

Einführung zu R*

18. August 2011

*Das Skript ist ein gemeinschaftliches Werk der Mitarbeiter des Instituts für Stochastik. Fehler oder Kommentare bitte an malte.spiess@uni-ulm.de senden.

Inhaltsverzeichnis

1. Einleitung	4
2. Datentypen	4
2.1. Vektoren	4
2.1.1. Erzeugung von Vektoren	5
2.1.2. Rechnen mit Vektoren	5
2.1.3. Funktionen auf Vektoren	6
2.1.4. Auswählen von Teilvektoren	6
2.1.5. Nicht-numerische Vektoren	7
2.2. Arrays	7
2.3. Matrizen und höherdimensionale Arrays	8
2.3.1. Erzeugen von Matrizen, Zugriff auf Matrizen und Extraktion von Teilmatrizen	8
2.3.2. Addieren und Multiplizieren von Matrizen	9
2.3.3. Funktionen auf Matrizen	10
2.4. Listen	11
2.4.1. Erzeugen von Listen, Zugriff auf Listen	11
2.5. Data Frames	12
2.5.1. Erzeugen von Data Frames, Zugriff auf Data Frames	12
3. Funktionen und Operatoren	13
3.1. Mathematische Funktionen	13
3.2. Definition eigener Funktionen	15
3.3. Manipulation und Ausgabe von Text	16
3.4. Kontrolle der im aktuellen Workspace vorhandenen Objekte	17
3.5. Sonstige interessante/wichtige Funktionen	17
4. Bedingte Ausführung von Programmcode und Schleifen	18
4.1. Bedingte Ausführung mit <code>if</code>	18
4.2. Schleifen	19
5. Verteilungsmodelle und Simulation von Zufallsvariablen	19
6. Graphiken	20
6.1. Graphische High-Level-Routinen	20
6.2. Graphische Low-Level-Routinen	21
6.3. Plotten von Funktionen	21
6.3.1. selbstgeschriebene Funktionen	22
6.4. Plotten von Histogrammen	23
6.5. Plotten von Funktionen auf \mathbb{R}^2	24
6.6. Graphiken ausdrucken	25
7. Umgang mit Daten-Dateien, Skript-Dateien und Paketen	26
7.1. Skript-Dateien einlesen	26
7.2. Zusatzpakete	26
7.3. Daten aus Dateien einlesen und in Dateien schreiben	26
8. Statistische Methoden	27
8.1. Methoden zur beschreibenden Statistik	28
8.1.1. Grundlegende Kennzahlen	28

8.1.2. Lorenz-Kurve und Gini-Koeffizient	28
8.2. Konfidenzintervalle	29
8.3. Parametertests	30
8.3.1. Der t-Test	30
8.3.2. Der F-Test	32
8.4. Weitere Tests	33
8.4.1. Pearsons Chi-Quadrat-Test	33
8.5. Verarbeiten der Testergebnisse	34
8.6. Lineare Modelle	34
8.6.1. Einfache lineare Regression	34
8.6.2. multiple lineare Regression	35
8.6.3. Verarbeiten der Ergebnisse	35
8.7. verallgemeinerte lineare Modelle	36
A. Anhang	38
A.1. Unterschied R und S-Plus	38
A.2. S-Plus allgemein	38
Index	39

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 \leftrightarrow für das Betätigen der Return-Taste steht):

Start von **R**: z.B. `thales$ R` \leftrightarrow .

Nach dem Start erscheint der sogenannte Prompt: `>`. Hier können Befehle eingegeben werden, und mit \leftrightarrow 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()` \leftrightarrow möglich

Auf Wunsch speichert **R** die Daten und Kommandozeilenhistorie in „.RData“ bzw. „.Rhistory“.

Mehrere Befehle in einer Eingabezeile 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 Funktionen (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 \mathbf{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 und Extraktion von Teilmatrizen

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. Genau so kann auch eine Teilmatrix ausgewählt werden, wenn die Argumente in den eckigen Klammern Vektoren sind. So liefert `m[c(1,2),c(1,2)]` die 2×2 -Matrix mit den ersten 2 Zeilen und Spalten.

Weitere Möglichkeiten zur Erzeugung von Matrizen

- Eine weitere Möglichkeit, Matrizen zu erzeugen, bieten die Funktionen `cbind()` und `rbind()`, mit denen man Vektoren zeilen-/ spaltenweise zu Matrizen zusammenbauen kann.
- Diagonalmatrizen lassen sich mit `diag()` erzeugen:

```

> diag(3) # liefert die 3-dimensionale Einheitsmatrix
  [,1] [,2] [,3]
[1,]  1   0   0
[2,]  0   1   0
[3,]  0   0   1
> diag(c(1, 2)) # liefert eine Diagonalmatrix mit Einträgen 1 und 2
  [,1] [,2]
[1,]  1   0
[2,]  0   2

```

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

2.3.2. Addieren und Multiplizieren von Matrizen

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%*%m
      [,1] [,2]
[1,]    1    3
[2,]    2    4

```

Wie man sieht, führt ein normales `*` eine komponentenweise Multiplikation 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. Man kann die Funktionen aber auch auf ganze Matrizen anwenden.

- **Transponieren:**

Eine Matrix wird mit dem Befehl `t()` transponiert.

- **Extrahieren der Diagonalelemente:**

Mit dem Befehl `diag()` lassen sich auch die Diagonalelemente einer Matrix extrahieren, was u.a. zur Berechnung der Spur nützlich ist.

