Einführung in R

Alexander Bauer

a.bauer@ma.tum.de

Lehrstuhl für Mathematische Statistik
Zentrum Mathematik
Technische Universität München

Ferienkurs, 6. - 9. April 2009

Organisatorisches

Webseite des Kurses:

`http://ferienkurse.ma.tum.de/Ferienkurse/WiSe0809/EinfR`

Zeiten Montag - Mittwoch:

- 9:00 - 10:30 Uhr und 11:00 - 12:30 Uhr (Theorie)
- 13:30 - 15:30 Uhr (Übung)

Zeiten Donnerstag:

- 11:30 - 12:30 Uhr (Theorie)
- 13:30 - 15:30 Uhr (Übung)

Räume:

- Theoriestunden: Hörsaal 1 (`00.02.001`)
- Übungsstunden: Rechnerhalle (`00.05.011`)

Was ist R?

- R ist eine statistische Programmierumgebung vergleichbar mit S und S-Plus
- R liefert eine Vielzahl von Grafiken zur explorativen Datenanalyse
- R ist frei erhältlich (GNU General Public License)
- R ist up-to-date und wird aktiv weiterentwickelt

Geschichte

1976 Entwicklung von S durch John Chambers in den Bell Labs von AT&T (heute Alcatel-Lucent)

1988 Entwicklung von S-Plus als kommerzielle Implementation von S durch Statistical Sciences (heute TIBCO)

1992 Start der Entwicklung einer freien Implementation von S zu Lehrzwecken durch Ross Ihaka und Robert Gentleman

1995 Veröffentlichung von R unter der GNU GPL

1997 Vereinigung des „R Development Core Team“

1998 Gründung des „Comprehensive R Archive Networks“

2009 Die aktuelle Version von R ist R-2.8.1, das Core Team umfasst 19 Mitglieder

Literatur

- P. Dalgaard. Introductory Statistics with R. Springer, Berlin, 2008.
- U. Ligges. Programmieren mit R. Springer, Berlin, 2008.
- W. Venables und B. Ripley. Modern Applied Statistics with S. Springer, Berlin, 2003.
- J. Verzani. Using R for Introductory Statistics. Chapman & Hall, London, 2004.

- R Manuals auf `http://www.r-project.org/`.

Software

R ist frei erhältlich auf `http://www.r-project.org/`.

Editoren:

- Tinn-R (Windows) – `http://www.sciviews.org/Tinn-R/`
- RWinEdt für WinEdt (Windows) – R Paket RWinEdt
- ESS für Emacs – `http://ess.r-project.org/`
- JGR – `http://jgr.markushelbig.org/JGR.html`
- Jeder beliebige Texteditor (evtl. weniger komfortabel)

In der Rechnerhalle wird R mit dem Befehl

`/usr/local/applic/bin/R`

gestartet.

Gliederung

1 Grundlagen

2 Datenstrukturen

3 Ein- und Ausgabe von Daten

4 Programmieren mit R

5 Grafik

6 Statistik mit R

7 Sweave

Programmstart

```
R version 2.8.1 (2008-12-22)
Copyright (C) 2008 The R Foundation for Statistical Computing ISBN
3-900051-07-0

R ist freie Software und kommt OHNE JEGLICHE GARANTIE.
Sie sind eingeladen, es unter bestimmten Bedingungen weiter zu
verbreiten. Tippen Sie 'license()' or 'licence()' für Details dazu.

R ist ein Gemeinschaftsprojekt mit vielen Beitragenden.
Tippen Sie 'contributors()' für mehr Information und 'citation()',
um zu erfahren, wie R oder R packages in Publikationen zitiert
werden können.

Tippen Sie 'demo()' für einige Demos, 'help()' für on-line Hilfe,
oder 'help.start()' für eine HTML Browserschnittstelle zur Hilfe.
Tippen Sie 'q()', um R zu verlassen.

>
```

Kommandozeile

- Die Kommunikation mit R erfolgt über die Kommandozeile
- Befehle werden am „Prompt“ > eingegeben

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

- Fortsetzungszeilen werden mit + gekennzeichnet

```
> 5 *
+ 2^3
[1] 40
```

- Zuweisungen erfolgen mit <-

```
> x <- 1
> x
[1] 1
> (y <- 2)
[1] 2
```

Kommandozeile

- Kommentare werden durch # gekennzeichnet

```
> sin(pi/2) # ein Kommentar
[1] 1
> 1 + 2i # komplexe Zahl
[1] 1+2i
> 1 / 0 # unendlich (Infinity)
[1] Inf
> 0 / 0 # nicht definiert (Not a Number)
[1] NaN
```

- Wie wir sehen ist R ein äußerst komfortabler Taschenrechner!

Arithmetische Operatoren und Funktionen

Operator / Funktion	Beschreibung
`^`	Potenz
`*`, `/`	Multiplikation, Division
`+`, `-`	Addition, Subtraktion
`%/%`	Ganzzahlige Division
`%%`	Modulo-Division
`abs()`	Betrag
`exp()`	Exponentialfunktion
`log()`	Logarithmus
`sqrt()`	Wurzel
`round()`	Runden
`sin()`, `cos()`, `tan()`	Trigonometrische Funktionen

Hilfesystem

- `help.start()`: Öffnet HTML-Hilfeseite im Webbrowser
- `help(log)`: Öffnet Hilfeseite der Funktion `log()`
- `?log`: Siehe `help(log)`
- `help.search("log")`: Durchsucht Hilfe nach dem Stichwort „`log`“
- `apropos("log")`: Sucht Objektnamen, die „`log`“ beinhalten
- `example(log)`: Gibt Beispiele zur Verwendung der Funktion `log()`

Workspace

Alle Objekte, die während einer R-Sitzung erzeugt werden, landen im Workspace.

- `ls()` zeigt die Objekte im aktuellen Workspace
- `rm(Objekt)` entfernt `Objekt` aus dem aktuellen Workspace

```
> (x <- 1)
[1] 1
> ls()
[1] "x"
> rm(x)
> x
Fehler: objekt "x" nicht gefunden
> ls()
character(0)
```

- `rm(list=ls())` löscht den gesamten Workspace

Workspace

- `getwd()` zeigt das aktuelle Arbeitsverzeichnis
- `setwd(Verzeichnis)` ändert dieses zu `Verzeichnis`

```
> setwd("C:/Programme/R")
> getwd()
[1] "C:/Programme/R"
```

- `q()` beendet R
- Vor dem Beenden kann der Workspace im Arbeitsverzeichnis gespeichert werden
- Die Datei `.Rhistory` enthält die zuletzt eingegebenen Befehle
- Die Datei `.RData` enthält den Workspace
- Der Workspace kann auch jederzeit mit `save.image()` gespeichert und mit `load(".RData")` geladen werden

1 Grundlagen

2 Datenstrukturen

3 Ein- und Ausgabe von Daten

4 Programmieren mit R

5 Grafik

6 Statistik mit R

7 Sweave

Atomare Datentypen

Datentyp	Beschreibung	Beispiel
`NULL`	leere Menge	`NULL`
`logic`	logische Werte	`TRUE`, `FALSE`
`numeric`	ganze und reelle Zahlen	`3.14`
`complex`	komplexe Zahlen	`1.3+2i`
`character`	Buchstaben und Zeichenfolgen	`"Hallo"`

Logische Werte

- Der Datentyp `logic` besitzt die Werte `TRUE` und `FALSE`

```
> 1 < 2
[1] TRUE
```

```
> 1==1 & 2>=3
[1] FALSE
```

- Mit diesen Werten kann auch gerechnet werden, wobei `TRUE` als `1` und `FALSE` als `0` interpretiert wird

```
> TRUE + FALSE
[1] 1
```

```
> (1==1) + (2>=3)
[1] 1
```

- Logische Werte spielen eine wichtige Rolle bei der Ablaufsteuerung und bei der Indizierung von Datenstrukturen!

Logische Operatoren und Verknüpfungen

Operator / Verknüpfung	Beschreibung
`!`	Negation
`&`	und
`|`	oder
`xor()`	entweder oder
`==`	gleich
`!=`	ungleich
`<, <=`	kleiner (gleich)
`>, >=`	größer (gleich)

Datenstrukturen

Datenstruktur	Beschreibung
`vector`	Vektor
`matrix`	Matrix
`array`	Array
`list`	Liste
`data frame`	Datensatz

Vektoren

- Fast alle Objekte in R werden intern durch Vektoren repräsentiert
- Skalare entsprechen Vektoren der Länge 1

```
> (x <- 1)
[1] 1
```

erster Eintrag des Vektors x

- Mit `c()` („concatenate“) können einfach Vektoren erzeugt werden

```
> c(1,2,3,4,5)
[1] 1 2 3 4 5
```

Index des ersten Eintrags in dieser Zeile

```
> c("Hallo","Welt")
[1] "Hallo" "Welt"
```

- Mit `length()` wird die Länge eines Vektors abgefragt

```
> x <- c(1,2,3,4,5)
> length(x)
[1] 5
```

Folgen

- Ganzzahlige Folgen mit Abstand 1 können durch `:` erzeugt werden

```
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
> 5:-5
[1] 5 4 3 2 1 0 -1 -2 -3 -4 -5
```

- Für allgemeinere Folgen steht die Funktion `seq()` zur Verfügung

```
> seq(from=1, to=10, by=2)
[1] 1 3 5 7 9
> seq(1, 10, 2)
[1] 1 3 5 7 9
> seq(1.5, 4.5, length=3)
[1] 1.5 3.0 4.5
```

Wiederholungen

- Für Wiederholungen kann `rep()` verwendet werden

```
> rep(c(1,2), times=3)
[1] 1 2 1 2 1 2

> rep(c(1,2), each=3)
[1] 1 1 1 2 2 2

> rep(c(1,2), length.out=3)
[1] 1 2 1

> rep(c(1,2,3), times=2, length.out=15, each=3)
[1] 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2
```

Funktionsaufrufe in R

- Ein Funktionsaufruf hat die Form
  `funktionsname(argument1=wert1, argument2=wert2, ...)`
- Eine Funktion muss keine Argumente besitzen (z.B. `getwd()`)
- Manche Argumente besitzen Voreinstellungen („`defaults`“), die nicht angegeben werden müssen (z.B. `each=1` in `rep()`)
- Unbenannte Argumente werden gemäß der formalen Definition der Funktion interpretiert (siehe entsprechende Hilfeseite)
- Bei `rep()` wäre die Reihenfolge `x`, `times`, `length.out`, `each`

```
> rep(c(1,2,3), 2, 15, 3)
[1] 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2
```

- Diese Art des Aufrufs ist bei mehreren Argumenten äußerst unübersichtlich und fehleranfällig!

Benannte Vektorelemente

- Die Einträge eines Vektors können mit Bezeichnungen versehen werden

```
> (x <- c(Montag=1, Dienstag=2, Mittwoch=3))
 Montag   Dienstag   Mittwoch
 1        2          3
```

- Mit `names()` werden diese Bezeichnungen abgefragt und geändert

```
> names(x)
[1] "Montag" "Dienstag" "Mittwoch"
> names(x) <- c("Freitag", "Samstag", "Sonntag")
> x
 Freitag   Samstag   Sonntag
 1         2         3
```

Indizierung von Vektorelementen

- Mit `x[i]` wird auf den `i`-ten Eintrag des Vektors `x` zugegriffen

```
> x <- 11:15
> x[5]
[1] 15
> x[length(x)]
[1] 15
> x[20] # existiert nicht (Not Available)
[1] NA
```

- Ein Minuszeichen schließt den entsprechenden Eintrag aus

```
> x[-3]
[1] 11 12 14 15
```

- Es können mehrere Elemente gleichzeitig indiziert werden

```
> x[c(1,3,5)]
[1] 11 13 15
```

Indizierung von Vektorelementen

- Logische Indizierung ist ebenfalls möglich

```
> x[c(TRUE, TRUE, TRUE, FALSE, FALSE)]
[1] 11 12 13
> x[x<=13]
[1] 11 12 13
```

- Ist der logische Vektor kürzer als `length(x)`, so wird er wiederholt

```
> x[c(TRUE, FALSE)]
[1] 11 13 15
```

- Auf benannte Vektoren kann man über deren Namen zugreifen

```
> names(x) <- c("Mo", "Di", "Mi", "Do", "Fr")
> x[c("Di", "Mi", "Do")]
Di  Mi  Do
 2   3   4
```

Rechnen mit Vektoren

- Viele mathematische Operatoren sind komponentenweise definiert

```
> x <- 1:6
> x + 2
[1] 3 4 5 6 7 8
> x
[1] 1 2 3 4 5 6
> (y <- rep(10, 6))
[1] 10 10 10 10 10 10
> x * y
[1] 10 20 30 40 50 60
> y / c(5,10)
[1] 2 1 2 1 2 1
```

Rechnen mit Vektoren

- Zahlreiche Funktionen ebenfalls

```
> exp(1:3)
[1] 2.718282 7.389056 20.085537
> options(digits=4)
> exp(1:3)
[1] 2.718 7.389 20.085
```

- Vektorarithmetik spielt eine zentrale Rolle in R!

```
> x <- 1:10
> max(x)
[1] 10
> sum(x)
[1] 55
> y <- seq(1, 20, by=2)
> intersect(x,y)
[1] 1 3 5 7 9
```

Rechnen mit Vektoren

Funktion	Beschreibung
`sum()`	Summe der Elemente
`prod()`	Produkt der Elemente
`max(), min()`	Maximum, Minimum
`union()`	Vereinigung der Elemente zweier Vektoren
`intersect()`	Schnitt der Elemente zweier Vektoren
`rev()`	Reihenfolge invertieren
`sort()`	Sortieren
`order()`	Reihenfolge im unsortierten Vektor

Sortieren

- Mit `sort()` werden die Einträge eines Vektors sortiert

```
> (x <- c(5,3,8,4,2,6,7,4))
[1] 5 3 8 4 2 6 7 4
> sort(x)
[1] 2 3 4 4 5 6 7 8
> sort(x, decreasing=TRUE)
[1] 8 7 6 5 4 4 3 2
```

- `order()` gibt für jeden Eintrag des sortierten Vektors den Index dieses Eintrags im unsortierten Vektor an

```
> order(x)
[1] 5 2 4 8 1 6 7 3
> order(x, decreasing=TRUE)
[1] 3 7 6 1 4 8 2 5
```

Logische Vektoren

- `any()` gibt an, ob `TRUE`-Einträge vorliegen

```
> x <- 1:5
> (y <- x < 3)
[1] TRUE TRUE FALSE FALSE FALSE
> any(y)
[1] TRUE
```

- `all()` gibt an, ob alle Einträge den Wert `TRUE` haben

```
> all(y)
[1] FALSE
```

- Mit `which()` erhalten wir die Indizes und mit `sum()` die Anzahl dieser Einträge (Wieso?)

```
> which(y)
[1] 1 2
> sum(y)
[1] 2
```

Fehlende Werte

- Fehlende Werte werden in R mit `NA` („Not Available“) bezeichnet
- Das Ergebnis einer Berechnung mit `NA`-Werten ist immer `NA`

```
> (x <- c(NA, 1:5))
[1] NA 1 2 3 4 5
> sum(x)
[1] NA
```

- Einige Funktionen erlauben das Ausschließen solcher Werte

```
> sum(x, na.rm=TRUE)
[1] 15
```

- Mit `is.na` wird auf fehlende Werte geprüft

```
> is.na(x[1])
[1] TRUE
> x[1] == NA # Falsch!
[1] NA
```

Faktoren

- Zur Darstellung qualitativer Merkmale bietet sich der (nichtatomare) Datentyp `factor` an

```
> x <- c("male", "female", "female", "male", "female")
> (students <- factor(x))
[1] male female female male female
Levels: female male
```

- Die Ausprägungen des Faktors lassen sich mit `levels()` abfragen

```
> levels(students)
[1] "female" "male"
```

- Diese werden intern numerisch kodiert

```
> str(students)
Factor w/ 2 levels "female", "male": 2 1 1 2 1
> mode(students)
[1] "numeric"
```

Datentypen und Vektoren

- Die Elemente eines Vektors haben alle denselben Datentyp
- Dieser ist der kleinste Datentyp, der alle Elemente umfasst
- Der Datentyp eines Objekts wird mit `mode()` abgefragt

```
> x <- c(TRUE, 3.14, "R-Kurs")
> mode(x)
[1] "character"
> x
[1] "TRUE" "3.14" "R-Kurs"
```

- Mit `is.`Datentyp`()` kann man diesen überprüfen

```
> is.character(x)
[1] TRUE
```

Objekte

- Die Attribute eines Objekts werden mit `attributes()` ausgegeben

```
> attributes(students)
$levels
[1] "female" "male"

$class
[1] "factor"
```

- Mit `str()` erhält man dessen Struktur

```
> x <- 1:10
> str(x)
int [1:10] 1 2 3 4 5 6 7 8 9 10
```

- Ein Datentyp kann mit `as.Datentyp()` erzwungen werden

```
> as.complex(3.14)
[1] 3.14+0i
```

Matrizen

- Matrizen werden in R durch die Funktion `matrix()` erzeugt

```
> (A <- matrix(1:9, ncol=3))
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> (A <- matrix(1:9, nrow=3, byrow=TRUE))
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

- Sie sind mit Dimensionen ausgestattete Vektoren

```
> attributes(A)
$ dim
[1] 3 3
```

Zeilen- und Spaltennamen

- Die Zeilen und Spalten einer Matrix können mit `dimnames()` benannt werden

```
> Anmeldungen <- matrix(1:9, nrow=3)
> dimnames(Anmeldungen)
NULL
> zeilen <- c("R", "Latex", "Matlab")
> spalten <- c("Mathe", "Info", "Sonstiges")
> dimnames(Anmeldungen) <- list(zeilen, spalten)
> Anmeldungen
       Mathe Info Sonstiges
R          1    4         7
Latex      2    5         8
Matlab     3    6         9
```

- Das Attribut `dimnames` ist `NULL` oder eine Liste

Indizierung von Matrixelementen

- Mit `A[i,j]` wird auf Zeile `i` in Spalte `j` zugegriffen

```
> A <- matrix(1:9, nrow=3)
> A[2,3]
[1] 8
```

- Ganze Zeilen und Spalten werden mit `A[i,]` und `A[,j]` indiziert

```
> A[1,]
[1] 1 4 7
```

- Logische Indizierung ist ebenfalls möglich
- Benannte Zeilen und Spalten können über ihre Namen indiziert werden

```
> Anmeldungen["R", "Mathe"]
[1] 1
```

Diagonalmatrizen

- Der Befehl `diag()` erzeugt eine Diagonalmatrix

```
> diag(rep(1,3))
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

- Bei Anwendung von `diag()` auf eine Matrix erhält man deren Hauptdiagonale

```
> A <- diag(1, nrow=3) # ebenfalls die Einheitsmatrix
> diag(A)
[1] 1 1 1
```

Rechnen mit Matrizen

- Viele mathematische Operatoren sind komponentenweise definiert
- Mit `rbind()` und `cbind()` werden Matrizen und Vektoren zeilen- bzw. spaltenweise zusammengefügt

```
> (A <- cbind(1:2, 2:1))
     [,1] [,2]
[1,]    1    2
[2,]    2    1
```

- %*% bezeichnet die Matrixmultiplikation

```
> A.inv <- solve(A) # Inverse Matrix
      [,1]  [,2]
[1,] -0.33  0.67
[2,]  0.67 -0.33
> A %*% A.inv
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Rechnen mit Matrizen

Funktion	Beschreibung
`t()`	Transponierte Matrix
`det()`	Determinante
`solve()`	Matrixinverse
`solve(A,b)`	Löst das Gleichungssystem `Ax=b`
`crossprod(X,Y)`	$X^T Y$ (sehr schnell)
`qr()`	QR-Zerlegung
`eigen()`	Eigenwerte und -vektoren
`%*%`	Matrixmultiplikation
`nrow(), ncol()`	Zeilen-, Spaltenanzahl

Arrays

- Arrays oder Felder verallgemeinern Vektoren und Matrizen in beliebig viele Dimensionen
- Sie werden mit `array()` erzeugt und benötigen einen Vektor `dim` der Dimensionen

```
> array(1:8, dim=rep(2,3))
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

- Mehr dazu mit `?array`

Listen

- Listen können Objekte verschiedener Datentypen und -strukturen beinhalten
- Sie werden mit der Funktion `list()` erzeugt

```
> (x <- list(c(TRUE,FALSE), matrix(1:4,2), "R-Kurs"))
[[1]]
[1] TRUE FALSE

[[2]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[3]]
[1] "R-Kurs"
```

Listen

- Der Zugriff auf Listenelemente erfolgt rekursiv mit `[[]]`

```
> x[[2]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> x[[2]][1,1]
[1] 1
```

- Die Elemente einer Liste können mit `names()` benannt werden

```
> names(x) <- c("logisch", "numerisch", "zeichen")
```

- Mit dem $-Operator können diese dann indiziert werden

```
> x$zeichen
[1] "R-Kurs"
```

Data frames

- Data frames sind spezielle Listen, deren einzelne Elemente Vektoren gleicher Länge sind
- Sie sind *die* typische Struktur für Datensätze in R!
- Man erzeugt sie mit der Funktion `data.frame()`

```
> namen <- c("Peter","Anna","Daniel","Melanie")
> x1 <- c("Mathe","Info","Mathe","Physik")
> x2 <- c(4,4,5,3)
> x3 <- c("m","w","m","w")
> (RKurs <- data.frame(Studium=x1, Semester=x2,
+   Geschlecht=x3, row.names=namen))
          Studium   Semester   Geschlecht
Peter       Mathe          4            m
Anna         Info          4            w
Daniel      Mathe          5            m
Melanie    Physik          3            w
```

Data frames

- Die eingegebenen Zeichenketten werden als Faktoren interpretiert

```
> str(RKurs)
'data.frame': 4 obs. of 3 variables:
$ Studium: Factor w/ 3 levels "Info","Mathe",..: 2 1 2 3
$ Semester: num 4 4 5 3
$ Geschlecht: Factor w/ 2 levels "m","w": 1 2 1 2
```

- Data frames werden wie Listen oder wie Matrizen indiziert

```
> RKurs[[1]][1]
[1] Mathe
Levels: Info Mathe Physik
> RKurs[2,2]
[1] 4
```

Data frames

- Mit Hilfe der Namen kann ebenfalls auf die Einträge zugegriffen werden

```
> RKurs$Geschlecht[3]
[1] m
Levels: m w
```

- Der Befehl `attach()` hängt einen Datensatz in den Suchpfad ein, wodurch man die Variablen direkt ansprechen kann

```
> attach(RKurs)
> Semester
[1] 4 4 5 3
```

- `detach()` entfernt den Datensatz wieder aus dem Suchpfad
- Der Suchpfad lässt sich mit `search()` anzeigen

Data frames

- Möchte man nur auf wenige Zeilen oder Spalten zugreifen, so kann man auch `with()` verwenden

```
> with(RKurs, max(Semester) - min(Semester))
[1] 2
```

