. wenn man alle Elemente haben will, die größer als 2 sind:

```
> x[x>2]
[1] 3 5
```

Hier lässt sich in die eckigen Klammern ein beliebiger boolescher Ausdruck schreiben, mehrere lassen sich mit einem einfachen `&` kombinieren:

```
> x[x>2 & x^2 < 16]
[1] 3
```

Das gleiche lässt sich auch mit `subset()` erreichen, wobei der Vorteil hierbei ist, dass die Auswahl sich leicht auf andere Objekttypen übertragen lässt:

```
> subset(x, x>2 & x^2 < 16)
[1] 3
```

2.1.5 Nicht-numerische Vektoren

Es gibt auch Vektoren, die keine Zahlen enthalten, z. B. character-Vektoren (Strings) und logical-Vektoren (Boolesche Ausdrücke).

Diese Vektoren lassen sich genau so wie die numerischen anlegen und ansprechen, z. B.:

```
> name <- "Name"
> name
[1] "Name"
> name[1]
[1] "Name"
> namen <- c("Name1", "Name2")
> namen[2]
[1] "Name2"
> bools <- c(TRUE, FALSE)
> bools
[1]  TRUE FALSE
> gemischt <- c("a", 2)
> gemischt <- c("a", 2, TRUE)
> gemischt
[1] "a"    "2"    "TRUE"
> gemischt <- c(2, TRUE)
> gemischt
[1] 2 1
```

Hier sieht man auch, dass sich die Typen in Vektoren nicht mischen lassen. Jeder „gemischte“ Vektor erhält den „Obertypen“ seiner Einträge, also bei characters und anderen den Typ character, bei numerisch und logical den numerischen Typ.

Näheres zu character-Objekten ist in Abschnitt 3.3 und zu logical-Objekten in Abschnitt 4.1.

2.2 Arrays

Ein-dimensionale Arrays unterscheiden sich nur geringfügig von Vektoren. Bei ihnen ist die Größe festgelegt; wenn man sie ändert, wird der Array zum Vektor. Interessant sind Arrays hauptsächlich wegen der Möglichkeit, höherdimensionale Objekte zu erzeugen.

Das Anlegen eines Arrays geht mit `array(data= , dim= )`

Beispiel:

```
> a<-array(0, 3) # Hier wird der Vektor 0 dreimal verwendet
> a
[1] 0 0 0
> attributes(a)
$dim
[1] 3

> a[4]<-5
> a
[1] 0 0 0 5
> attributes(a)
NULL
> attributes(1) # (1 ist ein Vektor)
NULL
```

Hier sieht man anhand der `attributes()`-Funktion, die die Eigenschaften eines Objektes auflistet, den Unterschied zwischen dem Array und dem Vektor.

Noch ein Beispiel zu den Dimensionen:

```
> a<-array(c(0,1),2)
> b<-array(c(1,2,3,4),4)
> a
[1] 0 1
> b
[1] 1 2 3 4
> a*b
Fehler in a * b : nicht passende Arrays
> a<-c(0,1)
> b<-c(1,2,3,4)
> a*b
[1] 0 2 0 4
```

Hier sieht man: Im Gegensatz zu Vektoren ist nicht erlaubt, zwei Arrays mit unterschiedlicher Dimension miteinander zu multiplizieren. Ansonsten erfolgen die Operationen so wie bei den Vektoren.

In der Anwendung werden Arrays hauptsächlich für Matrizen verwendet, siehe nächstes Kapitel.

2.3 Matrizen und höherdimensionale Arrays

Matrizen sind 2-dimensionale Arrays. Ansonsten verhalten sie sich wie alle Arrays bzw. wie Vektoren. Sie können also nur Variablen **eines** Typs enthalten (Zahlen, Zeichenketten,...).

2.3.1 Erzeugen von Matrizen, Zugriff auf Matrizen

Syntax: `matrix(data, nrow=, ncol=, byrow=FALSE)` [Dimension von Matrizen: `dim()`]

Beispiele:

```
> m <- matrix(0, 4, 5)  # eine 4 x 5 mit lauter Nullen
> m <- array(0, c(4, 5)) # das gleiche Objekt nur mit Array-Schreibweise
> m <- matrix(1:10, 5)  # eine 5 x 2 welche die Zahlen 1 bis 10 enthält
> m
     [,1] [,2]
```

```
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
> m[1,]
[1] 1 6
> m[,1]
[1] 1 2 3 4 5
> mm <- matrix( scan("mfile"), ncol=5, byrow=TRUE)  # zeilenweise Einlesen aus Datei
```

Hier kann mit `m[1,]` auf eine Zeile und mit `m[,1]` auf eine Spalte der Matrix zugegriffen werden. Das Ergebnis ist ein Vektor.

Höherdimensionale Matrizen können mit ‚`x<-array(0 , c(3,4,2))`' o.ä. erzeugt werden. In diesem Fall ist ‚`x[1,,]`' eine 4x2-Matrix.

2.3.2 Rechnen mit Matrizen

Zugriff auf Einträge von Matrizen:

```
> x <- c(1.4,3.7,2.0,4.6); m <-matrix(x,2,2,T)
> m[1,2]
[1] 3.7
```

Zugriff auf i-te Zeile: `m[i,]`
Zugriff auf j-te Spalte: `m[,j]`

Multiplizieren von Matrizen:

```
> m
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> n
     [,1] [,2]
[1,]    1    0
[2,]    0    1
> m*n
     [,1] [,2]
[1,]    1    0
[2,]    0    4
> m%*%n
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Wie man sieht, führt ein normales `*` eine komponentenweise Multiplizikation der Matrizen durch, während `%*%` die übliche Matrizenmultiplikation verursacht. Auch alle anderen Funktion wie `+` oder `sqrt()` werden bei Matrizen wie bei Vektoren komponentenweise angewendet.

2.3.3 Funktionen auf Matrizen

Hier ein paar der wichtigsten Funktionen:

- **Zeilen und Spalten hinzufügen**:
  Zu einer Matrix können mit den Befehlen `rbind(vektor1, vektor2,...)` bzw. `cbind(vektor1, vektor2,...)` Zeilen bzw. Spalten hinzugefügt werden.

- **Transponieren**:
  Eine Matrix wird mit dem Befehl `t()` transponiert.

- **Inversion**:
  Die Inverse lässt sich durch `solve()` bestimmen:

```
> m<-matrix(1:10,5)
> dim(m)
[1] 5 2
> mm<-t(m)%*%m
> dim(mm)
[1] 2 2
> mm
     [,1] [,2]
[1,]   55  130
[2,]  130  330
> solve(mm)%*%mm
              [,1]          [,2]
[1,] 1.000000e+00 7.105427e-15
[2,] 8.881784e-16 1.000000e+00
```

  Man sieht, dass die Inverse der Matrix numerisch bestimmt wird und hier Rundungsfehler auftreten.

  Deutlich stabiler ist der Aufruf `solve(mm, b)`, wenn `b` ein Vektor ist. Ergebnis ist hier $mm^{-1} * b$. Hierfür sind in **R** numerisch stabile Methoden implementiert, deshalb ist die letzte Vorgehensweise stets zu bevorzugen.

2.4 Listen

Listen sind Sammlungen von beliebigen Objekten. Sie sind ähnlich wie Vektoren, nur können in einer Liste **verschiedene** Objekttypen zusammengefasst werden. Außerdem können die Einträge benannt werden, was leserlicheren Code ermöglicht.

2.4.1 Erzeugen von Listen, Zugriff auf Listen

Eine Liste wird durch den Befehl `list()` erzeugt. Hierbei kann man durch `list(Name1 = ..., Name2 = ...)` den einzelnen Einträgen Namen geben. Beispiel:

```
> x <- list(Zahlen = c(1,2,3), Buchstaben = c("a","b"), c("a",1,"b",2))
  # Hier ist das erste Element ein Integer-Vektor und das zweite
  # (und dritte) ein Charakter-Vektor.
  # Beachte: nur die ersten zwei Einträge haben Namen.
> x[[1]]      # Liefert den Vektor "Zahlen"
[1] 1 2 3
> x[1]        # Gibt eine Liste zurück, die nur aus dem 1. Eintrag von x besteht
              # Das wird (fast) nie gebraucht, hier nur erwähnt, falls man mal
              # die 2. Klammer vergisst.
```

```
$ Zahlen
[1] 1 2 3
> x$Zahlen    # Entspricht "x[[1]]"
[1] 1 2 3
> x[[1]][1]   # Erstes Element des Vektors "Zahlen"
[1] 1
> x$Zahlen[1] # Entspricht "x[[1]][1]"
[1] 1
```

Beachte
Listen eignen sich vor allem sehr gut als Rückgabewert von Funktionen. Deshalb haben auch viele in **R** implementierte Funktionen diesen Typ als Rückgabewert, siehe auch Abschnitt 3.

2.5 Data Frames

Ein `data.frame` ist ist eine Kombination aus Liste und Vektor. Es ist eine Liste, die Vektoren der gleichen Länge aber mit unterschiedlichen Objekten als Elementen enthält. Man kann es sich auch als Verallgemeinerung des Typs Matrix vorstellen. Es ist vielleicht der wichtigste Datentyp in **R**, da bei der Erhebung von Messdaten oft solche Datenstrukturen vorliegen.

2.5.1 Erzeugen von Data Frames, Zugriff auf Data Frames

Es kann mit folgenden Befehlen erzeugt werden:

- `data.frame(Objekt1, Objekt2,...)`
  Zum Erzeugen bei vorhandenen Objekten
- `read.table("`*`Dateiname`*`")`
  Zum Lesen aus einer Datei, siehe Abschnitt 7.3.

Beispiel

```
> x <- data.frame("Gewicht"=c(65,75),"Groesse"=c(168,175),"Geschlecht"=c("m","w"))
> print(x)
  Gewicht Groesse Geschlecht
1      65     168          m
2      75     175          w
> Alter <- c(22,45)
> y <- cbind(x,Alter) # cbind hängt eine Spalte an, nicht c() verwenden!!
> y
  Gewicht Groesse Geschlecht Alter
1      65     168          m    22
2      75     175          w    45
```

Auf Spalten kann über `x$`*`Spaltenüberschrift`* zugegriffen werden
Beispiel
Wir haben den gleichen data.frame wie im obigen Beispiel:

```
> x$Gewicht # Liefert den Vektor "Gewicht"
[1] 65 75
> x[x$Gewicht<70, ] # Liefert alle Zeilen (als data.frame), in denen
                    # der Wert der Spalte "Gewicht" kleiner als 70 ist
  Gewicht Groesse Geschlecht
1      65     168          m
```

Wie weiter oben bereits erwähnt können Data Frames auch aus einer Datei eingelesen werden. Näheres dazu steht in Abschnitt 7.

3 Funktionen und Operatoren

In **R** gibt es viele nützliche Funktionen, die schon implementiert sind. Im folgenden Kapitel wollen wir ein paar der wichtigsten vorstellen.

Grundsätzlich gilt es in **R** zu beachten, dass es nur „call by value“ gibt, das heißt, dass Funktionen nie ihre Eingabewerte verändern können.

Wenn man also aufruft `f(x)`, wobei `f()` eine beliebige Funktion ist, ist in `x` nachher auf jeden Fall der gleiche Wert gespeichert wie vor dem Aufruf. Will man den Inhalt von `x` verändern, so muss man das mit dem Rückgabewert machen.

Als Beispiel hier das Sortieren eines Vektors:

```
> x <- 4:1
> x
[1] 4 3 2 1
> sort(x) # hier wird der Inhalt von x sortiert, aber der Wert wird nicht in x
          # gespeichert!
[1] 1 2 3 4
> x
[1] 4 3 2 1
> x <- sort(x) # (nur) so kann man den Wert von x verändern
> x
[1] 1 2 3 4
```

3.1 Mathematische Funktionen

R besitzt eine sehr große Auswahl an vordefinierten Funktionen. Es ist zu beachten, dass diese sowohl auf Zahlen als auch auf Vektoren/Matrizen (dann jeweils komponentenweise) anwendbar sind. Hier eine Liste wichtiger Funktionen:

Exponentialfkt:	`exp()`
Logarithmus(Basis: e):	`log()`
Logarithmus(Basis: base):	`logb(x,base=2)`
Wurzel:	`sqrt()`
Sinus:	`sin()`
Cosinus:	`cos()`
Tangens:	`tan()`

Mathematische Funktionen auf Vektoren:

Minimum, Maximum:	`min()`, `max()`
Sortieren:	`sort()`
Summe der Vektoreinträge:	`sum()`
Produkt aller Vektoreinträge:	`prod()`
Vektor mit Differenzen der Vektoreinträge (d. h. `x[i]-x[i-1]`):	`diff()`
kleinster und größter Vektoreintrag:	`range()`

Weitere nützliche Funktionen:

Integration:	`integrate()`

Beispiel 1
Anwendung von Funktionen:

```
> x <- c(1,2.3,4.2,3.2)
> sum(x)
[1] 10.7
> diff(x)
[1]  1.3  1.9 -1.0
> range(x)
[1] 1.0 4.2
> sqrt(x)
[1] 1.000000 1.516575 2.049390 1.788854
```

Beispiel 2
Anwendung von Funktionen:

```
> integrate(sin , -pi, pi)  # Berechnet das Integral von -pi bis pi
                            # der Sinusfunktion
0 with absolute error < 4.4e-14
> x <- integrate(sin , -pi, pi)
> print(x)
0 with absolute error < 4.4e-14
> x <- integrate(sin , -pi, pi)$value  # Zugriff auf das numerische Ergebnis
> print(x)
[1] 0
```

Eine wichtige Funktion ist die Indikatorfunktion $\mathbb{1}$. Diese kann in **R** durch eine Bedingung in runden Klammern "()" dargestellt werden.
Beispiel
$\mathbb{1}_{\{y:y<1\}}(x)$ soll implementiert werden:

```
> x <- 2
> x<1  # Das ist ein boolscher Ausdruck
[1] FALSE
> (x<1)*1  # Bei falscher Bedingung wird 0 zurückgeliefert
[1] 0
> x <- -1
> x<1
[1] TRUE
> (x<1)*1  # Bei wahrer Bedingung wird 1 zurückgeliefert
[1] 1
```

Beachte: Falls x ein Vektor ist, wird das Ergebnis komponentenweise bestimmt.

Mit dem Befehl `%*%` können Vektoren und Matrizen miteinander multipliziert werden. Das Ergebnis ist eine Matrix.
Beispiel

```
> x <- c(2,3,4)
> y <- c(1,2,3)
> s <- x %*% y  # Eine Möglichkeit zur Berechnung des Skalarprodukts.
> s             # Beachte, das Ergebnis ist eine Matrix
```

```
     [,1]
[1,]   20
> t(x) %*% y    # das gleiche Ergebnis wie ohne transponieren
     [,1]
[1,]   20
> x %*% t(y)    # Liefert eine 3x3 Matrix
     [,1] [,2] [,3]
[1,]    2    4    6
[2,]    3    6    9
[3,]    4    8   12
```

Bei Matrizen sind die Funktionen (wie bei Vektoren) komponentenweise wirksam, allerdings kann man auch festlegen, dass sie nur auf bestimmte Zeilen bzw. Spalten angewendet werden.

Beispiel

Eine gegebene Matrix m könnte wie folgt aussehen:

```
> m
     [,1] [,2]
[1,]    1    2
[2,]    3    4
> sqrt(m)  # Auf alle Elemente angewendet
         [,1]     [,2]
[1,] 1.000000 1.414214
[2,] 1.732051 2.000000
> sum(m)  # Summe aller Elemente
[1] 10
> apply(m,1,sum)  # Zeilensummen
[1] 3 7
```

allgemein kann man mit `apply(Matrix, Margin=1/2, FUNKTION)` Funktionen auf die Zeilen (`Margin = 1`) bzw. Spalten (`Margin = 2`) einer Matrix anwenden.

In **R** ist es möglich, sich eigene Funktionen zu definieren. Dies ist vor allem für größere Projekte sinnvoll. Da **R** ein open–source Programm ist, ist auch der Quellcode der Funktionen aus den verfügbaren Paketen frei zugänglich. Diese Funktionen sind nach dem selben Schema aufgebaut und können den erforderlichen Bedingungen selbst angepasst werden.

3.2 Definition eigener Funktionen

Beim Definieren von eigenen Funktionen muss man insbesondere auf dass call-by-value-Prinzip von **R** achten (siehe 3). Es besagt, dass die einer Funktion übergebenen Parameter nicht innerhalb einer Funktion verändert werden können.

Beispiel zum call-by-value-Prinzip:

```
> f <- function(x) {
+ x <- x+1
+ return(x^2)
+ }
> x <- 3
> f(x)
[1] 16
```

```
> x
[1] 3
```

Wie man sieht hat `x` trotz Veränderung in `f()` immer noch den gleichen Wert.
Beispiel einer eigenen Funktion:

```
> wurzel <- function(x,n=2){ # Name der Funktion ist 'wurzel'
          # es können zwei Parameter übergeben werden: x und n
          # für x muss immer ein Wert angegeben werde, jedoch nicht für n,
          # für das hier ein Default-Wert von 2 definiert wurde
          # (da n=2 im Funktionenkopf)
+ out <- x^(1/n)  # Der Variablen 'out' wird der Wert zugewiesen
+ return(out)          # 'out' wird als Rückgabewert deklariert
+ }
```

Aufruf: `> wurzel(x)     # hier ist n = 2 (Default-Wert)`
Aufruf: `> wurzel(x, 4)     # hier ist n = 4`

Beachte

- Der Wert x, also das Argument einer Funktion, muss immer als Vektor gesehen werden.
- Falls „`return()`“ fehlt, wird die zuletzt benutzte Variable zurückgeliefert.

Beispiel
Funktion mit Listen:

```
> meineFunktion <- function(x,n){
+ d <- x-n
+ m <- x*n
+ result <- list(differenz=d, multipl=m, x=x, n=n)
+ return(result)
+ }
> meineFunktion(2,3)
$differenz:
[1] -1

$multipl:
[1] 6

$x:
[1] 2

$n:
[1] 3
```

3.3 Manipulation und Ausgabe von Text

- `sink()` Umlenken der Ausgabe

```
> sink("RAufgabe1.text")
> 1 + 2
> sink()                 # Zurück auf die Standardausgabe
```

- `source()`
  Sind syntaktisch richtige **R**-Anweisungen in einer Datei abgelegt worden, so lassen sich diese durch die Anweisung `source(`*`Dateiname`*`)` zur Ausführung bringen. Beispiel bei: `source("myfunctions.R")`

- `cat('Die Wurzel von',x,'ist gleich',sqrt(x),'\n')`
  schreibt den Text zwischen den Anführungszeichen, druckt das Ergebnis der Funktion und macht ganz am Ende einen Zeilenvorschub (`\n`).

- `print(paste('blabla',sqrt(x),'blabla'))` macht im Prinzip das gleiche.
  Die Funktion `paste()` klebt Textstrings zusammen; nützlich z.B. für die Funktion `title()`. Das Trennsymbol wird durch den Parameter `sep` festgelegt, der standardmäßig auf " " gesetzt ist.