```

> A
      [,1] [,2]
[1,]    2    4
[2,]    1    5
> diag(A)
[1] 2 5

```

- **Inversion:**

Die Inverse einer Matrix m lässt sich durch `solve(m)` bestimmen, was aber aus numerischen Gründen vermieden werden sollte.

Viel besser funktioniert es, wenn man `solve()` noch einen zweiten Parameter übergibt, der eine Matrix oder ein Vektor sein kann. Ist z.B. b ein Vektor, so liefert `solve(m, b)` den Wert $m^{-1} * b$.

Wie man unten sieht ist aufgrund der numerisch stabileren Implementierung in **R** das Ergebnis wesentlich besser.

```

> m
      [,1] [,2]
[1,]   55  130
[2,]  130  330
> # Aufruf mit nur einem Argument, danach Multiplikation:
> solve(m) %*% m
      [,1] [,2]
[1,] 1.000000e+00 7.105427e-15

```

```
[2,] 8.881784e-16 1.000000e+00
> # Aufruf direkt mit zwei Argumenten:
> solve(m, m)
      [,1] [,2]
[1,] 1.000000e+00 0
[2,] -9.473903e-17 1
```

Auch im zweiten Fall ist das Ergebnis auch nicht ganz korrekt, aber der Fehler ist bei den Einträgen auf der Nebendiagonalen immerhin um den Faktor 10 kleiner bzw. ganz verschwunden.

- **Rechnung mit Eigenwerten / -vektoren:**

Mit der Funktion `eigen()` lassen sich die Eigenwerte und Eigenvektoren numerisch berechnen. Das Ergebnis ist eine Liste:

```
> A
      [,1] [,2]
[1,] 2 4
[2,] 1 5
> eigen(A)$values
[1] 6 1
> eigen(A)$vectors
      [,1] [,2]
[1,] -0.7071068 -0.9701425
[2,] -0.7071068 0.2425356
```

Hierbei steht der i -te Eigenvektor in der i -ten Spalte der Matrix `eigen(A)$vectors`, und der dazugehörige i -te Eigenwert ist der i -te Eintrag des Vektors `eigen(A)$values`.

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 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:	<code>exp()</code>
Logarithmus(Basis: e):	<code>log()</code>
Logarithmus(Basis: base):	<code>logb(x, base=2)</code>
Wurzel:	<code>sqrt()</code>
Binomialkoeffizient $\binom{n}{m}$:	<code>choose(n, m)</code>
Sinus:	<code>sin()</code>
Cosinus:	<code>cos()</code>
Tangens:	<code>tan()</code>

Mathematische Funktionen auf Vektoren:

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

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 boolescher 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 das 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 werden, jedoch nicht für
  n,
  # für das hier ein Default-Wert von 2 definiert wurde
  # (da n=2 im Funktionskopf)
+ 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, 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

Unter Windows lassen sich alle diese Funktionen auch durch Mausklicks auf die entsprechenden Befehle in der Menüleiste 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 mehrerer 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:

```

dverteilung() (Zähl-)Dichte
pverteilung() Verteilungsfunktion
qverteilung() Quantil
rverteilung() 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 realisierten
  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:

<code>type="p"</code>	Daten als Punkte
<code>type="l"</code>	Daten als Linien
<code>type="b"</code>	Daten als Punkte und Linien
<code>xlim=c(1,100)</code>	Grenzen der x-Achse (z.B.: 1 und 100)
<code>ylim=c(0,1)</code>	Grenzen der y-Achse (z.B.: 0 und 1)
<code>xlab="x-Achse"</code>	Beschriftung der x-Achse
<code>ylab="y-Achse"</code>	Beschriftung der y-Achse
<code>log="xy"</code>	x- und y-Achse logarithmisch; auch <code>log="x"</code> , <code>log="y"</code>
<code>main="Testplot"</code>	Überschrift zum Bild
<code>lty</code>	Linientyp (gestrichelt, ...)
<code>lwd</code>	Linienstärke
<code>sub="fett: Funktion A, gestrichelt: Funktion B"</code>	Bildunterschrift
<code>add=TRUE</code>	Fügt Grafik in bereits existierende Grafik hinzu, wenn nicht angegeben (d.h. <code>add=FALSE</code>) wird alte Grafik gelöscht. Achtung: Diese Option funktioniert nicht bei allen Graphikbefehlen!

