Einführung in die Statistik-Programmier-Sprache R

Müller, SS 2005

1 Grundlagen von R

Die Statistik-Software R ist eine objekt-orientierte, interaktive Programmiersprache, mit der einfach statistische Auswertungen vorgenommen, vielfältige Grafiken erstellt und Simulationen durchgeführt werden können. Sie kann umsonst vom Internet heruntergeladen werden. Es gibt viele Ergänzungspakete. Hier wird vor allem das Basis-Paket beschrieben. Änderungen für die neueste Version `R2.0.2` wurden nach Möglichkeit berücksichtigt. Es kann aber sein, dass einige Änderungen nicht übernommen wurden.

1.1 Herunterladen der freien Software R

Im folgenden wird das Herunterladen des Basis-Paketes für einen Windows-Rechner (z.B. Windows NT oder Windows 98, 2000, XP) beschrieben. Die Vorgehensweise ist aber für andere Rechner sehr ähnlich. Insbesondere gibt es unter der gleichen Internet-Adresse auch R-Versionen für andere Betriebssysteme.

Um R auf einen Windows-Rechner herunterzuladen, klickt man bei der Internet-Adresse **http://cran.r-project.org/** auf `Windows (95 and later)` und anschließend auf `base`. Dann speichert man alles aus `rw2001.exe` in einen Ordner des Computers, der die Programme von R enthalten soll. Um R zu installieren klickt man auf `rw2001.exe`. Dann wird R mit Unterordnern in dem ausgewählten Ordner installiert. Danach kann R gestartet werden. Im Ordner `\rw2001\doc\manual` enthält die Datei `R-intro.pdf` eine ausführliche Einführung in R auf Englisch.

1.2 Starten von R im Rechnerraum der Mathematik

Man geht nacheinander auf `Start`, `Programme`, `R` und startet mit Doppelklick `R2.0.1`.

1.3 Starten von R allgemein

Man geht dazu innerhalb des benutzten Ordners zu `\rw2001\bin\Rgui.exe` und klickt auf `Rgui.exe`. Es erscheint dann ein Fenster mit ungefähr folgendem Inhalt:

```
R : Copyright 2000, The R Development Core Team
Version 1.0.0  (February 29, 2000)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type    "?license" or "?licence" for distribution details.

R is a collaborative project with many contributors.
```

```
Type    "?contributors" for a list.

Type    "demo()" for some demos, "help()" for on-line help, or
        "help.start()" for a HTML browser interface to help.
Type    "q()" to quit R.

>
```

Nach dem Zeichen "`>`" können die R-Befehle eingegeben werden. Allerdings besteht bei dieser Vorgehensweise noch ein Nachteil. Alle gelesenen Dateien, wie zum Beispiel Datensätze, müssen in dem Unterordner `\rw2001\bin` liegen, und alles, was gespeichert wird, wird auch in diesem Ordner abgelegt. Der Ordner `\rw2001\bin` ist nämlich das voreingestellte Arbeitsverzeichnis von R. Im Rechnerraum ist aber darauf kein Zugriff möglich. Auf einem privaten Rechner ist das auch ein Nachteil, wenn verschiedene Projekte mit R bearbeitet werden. Dann sollten am besten die Datensätze und Ergebnisse der verschiedenen Projekte in verschiedenen Ordnern abgelegt werden. Das kann dadurch erreicht werden, dass Ikonen erzeugt werden, die eine Verknüpfung mit den gewünschten Arbeitsverzeichnissen beinhalten.

1.4 Erstellung einer Ikone mit einer Verknüpfung zu einem speziellen Arbeitsverzeichnis

Um eine Ikone mit einer speziellen Verknüpfung zu erstellen, klickt man mit der rechten Maustaste in den Hintergrund und dann mit der linken Maustaste auf folgende Befehle: `neu`, `Verknüpfung` und `Durchsuchen`. Dann sucht man den Ordner, in dem Rgui.exe steht. Wenn Rgui.exe markiert ist, klickt man auf `Öffnen`, dann auf `Weiter`. Danach kann ein neuer Name für die Verknüpfung angegeben werden, z.B. `R_Projekt1`. Ein Klick auf `Fertigstellen` ergibt, dass die Ikone im Hintergrund, die vorher den Namen `Neue Verknüpfung` hatte, hat jetzt den neuen Namen besitzt. Das Arbeitsverzeichnis dieser Ikone ist aber weiterhin das voreingestellte Arbeitsverzeichnis `\rw2001\bin`. Also sollte man für diese Ikone das gewünschte Arbeitsverzeichnis zuordnen. Dieses Arbeitsverzeichnis kann aber auch jederzeit nachträglich geändert werden. Dazu muss man nur mit der rechten Maustaste auf die Ikone klicken und dann mit der linken Maustaste nacheinander die Befehle `Eigenschaften` und `Verknüpfung` anklicken. In dem erscheinenden Fenster kann dann das gewünschte Arbeitsverzeichnis eingegeben werden. Nach dem Erstellen der Ikone muss zum Starten von R nur noch mit einem Doppelklick auf die Ikone geklickt werden.

1.5 Kommando-Fenster

Nach dem Starten von R erscheint das Kommando-Fenster. In diesem werden die Kommandos/Befehle geschrieben. Jeder Befehl beginnt mit "`>`", das von R gegeben wird. Der Befehl wird ausgeführt, wenn die Taste Enter gedrückt wird. Die allgemeine Form ist:

> *Befehl* Enter,

zum Beispiel: `> 3 * (4 + 2)` Enter.
R führt mit dem Drücken von Enter den Befehl aus. Bei dem Beispiel gibt R nach der Ausführung in der nächsten Zeile folgende Antwort: "`[1] 18`". Danach erscheint in einer weite-

ren neuen Zeile wieder das Zeichen "`>`", so dass ein neuer Befehl eingegeben werden kann. Im Kommando-Fenster steht also dann (das Drücken von Enter wird nicht angezeigt):

```
> 3*(4+2)
[1] 18
>
```

Ist der Befehl unvollständig, so antwortet R mit "`+`" und erwartet, dass hinter dem "`+`" der Befehl vervollständigt wird. Z.B., wenn "`> 3 * (4 + 2`" mit Enter abgeschickt wird, sieht man im Kommando-Fenster folgendes:

```
> 3*(4+2
+
```

Schreibt man hinter das Zeichen "`+`" das Zeichen "`)`" und schickt es mit Enter ab, so zeigt das Kommando-Fenster folgendes:

```
> 3*(4+2
+ )
[1] 18
>
```

Ist der Befehl unzulässig oder fehlerhaft, antwortet R mit einer Fehlermeldung. Diese kann z.B. so aussehen:

```
> 3+*(4+2)
Error: syntax error
>
```

Mit der Taste ↑ können vorangegangene Befehle wiederholt werden, ohne dass sie wieder eingetippt werden müssen. Dabei können fehlerhafte Befehle korrigiert werden, bevor sie mit Enter wieder abgeschickt werden.

Drückt man gleichzeitig die Tastenkombination Ctrl-c oder Ctrl-Break, dann wird die Ausführung eines R-Befehls abgebrochen. Das ist manchmal nötig, wenn die Ausführung länger dauert und man während der Ausführung einen Fehler feststellt.

1.6 Speichern von Rechenergebnissen

Rechenergebnisse, wie z.B. das Ergebnis von `3*(4+2)` können in Variablen gespeichert werden. Die Namen der Variablen können aus beliebig langen Zeichenketten bestehen. Die Zuweisung des Ergebnisses zu einer Variablen geschieht mit dem Zuweisungsoperator "`<-`" und sieht im allgemeinen folgenderweise aus: `>` *Variablenname* `<-` *Befehl*. Wird nacheinander mehrmals der gleichen Variablen verschiedene Ergebnisse zugewiesen, enthält die Variable das Ergebnis der letzten Zuweisung. Um nachzusehen, was eine Variable enthält, muss man nur den Variablennamen eingeben und mit Enter abschicken. Das kann z.B. so aussehen:

```
> Variable.1 <- 3*(4+2)
> Variable.1
[1] 18
> Variable.1 <- 3/(4+2)
```

```
> Variable.1
[1] 0.5
>
```

Eine Variable muss aber nicht nur einen Wert enthalten, sondern kann auch mehrere Werte enthalten. Der einfachste Fall ist der, dass die Variable eine Folge von Zahlenwerten in Form eines Vektors enthält. Z.B. ist folgende Zuweisung möglich:

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

Die Funktion `c` bewirkt hier, dass die Zahlen 1, 2 und 3 zu einem Spalten-Vektor zusammengefügt werden (`c` von "combine"). Generell fügt die Funktion `c` ihre Argumente zu einem Spalten-Vektor zusammen. Dabei können die Argumente auch Befehle oder selbst schon Vektoren sein. Z.B. ist folgendes möglich:

```
> Variable.1 <- c(1,2,3)
> Variable.1 <- c(Variable.1,Variable.1,3*(4+2))
> Variable.1
[1]  1  2  3  1  2  3 18
>
```

Ergebnisse können aber nicht nur sequentiell als Vektor gespeichert werden, sondern auch als Tafel/Tableau bzw. als Matrix gespeichert werden. Mit der Funktion `matrix` wird aus einem Vektor eine Matrix erzeugt. Dabei geben die Argumente `ncol` und `nrow` die Anzahl der Spalten bzw. der Zeilen an, wobei nur eins dieser Argumente spezifiziert werden muss. Per Voreinstellung wird dann aus dem Vektor spaltenweise eine Matrix erstellt. Soll die Erstellung zeilenweise erfolgen, muss noch das Argument `byrow=TRUE` bzw. kürzer `byrow=T` benutzt werden. Hier ein Beispiel:

```
> Variable.2<-c(1,2,3,4,5,6)
> Matrix.1<-matrix(Variable.2,ncol=2,byrow=T)
> Matrix.1
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
>
```

Mit der Funktion `dim` können die Dimensionen der Matrix abgefragt werden, und mit der Funktion `dimnames` können Namen für die Zeilen und Spalten der Matrix festgelegt und abgeändert werden. Dabei werden die Namen über eine Liste mit der Funktion `list` eingelesen. Diese Liste enthält zwei Komponenten. Die erste Komponente ist ein Vektor, der die Namen für die Zeilen enthält, und die zweite Komponente ist der Vektor mit den Namen für die Spalten. Wird eine dieser Komponenten als `NULL` gesetzt, werden keine Namen vergeben. Hier ein Beispiel für die Anwendung von `dim` und `dimnames`:

```
> Matrix.1<-matrix(c(1,2,3,4,5,6),ncol=2,byrow=T)
> dim(Matrix.1)
```

```
[1] 3 2
> dimnames(Matrix.1)<-list(c("Row1","Row2","Row3"),c("C1","C2"))
> Matrix.1
     C1 C2
Row1  1  2
Row2  3  4
Row3  5  6
> dimnames(Matrix.1)<-list(NULL,c("C1","C2"))
> Matrix.1
     C1 C2
[1,]  1  2
[2,]  3  4
[3,]  5  6
>
```

Mit dem Befehl > `objects()` oder > `ls()` werden alle vorhandenen Variablen aufgelistet. Hat man die obigen Variablen `Variable.1`, `Variable.2` und `Matrix.1` erzeugt, ergibt sich z.B.:

```
> ls()
 [1] "Matrix.1"             "Variable.1"           "Variable.2"
>
```

Mit dem Befehl > `rm(`*Variablenname*`)` or > `remove(`*Variablenname*`)` wird die Variable mit dem Namen *Variablenname* gelöscht. Z.B. löscht `> remove(Variable.1)` die Variable.1.

1.7 Funktionen in R

R stellt eine Vielzahl von Funktionen zur Verfügung. Beispiele davon sind die oben erwähnten Funktionen `c`, `matrix`, `ls`, `objects` und `remove`. Ausgehend von den Argumenten der Funktion werden mehrere Rechenschritte ausgeführt, deren Ergebnisse als "Funktionswert" ausgegeben werden. Der "Funktionswert" kann aus einem Ergebnis oder auch aus mehreren Ergebnissen bestehen. Der Aufruf einer Funktion sieht im allgemeinen wie folgt aus:

Funktionsname(*Argumentname1*=*Argument1*,...,*ArgumentnameN*=*ArgumentN*).

