RUHR
UNIVERSITÄT
BOCHUM

RUB

Visualisierung statistischer Daten mit R

Sebastian Jeworutzki

5. August 2010

Inhaltsverzeichnis

1 Einführung in R

R ist eine kostenlose Open-Source Software für statistische Datenverarbeitung, die über die Website http://www.r-project.org bezogen werden kann. Dabei enthält R zum einen eine Vielzahl an Möglichkeiten zur Verarbeitung und Auswertung von Daten, die sich ohne großen Aufwand nutzen lassen. Zum anderen kann man statistische Verfahren auch selber programmieren und R hierüber fast beliebig erweitern. Von Anwendern erstellte Erweiterungen werden als Pakete oder *packages* bezeichnet und von ihren Programmierern oftmals über das Internet im *Comprehensive R Archive Network* (kurz: CRAN) zugänglich gemacht. Hierdurch ist R in vielen Bereichen immer auf dem neuesten Stand und oftmals sogar das erste Softwarepaket, das neu entwickelte Techniken und Verfahren enthält. Die aktuelle Version von R trägt die Nummer 2.11.0 und steht unter der *GNU General Public License*, die eine freie, nicht-kommerzielle Verbreitung ermöglicht.

Die wesentlichen Vorteile von R lassen sich insgesamt wie folgt zusammenfassen:

- R kann kostenlos über das Internet bezogen werden.
- R wird von einem Kern-Team von Entwicklern ständig verbessert.
- Es gibt eine Vielzahl von frei zugänglichen Erweiterungen, die von der ständig wachsenden R-Community erstellt werden.
- R kann durch den Nutzer selbst erweitert werden.

Aufgrund dieser Vorteile findet R zunehmend Verbreitung und wird nicht nur im wissenschaftlichen Bereich, sondern auch für kommerzielle Anwendungen eingesetzt, beispielsweise bei den Unternehmen Shell, Google oder Facebook[1], welche die R-Projektgruppe teilweise auch finanziell unterstützen.

Die Nutzung von R bietet nicht nur bei der Auswertung von Daten viele Vorteile, auch bei der Erstellung von wissenschaftlichen Grafiken ist R ein nützliches Hilfsmittel. R ermöglicht es

- qualitativ hochwertige Grafiken zu erstellen, die auch für den professionellen Druck geeignet sind.

[1] http://dataspora.com/blog/predictive-analytics-using-r/

- Grafiken komplett in der R-Syntax zu erstellen und somit reproduzierbar zu machen.
- flexibel verschiedene Grafiktypen zu kombinieren oder eigene Darstellungsformen zu entwicklen.
- problemlos einheitlich gestaltete Grafiken zu erstellen.

Grundlagen

R unterscheidet sich von einigen anderen Statistikprogrammen unter anderem dadurch, dass Prozeduren und Befehle nicht über eine grafische Oberfläche aufgerufen werden, sondern R Befehle in „Textform“ entgegennimmt. Zunächst schauen wir uns an, wie man überhaupt Befehle eingeben kann.

Nach dem Start von R erscheint ein Fenster, in dem zuerst einige Lizenzinformationen abgebildet sind und dessen letzte Zeile mit dem Symbol

```
>
```

beginnt.

Dieses Symbol zeigt an, dass R auf eine Befehlseingabe wartet. Tippt man den Befehl `2*5+11` ein und drückt schließlich die Eingabetaste, sollte folgender Text zu sehen sein:

```
[1] 21
>
```

Die erste Zeile enthält das Ergebnis der Rechenformel und die letzte Zeile zeigt wieder mit > an, dass weitere Befehle eingegeben werden können. Diese können entweder direkt eingetippt oder auch in einem mit R kompatiblen Editor wie Tinn-R[2], Emacs[3] oder Vim[4] geschrieben werden.

Befehle ausführen

In R können vereinfacht zwei Arten von Objekten unterschieden werden: a) Funktionen und b) Datenobjekte.

Mit Funktionen gibt man R zu verstehen, bestimmte Aufgaben auszuführen. Diese haben immer die Form `funktionsname(Argumente 1, Argument 2, ...)`. Mit dem Hilfebefehl `help(mean)` kann bspw. eine Übersicht über die Argumente, die die Funktion `mean` entgegennimmt, abgerufen werden. Der Aufruf der Hilfe ist gleichzeitig ein einfaches Beispiel für eine Funktion in R: der Funktionsname

[2] http://www.sciviews.org/Tinn-R/
[3] http://ess.r-project.org/
[4] http://sites.google.com/site/jalvesaq/vimrplugin

ist `help` und das Argument in den runden Klammern `mean`. Ein weiteres etwas umfangreicheres Beispiel für Funktionen folgt im nächsten Abschnitt, in dem es um das Einlesen von Datensätzen geht.

Objekte in `R` sind Datenstrukturen, die bestimmte Werte oder Daten repräsentieren und quasi als deren „Name“ fungieren. Um beispielsweise eine Variable „a“ zu erstellen, die den Wert 1 haben soll, kann der Zuweisungsoperator „`<-` “ verwendet werden: `a <- 1`

Nachdem auf diese Weise ein neues Objekt erstellt wurde, kann dessen Inhalt mit „`a` “ einfach wieder abgerufen werden.

Auf diese Weise lassen sich nicht nur einzelne Werte einem Objekt zuweisen, sondern auch Vektoren, Matrizen oder ganze Datentabellen:

```
v <- c(1,2,3,4)          # Erstellt einen Vektor v
                         # mit den Elementen 1,2,3 und 4

v                        # Inhalt des Objektes anzeigen
v[1]                     # 1. Element anzeigen

m <- matrix(c(1,2,3,4), # Erstellt eine Matrix mit
                         # den selben vier Elementen
            ncols=2,     # und zwei Spalten
            nrows=2)     # und zwei Zeilen

m[1, ]                   # 1. Zeile anzeigen
m[ ,1]                   # 1. Spalte anzeigen
m[1,2]                   # 1. Wert in der 2. Spalte
```

Daten importieren

Bevor mit der Visualisierung von Daten begonnen werden kann, müssen diese erst einmal in `R` verfügbar gemacht werden. Die sicherste und einfachste Methode Datentabellen in `R` zu importieren, ist der Weg über Text- bzw. CSV-Dateien, da nahezu alle Statistik- oder Tabellenkalkulationsprogramme Daten in dieser Form abspeichern können. Wie der Begriff CSV – Comma Separated Values – andeutet handelt es sich hierbei um Wertetabellen, bei denen die Spalten durch bestimmte Symbole unterteilt werden. Gängige Trennzeichen sind Kommata, Semikolons aber auch Leerzeichen und „Tab-Stopps“.

Der Befehl `read.table()` dient dazu, Datentabellen dieser Art in `R` einzulesen.

```
dat <- read.table("C:/daten.csv", # Pfad zur Datei
```

```
header=TRUE,     # Variablennamen in der 1. Zeile
sep=";",         # Semikolon als Spaltentrennzeichen
dec="," )        # Komma als Dezimaltrenner
```

Bei der Verwendung des Befehls sind mehrere Parameter anzugeben. Der wichtigste ist natürlich der Pfad der zu importierenden Datei. Dabei ist für Windows-Nutzer zu beachten, dass ein „/“ statt eines „\“ verwendet wird. Die weiteren Parameter enthalten Angaben über die Struktur der Datei. Mit der Option `header` wird angegeben, ob die Datei Spaltenüberschriften in der ersten Zeile enthält, dazu werden die logischen Werte `TRUE` und `FALSE` genutzt. Die Option `sep` gibt an, wie die einzelnen Spalten voneinander getrennt sind. Wie alle Zeichenfolgen müssen auch die Trennzeichen mit Anführungsstrichen eingerahmt werden, um sie von Objektnamen zu unterscheiden – fehlen sie kommt es zu Fehlermeldungen. Eine Besonderheit ist bei Daten zu beachten, die durch Tab-Stopps getrennt sind: dieser wird mit `sep="\t"` angegeben. Die letzte wichtige Option ist `dec=","`, mit der das verwendete Dezimaltrennzeichen angegeben wird. Bei Dateien aus dem deutschen Sprachraum ist dies meistens das Komma.

Spalten, die nicht numerische Werte enthalten, werden beim Import von `R` automatisch in einen `factor` umgewandelt, d.h. die unterschiedlichen Zeichenketten werden als Ausprägungen einer kategoriale Variable interpretiert. Um die Werte als reine Zeichenketten einzulesen, muss die Option `stringsAsFactors` auf `FALSE` gesetzt werden.

