

---

# R-Hinweise zu Vorlesungen über Statistik

Werner Stahel, Seminar für Statistik, April 2003 / 2005

## 1 Allgemeines

### a Diese Hinweise

... ersetzen keine Einführung. Kurze Zusammenstellung des Wichtigsten, das Sie brauchen, für mathematisch orientierte Interessierte. Es sollte reichen, dass Sie die Übungen lösen können, und Sie zum Lesen einer Einführung animieren. Eine längere Einführung, die für Interessierte aus Anwendungsgebieten gedacht ist, finden Sie anderswo auf der meiner Homepage ([stat.ethz.ch/~stahel](http://stat.ethz.ch/~stahel)).

### b Aufruf, beenden

Wie das System aufgerufen wird, steht in den R-tutorials, verfügbar von <http://stat.ethz.ch/~stahel/courses/multivariate/>.

Man steigt aus, indem man `q()` eintippt.

### c Funktionen erzeugen Objekte

Grundlegend: Sie tippen etwas ein, was im System einen Funktionsaufruf erzeugt, z.B. `c(2,-1,9,0.4,100)`. Die Funktion

- erzeugt ein „Objekt“,
- erzeugt grafischen Output, der im Grafikfenster gezeigt wird, oder
- schreibt numerischen Output im Fenster, wo das System läuft.

### d Zuweisung

Sie können das resultierende Objekt einem Namen zuweisen mit dem Zuweisungspfeil `<-`, der sich leider aus zwei Zeichen `<` und `-` zusammensetzt. (Im ESS-Modus von emacs erzeugt der underline `_` den Zuweisungspfeil.)

```
t.v <- c(2,-1,9,0.4,100)
```

### e print

Wenn Sie keinen Zuweisungspfeil verwenden, wird die Funktion `print` auf das erzeugte Objekt angewendet und Sie sehen das Resultat auf dem Bildschirm.

```
t.v : es erscheint [1] 2.0 -1.0 9.0 0.4 100.0
```

( [1] am Anfang sagt, dass die erste Zahl das erste Element des Vektors `t.v` ist. Das wird nützlich, wenn das Resultat auf dem Bildschirm mehr als eine Zeile braucht.)

### f Hilfe

Zu allen Funktionen erhalten Sie eine Beschreibung, wenn Sie `?functionname` tippen. Beispiel: `?c`

Wenn Sie eine Funktion suchen, tippen Sie `help.search("name")` ein. `help.start()` führt zu einem web-basierten Hilfesystem.

### g Fehlermeldungen

Fehlermeldungen sind leider nicht immer so verständlich, wie man sie gerne hätte. Die S-Sprache ist so flexibel, dass Sie einiges schreiben können, was das System so halb versteht und deshalb vielleicht den Fehler am falschen Ort sucht.

### h Denken Sie an Matrizen, nicht an Zahlen!

Wenn Sie eine klassische Programmiersprache gelernt haben, dann denken Sie sofort an Deklarationen von Typen und Dimensionen. Das können Sie vergessen. Zudem haben Sie die Gewohnheit, Schleifen zu programmieren. Schleifen, die über Elemente von Arrays laufen, sind grundsätzlich der falsche Ansatz in einer Matrix-orientierten Sprache. Überlegen Sie es sich fünfmal, bevor Sie

eine Schleife verwenden! (Für iterative Verfahren braucht es Schleifen.)

### i Umgebung

Das System wartet auf eine Eingabe auf der Kommando-Zeile. Im Normalfall sollten Sie aber in einem anderen Fenster (Skript-Fenster) die Kommandos tippen und mit Hilfe eines intelligenten Texteditors wie emacs mit ESS oder wineDT ins R-Fenster „schicken“. Siehe R-Tutorials.

## 2 Objekte

### a Vektoren

`t.v <- c(2,-1,9,0.4,100)` erzeugt den Vektor  $[2, -1, 9, 0.4, 100]$  und speichert ihn unter dem Namen `t.v`

`rep(0,10)` : Vektor von 10 Nullen.