- Teilmengen eines Datensatzes erhält man mit `subset()`

```
> subset(RKurs, Studium == "Mathe")
        Studium   Semester   Geschlecht
Peter     Mathe          4            m
Daniel    Mathe          5            m
> subset(RKurs, Semester %in% 1:3)
          Studium   Semester   Geschlecht
Melanie    Physik          3            w
```

Data frames

- Datensätze können mit `split()` aufgeteilt werden

```
> split(RKurs, RKurs$Geschlecht)
$m
        Studium   Semester   Geschlecht
Peter     Mathe          4            m
Daniel    Mathe          5            m

$w
          Studium   Semester   Geschlecht
Anna         Info          4            w
Melanie    Physik          3            w
```

- Zusammenfügen geschieht wieder mit `rbind()` und `cbind()`, siehe aber auch `?merge`
- R enthält bereits eine ganze Reihe von Datensätzen, die man sich mit `data()` anzeigen lassen kann

Beispiel `iris`-Datensatz

- Andersons `iris`-Daten enthalten die Längen und Breiten der Blüten- und Kelchblätter dreier Schwertlilienarten
- Hilfeseite

  ```
  > ?iris
  ```

- Daten

  ```
  > iris
  ```

- Dateneditor

  ```
  > edit(iris)
  ```

- Struktur des Datensatzes

  ```
  > str(iris)
  ```

- Zusammenfassung des Datensatzes

  ```
  > summary(iris)
  ```

ASCII-Dateien

Die Funktion `read.table()` dient zum Einlesen von Datensätzen. Ihre wichtigsten Argumente sind:

- `file`: Pfad und Dateiname
- `header`: Falls Spaltennamen existieren, `header=TRUE` setzen
- `sep`: Trennzeichen zwischen den Spalten, Default ist `" "`

Zur Ausgabe von Datensätzen im ASCII-Format verwendet man `write.table()`.

```
> attach(iris)
> write.table(cbind(Sepal.Length, Sepal.Width),
+   "daten.txt", row.names=FALSE)
> read.table("daten.txt", header=TRUE, row.names=NULL)
> detach(iris)
```

ASCII-Dateien

- Einspaltiges Einlesen funktioniert auch mit der Funktion `scan()`

```
> write.table(1:10, "daten.txt", row.names=FALSE,
+   col.names=FALSE)
> scan("daten.txt")
Read 10 items
[1] 1 2 3 4 5 6 7 8 9 10
```

- Außerdem kann man damit sogar von der Tastatur einlesen

```
> scan()
1: 1 2 3 4 5
6:
Read 5 items
[1] 1 2 3 4 5
```

R Objekte

- Einzelne R Objekte lassen sich mit `dump()` exportieren

```
> dump("iris", file="iris.txt")
```

- Mit `source()` können diese wieder eingelesen werden

```
> rm(list=ls())
> source("iris.txt")
> ls()
[1] "iris"
```

- Alternativ können auch `dput()` und `dget()` verwendet werden, allerdings ohne Speicherung der Objektnamen

```
> dput(iris, "iris.txt")
> dget("iris.txt")
> ls()
character(0)
```

Grundlagen

- R ist eine interpretierte Sprache, d.h. der Code wird erst zur Laufzeit ausgewertet
- Der Geschwindigkeitsnachteil lässt sich durch vektorwertiges Programmieren teilweise ausgleichen
- Bereits implementierte Funktionen sind eigenem Code häufig vorzuziehen
- Aufwendige Operationen können in C, C++ oder Fortran ausgelagert werden

Bedingte Anweisungen

Bedingte Anweisungen implementieren Fallunterscheidungen. Dazu existieren zwei Möglichkeiten:

- `if (Bedingung) Ausdruck1 else Ausdruck2`
- `ifelse (Bedingung, Ausdruck1, Ausdruck2)`

Bei einem `ifelse`-Konstrukt darf die `Bedingung` auch vektorwertig sein.

Ist die `Bedingung` gültig (`TRUE`), so wird `Ausdruck1` ausgewertet, ansonsten `Ausdruck2`.

Bedingte Anweisungen

- `if ... else`

```
> x <- 1

> if (x == 1) {
+   x <- x + 1
+   y <- 1
+ } else
+   y <- 2

> c(x=x, y=y)
x y
2 1
```

- Für mehrzeilige Kommentare eignet sich der Ausdruck

```
> if (FALSE) {Kommentar}
```

Bedingte Anweisungen

- Bei `ifelse`-Konstrukten sind neben vektorwertigen `Bedingungen` auch vektorwertige `Ausdrücke` möglich

```
> x <- 1:5
> ifelse (x > 2, x, 0)
[1] 0 0 3 4 5
```

- Vergleiche dazu eine `if ... else` Konstruktion

```
> if (x > 2) x else 0
[1] 0
Warning message:
In if (x > 2) x else 0:
  Bedingung hat Länge > 1 und nur das erste Element
  wird benutzt
```

- Die Notation mit den runden Klammern ist für komplexe Ausdrücke ungünstig

Bedingte Anweisungen

- Müssen viele Fälle überprüft werden, so bietet sich auch die Funktion `switch()` an

```
> switch(2, x=1, y=2, z=3) # zweites Objekt
[1] 2
> switch("x", x=1, y=2, z=3) # Objekt x
[1] 1
> switch("a", x=1, y=2, z=3)
NULL
```

- Statt `NULL` kann man auch ein unbenanntes Objekt als letzten Fall angeben

```
> switch("a", x=1, y=2, z=3, 4)
[1] 4
```

Schleifen

Schleifen dienen dem wiederholten Ausführen von Programmteilen. In R existieren dazu drei Varianten und zwei wesentliche Kontrollbefehle:

- `repeat {Ausdruck}`: Wiederholung von `Ausdruck`
- `while (Bedingung) {Ausdruck}`: Wiederholung von `Ausdruck`, solange `Bedingung` erfüllt ist
- `for (i in M) {Ausdruck}`: Wiederhole `Ausdruck` für jedes `i` in `M`
- `next`: Sprung in den nächsten Iterationsschritt
- `break`: Sofortiges verlassen der Schleife

Schleifen

- Der `Ausdruck` einer `repeat`-Schleife wird solange wiederholt, bis er mit `break` beendet wird

```
> x <- 0
> repeat {
+   x <- x + 1
+   if (x < 5) next
+   print(x)
+   break
+ }
[1] 5
```

- Die `while`-Schleife endet, sobald die `Bedingung` ungültig wird

```
> x <- 0
> while (x < 5)
+   x <- x + 1
> x
[1] 5
```

Vektorwertiges Programmieren

Operator / Funktion	Beschreibung
`%*%`	Vektor- oder Matrixmultiplikation
`%o%`, `outer()`	Äußeres Produkt
`%x%`, `kronecker()`	Kronecker-Produkt
`rowSums()`, `colSums()`	Schnelle Zeilen-, Spaltensummen
`rowMeans()`, `colMeans()`	Schnelle Zeilen-, Spaltenmittel
`apply()`	Zeilen- und spaltenweises Anwenden einer Fkt. auf Matrizen bzw. Arrays
`lapply()`	Elementweises Anwenden einer Fkt. auf Listen, Datensätze und Vektoren
`sapply()`	Wie `lapply()`, liefert „einfaches“ Objekt
`mapply()`	multivariates `sapply()`
`tapply()`	Tabellen gruppiert nach Faktoren

Vektorwertiges Programmieren

Das äußere Produkt `outer(X, Y, FUN)` zweier Arrays `X` und `Y` ist ein Array `A` der Dimension `c(dim(X),dim(Y))` mit Einträgen

```
A[c(Index.X, Index.Y)] = FUN(X[Index.X], Y[Index.Y])
```

```
> (X <- 1:5)
[1] 1 2 3 4 5
> X %o% X # entspricht outer(X, X, FUN="*")
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5   X[1] * X[5]
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

Vektorwertiges Programmieren

Mit `apply(X, MARGIN, FUN)` wird eine Funktion `FUN` zeilen- oder spaltenweise auf ein Array `X` angewendet

```
> (X <- matrix(1:12, ncol=4))
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> apply(X, MARGIN=1, sum) # Zeilen
[1] 22 26 30
> apply(X, MARGIN=2, sum) # Spalten
[1] 6 15 24 33
> rowSums(X)
[1] 22 26 30
> colSums(X)
[1] 6 15 24 33
```

Vektorwertiges Programmieren

Mit `lapply(X, FUN)` lässt sich `FUN` sehr schnell elementweise auf die Liste, den Datensatz bzw. den Vektor `X` anwenden

```
> (x <- lapply(iris[,-5], mean)) # Mittelwert
$Sepal.Length
[1] 5.843333

$Sepal.Width
[1] 3.057333

$Petal.Length
[1] 3.758

$Petal.Width
[1] 1.199333

> mode(x)
[1] "list"
```

Vektorwertiges Programmieren

Die Funktion `sapply(X, FUN)` arbeitet völlig analog zu `lapply()`, versucht aber das Ergebnisobjekt zu vereinfachen.

```
> (x <- sapply(iris[,-5], mean))
Sepal.Length  Sepal.Width  Petal.Length  Petal.Width
    5.843333     3.057333      3.758000     1.199333
> mode(x)
[1] "numeric"
> x <- sapply(iris[,-5], mean, simplify=FALSE)
> mode(x)
[1] "list"
```

Vektorwertiges Programmieren

- `mapply(FUN, ...)` ist eine multivariate Version von `sapply()`. `FUN` wird dabei elementweise auf alle Objekte in `...` angewendet.

```
> mapply(sum, 1:10, 10:1, 5) # 1+10+5, 2+9+5, ...
[1] 16 16 16 16 16 16 16 16 16 16
```

- Mit `tapply(X, INDEX, FUN)` wird `FUN` auf den Datensatz `X`, gruppiert nach dem Faktor `Index`, angewendet.

```
> attach(iris)
> tapply(Sepal.Length, Species, mean)
setosa   versicolor   virginica
 5.006        5.936       6.588
> detach(iris)
```

Vektorwertiges Programmieren

Gruppierte Daten lassen sich auch durch `aggregate(X, by, FUN)` entsprechend den Faktoren in `by` mit `FUN` zusammenfassen

```
> (aggregate(iris[1:2], by=list(Species=iris$Species),
+   FUN=mean) -> x)
      Species   Sepal.Length   Sepal.Width
1      setosa          5.006         3.428
2  versicolor          5.936         2.770
3   virginica          6.588         2.974

> str(x)
'data.frame': 3 obs. of 3 variables:
$ Art: Factor w/ 3 levels "setosa", "versicolor",..: 1 2 3
$ Sepal.Length: num 5.01 5.94 6.59
$ Sepal.Width : num 3.43 2.77 2.97
```

Funktionen

- Eigene Funktionen werden mit dem Befehl `function` definiert. Die allgemeine Syntax ist:

```
Funktionsname <- function(Argumente){
   Befehlsfolge
}
```

- Die Rückgabe eines Objekts geschieht mit `return()`

```
> kreisumfang <- function(radius){
+     u <- 2 * pi * radius
+     return(u)
+ }

> kreisumfang(1)
[1] 6.283185

> kreisumfang(1:5)
[1] 6.283185 12.566371 18.849556 25.132741 31.415927
```

Funktionen

- Ohne explizites `return()` wird das zuletzt erzeugte Objekt zurückgegeben

```
> kreisumfang <- function(radius){
+     2 * pi * radius
+ }

> kreisumfang(2)
[1] 12.566371
```

- Mit `invisible()` erfolgt die Rückgabe ohne Ausgabe auf die Konsole

```
> kreisumfang <- function(radius){
+     invisible(2 * pi * radius)
+ }

> kreisumfang(3)

> (x <- kreisumfang(3))
[1] 18.849556
```

Funktionen

Soll mehr als ein Objekt zurückgegeben werden, so muss eine Liste verwendet werden.

```
> kreis <- function(radius){
+     a <- pi * radius^2
+     u <- 2 * pi * radius
+     liste <- list(Flaeche=a, Umfang=u)
+     return(liste)
+ }

> kreis(1)
$Flaeche
[1] 3.141593

$Umfang
[1] 6.283185
```

Funktionen

- Voreinstellungen für die Argumente werden mit = gesetzt

```
> kreisumfang <- function(radius=1){
+     2 * pi * radius
+ }

> kreisumfang() # Einheitskreis
[1] 6.283185
```

- Das ...-Argument erlaubt es, eingegebene Argumente ohne explizite formale Definition weiterzureichen

```
> dreipunkte <- function(X, ...){
+     matrix(X, ...)
+ }

> dreipunkte(1:6, ncol=3)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Funktionen

- In der Regel schreibt man Funktionen mit einem Editor in separate Dateien und liest sie dann mit `source()` in R ein

```
> dump("dreipunkte", "dreipunkte.txt")
> source("dreipunkte.txt")
```

- Durch Eingabe des Namens einer Funktion ohne Klammern und Argumente kann man sich deren Definition ansehen

```
> dreipunkte
function(X, ...){
matrix(X, ...)
}
```

- Alternativ kann auch die Funktion `get()` verwendet werden

```
> get("dreipunkte")
function(X, ...){
matrix(X, ...)
}
```

Programmierstil

- Wiederverwendbarkeit

  Eine Funktion sollte möglichst allgemein geschrieben sein, so dass sie nicht nur auf den aktuellen Datensatz anwendbar ist.

- Nachvollziehbarkeit

  Eine Funktion sollte so viele Kommentare wie möglich enthalten, am besten sogar eigene Dokumentationen wie z.B. Hilfeseiten.

- Lesbarkeit

  Es sollte ein Kompromiss zwischen kompaktem und noch lesbarem Code gefunden werden. Sinnvolle Bezeichner und Leerzeichen erhöhen die Übersichtlichkeit.

- Vektorwertiges Programmieren

  Schleifen sollten, wenn möglich, durch vektorwertige Operationen ersetzt werden

Sinnvolles Benutzen von Schleifen

- Wachsende Objekte zu Beginn vollständig initialisieren

```
> x <- NULL # langsam
> for (i in 1:n)
+     x <- c(x, funktion(i))

> x <- numeric(n) # schneller
> for (i in 1:n)
+     x[i] <- funktion(i)
```

- Zur Erzeugung von Vektoren bestimmter Datentypen dienen außerdem `logical()`, `integer()`, `complex()` und `character()`
- Mit `vector(mode="list", length=n)` lässt sich eine (leere) Liste der Länge `n` erzeugen

Sinnvolles Benutzen von Schleifen

Keine Berechnungen sollten mehrfach ausgeführt werden. Anstelle von

```
> x <- numeric(n)
> for (i in 1:n)
+     x[i] <- 2 * n * pi * funktion(i)
```

sollte man beispielsweise besser

```
> x <- numeric(n)
> for (i in 1:n)
+     x[i] <- funktion(i)
> x <- 2 * n * pi * x
```

schreiben. So werden einige Operationen pro Durchlauf gespart.

R-Pakete

- Funktionalität und Datensätze von R können durch Pakete erweitert werden
- Die Pakete `ISwR` und `UsingR` liefern beispielsweise die Datensätze zu den Lehrbüchern von Peter Dalgaard und John Verzani
- Sie werden mit `install.packages()` installiert und mit `update.packages()` aktualisiert
- Mit dem Befehl `library()` können die installierten Pakete angezeigt und geladen werden
- `detach()` entfernt sie wieder aus dem Suchpfad

```
> library(help = "UsingR")
> library(UsingR)
> detach(package:UsingR)
```

Grundlagen

- Eine der besonderen Stärken von R liegt im Grafikbereich
- High-level Grafikfunktionen erzeugen vollständige Grafiken mit Achsen, Beschriftungen und einer Darstellung der Daten
- Mit Low-level Grafikfunktionen können Grafiken initialisiert und mit zusätzlichen Punkten, Linien oder Beschriftungen versehen werden
- Das Grafiksystem von R ist generell nicht für dynamische und interaktive Grafik ausgelegt

High-level Grafik

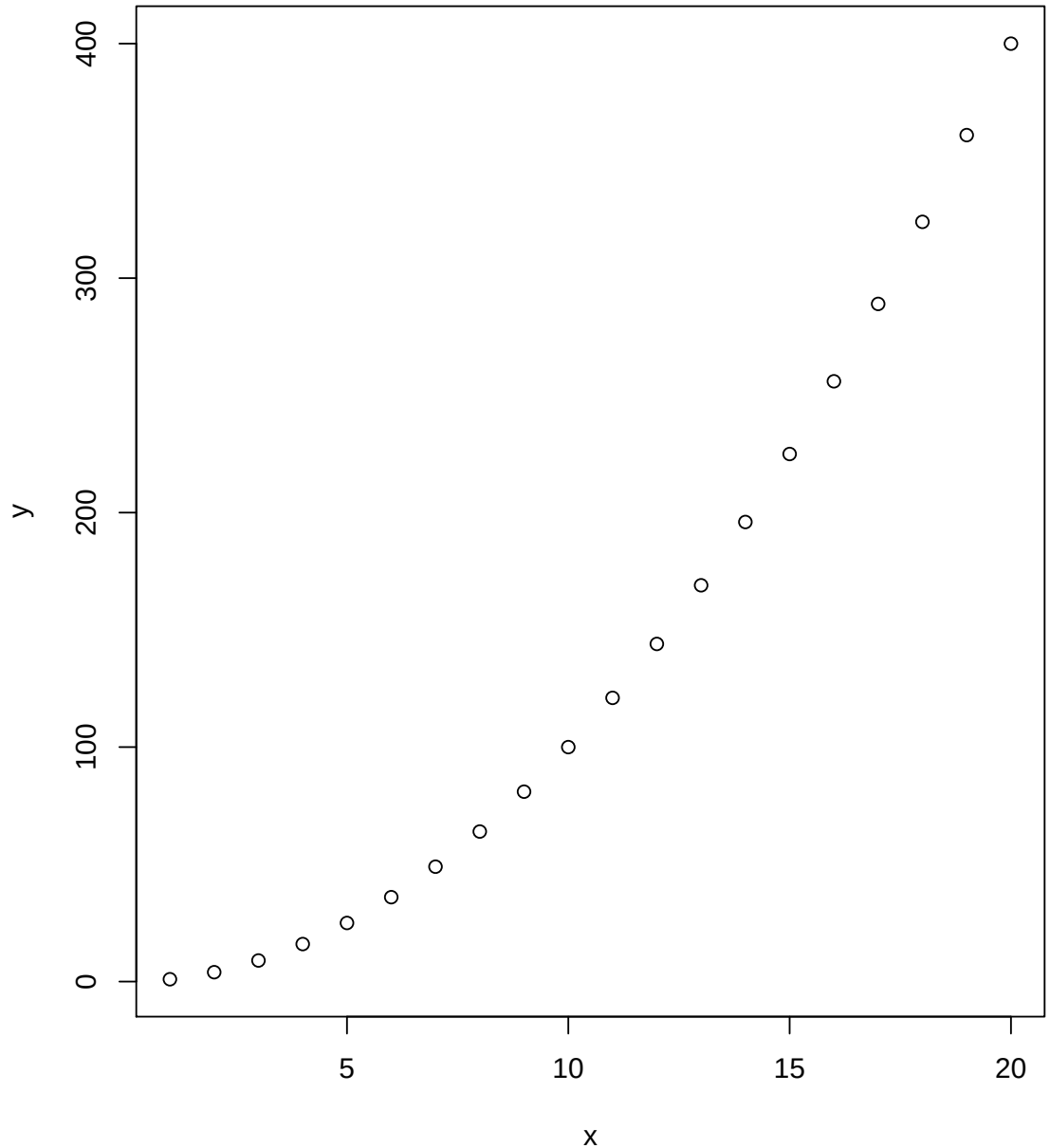
- Die wichtigste Funktion zum Erstellen von Grafiken ist `plot()`
- Als generische Funktion erzeugt sie unterschiedliche Grafiken für unterschiedliche Objekte
- Mit `plot(x,y)` können beispielsweise zwei gleich lange Vektoren x und y gegeneinander abgetragen werden

```
> x <- 1:20
> y <- x^2
> plot(x,y)
```

- Alternativ trägt `plot(y)` den Vektor y gegen `1:length(y)` ab
- Ist y stattdessen eine zweispaltige Matrix, so werden die Punkte (y_{i1}, y_{i2}) gezeichnet

```
> y <- cbind(1:20, (1:20)^2)
> plot(y)
```

High-level Grafik



High-level Grafik

- Mit dem Parameter `type` wird der Darstellungstyp geändert

`type`	Beschreibung
`"p"`	Punkte (Standard)
`"l"`	Linien
`"b"`	Punkte und Linien
`"o"`	überlagerte Punkte und Linien
`"h"`	vertikale Höhenlinien
`"s"`	Treppenfunktion
`"n"`	leere Grafik

- Das Darstellungsymbol der Punkte lässt sich mit `pch` einstellen
- Die Größe der Symbole wird mit `cex` skaliert

High-level Grafik

- Der Parameter `lty` legt die Dartsellungsart der Linien fest

`lty`	Beschreibung
`"solid"`	durchgezogen (Standard)
`"dashed"`	gestrichelt
`"dotted"`	gepunktet
`"dotdash"`	gestrichelt und gepunktet
`"longdash"`	lang gestrichelt
`"twodash"`	kurz und lang gestrichelt
`"blank"`	unsichtbar

- Die Breite gezeichneter Punkte und Linien ändert man mit `lwd`

High-level Grafik

- Überschrift und Untertitel der Grafik werden mit `main` und `sub` gesetzt
- Die Achsenbeschriftungen lassen sich mit `xlab` und `ylab` ändern
- Sind keine Beschriftungen erwünscht, so weist man dem jeweiligen Parameter die leere Zeichenkette `""` zu
- Die Einstellung `axes=FALSE` erzeugt eine Grafik ohne Achsen
- Mit `bty` wird die Darstellung der umgebenden Box festgelegt
- Der Plotbereich lässt sich mit `xlim=c(a1,b1)` und `ylim=c(a2,b2)` auf $[a_1, b_1] \times [a_2, b_2]$ einstellen