Nach Aufruf des Befehls sind die importierten Daten unter dem Objektnamen `dat` verfügbar. Einzelne Variablen können mit `dat$Variablename` angesprochen werden oder auch über Indizes: `dat[,1]` listet die Elemente der Variable in der ersten Spalte auf. `dat[1,]` listet die Werte des ersten Falls in allen Variablen auf und `dat[3,5:7]` zeigt die Werte des dritten Falls für die fünfte, sechste und siebte Variable an.

Erweiterungspakete installieren

Von Anwendern erstellte Erweiterungen werden als Pakete oder *packages* bezeichnet und von ihren Programmierern oftmals über das Internet zugänglich gemacht. Im *Comprehensive R Archive Network* (kurz: CRAN), einem Netz aus Webservern, die Pakete und Code für `R` bereitstellen, sind über 2000 solcher Pakete gelistet.

Erweiterungspakete lassen sich in der Windows Version von `R` entweder über das Menü „Pakete“ oder den Befehl `install.packages("Paketname")` installieren. Ein guter Einstiegspunkt für die Suche nach nützlichen Paketen ist die Homepage[5]

[5] `http://cran.r-project.org/web/packages/`

des CRAN Projektes, auf der alle verfügbaren Pakete mit einer Kurzbeschreibung aufgelistet sind.

Literatur zur Vertiefung

Die bisherigen Ausführungen können nur einen kleinen Einblick in die grundsätzliche Bedienung von R geben und ist auf das Notwendige zum Verständnis der folgenden Ausführungen beschränkt.

Einen tieferen Einblick in die Funktionsweise von R und die Anwendung verschiedener statistischer Verfahren liefern unter anderem:

- Christian Dudel und Sebastian Jeworutzki (2010): *Einführung in R*. url: http://www.stat.rub.de
- Michael J. Crawley (2009): *The R book*. Reprint. Chichester u.a.: Wiley. VIII, 942. isbn: 9780470510247
- Peter Dalgaard (2008): *Introductory statistics with R*. 10. Statistics and computing. New York u.a.: Springer. XV, 267. isbn: 0387954759
- Uwe Ligges (2007): *Programmieren mit R*. 2., überarb. und aktualisierte Aufl. Statistik und ihre Anwendungen. Berlin u.a.: Springer. XII, 247. isbn: 9783540363323. url: http://dx.doi.org/10.1007/978-3-540-36334-7
- W. N. Venables u. a. (2009): „An Introduction to R“. Version 2.10.0, abrufbar unter www.r-project.org

2 Grundlegende Grafikfunktionen

2.1 Ausgabegeräte

Die Grafikausgabe in R erfolgt über sogenannte graphic devices, kurz Ausgabegeräte. Bereits in der Standardinstallation sind verschiedene Ausgabegeräte mit unterschiedlichen Aufgabengebieten und Fähigkeiten verfügbar. Einige sind plattformspezifisch wie X11, Quartz oder Windows, andere wie pdf sind unter allen Betriebssystemen verfügbar.

Während sich pdf, xfig, tkiz oder svg auf die Dateiausgabe beschränken, verfügen die plattformspezifischen Ausgabegeräte meistens über die Fähigkeit, sowohl für die Bildschirmausgabe als auch für die Dateiausgabe zu sorgen.

Für LaTeX-Benutzer sind insbesondere die Ausgabegeräte tikz und pictex von Interesse, da mit ihnen LaTeX-Code für die tkizpicture- bzw. pictex-Umgebung generiert werden kann und die Grafiken sich so nahtlos in das LaTeX-Dokument einfügen. Eine nicht zwangsläufig vollständige Übersicht der verschiedenen Ausgabegeräte in R und die entsprechenden Pakete sind in Tabelle 2.1 aufgeführt.

Wird kein Ausgabegerät explizit aufgerufen, erfolgt die Grafikausgabe auf dem Standardgerät. Unter Windows ist dies windows, unter Mac OSX Quartz und unter Linux X11. Beim Aufruf der verschiedenen Ausgabegeräte können zusätztliche Optionen angegeben werden – diese werden im Folgenden für einige Ausgabegeräte vorgestellt.

Windows

Die grafische Ausgabe erfolgt unter Windows automatisch über das windows-Device. Es kann jedoch auch manuell ein (zusätzliches) Ausgabefenster unter Angabe weiterer Optionen gestartet werden:

```
windows(width=7,          # Breite und
        height=7,         # Höhe des Grafikfensters in Zoll
        pointsize=12,     # Standardschriftgröße in Punkt
        record=FALSE,     #
        rescale="R",      # "R" Breite und Höhe durch vergrößern/
```

Tabelle 2.1: Grafikausgabegeräte in R.

Gerät	Betriebssystem	Verfügbarkeit
PostScript(& Bitmap)	alle	R-Base
pictex	alle	R-Base
pdf	alle	R-Base
xfig	alle	R-Base
tikz	alle	tikzDevice (CRAN)
Java	alle	RJavaDevice (CRAN)
GTK	alle	gtkDevice (CRAN)
Cairo	alle	cairoDevice (CRAN)
libgd	alle	GDD (RForge)
SVG	alle	RSvgDevice (CRAN)
X11 (& PNG & JPEG)	UNIX	R-Base
GNOME	UNIX	R-Base
windows	Windows	R-Base
proxy	Windows	R-Base
Quartz	MacOS X	R-Base

Quelle: Murrel 2009

```
                    #     verkleinern des Fensters zulassen
                    # "fit" Breite und Höhe anpassen,
                    #       Seitenverhältnis fixieren
                    # "fixed" Breite und Höhe fixieren
  bg="transparent", # Hintergrundfarbe des Plots
  canvas="white",   # Hintergrundfarbe bei bg="transparent"
  gamma=1,          # Gammakorekktur für die Darstellung
  )
```

Die oben aufgeführten weiteren Optionen sind vor allem interessant, wenn die aktuelle Bildschirmausgabe in eine Datei gespeichert werden soll und das Seitenverhältnis, die Größe etc. vorab festgelegt werden soll – dazu mehr auf Seite 13.

Ein Grafikfenster kann entweder per Mausklick oder mit dem Befehl `dev.off()` geschlossen werden. Sind mehrere Grafikfenster geöffnet, wird das zuletzt geöffnete Fenster zuerst geschlossen. Zwischen den Fenstern kann mit den Funktionen `dev.next()` und `dev.prev()` gewechselt werden.

Ausgabe in Dateien

Die Ausgabe in eine Datei kann auf zwei Arten erfolgen: a) durch "kopieren" des Inhalts des Grafikfensters in eine Datei und b) durch das direkte Schreiben in die Ausgabedatei.

Ist man mit der erstellten Grafik zufrieden, kann man Sie mit dem Befehl `dev.copy` abspeichern:

```
dev.copy(device=Dateityp, file="Dateiname", width=x, height=y)
```

Als Dateityp sind unter Windows `postscript`, `pdf`, `png` oder `jpeg` zulässig. Die Rasterformate PNG und JPEG sollten nicht für Ausdrucke sondern nur für die Bilschirmdarstellung genutzt werden. Die Optionen `width` und `height` bestimmen die Dimensionen der Ausgabedatei. Bei den Rasterformaten sind die Angaben auf Pixel bezogen und bei PDF und Postscript wird die Breite und Höhe in Inch angegeben. Unter Windows besteht zusätzlich die Möglichkeit, den Inhalt des Grafikfensters über das Menü abzuspeichern oder direkt zu drucken.

Die Variante b) ist besonders hilfreich, wenn Grafiken im Rahmen von Skriptabläufen erstellt werden sollen. Bei dieser Variante wird zuerst ein Grafikausgabegerät gestartet: `pdf(file="Ausgabdatei.pdf", width=11, height=6)`. Danach folgen die Plot-Anweisungen und erst wenn mit `dev.off()` die Datei geschlossen wird, ist der Inhalt auch auf die Festplatte geschrieben und kann betrachtet werden.

```
pdf(file="sweavetmp/test.pdf",  width=11, height=6)
plot(1:10, 11:20)   # 1. Grafik
plot(21:30,1:10)    # 2. Grafik
dev.off()
```

Wie im Beispiel zu erkennen ist, können mehrere Plots in eine Datei geschrieben werden. Dies ist jedoch nur bei PDF-Dateien möglich. Rastergrafiken enthalten nur den letzten Plot.

```
png(file="sweavetmp/test.png",  width=400, height=300)
plot(1:10, 11:20)   # 1. Grafik
plot(21:30,1:10)    # 2. Grafik, nur dies ist gespeichert
dev.off()
```

Es können auch automatisch mehrere Dateien erzeugt werden:

```
png(file="sweavetmp/test%03d.png",  width=400, height=300)
plot(1:10, 11:20)   # 1. Grafik
plot(21:30,1:10)    # 2. Grafik, nur dies ist gespeichert
dev.off()
```

Das Einfügen von `%03d` sorgt dafür, dass automatisch für jeden Plot eine neue Datei mit dem Namen test001.png, test002.png usw. angelegt wird. Die Art der Nummerierung kann über ein beliebiges `printf`-Format gesteuert werden, so erzeugt `file ="sweavetmp/datei%1d.png"` Dateinamen folgender Art: datei1.png, datei2.png usw.