`rep(t.v,3)` : Wiederhole `t.v` 3 mal.

`t.vs <- seq(0,2,0.5)` : Folge von Zahlen von 0 bis 2 mit Inkrement 0.5, also  $[0, 0.5, 1, 1.5, 2]$

`t.i <- 1:7` : Abkürzung für  $[1, 2, \dots, 7]$  (das wird mit einem eigenen Spezialzeichen abgekürzt, weil es so oft gebraucht wird).

`rep` und `seq` können einiges mehr, siehe `?rep`, `?seq`

### b Matrizen

`t.m <- matrix(t.v,5,3)` : Matrix der Dimension  $5 \times 3$ , spaltenweise gefüllt mit dem Inhalt von `t.v`.

`t.m <- cbind(1:5,t.v,c(0,0,2,2,3))` : spaltenweise zusammenfügen,

`rbind` : zeilenweise zusammenfügen.

### c Listen

Listen fassen Objekte zu einem neuen Objekt zusammen. Sie entstehen üblicherweise als Resultate von Funktionen, die mehrere Teilresultate erzeugen – also von allen Funktionen, die ein statistisches Modell anpassen.

`list(t.v,t.m)` : erzeugt eine Liste.

### d mode

Die Objekte, die wir bisher erzeugt haben, enthalten nur Zahlen. Daneben gibt es andere Typen von „Inhalt“. `mode(t.m)` zeigt diesen „Typ“, nämlich `numeric`, `character`, `logical`, `complex` oder auch den Typ des Objekts, `list`, `function`. Es gibt noch einige weitere Typen ...

### e Umwandlung

Man kann einige Typen sinnvoll in einander umwandeln:

`numeric` in `character` (und manchmal umgekehrt),

`logical` in `numeric` (0 für FALSE, 1 für TRUE).

`as.character(1:3)` : ergibt `[1] "1" "2" "3"`.

`as.logical(c(0,1,-1,3,0.5))` : Was = 0 ist, gibt FALSE, alles andere TRUE.

### f Klassen von Objekten

Diese grundlegenden Typen werden ergänzt durch die Möglichkeit, beliebige weitere Typen zu definieren. Man spricht von Klassen von Objekten. Sie ermöglichen eine bestimmte Art von „objektorientiertem Programmieren“. In der neuesten Version von S hat jedes Objekt eine Klasse. Auskunft erteilt `class(t.m)`.

### g Bezeichnungen

Namen von Objekten sind alphanumerisch. Sie beginnen mit einem Buchstaben und dürfen ausser Buchstaben und Ziffern nur den Punkt enthalten. Grosse und kleine Buchstaben sind verschieden.

Es gibt natürlich viele Namen, die bereits vom System benützt werden, vor allem die Namen der

vorhandenen Funktionen. Man kann diese Namen trotzdem verwenden und wird nicht gewarnt, dass damit etwas „verdeckt“ wird.

`pi <- 2` ist also nicht gerade harmlos...

Immerhin: Wenn die Funktion `c()` gebraucht wird, findet das System sie, auch wenn Sie ein Objekt (das keine Funktion ist) unter dem Namen `c` gespeichert haben.

Ich persönlich halte Ordnung, indem die von mir benützten Namen speziell aussehen: Sie beginnen mit einem Buchstaben, gefolgt von einem Punkt, gefolgt von weiteren Buchstaben und Ziffern. Die Buchstaben vor dem Punkt werden dazu benützt, die Art der Objekte zu unterscheiden. Es ist `t.d` ein nur temporär benütztes Objekt, z.B. der momentan bearbeitete Datensatz.

`d.` : Datensätze

`t.` : temporäre Objekte

`f.` : eigene Funktionen mit numerischen Resultaten

`g.` : eigene Grafik-Funktionen

### 3 Funktionen

#### a Argumente

Argumente werden durch `,` getrennt. Identifikation über Position (erstes, zweites, ... Argument) oder über Namen der Argumente:

```
matrix(t.v, nrow=5, ncol=3)
```

Die „höheren“ Funktionen haben viele „freiwillige“ Argumente, für die die Funktion defaults (Weglasswerte) kennt.