High-level Grafik

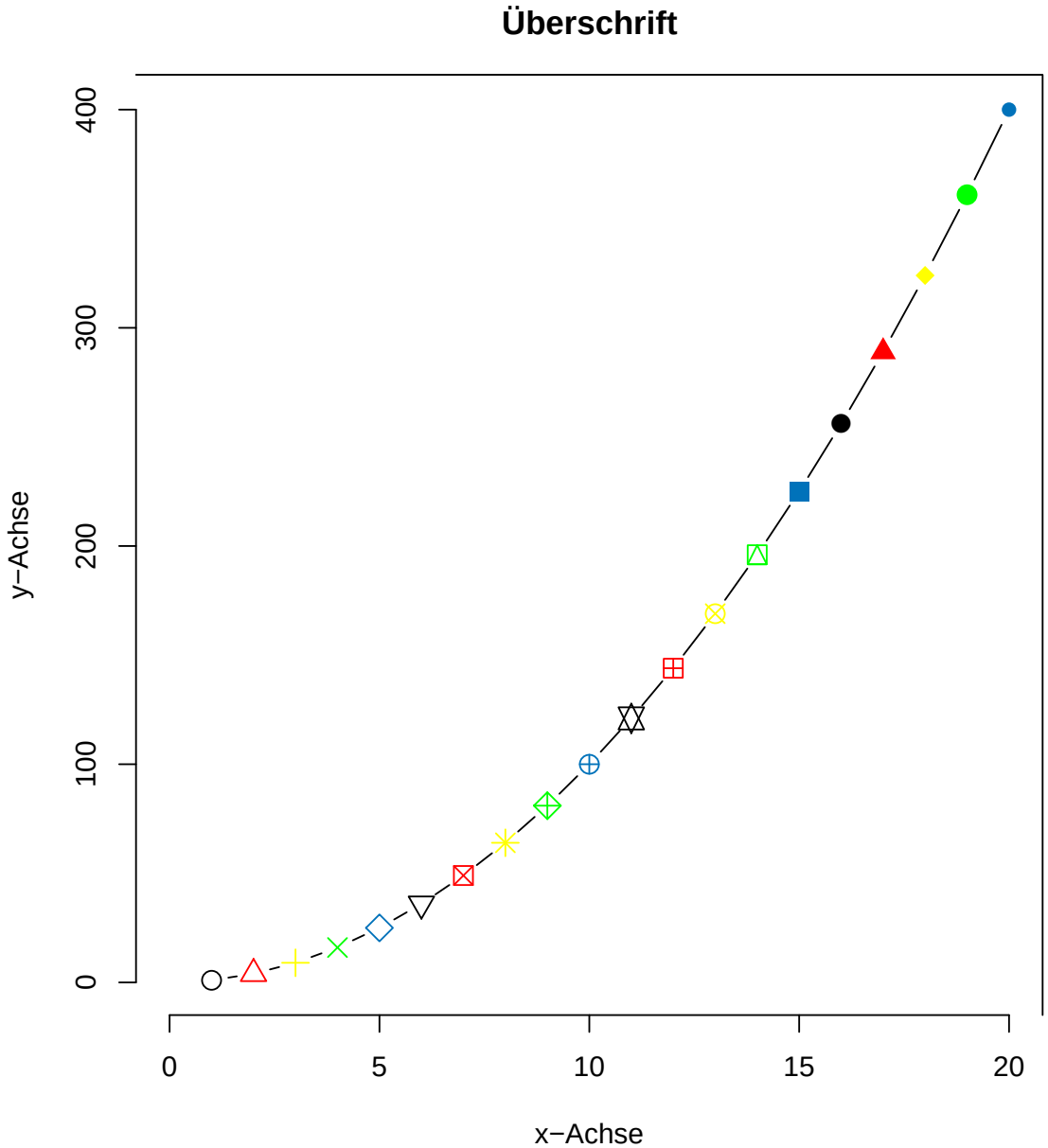
- Für die farbliche Gestaltung der Grafik steht der Parameter `col` zur Verfügung
- Die numerisch kodierten Werte von `col` findet man in `palette()`

```
> palette()
[1] "black" "red" "green3" "blue" "cyan" "magenta"
[7] "yellow" "gray"
```

- Weitere Farbbezeichnungen findet man mit dem Befehl `colors()`

```
> x <- cbind(1:20, (1:20)^2)
> blau <- rgb(0, 114, 186, maxColorValue=255)
> farbe <- c("black", "red", "yellow", "green", blau)
> plot(x, type="b", xlim=c(0,20), main="Überschrift",
+   xlab="x-Achse", ylab="y-Achse", pch=x[,1],
+   cex=1.5, bty="7", col=farbe)
```

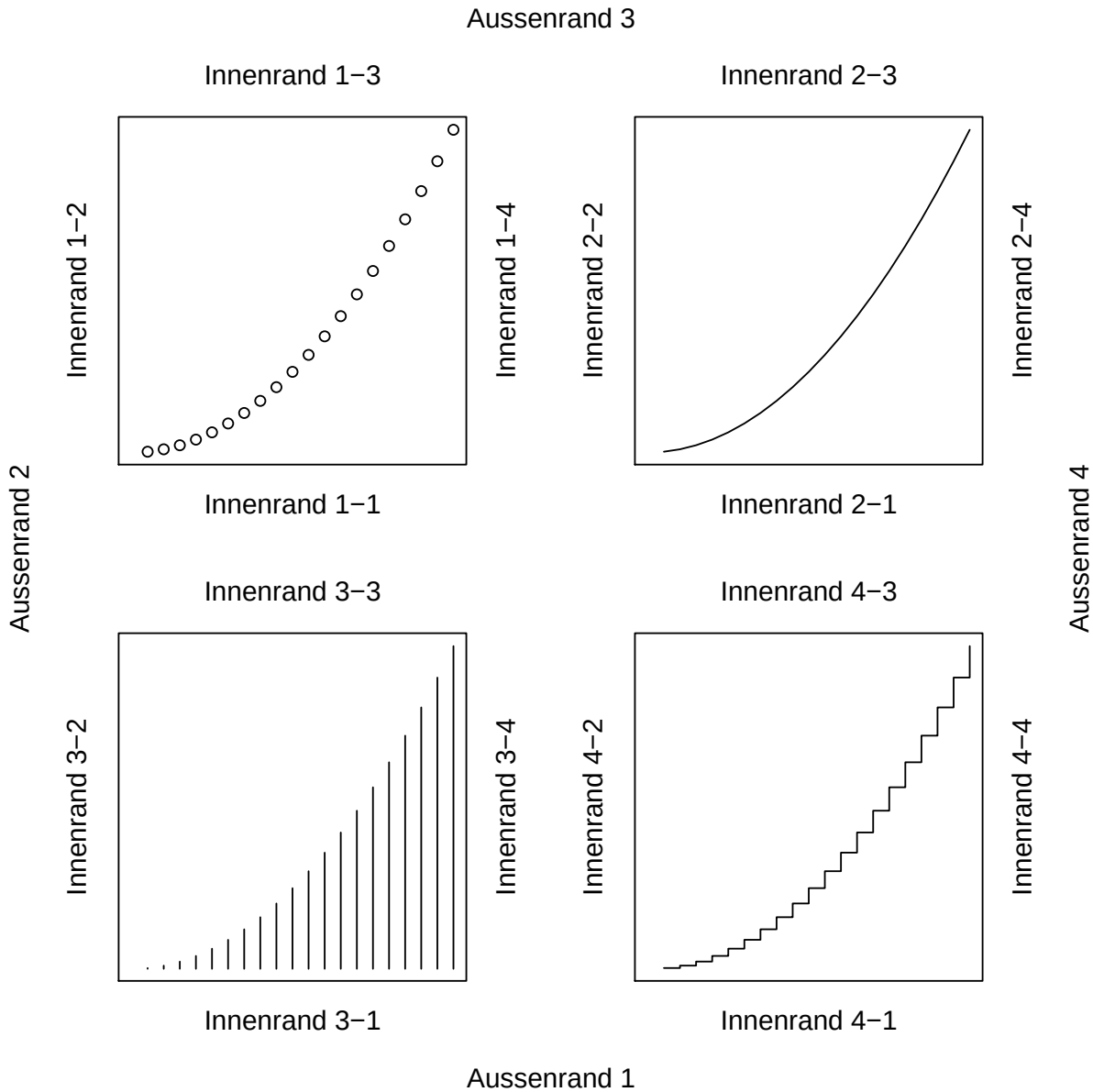
High-level Grafik



High-level Grafik

- Die oben genannten und zahlreiche weitere Parameter lassen sich auch mit dem Befehl `par()` einstellen
- Genauer gesagt setzt `par()` die Einstellungen global für alle weiteren Grafiken
- Es empfiehlt sich, die Hilfeseite `?par` zu studieren
- Mehrere Grafiken in einem Fenster können durch die Einstellung `par(mfrow=c(m,n))` erstellt werden
- Letzterer Befehl erzeugt eine Designmatrix mit `m` Zeilen und `n` Spalten
- Für die Größe der inneren und äußeren Ränder der Grafik stehen die Parameter `mar`, `mai`, `oma` und `omi` zur Verfügung

High-level Grafik



Wichtige Argumente in Grafikfunktionen und `par()`

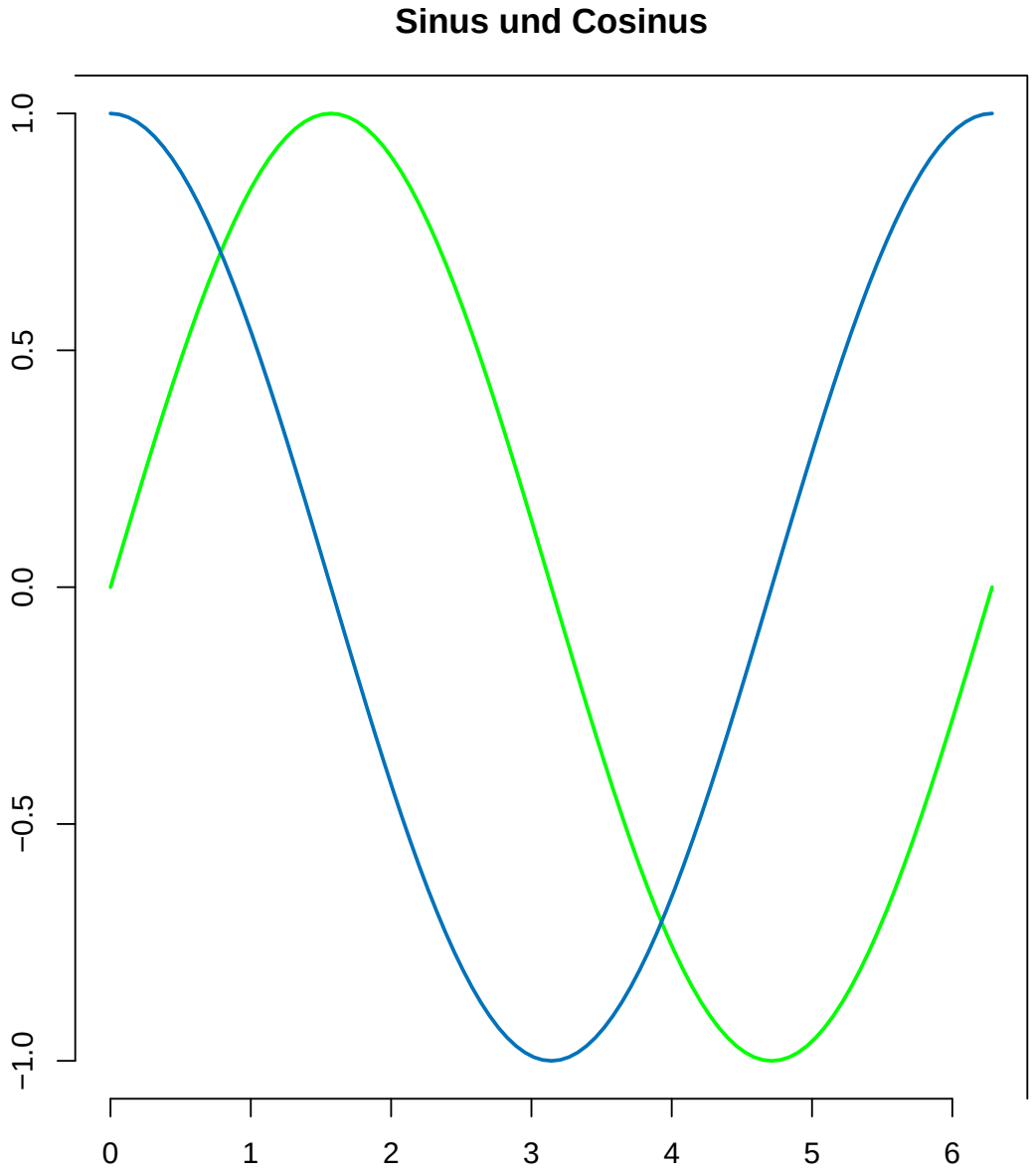
Argument	Beschreibung
`axes`	Achsen (nicht) zeichnen
`bg`	Hintergrundfarbe
`cex`	Skalierung der Symbolgröße
`col`	Farben
`las`	Ausrichtung der Achsenbeschriftungen
`lty`, `lwd`	Linientyp, -breite
`main`, `sub`	Überschrift, Untertitel
`mfcol`, `mfrow`	mehrere Grafiken in einem Bild
`pch`	Symbol für Punkte
`type`	Darstellungstyp
`xlab`, `ylab`	x-, y-Achsenbeschriftung
`xlim`, `ylim`	Plotbereich in x-, y-Richtung

High-level Grafik

- Kurven werden mit der Funktion `curve()` erzeugt
- Dem Parameter `expr` kann eine Funktion von x oder der Name einer solchen Funktion zugewiesen werden
- Mit `from` / `to` oder `xlim` legt man den Plotbereich in x-Richtung fest
- Die Option `add=TRUE` fügt die Kurve einer bestehenden Grafik hinzu

```
> curve(sin, from=0, to=2*pi, bty="7", xlab="", ylab="",
+   main="Sinus und Cosinus", lwd=2, col="green")
> curve(cos, from=0, to=2*pi, lwd=2, col=blau, add=TRUE)
```

High-level Grafik



Low-level Grafik

- Punkte und Verbindungslinien fügt man mit den Befehlen `points()` und `lines()` in die Grafik ein
- Mit `segments(x0,y0,x1,y1)` werden die Punkte $(x_0[i], y_0[i])$ und $(x_1[i], y_1[i])$ unabhängig durch Liniensegmente verbunden
- Analog erzeugt `arrows(x0,y0,x1,y1)` Verbindungspfeile
- Die Richtung der Pfeile wird mit dem Parameter `code` eingestellt
- Der Befehl `abline(a,b)` zeichnet die Gerade $y = a + bx$ in die Grafik ein
- Entsprechend können mit `abline(h=y)` und `abline(v=x)` horizontale und vertikale Linien erstellt werden

Low-level Grafik

- Der Text `labels` wird mit `text(x,y,labels)` an der Position (x, y) in die Grafik eingefügt
- Die Beschriftung der Ränder geschieht mit `mtext()`
- Mit `title(main,sub)` lassen sich Überschriften und Untertitel hinzufügen
- Durch die `plot`-Optionen `axes=FALSE`, `xaxt="n"` und `yaxt="n"` werden bekanntlich beim Erstellen der Grafik Teile der Box und der Achse weggelassen
- Zur individuellen Gestaltung dienen dann die Funktionen `box()` und `axis()`
- Mit dem Befehl `legend(x,y,legend)` wird eine Legende erzeugt
- Anstelle der (x, y)-Koordinaten kann auch `locator(1)` angegeben und die Legende mit einem Mausklick gesetzt werden

Low-level Grafik

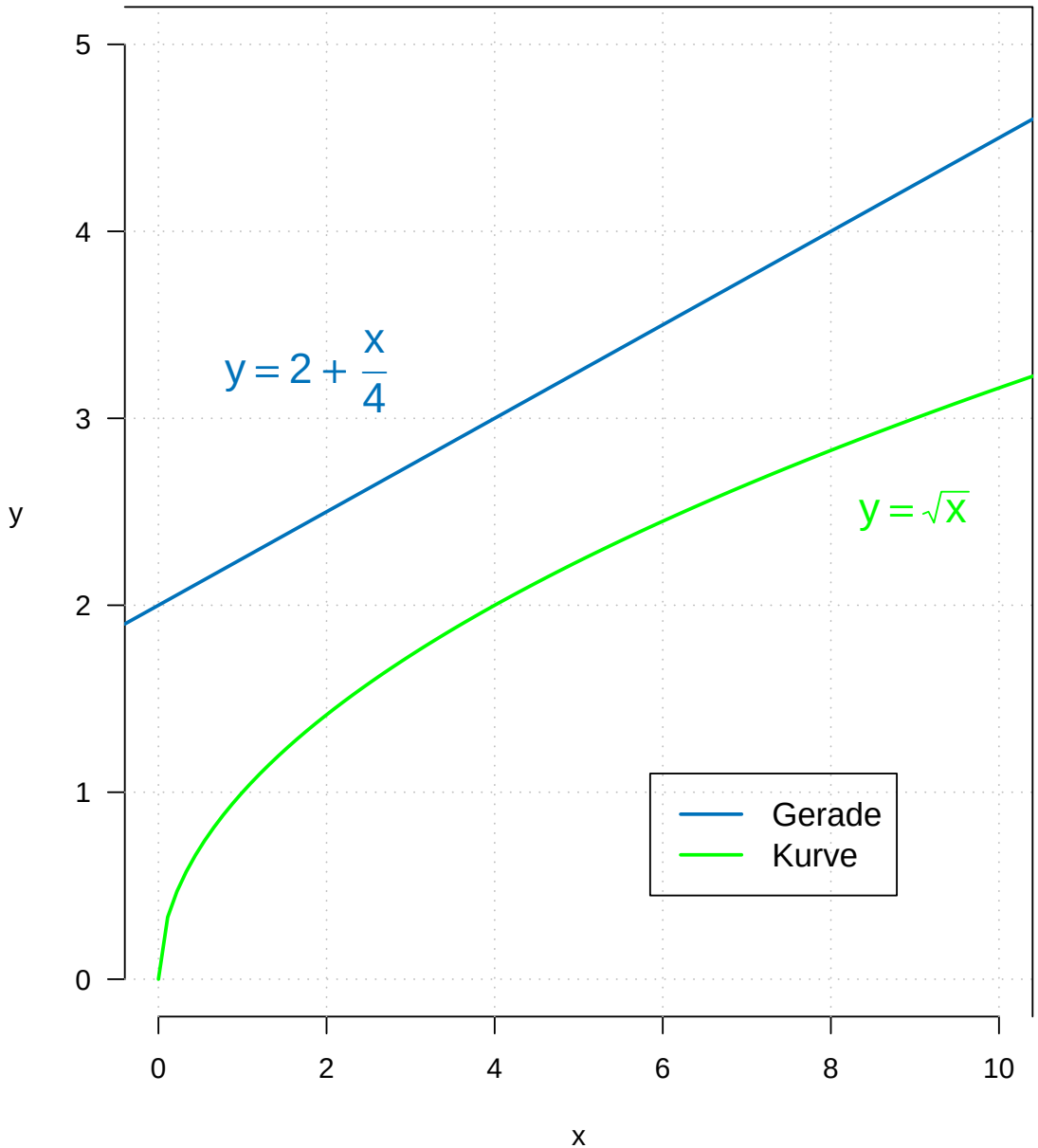
- Viele Grafikfunktionen können mathematische Notation in ihren Beschriftungen verarbeiten
- Diese wird mit `expression()` und geeigneten Schlüsselwörtern erzeugt und dann einem Beschriftungsparameter zugewiesen
- `expression(sqrt(sigma))` steht beispielsweise für $\sqrt{\sigma}$
- Genauere Details erhält man mit `?plotmath` und `demo(plotmath)`
- Da Variablen in `expression()` nicht ausgewertet werden, ersetzt man deren Werte mit `substitute()` oder `bquote()` und `.()`

```
> x <- 1
> substitute(x == X, list(X = x))
x == 1
> bquote(x == .(x))
x == 1
```

Low-level Grafik

```
> plot.new()
> plot.window(xlim=c(0,10), ylim=c(0,5))
> axis(1, at=seq(0, 10, by=2))
> axis(2, at=0:5, las=1)
> mtext("x", side=1, line=3)
> mtext("y", side=2, line=3, las=2)
> grid()
> box(bty="7")
> curve(sqrt(x), from=0, to=11, lwd=2, col="green", add=TRUE)
> text(9, 2.5, labels=expression(y == sqrt(x)), cex=1.5,
+   col="green")
> abline(2, 0.25, lwd=2, col=blau)
> text(1.75, 3.25, labels=expression(y == 2~+~frac(x,4)),
+   cex=1.5, col=blau)
> legend(5.85, 1.1, legend=c("Gerade", "Kurve"), lwd=2,
+   cex=1.2, col=c(blau, "green"))
```

Low-level Grafik



Low-level Grafik

Funktion	Beschreibung
`abline()`	„intelligente“ Linie
`arrows()`	Pfeile
`axis()`	Achsen
`grid()`	Gitternetz
`legend()`	Legende
`lines()`	Linien (schrittweise)
`mtext()`	Text in den Rändern
`plot.new()`	Grafik initialisieren
`plot.window()`	Koordinatensystem initialisieren
`points()`	Punkte
`polygon()`	Polygone
`segments()`	Linien (vektorwertig)
`text()`	Text
`title()`	Beschriftung

Zeichenketten

- Zeichenketten kommen in Objekten vom Typ `character` vor
- Die Funktion `paste()` wandelt ihre Argumente in Zeichenketten um und fügt diese dann zusammen

```
> paste("Tag", 1:4, sep=" ")
[1] "Tag 1" "Tag 2" "Tag 3" "Tag 4"
```

- Mit `parse()` wird eine Zeichenkette in eine `expression` konvertiert

```
> (ep <- parse(text="x <- 1"))
expression(x <- 1)
attr(,"srcfile")
<text>
> eval(ep)
> x
[1] 1
```