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.:

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

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

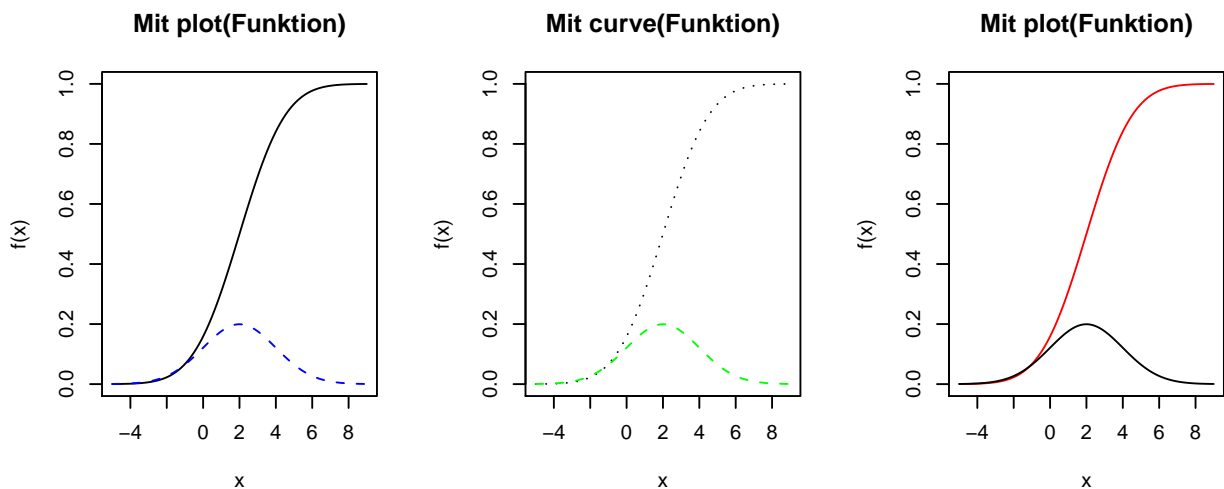


Abbildung 1: Die Verteilungsfunktion und die Dichte der Normalverteilung.

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.

6.3.1. selbstgeschriebene Funktionen

Bei selbstgeschriebenen Funktionen kann man auch einen Plot mit

```
> plot(function(x) myfunc(x))
```

erreichen. Allerdings muss hierbei die Funktion `myfunc()` Vektoren als Eingabe akzeptieren.

Hat man eine Funktion geschrieben, bei der das eigentlich nicht vorgesehen ist, so kann man das leicht „nachrüsten“, wie in folgendem Beispiel demonstriert wird:

Nehmen wir an, wir haben eine Funktion geschrieben, die für eine positive Zahl die größte Ganze Zahl findet, die kleiner ist, also die Gauß'sche Klammerfunktion:

```
gauss <- function(x) {  
  k <- 0  
  while(k <= x) {  
    k <- k + 1  
  }  
  return(k - 1)  
}
```

Nun wollen wir einen Plot der Funktion ausgeben lassen. Das funktioniert aber nicht, stattdessen erhält man die dubiose Fehlermeldung:

```
> plot(function(x) gauss(x), 0, 5)  
Fehler in xy.coords(x, y, xlabel, ylabel, log) :  
'x' and 'y' lengths differ
```

Das liegt daran, dass die Funktion auch bei Eingabe eines n -dimensionalen Vektors nur eine Zahl zurückgibt, deshalb ist die Länge der x - und y -Werte verschieden.

Die Lösung ist nun, dass man in der Funktion nur vektorwertige Funktionen verwendet. Ist dies nicht möglich, bleibt immer noch der Ausweg, eine Schleife einzusetzen:

```
gauss <- function(x) {  
  # k muss nun ein Vektor sein  
  k <- rep(0, length(x))  
  
  # die Schleife  
  for(i in 1:length(x)) {  
    while(k[i] <= x[i]) { # beachte: überall [i] ergänzen!  
      k[i] <- k[i] + 1  
    }  
  }  
  return(k - 1)  
}
```

So kann man die Funktion einfach umbauen, und der Plot funktioniert.