Für PDF-Dateien ist es nützlich, sich eine Variable `zoll <-0.3937` zu erstellen, so dass mit `width=10*zoll` direkt bei der Längenangabe von cm in Zoll umgerechnet wird.

Cairo

Das R-Paket „Cairo“ stellt unter Windows und Unix ein Gerät für die Ausgabe von hochauflösenden PNG-, JPEG- und TIFF-Bitmap-Grafiken bereit sowie qualitativ hochwertigen PDF-, SVG- und PostScript-Dateien. Das `Cairo`-Device unterstützt für alle Ausgabeformate viele Grafikeigenschaften wie beispielsweise alpha blending oder anti-aliasing. Eine weitere Besonderheit ist die Unterstützung von im System installierten TrueType-Schriften (siehe Seite 37).

Ein interaktives Grafikfenster muss im Gegensatz zur Standardausgabe explizit aufgerufen werden. Unter Windows geschiet dies mit `CairoWin()` und unter Unix mit `CairoX11()`. Das Ausgabeformat wird durch die Auswahl der Ausgabefunktion bestimmt: a) CairoPNG b) CairoJPEG c) CairoTIFF d) CairoPDF e) CairoSVG f) CairoPS.

Bei der Bitmap-Ausgabe kann die Auflösung der Ausgabedatei direkt in Pixeln angegeben werden:

```
CairoPNG("test.png",          # Dateiname
         width=480,           # Breite
         height=480,          # Höhe
         units="px",          # Einheiten für Breite/Höhe
         bg="transparent")    # Hintergrundfarbe
```

Das Cairo-Gerät untersützt bei PNG-Dateien Tranzparenz, so dass im obigen Beispiel der Hintergund durchscheinend ist.

Welches Ausgabeformat ist das Richtige?

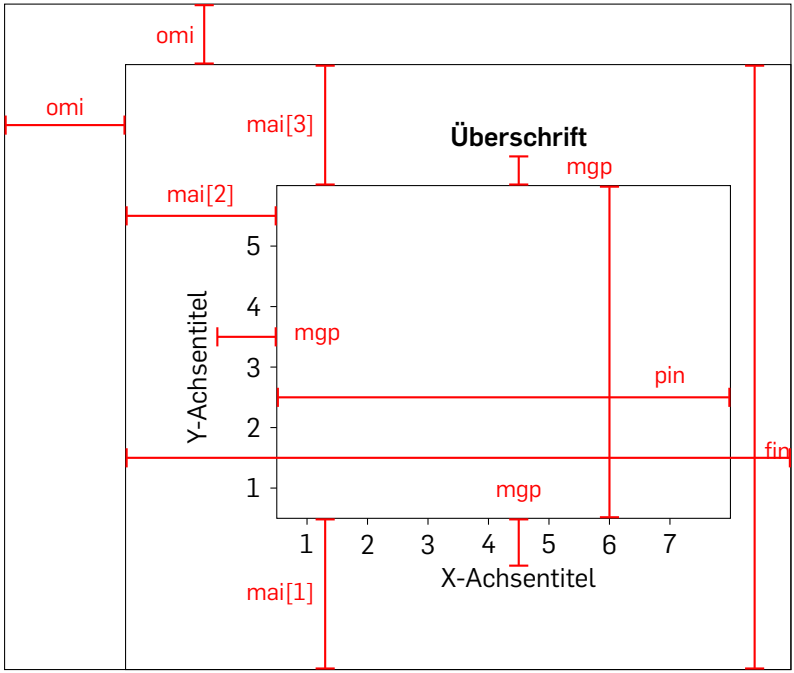
Die Vielzahl der möglichen Ausgabegeräte lässt natürlich die Frage aufkommen, welches Gerät sich für welche Aufgaben eignet. Generell lässt sich folgende Unterscheidung treffen: Für die Druckausgabe sollten immer Vektorgrafikformate verwendet werden und für die Bildschirmausgabe können die Bitmapformate genutzt werden.

Vektorformate wie PDF, Postscript oder SVG nutzen grafische Primitive wie Linien, Kreise oder allgemeiner Polygone und Splines zur Darstellung des Bildinhaltes. Dieses Vorgehen hat mehrere Vorteile: zum einem sind die resultierenden Dateien verhältnismäßig klein, da beispielsweise für eine gerade Linie nur der Start- und Endpunkt gespeichert werden müssen. Zum anderen lassen sich die Grafiken im gegensatz zu Bitmapgrafiken ohne Qualitätsverlust skalieren. Welches der verschiedenen Vektorformate schließlich genutzt werden sollte, hängt vor allem von der zur Weiterverabeitung genutzen Software ab. Sollen die Grafiken beispielsweise in ein Microsoft Word Dokument eingefügt werden, bieten sich die Formate EPS oder WMF an, da diese nativ von Word unterstützt werden. Für LaTeX-Anwender eignen sich sowohl Postscript- und PDF-Dateien, wie auch die Ausgabegeräte `tikzDevice` oder `pictex`, wobei letztere direkt LaTeX-Code produzieren.

Für die Einbindung von Grafiken in Powerpoint- oder OpenOffice-Bilschirmpräsentationen können sämtliche Bitmapformate genutzt werden. Bitmap- oder Pixelgrafiken verwenden ein Raster von Bildpunkten, um den Bildinhalt darzustellen. Dies lässt sich gut mit einem Bildbearbeitungsprogramm nachvollziehen: stellt man die Vergrößerung auf die höchste Stufe ein, lassen sich die einzelnen farbigen Quadrate erkennen, aus denen das Bild zusammengesetzt ist. Dieser Effekt tritt auch leicht auf, wenn Bilder mit zu geringer Auflösung im Druck verwendeten werden. Bei der Skalierung werden die einzelnen Bildpunkte größer und können so schnell die Darstellung trüben. Auch bei der Verwendung in Bildschirmpräsentationen ist es sinvoll vorab zu überlegen, auf welchen physischen Ausgabegerät die Darstellung erfolgt. Die meisten Beamer arbeiten heute mit einer Auflösung von 1024 mal 786 Bildpunkten, eine Grafik die etwas die Hälfte der Bildfläche einnimmt muss also mindestens 512 mal 393 Pixel groß sein, um in bester Qualität dargestellt zu werden. Ist die Bilddatei größer als die Auflösung des Ausgabemediums ergeben sich meist keine Probleme, da die Verkleinerung meist ohne sichtbaren Qualitätsverlust erfolgt. Lediglich die Dateigröße wird unnötigerweise erhöht. Die Frage welches der Bitmapformate verwendet werden sollte, richtet sich wieder nach der weiteren Verwendung. Generell ist das PNG-Format zu empfehlen, da es im Gegensatz zu JPEG Tranzparenz unterstützt und auch nicht wie GIF-Dateien in der Anzahl der nutzbaren Farben eingeschränkt ist.

2.2 Einstellungen für Ränder

Die Grafikausgabe ist in verschiedene Bereiche eingeteilt, die jeweils eigene Einstellungen für Ränder und Bereichsgrößen haben.