Die Argumente können erforderlich oder optional sein. Die optionalen Argumente müssen nicht angegeben werden, weil sie eine Voreinstellung besitzen. Es gibt Funktionen wie `ls` und `objects` die nur optionale Argumente oder gar keine Argumente besitzen. Diese werden mit *Funktionsname*`()` aufgerufen. Gibt es erforderliche Argumente oder sollen optionale Argumente verändert werden, müssen diese Argumente angegeben werden. Wird die Reihenfolge der Argumente eingehalten, müssen die Argumentnamen nicht angegeben werden. Dann kann die Funktion wie folgt aufgerufen werden: *Funktionsname*(*Argument1*,...,*ArgumentN*).

Was eine Funktion tut und was für Argumente sie hat, kann durch die Hilfe-Abfrage > `?`*Funktionsname* abgefragt werden. Es wird dann ein Fenster mit der Beschreibung der Funktion geöffnet. Z.B. ergibt

```
> ?matrix
```

eine Beschreibung der Funktion `matrix`.

1.8 Arithmetische Operatoren und mathematische Funktionen

Operatoren sind eigentlich Funktionen mit zwei Argumenten, wobei die beiden Argumente rechts und links vom Operator geschrieben werden. Bei arithmetischen Operatoren müssem diese beiden Argumente arithmetische Ausdrücke sein, d.h. es müssen einzelne Zahlen, Vektoren oder Matrizen sein, und das Ergebnis ist wieder eine Zahl, ein Vektor oder eine Matrix. Daneben gibt es noch Vergleichs-Operatoren und logische Operatoren. Diese werden aber erst später behandelt. Es gibt folgende arithmetische Operatoren:

Arithmetische Operatoren	
$*$	Multiplikation
$+$	Addition
$-$	Subtraktion
$/$	Division
^	Exponentiation
%%	Modulo Operator
%/%	Ganzzahlige Division
%*%	Matrix Multiplikation

Bei allen oben aufgeführten Operatoren können die Argumente immer auch Vektoren oder Matrizen sein. Stimmen dabei die Anzahlen der Komponenten des linken und rechten Argumentes überein, wird die Operation komponentenweise vorgenommen. Sind die Anzahlen der Komponenten verschieden, dann wird das Argument mit der geringeren Anzahl von Komponenten spaltenweise solange intern wiederholt, dass die Anzahlen übereinstimmen und die Operation wieder komponentenweise ausgeführt werden kann. Eine Ausnahme bildet nur die Matrix-Multiplikation. Da die Matrix-Multiplikation nur definiert ist, wenn die Anzahl der Spalten des linken Argumentes mit der Anzahl der Zeilen des rechten Argumentes übereinstimmen, muss diese Voraussetzung bei der Anwendung dieses Operators erfüllt sein. Zum Beispiel erhalten wir folgende Ergebnisse:

```
> Matrix.2<-matrix(c(1,2,1,2,3,0),ncol=2,byrow=T)
> Matrix.2
     [,1] [,2]
[1,]    1    2
[2,]    1    2
[3,]    3    0
> Matrix.2*10
     [,1] [,2]
[1,]   10   20
[2,]   10   20
[3,]   30    0
> Matrix.2*Matrix.2
     [,1] [,2]
[1,]    1    4
[2,]    1    4
[3,]    9    0
> Matrix.2*c(1,1,0,0)
     [,1] [,2]
```

```
[1,]    1    0
[2,]    1    2
[3,]    0    0
> Matrix.2*c(1,2)
     [,1] [,2]
[1,]    1    4
[2,]    2    2
[3,]    3    0
> Matrix.2%*%c(1,2)
     [,1]
[1,]    5
[2,]    5
[3,]    3
> Matrix.2%*%10
Error in Matrix.2 %*% 10 : non-conformable arguments
>
```

Neben den arithmetischen Operatoren gibt es noch etliche mathematische Funktionen, die als richtige R-Funktionen gegeben sind. Die folgende Tabelle gibt eine Auflistung der wichtigsten mathematischen Funktionen.

Mathematische Funktionen

abs	Absoluter Wert (Betrag)
ceiling	Nächster größerer ganzzahliger Wert
floor	Nächster kleinerer ganzzahliger Wert
trunc	Nächster ganzzahliger Wert in Richtung der Null
sqrt	Wurzel-Funktion
exp	Exponential-Funktion
log	Natürlicher Logarithmus (zur Basis e)
log10	Logarithmus zur Basis 10
gamma, lgamma	Gamma-Funktion und ihr natürlicher Logarithmus
cos, sin, tan	Trigonometrische Funktionen
cosh, sinh, tanh	Hyperbolische Trigonometrische Funktionen
acos, asin, atan	Inverse Trigonometrische Funktionen
acosh, asinh, atanh	Inverse Hyperbolische Trigonometrische Funktionen
max	Maximaler Wert eines Vektors
min	Minimaler Wert eines Vektors
length	Länge eines Vektors
t	Transposition einer Matrix

Die Argumente dieser Funktionen können auch wieder Vektoren oder Matrizen sein. Macht die Funktion eigentlich nur Sinn für eine einzelne Zahl - d.h. bei einem Vektor der Dimension 1 -, wird bei einem höher dimensionalen Argument wieder die Funktion komponentenweise angewendet. Zum Beispiel gilt für die obige Matrix `Matrix.2`:

```
> exp(Matrix.2)
          [,1]     [,2]
[1,]  2.718282 7.389056
```

```
[2,]  2.718282 7.389056
[3,] 20.085537 1.000000
>
```

1.9 Funktionen zum Erzeugen von Vektoren und Matrizen

Neben der Funktion `c` zum Erzeugen eines Vektors und der Funktion `matrix` zum Erzeugen einer Matrix gibt es auch noch weitere Funktionen zum Erzeugen von Vektoren und Matrizen. Zum Erzeugen von Vektoren sind dies:

Funktion	*Beschreibung*	*Beispiel*
scan	Einlesen von einer externen Datei	scan(), scan("data.dat")
c	Kombiniert beliebige Werte und Vektoren	c(1,3,2,6), c(2,Vektor.1)
rep	Wiederholt Werte und Vektoren	rep(2,5), rep(c(1,3),5)
:	Numerische Folge	1:5, 2:-2
seq	Numerische Folge	seq(-pi,pi,.5)
numeric	Initialisiert einen numerischen Vektor	numeric(5)

Zum Beispiel haben wir:

```
> 1:5
[1] 1 2 3 4 5
> 2:-2.5
[1]  2  1  0 -1 -2
> seq(1,5,1)
[1] 1 2 3 4 5
> seq(1,5,0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
> rep(c(1,3),5)
 [1] 1 3 1 3 1 3 1 3 1 3
>
```

Matrizen können wie folgt erzeugt werden:

Funktion	*Beschreibung*	*Beispiel*
matrix	Erzeugt eine Matrix	matrix(1:12, ncol=4, byrow=T)
cbind	Fügt Spalten (columns) zusammen	cbind(1:10,rep(c(1,2),5))
rbind	Fügt Zeilen (rows) zusammen	rbind(1:10, rep(3,10))

1.10 Funktionen zum Erstellen von Graphiken

In R gibt es etliche Funktionen zum bequemen Erzeugen von Graphiken. Hier wird erstmal die vielseitigste Graphik-Funktion, nämlich die Funktion `plot` vorgestellt. Die anderen folgen später. Diese Funktion hat folgende Form:

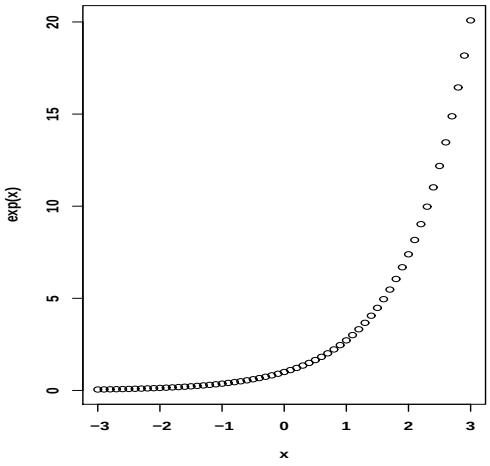

Abbildung 1: Einfache Anwendung der Funktion `plot` zur Darstellung der Exponential-Funktion

```
plot(x, y, xlim=range(x), ylim=range(y), type="p", main, xlab, ylab, ...)
```

Die Argumente `x` und `y` sind erforderliche Argumente und die restlichen Argumente sind optional, weshalb sie auch weggelassen werden können. Die Argumente `x` und `y` stellen die x- und y-Komponenten der zuplottenden Funktion dar und müssen daher Vektoren der gleichen Länge sein. Gilt $x = (x_1, \ldots, x_N)^\top$ und $y = (y_1, \ldots, y_N)^\top$, dann werden bei der Voreinstellung `type=''p''` Punkte/Kreise an den Stellen $(x_1, y_1), \ldots, (x_N, y_N)$ gedruckt. Soll z.B. $x = (x_1, \ldots, x_{61})^\top$ der Vektor sein, der aus $-3, -2.9, -2.8, \ldots, 2.8, 2.9, 3$ besteht, und y der Vektor, der aus $e^{x_1}, \ldots, e^{x_{61}}$ gebildet wird, so können die Punkte $(x_1, y_1), \ldots, (x_{61}, y_{61})$ mit folgenden Befehlen graphisch dargestellt werden:

```
> x<-seq(-3,3,0.1)
> plot(x,exp(x))
```

Es erscheint dann ein zusätzliches Fenster, das die Graphik in Abbildung 1 enthält.

Durch Setzen der optionalen Argumente kann der Ausdruck verändert werden. Die optionalen Argumente haben folgende Bedeutung:

Optionale Argumente der Funktion `plot`

Argument	*Beschreibung*	*Beispiel*
xlim	Dargestellter Bereich der x-Achse	xlim=c(-2,2)
ylim	Dargestellter Bereich der y-Achse	ylim=c(0,30)
main	Überschrift	main="Graph von f(x)=exp(x)"
xlab	Bezeichnung der x-Achse	xlab="x"
ylab	Bezeichnung der y-Achse	ylab="y=exp(x)"
type	Art der Graphik	type="l", Voreinstellung: type="p"

Das Argument `type` bestimmt, wie die Punkte $(x_1, y_1), \ldots, (x_N, y_N)$ dargestellt werden:

type="p"	Die Punkte werden als einzelne Punkte dargestellt (p für "points")

type="l"	Die Punkte werden mit Linien verbunden (l für "lines")
type="b"	Die Punkte werden sowohl als Punkte dargestellt, als auch mit Linien verbunden (b für "both")
type="o"	Auf die Linien werden die Punkte gesetzt (o für "overplotted")
type="h"	Die Punkte werden als senkrechte Linien dargestellt (h für "histogram like")
type="s"	Die Punkte werden stufenförmig mit waagerechten und senkrechten Linien verbunden (s für "steps")
type="S"	Wie type="s", nur mit anderer Stufendarstellung
type="n"	Kein Ausdruck (`n` für "no plotting")

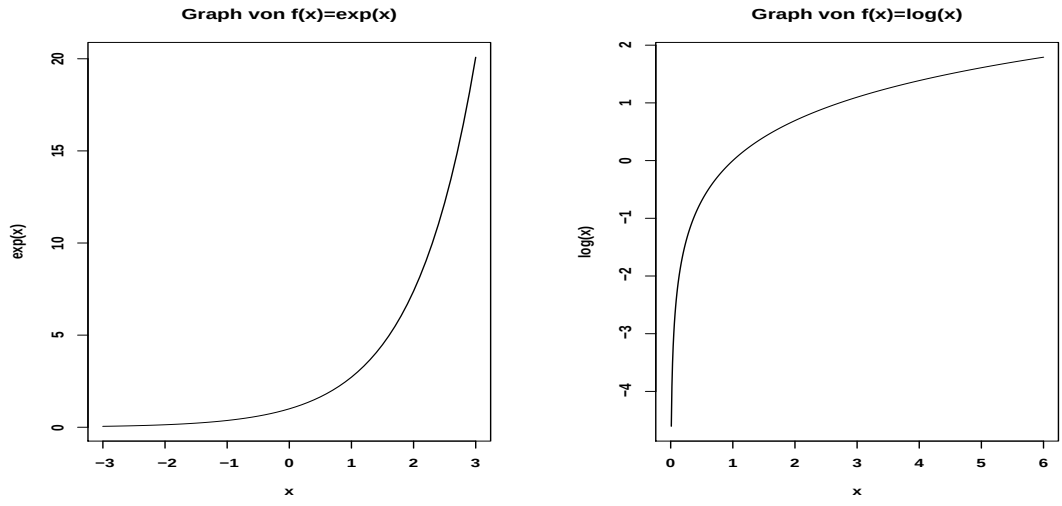

Abbildung 2: Die Exponential- und die Logarithmus-Funktion

Die Darstellungen der Exponential- und Logarithmus-Funktionen in Abbildung 2 wurden mit folgenden Befehlen erzeugt:

```
> x<-seq(-3,3,0.1)
> plot(x,exp(x),type="l",main="Graph von f(x)=exp(x)")
> x<-seq(0,6,0.01)
> plot(x,log(x),type="l",main="Graph von f(x)=log(x)")
```

1.11 Speichern von R-Befehlen

Da R eine interaktive Programmiersprache ist, wird im Kommando-Fenster viel herumprobiert. Alle diese Kommandos können auf Wunsch beim Verlassen von R in der Datei `.Rhistory` gespeichert. Dann werden aber auch alle Kommandos mitgespeichert, die nicht zum Ziel geführt haben. Will man nur die gelungenen Kommandos speichern, gibt es zwei Möglichkeiten. Die einfachste Möglichkeit besteht darin, sie einzeln zu markieren und dann in eine andere Datei zu kopieren. Die andere Möglichkeit besteht darin, sie in selbstdefinierten Funktionen zusammenzufassen. Darauf wird aber erst später eingegangen.