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...` 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, main="Histogramm mit truehist")
```

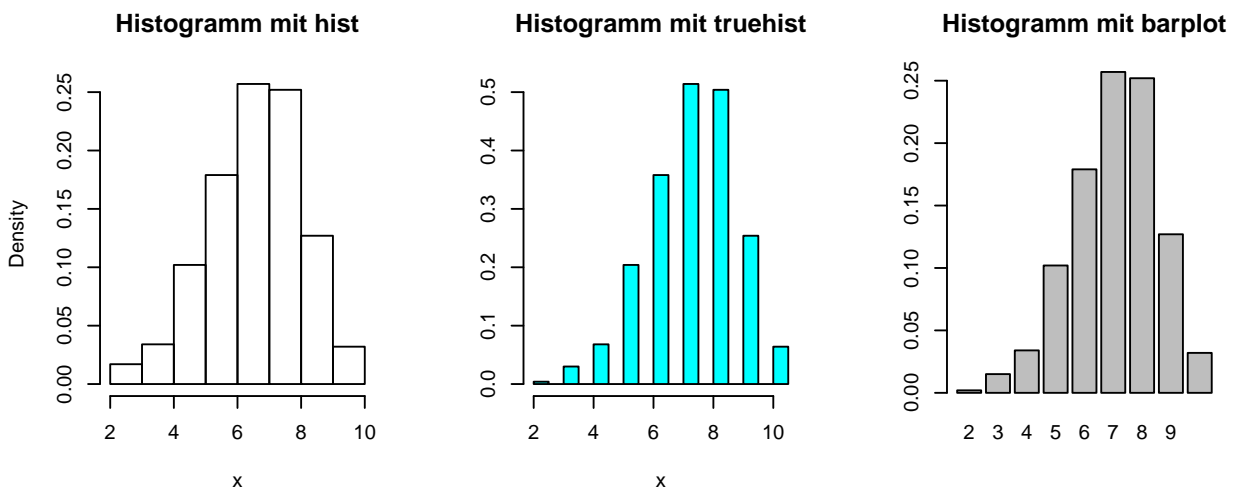


Abbildung 2: 3 verschiedene Histogramme zu den selben Daten

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. Plotten von Funktionen auf \mathbb{R}^2

Zum Plotten von Funktionen auf \mathbb{R}^2 , also $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ gibt es in **R** 3 Funktionen. Diese erfordern jedoch jeweils, dass vorher die darzustellenden Daten generiert werden.

So müssen die x - und y -Werte jeweils in einen Vektor gepackt werden und die dazugehörigen $f(x, y)$ -Werte in eine Matrix.

Hier ist ein Beispiel für die Funktion $f(x, y) = x^2 \cdot y$, die auf dem Intervall $[0, 1] \times [0, 1]$ dargestellt werden soll:

```
> x <- seq(0, 1, .01)
> y <- seq(0, 1, .01)
> f <- function(x, y) {
+   return(x^2 * y)
+ }
> z <- outer(x, y, f)
```

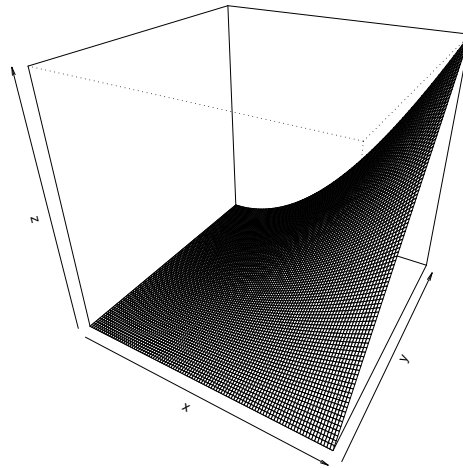
Hier kann man `outer()` geschickt einsetzen zur Erzeugung der z -Werte, was jedoch nicht immer möglich ist. `outer()` wendet die Funktion `f()` auf die jeweiligen Daten an, allerdings muss dafür `f()` vektorwertige Eingaben verarbeiten können, sowohl als 1. als auch als 2. Argument, dafür sind ggf. Anpassungen analog zu [Abschnitt 6.3.1](#) nötig.

Hier noch einige Befehle zum eigentlichen Plotten der Graphiken:

- `persp()`: Mit diesem Befehl kann man die Daten als 3D-Schaubild plotten lassen, vgl. [Bild 3\(a\)](#):

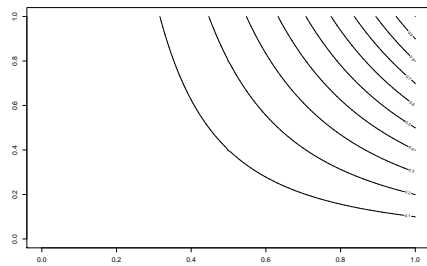
```
> persp(x, y, z)
```


Bsp. für persp()



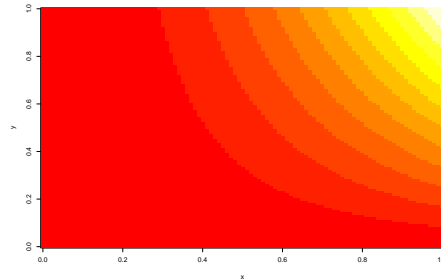
(a) Beispiel für persp()

Aufruf: `persp(x, y, z, phi= 30, theta = 30, main= 'Bsp. für persp()')`



(b) Beispiel für contour()

Aufruf: `countour(x, y, z)`



(c) Beispiel für image()

Aufruf: `image(x, y, z)`

Abbildung 3: 3D-Plots

Dabei bietet der Befehl noch interessante Möglichkeiten, den Blickwinkel zu verändern mit den Parametern `theta`, `phi` und `r`, neben vielen anderen praktischen Optionen.

- `contour()`: Plottet Höhenlinien der Funktion, vgl. Bild 3(b):