#### b Funktionen sind oft sehr flexibel!

Viele Standard-Funktionen erzeugen recht verschiedenartige Ergebnisse, je nach der Art des ersten (und weiterer) Argumente. Beispiel:

`diag(1:3)` : erzeugt eine Diagonalmatrix mit den Diagonal-Elementen 1, 2, 3.

`diag(t.m)` : gibt die Diagonale einer Matrix zurück.

#### c Generic functions

Diese Flexibilität wird mit dem Konzept der generic functions klar ausgewiesen. Es gibt zwei Generationen von Elementen des objekt-orientierten Programmierens in S. In der älteren, genannt S3, sieht sich eine generic function die „class“ des ersten Objektes an und ruft dann die entsprechende „method“ (eine gewöhnliche Funktion) auf.

Es gibt drei grundlegende generische Funktionen, die für alle Klassen von Objekten eine geeignete Methode kennen sollten:

`print` : numerische Ausgabe auf den Bildschirm. Achtung: Einige Objekte enthalten viel mehr Information, als man sieht, wenn man sie printet (also ohne Zuweisung eintippt).

`summary` : numerische Ausgabe einer sinnvollen Zusammenfassung.

`plot` : wenn es sinnvoll ist, ein Objekt grafisch darzustellen.

Die neue Generation, S4, kann auf die Klassen aller Argumente reagieren. S4-Klassen sind rigider und damit sauberer definiert als S3-Klassen, aber deshalb auch umständlicher zu definieren.

#### d Klammern nicht vergessen!

Eine Funktion wird nur ausgeführt, wenn der Name von `()` gefolgt ist. Eingabe des Funktionsnamens ohne Klammern zeigt die Definition der Funktion. Deshalb kann man die Session nicht mit `q` abschliessen, sondern nur mit `q()`.

#### e Optionen

Was eine Funktion tut, wird manchmal nicht nur von den Argumenten bestimmt, sondern auch von den zwei Objekten, die über die Funktionen `options` (beeinflusst numerische Resultate) und `par` (beeinflusst grafischen Output) abgefragt und geändert werden.

- f Funktionen kann man selber schreiben, siehe später.

## 4 Operationen

### a Arithmetische Operationen

Die arithmetischen Operatoren `+` `-` `*` `/` `^` werden auf zwei Vektoren oder Matrizen elementweise angewandt.

`c(2, 4)*c(3, -1)` liefert `[1] 6 -4`.

### b Recycling

Hat das eine Argument mehr Elemente als das andere, dann werden die Elemente des kürzeren zyklisch wiederholt.

`c(2, 4)*3` liefert `[1] 6 12`

### c Uneigentliche Werte

S kennt auch die uneigentlichen Werte

`Inf` und `-Inf` : unendlich

`NaN` : unbestimmt

`NA` : fehlend oder undefiniert

### d Mathematische Funktionen

Z.B. `log10(t.v)`, wird ebenfalls elementweise ausgeführt. Negative Elemente von `t.v` ergeben `NA`. Die üblichen Funktionen sind vorhanden: `log`, `exp`, `sin`, `cos`, `tan`, `ctg`, `asin`, ... ,

`min(t.v, 500, 1:10)` liefert das Minimum über alle Elemente aller Argumente.

`which.min(t.v)` : Index des (ersten) Elements, das gleich dem Minimum ist.

`pmin` elementweises Minimieren von Vektoren.

### e Matrix-Multiplikation

`t.v %*% t.m`

S unterscheidet nicht zwischen Zeilen- und Spaltenvektoren. Es probiert mit beiden Versionen, ob die Operation geht. Wenn man ausdrücklich einen Spalten- oder Zeilenvektor will, muss man dies als einspaltige oder einzeilige Matrix speichern.