2 Datensätze in R

2.1 Einlesen von Datensätzen

Besteht der Datensatz aus Merkmalsausprägungen eines Merkmals, wobei die Merkmalsausprägungen in einer Intervallskala oder Verhältnisskala gegeben sind, so können die Merkmalsausprägungen mit Hilfe der Funktion `scan` zu einem Vektor eingelesen werden. Besteht zum Beispiel die Datei `Lauf.dat` im Ordner `D:\Projekt1` aus folgenden Laufzeiten [in s] von 100m Läufen

```
10.2, 11.4, 10.5, 9.8, 9.9, 9.9, 10.3, 10.8, 10.2, 11.5,
10.0, 11.4, 11.2, 10.1, 10.5, 12.0, 11.4, 10.2, 10.1, 12.1,
```

so wird die Datei mit folgendem Befehl eingelesen:

```
> scan("D:\\Projekt1\\Lauf.dat",sep=",")
Read 20 items
 [1] 10.2 11.4 10.5  9.8  9.9  9.9 10.3 10.8 10.2 11.5 10.0 11.4 11.2 10.1 10.5
[16] 12.0 11.4 10.2 10.1 12.1
```

Das Argument `sep=’’,’’` gibt an, dass die Merkmalsausprägungen durch Kommas getrennt sind. Die Voreinstellung ist `sep=’’ ’’`, d.h. Leerzeichen trennen die Einzeldaten. Man beachte auch, dass bei der Pfadangabe der Backslash doppelt angegeben werden muss.

Mit der Funktion `scan` können auch Merkmalsausprägungen in einer Ordinalskala eingelesen werden. Da aber Daten in einer Ordinalskala eine andere Bedeutung als Daten in einer Intervallskala oder Verhältnisskala haben, besteht in R die Möglichkeit, den resultierenden Datenvektor zu kennzeichnen, nämlich als “geordneten Faktor“. Mit der Funktion `as.ordered` werden dem Datenvektor die Attribute `level` und `class` zugeordnet. Das Attribute `level` gibt die Stufen der Ordinalskala an, und das Attribute `class` zeigt an, dass es sich um einen gordneten Faktor handelt. Die Attribute können mit der Funktion `attributes` abgefragt werden. In der Datei `examen.dat` befinden sich die folgenden Examensnoten von 12 Teilnehmern:

```
3, 2, 2, 4, 1, 2, 3, 3, 4, 2, 2, 2
```

Diese Examensnoten sollten wie folgt eingelesen werden:

```
> examen<-scan("D:\\Projekt1\\examen.dat", sep=",")
Read 12 items
> examen
 [1] 3 2 2 4 1 2 3 3 4 2 2 2
> examen.ord<-as.ordered(examen)
> examen.ord
 [1] 3 2 2 4 1 2 3 3 4 2 2 2
Levels:  1 < 2 < 3 < 4
> attributes(examen.ord)
$levels
[1] "1" "2" "3" "4"
$class
[1] "ordered" "factor"
```

Die Faktorstufen können auch umbenannt werden. Da es bei einer Ordinalskala nicht auf den Zahlenwert ankommt, sondern nur auf die größer-kleiner Ralationen, können die Faktorstufen zu Wörtern oder allgemein zu Zeichenketten umbenannt werden. Zum Beispiel ist folgendes möglich:

```
> attributes(examen.ord)$levels<-c("Eins","Zwei","Drei","Vier")
> examen.ord
 [1] Drei Zwei Zwei Vier Eins Zwei Drei Drei Vier Zwei Zwei Zwei
Levels:  Eins < Zwei < Drei < Vier
```

Hier zeigt sich, das die Funktion `c` nicht nur Vektoren bestehend aus Zahlen (numerischen Werten) erzeugen kann, sondern auch Vektoren bestehend aus Zeichenketten (characters) erzeugen kann. Zeichenketten müssen aber immer mit den Anführungszeichen “ umgeben sein. Ebenso können Vektoren bestehend aus den logischen Werten TRUE und FALSE erzeugt werden, wobei statt TRUE und FALSE auch T bzw. F benutzt werden kann. Zum Beispiel erzeugt `c(TRUE,FALSE,FALSE,TRUE)` einen logischen Vektor der Länge 4.

Liegen die Merkmalsausprägungen in einer Nominalskala vor, so sind sie oft nicht in Form von Zahlen gegeben sondern in Form von Zeichenketten. In solchen Fällen muss in der Funktion `scan` noch das Argument `what=''character''` angegeben werden. Dann entsteht ein Vektor, dessen Komponenten aus Zeichenketten (characters) bestehen. So ein Vektor kann dann zu einem ungeordneten Faktor mittels der Funktion `as.factor` umgewandelt werden. Innerhalb des Faktors können dann die Zeichenketten einfach zu Zahlenwerten oder auch logischen Werten umgewandelt werden. Die Datei `augen.dat` enthält die folgenden Augenfarben von 10 Personen

```
gruen, blau, blau, braun, grau, gruen, braun, blau, grau, gruen
```

Diese Augenfarben können wie folgt eingelesen und weiterverarbeitet werden:

```
> Augenfarbe<-scan("D:\\Projekt1\\augen.dat",what="character",sep=",")
Read 10 items
> Augenfarbe
 [1] "gruen" "blau"  "blau"  "braun" "grau"  "gruen" "braun" "blau"  "grau"
[10] "gruen"
> Augenfarbe.fac<-as.factor(Augenfarbe)
> Augenfarbe.fac
 [1] gruen blau  blau  braun grau  gruen braun blau  grau  gruen
Levels:  blau braun grau gruen
> attributes(Augenfarbe.fac)
$levels
[1] "blau"  "braun" "grau"  "gruen"
$class
[1] "factor"
> attributes(Augenfarbe.fac)$levels<-c(1,2,3,4)
> Augenfarbe.fac
 [1] 4 1 1 2 3 4 2 1 3 4
Levels:  1 2 3 4
```

Normalerweise besteht aber ein Datensatz nicht nur aus den Merkmalsausprägungen eines Merkmals sondern aus den Ausprägungen mehrerer Merkmale, und diese sind in der Regel in verschiedenen Skalen gegeben und können sowohl numerische Werte als auch Zeichenketten enthalten.

In solchen Fällen kann die Funktion `scan` nicht zum Einlesen der Daten benutzt werden, sondern es muss die Funktion `read.table` benutzt werden. Diese Funktion ergibt dann ein R-Objekt mit dem `class`-Attribute `data.frame` (`data.frame` für "Datentabelle"). Der folgende Datensatz in der Datei `Schlaf.dat` enthält die Veränderung der Schlafzeiten von 20 Patienten, wobei die Patienten entweder das Medikament A oder das Medikament B erhielten. Dabei enthält das Merkmal "Extra" die zusätzlichen Schlafzeiten in Stunden und das Merkmal "Medikament" das Medikament.

"Pat.Nr."	"Extra"	"Medikament"
1	0.0	A
2	-0.2	A
3	3.4	A
4	1.1	B
5	4.4	B
6	3.4	B
7	2.0	A
8	0.8	A
9	1.9	B
10	1.6	B
11	-0.1	A
12	0.8	B
13	5.5	B
14	-1.2	A
15	-0.1	B
16	0.7	A
17	3.7	A
18	4.6	B
19	0.1	B
20	-1.6	A

Dieser Datensatz wird wie folgt eingelesen:

```
> Schlaf<-read.table("D:\\Projekt1\\Schlaf.dat",header=T)
> Schlaf
   Pat.Nr. Extra Medikament
1        1   0.0          A
2        2  -0.2          A
3        3   3.4          A
4        4   1.1          B
5        5   4.4          B
6        6   3.4          B
7        7   2.0          A
8        8   0.8          A
9        9   1.9          B
10      10   1.6          B
11      11  -0.1          A
12      12   0.8          B
13      13   5.5          B
14      14  -1.2          A
```

```
15      15  -0.1          B
16      16   0.7          A
17      17   3.7          A
18      18   4.6          B
19      19   0.1          B
20      20  -1.6          A
> attributes(Schlaf)
$names
[1] "Pat.Nr."    "Extra"      "Medikament"
$class
[1] "data.frame"
$row.names
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14" "15"
[16] "16" "17" "18" "19" "20"
> attributes(Schlaf$Medikament)
$levels
[1] "A" "B"
$class
[1] "factor"
> Schlaf$Medikament
 [1] A A A B B B A A B B A B B A B A A B B A
Levels:  A B
```

2.2 Auswahl aus Datensätzen

Bei der statistischen Auswertung von Datensätzen benutzt man oft nur Teile des Datensatzes. Diese müssen dann aus dem Gesamtdatensatz extrahiert werden. Einzelne Komponenten, Zeilen und Spalten erhält man wie folgt:

Vektorname[*i*]	i-te Komponente des Vektors
Tabellenname[*i,j*]	(i,j)-te Komponente der Tabelle (data.frame) oder Matrix
Tabellenname[*i,*]	i-te Zeile der Tabelle (data.frame) oder Matrix
Tabellenname[*,j*]	j-te Spalte der Tabelle (data.frame) oder Matrix

Zum Beispiel erhält man beim Datensatz mit den Schlafzeiten mit `Schlaf[,2]` nur die zusätzlichen Schlafzeiten in der Spalte "Extra". Haben die Komponenten, Zeilen und Spalten Namen können auch die Namen zur Auswahl benutzt werden. So haben `Schlaf[,''Extra'']` und `Schlaf$Extra` den gleichen Effekt wie `Schlaf[,2]`. An Stelle von einzelnen Komponenten, Zeilen und Spalten können auch mehrere Komponenten, Zeilen und Spalten gleichzeitig ausgewählt werden. Dazu gibt es drei Möglichkeiten:

1. Ein Vektor gibt die Komponenten, Zeilen bzw. Spalten an, die ausgewählt werden. Zum Beispiel werden mit `Schlaf[1:5,]` die ersten fünf Zeilen des Datensatzes `Schlaf` ausgewählt, und mit `Schlaf[c(2,5,8),]` werden die zweite, die fünfte und die achte Zeile ausgewählt.

2. Mit einem negativen Vektor werden die Komponenten, Zeilen bzw. Spalten weggelassen, die der negative Vektor enthält. Zum Beispiel werden mit `Schlaf[-(1:5),]` alle Zeilen bis auf die ersten fünf Zeilen des Datensatzes `Schlaf` ausgewählt.