```
> contour(x, y, z)
```

Mit `nlevels` bzw. `levels` lassen sich noch die Anzahl der Höhenlinien bzw. die einzelnen Levels genauer festlegen.

- `image()`: Zeichnet eine „Karte“ der Funktion, wobei in der Standardeinstellung „tiefe“ Stellen rot und „hohe“ Stellen weiß sind, vgl. Bild 3(c):

```
> image(x, y, z)
```

6.6. 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 Graphik-

bereich 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.

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 V_1, \dots, V_n (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
```

8.2. Konfidenzintervalle

In **R** werden viele Konfidenzintervalle von Test-Funktionen berechnet, beispielsweise bei `t.test()` oder `var.test()`. Diese werden im nächsten Abschnitt 8.3 behandelt.

Darüber hinaus gibt es die Möglichkeit, selbst Funktionen zu schreiben, die Konfidenzintervalle liefern. Durch die Quantilfunktionen (siehe 5) kann man solche Funktionen leicht realisieren.

Als einfaches Beispiel betrachten wir das Konfidenzintervall zum Niveau $1 - \alpha$ für den Erwartungswert μ einer normalverteilten Stichprobe (X_1, \dots, X_n) mit bekannter bzw. unbekannter Varianz σ^2 .

Beispiel Varianz σ^2 bekannt

```
> x <- rnorm(100,1,1) # Simulieren normalverteilter Zufallsvariablen ,  
  Varianz 1  
> confintLow <- function(gamma, x) { # untere Grenze des  
  Konfidenzintervalls  
+ return(mean(x)+qnorm((1-gamma)/2)*1/sqrt(length(x)))  
+ }  
> confintHigh <- function(gamma, x) { # obere Grenze des  
  Konfidenzintervalls  
+ return(mean(x)+qnorm(1-(1-gamma)/2)*1/sqrt(length(x)))  
+ }  
> c(confintLow(0.95,x), confintHigh(0.95,x)) # Das konkrete 95%  
  Konfidenzintervall  
[1] 0.6413903 1.0333831  
> # Plotten des Intervalls zu verschiedenen Niveaus  
> plot(function(gamma) confintLow(gamma,x),0,1,ylim=c(0.5,1.2),xlab="gamma",  
  ylab="KI")  
> plot(function(gamma) confintHigh(gamma,x),0,1,add=TRUE)
```

Der Plot des Konfidenzintervalls ist in Abbildung 4 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(gamma,x) { # Die untere Intervallgrenze  
+ return(mean(x)+qt((1-gamma)/2,length(x)-1)*sd(x)/sqrt(length(x)))  
+ }  
> tconfintup <- function(gamma,x) { # Die obere Intervallgrenze  
+ return(mean(x)+qt(1-(1-gamma)/2,length(x)-1)*sd(x)/sqrt(length(x)))  
+ }  
> c(tconfintlow(0.95,x),tconfintup(0.95,x)) # Das konkrete 95%  
  Konfidenzintervall
```

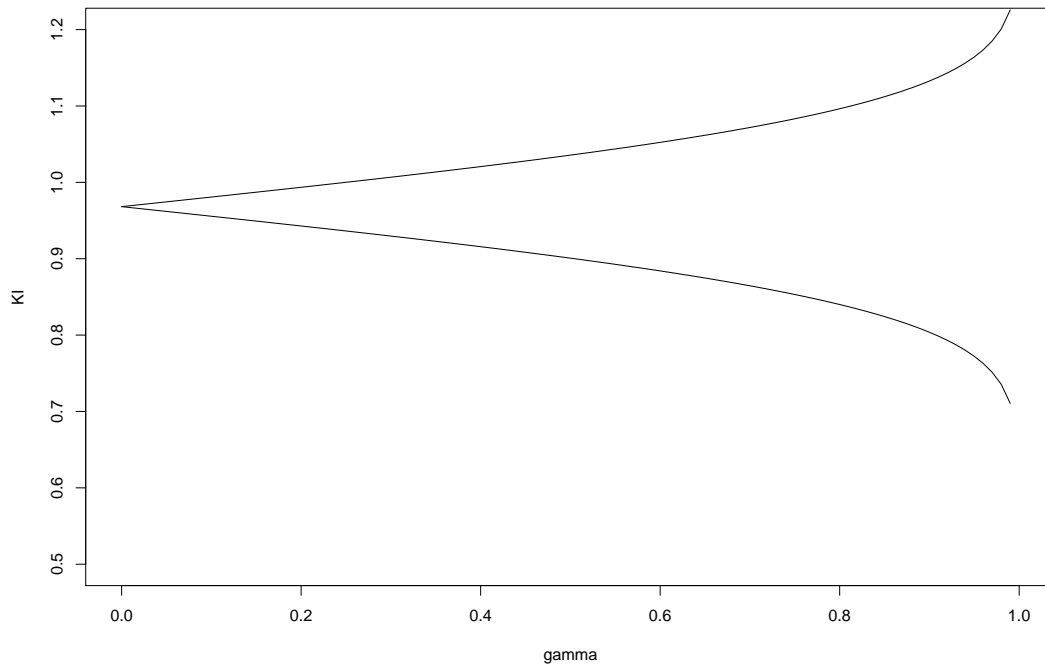


Abbildung 4: Konfidenzintervall geplottet zum Niveau $1 - \alpha$

