Einführung zu R bzw. S-Plus*

14. November 2007

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

Inhaltsverzeichnis

1 Einleitung

Wir wollen mit ein paar allgemeinen Informationen über R und S-Plus beginnen.

1.1 Unterschied R und S-Plus

R und S-Plus sind sehr ähnlich, da sie auf das selbe 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 für S-Plus auch eine graphische Oberfläche vorhanden, was bei R nicht der Fall ist. Ein weiterer wichtiger Unterschied, der sich daraus ergibt, ist, dass davon ausgegangen werden kann, dass die Methoden/Funktionen/Algorithmen in S-Plus geprüft sind, wohingegen in R z.T. doch Fehler auftreten können. Der Vorteil von R ist, dass es viele Zusatzpakete gibt, welche kostenlos installiert und benutzt werden können, sowie selbständig weiterentwickelt werden können.

1.2 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. **thales$ Splus** ↩
- Beenden von S-Plus: q() ↩
- Befehlszeile: >
- Bestätigung einer Eingabe: ↩
- "+" zu Beginn der Befehlszeile: 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!)

1.3 R allgemein

R ist, ebenso wie S-Plus, 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. Im Folgenden eine kleine Liste der wichtigsten R–Befehle:

- Start von R: z.B. **dublin$ R** ↩ # Sollte demnächst auch auf thales möglich sein!!
- Beenden von R: q() ↩
- Befehlszeile: >
- Bestätigung einer Eingabe: ↩
- "+" zu Beginn der Befehlszeile: Eingabe kann/muss fortgesetzt werden
- Mehrere Befehle in einer Eingabeziele durch "**;**" trennen
- **Hilfe:** > help(*name*) oder > ?*name*
- Kommentare mit # einleiten

Variablen wird ihr Wert mit '<−' zugewiesen. Bei neueren Versionen ist eine Zuweisung auch mit '=' möglich. 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,...

Kontrolle der im aktuellen Workspace vorhandenen Objekte:

- getwd(): liefert das aktuelle Arbeitsverzeichnis
- setwd("*Pfad*"): ändert das aktuelle Arbeitsverzeichnis in das in *Pfad* spezifizierte Verzeichnis
- ls(): listet alle gespeicherten Objekte auf, d.h. beim Arbeiten auf der Shell liefert es alle bisher genannten Objekte, welche 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
- load("*Dateiname*"): lädt ein mit save in der Datei *Dateiname* gespeicherten Workspace

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

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 standartmäßig 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\\Scripte\\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!

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 werden dafür folgende Befehle benutzt:
> chooseCRANmirror() # Öffnet Liste mit möglichen Verteilern,
hier muss einmalig ein Verteiler definiert werden
> utils:::menuInstallPkgs() # Öffnet Liste mit verfügbaren Paketen

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)(Pakete sind ähnlich wie Funktionen, deshalb siehe hierzu auch Abschnitt 2.1.5).

2 Datentypen, Datenverarbeitung, vorgefertigte Funtionen und Programmieren in R

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. Insbesondere sei auf die implementierten Verteilungsmodelle in Abschnitt 2.4 verwiesen, da diese für statistische Anwendungen enorm wichtig sind. Es soll aber auch gezeigt werden, wie man in R eigene Programme und Funktionen schreiben kann.

2.1 Datengenerierung und -strukturierung

2.1.1 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. Systematisch werden Objekte durch den Befehl **c(element1,element2,...)** zu einem Vektor verbunden - das c = concatenate.

Weitere Möglichkeiten zur Generierung von Vektoren sind:

> rep(): Syntax: rep(x, times=n) # wiederhole Objekt x times 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)

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

Beispiel für den Zugriff auf Einträge von Vektoren:
> x <− c(1.4, 3.7, 2.0, 4.6, 5.1)
> x[2]
[1] 3.7

```
> x[1:4]      # Die ersten 4 Elemente
[1] 1.4  3.7  2.0  4.6
> 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 nächsten Abschnitt.

2.1.2 Matrizen

Syntax: matrix(data, nrow=, ncol=, byrow=F) [Dimension von Matrizen: **dim()**]
Matrizen können nur Variablen **eines** Typs enthalten (Zahlen, Zeichenketten,...).

Beispiele:

```
> m <- matrix(0, 4, 5)      # eine 4 x 5 mit lauter Nullen
> 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

> dim(m)
[1] 5 2
> mm <- matrix( scan("mfile"), ncol=5, byrow=TRUE)
  # zeilenweise Einlesen aus Datei
```

siehe auch: **rbind(vektor1, vektor2,...)** bzw. **cbind(vektor1, vektor2,...)**

Zugriff auf Einträge von Matrizen:

```
> x <- c(1.4,3.7,2.0,4.6,5.1);  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]

Eine Matrix wird mit dem Befehl **t()** transponiert und mit %*% können Matrizen multipliziert werden. Die Inverse läßt sich durch **solve()** bestimmen:

```
> m <- matrix(1:10, 5)      # eine 5 x 2 welche die Zahlen 1 bis 10 enthält
> 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   -2.553513e-15
[2,] 6.661338e-16    1.000000e+00
```

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

2.1.3 Listen

Listen sind Sammlungen von beliebigen Objekten. Sie sind ähnlich wie Vektoren, nur können in einer Liste (im Gegensatz zu Vektoren) **verschiedene** Objekttypen zusammengefasst werden. Eine Liste wird durch den Befehl **list()** erzeugt.