### f Logische Operationen

Die logischen Werte heissen `TRUE` und `FALSE`. Für Schreibfaule sind die Objekte `T` und `F` gespeichert. Durch Überschreiben dieser Objekte (`T <- FALSE`) kann man beliebigen Unsinn produzieren.

Die logischen Operatoren heissen

`!` : not

`&` : and

`|` : or

Logische Objekte sind natürlich wichtig für `if(logical)` - Konstruktionen.

### g Vergleiche

`==` , `!=` , `>` , `<` , `>=` , `<=`

Aufgepasst mit `t.v<-3!` Das überschreibt `t.v`. Den Vergleich mit `-3` erhält man mit einem Abstand: `t.v< -3`.

### h Umwandlungen

Die arithmetischen Operationen sind nur für `numeric` Objekte definiert. Logische Objekte werden in numerische verwandelt. Generell versucht S, durch Umwandlung der Objekt-, „Typen“ wenn möglich ein Resultat zu produzieren.

**i Priorität**

Die Priorität ist generell wie folgt geregelt:

Funktionsauswertung vor  $\wedge$  :  $*$ / $+$ - vor Vergleichen vor logischen Operationen.

**j Zusammenfassende Funktionen**

`sum, mean, median, var, ...`

**k apply**

Eine Funktion, die als Argument einen Vektor hat, will man oft auf alle Spalten (Zeilen) einer Matrix anwenden. Hier hilft `apply`, siehe `?apply`,

```
apply(t.m,2,mean)
```

Schreiben Sie keine Schleife!

(Für Listen gibt es die Funktionen `lapply` und `sapply`.)

## 5 Namen und Auswahl von Elementen

**a Elemente von Vektoren und Matrizen**

```
t.v[2:3], t.m[c(3,5),3],
```

`t.m[,3]` : ganze dritte Spalte

**b names**

Man kann den Elementen eines Vektors Namen geben:

```
names(t.v) <- c("Hans","Fritz","Elsa","Trudi","Olga")
```

und dann mit `t.v["Elsa"]` auswählen.

(Die Syntax ist überraschend, sie sieht nach Zuweisung an den Wert einer Funktion aus!)

`t.vn <- names(t.v)` gibt die Namen wieder zurück.

Analog für Matrizen: `dimnames(t.m) <- list(t.vn, c("V1","V2","V3"))`

Auswahl: `t.m[c("Elsa","Fritz"),"V1"]` oder auch gemischt: `t.m["Elsa",1]`.

**c Direkte Zuweisung von Namen**

```
t.v <- c(Hans=2,Fritz=-1,Elsa=9,Trudi=0.4,Olga=100)
```

```
t.m <- cbind(V1=1:5, V2=t.v)
```

**d paste**

Für die Erzeugung von Namen (und character Vektoren für anderes) ist die Funktion `paste` sehr nützlich:

```
paste("V", 1:3, sep=".")
```

erzeugt `[1] "V.1" "V.2" "V.3"`

Das Argument `sep` muss immer mit seinem Namen `sep=` angegeben werden, da alle Argumente ohne Namen zusammengebunden werden. Default ist ein Leerzeichen " ", also ergibt `paste("AB","CD","ef")` `[1] "AB CD ef"`.

## 6 Daten

**a data.frame**

Statistische Daten enthalten oft numerische Variable und „Faktoren“, die als character Variable gespeichert werden müssen. Zusammen werden diese Variablen in einem Objekt der Klasse `data.frame` gespeichert.

**b Daten einlesen**

von einem ASCII-file: `d.abst <-`

```
read.table("http://stat.ethz.ch/stahel/data/abst.dat",header=TRUE)
```

c **System-Datensätze**

```
data(iris)
```

d **Fehlende Werte**

Wie bereits gesagt, können Werte, die nicht beobachtet werden konnten, durch den uneigentlichen Wert `NA` gekennzeichnet werden. Statistische Funktionen müssen mit solchen Werten in geeigneter Weise umgehen können.

e **Umwandlung**

```
t.d <- data.frame(t.m)
t.x <- matrix(iris[,1:4])
```

f **Auswahl von Elementen**

... wie bei Matrizen. `Data.frames` haben immer Namen für die Zeilen (Beobachtungen) und für die Spalten (Variablen).