3. Wird ein Vektor mit logischen Werten benutzt, werden nur die Komponenten, Zeilen bzw. Spalten ausgewählt, bei denen der logische Vektor den Wert T bzw. TRUE hat. Dieser logische Vektor kann mit Vergleichsoperatoren und logischen Operatoren erzeugt werden. Es gibt folgende Vergleichsoperatoren und logische Operatoren, die bei Vektor-wertigen Argumenten wieder komponentenweise agieren:

Vergleichs-Operatoren

$!=$	ungleich
$<$	kleiner als
$<=$	kleiner als oder gleich
$==$	gleich
$>$	größer als
$>=$	größer als oder gleich

Logische Operatoren

!	Negation
|	oder
&	und

Zum Beispiel werden mit dem folgendem Befehl aus dem Datensatz `Schlaf` nur die Patienten ausgewählt, die das Medikament A bekommen haben.

```
> MedA<-Schlaf[,"Medikament"]=="A"
> MedA
 [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE
[13] FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE
> Schlaf[MedA,]
   Pat.Nr. Extra Medikament
1        1   0.0          A
2        2  -0.2          A
3        3   3.4          A
7        7   2.0          A
8        8   0.8          A
11      11  -0.1          A
14      14  -1.2          A
16      16   0.7          A
17      17   3.7          A
20      20  -1.6          A
```

3 Graphische Darstellung von Daten

3.1 Stabdiagramme

Das Stabdiagramm in Abbildung 3 erhält man mit folgenden Befehlen:

```
> examen<-scan("D:\\Projekt1\\examen.dat", sep=",")
> examen.ord<-as.ordered(examen)
```

```
> tabulate(examen.ord)
[1] 1 6 3 2
> examen.tab<-tabulate(examen.ord)/length(examen)
> examen.tab
[1] 0.08333333 0.50000000 0.25000000 0.16666667
> examen.level<-attributes(examen.ord)$levels
> plot(examen.level, examen.tab,type="h", main="Stabdiagramm der Examensnoten",
+ xlab=" ",ylab=" ",xlim=c(0,5),ylim=c(0.0,0.7))
> points(examen.level, examen.tab)
> abline(0,0)
```

Mit der Funktion `tabulate` werden die Häufigkeiten der einzelnen Stufen des Faktors `examen.ord` ermittelt. Teilung dieser Häufigkeiten durch die Länge des Datenvektors `examen` mittels `length(examen)` ergibt die relativen Häufigkeiten. Diese relativen Häufigkeiten werden dann gegen die Stufen geplottet. Dabei wird bei der Funktion `plot` das Argument `type` als `type=''h''` gesetzt, damit senkrechte Stäbe geplottet werden. Die Funktion `points` hat die gleiche Wirkung wie die Funktion `plot`, wenn `type=''p''` gesetzt wird, nur dass bei `points` keine neue Graphik erzeugt wird, sondern die Punkte der vorangegangenen Graphik hinzugefügt werden. Solche Graphik-Funktionen nennt man auch "Low-Level-Graphik-Funktionen", während Graphik-Funktionen, die immer neue Graphiken erzeugen, "High-Level-Graphik-Funktionen" genannt werden. Die Funktion `abline` ist auch eine Low-Level-Garphik-Funktion und fügt eine Gerade mit vorgegebenem konstanten Term und vorgegebener Steigung hinzu, die als Argumente eingesetzt werden. Bei der Darstellung der Examensnoten wird eine waagerechte Gerade durch den Nullpunkt hinzugefügt, so dass der konstante Term und die Steigung gleich Null sind.

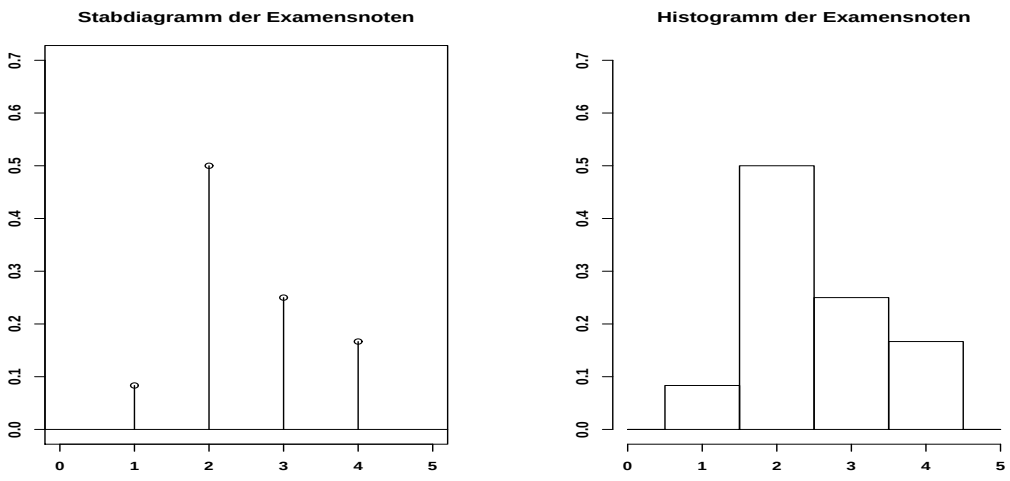

Abbildung 3: Stabdiagramm und Histogramm (Säulendiagramm)

3.2 Histogramme

Das Histogramm in Abbildung 3 wird mit folgendem Befehl erzeugt:

```
> hist(examen,br=c(0,0.5,1.5,2.5,3.5,4.5,5),right=T,freq=F,main=
```

```
+ "Histogramm der Examensnoten",xlim=c(0,5),ylim=c(0,0.7),xlab=" ",ylab=" ")
```

Das Argument `br` (von "break") gibt die Klassengrenzen an, und das Argument `right=T` bewirkt, dass die rechten Klassengrenzen noch zu den Klassen gehören, d.h. rechts abgeschlossene Intervalle als Klassen benutzt werden. Bei `right=F` werden dagegen links abgeschlossene Intervalle benutzt.

Bei ordinalen Daten ist in der Regel eine Klassierung nicht nötig. Dennoch ist es bei der Funktion `hist` sinnvoll Klassengrenzen anzugeben, weil diese bestimmen, wo die Säulen des Histogramms gesetzt werden. Bei dem obigen Beispiel wird dadurch erreicht, dass die Säulen symmetrisch über den Zahlen 1, 2, 3 und 4 stehen. In diesem Fall ist es dann egal, ob die Intervalle rechts abgeschlossen oder links abgeschlossen sind.

Bei stetigen Daten ist aber eine Klassierung notwendig und das Argument `br` erfüllt dann eine wichtige Funktion. Das Histogramm für die Bearbeitungszeiten bei nicht-äquidistanter Klassierung $K_1^*, \ldots, K_9^*$ in Abbildung 4 wurde wie folgt erzeugt:

```
> bearb.zeiten<-scan("bearb.dat",sep="&")
> hist(bearb.zeiten,br=c(60,80,90,100,105,110,115,120,130,150),freq=F,
+ main="Histogramm der Bearbeitungszeiten",ylab=" ",xlab="Klassengrenzen",
+ xlim=c(40,160),ylim=c(0,0.025))
> abline(0,0)
```

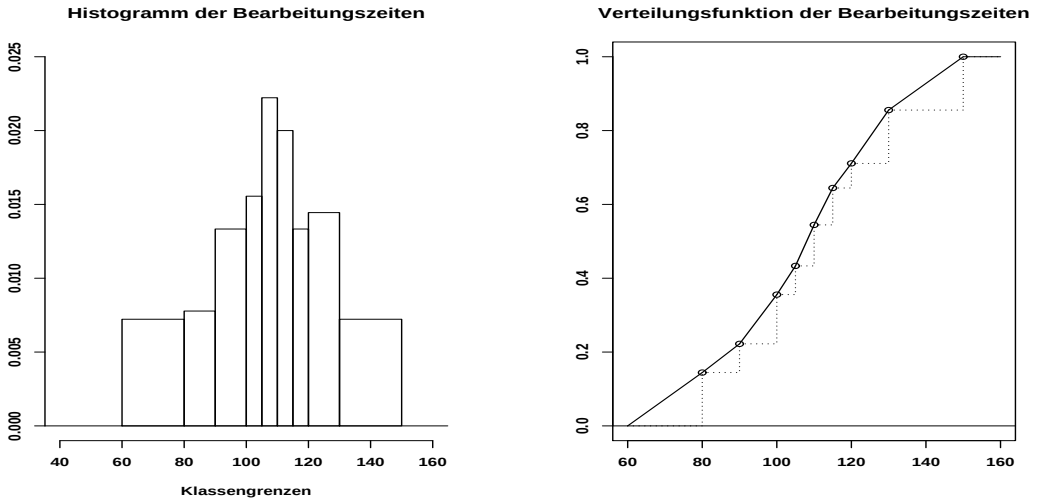

Abbildung 4: Histogramm und Verteilungsfunktion bei nicht-äquidistanter Klassierung

Wird in der Funktion das Argument `plot=F` gesetzt, wird keine Graphik erzeigt. Statt dessen wird eine Liste erstellt, die wichtige Größen zur Erstellung des Histogramms enthält, nämlich die Klassengrenzen, die absoluten Häufigkeiten der Klassen, die Intensitäten h_j und die Klassenmittelpunkte. Zum Beispiel erhält man folgendes:

```
> hist(bearb.zeiten,br=c(60,80,90,100,105,110,115,120,130,150),plot=F)
$breaks
 [1]  60  80  90 100 105 110 115 120 130 150
```

```
$counts
[1] 13  7 12  7 10  9  6 13 13
$intensities
[1] 0.007222222 0.007777778 0.013333333 0.015555556 0.022222222 0.020000000
[7] 0.013333333 0.014444444 0.007222222
$mids
[1]  70.0  85.0  95.0 102.5 107.5 112.5 117.5 125.0 140.0
```

Diese Größen können zur Erzeugung der Verteilungsfunktion in Abbildung 4 wie folgt benutzt werden.

```
> bearb.hist<-hist(bearb.zeiten,br=c(60,80,90,100,105,110,115,120,130,150),
+ plot=F)
> bearb.br<-bearb.hist$breaks[-1]
> bearb.br.1<-c(bearb.hist$breaks,160)
> bearb.counts<-cumsum(bearb.hist$counts)/length(bearb.zeiten)
> bearb.counts.1<-cumsum(c(0,bearb.hist$counts,0))/length(bearb.zeiten)
> plot(bearb.br.1,bearb.counts.1,type="l",lty=1,main=
+ "Verteilungsfunktion der Bearbeitungszeiten",xlab=" ",ylab=" ")
> lines(bearb.br.1,bearb.counts.1,type="s",lty=3)
> points(bearb.br,bearb.counts)
> abline(0,0)
```

Hier wird zuerst mit der Funktion `plot` die approximierende empirische Verteilungsfunktion und dann mit der Funktion `lines` die eigentliche empirische Verteilungsfunktion erstellt. Damit die approximierende und die eigentliche Verteilungsfunktion verschiedene Arten von Linien (durchgezogene Linie - gestrichelte Linie) erhalten, wird das Argument `lty` als `lty=1` bzw. `lty=3` gesetzt. Bei der Funktion `lines` wird das Argument `type=''s''` benutzt, um eine Treppenfunktion zu erhalten. Mit der Funktion `cumsum` (cumulative sums) werden die Häufigkeiten zu Summenhäufigkeiten aufaddiert. Da der Vektor der Klassengrenzen um einen Wert länger als der Vektor der Häufigkeiten ist, muss die erste Klassengrenze weggelassen werden. Das wird durch `bearb.hist$breaks[-1]` erreicht. Daneben ist es sinnvoll einen weiteren Vektor `bearb.br.1` zu erzeugen, der zusätzlich eine Klassengrenze enthält, die größer als alle anderen Klassengrenzen ist. Hier wird zum Beispiel die Klassengrenze 160 zugefügt. Entsprechend muss an dem Vektor der Häufigkeiten am Anfang und am Ende eine Null angehängt werden, da die erste und letzte Klassengrenze Klassen von oben begrenzen, in die keine Daten fallen. Wird dann die Treppenfunktion mit den erweiterten Vektoren `bearb.br.1` und `bearb.counts.1` erzeugt, gibt es bei der obersten Stufe auch eine waagerechte Linie. Diese Linie würde fehlen, wenn nur die Vektoren `bearb.br` und `bearb.counts` benutzt würden.