Beispiel
Aufruf erstes Element:

```
> x <- list(Zahlen = c(1,2,3), Buchstaben = c(a,b), Beides = c("a",1,"b",2))
# Anlegen einer Liste
> x[1]       # Gibt nicht nur den Vektor "Zahlen" zurück
$ Zahlen
[1] 1 2 3
> x[[1]]       # Liefert nur den Vektor "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
Auf die gleiche Weise kann auch auf Ausgaben von R Funktionen zugegriffen werden, siehe

Abschnitt 2.1.5.

2.1.4 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. Es kann mit folgenden Befehlen erzeugt werden:

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

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() vewenden!!

> 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 in 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. Dies macht die Funktion

$$read.table(file = \text{"}myfile\text{"}, header = T/F).$$

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

$$write.table(DataFrame, file = \text{“}myfile\text{“})$$

in eine Datei geschrieben werden.

Falls keine Spaltenüberschriften in den eingelesenen Daten vorhanden sind (oder nicht mit eingelesen werden 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
> data <- read.table("myfile.dat")      # Einlesen
> print(data)
     Gewicht   Groesse   Geschlecht   Alter
 1       65        168           m        22
 2       75        175           w        45
> print(data$VGroesse)
[1] 65 75
```

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   65   168   m   22
   2   75   175   w   45
> print(data$V2)       #Zugriff auf zweite Spalte
[1] 65 75
```

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

min()	Minimum		exp()	Exponentialfkt
max()	Maximum		log()	Logarithmus(Basis: e)
sort()	Sortieren		logb(x,base=2)	Logarithmus(Basis: base)
sqrt()	Wurzel		sum()	Summe der Vektoreinträge
sin()	Sinus		prod()	Produkt aller Vektoreinträge
cos()	Cosinus		diff()	Vektor mit Differenzen der Vektoreinträge
tan()	Tangens		range()	kleinster und größter Vektoreintrag
			integrate()	Integration

Tabelle 1: Liste wichtiger Funktionen

2.1.5 Funktionen und Operatoren

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. Eine Liste wichtiger Funktionen ist in Tabelle 1 zusammengestllt.

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] 1.0000 1.5166 2.0494 1.7889

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 die Bedingung "()" dargestellt werden.

Beispiel
$\mathbb{1}_{\{y:y<1\}}(x)$ soll dargestellt 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.
# Beachte, das Ergebnis ist eine Matrix
     [,1]
[1,]   20
> t(x) %*% y       # das selbe Ergebnis wie ohne transponieren
     [,1]
[1,]   20
> x %*% t(y)       # Liefert eine 3×3 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)
[1] 10       # Summe aller Elemente
```

```
> apply(m,1,sum)        # Zeilensummen
#“1“: zeilenweise, “2“: spaltenweise
[1]  3  7
```
allgemein: apply(Matrix, Margin=1/2, FUNKTION)

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.

Beispiel
Definition einer 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

2.2 Konditionalabfragen

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 **und** test2==TRUE }
if (test1 || test2) { Anweisungen für test1 **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
für die anderen FALSE
[1] F F F F F T T T T T

> 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

2.3 Schleifen in R

- **for (Variable in Werte) {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
```

Beachte
R ist eine interpretierte Sprache, d.h. der Code wird erst während de Laufzeit übesetzt. Deshalb sollte auf die Verwendung von Schleifen möglichst verzichtet werden.

2.4 Verteilungsmodelle

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() Verteilung der Normalverteilung
qnorm() Quantil der Normalverteilung
rnorm() Zufallszahlengenerator von normalverteilten ZV

Beispiel 2
Es sollen 100 Realisierungen von normalverteilten ZV mit Erwartungswert 2 und Varianz 9 erzeugt werden:

```
> x <- rnorm(100, mean = 2, sd = 3)      # x ist nun ein Vektor
# mit 100 Einträgen
> print(x[1])      # Liefert den ersten realiserten Wert, hier z.B.:
[1] 2.778157
```

3 Graphiken in R

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.

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=T	Fügt Grafik in bereits existierende Grafik hinzu, wenn nicht angegeben (d.h. add=F) 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.

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))

Beispiel: 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 $add = T$ gesetzt wird. Außerdem soll die Kurve gestrichelt und blau sein:

```
>plot(function(x) dnorm(x,mean=2,sd=2),-5,9,add=T,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=T,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.