## 7 Grundlegendes zur Grafik

a **plot**

```
plot(d.abst[, "a"], d.abst[, "c"])
```

Die Funktion `plot` hat viele Argumente für Beschriftungen, Datenbereiche etc. `?plot` !

b **Treiber**

Damit grafische Funktionen ihre Aufgabe erledigen können, muss ein Grafikfenster geöffnet werden. Das geschieht in `R` automatisch. In `S-Plus` muss (?) zuerst die geeignete Treiber-Funktion, z.B. `motif()` aufgerufen werden.

`dev.print` schickt die momentan sichtbare Grafik auf den Drucker.

Man kann die Grafik-Ausgabe auch auf ein Postscript-File oder PDF, ..., schreiben lassen, indem man die entsprechende Treiberfunktion aufruft. Es gibt die `SfS`- (Seminar für Statistik-) Funktionen `ps.do` und `ps.latex`; Abschluss durch `ps.end`.

`dev.list()` zeigt, welche Grafik-Fenster offen sind.

c **high und low level**

Neben `plot` gibt es viele andere Funktionen, die Grafiken erzeugen. Man kann mit "low level" grafischen Funktionen bereits erzeugte Grafiken ergänzen.

`text(3,5, "Fritz")` schreibt diesen Namen an die entsprechende Stelle im `plot`.

d **par**

Die Grafik wird mitbestimmt durch „grafische Parameter“, die in einem speziellen Objekt gespeichert sind, das über die Funktion `par` abgefragt und geändert wird.

`par(mar=c(3,3,4,1))` setzt die Breite des Randes (unten und links auf 3 Linien, oben auf 4, rechts auf 1).

Es gibt grafische Parameter, die nur durch `par` gesetzt werden können, solche, die nur durch Argumente der "high level" grafischen Funktionen gewählt werden können, und andere, die auf beide Arten wählbar sind.

e **Interaktive und dynamische Grafik**

`identify` : Punkte auf einer Grafik identifizieren.

`brush` : Interaktiv Punkte einfärben und 3d-Rotation

Die Auswahl in Grundpaket ist beschränkt, da `S` möglichst Plattform-unabhängig sein will, und weil die Grafik archaisch ist: Die Grafik ist der älteste Teil des Systems, mehr als 30 Jahre alt!

## 8 Verwaltung des Workspace

### a Global environment, Packages

S sucht die Objekte, die gebraucht werden, in so genannten “environments” (in R) oder “frames” (in S).

Wenn ein Objekt von der Kommando-Zeile aus einem Namen zugewiesen wird, dann wird es im “global environment” gespeichert. Die Suche nach einem Objekt beginnt ebenfalls dort und geht in den angehängten “packages” (und angehängten data.frames oder Objektlisten) weiter. Immer angehängt ist das package `package:base` und einige mehr.

`search()` zeigt die angehängten environments in der Such-Reihenfolge.

### b Lokale environments

Der Aufruf einer Funktion erzeugt ein environment mit lokalen Variablen. Während der Ausführung sucht das System zunächst in diesem lokalen environment, dann aber auch in den vorher erwähnten. Also Achtung: Funktionen können auch auf Objekte zugreifen, die ihnen nicht als Argumente übergeben werden! (Anständig geschriebene Funktionen tun das nicht, ausser dass sie `options` und `par` benützen.)

Geschachtelte Aufrufe erzeugen eine Hierarchie solcher temporärer environments. Darüber müssen Sie sich erst informieren, wenn Sie selber Funktionen schreiben. Achtung: Hier gibt es tief liegende Unterschiede zwischen R und S-Plus!

### c Libraries

Es gibt zurzeit etwa 350 packages, die am Seminar für Statistik verfügbar sind und die Sie von der R website herunterladen können.