3.3 Boxplots

Boxplots wie in Abbildung 5 werden mit der Funktion `boxplot` wie fogt erstellt:

```
> Schlaf<-read.table("Schlaf.dat",header=T)
> Schlaf1<-Schlaf[Schlaf[,"Medikament"]=="A","Extra"]
> Schlaf2<-Schlaf[Schlaf[,"Medikament"]=="B","Extra"]
> boxplot(Schlaf1,Schlaf2,main="Boxplots der Schlafzeiten",
+ ylab="Verlaengerung des Schlafes in Stunden",
```

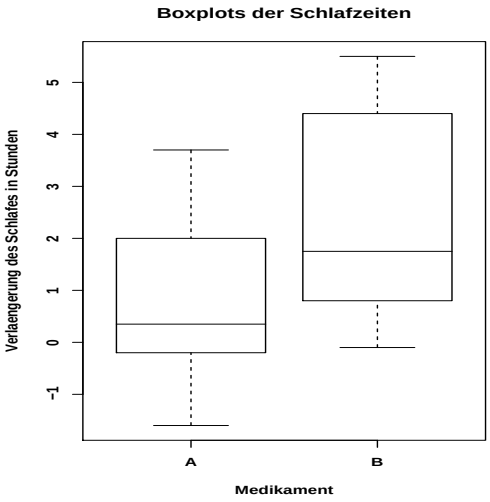

Abbildung 5:

```
+ xlab="Medikament",names=c("A","B"))
```

Wird nur ein Datenvektor als Argument angegeben, wird nur ein Boxplot erstellt. Durch das Setzen des Argumentes `plot=F` wird wieder keine Graphik erstellt. Es werden dann wieder die Größen ausgegeben, die zur Erstellung der Graphik berechnet werden.

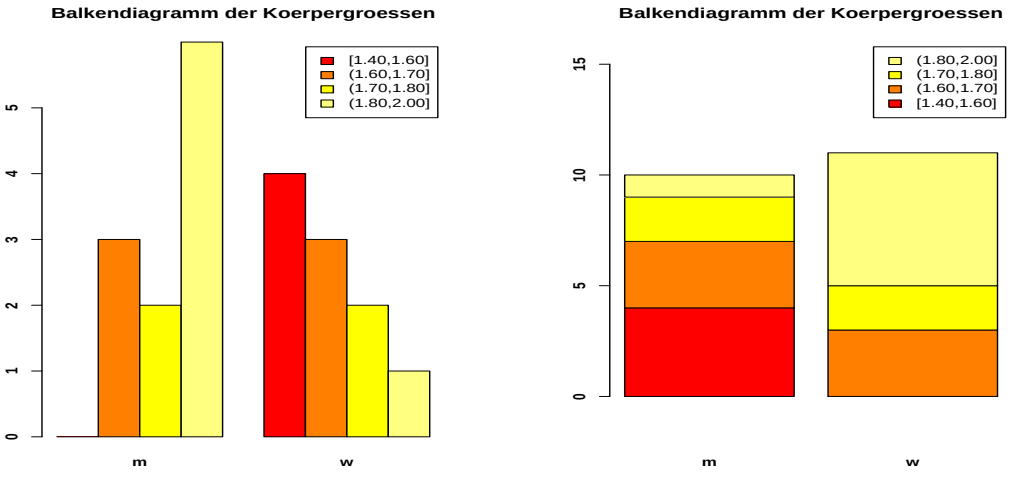

Abbildung 6:

3.4 Balkendiagramme

Balkendiagramme wie in Abbildung 6 werden mit der Funktion `barplot` erstellt. Sollen dabei die Ergebnisse von zwei Gruppen gegenübergestellt werden, müssen die Häufigkeiten der einzelnen Klassen in einer Matrix gegeben werden, deren Zeilen die Klassen darstellen und die Spalten die

Häufigkeiten in den Gruppen enthalten. Bei dem Datensatz mit den Körpergrößen von Männern und Frauen müssen daher die Daten wie folgt bearbeitet werden, bevor die Funktion `barplot` angewendet werden kann.

```
> geschlecht<-matrix(scan("geschlecht.dat"),byrow=T,ncol=2)
> geschlecht.w<-geschlecht[geschlecht[,1]==1,]
> geschlecht.m<-geschlecht[geschlecht[,1]==0,]
> geschlecht.w.tab<-hist(geschlecht.w[,2],
+ br=c(1.4,1.6,1.7,1.8,2.0),plot=F)$counts
> geschlecht.m.tab<-hist(geschlecht.m[,2],
+ br=c(1.4,1.6,1.7,1.8,2.0),plot=F)$counts
> geschlecht.tab<-cbind(geschlecht.m.tab,geschlecht.w.tab)
> geschlecht.tab
     geschlecht.m.tab geschlecht.w.tab
[1,]                0                4
[2,]                3                3
[3,]                2                2
[4,]                6                1
> barplot(geschlecht.tab,beside=T,legend.text=
+ c("[1.40,1.60]","(1.60,1.70]","(1.70,1.80]","(1.80,2.00]"),
+ names.arg=c("m","w"),main="Balkendiagramm der Koerpergroessen")
> barplot(geschlecht.tab[,c(2,1)],beside=F,legend.text=
+ c("[1.40,1.60]","(1.60,1.70]","(1.70,1.80]","(1.80,2.00]"),
+ names.arg=c("m","w"),main="Balkendiagramm der Koerpergroessen",
+ ylim=c(-2,16))
```

Durch das Argument `beside=T` bzw. `beside=F` wird bestimmt, ob die Balken nebeneinander oder übereinander gepackt werden.

3.5 Streudiagramme und Regressionsgeraden

Sreudiagramme können ganz einfach mit der Funktion `plot` erstellt werden, wobei das Argument `type=''p''` benutzt wird. Soll in das Streudiagramm noch eine Regressionsgerade eingetragen werden, kann das mit der Low-Level-Graphik-Funktion `abline` erfolgen, wobei der konstante Term und die Steigung mit der Funktion `lsfit` (von "least squares fit"="kleinste Quadrate Anpassung") bestimmt werden kann. Die Funktion `lsfit` gibt eine Liste von verschiedenen Rechenergebnissen aus, wobei der konstante Term (Intercept) und die Steigung in der Listenkomponente `coef` (`coef` für "Koefficient") enthalten ist. Das folgende Beispiel zeigt, wie das Streudiagramm und die Regressionsgerade für die Körpergewicht/Körpergrößen-Daten in Abbildung 7 erzeugt werden:

```
> KGewicht<-matrix(scan("Kgewicht.dat"),byrow=T,ncol=2)
> plot(KGewicht[,2],KGewicht[,1],type="p",main=
+ "Regressionsgerade: Groesse/Gewicht",ylab="Koerpergewicht [in kg]",
+ xlab="Koerpergroesse [in cm]")
> lsfit(KGewicht[,2],KGewicht[,1])$coef
  Intercept           X
-37.4970610   0.6033273
> abline(lsfit(KGewicht[,2],KGewicht[,1])$coef)
> text(175,55,"f(x)=0.603x-37.497")
```

Der letzte Befehl mit der Low-Level-Graphik-Funktion `text` bewirkt, dass der Text `f(x)=0.603x-37.497` an der Stelle mit den Koordinaten (175,55) hinzugefügt wird.

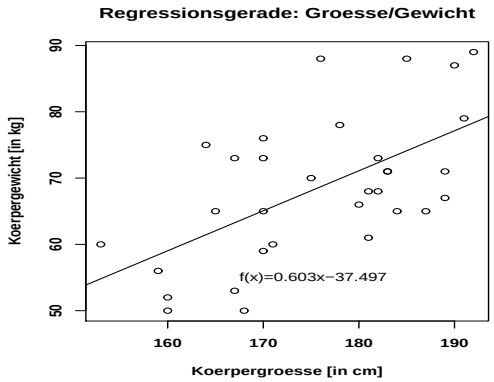

Abbildung 7:

Gibt es mehr als zwei quantitative Variablen, so können mit der Funktion `pairs` die Streudiagramme für alle Paare von Variablen in einer Grafik dargestellt werden. Der Datensatz `Irises.dat` enthält die Längen und Breiten der Kelch-und Blütenblätter (`SepL`=Sepal Length, `SepW`=Sepal Width, `PetL`=Petal Length, `PetW`=Petal Width) von den Iris-Arten Iris setosa, Iris versicolor, Iris virginica. Die Streudiagramme aller Paare der vier Variablen für Iris setosa werden wie folgt erzeugt:

```
> iris.dat<-matrix(scan("Irises.dat"),ncol=12,byrow=T)
Read 600 items
> iris0.dat<-rbind(iris.dat[,1:4],iris.dat[,5:8],iris.dat[,9:12])
> colnames(iris0.dat)<-c("SepL","SepW","PetL","PetW")
> art<-factor(c(rep("set",50),rep("vers",50),rep("virg",50)))
> iris.df<-data.frame(iris0.dat,art)
> pairs(iris.df[1:50,1:4])
> title("Iris setosa")
```

Mit der Funktion `lsfit` kann auch eine multiple Regression zum Beispiel für die Variable `SepL` in Abhängigkeit von den Variablen `SepW, PetL, PetW` für Iris setosa durchgeführt werden:

```
> lsfit(iris.df[1:50,2:4],iris.df[1:50,1])
```

4 Statistische Kenngrößen und Berechnungen

Die wichtigsten statistischen Kenngrößen werden durch folgende Funktionen gegeben:

Statistische Kenngrößen

mean	Arithmetisches Mittel
median	Median
quantile	Empirisches Quantil
mode	Modalwert
max	Maximaler Wert
min	Minimaler Wert
length	Länge des Datenvektors
mad	Median der absoluten Abweichungen (MAD)
var	Varianz bei einem Vektor, Kovarianz-Matrix bei einer Matrix
cor	Korrelation zwischen Matrizen oder Vektoren
sum	Summe der Werte eines Vektors
prod	Produkt der Werte eines Vektors
any	Logische Summe eines logischen Vektors
all	Logische Produkt eines logischen Vektors
lsfit	Schätzung der Parameter der linearen Regression
density	Schätzung der Wahrscheinlichkeitsdichte

Das arithmetische Mittel eines Vektors `x` erhält man mit `mean(x)` aber ebenso mit `sum(x)/length(x)`. Sowohl der Befehl `var(x)` als auch der Befehl `var(x,x)` ergeben die Varianz des Vektors `x`. Die Kovarianz zweier Vektoren `x1` und `x2` erhält man mit `var(x1,x2)`.

Neben den Funktionen zur Berechnung der statistischen Kenngrößen gibt es noch etliche Funktionen, die für die statistische Auswertung von Bedeutung sind. Darunter fallen Sortierfunktionen, Funktionen, die Klasseneinteilungen vornehmen, und Funktionen, die Häufigkeiten bestimmen. Ein Teil dieser Funktionen wurde schon in den vorangegangenen Beispielen benutzt.

Sortier-Funktionen und andere statistische Funktionen

rev	Die Werte eines Vektors werden in umgekehrter Reihenfolge gegeben
sort	Die Werte eines Vektots werden sortiert
order	Ergibt die Permutation der Komponenten, die den sortierten Vektor ergeben
rank	Bildet die Ränge der Werte eines Vektors
match	Zeigt, welche Werte eines Vektors mit welchen Werten einer Tabelle übereinstimmen
cumsum	Kummulierte Summen der Werte eines Vektors
cumprod	Kummulierte Produkte der Werte eines Vektors
cut	Zerteilt einen Vektor in Klassen und erzeugt einen Faktor
tabulate	Ergibt die Häufigkeiten der Werte/Stufen eines Vektors/Faktors
table	Ergibt eine Tafel der Häufigkeiten von Stufenkombinationen

Das Sortieren eines Vektors `x` kann sowohl mit `sort(x)` als auch mit `x[order(x)]` erreicht werden. Mit Hilfe der Funktionen `cor` und `rank` wird der Rangkorrelationskoefficient von Spearman von Vektoren `x1` und `x2` durch `cor(rank(x1),rank(x2))` berechnet.

Soll eine statistische Funktion auf mehrere Spalten oder Zeilen einer Matrix oder einer Datentabelle angewendet werden, so muss man dies nicht für die einzelnen Spalten (Zeilen) getrennt durchführen, sondern kann es mit der Funktion `apply` für alle Spalten (Zeilen) gleichzeitig

durchführen. Die allgemeine Gestalt der Funktion `apply` ist

`apply(`*Matrixname, 2, Funktionsname, zusätzliche Argumente der Funktion*`)`

für die Anwendung auf Spalten und

`apply(`*Matrixname, 1, Funktionsname, zusätzliche Argumente der Funktion*`)`

für die Anwendung auf Zeilen.