Die Einstellungen werden mit der Funktion `par()` für die gesamte Sitzung festgelegt. Die Optionen werden in Form einer Liste geschrieben, d.h. `option=werte`. Die wichtigsten Einstellungen zur Beeinflussung der Ränder werden im Folgenden erläutert:

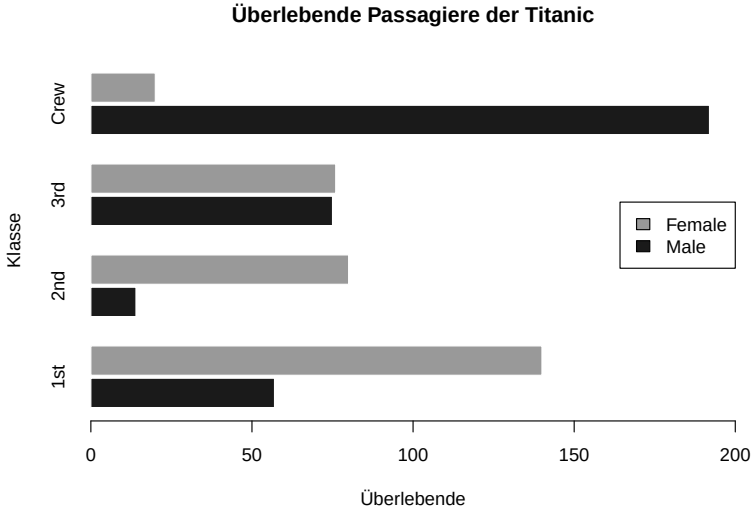
```
par(mar=c(5,4,4,2)+0.1, # Bereich um die Plotfläche für Beschriftun-
                        #  gen und Titel in Zeilen (unten, links,
                        #  oben, rechts)
    mai=c(5,4,4,2),     # Dieselbe Angabe in Zoll

    oma=c(0,0,0,0),     # Rand um die Beschriftungen in Zeilen
    omi=c(0,0,0,0),     # Dieselbe Angabe in Zoll

    fin=c(6.994488,     # Breite und Höhe des Bereiches
          6.999377))    #  mit Plot und Beschriftungen
```

3 Standardiagramme in R

3.1 Balkendiagramme

Balkendiagramme, auch als Säulen-, Stab-, Streifen- oder Blockdiagramme (englisch: bar chart) bezeichnet, existieren in verschiedenen Varianten. Im Beispiel unten kann an der Länge der Balken die Zahl der Überlebenden des Titanic-Unglücks getrennt nach Geschlecht und Passagierklasse abgelesen werden.

Die Option `beside=TRUE` sorgt dafür, dass die Balken für Männer und Frauen nebeneinander stehen, eine andere Darstellung erhält man über `beside=FALSE`: Die Balken werden aufeinander gestapelt und geben die Geschlechterverteilung innerhalb der Klassen wieder. Grundsätzlich lassen sich alle Daten, die mit einer Häufigkeitstabelle oder einer Kreuztabelle beschrieben werden können sinnvoll als Balkendiagramm darstellen. Auch andere numerische Werte lassen sich durch Balkendiagramme darstellen. Bei gestapelten Balkendiagrammen muss dann jedoch beachtet werden, dass die Höhe der Balken die Summe der numerischen Werte abbildet.




```
> barplot(t(Titanic[,,2,2]),
```

```
beside=TRUE,                # Nebeneinander
horiz=T,                    # Balken horizontal
col=c("gray10","gray60"),   # Farbwerte für die Balken
border="white",             # Farbe der Balkenränder,
                            #  NA bedeutet keinen Rand
space=c(0.1,1),             # Abstand links vom Balken in
                            #  Prozent der Balkenbreite
                            # Wenn beside=TRUE
                            #  1. Wert: Abstand innerhalb
                            #           der Kategorie
                            #  2. Wert: Abstand zwischen
                            #           den Kategorien
legend=TRUE,                # Legende drucken
args.legend=list(           # Argumente für die Legende
            x=200,          # x-Position
            y=8),           # y-Position
xlim=c(0,200),              # Y-Achse skalieren (von, bis)
xlab="Überlebende",         # Beschriftung X-Achse
ylab="Klasse",              # Beschriftung der Y-Achse
# Diagrammtitel
main="Überlebende Passagiere der Titanic"
)
```

Die Definition der Kategorien, deren Häufigkeitsverteilung dargestellt werden sollen, wird durch den Aufbau der Kreuztabelle bestimmt: Jede Spalte bildet eine Kategorie, jede Zeile wird gesondert innerhalb der Kategorie dargestellt. Deutlich wird der Unterschied, wenn die Darstellung für `Titanic[,,2,2]` und die transformierte Tabelle `t(Titanic[,,2,2])` verglichen werden.

Titanic[,,2,2]

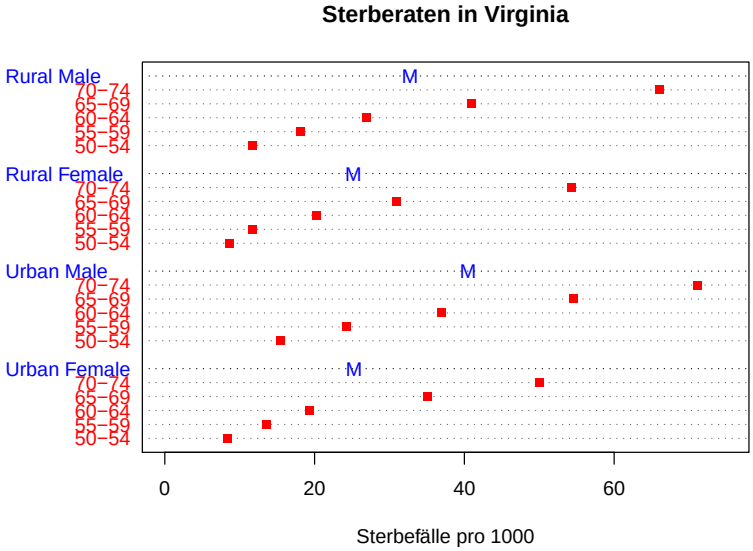
	Sex	
Class	Male	Female
1st	57	140
2nd	14	80
3rd	75	76
Crew	192	20

t(Titanic[,,2,2])

	Class			
Sex	1st	2nd	3rd	Crew
Male	57	14	75	192
Female	140	80	76	20

3.2 Punktdiagramme

Das Punktdiagramm (englisch: dot chart) unterscheidet sich nur wenig vom Balkendiagramm. Die Häufigkeiten werden nicht als Balken dargestellt, stattdessen symbolisieren einzelne Punkte die Werte. Dadurch kann die Übersichtlichkeit deutlich gesteigert und so eine größere Zahl von Einzelwerten visualisiert werden.



Die Funktion `dotchart` ermöglicht es, zusätzlich zu den Werten für die einzelnen Kategorien (im Beispiel oben die Altersstufen und Wohnort) Durchschnittswerte darzustellen. Im folgendem Beispiel werden dafür die durchschnittlichen Sterberaten für Männer und Frauen in den städtischen bzw. ländlichen Gebieten berechnet.

```
> mean.deaths <- mean(data.frame(VADeaths))
```

Anschließend können die Mittelwerte mit dem Parameter `gdata` beim Aufruf der Funktion angegeben werden. Das Aussehen dieser Gruppenwerte wird mit den Optionen `gpch` für die Symbole und `gcolor` für die Farbe gesteuert.

```
> dotchart(VADeaths,                    # Matrix mit den Werten
         pch=15,                        # Symbole für Kategorien
         col="red",                     # Farbe der Symbole
         xlim=c(0,75),                  # Y-Achse (von, bis)
         gdata=mean.deaths,             # Werte für die Gruppen
         gpch="M",                      # Symbol für Gruppewerte
```

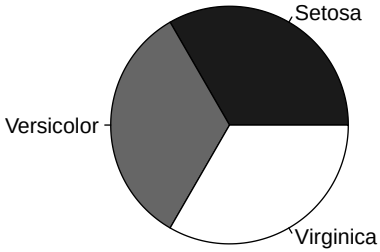
```
    gcolor="blue",                  # Farbe der Gruppensymbole
    lcolor=gray(0.4),               # Farbe der horiz. Linien
    xlab="Sterbefälle pro 1000",    # Beschriftung Y-Achse
    main="Sterberaten in Virginia") # Diagrammtitel
```

VADeaths

	Rural Male	Rural Female	Urban Male	Urban Female
50-54	11.7	8.7	15.4	8.4
55-59	18.1	11.7	24.3	13.6
60-64	26.9	20.3	37.0	19.3
65-69	41.0	30.9	54.6	35.1
70-74	66.0	54.3	71.1	50.0

3.3 Kreisdiagramme

Kreis- oder Tortendiagramme (englisch: pie chart) sind zwar weit verbreitet, weisen jedoch einige Nachteile auf: a) Unterschiede zwischen den Anteilswerte sind weniger gut erkennbar, da dazu die Fläche der Kreissegmente verglichen werden muss, b) bei vielen Kategorien wird die Darstellung schnell unübersichtlich, c) sehr kleine Anteilswerte können oftmals nicht im Kreisdiagramm dargestellt werden. Aufgrund dieser Nachteile bietet sich die Verwendung von Kreisdiagramme nur in selten Fällen an – meist liefern Punkt- oder Balkendiagramme bessere Darstellungen (vgl. Cleveland und McGill 1984: S. 545).




```
> pie(table(iris$Species),              # Häufigkeitstabelle
     labels=c("Setosa", "Versicolor",   # Beschriftung Segmente
              "Virginica"),
     col=grey(c(0.1, 0.4, 1)))          # Grauton der Segmente
```