Beispiel: 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 $p = 0.7$:

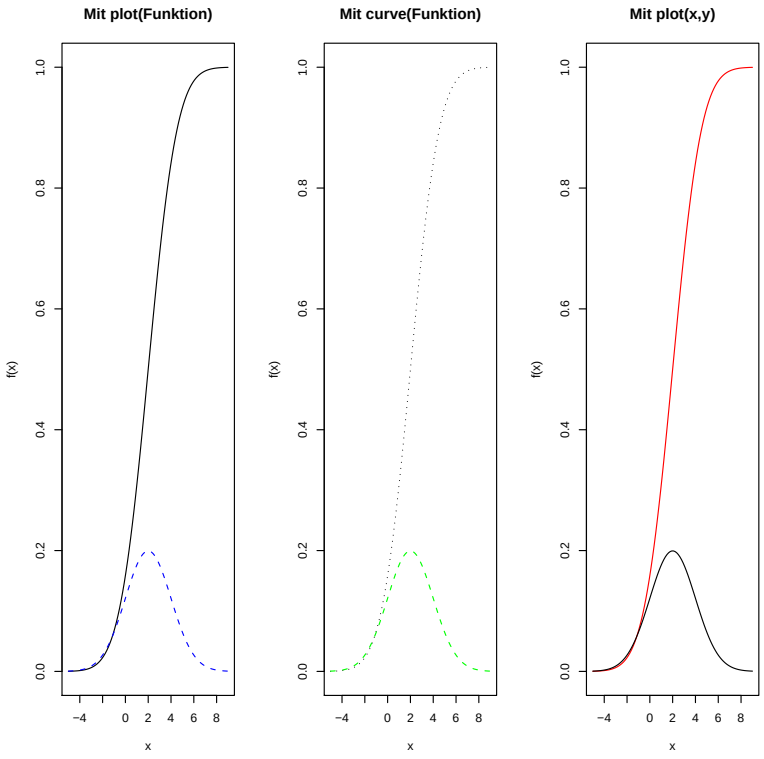

Abbildung 1: Die Verteilungsfunktion und die Dichte der Normalverteilung.

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

Die Option $freq = F$ liefert die relativen Häufigkeiten, mit $freq = T$ (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=F,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.

Von der Graphik am Bildschirm zum Ausdruck

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.

4 Sonstige interessante/wichtige Kommandos

- 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

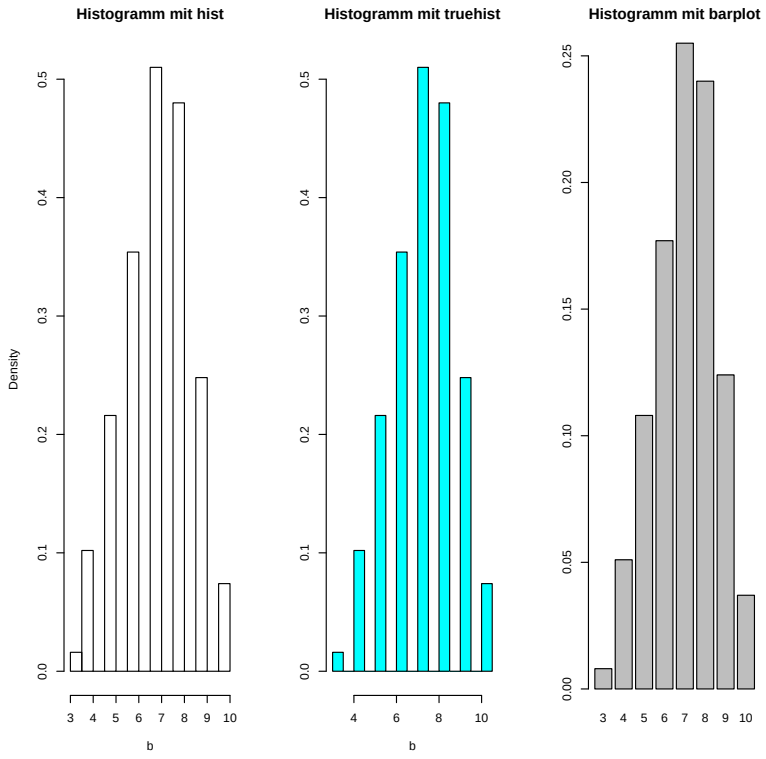

Abbildung 2: 3 verschiedene Histogramme zu den selben Daten

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.