```
[1] 0.1249187 1.2428206
```

8.3. 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. Ist ein Niveau α (typischerweise zwischen 0.01 und 0.1) vorgegeben, und falls der p-Wert kleiner als α ist, dann wird die Nullhypothese abgelehnt.

Anfangen wollen wir mit dem t-Test, mit dem man z.B. die Hypothese $H_0 : \mu = \mu_0$ gegen $H_1 : \mu \neq \mu_0$ testen kann.

8.3.1. 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 Gleichheit der Erwartungswerte, 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, \dots, 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] 0.7338278
> var(x) # Stichprobenvarianz
[1] 3.980814
> t.test(x)

One Sample t-test

data:  x
t = 2.0145, df = 29, p-value = 0.05331
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 -0.01119133  1.47884684
sample estimates:
mean of x
0.7338278
```

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 ($t=2.0145$), die Anzahl der Freiheitsgrade ($df = 29$) und der p-Wert ($p\text{-value} = 0.05331$) 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 $\alpha = 0.1$ wird H_0 verworfen, da der p-Wert 0.05331 kleiner als α ist. Testen wir nun $H_0 : \mu_0 = 1$ gegen $H_1 : \mu_0 > 1$. Dies kann wie folgt gemacht werden.

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

One Sample t-test

data:  x
t = -0.7307, df = 29, p-value = 0.7646
alternative hypothesis: true mean is greater than 1
95 percent confidence interval:
 0.1148840      Inf
sample estimates:
mean of x
0.7338278
```

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 0.7646, 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] 0.004362171
> var(y)
[1] 9.471041
> t.test(x,y)
```

```
Welch Two Sample t-test
```

```

data:  x and y
t = 0.9369, df = 29.616, p-value = 0.3564
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.8615488  2.3204799
sample estimates:
  mean of x  mean of y
0.733827756  0.004362171

```

Gibt man zwei Datensätze an, so wird ein Zwei-Stichproben-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 0.3564 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=TRUE)

Two Sample t-test

data:  x and y
t = 1.0186, df = 48, p-value = 0.3135
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.7104013  2.1693324
sample estimates:
  mean of x  mean of y
0.733827756  0.004362171

```

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

Weitere Optionen

Wird die Option `paired=TRUE` 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.

8.3.2. Der F-Test

Zum Test der Gleichheit von den Varianzen σ_1^2 bzw. σ_2^2 bei zwei normalverteilten Zufallsstichproben 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 = 0.4203, num df = 29, denom df = 19, p-value = 0.03448
alternative hypothesis: true ratio of variances is not equal to 1

```



```
95 percent confidence interval:
```

```
0.1749893 0.9378363
```

```
sample estimates:
```

```
ratio of variances
```

```
0.4203143
```

Hier wird wieder die Test-Statistik angegeben ($F = 0.4203$), 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] 2.335488
```

```
> var(y)
```

```
[1] 6.589322
```

```
> var.test(x,y,alternative="less")
```

```
      F test to compare two variances
```

```
data:  x and y
```

```
F = 0.6041, num df = 29, denom df = 19, p-value = 0.1077
```

```
alternative hypothesis: true ratio of variances is less than 1
```

```
95 percent confidence interval:
```

```
0.000000 1.182977
```

```
sample estimates:
```

```
ratio of variances
```

```
0.6041311
```

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

8.4. 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.

Die Durchführung und Ausgabe sind sehr ähnlich wie im Fall der oben genannten Parametertests, deshalb werden hier nur einige davon vorgestellt.

8.4.1. Pearsons Chi-Quadrat-Test

Der Chi-Quadrat-Test von Pearson kann in **R** mit der Funktion `chisq.test()` ausgeführt werden.

Sind z. B. im Vektor **x** multinomialverteilte Klassenstärken gespeichert, und es soll getestet werden, ob die Multinomialverteilung den Parametervektor **p** hat, so lautet der Aufruf `chisq.test(x, p=p)`.

Beispiel