Im obigen Beispiel wurden drei Grautöne durch den Aufruf von `grey(c(0.1, 0.4, 1))` erzeugt. An diesem Beispiel zeigt sich, dass fast alle R-Funktionen Vektoren als Argument akzeptieren und einen Ergebnisvektor zurückliefern.[1]

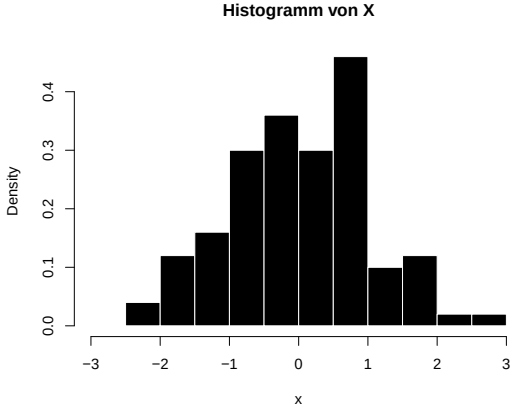
iris$Species

```
      setosa versicolor virginica
[1,]     50         50        50
```

3.4 Histogramme

Histogramme sehen auf den ersten Blick Balkendiagrammen sehr ähnlich. Während das Balkendiagramm zur Darstellung der Häufigkeit kategorialer Daten dient, werden im Histogramm die Häufigkeitsverteilung quantitativer Variablen dargestellt. Im Unterschied zum Balkendiagramm muss im Histogramm nicht nur die Höhe der Balken betrachtet werden, sondern auch deren Breite.

Bei der Konstruktion eines Histogramms wird der Merkmalsbereich in Intervalle der Form $[a, b)$ eingeteilt. Die Intervallgrenzen bestimmen die Breite der Balken im Diagramm. Die Höhe der Balkens für das Intervall j wird wie folgt berechnet: $h_j := n_j/d_j$, wobei n_j die Anzahl der Fälle in der Klasse j bezeichnet und d_j die Breiten der Klasse, also $a_{j+1} - a_j$. Die Fläche der Rechtecke ist also proportional zur Häufigkeit der Fälle im entsprechenden Intervall und die Gesamtfläche entspricht 1 beziehungsweise 100%.



[1] hier: `c("#1A1A1A", "#666666", "#FFFFFF")`

Die relativen Häufigkeiten lassen sich einfach bestimmen, indem die Breite des Intervalls mit der Höhe des Balkens multipliziert wird. Im oben abgedruckten Beispiel ist die relative Häufikeit der Fälle im Intervall [0,0.5): $0.5 - 0 \cdot 0.3 = 0.15$.

Eine Besonderheit ergibt sich, wenn alle Intervalle gleich breit sind. In diesem Fall entspricht die Höhe der Balken der Häufigkeit n_j. In `R` wird dies durch die Option `freq=TRUE` erreicht.

Die Intervallgrenzen werden von `R` automatisch bestimmt, können jedoch auch explizit angegeben werden. Die Angabe `breaks=c(-3,1,0,1,3)` liefert bspw. die Intervalle $[-3, 1)$, $[1, 0)$, $[0, 1)$ und $[1, 3)$; die Angabe einer einzelnen Zahl (wie im obigen Beispiel) erzeugt die entsprechende Anzahl von Intervallen. Eine dritte Möglichkeit besteht darin, eine Funktion zur Berechnung der Intervalle anzugeben.

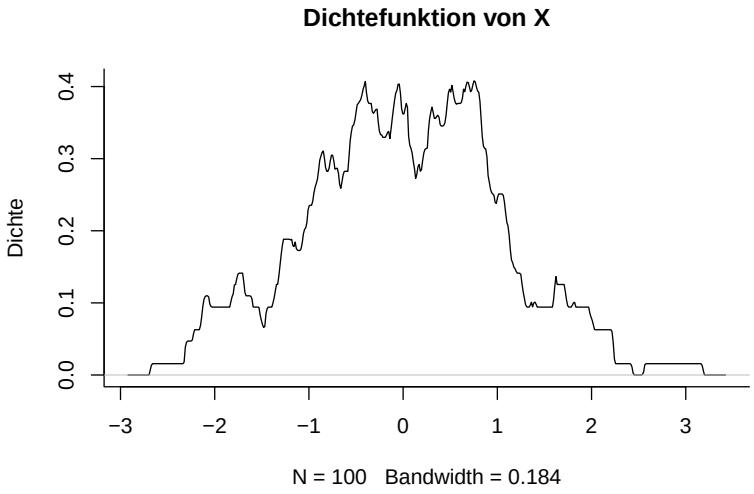
```
> set.seed(19)            # Startwert für Zufallsgenerator
> x <- rnorm(100)         # 100 normalverteilte Zufallszahlen

> hist(x=x,               # Variable
      breaks=12,          # Anzahl oder Vektor mit Intervallgrenzen
      freq=FALSE,         # Höhe soll Häufigkeiten zeigen; nur bei
                          #  äquidistanten Intervallen
      right=T,            # Intervalle der Form [a,b), sonst (a,b]
      col="black",        # Farbe der Flächen
      border="white",     # Farbe der Ränder
      xlim=c(-3,3),       # Grenzen X-Achse
      main="Histogramm von X")
```

3.5 Dichteschätzer

Plots von Dichtefunktionen (engl: kernel density estimation) sind eine Verallgemeinerung des Histogramms. Auch bei der Verwendung von Dichteschätzern ist die Häufigkeit proportional zur Fläche unter der Kurve, mit dem Unterschied, dass keine festen Intervalle gebildet werden, sondern eine stetige Dichtefunktion $f(z) := \frac{1}{n} \sum_{\omega \in \Omega} k_\omega(z, h)$ gebildet wird.

Die genaue Form der Dichtefunktion hängt von den n Kernfunktionen k_ω der Fälle $\omega_1, \ldots, \omega_n$ ab. Die Kernfunktion gibt an, ob und wenn ja mit welchem Gewicht ein Fall für die Bestimmung der Dichte am Punkt z auf der X-Achse einbezogen wird. Der Parameter h gibt die Bandbreite an, also die Breite des Intervalls um z herum, in dem Werte für die Dichtebestimmtung berücksichtigt werden.

In der Standardeinstellung wird eine glockenförmigen Kernfunktion verwendet, es sind jedoch einige weitere Kernfunktionen implementiert: a) epanechnikov b) rectangular c) triangular d) biweight e) cosine f) optcosine



Ein „density plot“ wird in zwei Schritten erstellt. Zuerst muss die Dichtefunktion mit einer der oben genannten Kernfunktionen berechnet werden:

```
> den.x <- density(x,                  # Vektor mit Daten
                   bw="nrd0",          # Bandbreite oder Funktion für bw
                   adjust=0.5,         # Gewicht für bw: hier 0.5*bw
                   kernel="rectangular", # Kernfunktion
                   n = 512)            # Anzahl der Stützstellen z
```

Anschließend werden die Datenpunkte als Liniendiagramm dargestellt.

```
> plot(den.x,                          # Dichtefunktion von x
     col="black",                      # Farbe der Line
     bty="l",                          # Form der Box um den Plot
     main="Dichtefunktion von X",      # Überschrift
     ylab="Dichte",                    # Y-Achsenbeschriftung
     zero.line = TRUE)                 # Null-Linie einzeichnen
```

x

```
 [1] -1.189  0.389 -0.344 -0.548  0.981 -0.237  0.810 -0.745 -0.260
[10] -0.183  0.519  0.883  0.590 -0.197  0.660 -0.261 -0.573  1.407
[19]  0.504 -0.700  1.423 -1.043  0.001  1.090
```

3.6 Streudiagramme und Boxplots

3.6.1 Streudiagramm

Streudiagramme (englisch: scatterplot) sind eines der hilfreichsten Instrumente zur Darstellung von Zusammenhängen zwischen quantitativen Variablen. Im Streudiagramm werden die Ausprägungen der zwei Variablen als Punkte der Form (x-Wert,y-Wert) repräsentiert. So kann die Häufigkeitsverteilung der beiden Variablen gemeinsam dargestellt werden. Die Diagramme können leicht um zusätzliche Informationen angereichert werden, indem unterschiedliche Symbole und Farben als Kodierung für ein drittes Merkmal genutzt werden.