Zeichenketten

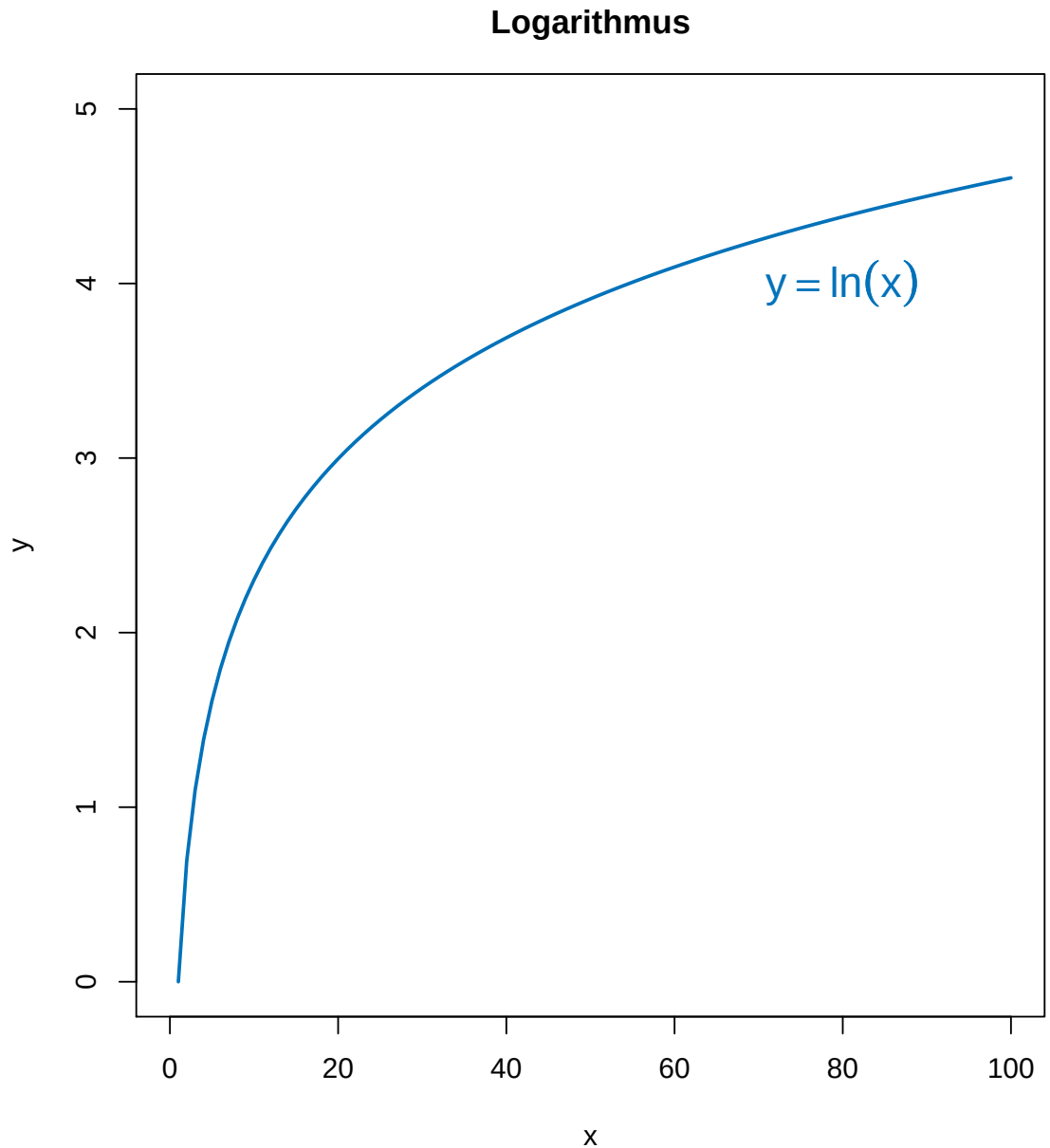
Funktion	Beschreibung
`cat()`	Ausgabe in Konsole und Dateien
`deparse()`	`expression` in Zeichenfolge konvertieren
`grep()`	Zeichenfolge in Vektoren suchen
`match()`, `pmatch()`	Suchen von Zeichenketten in anderen
`nchar()`	Anzahl Zeichen in Zeichenkette
`parse()`	Konvertierung in eine `expression`
`paste()`	Zusammensetzen von Zeichenketten
`strsplit()`	Zerlegen von Zeichenketten
`sub()`, `gsub()`	Ersetzen von Teilzeichenfolgen
`substring()`	Ausgabe und Ersetzen von Teilzeichenfolgen
`toupper()`, `tolower()`	Umwandlung in Groß-, Kleinbuchstaben

Ausgabegeräte

- Grafiken können auch auf anderen Geräten als dem Bildschirm ausgegeben werden
- Ein solches `Device` wird mit einer entsprechenden Funktion wie `pdf()` geöffnet
- Danach folgen die üblichen Grafikbefehle
- Mit `dev.off()` wird das Ausgabegerät wieder geschlossen
- Der Befehl `graphics.off()` schließt alle geöffneten Devices

```
> pdf("grafik.pdf")
> curve(log, xlim=c(0,100), ylim=c(0,5), lwd=2,
+   main="Logarithmus", xlab="x", ylab="y", col=blau)
> text(80, 4, labels=expression(y == ln(x)), cex=1.5,
+   col=blau)
> dev.off()
```

Ausgabegeräte



Ausgabegeräte

- Das aktuelle Device erhält man mit `dev.cur()`
- Der Befehl `dev.list()` gibt eine Liste aller aktiven Geräte aus
- Der Inhalt eines Gerätes kann mit `dev.copy()` und `dev.print()` in ein anderes kopiert werden

```
> curve(log, xlim=c(0,100), ylim=c(0,5), lwd=2,
+   main="Logarithmus", xlab="x", ylab="y", col=blau)
> text(80, 4, labels=expression(y == ln(x)), cex=1.5,
+   col=blau)
> dev.print(device=pdf, "grafik.pdf")
```

- `eps`-Dateien werden mit `postscript("Zieldatei.eps")` erzeugt
- Die Funktion `dev.copy2eps()` kopiert direkt in eine `eps`-Datei

Ausgabegeräte

Funktion	Beschreibung
`bmp()`	Bitmap
`jpeg()`	JPEG
`pdf()`	PDF
`png()`	PNG
`postscript()`	PostScript
`tiff()`	TIFF
`X11()`	Bildschirmgrafik
`win.print()`	Ausgabe an Drucker (Windows)
`windows()`	Bildschirmgrafik (Windows)

Statistische Daten

Bei statistischen Daten unterscheiden wir:

- qualitative (kategorielle) Merkmale

  ```
  studiengang <- c("Mathe", "Info", "Mathe", "Physik")
  ```

- quantitative (diskrete oder stetige) Merkmale

  ```
  note <- c(1.0, 2.7, 4.0, 1.3)
  ```

Daneben existiert die folgende Klassifikation:

- univariat (nur eine Variable)
- bivariat (zwei Variablen)
- multivariat (zwei oder mehr Variablen)

Beispiel `juul`-Daten

Die `juul`-Daten aus dem `ISwR`-Paket enthalten klinische Messungen des „Insulinähnlichen Wachstumsfaktors 1“ (IGF-1).

```
> library(ISwR)
> ?juul
> juul
> str(juul)
```

Wir wollen die kategoriellen Variablen `menarche`, `sex` und `tanner` mit Hilfe des Befehls `transform()` in Faktoren umwandeln.

```
> juul <- transform(juul,
+   menarche=factor(menarche, labels=c("No","Yes")),
+   sex=factor(sex, labels=c("M","F")),
+   tanner=factor(tanner, labels=c("I","II","III","VI","V")))
> summary(juul)
> attach(juul)
```

Kategorielle Daten

- Kategorielle Daten werden üblicherweise als Faktoren gespeichert
- Sie lassen sich mit `table()` zu Tabellen zusammenfassen
- Wir beginnen univariat mit dem Merkmal `tanner` aus `juul`

```
> (tab <- table(tanner)) # absolute Häufigkeiten
tanner
  I   II  III   VI    V
515 103   72   81  328
> options(digits=1)
> tab / sum(tab) # relative Häufigkeiten
tanner
   I    II   III    VI     V
0.47 0.09 0.07 0.07 0.30
```

Kategorielle Daten

- Säulendiagramme dienen der Veranschaulichung dieser Tabellen
- Sie werden mit der Funktion `barplot()` erzeugt

```
> barplot(tab, main="Tanner-Stadien", col=blau)
```

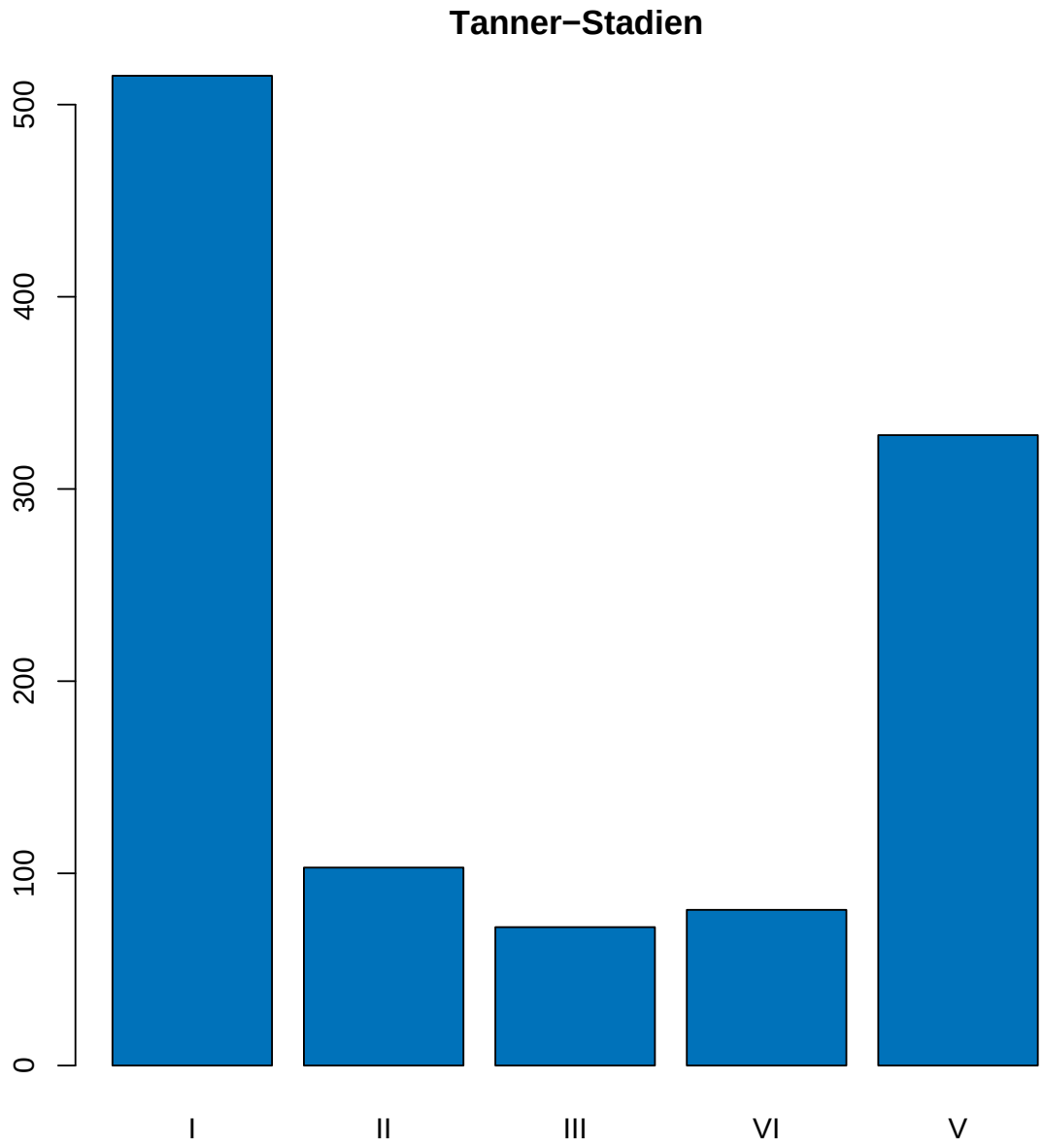
- Daneben stehen Kreisdiagramme zur Verfügung
- Diese werden mit `pie()` gezeichnet

```
> farben <- c(blau, "white", "red2", "yellow", "green2")
> pie(tab, main="Tanner-Stadien", cex=1.2, col=farben)
```

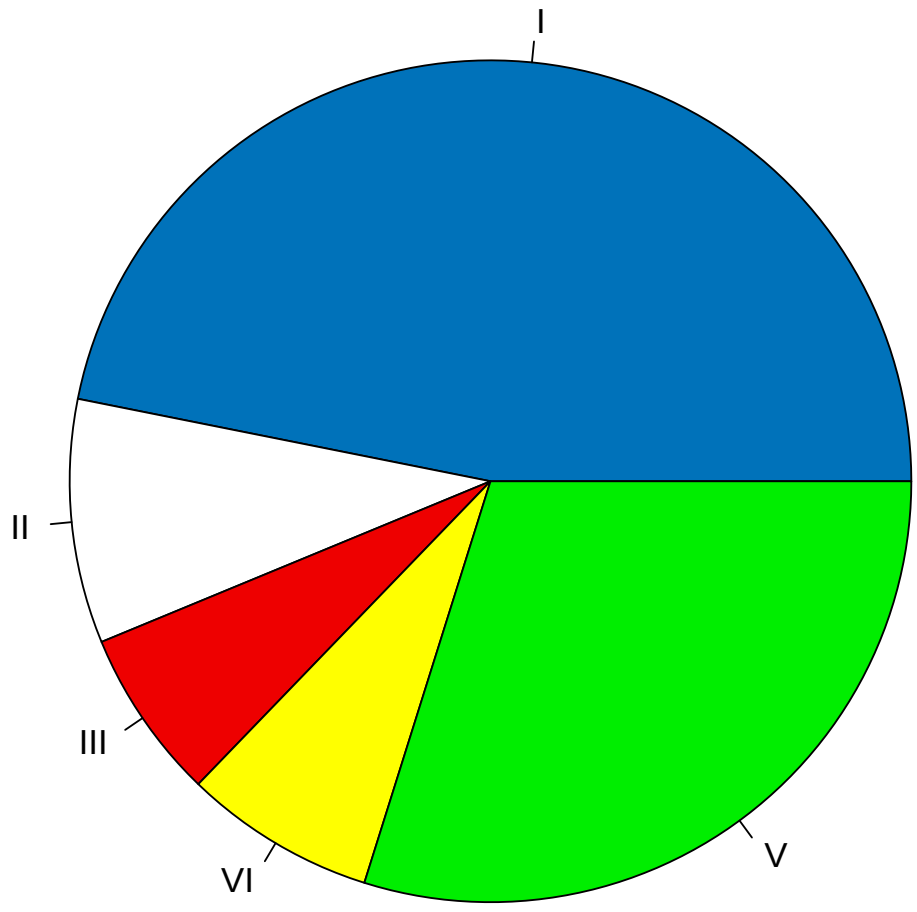
- Von der Verwendung eines Kreisdiagramms ist aber abzuraten, da Flächenunterschiede häufig schwer zu erkennen sind!
- Eine bessere Alternative stellen Dotcharts dar

```
> dotchart(tab, main="Tanner-Stadien", pch=16, cex=1.2)
```

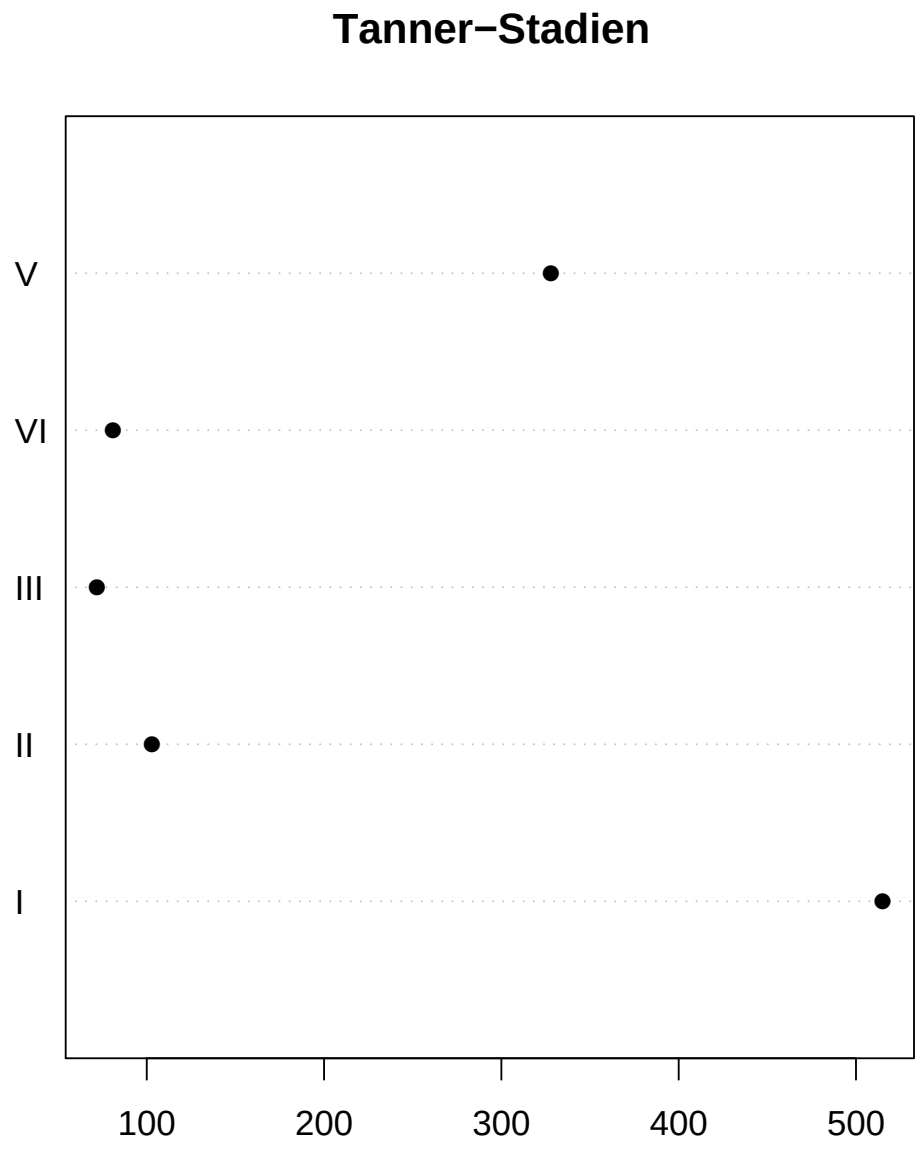
Säulendiagramm



Kreisdiagramm



Dotchart



Kategorielle Daten

- Bivariate kategorielle Daten können mit `table()` zu zweidimensionalen Kontingenztafeln zusammengefasst werden
- Die Funktion `addmargins()` ergänzt die Tabelle um ihre Ränder

```
> tab <- table(sex, tanner)
> addmargins(tab)
```

```
      tanner
sex      I   II  III  VI   V  Sum
  M    291   55   34  41 124  545
  F    224   48   38  40 204  554
Sum    515  103   72  81 328 1099
```

- Die Randverteilungen der Tabelle erhält man mit `margin.table()`

```
> margin.table(tab, margin=1)
```

```
sex
  M   F
545 554
```

Kategorielle Daten

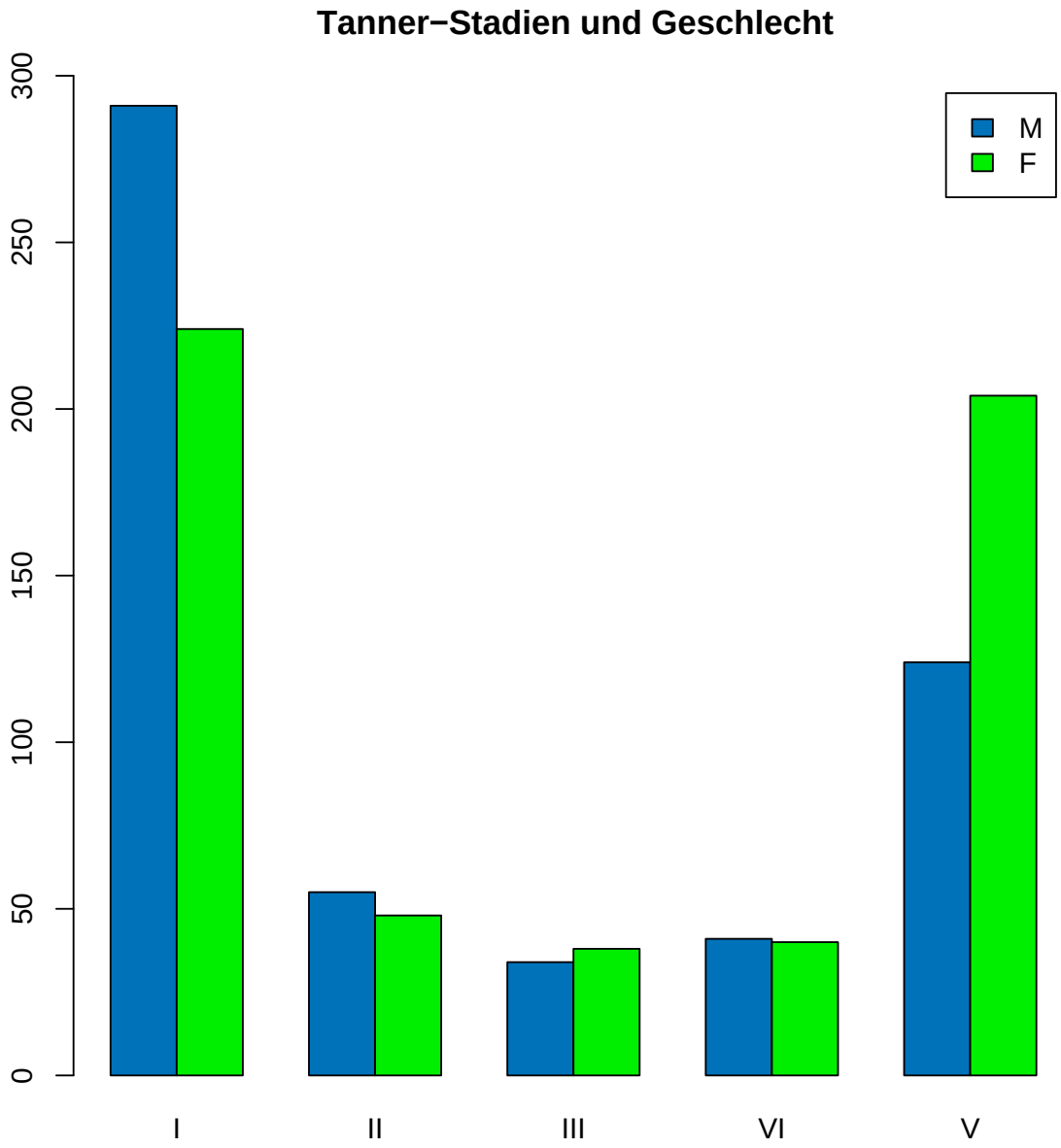
- Eine Tabelle der bedingten relativen Häufigkeiten wird mit dem Befehl `prop.table()` erzeugt

```
> prop.table(tab, margin=NULL)
     tanner
sex         I     II    III     VI      V
  M     0.26   0.05   0.03   0.04   0.11
  F     0.20   0.04   0.03   0.04   0.19
```