```
> x <- c(94, 89, 34)
```

```
> p <- c(3/7, 3/7, 1/7)
```

```
> chisq.test(x, p=p)
```

```
Chi-squared test for given probabilities
```

```
data: x
X-squared = 0.4731, df = 2, p-value = 0.7893
```

Um nicht normierte Wahrscheinlichkeitsvektoren `p` verwenden zu können, ist die Option `rescale.p = TRUE` hilfreich.

8.5. Verarbeiten der Testergebnisse

Bei allen in **R** implementierten Tests kann man auf die Testergebnisse zugreifen. Beispielsweise kann man beim t-Test mit den Operatoren `t.test(x)$confint` bzw. `t.test(x)$p.value` das Konfidenzintervall bzw. den p-Wert auch einzeln als Rückgabewert bekommen und später weiterverarbeiten. Ähnliches ist auch bei anderen Tests möglich.

Mit dem Aufruf `names(<testname>)` erhält man die Felder, die der Test `<testname>` zur Verfügung stellt. Auf diese kann man dann mit dem `$` zugreifen.

8.6. 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.

8.6.1. Einfache lineare Regression

In diesem Fall gibt es nur einen Einflussfaktor. Das Modell ist $Y_i = \alpha + \beta x_i + \varepsilon_i$. Wenn die Werte der Zielvariablen im Vektor `y` gespeichert sind, und die Werte des Einflussfaktors in `x`, so ist der Aufruf wie folgt:

```
> x <- c(1,2,3,4)
> y <- c(5,2,6,1)
> lm(y ~ x + 1)
```

Call:

```
lm(formula = y ~ x + 1)
```

Coefficients:

```
(Intercept)          x
          5.5          -0.8
```

Das Modell wird also durch `y ~ x + 1` beschrieben. Soll das Modell ohne Konstante α sein, wäre die Beschreibung entsprechend `y ~ x - 1`:

```
> lm(y ~ x - 1)
```

Call:

```
lm(formula = y ~ x - 1)
```

Coefficients:

```
          x
     1.033
```

Wie man sieht fehlt hier der **(Intercept)**-Koeffizient, also α .

8.6.2. multiple lineare Regression

Hier ist das Modell $Y_i = \alpha + \beta_1 x_{1i} + \dots + \beta_m x_{mi} + \varepsilon_i$. Oft wird α hier weggelassen. In **R** lässt sich das Modell wie folgt beschreiben: $\mathbf{y} \sim \mathbf{x}_1 + \dots + \mathbf{x}_m + 1$, und ohne α wieder entsprechend mit -1 .

Darüberhinaus gibt es noch die Möglichkeit, die Einflussfaktoren in einer Designmatrix zu speichern. Die enthält dann in der j -ten Spalte die n Werte von x_{j1} bis x_{jn} . Zum Erzeugen einer Matrix siehe auch Abschnitt 2.3.1.

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)
> # Num erzeugen wir die Daten.
> Y <- X %*% beta + rnorm(10, sd=0.05)
> # Jetzt erfolgt die eigentliche Regression durch den Aufruf von lm.
> lm(Y ~ X - 1)
```

Call:

```
lm(formula = Y ~ X - 1)
```

Coefficients:

X1	X2	X3
1.074	1.872	3.039

R führt so standardmäßig eine multiple lineare Regression durch.

8.6.3. Verarbeiten der Ergebnisse

Zusätzlich bietet **R** noch die Möglichkeit, das Ergebnis zu speichern und Daten, wie z.B. die Koeffizienten, auszulesen.

Dafür kann man einfach durch den Aufruf

```
> mod <- lm(Y ~ X - 1)
```

das Ergebnis speichern.

Dann kann man mit `mod` einige nützliche Daten ausgeben oder speichern, z.B. die Koeffizienten:

```
> mod$coefficients
      X1      X2      X3
1.074323 1.871672 3.039064
```

Sinnvoll ist es sich eine Zusammenfassung der linearen Regression ausgeben zu lassen. Dies geht mit dem Befehl `summary()`.

```
> summary(mod)
```

Call:

```
lm(formula = Y ~ X - 1)
```

Residuals:

Min	1Q	Median	3Q	Max
-0.071453	-0.027060	-0.002819	0.041261	0.068402

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)	
X1	1.07432	0.17724	6.061	0.00051	***
X2	1.87167	0.18932	9.886	2.31e-05	***
X3	3.03906	0.04687	64.840	5.45e-11	***

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

Residual standard error: 0.05318 on 7 degrees of freedom

Multiple R-squared: 1, Adjusted R-squared: 1

F-statistic: 4.867e+05 on 3 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. In der Tabelle nach `coefficients` befinden sich dann die geschätzten Werte des Parametervektors β und Eigenschaften dieses Schätzers. Der Parametervektor wurde geschätzt als $\hat{\beta} = (1.07432, 1.87167, 3.03906)$. 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 Bestimmtheitsmaßes R^2 angegeben, dass hier bei 1 liegt, d.h. die Daten sind gemäß eines linearen Modells angeordnet. In der letzten Zeile wird 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.