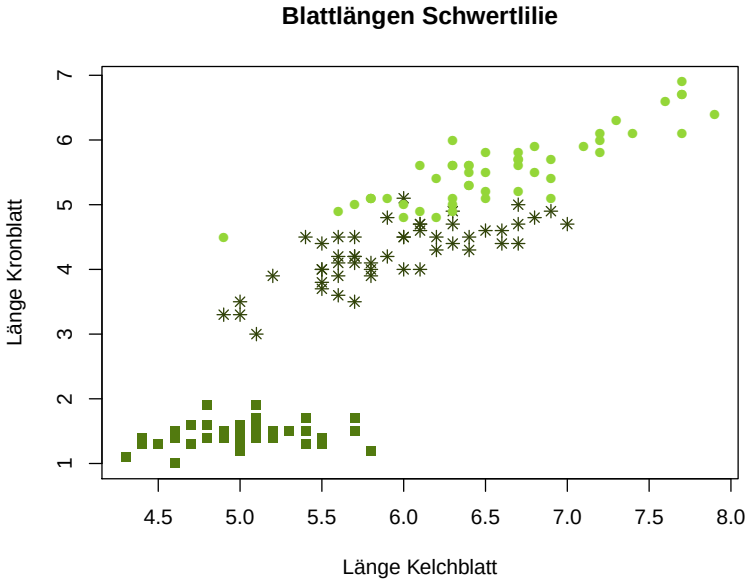
Im Folgenden werden einige Beispiele für Streudiagramme vorgestellt. Zuerst wird anhand des Iris-Datensatzes gezeigt, wie verschiedene Untergruppen in den Daten im Streudiagramm gekenzeichnet werden können.

Dazu wird in Abhängigkeit von der Gruppenzugehörigkeit eine Farbe für jeden Fall definiert. Das Ergebnis ist ein Vektor, der für jeden Fall einen Farbwert enthält. Zur Definition der Farben werden Angaben im RGB-Format genutzt (siehe S. 33).

```
> col.spec <- NULL    # Leeren Vektor für die Farbwerte erstellen
> col.spec[iris$Species=="virginica"]  <- rgb(0.58, 0.84, 0.22)
> col.spec[iris$Species=="setosa"]     <- rgb(0.32, 0.48, 0.06)
> col.spec[iris$Species=="versicolor"] <- rgb(0.19, 0.26, 0.03)
```

Zu Demonstrationszwecken wird zusätzlich ein Vektor erstellt, der für jeden Fall einen eigenen Symboltyp definiert.

```
> pch.spec <- NULL    # Leeren Vektor für die Symbole erstellen
> pch.spec[iris$Species=="virginica"]  <- 16
> pch.spec[iris$Species=="setosa"]     <- 15
> pch.spec[iris$Species=="versicolor"] <- 8
```



Beim Aufruf der Funktion werden nun statt einer bestimmten Farbe oder eines bestimmten Symboltyps die entsprechenden Vektoren angegeben.

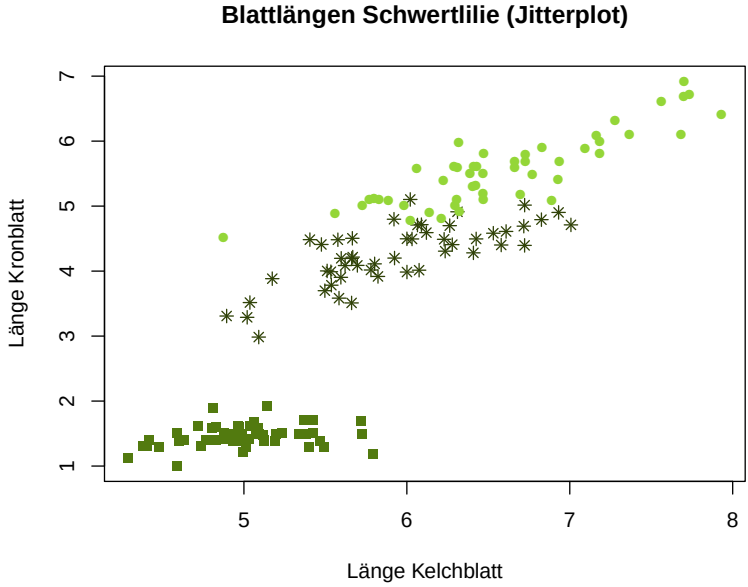
```
> plot(x=iris$Sepal.Length,        # X-Werte
       y=iris$Petal.Length,        # Y-Werte
       col=col.spec,               # Farben der Symbole
       pch=pch.spec,               # Symbole auswählen
       cex=1,                      # Vergrößerungsfaktor für Symbole
       xlab="Länge Kelchblatt",
       ylab="Länge Kronblatt",
       main="Blattlängen Schwertlilie")
```

head(iris[,c(1,2)]

	Sepal.Length	Petal.Length
1	5.1	1.4
2	4.9	1.4
3	4.7	1.3
4	4.6	1.5
5	5.0	1.4
6	5.4	1.7

Problematisch wird die Darstellung durch Streudiagramme, wenn verschiedene Werte in den zugrundeliegenden Daten mehrfach vorkommen. Die Punkte überlagern sich und die Häufig dieser Werte lässt sich in der Grafik nicht erkennen.

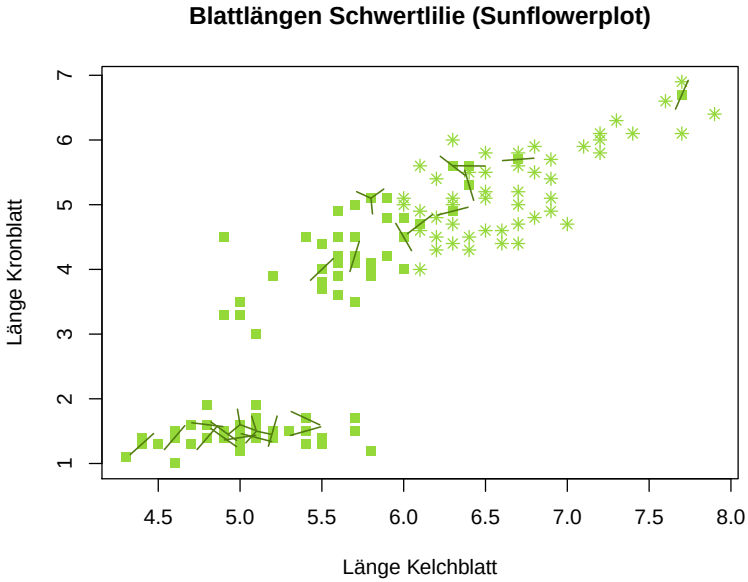
Um die Verzerrung in der Darstellung zu korrigieren, können kleine Zufallsschwankungen um die Werte von x erzeugt werden, so dass Punkte mit den selben Koordinaten leicht versetzt dargestellt werden. So sind auch Häufungen an einer Koordinate oder nahe beieinander liegenden Koordinaten erkennbar.



Mit der `jitter`-Funktion werden einfach zwei neue Vektoren mit den geänderten Datenpunkten erzeugt und dann anstatt der Ausgangsvariablen `Setal.Length` und `Petal.Lengt` für das Streudiagramm genutzt.

```
> x <- jitter(iris$Sepal.Length,    # "Jitter" erzeugen
              factor=2)             # Größe des Versatzes
> y <- jitter(iris$Petal.Length)
```