- 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 standartmäßig 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>
  ```

- 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
  ```

- quantile(x)

0%	25%	50%	75%	100%
1	1.75	2.5	3.25	4

- 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"  "parameters"  "p.value"  "conf.int"  "estimate"
  [6] "null.value"  "alternative"  "method"  "data.name"
  ```

5 Optimierung und Integration

Nun wollen wir Möglichkeiten zur numerischen Optimierung mit R diskutieren. Diese können insbesondere zur Berechnung von Maximum-Likelihood-Schätzern verwendet werden, siehe auch Abschnitt 6.1. AuSSerdem soll auch noch die numerische Integration mit R erwähnt werden.

5.1 Optimierung

In R stehen mehrere Methoden zur numerischen Optimierung zur Verfügung. Eine kleine Liste ist im Folgenden gegeben. Alle Methoden sind im Paket *stats* enthalten. Beachte, dass es weitere Parameter zu den hier erwähnten Funktionen gibt, welche z.B. auf den Hilfeseiten von R genauer beschrieben sind.

nlm()	Minimierung (auch mehrdimensional) mit Newton–Verfahren
optim()	Sammlung von (mehrdimensionalen) Optimierungsmethoden
optimize()	(Eindimensionale) Optimierung mit Nebenbedingungen

Beachte, dass durch die Vertauschung des Vorzeichens aus jeder Maximierungsaufgabe eine Minimierungsaufgabe gemacht werden kann, d.h. für eine Funktion f gilt:

$$arg\,max f = arg\,min(-f)\,.$$

Die Methode *optim()* bietet eine große Auswahl an Optimierungsmethoden. Hierbei kann zwischen direkten Optimierungsmethoden, Gradientenverfahren oder Simulated–Annealing ausgewählt werden.

Beispiele

```
> fr <- function(x) { # die zu minimierende Funktion
+ x1 <- x[1]
+ x2 <- x[2]
+ 100 * (x2 - x1 * x1)^ 2 + (1 - x1)^ 2
+}


>grfr <- function(x) { # Gradientenfunktion von 'fr'
+ x1 <- x[1]
+ x2 <- x[2]
+ c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
+ 200 * (x2 - x1 * x1))
+}


# Die nlm()–Methode
# Minimierung mit Newtonverfahren
# Hier wird die Funktion fr minimiert. Der Startwert der numerischen Minimierung
# ist durch den Vektor c(-1.2, 1) gegeben.
> out <- nlm(fr,c(-1.2, 1))


# Die optim()–Methode
# Minimierung mit Nelder–Mead
# Hierbei wird nur der Startpunkt und die zu minimierende
# Funktion übergeben
```

```
> out <− optim(c(-1.2,1), fr)
# Minimierung mit einem Quasi–Newton–Verfahren
# Hierbei wird zusätzlich die Gradientenfunktion 'grfr' von 'fr'
# und der Name der Methode übergeben
> out <− optim(c(-1.2,1), fr, grfr, method = "BFGS" )
# Minimierung mit einem Konjugierten–Gradientenverfahren
> out <− optim(c(-1.2,1), fr, grfr, method = "CG" )
# Modifiziertes Quasi–Newton–Verfahren
> out <− optim(c(-1.2,1), fr, grfr, method = "L-BFGS-B" )


# Die optimize()–Methode
> f <- function(x){
+ x^ 2
+ }


# Optimieren der Funktion f. Hierbei ist der Startwert x=1,
# die untere Schranke ist -10, die obere Schranke 20.
> out <− optimize(f,1,-10,20)
```

Hinweis

- Bei der Optimierung mit der Funktion *optim()* ist die Übergabe des Gradienten nicht erforderlich! Dieser wird dann numerisch bestimmt, was allerdings numerisch nicht stabil ist. Deshalb ist es wie in obigem Beispiel ratsam, die Gradientenfunktion mit zu übergeben.
- Auf das Minimum, das die Funktion *nlm()* liefert, kann mit *out$minimum* zugegriffen werden; auf die Minimumstelle mit *out$estimate*. Beides sind Vektorwerte.
- Auf das Minimum, das die Funktion *optim()* liefert, kann mit *out$value* zugegriffen werden; auf die Minimumstelle mit *out$par*. Beides sind Vektorwerte.
- Auf das Minimum, das die Funktion *optimize()* liefert, kann mit *out$minimum* zugegriffen werden; auf die Minimumstelle mit *out$objective*. Beides sind Vektorwerte.

5.2 Integrieren

In R gibt es die Möglichkeit zur numerischen Integration von Funktionen. Dies kann mit dem Befehl *integrate()* aus dem Paket *stats* erfolgen. Als Parameter müssen mindestens die zu integrierende Funktion sowie die untere und ober Integrationsschranke übergeben werden. Hierbei ist auch der Wert Unendlich (*Inf*) möglich, wobei dies in der Praxis zu (numerischen) Problemen führen kann.

Beispiel