`library()` (ohne Argument) : Verzeichnis aller auf Ihrem System installierten packages

`help(package=fda)` Informationen über ein Package.

`library(fda)` : ein package wird für die momentane Session verfügbar machen.

### d .First

Wenn man eine Funktion `.First` speichert, wird sie am Anfang jeder Session ausgeführt.

`.First <- function() { options(digits=3) }` : Setzt für jede Session die genannte Option.

### e objects

`objects()` liefert die Namen aller Objekte im global environment, `objects(5)` im angehängten package mit Nummer 5.

`objects(pattern="^t\\.*)"` : Namen der Objekte, die mit `t.` beginnen.

### f str

`str(object)` liefert Informationen darüber, welcher Art object ist.

### g rm

`rm(t.d)` löscht das Objekt `t.d`

`rm(list=objects(pattern="t\\.*))` löscht alle Objekte, die mit `t.` beginnen.

## 9 Verteilungen und Zufallszahlen

### a Normalverteilung

`dnorm(7,5,2)` : Dichte der Normalverteilung für  $x = 7$ ,  $\mu = 5$ ,  $\sigma = 2$ .

`pnorm(7,5,2)` :  $P\{X \leq 7\}$  für  $X \sim \mathcal{N}(5, 2^2)$ .

`qnorm(c(0.025, 0.5, 0.975))` : Quantile der Standard-Normalverteilung.

`rnorm(20)` : Erzeuge 20 standard-normalverteilte Zufallszahlen

## b Andere Verteilungen

Es gibt die entsprechenden Funktionen für alle bekannten Verteilungen.

`d... (x)` : Dichte respektive, bei diskreten Verteilungen, Wahrscheinlichkeit für den Wert  $x$ .

`p... (x)` : Kumulative Verteilungsfunktion, nützlich zur Berechnung von p-Werten: `1-pchisq(3.6,2)`.

`q... (x)` : Quantile (inverse kumulative Verteilungsfunktion), nützlich zur Berechnung von kritischen Werten für Tests: `qchisq(0.95,2)`

`r... (n)` :  $n$  Zufallszahlen gemäss der Verteilung (siehe unten).

Es gibt folgende Verteilungen:

beta, binom, cauchy, chisq, exp, f, gamma, geom, hyper, lnorm, logis, nbinom, norm, pois, t, unif, weibull, wilcox

und weitere in verschiedenen speziellen packages.

## c Zufallszahlen

`runif(5)` : erzeugt 5 Zufallszahlen gemäss der uniformen Verteilung in  $(0,1)$ .

`rnorm(20, mean=5, sd=2)` : 20 normalverteilte Zufallszahlen mit Erwartungswert 5 und Standardabweichung 2.

`set.seed(28)` : Startwert für (Pseudo-) Zufallszahlen-Generator festlegen. Führt zu reproduzierbaren Ergebnissen.

`sample(8)` : Zufällige Reihenfolge der Zahlen 1 bis 8

`sample(3, size=10, replace=TRUE)` :

# 10 Grafische Funktionen

## a Wichtige grafische (high level) Funktionen

Eindimensional: `hist`, `barplot` : Histogramm und Stabdiagramm

`qqnorm`, `qqplot` : QQ-Diagramme zur Beurteilung von Verteilungen

Zweidimensional: `plot` : Streudiagramm

`sunflowerplot` : Streudiagramm mit multiplen Punkten

`image` : zweidimensionales Histogramm

Mehrdimensional: `persp` : Perspektivische Ansicht für 3-dimensionale Daten

`coplot` : Für 3- und 4-dimensionale Daten: Streudiagramme von  $Y$  gegen  $X$ , bedingt auf die Werte von  $A$  und evtl.  $B$

`matplot` : Mehrere Streudiagramme im gleichen Diagramm (z.B. mehrere  $y$ -Variable zu gleichen  $x$ -Werten)

`pairs` : Streudiagramm-Matrix

`brush` : Interaktive Version von `pairs`, mit 3d-Rotation

`symbols` : Streudiagramm mit Symbolen, die weitere Variable darstellen

## b Wichtigste allgemeine Elemente der Grafik-Funktionen

`type="p"` : Es sollen Punkte gezeichnet werden, `type="l"` : Linien, `type="b"` : beides

`xlim=`, `ylim=` : Begrenzung der Koordinaten. Sie werden nur exakt so verwendet, wenn man auch `xaxs="i"`, `yaxs="i"` setzt.