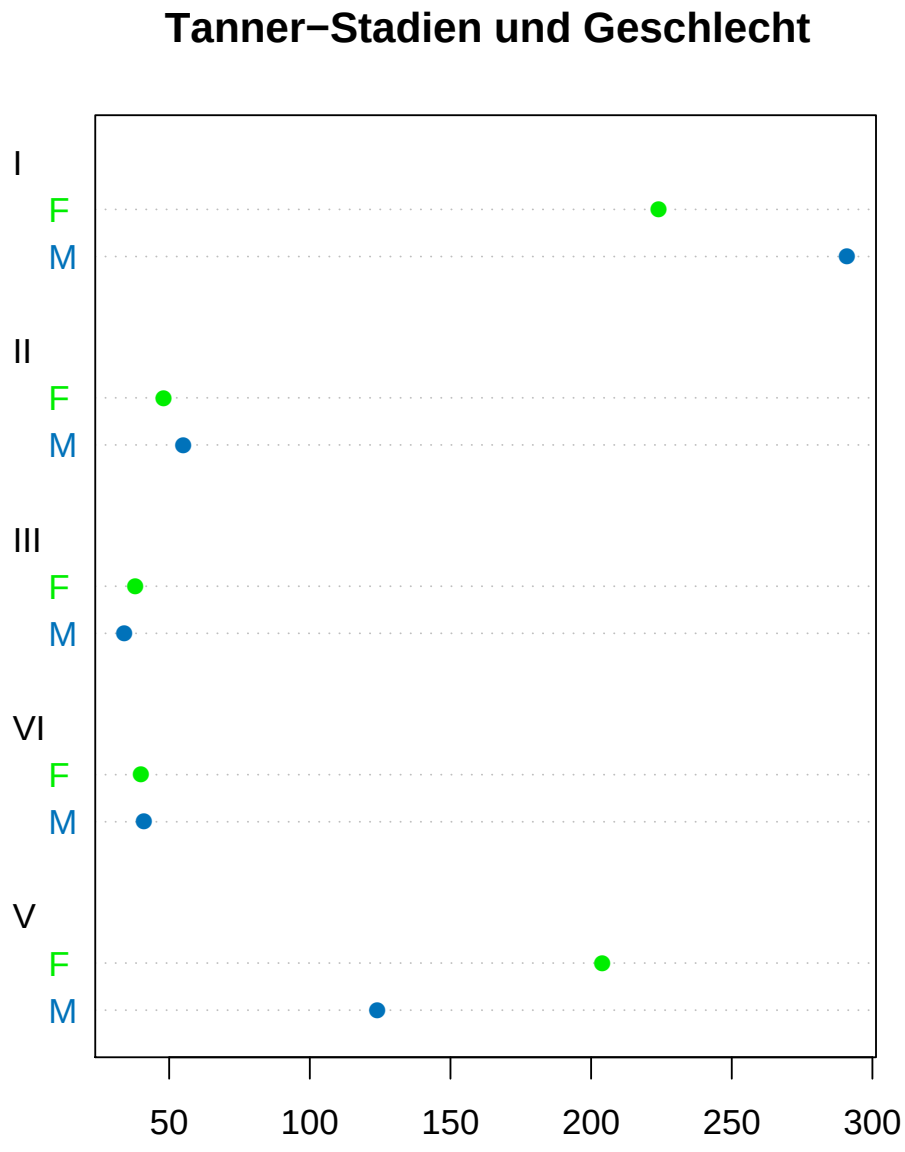
- Zur Veranschaulichung der Daten dienen wieder die Funktionen `barplot()` und `dotchart()`

```
> barplot(tab, beside=TRUE, ylim=c(0,300),
+   main="Tanner-Stadien und Geschlecht",
+   legend.text=TRUE, col=c(blau, "green2"))
> dotchart(tab, main="Tanner-Stadien und Geschlecht",
+   pch=16, cex=1.2, col=c(blau, "green2"))
```

Säulendiagramm



Dotchart


Tanner−Stadien und Geschlecht
I
F
M
II
F
M
III
F
M
VI
F
M
V
F
M
50
100
150
200
250
300

Kategorielle Daten

- Multivariate kategorielle Daten lassen sich allgemein mit `table()` zu n-dimensionalen Kontingenztafeln zusammenfassen
- Setzen wir das Merkmal `menarche` der Männer aus `juul` auf `"No"`

```
> detach(juul)
> men <- replace(juul$menarche, juul$sex=="M", "No")
> juul <- transform(juul, menarche=men)
> attach(juul)
```

- Die Funktion `ftable()` erzeugt eine flache Kontingenztafel

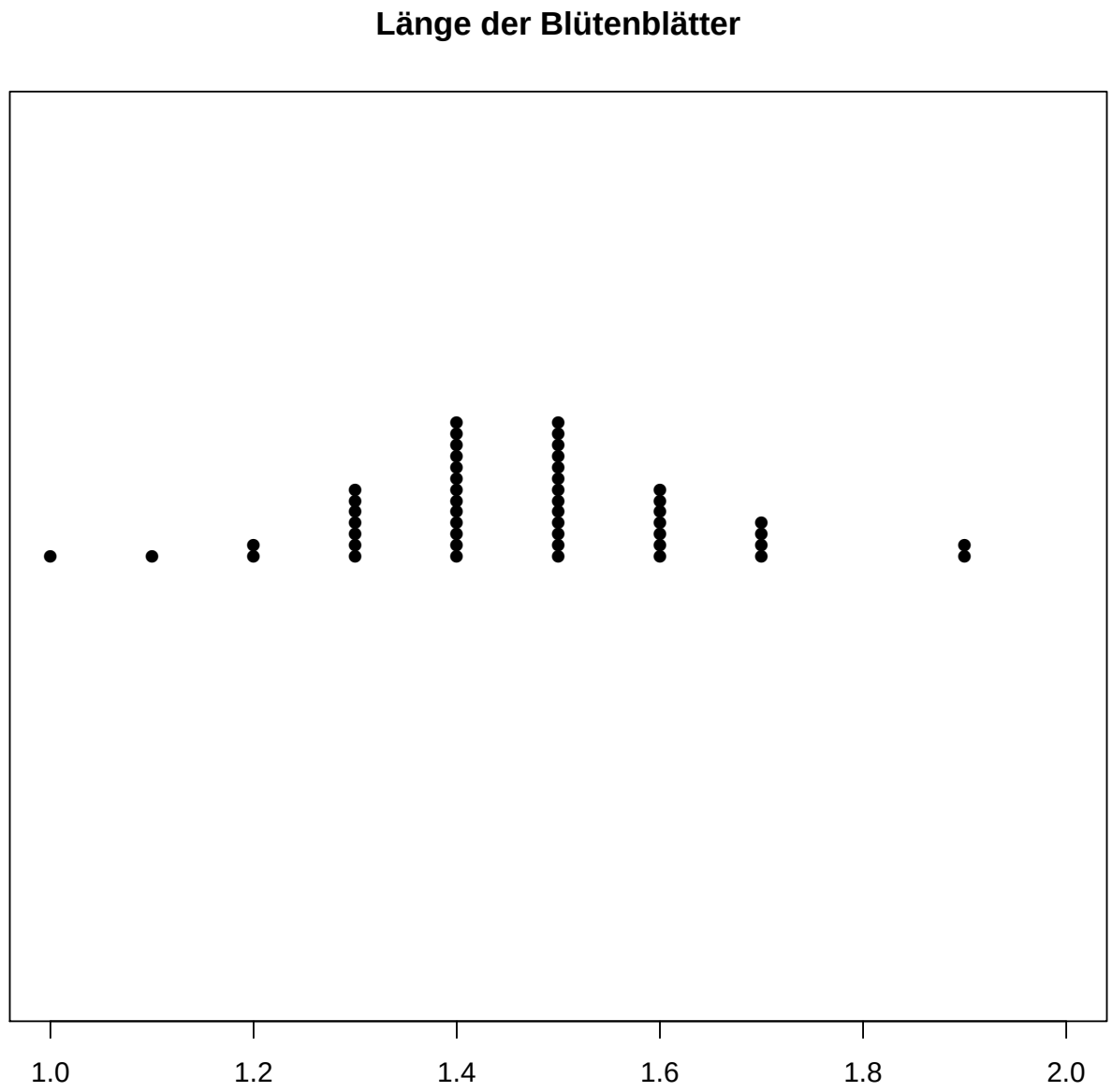
```
> ftable(sex, menarche, tanner)
                    tanner     I   II   III   VI     V
sex   menarche
M     No                     291   55    34   41   124
      Yes                      0    0     0    0     0
F     No                     221   43    32   14     2
      Yes                      1    1     5   26   202
```

Quantitative Daten

- Die Verteilung quantitativer Daten kann durch eine ganze Reihe von Kenngrößen und Grafiken beschrieben werden
- In einem Stripchart werden die Beobachtungswerte entlang einer Geraden aufgereiht
- Wir untersuchen im Folgenden wieder den `iris`-Datensatz aus R

```
> attach(iris)
> pet.len <- Petal.Length[Species=="setosa"]
> stripchart(pet.len, method="stack", xlim=c(1,2),
+   main="Länge der Blütenblätter", pch=16)
```

Stripchart



Anordnungen und Ränge

- Wir haben bereits die Funktionen `sort()` und `order()` zum Sortieren von Stichproben kennengelernt

```
> x <- c(5, 7, 2, 7, 8, 9)
> sort(x)
[1] 2 5 7 7 8 9
> order(x)
[1] 3 1 2 4 5 6
> x[order(x)]
[1] 2 5 7 7 8 9
```

- Der Rang eines Beobachtungswertes ist dessen Position in der sortierten Stichprobe

```
> rank(x)
[1] 2.0 3.5 1.0 3.5 5.0 6.0
```

- Treten einzelne Werte mehrfach auf, so wird deren Rang gemittelt

Anordnungen und Ränge

- Mehrfach auftretende Einträge identifiziert man mit `duplicated()`

```
> x <- c(5, 7, 2, 7, 8, 9)
> duplicated(x)
[1] FALSE FALSE FALSE TRUE FALSE FALSE
```

- Mit `unique()` werden die entsprechenden Duplikate entfernt

```
> unique(x)
[1] 5 7 2 8 9
```

- Kumulierte Summen und Produkte bestimmt man mit `cumsum()` und `cumprod()`

```
> cumsum(x) # 5, 5+7, 5+7+2, ...
[1] 5 12 14 21 29 38
> cumprod(x) # 5, 5*7, 5*7*2, ...
[1] 5 35 70 490 3920 35280
```

Empirische Verteilungsfunktion

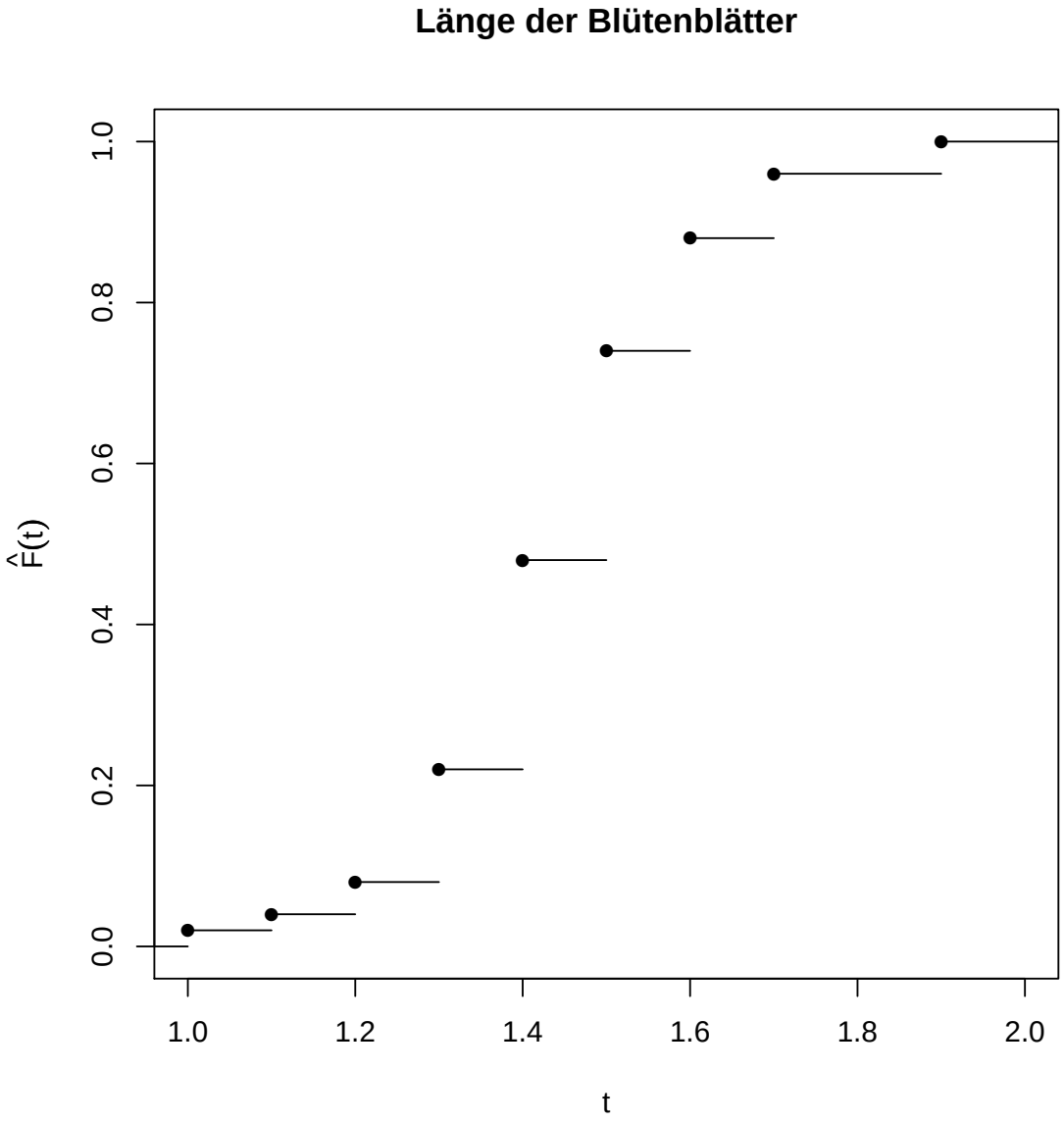
- Die empirische Verteilungsfunktion einer Stichprobe $x \in \mathbb{R}^n$ ist definiert als

$$\widehat{F}(t) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}_{\{x_i \leq t\}}$$

- Sie summiert die relativen Häufigkeiten der Beobachtungswerte bis zum Wert t auf
- Man bestimmt sie mit der Funktion `ecdf()` und zeichnet sie dann mit `plot()`

```
> pet.len <- Petal.Length[Species=="setosa"]
> evf <- ecdf(pet.len)
> plot(evf, xlim=c(1,2), col.01line=NULL, pch=16,
+   xlab="t", ylab=expression(hat(F)(t)),
+   main="Länge der Blütenblätter")
```

Empirische Verteilungsfunktion



Lagemaße

- Das arithmetische Mittel einer Stichprobe $x \in \mathbb{R}^n$ ist definiert als

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

- Man bestimmt es mit der Funktion `mean()`

```
> x <- c(5, 7, 2, 7, 8, 9)
> mean(x)
[1] 6.333333
```

Lagemaße

- Der Median ist der Mittelpunkt der sortierten Stichprobe
- Jeweils (höchstens) die Hälfte der Beobachtungswerte liegt links und rechts von ihm
- In R wird er mit der Funktion `median()` bestimmt

```
> x <- c(5, 7, 2, 7, 8, 9)
> median(x) # Teil der Stichprobe
[1] 7

> y <- 1:6
> median(y) # kein Teil der Stichprobe
[1] 3.5
```

Lagemaße

- Quantile sind eine Verallgemeinerung des Median-Konzepts
- Das p-Quantil ($p \in [0, 1]$) teilt eine sortierte Stichprobe derart, dass p der Beobachtungswerte links und $1 - p$ rechts von ihm liegen
- Die 0.25-, 0.5- und 0.75-Quantile nennt man auch Quartile
- Analog definiert man Quintile, Dezile und Perzentile
- Der Median ist das 0.5-Quantil
- Die Quantile einer Stichprobe erhält man mit `quantile()`

```
> x <- c(5, 7, 2, 7, 8, 9)
> quantile(x)
  0%   25%   50%   75%  100%
2.00  5.50  7.00  7.75  9.00
```

Lagemaße

- Andere p-Quantile stellt man mit dem Parameter `probs` ein

```
> x <- c(5, 7, 2, 7, 8, 9)
> p <- seq(0, 1, by=0.2)
> quantile(x, probs=p)
0%   20%   40%   60%   80%   100%
 2     5     7     7     8      9
```

- Eine Zusammenfassung der wichtigsten Maße erhält man mit dem Befehl `summary()`

```
> summary(x)
 Min.   1st Qu.   Median    Mean   3rd Qu.    Max.
2.000     5.500    7.000   6.333     7.750   9.000
```

Streuungsmaße

- Der Abstand zwischen dem kleinsten und dem größten Wert einer Stichprobe heißt Spannweite
- Die Funktion `range()` liefert das Minimum und das Maximum der beobachteten Werte

```
> x <- c(5, 7, 2, 7, 8, 9)
> range(x)
[1] 2 9
> diff(range(x)) # Spannweite
[1] 7
```

- Den Abstand zwischen dem 0.25- und dem 0.75-Quantil bezeichnet man als Interquartilsabstand
- Dieser wird mit der Funktion `IQR()` bestimmt

```
> IQR(x)
[1] 2.25
```

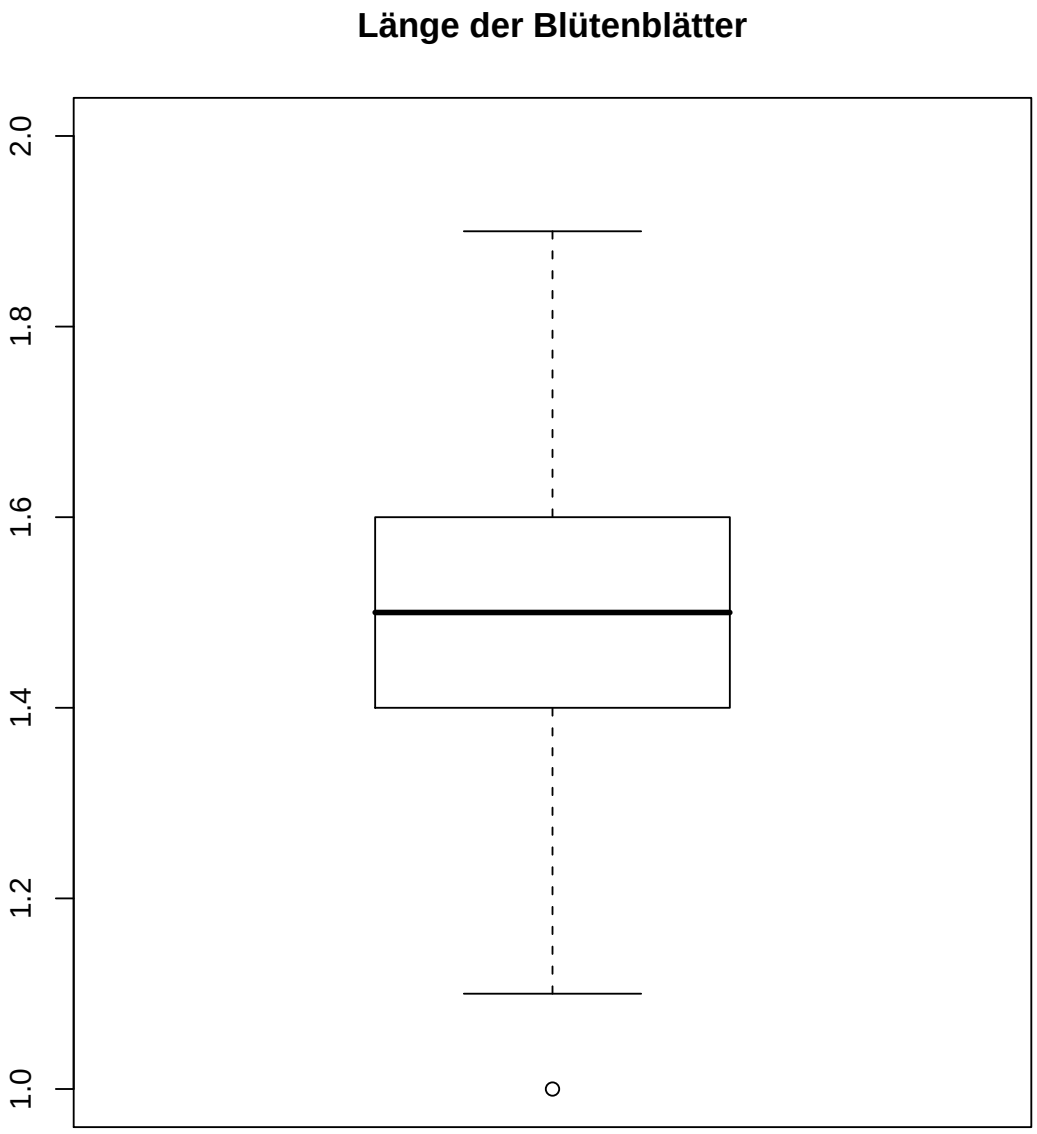
Boxplot

- Ein Boxplot fasst die Stichprobenverteilung grafisch zusammen

```
> pet.len <- Petal.Length[Species=="setosa"]
> summary(pet.len)
 Min.   1st Qu.   Median    Mean   3rd Qu.    Max.
1.000     1.400    1.500   1.462     1.575   1.900
> boxplot(pet.len, ylim=c(1,2),
+   main="Länge der Blütenblätter")
```

- Die Box veranschaulicht die 0.25-, 0.5- und 0.75-Quantile
- Die maximale Länge der beiden „Whisker“ beträgt jeweils den 1.5-fachen Interquartilsabstand
- Der kleinste und größte Beobachtungswert innerhalb dieses Bereichs gibt dann die tatsächliche Länge vor
- Alle Werte außerhalb dieser Grenzen werden als Ausreißer interpretiert und mit einem Punkt dargestellt

Boxplot



Streuungsmaße

- Die Varianz einer Stichprobe $x \in \mathbb{R}^n$ ist definiert als

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2$$

- Die Wurzel $\sqrt{s^2}$ der Varianz ist die Standarabweichung
- Man bestimmt sie mit den Funktionen `var()` und `sd()`

```
> x <- c(5, 7, 2, 7, 8, 9)
> var(x)
[1] 6.266667
> sd(x)
[1] 2.503331
> sqrt(var(x))
[1] 2.503331
```