```
> print(paste('blabla', x<-1:3 ,'blabla'))
[1] "blabla 1 blabla" "blabla 2 blabla" "blabla 3 blabla"
> cat('blabla',x<-1:3 ,'blabla')
blabla 1 2 3 blabla>
```

3.4 Kontrolle der im aktuellen Workspace vorhandenen Objekte

- `ls()`: listet alle gespeicherten Objekte auf, d.h. beim Arbeiten auf der Shell liefert es alle bisher genannten Objekte, w elche nicht gelöscht worden sind

- `rm(ob1)` oder `remove(ob1)`: entfernt Objekt ob1

- `save(ob1,file="`*`Dateiname`*`")`: speichert das Objekt ob1 in der Datei *Dateiname*

- `save.image(file="`*`Dateiname`*`")`: speichert den aktuellen Workspace in der Datei *Dateiname*

- `load("`*`Dateiname`*`")`: lädt ein mit save in der Datei *Dateiname* gespeichertes Objekt in den Workspace

Unter Windows lassen sich alle diese Funktionen auch durch Mauslicks auf die entsprechenden Befehle in der Menuleiste ausführen.

3.5 Sonstige interessante/wichtige Funktionen

- `summary(objekt)`
  gibt eine Zusammenfassung von objekt. Der Befehl ist generisch und reagiert je nach Beschaffenheit von objekt anders. Ist z.B. x ein numerischer Vektor, werden das Minimum, das Maximum, der Mittelwert sowie die 3 Quartile (in Vektorform) ausgegeben.

```
> x <- 1:4
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    1.75    2.50    2.50    3.25    4.00
```

- `names(objekt)` Namen von Teilobjekten von x ausgeben interessant z.B. bei `read.table()`

```
> x <- data.frame("Gewicht"=c(65,75),"Groesse"=c(168,175))
> names(x)
[1] "Gewicht" "Groesse"
```

oder bei **R** eigenen Funktionen:

```
> names(t.test(x))
[1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
[6] "null.value"  "alternative" "method"      "data.name"
```

- `numeric()` Mit dem Aufruf `numeric(0)` lässt sich ein leerer numerischer Vektor erzeugen. Das kann praktisch sein für for-Schleifen, bei denen man immer etwas mit `c()` an den Vektor anhängt (ggf. auch mit einer if-Bedingung).

```
> v2 <- numeric(0)
> v2
numeric(0)
> v <- 3:7
> for(i in 1:10) {
+ v2 <- c(v2, length(v[v<i^2]))
+ }
> v2
 [1] 0 1 5 5 5 5 5 5 5 5
```

- `getwd()`: liefert das aktuelle Arbeitsverzeichnis
- `setwd("`*`Pfad`*`")`: ändert das aktuelle Arbeitsverzeichnis in das in *Pfad* spezifizierte Verzeichnis

4 Bedingte Ausführung von Programmcode und Schleifen

In diesem Abschnitt werden die Möglichkeiten vorgestellt, in **R** den so genannten Programmfluss zu beeinflussen, also welche Teile wann ausgeführt werden.

4.1 Bedingte Ausführung mit `if`

Oft benötigt man innerhalb einer Funktion die Möglichkeit, sich je nach Lage der Situation zu entscheiden und fortzufahren. In **R** kann dies u.a. folgendermaßen realisiert werden:

```
if(test) {
  # Anweisungen für test==TRUE
} else {
  # Anweisungen für test==FALSE
}
```

Hinter `test` können sich zum Beispiel einfache Abfragen wie "`n > 10`" oder "`n == 3`" verbergen. Die Auswahl kann auch aufgrund mehrer Testabfragen entschieden werden:

```
if(test1 && test2){
  # Anweisungen für test1==TRUE und test2==TRUE
}
if(test1 || test2){
  # Anweisungen für test1==TRUE oder test2==TRUE
}
```

Es gibt noch andere Möglichkeiten, Bedingungen abzufragen.

```
> x <- 1:10  # erzeugt einen Vektor mit den Einträgen 1 2 3 4 5 6 7 8 9 10
> y <- c(1,2,1,2,1,1,1,2,2,2)
> x > 5  # Liefert für alle Werte von x, die größer als 5 sind, TRUE
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
> x[x>5]  # Liefert alle Werte von x, die größer als 5 sind
[1]  6  7  8  9 10
> x[y==2]  # Liefert die Werte der Positionen, an denen
           # der Wert des Vektors y gleich 2 ist
[1]  2  4  8  9 10
```

4.2 Schleifen

- `for (Variable in Vektor) {`**R**`-Ausdrücke}`
  Hierbei wird die Anzahl der Iterationen vor Beginn der Schleife genau festgelegt. Von der Laufvariable, welche die Iterationen zählt, kann immer der aktuelle Wert abgegriffen werden. Die Summe der ersten 100 natürlichen Zahlen kann man also folgendermaßen als Schleife darstellen:

  ```
  > z <- 0
  > for (i in 1:100){
  + z <- z+i; print(z)
  + }
  ```

- `while (Bedingung) {`**R**`-Ausdrücke}`
  Solange die Bedingung erfüllt ist, wird die Schleife nicht verlassen. Das folgende Programm addiert Zahlen, bis deren Summe größer als 1000 ist:

  ```
  > n <- 0; summe <- 0
  > while(summe <= 1000){
  + n <- n+1
  + summe <- summe+n
  + }
  > print(summe)
  [1] 1035
  > print(n)
  [1] 45
  ```

Bei geschwindigkeitskritischen Teilen sollte man Schleifen vermeiden und stattdessen lieber eingebaute Funktionen nutzen. Beispiel: `sum()` statt aufaddieren in einer Schleife, Matrizenrechnungen.

5 Verteilungsmodelle und Simulation von Zufallsvariablen

In **R** sind eine ganze Reihe theoretischer Verteilungen implementiert, auf deren Dichte, Verteilungsfunktion etc. zugegriffen werden kann. Die Abfragen sind bei allen Verteilungen folgendermaßen aufgebaut:

`d`*verteilung*() (Zähl-)Dichte

`p`*verteilung*() Verteilungsfunktion

`q`*verteilung*() Quantil

`r`*verteilung*() Zufallszahl

verteilung: norm, unif, exp, pois, binom, t, f, chisq etc.
exakte Aufrufe siehe `help()`.

Beispiel 1

`dnorm()`	Dichte der Normalverteilung
`pnorm()`	Verteilungsfunktion der Normalverteilung
`qnorm()`	Quantil der Normalverteilung
`rnorm()`	Zufallszahlengenerator von normalverteilten ZVen

Beispiel 2

Es sollen 100 Realisierungen von normalverteilten ZV mit Erwartungswert -1 und Varianz 4 erzeugt werden:

```
> x <- rnorm(100, mean = -1, sd = 2)  # x ist nun ein Vektor mit 100 Einträgen
> print(x[1])                         # Liefert den ersten realiserten Wert
[1] -0.3020991
```

6 Graphiken

R besitzt vielseitige Graphik-Routinen. Sie können nur bei einem geöffneten graphischen Device aktiv werden. Unter Windows leistet dies die Anweisung win.graph(), unter UNIX oft motif().

`motif()` bzw. `win.graph()`	Öffnen eines Graphikfensters (normalerweise nicht notwendig!)
`dev.off()`	Schließen eines Graphikfensters; z.B. dev.off(2)
`par()`	ermöglicht die (allgemeine) Kontrolle über den Graphikbereich.

Um zum Beispiel eine 2x2-Matrix von Bildern in einem Bereich zu erzeugen, kann der Befehl `par(mfrow=c(2,2))` benutzt werden - die Bilder werden zeilenweise erzeugt; vgl. `mfcol`.

6.1 Graphische High-Level-Routinen

Mit folgenden Funktionen lassen sich Grafiken in **R** erzeugen. Bei diesen Funktionen wird jeweils ein Grafikfenster von selbst geöffnet.

`plot(x,y)`	erzeugt Scatterplot (isolierte Punkte!); z.B. `plot(x,y)`
`plot(`*`Function`*`,a,b)`	Erzeugt einen Plot von *Function* im Intervall von a bis b
`curve(`*`Function`*`,a,b)`	Erzeugt einen Plot von *Function* im Intervall von a bis b
`barplot(x)`	erzeugt Balkendiagramm
`boxplot(x)`	erzeugt Boxplot
`hist(x)`	erzeugt Histogramm
`truehist(x)`	erzeugt Histogramm (Paket MASS muß geladen werden)
`pairs(x)`	erzeugt paarweise Scatterplots
`qqplot(x)`	erzeugt QQ-Plot
...	

Aufruf von motif() bzw. win.graph() ist für diese Funktionen nicht notwendig, wenn nur ein Fenster geöffnet werden soll.

Daneben können der Funktion eine Vielzahl von Parametern mitgegeben werden, die das endgültige Layout verbessern können. Hier ist eine kleine Auswahl:

`type="p"`	Daten als Punkte
`type="l"`	Daten als Linien
`type="b"`	Daten als Punkte und Linien
`xlim=c(1,100)`	Grenzen der x-Achse (z.B.: 1 und 100)
`ylim=c(0,1)`	Grenzen der y-Achse (z.B.: 0 und 1)
`xlab="x-Achse"`	Beschriftung der x-Achse
`ylab="y-Achse"`	Beschriftung der y-Achse
`log="xy"`	x- und y-Achse logarithmisch; auch `log="x"`, `log="y"`
`main="Testplot"`	Überschrift zum Bild
`add=TRUE`	Fügt Grafik in bereits existierende Grafik hinzu, wenn nicht angegeben (d.h. add=FALSE) wird alte Grafik gelöscht

Die aufgelisteten Parameter können mit den gewünschten Werten durch Kommata getrennt nach den Daten der Funktion plot() übergeben werden. Diese Parameter lassen sich, wenn sie dort Sinn machen, auch für die anderen aufgeführten High-Level-Plot-Funktionen verwenden. Wem das noch nicht reicht, kann über help(par) weitere Parameter ausfindig machen und deren Bedeutung ermitteln. Mit der Funktion par() lassen sich eine Reihe von Abfrage- und allgemeinen Änderungswünschen zu den graphischen Einstellungen realisieren.

6.2 Graphische Low-Level-Routinen

Bei Ergänzungswünschen zu einer Graphik helfen folgende Funktionen. Sie öffnen kein eigenes Fenster, sondern ergänzen schon existierende Plots.:

`abline`	`abline(2,3)`	zeichnet Geraden (hier: y=2+3*x)
	`abline(h=1)`	horizontale Linie f(x)=1
	`abline(v=5)`	vertikale Linie f(y)=5
`lines(x,y)`		Polygonzug durch die Punkte (x,y), x,y: Vektoren gleicher Länge
`segments(x1,y1,x2,y2)`		zeichnet Strecken von (x1,y1) nach (x2,y2)
`points(x,y)`		zeichnet die Punkte (x,y) x,y: Vektoren gleicher Länge
`title("Entwicklung")`		Überschriften
`text(x,y,textxy)`		zeichnet den Text "textxy" an den Stellen (x,y)
`text(locator(1),textxy)`		Mit der Maus kann in die Grafik geklickt werden. An dieser Stelle wird der Text "textxy" geschrieben
`symbols(x,y,circles=z)`		Kreise um die Punkte (x,y) vom Umfang z (auch: squares, stars etc.)
`legend`		Legende `legend(x=1,y=20, legend=c("DG","C"), lty=c(1,2))`

6.3 Plotten von Funktionen

Wir wollen die Dichte der Normalverteilung mit Mittelwert $\mu = 2$ und Varianz $\sigma^2 = 4$ gemeinsam mit der Verteilungsfunktion plotten. Hier gibt es verschiedene Möglichkeiten. Zuerst verwenden wir den Befehl *plot* mit der Funktion *pnorm* um die Verteilungsfunktion zu plotten:

```
> par(mfrow=c(1,3))
> plot(function(x) pnorm(x,mean=2,sd=2),-5,9,main="Mit plot(Funktion)",ylab="f(x)")
```

Nun soll die Dichte in die gleiche Grafik hinzugefügt werden. Das kann dadurch gemacht werden, dass die Option $add = TRUE$ gesetzt wird. Außerdem soll die Kurve gestrichelt und blau sein:

```
> plot(function(x) dnorm(x,mean=2,sd=2),-5,9,add=TRUE,col="blue",lty=2)
```

Eine andere Möglichkeit bietet sich mit der Funktion *curve* :

```
> curve(pnorm(x,mean=2,sd=2),-5,9,main="Mit curve(Funktion)",ylab="f(x)",lty=3)
> curve(dnorm(x,mean=2,sd=2),-5,9,add=TRUE,col="green",lty=2)
```

Als letztes lassen sich die Plots auch als Scatterplots zeichnen, wenn die Option *type* ="*l*" gesetzt ist:

```
> x <- seq(-5,9,length=100)
> y <- pnorm(x,mean=2,sd=2)
> plot(x,y,main="Mit plot(Funktion)",ylab="f(x)",type="l",col="red")
> y <- dnorm(x,mean=2,sd=2)
> lines(x,y)
```

Dies liefert den Plot in Abbildung 1.

6.4 Plotten von Histogrammen

Wir wollen nun ein Histogramm von diskreten Werten plotten. Dazu wollen wir 3 Möglichkeiten vorstellen. Zuerst simulieren wir 1000 Binomialverteilte Zufallsvariablen mit Parameter $n = 10$ und $p = 0.7$:

```
> x <- rbinom(1000,10,0.7)
> par(mfrow=c(1,3))
> hist(x,freq=FALSE,main="Histogramm mit hist")
```

Die Option `freq=FALSE` liefert die relativen Häufigkeiten, mit `freq=TRUE` (default-Wert) bekommen wir die absoluten Häufigkeiten. Nun wollen wir die Daten mit `truehist` plotten. Hierzu muß unter $Pakete- > Lade\ Paket\ldots$ das Paket $MASS$ geladen werden (Das Gleiche kann mit dem Befehl `library()` erreicht werden). Falls ein Paket nicht installiert ist, dann kann es unter $Pakete- > Installiere\ Paket(e)$ heruntergeladen und installiert werden. Nun plotten wir die Daten mit *truehist*:

```
> truehist(x,freq=FALSE,main="Histogramm mit truehist")
```

Als 3. Möglichkeit gibt es noch den *barplot* für diskrete Werte. Dazu müssen die Daten in ein *table* umgewandelt werden:

```
> t <- table(x)
> rt <- t/sum(t)
> barplot(rt,main="Histogramm mit barplot")
```

Dies liefert dann den Plot in Abbildung 2.

6.5 Graphiken ausdrucken

Ist eine Graphik im entsprechenden Bereich bereits erstellt worden, kann diese mit `dev.copy(postscript, 'name.ps')` als post-script Datei abgespeichert werden (der aktive Graphikbereich wird verwandt; vgl. `dev.cur()`). Es ist wichtig, dass nach dem Kopierbefehl ein `dev.off()` eingegeben wird, was den Vorgang sozusagen abschliesst. Das Bild steht nun als name.ps zur Verfügung.

Die zweite Möglichkeit ist die, das Bild frisch für die Datei zu erzeugen. Mit dem Befehl `postscript('name.ps')` wird eine leere Datei name.ps angelegt. Alle nun folgenden Graphikbefehle werden nicht an das Device motif() geschickt sondern in die Datei geschrieben. Ist der letzte nötige Graphikbefehl eingegeben worden, dann muss wie oben die Erzeugung mit `dev.off()` abgeschlossen werden.

Unter Windows ist es auch möglich, eine Graphik mit Hilfe der Maus abzuspeichern. Hierzu einfach mit der rechten Maustaste auf die zu speichernden Graphik klicken und dann *Abspeichern als Postskript* auswählen.

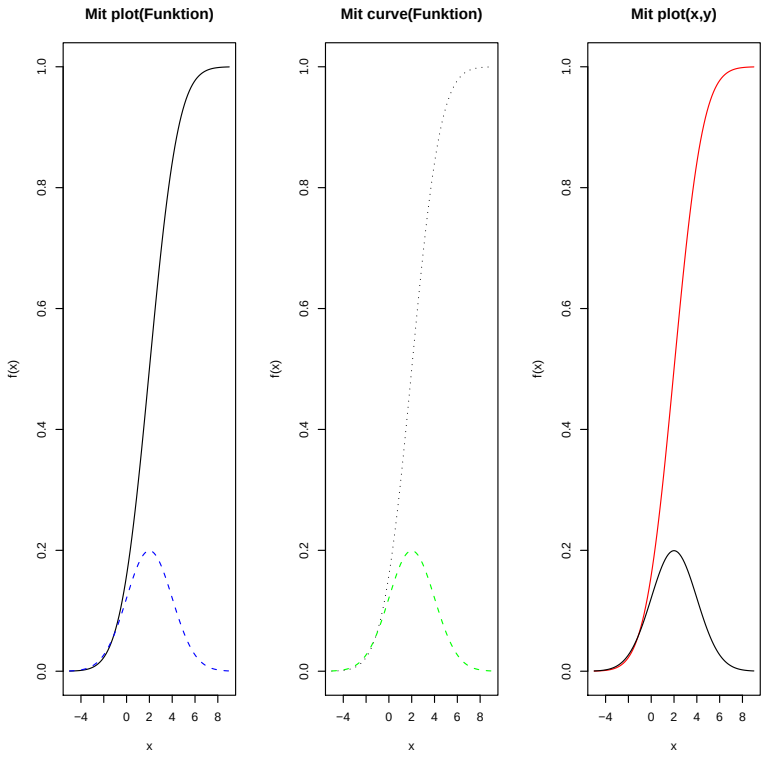

Abbildung 1: Die Verteilungsfunktion und die Dichte der Normalverteilung.

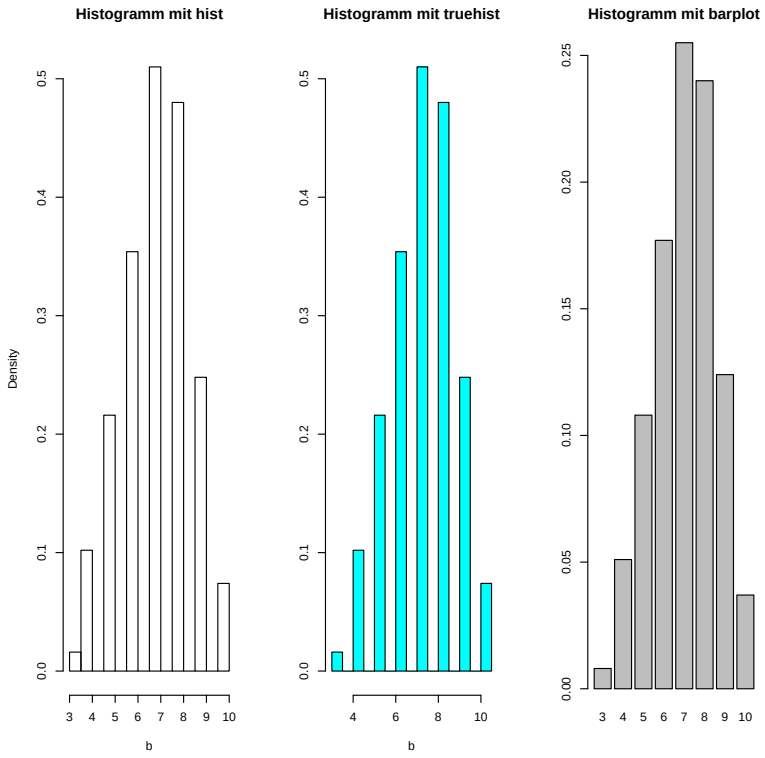

Abbildung 2: 3 verschiedene Histogramme zu den selben Daten

7 Umgang mit Daten-Dateien, Skript-Dateien und Paketen

7.1 Skript-Dateien einlesen

In **R** gibt es auch die Möglichkeit, in einer externen Datei ein Skript zu schreiben, das in **R** aufgerufen wird und die Befehle dann sukzessive abarbeitet. Sinnvoll hierbei ist, dass die Endung des Skripts '.R' lautet, z.B. 'test.R', da solche Dateien von **R** standardmäßig als Skripte erkannt werden.

Der Aufruf des Skripts 'test.R' erfolgt dann in **R** einfach mit dem Befehl source(), z.B.