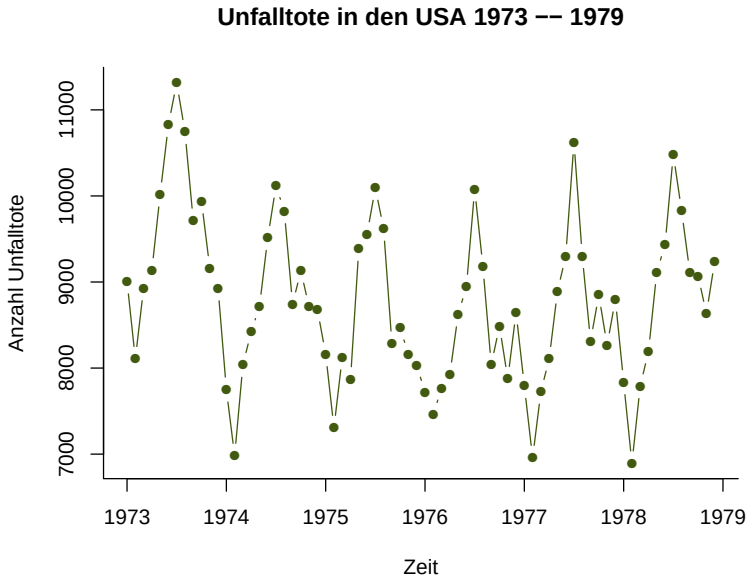
Der Sunflowerplot begegnet dem Problem des Überlagerns mehrerer Punkte auf eine andere Weise: liegt auf einer Koordinate nur ein Punkt wird das herkömmliche Symbol gezeichnet. Liegen mehre Fälle auf einer Koordinate wird für jeden zusätzlichen Fall ein “Blütenblatt” gezeichnet. Die Anzahl der Blüttenblätter entspricht also der Anzahl der Fälle an der entsprechenden Koordinate.




```
> sunflowerplot(x=iris$Sepal.Length,     # X-Werte
                y=iris$Petal.Length,     # Y-Werte
                pch=pch.spec,            # Symbol Punkte & Blumenmitte
                cex=1,                   # Vergrößerungsfaktor Symbole
                cex.fact=1,              #  für Symbole in der Blume
                col="#94d839",           # Farbe Punkte & Blumenmitte
                seg.col="#527b10",       # Farbe der Blütenblätter
                seg.lwd=1.2,             # Liniendicke Blütenblätter
                rotate=T,                # Blumen zufällig rotieren
                add=F,                   # Zu bestehenden Plot hinzufügen
                xlab="Länge Kelchblatt",
                ylab="Länge Kronblatt",
                main="Blattlängen Schwertlilie (Sunflowerplot)")
```

Einen Sonderfall bilden Zeitreihen-Daten, da hierbei die Zeitangaben als Werte für die X-Achse genutzt werden. Oftmals werden bei Zeitreihen die einzelnen Punkte durch eine Linie verbunden. Bei Zeitreihen und anderen Daten, deren Reihenfolge an sich bedeutsam ist, muss also keine zweite Variable angegeben werden. Die Plot-Funktion erstellt einfach eine X-Achse mit den Zeilenrängen 1 bis n. Datenobjekte der Klasse `ts` sind speziell für Zeitreihen gedacht und können die historischen Zeitstellen als Attribute enthalten, so dass die X-Achse wie im folgenden Beispiel au-

tomatisch die richtige Beschriftung erhält.[2]




```
> data(USAccDeaths)
> plot(USAccDeaths,              # Zeitreihe
      type="b",                 # Typ des Plots
                                # "p" Punkte
                                # "l" Linie
                                # "b" Punkte und Linie
                                # "n" Keine Anzeige
      pch=16,                   # Symbol für Datenpunkte
      cex=1,                    # Vergrößerungsfaktor Symbole
      col="#425a10",            # Farbe der Punkte
      bty="l",                  # Rahmen nur links und unten
      xlab="Zeit",
      ylab="Anzahl Unfalltote",
      main="Unfalltote in den USA 1973 -- 1979")
```

[2] Ein Aufruf der Plot-Funktion mit einem einfachen Vektor, den man über `unclass(USAccDeaths)` erhält, liefert mit Ausnahme der fehlenden Beschriftungen das gleiche Ergebnis.

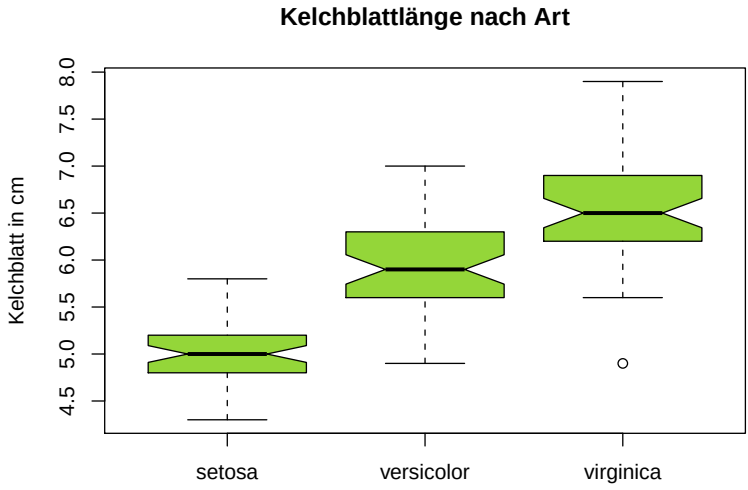
USAccDeaths

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct
1973	9007	8106	8928	9137	10017	10826	11317	10744	9713	9938
1974	7750	6981	8038	8422	8714	9512	10120	9823	8743	9129
1975	8162	7306	8124	7870	9387	9556	10093	9620	8285	8466
1976	7717	7461	7767	7925	8623	8945	10078	9179	8037	8488
1977	7792	6957	7726	8106	8890	9299	10625	9302	8314	8850
1978	7836	6892	7791	8192	9115	9434	10484	9827	9110	9070

	Nov	Dec
1973	9161	8927
1974	8710	8680
1975	8160	8034
1976	7874	8647
1977	8265	8796
1978	8633	9240

3.6.2 Boxplots

Der Boxplot dient zum Vergleich der Verteilung einer numerischen Variable in verschiedenen Subpopulationen. Im Beispiel unten wurde für jede Schwertlilienart ein Boxplot der Kelchblattlänge gezeichnet.



Die Breite der Box wurde hier so gewählt, dass sie der Wurzel der Fallzahl in der Gruppe entspricht (da die Gruppen gleich groß sind, sind keine Unterschiede zu erkennen). Die dunkle Linie innerhalb der Box zeigt die Lage des Medians in der jeweiligen Gruppe an. Die Einschnürung kann mit `notch=TRUE` erzeugt werden und stellt das 95% Konfidenzintervall des Medians dar. Überschneiden sich die eingeschnürten Bereich zweier Gruppen, liegt vermutlich kein signifikanter Unterschied vor. Die oberen und unteren Begrenzungen der Box entsprechen dem 1. und 3. Quartil, die Box umfasst also 50% der Fälle. Die Whisker („Schnurrbärte“) entsprechen dem Wert, der am weitesten vom Median entfernt liegt, aber nicht weiter als das 1.5fache des Interquartilabstandes (der Faktor kann mit `range=` angepasst werden). Werte die außerhalb dieses Bereiches liegen, werden durch Symbole als Ausreißer kenntlich gemacht.

Der Aufruf der Boxplot-Funktion unterscheidet sich ein wenig von den bisher betrachteten Funktionen. Sie erwartet ein Objekt der Klasse `formular` als Argument. Ein `formula`-Objekt wird nach folgendem Schema definiert: numerische Variable ~ Gruppierungsvariablen (durch + getrennt).

```
> formel <- iris$Sepal.Length ~ iris$Species
```

Die Formel kann auch direkt in der `boxplot`-Funktion definiert werden – darauf wurde an dieser Stelle der Übersichtlichkeit halber verzichtet.

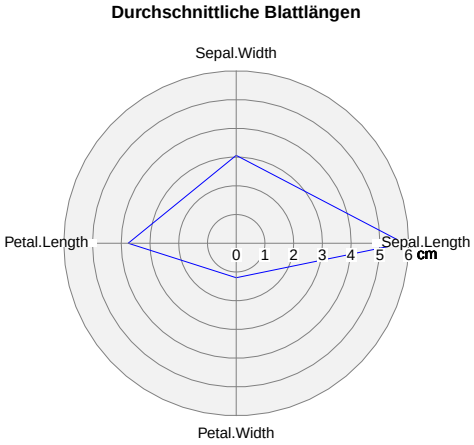
```
> boxplot(formel,                 # Formel von oben
          range=1.5,               # Entfernung für die Whisker
                                   #  max. range*Interquartilsabstand
          width=NULL,              # Breite der Box
          varwidth=TRUE,           # Breite entspricht der Wurzel der
                                   #  Anzahl der Fälle in der Gruppe
          notch=TRUE,              # Einschnürungen einzeichnen
          outline=TRUE,            # Ausreißer einzeichnen
          boxwex=0.8,              # Skalierung der Box
          staplewex=0.5,           # Skalierung der Whisker
          border="black",          # Randfarbe (auch Vektor)
          col="#94d639",           # Farbe für die Boxen
          horizintal=FALSE,        # horizontale Anordnung
          add=FALSE,               # In bestehenden Plot einzeichnen
          at=NULL,                 # X-Koordinaten der Boxen
                                   #  bei add=TRUE festlegen
          ylab="Kelchblatt in cm",
          main="Kelchblattlänge nach Art")
```

Auszug aus dem Iris-Datensatz

	Sepal.Length	Species
68	5.8	versicolor
91	5.5	versicolor
82	5.5	versicolor
40	5.1	setosa
142	6.9	virginica
20	5.1	setosa
6	5.4	setosa
92	6.1	versicolor
18	5.1	setosa
7	4.6	setosa

3.7 Spiderwebplots

Spiderweb oder Radial-Plots erlauben die Darstellung numerischer Werte mehrerer Variablen. Im unten dargestellten Beispiel sind die mittleren Längen und Breiten der Blütenblätter der Schwertlilie eingezeichnet. Die Spitzen des Polygons geben den Wert für die entsprechende Variable an. Radial-Plots können so eine größere Zahl von numerischen Werten übersichtlich darstellen. Durch die charakteristischen Formen, die das durch die Werte gebildetete Polygon annimt, kann diese Art von Grafik auch zum Vergleich zwischen verschiedenen Subpopulationen oder Variablen dienen.