Soll zum Beispiel das arithmetische Mittel und das 25%-getrimmte Mittel sowohl für die Körpergewichte als auch die Körpergrößen der Körpergewicht/Körpergrößen-Daten berechnet werden, kann das wie folgt ausgeführt werden:

```
>  KGewicht<-matrix(scan("Kgewicht.dat"),byrow=T,ncol=2)
Read 64 items
> apply(KGewicht,2,mean)
[1]  68.5000 175.6875
> apply(KGewicht,2,mean,trim=0.25)
[1]  68.25 176.25
```

Man beachte, dass das Argument `trim` ein optionales Argument der Funktion `mean` ist. Die Voreinstellung ist `trim=0`. Ist das Argument `trim` gleich einem α, dann werden 100*α% der niedrigsten und 100*α% der höchsten Werte weggeschnitten (weggetrimmt) und der Mittelwert wird von den verbleibenden Werten gebildet.

Liegt statt einer Matrix eine Daten-Tabelle oder eine Liste vor, so muss statt `apply` entweder `tapply`, `lapply` oder `sapply` verwendet werden.

5 Wahrscheinlichkeitsverteilungen

R enthält zahlreiche Wahrscheinlichkeitsverteilungen. Von diesem können sowohl die Dichte-Funktionen, die empirischen Verteilungsfunktionen und die Quantile benutzt werden. Daneben können Zufallszahlen (eigentlich Pseudo-Zufallszahlen) mit einer gegebenen Wahrscheinlichkeitsverteilung erzeugt werden. Bezeichnet *distname* eine Wahrscheinlichkeitsverteilung ("dist" von "distribution"="Verteilung"), so können allgemein die Funktionswerte der Dichte-Funktionen, der empirischen Verteilungsfunktionen, die Quantile und die Zufallszahlen wie folgt aufgerufen werden:

Charakteristika von Wahrscheinlichkeitsverteilungen

d*distname*(x)	Berechnet die Dichte-Funktion für die Werte in `x`.
p*distname*(q)	Berechnet die Verteilungsfunktion für die Werte in `q`.
q*distname*(p)	Berechnet die Quantile von den Werten in `p`.
r*distname*(n)	Erzeugt `n` Zufallszahlen.

Die Argumente `x`,`q` und `p` können Vektoren sein.

Folgende Wahrscheinlichkeitsverteilungen stehen in R zur Verfügung:

Wahrscheinlichkeitsverteilungen und deren Argumente

distname	Verteilung (in Englisch)	Erforderliche Argumente	Optionale Argumente	Voreinstellung
beta	Beta	shap1, shape 2		
binom	Binomial	size, prob		
cauchy	Cauchy		location, scale	0, 1
chisq	Chi-squared	df		
exp	Exponential		rate	1
f	F	df1, df2		
gamma	Gamma	shape		
geom	Geometric	prob		
hyper	Hypergeometric	m, n, k		
lnorm	Log-normal		meanlog, sdlog	0, 1
logis	Logistic		location, scale	0, 1
nbinom	Negative binomial	size, prob		
norm	Normal		mean, sd	0, 1
pois	Poisson	lambda		
stab	Stable	index	skewness	-, 0
t	Student's t	df		
unif	Uniform		min, max	0, 1
weibull	Weibull	shape	scale	-, 1
wilcox	Wilcoxon rank sum	m, n		

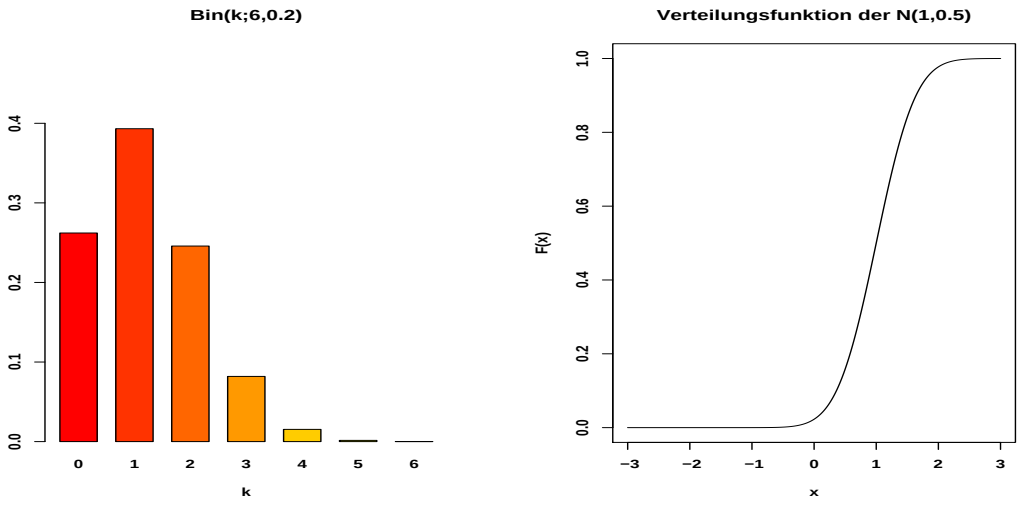

Abbildung 8: Dichte einer Binomialverteilung und Verteilungsfunktion einer Normalverteilung

Die Darstellungen in Abbildung 8 wurden durch

```
> barplot(dbinom(0:6,6,0.2),space=0.5,names.arg=c("0","1","2","3","4","5","6"),
+ ylim=c(-0.001,0.5),main="Bin(k;6,0.2)",xlab="k")
```

für die Dichte der Binialverteilung und durch

```
> plot(seq(-3,3,0.01),pnorm(seq(-3,3,0.01),1,0.5),main=
```

```
 + "Verteilungsfunktion der N(1,0.5)",xlab="x",ylab="f(x)",ylim=c(0,1),type="l")
```
für die Verteilungsfunktion der Normalverteilung erzeugt.

6 Sebstdefinierte Funktionen

6.1 Definition einer Funktion im Kommando-Fenster

Eine selbstdefinierte Funktion mit dem Namen *Funktionsname* kann wie folgt definiert werden:

Funktionsname `<- function(`*Argumente*`){`*R-Befehle*`}`

Die Argumente spielen die gleiche Rolle wie bei den von R vorgegebenen Funktionen. Es gibt erforderliche Argumente und optionale Argumente, die eine Voreinstellung besitzen. Auch kann eine Funktion gar keine Argumente voraussetzen. Generell gibt es keine Unterschiede zwischen selbstdefinierten Funktionen und den vorgegebenen Funktionen. Der einzige Unterschied besteht darin, dass die selbstdefinierten Funktionen mittels `ls()` oder `objects()` zusammen mit den anderen R-Objekten aufgelistet werden.

Die R-Befehle können beliebig viele Befehle enthalten. Als letztes kommt der Wert oder auch eine Liste von Werten, die ausgegeben werden soll. Zum Beispiel kann man folgende Funktion definieren, die die Zahlen von 1 bis n und 1 bis m aufaddiert:

```
> sumnm<-function(n,m){
+ x<-1:n
+ y<-1:m
+ z<-sum(x)+sum(y)
+ z
+ }
> sumnm(5,10)
[1] 70
```

6.2 Definition einer Funktion im R-Editor

Bei komplizierteren Funktionen ist es sinnvoll, diese nicht im Kommando-Fenster zu schreiben sondern in einem Extra-Fenster. Um eine neue Funktion namems *Funktionsname.neu* im Extra-Fenster zu schreiben, benutzt man den Befehl:

`> fix(`*Funktionsname.neu*`)`

Es wird dann ein Fenster mit einem von R vorgegebenen Editor aufgemacht. Zwischen die geschweiften Klammern schreibt man, ohne die R-Anfangszeichen “`>`“ und “`+`“ zu benutzen, die R-Befehle. So würde die obige Funktion `sumnm` im Editor-Fenster folgendermaßen geschrieben werden:

```
function(n,m){
x<-1:n
y<-1:m
z<-sum(x)+sum(y)
z
}
```

Nach dem Speichern und Schließen des Editor-Fensters, ist dann die neue Funktion in R unter dem Namen *Funktionsname.neu* vorhanden und kann im Kommando-Fenster benutzt werden. Ist allerdings die Syntax der neuen Funktion nicht korrekt, kommt die Auffordung, sie mit

Funktionsname.neu`<- edit()`

zu korrigieren. Mit `edit()` wird wieder das Editor-Fenster mit der fehlerhaften Funktion geöffnet. Will man dagegen eine vorhandene, korrekte Funktion namens *Funktionsname* abändern, dann wird mit

Funktionsname`<- edit(`*Funktionsname*`)`

das Editor-Fenster mit der Funktion *Funktionsname* geöffnet, und die Änderung kann darin vorgenommen werden.

6.3 Definition und Speichern von Funktionen in einer externen Datei

Will man die Funktion nicht in dem von R vorgegebenen Editor schreiben, kann man sie auch in einer externen Datei schreiben. Dabei muss in der externen Datei zusätzlich noch der Funktionsname hinzugefügt werden. Z.B. muss für das obige Beispiel der Funktion `sumnm` dann die Datei folgendes enthalten:

```
sumnm<-
function(n,m){
x<-1:n
y<-1:m
z<-sum(x)+sum(y)
z
}
```

Ist dieser Inhalt in der Datei `sumnm.txt` im Ordner `D:\Projekt1` abgespeichert, kann die Funktion im Kommando-Fenster von R wie folgt mit der Funktion `source` geladen werden:

```
> source("D:\\Projekt1\\sumnm.txt")
```

Die externe Datei kann auch mehrere selbstdefinierte Funktionen enthalten. Ebenso können darin auch andere R-Objekte definiert werden. Die Umkehrung ist aber auch möglich. Selbstdefinierte Funktionen und Objekte, die in R vorhanden sind, können mit der Funktion `dump` in einer externen Datei im ASCII-Format gespeichert werden. Aus dieser Datei können sie später mittels der Funktion `source` wieder geladen werden. Allgemein sieht die Anwendung von `dump` wie folgt aus:

`dump(`*Liste*`,fileout=''`*Dateiname*`'')`

Die *Liste* ist ein Vektor bestehend aus den Zeichenketten, die die Namen der zu speichernden Funktionen und Objekte angeben. Z.B. kann eine Anwendung wie folgt aussehen:

```
> dump(c("Funktion1","Objekt1","Funktion2"),"D:\\Projekt1\\Neu.txt")
```

6.4 Bedingte und wiederholte R-Befehle

Das Schreiben von Funktionen ist insbesondere dann sinnvoll, wenn bedingte Abfragen durchgeführt oder Befehle sehr oft wiederholt werden sollen.

Die bedingte Abfrage hat entweder die Gestalt

`if(`*Bedingung*`)` { *R-Befehle* }

oder

`if(`*Bedingung*`)` { *R-Befehle* }
`else` { *R-Befehle* }

Die *Bedingung* muss ein logischer Wert sein. Nur wenn dieser Wert `TRUE` bzw. `T` ist, werden die R-Befehle hinter `if` ausgeführt. Ist dieser Wert `FALSE` bzw. `F` werden im zweiten Fall die R-Befehle hinter `else` ausgeführt.

Befehle werden durch sogenannte Schleifen wiederholt. In R hat die Schleife die Form

`for(` *Name* `in` *Vektor* `)` { *R-Befehle* }

Beim Durchlaufen der Schleife nimmt die Variable *Name* nacheinander die Werte in dem *Vektor* an. Zum Beispiel erhalten wir folgendes:

```
> x<-1
> for(i in 1:5){x<-x*i}
> x
[1] 120
```

Das heißt, nach dem Durchlaufen der Schleife enthält die Variable `x` den Wert $1*2*3*4*5 = 120$.

6.5 Zwischenausgabe und Kommentare

Um Fehler in einer Funktion zu finden, ist es hilfreich, Zwischenergebnisse ausgeben zu lassen. Das kann mit der Funktion `cat` gemacht werden. Zum Beispiel gibt `cat('': '', x, ''\n'')` zuerst im Kommando-Fenster die Zeichenkette `''x: ''` aus, dann den Inhalt der Variable `x`, und beginnt dann mit `''\n''` eine neue Zeile.

Selbstdefinierte Funktionen sind leserlicher, wenn die einzelnen Schritte durch Kommentare erklärt werden. Solche Kommentare können mit dem Zeichen "#" gekennzeichnet werden. Alles, was nach dem Zeichen "#" kommt, wird von R nicht berücksichtigt.