```
> source("C:\\Programme\\R-2.4.1\\Skripte\\test.R")    # Windows
> source("/home/login/verzeichnis/test.R" )    # Linux
```

Beachte: Die jeweiligen Pfade müssen an die EIGENEN Werte angepasst werden!

Unter Windows kann ein Skript auch einfach durch das Aufrufen unter *Datei*, *Lese* ***R*** *Code ein ...* geschehen. Beachte, dass dabei die Konsole aktiv sein muss!

7.2 Zusatzpakete

Für **R** gibt es viele zusätzliche Pakete, welche meist frei im Netz verfügbar sind. Diese können einfach installiert werden. Unter Windows kann dies wie folgt geschehen.

Unter *Pakete* den Punkt *Installiere Paket(e)* auswählen. Anschließend einen (möglichst nahe gelegenen) Verteiler auswählen und mit *ok* bestätigen. Im nächsten Schritt einfach das gewünschte Paket markieren und wieder bestätigen.

Auf der Shell wird dafür der Befehl `install.packages(''Paketname'')` benutzt.

Die bereits installierten Pakete können dann einfach mit dem Befehl

```
> library(Paketname)
```

für das aktuelle Workspace geladen werden.

Beachte

In **R** ist es möglich, eigene Pakete zu schreiben und auch (frei) zu verteilen. Dementsprechend KANN natürlich jedes benutzte/installierte Paket auch Fehler enthalten! Ein Vorteil einer open–source Software wie **R** ist es dagegen, dass der Quellcode zur Verfügung steht. Somit kann man jedes Paket nach seinen eigenen Bedürfnissen anpassen (bzw. auf seine Richtigkeit prüfen).

7.3 Daten aus Dateien einlesen und in Dateien schreiben

Neben der sehr einfachen Funktion `scan()` (siehe 2.1.1), gibt es in **R** noch die Möglichkeit, Tabellen aus externen Dateien einzulesen.

Das leistet die Funktion

```
read.table(file="myfile", header=TRUE/FALSE)
```

Die Option `header=TRUE` muß verwendet werden, falls die Spaltenbezeichnungen in der Datei *myfile* in der ersten Zeile stehen. Ein Data Frame kann mit

```
write.table(Data Frame, file= "myfile")
```

in eine Datei geschrieben werden.

Falls keine Spaltenüberschriften in den eingelesenen Daten vorhanden sind (oder nicht mit eingelesen wurden), kann z.B. mit `data$V1` auf die erste Spalte zugegriffen werden.

Beispiel 1

Es liegt eine Datei *myfile.dat* im aktuellen Arbeitsverzeichnis mit folgender Datenstruktur vor:

```
   Gewicht  Groesse  Geschlecht  Alter
1       65      168           m         22
2       75      175           w         45
```

Dann kann sie wie folgt eingelesen werden:

```
> data <- read.table("myfile.dat") # Einlesen
> print(data)
  Gewicht Groesse Geschlecht Alter
1      65     168          m    22
2      75     175          w    45
> print(x$Groesse)
[1] 168 175
```

Beachte
Bei der Funktion `read.table` wird der Parameter `header` automatisch auf $TRUE$ gesetzt, wenn die erste Zeile kürzer als die restlichen Zeilen ist. Andernfalls wird `header` auf $FALSE$ gesetzt.

Beispiel 2
Es liegen keine Spaltenüberschriften vor:
Z.B. *myfile1.dat* hat folgendes Aussehen:

```
 1  65  168  m  22
 2  75  175  w  45
```

```
> data <- read.table("myfile1.dat")
> print(data)
  V1 V2  V3 V4 V5
1  1 65 168  m 22
2  2 75 175  w 45
> print(data$V2)  #Zugriff auf zweite Spalte
[1] 65 75
```

Beachte
Falls keine Spaltennamen eingelesen werden, vergibt **R** die Standardnamen V1,. . ., Vn (bei n Spalten).

8 Statistische Methoden

In diesem Abschnitt gehen wir immer davon aus, dass wir eine Stichprobe haben, welche in einem Vektor `x` zusammengefasst ist. Diese Daten können sowohl von Simulationen als auch aus Beobachtungen stammen.

8.1 Methoden zur beschreibenden Statistik

8.1.1 Grundlegende Kennzahlen

Wichtige Kennzahlen, sog. *Summary–Statistics*, können mit dem Befehl `summary()` berechnet werden. Dabei wird der kleinste Wert, das 1. Quartil (= 25%–Quantil), der Median, der Mittelwert, das 3. Quartil (= 75%–Quantil) und der größte Wert ausgegeben.

Beispiel mit 20 standardnormalverteilten Zufallsvariablen

```
> x <- rnorm(20)
> summary(x)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-1.24200 -0.60610 -0.04775  0.04813  0.46660  2.25200
```

Eine ganz ähnliche Funktion ist `quantile()`:

```
> quantile(x)
         0%         25%         50%         75%        100%
-1.24182113 -0.60606758 -0.04774584  0.46655292  2.25224109
```

Sie liefert fast den gleichen Output.

Weitere grundlegende Funktionen sind `mean()` für den empirischen Erwartungswert und `var()` für die empirische Varianz.

8.1.2 Lorenz-Kurve und Gini-Koeffizient

Mit dem Paket *ineq* hat man einfache Methoden zur Hand zum Plotten der Lorenz-Kurve und zum Berechnen des Gini-Koeffizienten. **Achtung:** Leider kommt *ineq* nur mit nicht-negativen Daten zurecht.

Aufruf:

```
> x <- runif(20, 0, 1)
> Lc(x, plot = TRUE)
$p
 [1] 0.00 0.05 0.10 0.15 0.20 0.25 0.30 0.35 0.40 0.45 0.50 0.55 0.60 0.65 0.70
[16] 0.75 0.80 0.85 0.90 0.95 1.00

$L
 [1] 0.000000000 0.004025828 0.012682614 0.026918337 0.051859219 0.076972735
 [7] 0.107088853 0.141916975 0.178845468 0.215844258 0.259489901 0.304094197
[13] 0.361773845 0.427813587 0.498314116 0.569668756 0.643819002 0.726159163
[19] 0.813037502 0.905588337 1.000000000

$L.general
 [1] 0.000000000 0.002021202 0.006367418 0.013514588 0.026036376 0.038644838
 [7] 0.053764900 0.071250665 0.089790939 0.108366506 0.130279185 0.152673163
[13] 0.181631738 0.214787570 0.250182980 0.286007205 0.323234987 0.364574589
[19] 0.408192622 0.454658582 0.502058787

attr(,"class")
[1] "Lc"
> Gini(x)
[1] 0.3174087
```

A Anhang

A.1 Unterschied R und S-Plus

R und S-Plus sind sehr ähnlich, da sie auf dasselbe Programm S aufbauen. Mit beiden können die verschiedensten Fragestellungen, z.B. aus der Statistik, bearbeitet werden. Die hier vorgestellten, sowie die meisten wichtigen Befehlsaufrufe sind im Wesentlichen gleich. Dennoch kann sich bei einzelnen (z.T. auch hier vorgestellten) Funktionen die Eingabe der Parameter unterscheiden. Wir konzetrieren uns im Weiteren auf **R** und übernehmen keine Garantie, dass die Programme auch auf S-Plus laufen.

Der wohl entscheidenste Unterschied ist, dass **R** eine open–source Sofware ist, wohingegen S-Plus kommerziell vertrieben wird. Dementsprechend ist bei S-Plus z. B. eine graphische Oberfläche gleich dabei, während **R** an für sich nicht graphisch ist, es gibt aber mehrere gute Oberflächen aus seperaten Projekten, z.B. rkward für Linux. Für die Windows-Version wurde bereits eine Oberfläche integriert. Ein weiterer wichtiger Unterschied, der sich daraus ergibt, ist, dass davon ausgegangen werden kann, dass die Methoden/Funktionen/Algorithmen in S-Plus professionell geprüft sind, wohingegen in **R** z.T. Fehler auftreten können, allerdings hauptsächlich in Zusatzpaketen. Ein Vorteil von **R** ist, dass es viele Zusatzpakete gibt, welche kostenlos installiert und benutzt werden können, sowie selbständig weiterentwickelt werden können.

A.2 S-Plus allgemein

S-Plus ist ein sehr mächtiges Statistikprogramm. Im Grunde ist es eine Programmierumgebung, in der sich sehr viel realisieren lässt. Die UNIX- und WINDOWS-Versionen unterscheiden sich nur unwesentlich.

- Start von S-Plus: z.B. **localhost$ Splus** ↩
- Beenden von S-Plus: q() ↩
- Prompt: >
- Bestätigung einer Eingabe: ↩
- "+" als Prompt bedeutet: Eingabe kann/muss fortgesetzt werden
- Mehrere Befehle in einer Eingabeziele durch " **;** " trennen
- **Hilfe:** > help(*name*) oder > ?*name*
- Kommentare mit # einleiten

Variablen

> objekt1 <− 1.043 # d.h.: GleitPUNKTzahlen, **Zuweisung** mit <−

> objekt1 <− 3 # überschreibt die erste Definition

> Objekt1 <− 2 # nicht = "objekt1" (Groß-/Kleinschreibung beachten!)

Definierte Objekte werden (z.B.) unter /home/thales/user/MySwork/.Data gespeichert. Bereits belegte Variablennamen: z.B. pi, t, f, T, F, mean, var,...

- ls(): listet alle gespeicherten Objekte auf
- rm(objekt1): entfernt objekt1 (remove(ls()): entfernt alle gespeicherten Objekte!)

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

Q

R

S

T

W