```
> f <− function(x) { x ^ 2 }
# Integral von 0 bis 1
> integrate(f, lower = 0, upper = 1)
```

0.3333333 with absolute error < 3.7e-15
> f1 <− function(x) { 1 / (x ^ 2) }
Integral von 1 bis unendlich
> out <− integrate(f, lower = 1, upper = Inf)
1 with absolute error < 1.1e-14

Der Rückgabewert von integrate ist ein Objekt. Auf den Wert des Integrals kann in obigem Beispiel folgerndermaßen zugegriffen werden:
> out$value
1

6 Statistik mit R

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.

6.1 Maximum–Likelihood–Schätzer

Die Maximum–Likelihood–Methode (ML–Methode) gehört zur Gruppe der Punktschätzer, d.h. mit ihr können Parameter einer vorher festgelegten/bekannten Familie von Verteilungen bestimmt werden.

Für die Bestimmung des ML–Schätzers ist in R die Methode *mle()* aus dem Paket *stats4* implementiert. Diese kann wie im folgenden Beispiel aufgerufen werden. Hierbei ist zu beachten, dass *mle()* die negative Log–Likelihood–Funktion numerisch minimiert und so den ML–Schätzer bestimmt. Um die Log–Likelihood–Funktion anzugeben kann man für alle in R definierten Verteilungen die Option *log=T* setzen, siehe auch das folgende Beispiel. Beachte, dass hier der ML-Schätzer durch numerische Optimierung bestimmt wird, auch wenn unter der Normalverteilungsannahme eine analytische Lösung existiert. Aber für viele Verteilungsfamilien muß man zur Bestimmung des ML–Schätzers auf numerische Verfahren zurückgreifen.

Beispiel
# Normalverteilung
# Die negative Log–Likelihood–Funktion der Normalverteilung
> likelihood <− function(x , mu = 0, sigma= 1){
+ -sum(dnorm(x, mean = mu, sd = sigma, log = T))
+ }
# 100 Realisierungen standartnormalverteilter ZV
> x <- rnorm(100)
Berechnen des ML–Schätzers
> out <− mle(function(mu=1, sigma=2) likelihood(x, mu, sigma))
Hierbei wird zur Minimierung (standartmäßig) die Newton–Methode verwendet

```
# Im folgenden Beispiel wird die Minimierung durch die Nelder–Mead–Methode vorgenommen
# Beachte, dass Default-Werte für die Parameter angegeben werden müssen
> out <− mle(function(mu=1, sigma=2) likelihood(x, mu, sigma), method = "Nelder-Mead")
> out

Call:
mle(minuslogl = function(mu = 1, sigma = 2) likelihood(x, mu, sigma))

Coefficients:
        mu       sigma
0.03188329 0.89859088
```

Auf die Werte im Objekt *out* (Type: mle-class) kann dann folgendermaßen zugegriffen werden:

```
> coef(out)
        mu       sigma
0.03188329 0.89859088
> coef(out)[1]
        mu
0.03188329
> coef(out)[[1]]
[1] 0.03188329
```

Beachte

- In obigem Beispiel wurde für die Maximierung der Likelihood–Funktion die negative Log–Likelihood–Funktion betrachtet. Falls man die Maximierung allerdings direkt mit der Likelihood–Funktion durchführt, liefert R kein brauchbares Ergebnis, da es bei der numerischen Optimierung zu Problemen kommt!
- Eine weitere Möglichkeit, den MLS zu gewinnen besteht darin, das Optimierungsproblem direkt aufzustellen und mit den Methoden aus Abschnitt 5.1 zu lösen.

6.2 Konfidenzintervalle

Konfidenzintervalle können mit R leicht durch die Quantilfunktionen der Verteilungen berechnet werden. Als einfaches Beispiel betrachten wir das Konfidenzinterval zum Niveau $1-\alpha$ für den Erwartungswert μ einer normalverteilten Stichprobe $(X_1, \ldots, X_n)$ mit bekannter bzw. unbekannter Varianz σ^2.

Beispiel Varianz σ^2 bekannt

```
>x<-rnorm(20,1,1) # Simulieren normalverteilter Zufallsvariablen, Varianz 1
>confintlow <- function(a,x)  # Die untere Intervallgrenze
+ return(mean(x)+qnorm(a/2)*1/sqrt(length(x))
```

```
>confintup <- function(a,x)  # Die obere Intervallgrenze
+return(mean(x)+qnorm(1-a/2)*1/sqrt(length(x)))
>c(confintlow(0.05,x),confintup(0.05,x)) # Das konkrete 95% Konfidenzintervall
[1] 0.6150851 1.4916076
> # Plotten des Intervalls zu verschiedenen Niveaus
>plot(function(a) confintlow(a,x),0,1,ylim=c(0.4,1.7),ylab="KI")
>plot(function(a) confintup(a,x),0,1,ylim=c(0.4,1.7),ylab="KI",add=T)
```

Der Plot des Konfidenzintervalls ist in Abbildung 3 zu sehen.

Genauso kann auch das Konfidenzintervall bei unbekannter Varianz σ^2 berechnet werden. Hierzu wird die Quantilfunktion der t-Verteilung verwendet.

Beispiel Varianz σ^2 unbekannt