7 Zweidimensionale Normalverteilung

Es gibt in R keine vorgegebene Funktion für die zweidimensionale Normalverteilung. Sie kann aber mit folgender Funktion selber geschrieben werden:

```
dnorm.two<-
function(x,y,mu,sigma1,sigma2,rho)
# Erzeugt eine Matrix mit den Werten der zwei-dimensionalen Normalverteilung mit
# Erwartungsvektor mu, Varianzen sigma1^2 und sigma2^2 und Korrelation rho
# an den Stellen z=(x[i],y[j]) aus den Vektoren x und y
{
matnorm<-matrix(rep(0,length(x)*length(y)),ncol=length(y))
Sigma<-matrix(c(sigma1^2,rho*sigma1*sigma2,rho*sigma1*sigma2,sigma2^2),ncol=2)
```

```
Sigma.det<-sigma1^2*sigma2^2-(rho*sigma1*sigma2)^2
Sigma.inv<-matrix(c(sigma2^2,-rho*sigma1*sigma2,-rho*sigma1*sigma2,sigma1^2),
      ncol=2)/ Sigma.det
for(i in 1:length(x)){
   for(j in 1:length(y)){
      z<-c(x[i],y[j])-mu
      matnorm[i,j]<-exp((-0.5)*t(z)%*%Sigma.inv%*%z)/(2*pi*sqrt(Sigma.det))
   }
}
matnorm
}
```

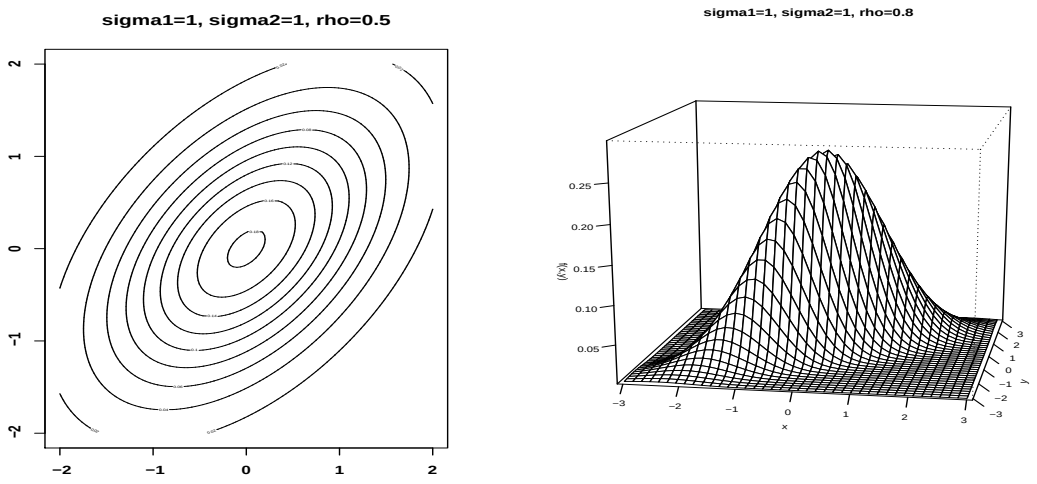

Abbildung 9: Zweidimensionale Normalverteilung

Zweidimensionale Wahrscheinlichkeitsverteilungen und generell Funktionen mit zweidimensionalen Argumenten können mit den Graphik-Funktionen `contour`, `persp` und `image` garphisch dargestellt werden. Die Darstellungen der zweidimensionalen Normalverteilungen in Abbildung 9 wurden wie folgt erzeugt:

```
> x<-seq(-2,2,length=100)
> y<-seq(-2,2,length=100)
> twonorm..1..1..0.5<-dnorm.two(x,y,mu=c(0,0),sigma1=1,sigma2=1,rho=0.5)
> contour(x,y,twonorm..1..1..0.5)
> title("sigma1=1, sigma2=1, rho=0.5")

> x<-seq(-3,3,length=40)
> y<-seq(-3,3,length=40)
> twonorm..1..1..0.8<-dnorm.two(x,y,mu=c(0,0),sigma1=1,sigma2=1,rho=0.8)
> persp(x,y,twonorm..1..1..0.8,zlab="f(x,y)",ticktype="detailed",
+ theta=10,zlim=c(0,0.3),xlim=c(-3.1,3.1),ylim=c(-3.1,3.1),r=9)
```

```
> title("sigma1=1, sigma2=1, rho=0.8")
```

8 Simulationen

Das Verhalten von Zufallsprozessen kann mit Hilfe der Zufallszahlen einfach simuliert werden. So können die relativen Häufigkeiten der Sechs bei 1 bis n Würfelwurfen mit folgender selbstdefinierten Funktion simuliert werden:

```
freq6<- function(n) # Anteil der Sechsen in 1 bis n W\"{u}rfelwuerfen
{ wurf<-ceiling(6*runif(n)) wurf<-as.numeric(wurf==6)
result<-rep(0,n) for(i in 1:n){
  result[i]<-sum(wurf[1:i])/i
}
result
}
```

Mit der Funktion `runif` werden Zufallszahlen erzeugt, die gleichförmig im Intervall $[0, 1]$ verteilt sind. Durch Multiplikation dieser Zufallszahlen mit 6 und Aufrunden erhält man die Zahlen 1, 2 , 3, 4, 5, 6 mit gleicher Wahrscheinlichkeit. Mit `wurf==6` wird ein logischer Vektor erzeugt, der den Wert `TRUE` an den Stellen hat, wo vorher die 6 stand. Mit der Funktion `as.numeric` wird der logische Vektor wieder in einen numerischen Vektor umgewandelt, der an Stelle von `TRUE` die 1 und an Stelle von `FALSE` die 0 enthält.

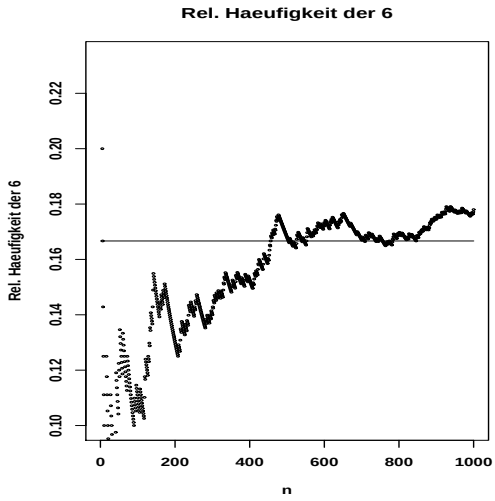

Abbildung 10:

Mit folgenden Befehlen wurde dann die graphische Darstellung der Simulationsergebnisse in Abbildung 10 erzielt:

```
> freq6.1000<-freq6(1000)
> par(cex=1.5)
> plot(1:1000,rep(1/6,1000),xlab="n",ylab="Rel. Haeufigkeit der
> 6",type="l", main="Rel. Haeufigkeit der 6")
> par(cex=0.5)
```

```
> points(1:1000,freq6.1000)
```

Die Funktion `par` setzt verschiedenste Graphik-Parameter. Mit dem Argument `cex` wird hier die Font-Größe so beeinflußt, dass der Rahmen und die waagerechte Linie durch 1/6 dicker als die Punkte sind, die die Häufigkeiten angeben.

9 Libraries (Programm-Bibliotheken)

Für einige statistische Analysen müssen spezielle Pakete geladen werden. Mit dem Befehl

```
> library()
```

werden die vorhandenen Pakete aufgelistet.

```
> library(help=Paket-Name)
```

ergibt eine Beschreibung des Paketes. Mit

```
> library(Paket-Name)
```

wird das Paket geladen.

Einige Pakete wurden schon bei der Installation des Basis-Paketes mitinstalliert, andere können durch "Packages", das in der ersten Zeile des Menü-Fensters steht, dazu installiert werden. Dazu braucht man nur die ZIP-Datei des Paketes, die dann durch "Packages" gewählt wird.

10 Statistische Analyse für univariate Daten

Ein Paket, das mit dem Basis-Paket mitinstalliert wurde, ist das Paket `ctest`, das die klassischen Tests wie den t-Test, den Binomial-Test oder den Wilcoxon-Test enthält. Dieses Paket wird wie folgt geladen (für `R2.0.1` nicht mehr nötig, da in `R2.0.1` diese Library in `stats` integriert wurde, die automatisch geladen wird):

```
> library(ctest)
```

Dann sind folgende Tests vorhanden:

t.test	wilcox.test	chisq.test
var.test	kruskal.test	fisher.test
binom.test	friedman.test	mcnemar.test
prop.test	cor.test	mantelhaen.test

Um z.B. die Vokabelanzahlen, die mit den Methoden 1 und 2 gelernt wurden, mit dem t-Test auf Gleichheit der Erwartungswerte zu testen, führt man folgende Schritte durch:

```
> vokabel.dat<-read.table("vokabel.dat")
> dimnames(vokabel.dat)[[2]]<-c("vokabelanz","methode")
> vokabel.meth1<-vokabel.dat[vokabel.dat[,"methode"]==1,]
> vokabel.meth2<-vokabel.dat[vokabel.dat[,"methode"]==2,]
> library(ctest)
> t.test(vokabel.meth1[,"vokabelanz"],vokabel.meth2[,"vokabelanz"])
```

```
        Welch Two Sample t-test

data:  vokabel.meth1[, "vokabelanz"] and vokabel.meth2[,
"vokabelanz"] t = 7.1621, df = 14.71, p-value = 3.657e-06
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 18.09111 33.45889
sample estimates: mean of x mean of y
   82.375    56.600
```

Im Basis-Paket sind aber auch schon etliche Tests enthalten. Dies sind vor allem Tests, die die lineare Regression und die Varianzanalyse betreffen:

lsfit	Fits a linear model with least squares regression.
lm	Fits a linear model with least squares method.
glm	Fits a generalized linear model.
aov	Fits analysis of variance models.
summary.lm	Prints out a summary of the fit.
lm.influence	Returns a list with information on the influence of each observation.
update	Updates a fit.
anova	Produces the usual ANOVA table for one fit or compares nested models.

Um eine einfaktorielle Varianzanalyse auf den Datensatz `vokabel.dat` zum Testen der gleichen Erwartungswerte bei den drei Methoden anzuwenden, sind folgende Schritte nötig. Dabei ist wichtig, dass die Spalte, die die Methoden enthält, in einen Faktor umgewandelt wird, da die Methoden durch Zahlen angegeben sind. Ohne das Umwandeln in einen Faktor würde R diese dann als Regressoren wie in einer linearen Regression benutzen.

```
> vokabel.dat<-read.table("vokabel.dat")
> dimnames(vokabel.dat)[[2]]<-c("vokabelanz","methode")
> vokabel.dat[,"methode"]<-as.factor(vokabel.dat[,"methode"])
> anova(lm(vokabelanz~methode,vokabel.dat))
Analysis of Variance Table

Response: vokabelanz
           Df Sum Sq Mean Sq F value    Pr(>F)
 methode    2 3746.2  1873.1  24.612 1.537e-06 ***
 Residuals 24 1826.5    76.1
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1
‘ ’ 1

> summary(lm(vokabelanz~methode,vokabel.dat))

Call: lm(formula = vokabelanz ~ methode, data = vokabel.dat)

Residuals:
    Min      1Q  Median      3Q     Max
```

```
-17.556  -4.987  -0.375   5.512  14.400

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   82.375      3.084  26.708  < 2e-16 ***
 methode2    -25.775      4.138  -6.229 1.95e-06 ***
 methode3    -25.819      4.239  -6.091 2.73e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1
‘ ’ 1

Residual standard error: 8.724 on 24 degrees of freedom Multiple
R-Squared: 0.6722,     Adjusted R-squared: 0.6449 F-statistic:
24.61 on 2 and 24 degrees of freedom,      p-value: 1.537e-006
```

Die Schätzungen 82.375, -25.775, -25.819 sind Schätzungen für die Parameter ν, α_2 und α_3 in der Modellierung

$$\begin{aligned} Y_{1j} &= \nu + \epsilon_{1j}, \\ Y_{ij} &= \nu + \alpha_i + \epsilon_{ij} \text{ für } i = 2, 3, \end{aligned}$$

Es kann aber auch die Modellierung

$$Y_{ij} = \mu + a_i + \epsilon_{ij} \text{ für } i = 1, 2, 3,$$

mit der Nebenbedingung $a_1 + a_2 + a_3 = 0$ benutzt werden. Dabei bestehen folgende Beziehungen zwischen den Parametern:

$$\begin{aligned} \nu &= \mu + a_1, \\ \nu + \alpha_2 &= \mu + a_2, \\ \nu + \alpha_3 &= \mu + a_3. \end{aligned}$$