8.7. verallgemeinerte lineare Modelle

Verallgemeinerte lineare Modelle können in **R** ganz ähnlich wie die linearen Modelle analysiert werden. Der Hauptunterschied besteht darin, dass bei den verallgemeinerten linearen Modellen zusätzlich die zugrundeliegende Verteilungsfamilie angegeben werden muss. Dabei gibt es unter anderem folgende Modell mit folgenden Bezeichnungen:

Modell	Bezeichnung in R
Bernoulli-Verteilung, logit-Link-Funktion	<code>binomial(link = 'logit')</code>
Bernoulli-Verteilung, probit-Link-Funktion	<code>binomial(link = 'probit')</code>
Normalverteilung, natürliche Link-Funktion	<code>gaussian(link = 'identity')</code>
Poissonverteilung, natürliche Link-Funktion	<code>poisson(link = 'log')</code>

Für Details und weitere Familien siehe Hilfeseite von `family`.

Ein Aufruf kann z.B. wie folgt aussehen, hier mit Poisson-verteiltern Zielgrößen:

```
> summary(glm(detg$Fr ~ detg$M.user + detg$Brand, family = poisson))
```

Call:

```
glm(formula = detg$Fr ~ detg$M.user + detg$Brand, family = poisson)
```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-3.7714	-2.2350	0.1332	1.5088	3.3176

Coefficients:

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	3.78830	0.05364	70.621	<2e-16 ***
detg\$M.userY	-0.08736	0.06305	-1.385	0.166
detg\$BrandM	-0.01587	0.06300	-0.252	0.801

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

(Dispersion parameter for poisson family taken to be 1)

Null deviance: 118.63 on 23 degrees of freedom

Residual deviance: 116.64 on 21 degrees of freedom

AIC: 255.06

Number of Fisher Scoring iterations: 4

Die Ausgabe ist analog zum linearen Modell zu interpretieren.

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 konzentrieren uns in diesem Skript auf **R** und übernehmen keine Garantie, dass die Programme auch auf S-Plus laufen.

Der wohl entscheidendste Unterschied ist, dass **R** eine open-source Software 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 separaten Projekten, z.B. rkwad 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 Eingabezeile 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 Variablenamen: 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

	18
==	18
\$	28
%*%	14
&&	18

A

abline()	21
apply()	15
array()	8

B

barplot()	20
Binomialkoeffizient	13
boxplot()	20

C

c()	5
call by value	13
cat()	17
cbind()	10
character	7
chisq.test()	34
choose()	13
conf.level	33
cos()	13
curve()	20

D

d<Verteilung>()	19
data.frame()	12
dev.copy()	27
dev.cur()	27
dev.off()	20, 27
diag()	9f.
diff()	13
dim()	8

E

eigen()	11
else	18
exp()	13

F

FALSE	18
for()	19
function()	16

G

getwd()	18
glm()	37

H

help()	4
hist()	20

I

if	18
Indikatorfunktion	14
install.packages()	27
integrate()	13

K

ks.test()	34
-----------	----

L

Lc()	30
legend()	21
length()	6
library()	27
lines()	21
list()	11
lm()	35
load()	17
locator()	21
log()	13
logb()	13
logical	7
ls()	17

M

matrix()	8
max()	13
mean()	29
mfcoll()	20
mfrow()	20
min()	13

motif() 20

N

names() 17

numeric() 18

P

p<Verteilung>() 19

pairs() 20

par() 20

paste() 17

persp() 25

plot() 20

points() 21

postscript() 27

print() 17

prod() 13

Q

q<Verteilung>() 19

qqplot() 20

quantile() 29

R

r<Verteilung>() 19

range() 13

rbind() 10

read.table() 12

rep() 5

return() 16

rm() 17

S

save() 17

save.image() 17

scan() 5

segments() 21

seq() 5

setwd() 18

shapiro.test() 34

sin() 13

sink() 16

solve() 10

sort() 13

source() 16, 27

sqrt() 13

subset() 7

sum() 13

summary() 17, 29, 36

symbols() 21

T

t() 6, 10

t.test() 31

table() 24

tan() 13

text() 21

title() 21

TRUE 18

truehist() 20

V

var() 29

var.test() 33

W

while() 19

wilcox.test() 34

win.graph 20

write.table() 28