```
>x<-rnorm(20,1,1) # Simulieren normalverteilter Zufallsvariablen, Varianz 1
>tconfintlow <- function(a,x)  # Die untere Intervallgrenze
+ return(mean(x)+qt(a/2,length(x)-1)*sd(x)/sqrt(length(x))
>tconfintup <- function(a,x)  # Die obere Intervallgrenze
+return(mean(x)+qt(1-a/2,length(x)-1)*sd(x)/sqrt(length(x)))
>c(tconfintlow(0.05,x),tconfintup(0.05,x)) # Das konkrete 95% Konfidenzintervall
[1] 0.576115 1.530578
```

Beachte, dass beim Aufruf der Funktion *t.test* zum Test auf den Mittelwert auch automatisch ein Konfidenzintervall für den Erwartungswert μ berechnet wird. Auf ähnliche Weise lassen sich fast alle Konfidenzintervalle zu jedem Niveau berechnen. Dadurch wird es überflüssig in Tabellen Werte nachzuschauen.

6.3 Parametertests

In R sind eine Vielzahl an statistischen Tests implementiert. Wir wollen in diesem Abschnitt kurz die wichtigsten Tests in erwähnen. Dabei wird bei R normalerweise der p-Wert (p-value) eines Tests zurückgegeben. Der p-Wert ist das kleinste Signifikanzniveau α, zu welchem die Nullhypothese bei gegebenen Daten abgelehnt werden kann. Man gibt sich also ein Niveau α (typischerweise 0.05 oder 0.01) vor und falls der p-Wert kleiner als α ist, dann wird die Nullhypothese abgelehnt.

Anfangen wollen wir mit dem t-Test, also z.B. dem Test der Hypothese $H_0 : \mu = \mu_0$ gegen $H_1 : \mu \neq \mu_0$.

6.3.1 Der t-Test

In R ist die Funktion *t.test* implementiert. Man kann mit dieser Funktion Parametertests bei normalverteilten Daten auf den Erwartungswert μ durchführen, wenn die Varianz σ^2 unbekannt ist (bei "großem" Stichprobenumfang kann der t-Test auch ohne die Normalverteilungsannahme verwendet werden). Es können aber auch Zwei-Stichproben-Tests auf gleichen Erwartungswerte,

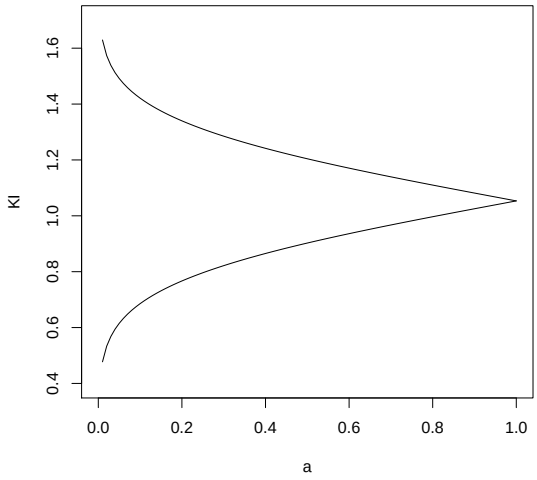

Abbildung 3: Konfidenzintervall geplottet zum Niveau $1 - \alpha$

d.h. $H_0 : \mu_1 = \mu_2$ gegen $H_1 : \mu_1 \neq \mu_2$ durchgeführt werden. Dies soll an Beispielen erklärt werden.

Beispiel Test auf Erwartungswert bei unbekannter Varianz σ^2

Wir betrachten in diesem Beispiel eine Zufallsstichprobe $(X_1, \ldots, X_n)$ mit $X_i \sim N(\mu, \sigma^2)$, wobei σ^2 unbekannt ist. Dann kann man einen Test $H_0 : \mu = \mu_0$ gegen $H_1 : \mu \neq \mu_0$ folgendermaßen durchführen.