Daraus ergeben sich folgende Beziehungen zwischen den Schätzungen:

$$\begin{aligned} \widehat{\nu} &= \widehat{\mu} + \widehat{a}_1, \\ \widehat{\alpha}_2 &= \widehat{\mu} + \widehat{a}_2 - \widehat{\nu} = \widehat{a}_2 - \widehat{a}_1 = \overline{Y}_{2\cdot} - \overline{Y}_{1\cdot}, \\ \widehat{\alpha}_3 &= \widehat{\mu} + \widehat{a}_3 - \widehat{\nu} = \widehat{a}_3 - \widehat{a}_1 = \overline{Y}_{3\cdot} - \overline{Y}_{1\cdot}. \end{aligned}$$

Eine zweifaktorielle Varianzanalyse wird auf den Datensatz `antibiotika.dat` wie folgt angewendet. Dabei wird durch die Addition von `V2*V3` in der Modellgleichung angegeben, dass Wechselwirkungen zwischen den Faktoren `V2` und `V3` im Modell angenommen werden.

```
> antibiotika.dat<-read.table("antibiotika.dat")
> antibiotika.dat
    V1            V2       V3
```

```
 1  38 Antibiotikum1 Erreger1
 2  40 Antibiotikum2 Erreger1
 .   .      .           .
 .   .      .           .
 .   .      .           .
> anova(lm(V1~V2+V3+V2*V3,antibiotika.dat))
Analysis of Variance Table

Response: V1
          Df Sum Sq Mean Sq F value  Pr(>F)
 V2        2 57.333  28.667    6.88 0.01538 *
 V3        2  4.333   2.167    0.52 0.61135
 V2:V3     4 93.333  23.333    5.60 0.01521 *
 Residuals 9 37.500   4.167
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1
‘ ’ 1

> summary(lm(V1~V2+V3+V2*V3,antibiotika.dat))

Call: lm(formula = V1 ~ V2 + V3 + V2 * V3, data = antibiotika.dat)

Residuals:
       Min         1Q     Median         3Q        Max
-3.000e+00 -1.250e+00  8.728e-18  1.250e+00  3.000e+00

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
 (Intercept)   36.500      1.443  25.288 1.14e-09 ***
  V22           4.000      2.041   1.960  0.08170 .
  V23           2.000      2.041   0.980  0.35279
  V32           7.000      2.041   3.429  0.00752 **
  V33           3.000      2.041   1.470  0.17571
 V22.V32      -11.500      2.887  -3.984  0.00319 **
 V23.V32      -12.000      2.887  -4.157  0.00246 **
 V22.V33       -5.500      2.887  -1.905  0.08914 .
 V23.V33       -7.000      2.887  -2.425  0.03830 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1
‘ ’ 1

Residual standard error: 2.041 on 9 degrees of freedom Multiple
R-Squared: 0.8052,     Adjusted R-squared: 0.632 F-statistic:
4.65 on 8 and 9 degrees of freedom,       p-value: 0.01694
```

Die bei der Funktion `summary` angegebenen Schätzungen sind Schätzungen für die Parameter ν, α_i, β_j, γ_{ij}, $i, j = 2, 3$ der Modellierung

$$\begin{aligned} Y_{11k} &= \nu + \epsilon_{11k}, \\ Y_{1jk} &= \nu + \beta_j + \epsilon_{1jk} \text{ für } j = 2, 3, \\ Y_{i1k} &= \nu + \alpha_i + \epsilon_{i1k} \text{ für } i = 2, 3, \\ Y_{ijk} &= \nu + \alpha_i + \beta_j + \gamma_{ij} + \epsilon_{ijk} \text{ für } i, j = 2, 3. \end{aligned}$$

11 Statistische Analyse für multivariate Daten

In der Library `mva` (in `R2.0.1` ist diese Library in `stats` integriert worden, die automatisch geladen wird) befinden sich Funktionen für die multivariate Datenanalyse. Die Hauptkomponenten-Analyse (Principal Component Analysis) erhält man mit den Funktionen `princomp` und `prcomp`. Für die Iris setosa-Daten erhält man folgende Hauptkomponenten-Analyse:

```
> library(mva)
> prcomp(iris.df[1:50,1:4])
 Standard deviations:
 [1] 0.48626710 0.19214248 0.16369606 0.09504347

 Rotation:
            PC1        PC2        PC3         PC4
 SepL -0.6690784 -0.5978840 -0.4399628 -0.03607712
 SepW -0.7341478  0.6206734  0.2746075 -0.01955027
 PetL -0.0965439 -0.4900556  0.8324495 -0.23990129
 PetW -0.0635636 -0.1309379  0.1950675  0.96992969
```

Dabei enthält der Vektor `Standard deviations` die Wurzeln aus den Eigenwerten der Kovarianzmatrix.

Die kanonische Korrelation wird durch die Funktion `cancor` gegeben:

```
 > iris.cancor<-cancor(iris.df[1:50,1:2],iris.df[1:50,3:4])
 > iris.cancor
 $cor [1] 0.33432931 0.06043577

 $xcoef
            [,1]       [,2]
 SepL 0.39574604 -0.4577216
 SepW 0.01182770  0.5625376

 $ycoef
           [,1]       [,2]
 PetL 0.4809926 -0.7272854
 PetW 0.8678066  1.1452230
```

```
$xcenter
 SepL  SepW
5.006 3.428

$ycenter
 PetL  PetW
1.462 0.246

> cor(as.matrix(iris.df[1:50,1:2])%*%iris.cancor$xcoef[,1],
+ as.matrix(iris.df[1:50,3:4])%*%iris.cancor$ycoef[,1])
          [,1]
[1,] 0.3343293
```

Die Library `MASS` enthält die lineare Diskriminanz-Analyse. Auf die Iris-Daten kann diese wie folgt angewendet werden:

```
> library(MASS)
> iris.lda<-lda(art~.,iris.df)
> iris.lda
Call: lda.formula(art ~ ., data = iris.df)

Prior probabilities of groups:
      set      vers      virg
0.3333333 0.3333333 0.3333333

Group means:
      SepL  SepW  PetL  PetW
set  5.006 3.428 1.462 0.246
vers 5.936 2.770 4.260 1.326
virg 6.588 2.974 5.552 2.026

 Coefficients of linear discriminants:
            LD1         LD2
SepL -0.8293776  0.02410215
SepW -1.5344731  2.16452123
PetL  2.2012117 -0.93192121
PetW  2.8104603  2.83918785

Proportion of trace:
   LD1    LD2
0.9912 0.0088

> iris.lda$scaling
            LD1         LD2
SepL -0.8293776  0.02410215
SepW -1.5344731  2.16452123
PetL  2.2012117 -0.93192121
```

```
PetW  2.8104603  2.83918785
> iris.lda.scaled<-as.matrix(iris.df[,1:4])%*%iris.lda$scaling

> plot(iris.lda.scaled[1:50,],cex=1.2,xlim=c(-10,12),ylim=c(4,10))
> points(iris.lda.scaled[51:100,],pch=2)
> points(iris.lda.scaled[101:150,],pch=0)
```

12 Allgemeines über Graphiken

Mit der Anweisung

```
> postscript(file=''myfile.ps'')
```

wird die anschließend erzeugte Graphik als Postscript-Datei in der Datei *myfile.ps* gespeichert. Um das Speichern in der Datei *myfile.ps* zu beenden, gibt man

```
> dev.off()
```

ein.

In R wird zwischen *High-Level*-Graphikfunktionen und *Low-Level*-Graphikfunktionen unterschieden. Durch eine High-Level-Graphikfunktion wird immer eine neue Graphik im Graphikfenster erzeugt, während die Low-Level-Graphikfunktionen Ergänzungen an der vorhandenen Graphik vornehmen.

• High-level graphics functions:

Univariate Data

barplot	Creates a simple bar plot.
boxplot	Creates side-by-side box plots.
hist	Creates a histogram.

Bivariate Data

plot	Creates a scatterplot.
barplot	Creates a simple bar plot.
boxplot	Creates side-by-side box plots.
qqnorm	Plot quantile-quantile plot for one sample against standard normal.
qqplot	Plot quantile-quantile plot for two samples.

Three-Dimensional Plots

contour	Creates contour plot.
persp	Creates perspective (mesh) plot.
image	Creates an image plot.

Multivariate Data

matplot	Create matplot.
pairs	Pairwise scatterplots.
stars	Starplots.

• Low-level graphics functions:

title	Adds titles, subtitles and axis titles

par	Sets graphic parameters
abline	Adds a line
lines	Adds a scatterplot, where the points are connnected with lines
points	Adds the points of a scatterplot
text	Adds text
legend	Adds a legend

Die Graphiken können durch Graphik-Parameter beeinflusst werden. Es gibt vier Typen von Graphik-Parametern:

a) *High-level-Graphik-Parameter*: Kontrollieren das Aussehen des Druckbereiches und können nur als Argumente in High-Level-Graphikfunktionen gesetzt werden.

b) *Layout-Graphik-Parameter*: Kontrollieren den Ausdruck einer Seite und können nur mit der Funktion **par** gesetzt werden.

c) *Allgemeine (general) Graphik-Parameter*: Können sowohl in einer High-Level-Graphikfunktion als auch mit der Funktion **par** gesetzt werden. Wenn sie mit der Funktion **par** gesetzt werden, gelten sie, bis sie wieder abgeändert werden.

d) *Informations-Graphik-Parameter*: Können nicht vom Anwender verändert werden. Sie können aber mit der Funktion **par** abgefragt werden.

Die folgende Beschreibung der Graphik-Parameter ist von der Statistik-Software `S-Plus` übernommen worden, die fast ganz mit R übereinstimmt. Trotzdem könnte es sein, dass einige der Graphik-Parameter in R nicht vorhanden sind oder eine andere Bedeutung haben.

name	type	mode	description	example
For Multiple Figures				
fig	layout	numeric	figure location	c(0,.5,.3,1)
fin	layout	numeric	figure size	c(3.5,4)
fty	layout	character	figure type	"r"
mfg	layout	integer	location in figure array	c(1,1,2,3)
mfcol	layout	integer	figure array size	c(2,3)
mfrow	layout	integer	figure array size	c(3,2)
Text				
adj	general	numeric	text justification	.5
cex	general	numeric	height of font	1.5
crt	general	numeric	character rotation	90
csi	general	numeric	height of font	.11
main	title	character	main title	"Y versus X"
srt	general	numeric	string rotation	90
sub	title	character	subtitle	"Y versus X"
xlab	title	character	axis title	"X (in dollars)"
ylab	title	character	axis title	"y (in dollars)"
Symbols				

lty	general	integer	line type	2
lwd	general	numeric	line width	3
mkh	general	numeric	symbol height	.2
pch	general	character integer	plot symbol	“*“ or 4
smo	general	integer	curve smoothness	1
type	general	character	plot type	“p“, “l“, “b“, “o“, “h“
xpd	general	logical	symbols in margins?	T
Axes				
axes	high-level	logical	plot axes?	F
bty	general	integer	box type	4
exp	general	numeric	format for exponential numbers	1
lab	general	integer	tick marks/labels	c(3,7,4)
las	general	integer	label orientation	1
log	high-level	character	logarithmic axes	“xy“
mgp	general	numeric	axis locations	c(3,1,0)
tck	general	numeric	tick mark length	1
xaxs yaxs	general	character	style of limits	“i“
xaxt yaxt	general	character	axis type	“n“
Margins				
mai	layout	numeric	margin size	c(.4,.5,.6,.2)
mar	layout	numeric	margin size	c(3,4,5,1)
mex	layout	numeric	margin units	.5
oma	layout	numeric	outer margin size	c(0,0,5,0)
omd	layout	numeric	outer margin size	c(0,.95,0,1)
omi	layout	numeric	outer margin size	c(0,0,.5,0)
Plot Area				
pin	layout	numeric	plot area	c(3.5,4)
plt	layout	numeric	plot area	c(.05,.95,.1,.9)
pty	layout	character	plot type	“s“
uin	information	numeric	inches per usr unit	c(.73,.05)
usr	layout	numeric	limits in plot area	c(76,87,3,8)
xlim ylim	high-level	numeric	limits of axis	c(3,8)
Miscellaneous				
col	general	integer	color	2
err	general	integer	print warnings?	-1
new	layout	logical	is figure black?	T