Histogramm

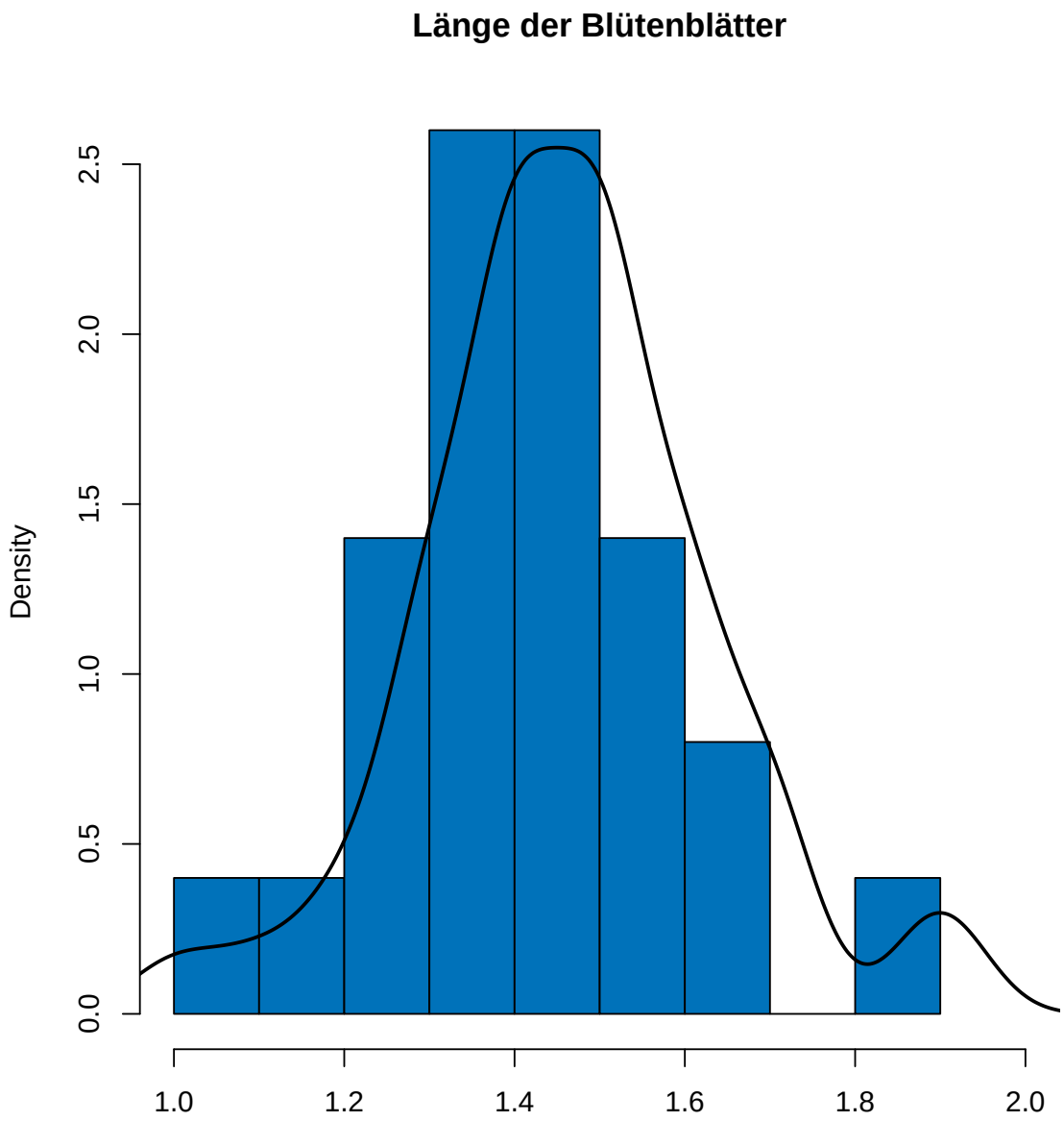
- Die Verteilung einer Stichprobe lässt sich durch Histogramme sehr gut veranschaulichen
- Diese werden mit der Funktion `hist()` erzeugt
- Man zerlegt dabei den Wertebereich der Beobachtungen in benachbarte Intervalle
- Die Anzahl und Länge dieser Intervalle lässt sich mit dem Parameter `breaks` regeln
- Die Höhe der gezeichneten Balken entspricht den absoluten Häufigkeiten der enthaltenen Beobachtungswerte
- Mit `probability=TRUE` werden stattdessen die relativen Häufigkeiten ausgegeben
- Diesen entspricht dann die Fläche der gezeichneten Balken

Histogramm

- Wählt man alle Intervalle gleich groß, so lässt sich auch die Balkenhöhe interpretieren
- In diesem Fall kann mit `density()` eine Dichtschätzung erzeugt und mit `lines()` hinzugefügt werden

```
> hist(pet.len, xlim=c(1,2), probability=TRUE,
+   main="Länge der Blütenblätter", xlab="", col=blau)
> den <- density(pet.len)
> lines(den, lwd=2)
```

Histogramm

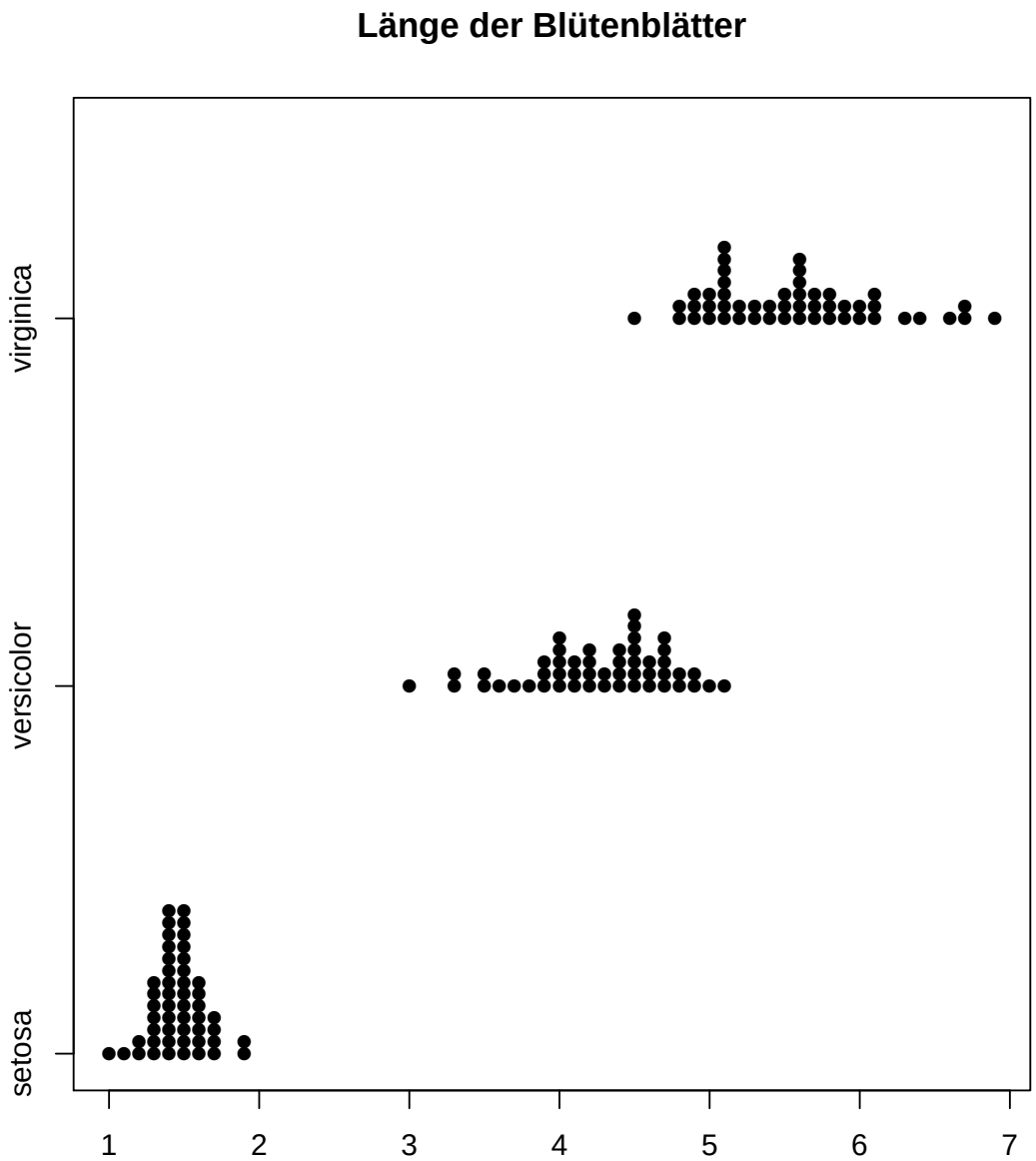

Länge der Blütenblätter
Density
0.0
0.5
1.0
1.5
2.0
2.5
1.0
1.2
1.4
1.6
1.8
2.0

Gruppierte Daten

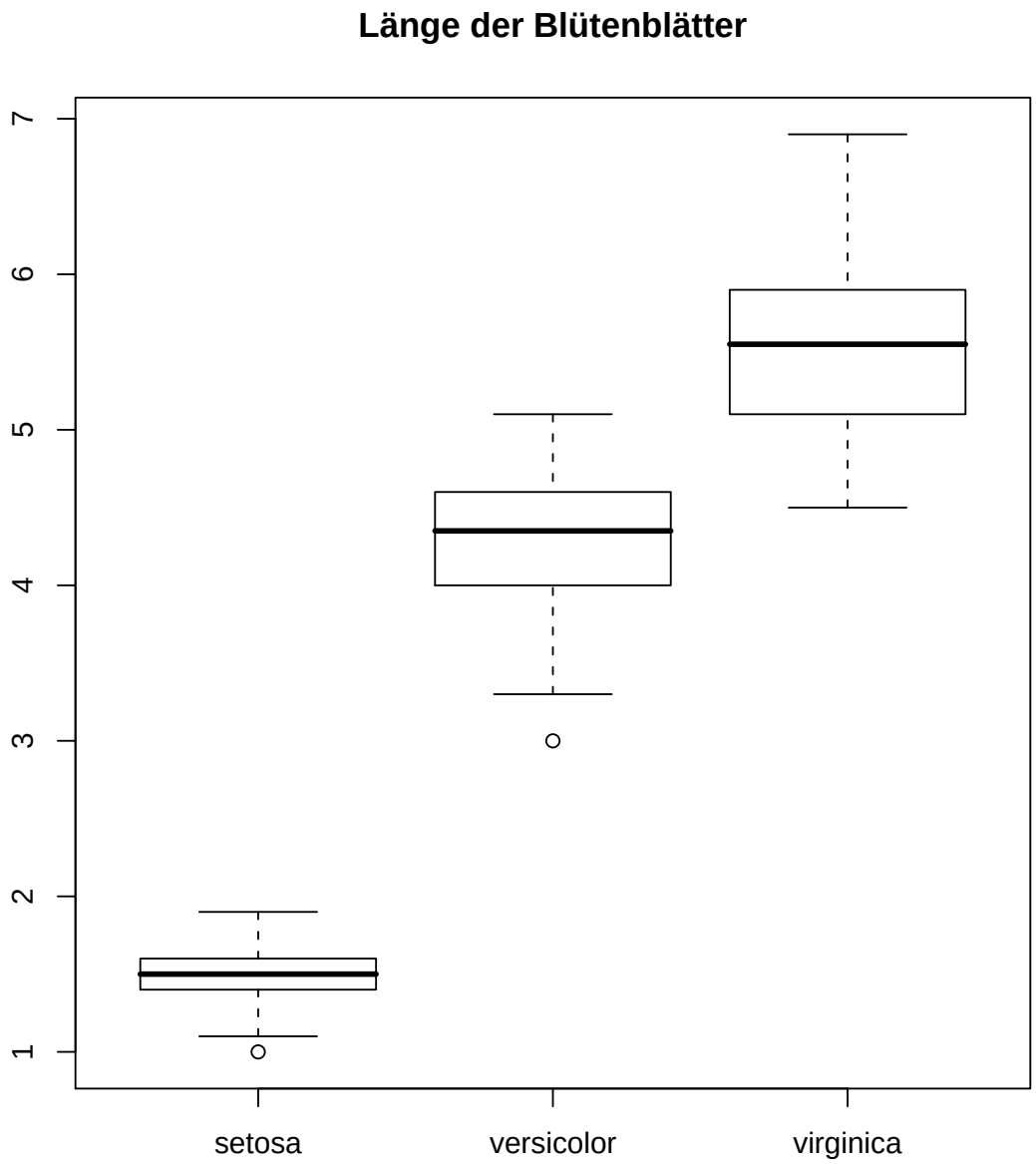
- Der `iris`-Datensatz ist bekanntlich nach `Species` gruppiert
- Die Unterschiede der Verteilungen zwischen den Gruppen wollen wir wie zuvor veranschaulichen
- Dabei ist häufig die Formelnotation `y~x` („`y` wird mit Hilfe von `x` beschrieben“) hilfreich

```
> stripchart(Petal.Length~Species, method="stack",
+   main="Länge der Blütenblätter", xlab="", pch=16)
> boxplot(Petal.Length~Species,
+   main="Länge der Blütenblätter")
```

Stripchart


Länge der Blütenblätter
virginica
versicolor
setosa
1
2
3
4
5
6
7

Boxplots


Länge der Blütenblätter
1
2
3
4
5
6
7
setosa
versicolor
virginica

Beispiel `diamond`-Datensatz

Die `diamond`-Daten aus dem `UsingR`-Paket enthalten die Preise von 48 Diamantringen und das Gewicht des jeweiligen Diamanten.

```
> library(UsingR)
> ?diamond
> diamond
> str(diamond)
> summary(diamond)
> attach(diamond)
```

Q-Q-Plot

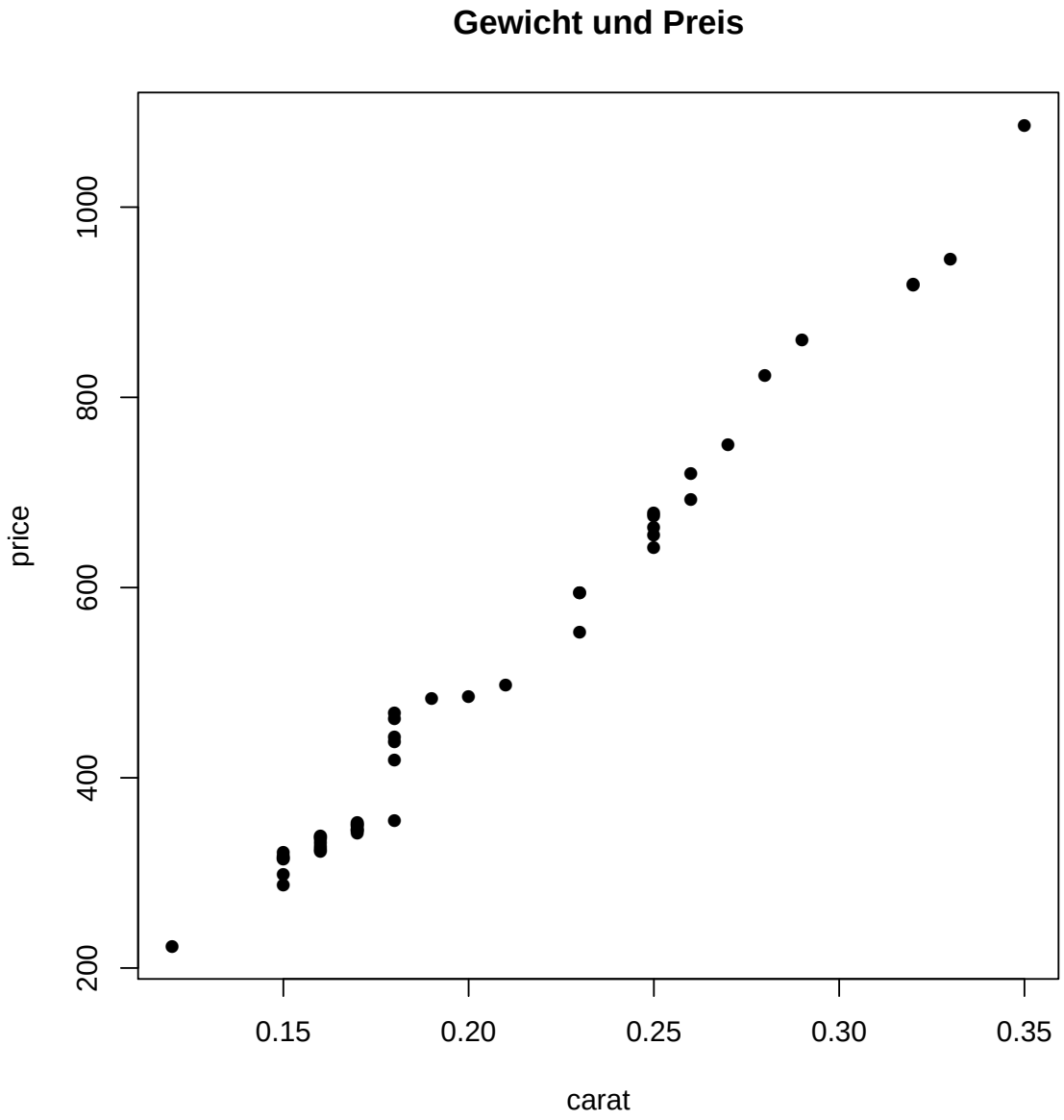
- Ein Q-Q-Plot vergleicht die Verteilungen zweier Merkmale, indem er die empirischen Quantile gegeneinander abträgt
- Liegen die gezeichneten Punkte auf einer Geraden, so deutet dies auf eine starke Ähnlichkeit hin
- Q-Q-Plots werden mit der Funktion `qqplot()` erzeugt

```
> qqplot(carat, price, pch=16, main="Gewicht und Preis")
```

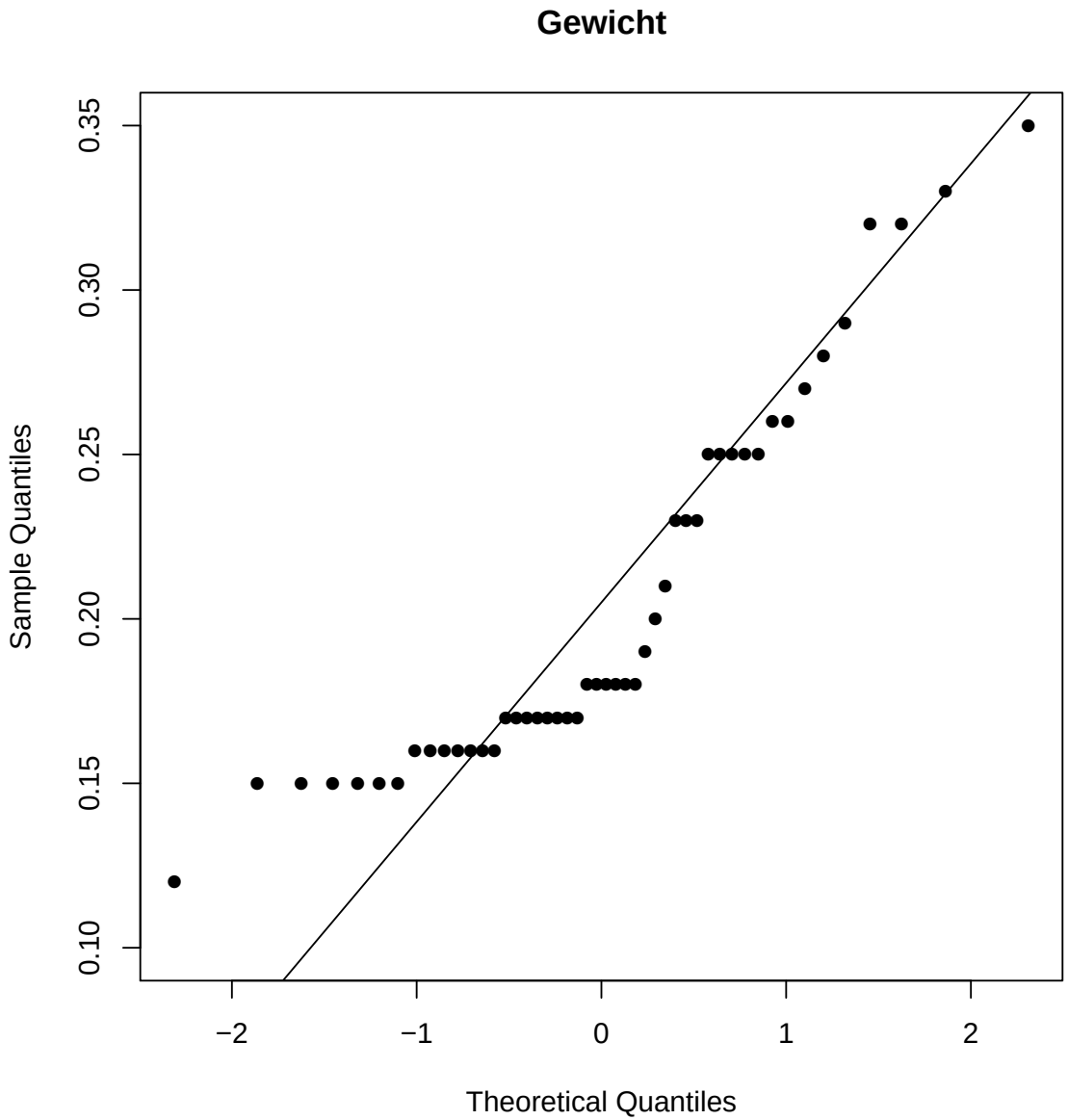
- Gewicht und Preis eines Diamanten sind offensichtlich ähnlich verteilt
- Mit `qqnorm()` können die empirischen Quantile gegen die Quantile einer Normalverteilung abgetragen werden

```
> qqnorm(carat, ylim=c(0.1,0.35), pch=16,
+   main="Normal Q-Q-Plot des Gewichts")
> qqline(carat)
```

Q-Q-Plot


Gewicht und Preis
price
carat
200
400
600
800
1000
0.15
0.20
0.25
0.30
0.35

Normal Q-Q-Plot

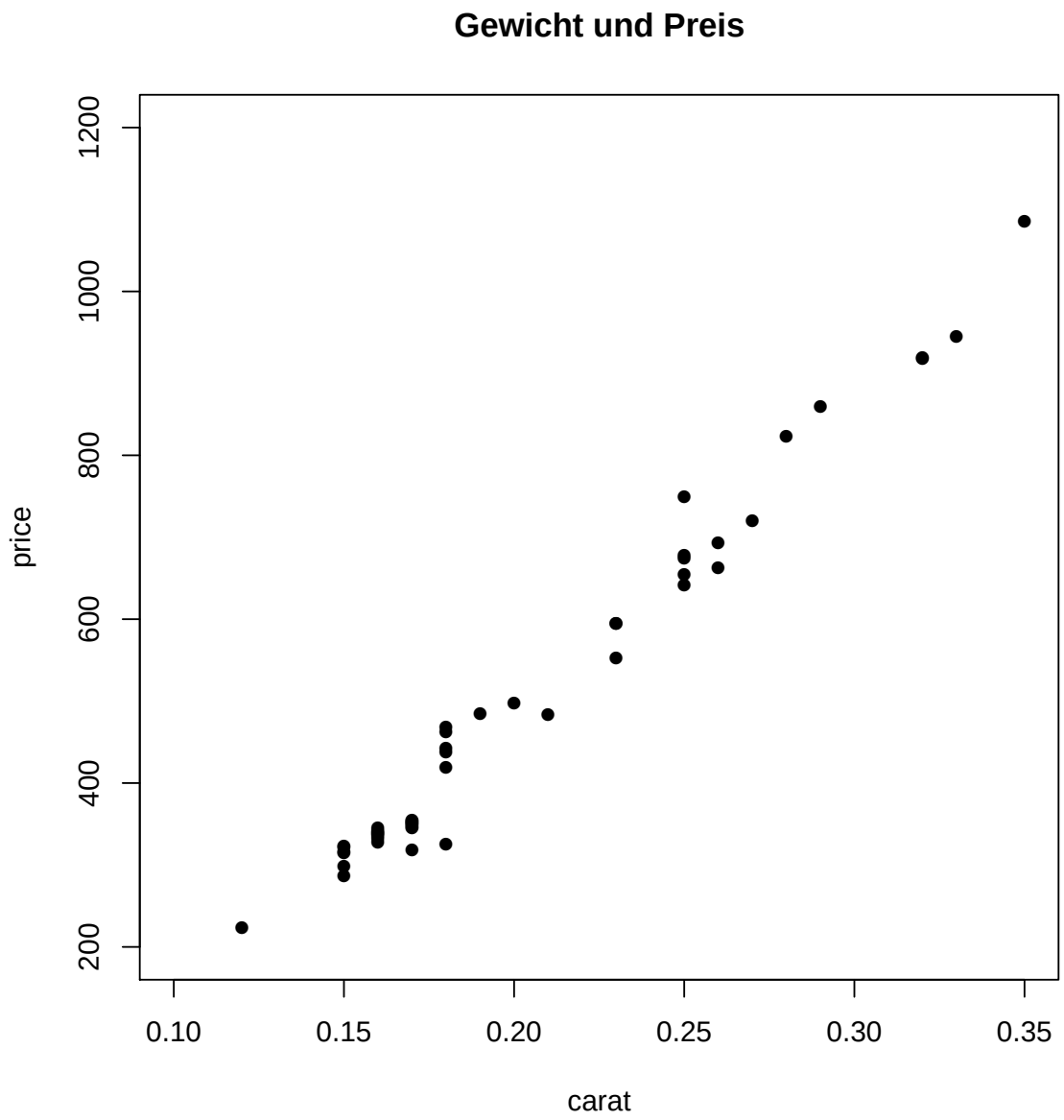


Streudiagramm

- Korrelationen zwischen Variablen können mit Streudiagrammen erkannt werden
- Dazu werden einfach die Beobachtungswerte zweier Variablen mit `plot()` gegeneinander abgetragen
- Liegen die Punkte annähernd auf einer Geraden, so besteht ein linearer Zusammenhang

```
> plot(price~carat, pch=16, xlim=c(0.1,0.35),
+   ylim=c(200,1200), main="Gewicht und Preis")
```