```
>x<-rnorm(30,1,sd=2) # Erzeugen 30 normalverteilte Zufallszahlen
>mean(x) # Stichprobenmittel
[1] 1.057930
> var(x) # Stichprobenvarianz
[1] 2.276265
> t.test(x)

One Sample t-test

data: x
t = 3.8407, df = 29, p-value = 0.0006156
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
0.4945606 1.6212985
sample estimates:
mean of x
1.057930
```

Standardmäßig wird ein auf die Nullhypothese $H_0 : \mu_0 = 0$ getestet. Die Ausgabe gibt in der ersten Zeile den durchgeführten Test an. Dann wird der Wert der Test-Statistik (t=3.8407), die Anzahl der Freiheitsgrade (df = 29) und der p-Wert (p-value = 0.0006156) angegeben. In der nächsten Zeile steht die Alternativ-Hypothese (hier $H_1 : \mu_0 \neq 0$). Außerdem werden noch das 95%-Konfidenzintervall (symmetrisch) und das Stichprobenmittel angegeben. Wir haben also $H_0 : \mu_0 = 0$ gegen $H_1 : \mu_0 \neq 0$ getestet. Zum Niveau $\alpha = 0.05$ wird H_0 verworfen, da der p-Wert 0.0006156 kleiner als α ist. Testen wir nun $H_0 : \mu_0 = 1$ gegen $H_1 : \mu_0 > 1$. Dies kann wiefolgt gemacht werden.

```
> t.test(x,mu=1,alternative="greater")

One Sample t-test

data: x
t = 0.2103, df = 29, p-value = 0.4175
alternative hypothesis: true mean is greater than 1
95 percent confidence interval:
0.5898964 Inf
sample estimates:
mean of x
1.057930
```

Die Ausgabe ist ähnlich wie oben. Nur wird nun das einseitige Konfidenzintervall betrachtet. Zum Niveau $\alpha = 0.05$ wird H_0 nun nicht mehr verworfen, da der p-Wert 0.4175, also größer als α ist. Als nächstes betrachten wir Zwei-Stichproben-Tests.

Beispiel Test auf Gleichheit von Erwartungswerten

Nun wollen wir $H_0 : \mu_1 = \mu_2$ gegen $H_1 : \mu_1 \neq \mu_2$ testen. Dazu erzeugen wir eine zweite Stichprobe.

```
> y<-rnorm(20,1.7,sd=3.5) # Erzeugen 30 normalverteilte Zufallszahlen
> mean(y)
[1] 2.772165
> var(y)
[1] 17.63025
> t.test(x,y)

Welch Two Sample t-test

data: x and y
t = -1.752, df = 22.303, p-value = 0.09352
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-3.7418428 0.3133715
sample estimates:
mean of x mean of y
1.057930 2.772165
```

Gibt man zwei Datensätze an, so wird ein Zwei-Stcihproben-Test durchgeführt. Als Standard wird davon ausgegangen, dass die Varianzen unterschiedlich sind. Dann wird die Varianz in beiden Gruppen getrennt geschätzt und dann eine Modifikation der Freiheitsgrade durchgeführt. Dieser modifizierte t-Test wird Welch-Test genannt, aber auf die Details soll hier nicht eingegangen werden. Da der p-Wert 0.09352 ist, wird die Nullhypothese nicht verworfen für $\alpha = 0.05$. Wir können auch von gleichen Varianzen ausgehen und bekommen dann den normalen Zwei-Stichproben t-Test. Hier wird die gepoolte Stichprobenvarianz aus beiden Stichproben benützt.

```
> t.test(x,y,var.equal=T)