`xlab="x"`, `ylab="y"` : Achsen-Bezeichnung

`main="Titel"` : Titel

## c Wichtige grafische "low level"-Funktionen

Typischerweise schaltet man Teile der high level grafischen Funktion ab, indem man schreibt: `plot(..., type="n", axes=FALSE)` (keine Punkte resp. keine Achsen zeichnen). Dann ruft man die entsprechenden Funktionen auf:

Punkte:

`points` : Punkte zeichnen

`lines`, `arrows`, `segments` : Linien zeichnen  
`text` : Punkte mit Text beschriften

Achsen:

`axis` : Achsen-Beschriftung  
`box` : Umrandung  
`mtext`, `title` : Text im Rand der Figur  
`rug` : Daten am Rand zeichnen durch Striche

Zusatzelemente:

`abline` : Gerade zeichnen  
`polygon` : Polygon zeichnen und einfärben oder schraffieren  
`legend` : Legende

Punkte identifizieren:

`identify` : Index des Punktes, den man anklickt  
`locator` : Koordinaten

#### d Wichtigste grafische Parameter

`pch=` : plotting Symbol (numerisch) oder character  
`lty`, `lwd` : Linientyp und -dicke  
`cex`, `ps` : Schriftgrösse  
`mar=c(3,3,2,1)` : Breite des Randes (unten, links, oben, rechts)  
`col` : Farbe

#### e Mehrere Diagramme (frames) auf einer Seite

`par(mfrow=c(2,3))` : Aufteilung in 2 Zeilen und 3 Spalten, zeilenweise gefüllt.  
`par(mfcol=c(2,3))` : ... spaltenweise gefüllt  
`frame()` : Gehe zum nächsten Feld (das tun die high-level Funktionen automatisch)

Diese beiden Argumente von `par` bestimmen die alte, wenig flexible Art der Einteilung des „Bildschirms“. Die moderne Art, die alternativ verwendet werden soll (nicht mischen!):

`split.screen` : Flexible Aufteilung in Felder; `screen` : Wahl des Feldes

## 11 Funktionen schreiben

#### a Eine Funktion für eine einfache Simulation:

```
f.simmed <- function(n, mu=0, sig=1, nrep=1000, seed=1) {
  ## Simulation der Verteilung des Medians
  ## n      Stichprobenumfang
  ## mu     Erwartungswert
  ## sig    Standardabweichung
  ## nrep   Anzahl Replikate
  ## seed   Startwert fuer Zufallszahlen
  ## -----
  set.seed(seed)
  smed <- rep(NA,nrep)
  for (li in 1:nrep) {
    dat <- rnorm(n, mu, sig)
    smed[li] <- median(dat)
  }
  smed
}
```

---

**b Schlaufen**

Die Funktion enthält eine `for`-Schleife. Die Syntax lautet `for( Schlaufen-Variable in Werte-Vektor ) { statements }`. Es gibt auch `repeat` und `while`-Schlaufen. Solche Schlaufen treten typischerweise bei iterativen Berechnungen und komplizierten Simulationen auf.

(Die `for`-Schleife kann man im obigen Fall mit `apply` vermeiden.)

**c if ... else**

```
if (abs(eps)<0.001) r <- NA else r <- 1/eps
if (nrep <= 2) stop("Eine solche Simulation macht keinen Sinn")

  for (li in 1:niter) {
    ...
    if ( abs(diff) < tolerance ) break
  }
```