Streudiagramm



Zusammenhangsmaße

- Die Korrelation ist ein Maß für die Stärke und Richtung des Zusammenhangs zweier Variablen
- Der Pearson-Korrelationskoeffizient zweier Stichproben $x, y \in \mathbb{R}^n$ ist definiert als

$$r = \frac{\sum_{i=1}^{n} (x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \overline{x})^2 \sum_{i=1}^{n} (y_i - \overline{y})^2}}$$

- $r = 1$ deutet auf einen starken positiven, $r = -1$ auf einen starken negativen Zusammenhang hin
- $r = 0$ lässt auf keinen linearen Zusammenhang schließen
- Die Pearson-Korrelation lässt sich mit `cor()` bestimmen

```
> cor(carat, price)
[1] 0.9890707
```

Lineare Regression

- Da die `diamond`-Daten stark positiv korreliert sind, wollen wir nun eine Gerade durch die Punkte des Streudiagramms legen
- Diese soll die Summe der quadratischen Abstände zu den einzelnen Punkten minimieren
- Als Lösung erhält man die Gerade

$$y = \beta_0 + \beta_1 x$$

mit

$$\beta_1 = \frac{\sum (\texttt{carat[i]} - \overline{\texttt{carat}}) \left(\texttt{price[i]} - \overline{\texttt{price}}\right)}{\sum (\texttt{carat[i]} - \overline{\texttt{carat}})^2}$$

und

$$\beta_0 = \overline{\texttt{price}} - \beta_1 \times \overline{\texttt{carat}}$$

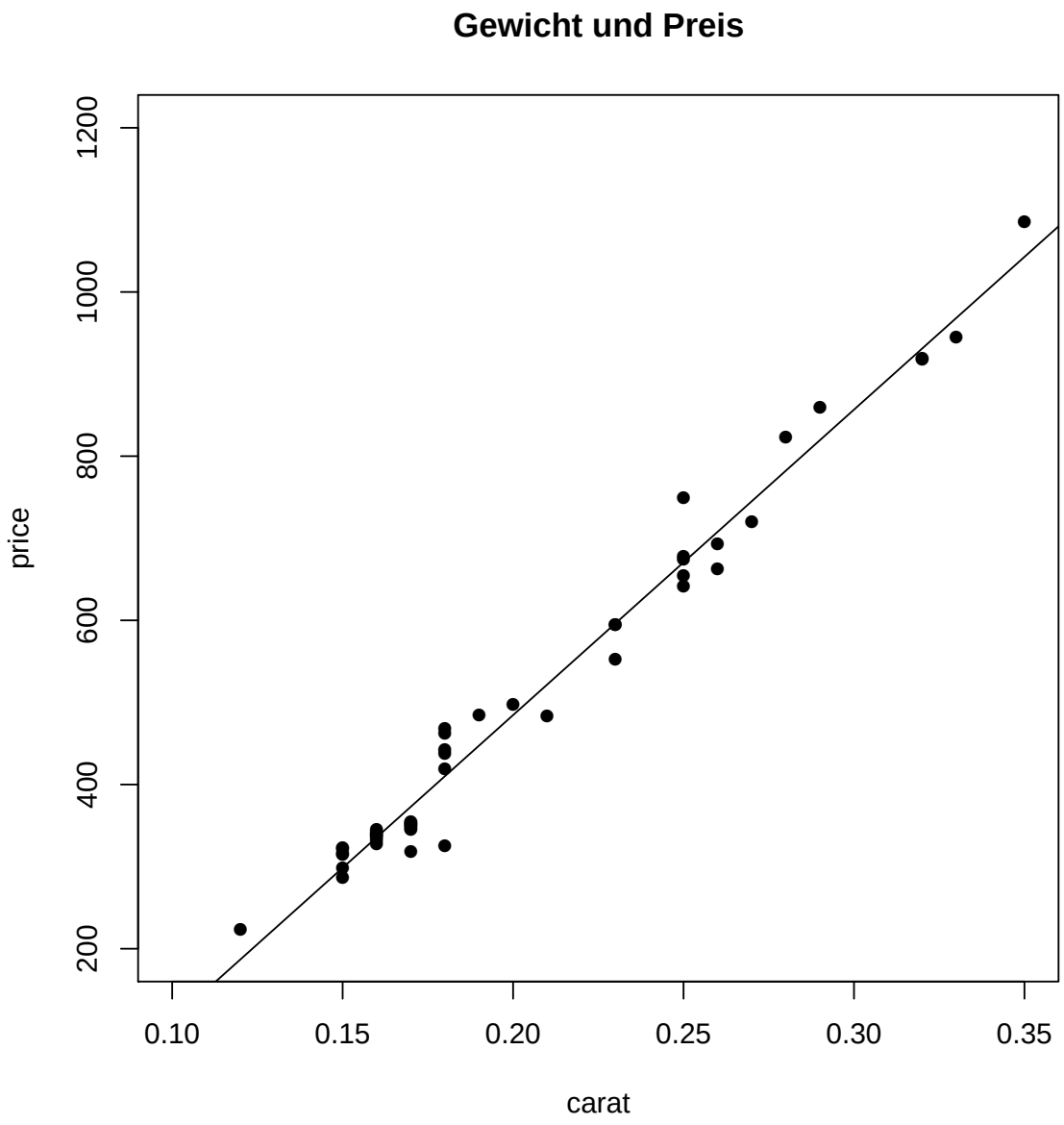
- Genaueres erfährt man in den Vorlesungen zu linearen Modellen

Lineare Regression

- Die gesuchten Koeffizienten erhält man mit der Funktion `lm()`
- Mit `abline()` können wir die Gerade einzeichnen

```
> (reg <- lm(price~carat))
> summary(mod)
> plot(price~carat, pch=16, xlim=c(0.1,0.35),
+   ylim=c(200,1200), main="Gewicht und Preis")
> abline(reg$coefficients)
```

Streudiagramm



Zusammenhangsmaße

- Geht man von einem nicht zwangsläufig linearen Zusammenhang aus, so bieten sich Rangkorrelationskoeffizienten an
- Den Spearman-Rangkorrelationskoeffizienten ρ erhält man, indem man in der Definition von r sämtliche Beobachtungswerte durch ihre Ränge ersetzt
- Er wird mit `method="spearman"` in `cor()` gesetzt

```
> cor(carat, price, method="spearman")
[1] 0.9656018

> cor(rank(carat), rank(price))
[1] 0.9656018
```

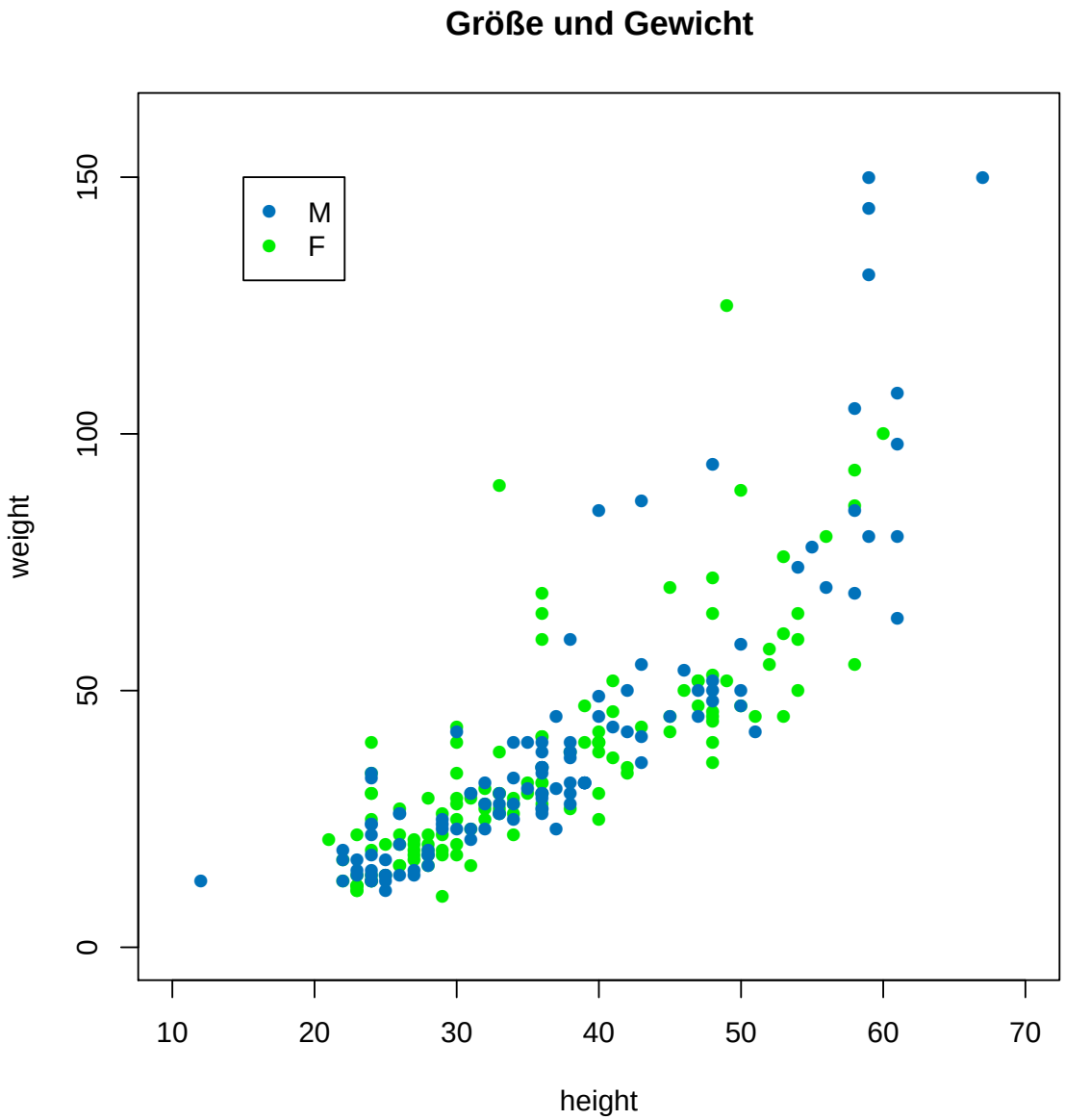
Beispiel `kid.weights`-Datensatz

Die `kid.weights`-Daten aus dem `UsingR`-Paket enthalten Alter, Gewicht, Größe und Geschlecht von 250 amerikanischen Kindern.

```
> library(UsingR)
> ?kid.weights
> kid.weights
> str(kid.weights)
> summary(kid.weights)
> with(kid.weights, cor(weight, height))
[1] 0.8237564
> with(kid.weights, cor(weight, height, m="spearman"))
[1] 0.8822136
```

Es besteht ein positiver Zusammenhang zwischen `weight` und `height`. Dieser scheint allerdings nicht unbedingt linear zu sein.

Beispiel `kid.weights`-Datensatz



Lage-, Streuungs- und Zusammenhangsmaße

Funktion	Beschreibung
`mean()`	arithmetisches Mittel
`median()`	Median
`quantile()`	Quantile
`summary()`	Zusammenfassung eines Objekts
`range()`	Minimum und Maximum (Spannweite)
`IQR()`	Interquartilsabstand
`mad()`	mittlere absolute Abweichung
`var()`	Varianz
`sd()`	Standardabweichung
`cov()`	Kovarianz
`cor()`	Korrelationskoeffizient

Multivariate Daten

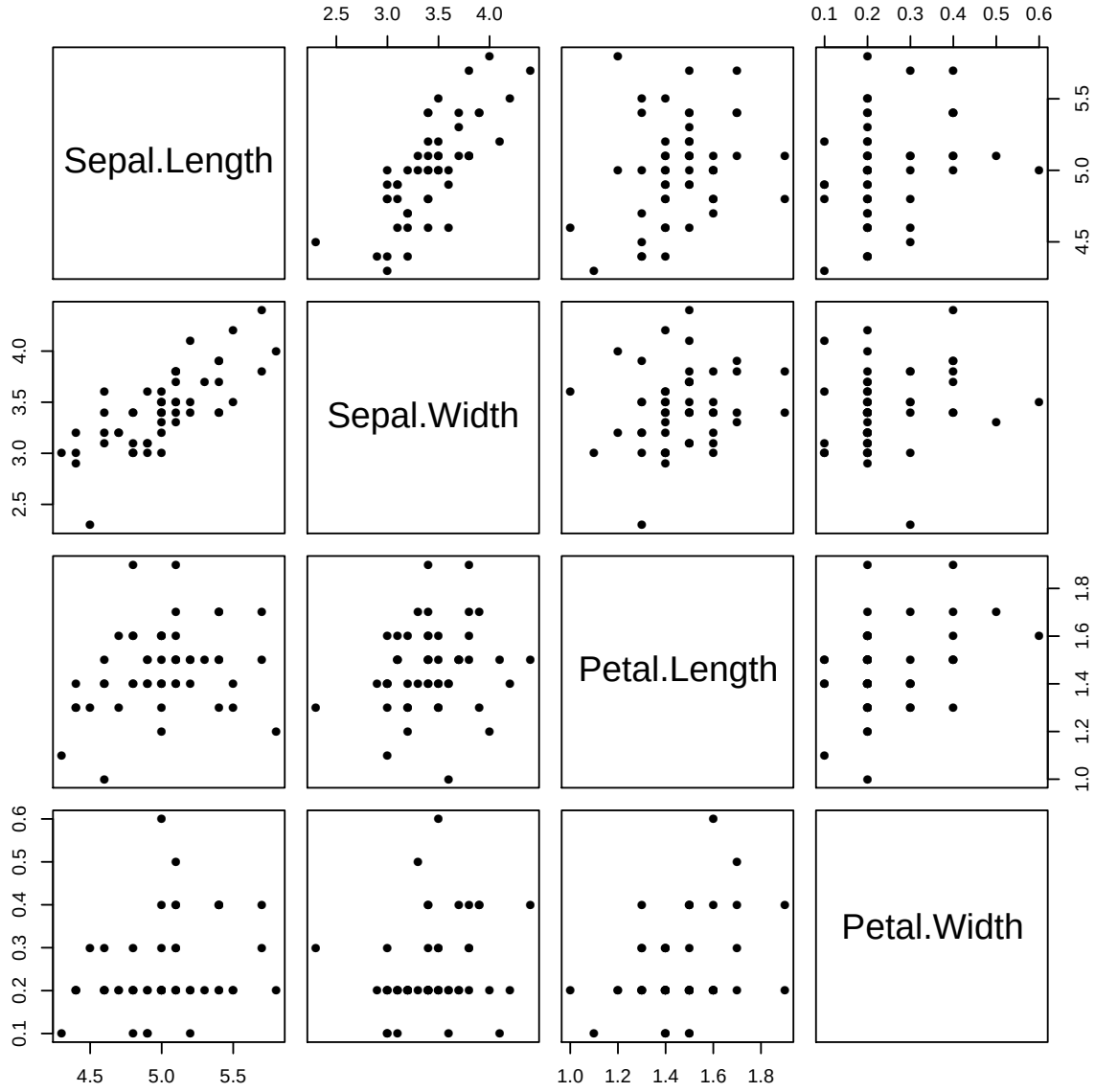
- Eine Streudiagramm-Matrix mehrerer Variablen kann mit `pairs()` erstellt werden

```
> detach(iris)
> iris <- subset(iris, Species=="setosa")
> attach(iris)
> pairs(iris[,-5], pch=16)
```

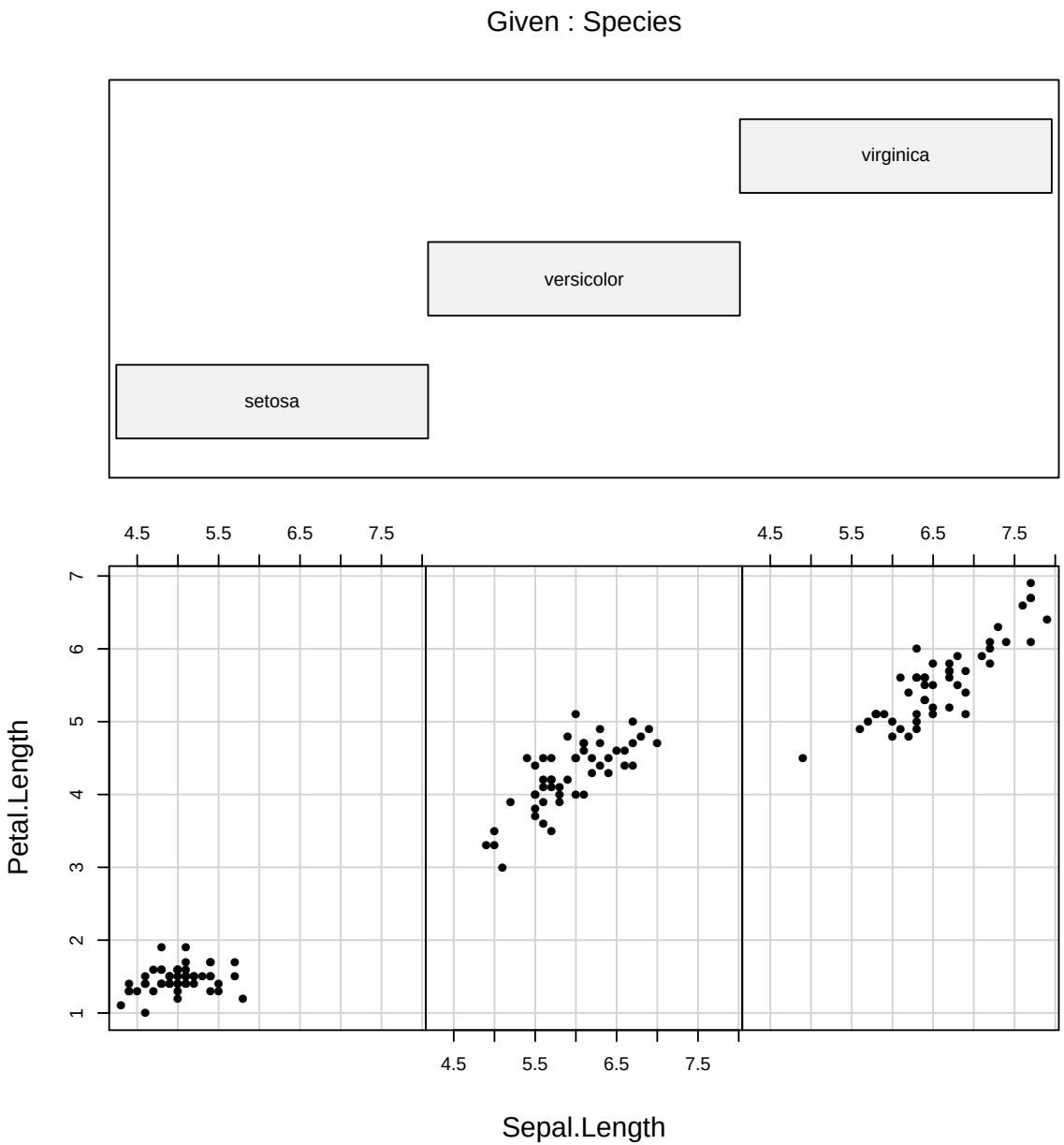
- Für gruppierte Daten eignet sich auch ein Conditioning Plot, der zwei Variablen bedingt an eine dritte grafisch darstellt
- Er wird mit der Funktion `coplot()` erzeugt

```
> detach(iris)
> iris <- datasets::iris
> attach(iris)
> coplot(Petal.Length~Sepal.Length | Species,
+   columns=3, pch=16)
```

Streudiagramm-Matrix



Conditioning Plot



High-level Grafikfunktionen

Funktion	Beschreibung
`plot()`	generische Grafikfunktion
`barplot()`	Säulendiagramm
`boxplot()`	Boxplot
`contour()`	Höhenlinien-Plot
`coplot()`	Conditioning Plot
`curve()`	Kurven
`dotchart()`	Dotchart
`hist()`	Histogramm
`image()`	Bilder (3. Dimension als Farbe)
`mosaicplot()`	Mosaikplot
`pairs()`	Streudiagramm-Matrix
`persp()`	perspektivische Flächen
`qqplot()`	Q-Q-Plot
`stripchart()`	Stripchart

Diskretisierung

- Zur Diskretisierung eines numerischen Vektors kann die Funktion `cut()` verwendet werden, die ein Objekt der Klasse `factor` erzeugt
- Die Klassengrenzen werden mit dem Parameter `breaks` gesetzt