Two Sample t-test

data: x and y
t = -2.0546, df = 48, p-value = 0.04539
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-3.39182933 -0.03664197
sample estimates:
mean of x mean of y
```

1.057930 2.772165

Nun wird die Nullhypothese zum Niveau $\alpha = 0.05$ verworfen.

Weitere Optionen

Wird die Option $paired = T$ gesetzt, so wird von verbundenen Stichproben ausgegangen. Hierzu müssen beide Datensätze die gleiche Länge haben. Außerdem kann man z.B. mit $conf.level = 0.99$ das 99%-Konfidenzintervall ausgeben lassen.

6.3.2 Der F-Test

Zum Test der Gleichheit von den Varianzen σ_1^2 bzw. σ_2^2 bei zwei normalverteilten Zufallsvariablen ist der F-Test in R implementiert. Aufgerufen wird er über die Funktion *var.test*.

Beispiel Test auf Gleichheit von den Varianzen bei unbekanntem μ

Wir wollen nun $H_0 : \sigma^2 = \sigma_0^2$ gegen $\sigma^2 \neq \sigma_0^2$ testen. Zuerst nehmen wir die gleichen Zufallszahlen wie in den Beispielen für den t-Test.

> var.test(x,y)

F test to compare two variances

data: x and y
F = 0.1291, num df = 29, denom df = 19, p-value = 1.367e-06
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
0.05375289 0.28808279
sample estimates:
ratio of variances
0.1291114

Hier wird wieder die Test-Statistik angegeben (F = 0.1291), die Freiheitsgrade vom Zähler (num df = 29) bzw. vom Nenner (denom df = 19) und der p-Wert. Wie man sieht wird H_0 verworfen für $\alpha = 0.05$, also konnten wir beim t-Test für die Gleichheit der Erwartungswerte eigentlich nicht von gleichen Varianzen ausgehen. Man kann auch von der Alternativ-Hypothese $H_1 : \sigma_1^2 < \sigma_2^2$ ausgehen.

> y<-rnorm(20,3,sd=2.5) # Wir simulieren eine neue Stichprobe y
> mean(y)
[1] 2.068520
> var(y)
[1] 3.751109
> var.test(x,y,alternative="less")

F test to compare two variances

data: x and y
F = 0.6068, num df = 29, denom df = 19, p-value = 0.1097
alternative hypothesis: true ratio of variances is less than 1
95 percent confidence interval:
0.000000 1.188251
sample estimates:
ratio of variances
0.6068246

Nun wird die Nullhypothese für $\alpha = 0.05$ nicht verworfen.

6.4 Weitere Tests

Es sind in R noch eine Vielzahl weiterer Tests implementiert. So kann man zum Beispiel mit *ks.test* den Kolmogoroff-Smirnov-Test, mit *chisq.test* den Chi-Quadrat-Anpassungstest, mit *wilcox.test* den Wilcoxon-Rangsummentest oder mit *shapiro.test* den Shapiro-Wilk-Test durchführen.

6.5 Lineare Modelle

Mit dem Befehl *lm* in R lassen sich die Koeffizienten bei der linearen Regression schätzen, aber auch gleichzeitig Tests auf die Signifikanz des Modells und einzelner EinfluSSfaktoren durchführen. Dies soll an einem Beispiel veranschaulicht werden.

Wir erzeugen uns künstliche Daten eines linearen Modells. Dafür definieren wir zuerst die Designmatrix X und β.
>X<-cbind(seq(from=1,to=1,len=10),seq(from=1,to=3,len=10),seq(from=1,to=3,len=10)^2)
>beta<-c(1,2,3)

Nun erzeugen wir die Daten.
>Y<-X % * % beta + rnorm(10,sd=0.05)

Jetzt erfolgt die eingentliche Regression durch den Aufruf von *lm*.
>mod<-lm(Y ~ X[,2] + X[,3])

R führt so standardmäßig eine multiple lineare Regression durch, wobei $Y \sim X[,2]+X[,3]$ angibt, daßdie Zielvariablen in Y enthalten sind, die Einflußfaktoren in $X[,2]$ und $X[,3]$ enthalten sind und die Spalte mit Einsen wird automatisch hinzugefügt.Auf die Koeffizienten kann man dann wiefolgt zugreifen:
> mod$coefficients
(Intercept) X[, 2] X[, 3]
0.8175458 2.2022402 2.9513334

Am sinnvollsten ist es sich eine Zusammenfassung der linearen Regression auszugeben. Dies geht mit dem Befehl *summary*.

>summary(mod)

Call: lm(formula = Y ~ X[, 2] + X[, 3])

Residuals: # Das sind die Summary-Statistics der Daten

Min	1Q	Median	3Q	Max
-0.064959	-0.010477	0.001555	0.019652	0.065109

Geschätzten Werte für β

Coefficients:

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	0.81755	0.14197	5.759	0.000692 ***
X[, 2]	2.20224	0.15165	14.522	1.75e-06 ***
X[, 3]	2.95133	0.03754	78.612	1.42e-11 ***

—

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.0426 on 7 degrees of freedom
Multiple R-Squared: 1, Adjusted R-squared: 1
F-statistic: 2.233e+05 on 2 and 7 DF, p-value: < 2.2e-16

Nun interpretieren wir diese Ausgabe. Als erstes wird das benutzte Modell nochmal angegeben und dann die Summary-Statistics wie Minimum, Maximum, Median, und die Quartile der Daten angegeben. Im der Tabelle nach $coefficients$ befinden sich dann die geschätzten Werte des Parametervektors β und Eigenschaften dieses Schätzers. Der Parametervektor wurde geschätzt als $\widehat{\beta} = (0.81755, 2.20224, 2.95133)$. Als nächstes in der Tabelle folgt der Standardfehler der einzelnen Komponenten und dann der Wert der t-Statistik für den Test der Hypothese $H_0 : \beta_j = 0$. Dann wird noch der $p-Wert$ für diesen Test angegeben. Da alle p-Werte kleiner als 0.01 sind, können wir H_0 für jede Komponente von β verwerfen.
Am Ende der Ausgabe wird noch der geschätzte Wert des Bestimmtehitsmaßes R^2 angegeben, dass hier bei 1 liegt, d.h. die Daten sind gemäßeines linearen Modells angeordnet. In der letzten Zeile wird der Wert der Wert der F-Statistik für den Test von $H_0 : \beta_2 = \beta_3 = 0$ angegeben zusammen mit dem p-Wert für diesen Test. Wieder wird der Test H_0 verworfen, dass heißt es kann nicht abgelehnt werden, dass ein multiples lineares Regressionsmodell vorliegt.