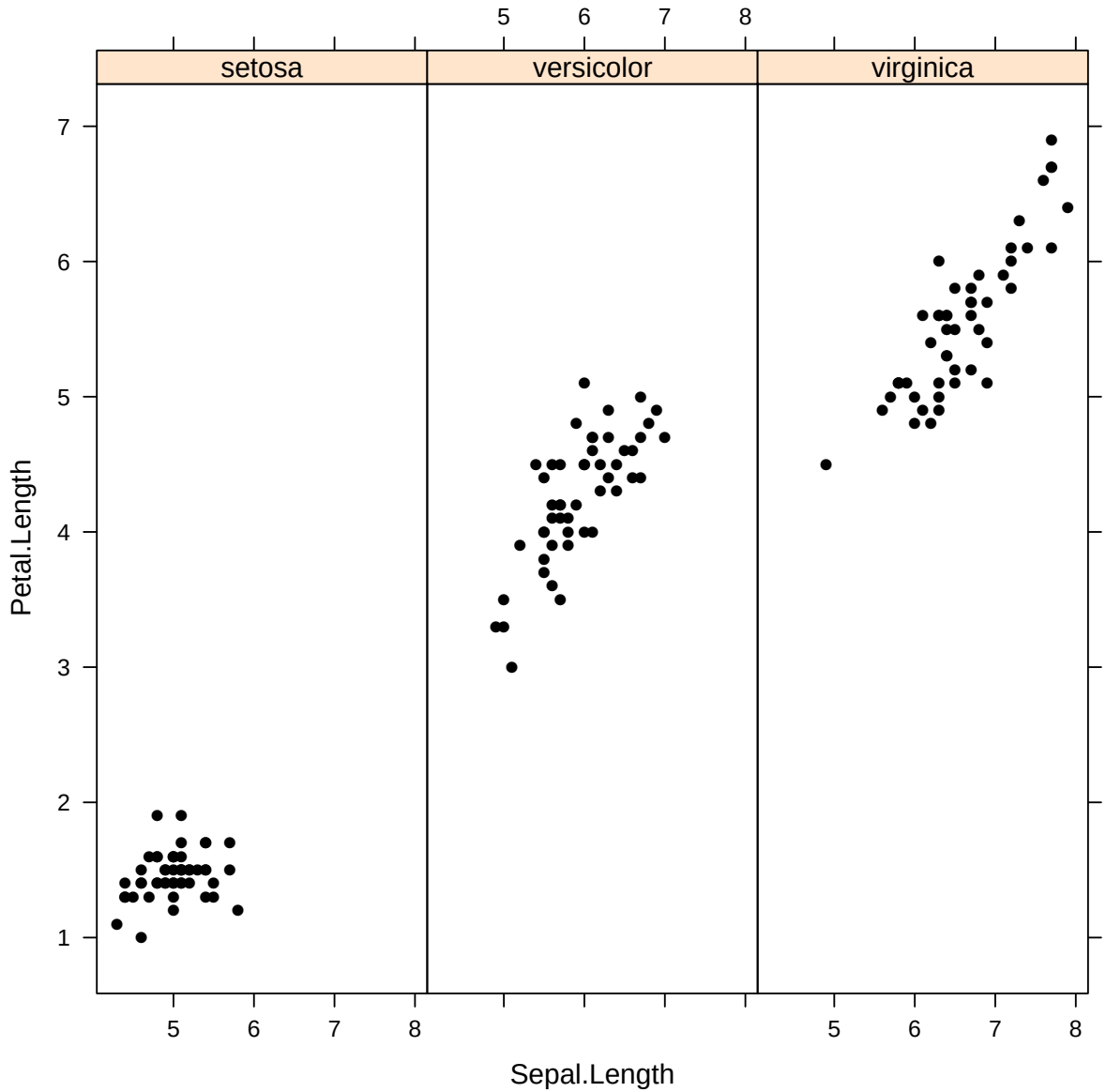
```
> juul
> alter <- cut(age, breaks=seq(0, 100, by=20))
> summary(alter)
(0,20]   (20,40]   (40,60]   (60,80]   (80,100]   NA's
  1187        78        52        13          4      5
```

Trellis-Grafik

- Trellis-Grafiken werden durch das Paket `lattice` implementiert
- Sie können sehr einfach viele Grafiken desselben Typs in einem Gitter darstellen
- Das Trellis-Device wird mit `trellis.device()` gestartet
- Mit dem Parameter `device` lässt sich ein Ausgabegerät einstellen
- Die Trellis-Grafikfunktionen verwenden die Formelnotation `y~x | z`

```
> library(lattice)
> trellis.device("pdf", file="trellis.pdf")
> xyplot(Petal.Length~Sepal.Length | Species,
+   layout=c(3,1), col="black", pch=16)
> dev.off()
```

Trellis-Grafik



Trellis-Grafik

- Trellis-Objekte werden vor dem Zeichnen komplett generiert
- Bei nicht interaktiven Aufrufen (wie in Funktionen) müssen sie deshalb mit `print()` ausgegeben werden

```
> ausgabe <- function(){
+  grafik <- xyplot(Petal.Length~Sepal.Length|Species,
+    layout=c(3,1), col="black", pch=16)
+  print(grafik)
+ }
> ausgabe()
```

Trellis-Grafik

Funktion	Beschreibung
`barchart()`	Säulendiagramme
`bwplot()`	Boxplots
`cloud()`	3D-Punktewolken
`contourplot()`	Höhenlinien-Plots
`densityplot()`	Dichten
`dotplot()`	Dotplots
`histogramm()`	Histogramme
`levelplot()`	Levelplots
`piechart()`	Kuchendiagramme
`qq()`	Q-Q-Plots
`splom()`	Streudiagramm-Matrix
`wireframe()`	3D-Flächen
`xyplot()`	generische Grafikfunktion

Zufallszahlen

Beim Ziehen von Stichproben und bei Simulationen werden Pseudo-Zufallszahlen benötigt. Für diese gilt:

- Sie werden von Pseudo-Zufallszahlen-Generatoren erzeugt
- Sie sollten (fast) keine Regelmäßigkeiten enthalten
- Sie sollten möglichst schnell erzeugt werden können
- Sie sollten reproduzierbar sein (Wiederholung von Simulationen)

Der Standard-Zufallszahlen-Generator in R heißt Mersenne-Twister. Weitere Generatoren können mit `RNGkind()` gewählt werden.

Stichproben

- Zum Ziehen von Stichproben kann man die Funktion `sample(x, size, replace=FALSE, prob=NULL)` verwenden
- Aus dem Vektor `x` wird eine Stichprobe der Länge `size` ohne Zurücklegen (`replace=FALSE`) gezogen
- Mit `prob` können unterschiedliche Auswahlwahrscheinlichkeiten für die einzelnen Elemente aus `x` spezifiziert werden
- Reproduzierbare Ergebnisse erhält man, indem man mit `set.seed()` einen Startwert für den Zufallszahlen-Generator setzt

```
> set.seed(135)
> sample(1:10, 4) # ohne Zurücklegen
[1] 4 1 3 8
> sample(1:10, 4, replace=TRUE) # mit Zurücklegen
[1] 4 1 4 5
```

Verteilungen

Zur Berechnung von Dichte- und Verteilungsfunktion, Quantilen und Zufallszahlen gängiger Verteilungen sind vier Typen von Funktionen implementiert, denen jeweils derselbe Buchstabe vorangeht:

- `d` (`density`) Dichtefunktion
- `p` (`probability`) Verteilungsfunktion
- `q` (`quantiles`) Quantile
- `r` (`random`) Pseudo-Zufallszahlen

Auf diese Buchstaben folgt die Bezeichnung der Verteilung, z.B. `norm` für die Normalverteilung.

```
> set.seed(123)
> rnorm(n=3) # Standardnormalverteilung
[1] -0.56047565 -0.23017749 1.55870831
```

Verteilungen

Funktion	Verteilung
`_beta()`	Beta-
`_binom()`	Binomial-
`_chisq()`	χ^2-
`_exp()`	Exponential-
`_f()`	F-
`_gamma()`	Gamma-
`_geom()`	Geometrische-
`_hyper()`	Hypergeometrische-
`_lnorm()`	Lognormal-
`_nbin()`	negative Binomial-
`_norm()`	Normal-
`_pois()`	Poisson-
`_t()`	t-
`_unif()`	Gleich-

Diskrete Verteilungen

Die Verteilung einer diskreten Zufallsvariable X wird durch ihre Zähldichte

$$f(x) = \mathbb{P}(X = x)$$

oder durch ihre Verteilungsfunktion

$$F(x) = \mathbb{P}(X \leq x)$$

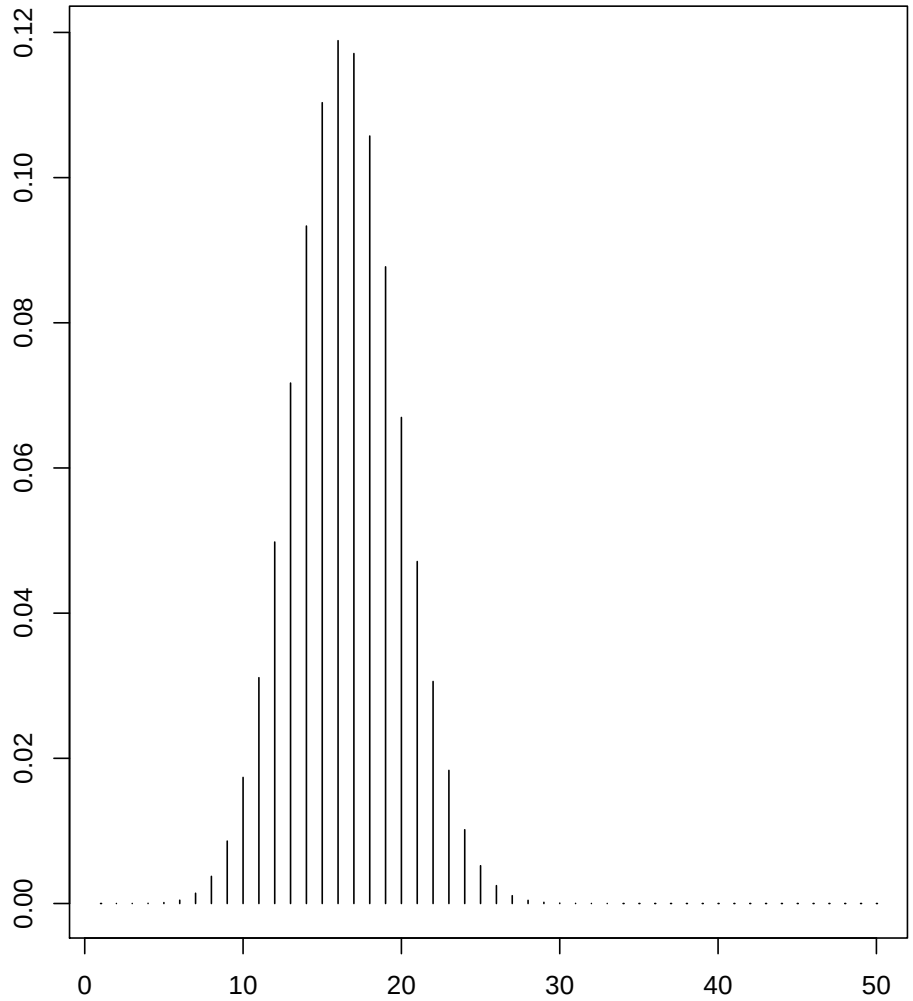
beschrieben.

Beispiel: Binomialverteilung mit Anzahl n und Wahrscheinlichkeit p.

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}, \quad x = 0, \ldots, n.$$

Diskrete Verteilungen

Zähldichte der Binomialverteilung mit Parametern $n = 50$ und $p = \frac{1}{3}$



Binomialkoeffizienten

- Binomialkoeffizienten lassen sich mit `choose()` berechnen

```
> choose(5, 3)
[1] 10
```

- Fakultäten erhält man mit `factorial()`

```
> factorial(5) / (factorial(3) * factorial(2))
[1] 10
```

- Wegen $n! = \Gamma(n+1)$ kann auch `gamma()` verwendet werden

```
> gamma(6) / (gamma(4) * gamma(3))
[1] 10
```

Stetige Verteilungen

Die Verteilung einer stetigen Zufallsvariable wird durch ihre Dichte $f(x)$ oder durch ihre Verteilungsfunktion

$$F(x) = \int_{-\infty}^{x} f(x)\, dx$$

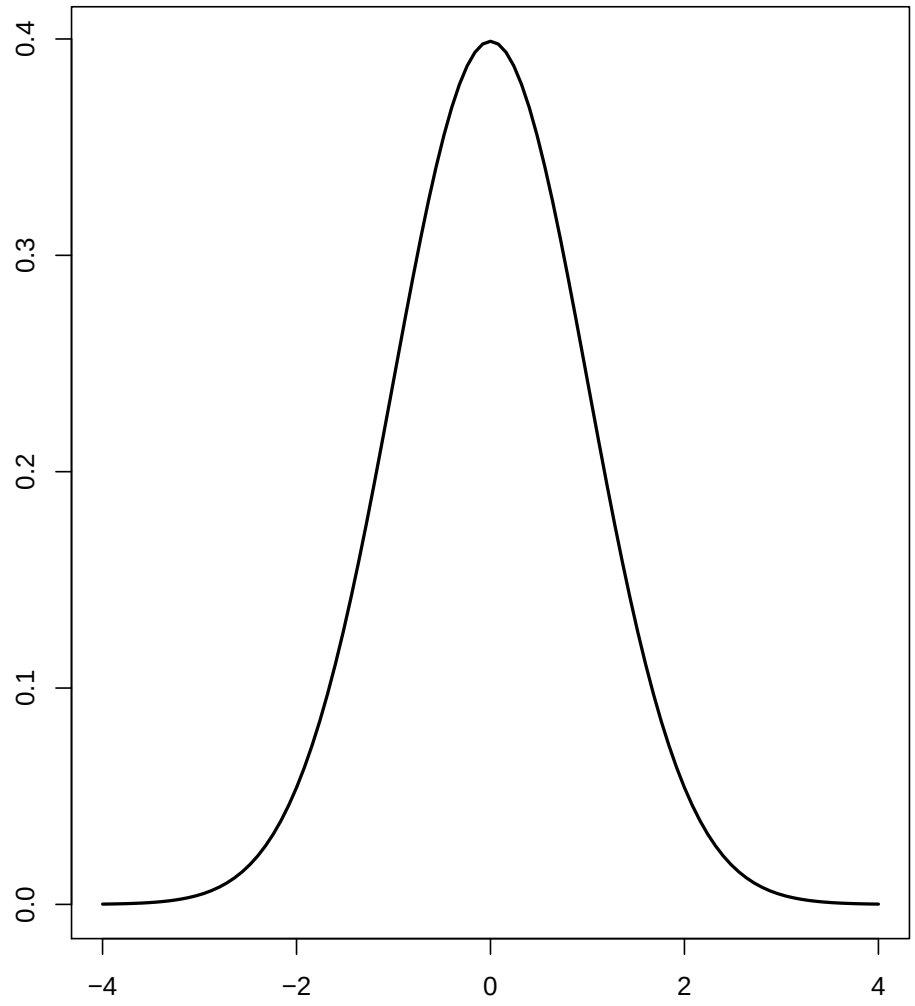
beschrieben.

Beispiel: Normalverteilung mit Erwartungswert μ und Varianz σ^2.

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \quad x \in \mathbb{R}.$$

Stetige Verteilungen

Dichte der Standardnormalverteilung



Quantilfunktion

Die Quantilfunktion einer Zufallsvariablen X mit Verteilungsfunktion F ist definiert als

$$F^{-1}(p) = \inf\{x \in \mathbb{R}\colon F(x) \geq p\}$$

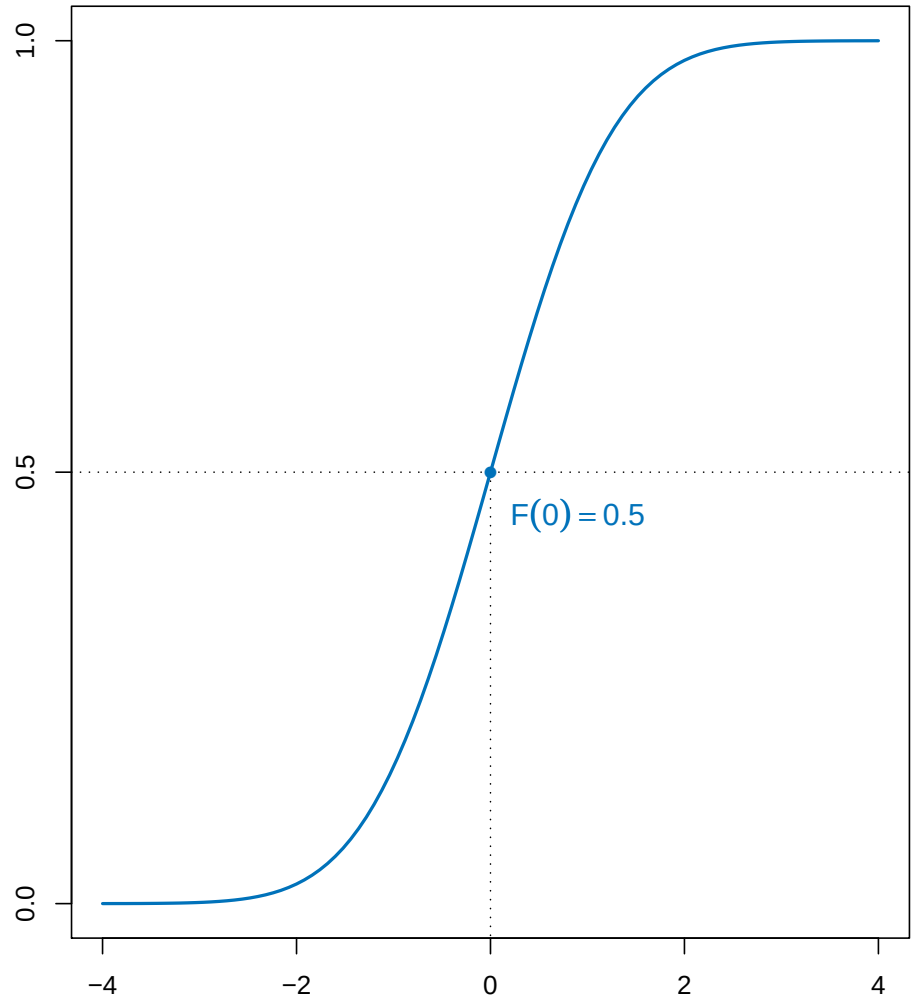
für alle $p \in (0, 1)$.

Für einen festen Wert p nennt man $F^{-1}(p)$ auch das p-Quantil von F.

```
> (x <- qbinom(0.5, size=50, prob=0.33))
[1] 16
> pbinom(x, size=50, prob=0.33)
[1] 0.5068675
> (y <- qnorm(0.5))
[1] 0
> pnorm(y)
[1] 0.5
```

Quantilfunktion

Verteilungsfunktion und Median der Standardnormalverteilung



Grundlagen

- Sweave ist eine von Friedrich Leisch entwickelte Funktion zum Einbetten von R-Code und -Output in LaTeX-Dokumente
- Sie ermöglicht das automatische Einfügen und Aktualisieren von Analyseergebnissen in kompilierbare LaTeX-Dateien
- Unter `http://www.statistik.lmu.de/~leisch/Sweave/` finden sich weitere Informationen zu Sweave

Sweave-Dateien

- Sweave-Dateien besitzen die Endung `.Rnw` (R-Noweb) und können mit jedem Texteditor bearbeitet werden
- Sie bestehen aus einer Kombination von LaTeX- und R-Code zusammen mit den zugehörigen Sweave-Strukturierungselementen
- Eine R-Umgebung beginnt mit `«»=` und endet mit einem `@`-Symbol
- Mit `«Option=Wert»=` können für diesen Abschnitt gewisse Optionen eingestellt werden
- Sweave-Dateien werden in R mit dem Befehl `Sweave()` zu kompilierbaren LaTeX-Dateien (`.tex`) verarbeitet
- Die Funktion `Stangle()` extrahiert nur den R-Code in eine ausführbare R-Datei (`.R`)

Beispiel

- Wir erzeugen mit einem Texteditor die Datei `Sweavebsp.Rnw`
- Diese soll neben einer Konsolenausgabe auch eine Grafik enthalten
- Die Grafikumgebung beginnen wir mit `«fig=TRUE, echo=FALSE»=`
- Durch `echo=FALSE` wird der Quellcode der Grafik nicht in das LaTeX-Dokument übernommen
- Sweave rufen wir dann in R mit `Sweave("Sweavebsp.Rnw")` auf
- Wir erhalten die Datei `Sweavebsp.tex`, die wir mit PDF-LaTeX zu `Sweavebsp.pdf` kompilieren und dann betrachten
- Mit `Stangle("Sweavebsp.Rnw")` erzeugen wir schließlich noch zu Testzwecken die Datei `Sweavebsp.R`
- Diese können wir mit `source("Sweavebsp.R")` in R ausführen

Sweavebsp.Rnw

```
\documentclass[a4paper, 12pt]{scrartcl}

\usepackage[latin1]{inputenc}
\usepackage[ngerman]{babel}

\usepackage{Sweave}

\title{Sweave-Beispiel}
\author{Alexander Bauer}

\begin{document}

\parindent0em
\maketitle

Wir wollen die Korrelation der Variablen \texttt{carat} und
\texttt{price} im \texttt{diamond}-Datensatz des
\texttt{UsingR}-Pakets untersuchen.

...
```

Sweavebsp.Rnw

```
...

<<>>=
library(UsingR)
cor(diamond$carat, diamond$price)
@

Zur Veranschaulichung des Zusammenhangs zeichnen wir ein
Streudiagramm.

\begin{center}
<<fig=TRUE, echo=FALSE>>=
attach(diamond)
par(mar=c(4,4,2,4))
plot(price~carat, xlim=c(0.1,0.35), ylim=c(200,1200), pch=16)
coeff <- lm(price~carat)$coefficients
abline(coeff)
detach(diamond)
@
\end{center}

\end{document}
```

Sweavebsp.tex

```
\documentclass[a4paper, 12pt]{scrartcl}

\usepackage[latin1]{inputenc}
\usepackage[ngerman]{babel}

\usepackage{Sweave}

\title{Sweave-Beispiel}
\author{Alexander Bauer}

\begin{document}

\parindent0em
\maketitle

Wir wollen die Korrelation der Variablen \texttt{carat} und
\texttt{price} im \texttt{diamond}-Datensatz des
\texttt{UsingR}-Pakets untersuchen.

...
```

Sweavebsp.tex

```
...

\begin{Schunk}
\begin{Sinput}
> library(UsingR)
> cor(diamond$carat, diamond$price)
\end{Sinput}
\begin{Soutput}
[1] 0.9890707
\end{Soutput}
\end{Schunk}

Zur Veranschaulichung des Zusammenhangs zeichnen wir ein
Streudiagramm.

\begin{center}
\includegraphics{sweavebsp-002}
\end{center}

\end{document}
```

Sweavebsp.R

```
###################################################
### chunk number 1:
###################################################
library(UsingR)
cor(diamond$carat, diamond$price)

###################################################
### chunk number 2:
###################################################
attach(diamond)
par(mar=c(4,4,2,4))
plot(price~carat, xlim=c(0.1,0.35), ylim=c(200,1200), pch=16)
coeff <- lm(price~carat)$coefficients
abline(coeff)
detach(diamond)
```

Sweave-Optionen

- Die Optionen der „Code-Chunks“ können mit `«Option=Wert»=` oder global mit `SweaveOpts{Option=Wert}` gesetzt werden
- Die Parameter `echo` und `fig` haben wir bereits kennengelernt
- Pro Umgebung kann allerdings nur eine Grafik erzeugt werden
- Mit `label=Name` kann eine Umgebung bezeichnet und später mit `«Name»` wieder verwendet werden

```
«a»=
x <- 1
@

«b»=
«a»
x + 1
@
```

- Ist die erste Option unbezeichnet, so wird sie als `label` interpretiert

Sweave-Optionen

Option	Beschreibung
`echo`	Code in Dokument einbinden
`eval`	Code auswerten
`split`	separate Dateien für jede Code-Umgebung erstellen
`prefix`	Dateinamen haben ein gemeinsames Präfix
`prefix.string`	gemeinsames Präfix der Dateinamen
`include`	`include`-Befehle werden automatisch erzeugt
`fig`	Ausgabe enthält eine Grafik
`eps`	EPS-Grafiken erzeugen
`pdf`	PDF-Grafiken erzeugen
`height`	Höhe der Grafikdateien
`width`	Breite der Grafikdateien