Eine Funktion zum Zeichnen von Radial-Plots stellt das Paket `plotrix` bereit und kann mit folgendem Aufruf installiert und geladen werden:

```
> install.packages("plotrix")
> library(plotrix)
```

In diesem Beispiel sollen die mittleren Kelch- bzw. Kronblattlängen und -weiten visualisiert werden. Dazu werden zunächst die Mittelwerte berechnet:

```
> mean.iris <- mean(iris[,1:4])
```

Der Plot selbst folgt dem üblichen Schema. Die für diese Art von Darstellung verfügbaren Gestaltungsoptionen sind im folgendem R-Code erläutert:

```
> radial.plot(lengths=mean.iris,        #  Daten
              rp.type="p",              # p: Polygon
                                        # r: Linien
                                        # s: Symbole
              line.col="blue",          # Farbe der Linien
              labels=names(iris[,1:4]), # Achsenbeschriftung
              label.prop=1.1,           # Abstand der Label
              radlab=F,                 # Label am Kreis ausrichten
              radial.lim=c(0,6),        # Start/Ende Radius
              poly.col="transparent",   # Farbe für das Polygon
              grid.col=gray(0.5),       # Gitterfarbe
              grid.bg=gray(0.95),       # Gitterhintergrund
              grid.left=T,              # Gitterbeschriftung links
              grid.unit="cm",           # Gitter Einheit
              point.symbols=20,         # falls "s",Symboltyp
              point.col="red",          # Symbolfarbe
              lty="solid",              # Linientyp
              lwd=1,                    # Linienstärke
              mar=c(4,4,5,5),           # Rändereinstellung
              main="Durchschnittliche Blattlängen")
```

mean.iris

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

4 Farben & Formen

4.1 Definition von Farben

Farbangaben können in verschiedenen Farbmodellen definiert werden. Eine übliche und geläufige Möglichkeit ist das RGBA-Model, in dem Farben aus unterschiedlichen Anteilen der Grundfaben Rot, Grün und Blau gemischt werden. Das A in RGBA steht für den Alpha-Kanal, der die Tranzparenz einer Farbe definiert. In `R` dient der Befehl `rgb` zur Definition von Farben nach diesem Farbmodel. Weitere Farbmodelle und die entsprechenden Funktion sind `hsv` (Hue-Saturation-Value) oder `hcl` (Hue-Chroma-Luminance).

Unabhängig davon, mit welcher Funktion Farben definiert wurden, speichert `R` Farbwerte in hexadezimaler Form, wie sie beispielsweise auch in HTML-Seiten Verwendung finden. Dabei werden die Prozentangaben durch eine Zahl im Intervall von 0 bis 255 bzw. 0 bis FF beschrieben. Die Farbe Rot wird zum Beispiel durch `"#FF0000"` angegeben, in dezimaler Form entspricht dies (255,0,0) – also 100% Rot-Anteil und 0% Blau- und Grün-Anteil. Die meisten Bildverarbeitungsprogramme sind in der Lage Farbewerte in hexadezimaler Form anzuzeigen, so dass auch eine „visuelle Farbmischung“ in diesen Programmen möglich ist.

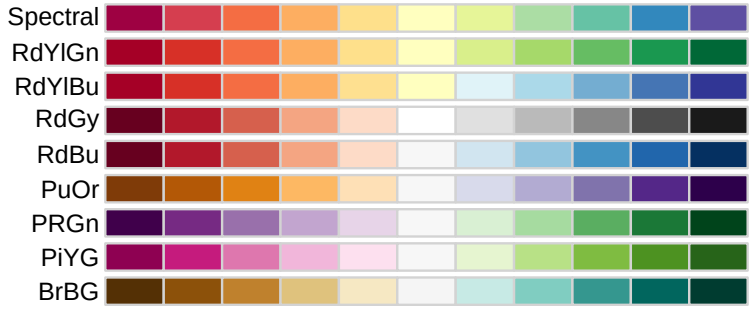
```
violett <- rgb(
    red  =0.7,          # Rot-Anteil
    blue =0.9,          # Grün-Anteil
    green=0,            # Blau-Anteil
    alpha=1,            # Grad der Tranzparenz
    names = NULL,       # Namen für die erzeugten Farben
    maxColorValue = 1)  # Höchstwert für die Farbangaben
                        #  z.B. [0:1] oder [0:255]
```

4.2 Farbpaletten

Das Paket `RColorBrewer` enthält vordefinierte Farbpaletten, die aufeinander abgestimmte Farbtöne bereitstellen. Der Befehle `display.brewer.all()` gibt eine Übersicht über die bereitsgestellten Paletten. Um eine dieser Farbpaletten zu nutzen, erzeugt man mit `brewer.pal(n, name)` einen Vektor mit `n` Farbwerten aus der jeweiligen Palette. So liefert der Aufruf von `brewer.pal(11,"Spectral")` folgende Farbwerte:

```
[1] "#9E0142" "#D53E4F" "#F46D43" "#FDAE61"
[5] "#FEE08B" "#FFFFBF" "#E6F598" "#ABDDA4"
[9] "#66C2A5" "#3288BD" "#5E4FA2"
```

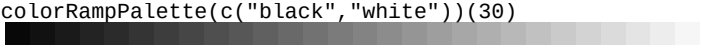
In der nebenstehenden Grafik sind exemplarisch einige Farbpaletten dargestellt, an erster Stelle findet sich auch die soeben erstellte Palette `Spectral`.



4.3 Farbverläufe

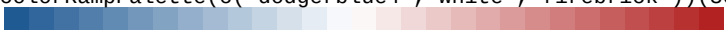
Farbverläufe lassen sich auf einfache Weise mit `colorRampPalette` erstellen. Die Funktion hat ungewöhnlicherweise zwei Klammerpaare zur Angabe von Optionen. Das erste Klammerpaar nimmt die Start-, Zwischen- und Endfarben als Vektor der Form `c("1. Farbe","2. Farbe")` entgegen. Zusätzlich kann mit den Argumenten `space="rgb"` oder `"lab"` der Farbraum und mit `interpolate="linear"` oder `"spline"` die Berechnung der Farbverläufe beeinflusst werden. Im zweiten Klammernpaar wird anschließend die Anzahl der Farbstufen festgelegt.

```
colorRampPalette(c("black","white"))(30)
```



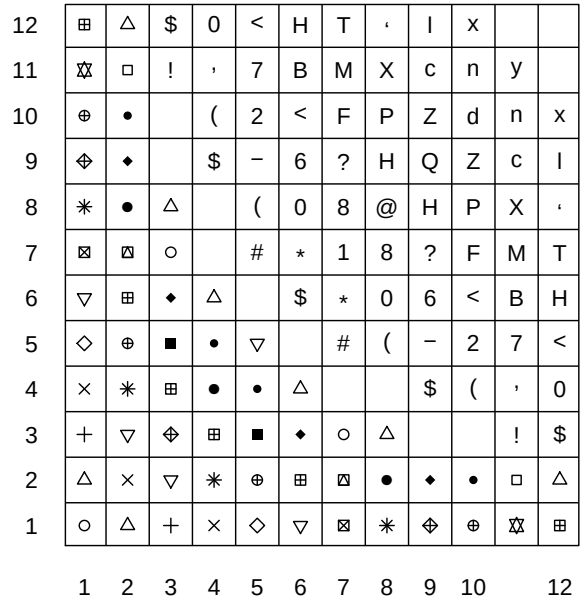
```
colorRampPalette(c("dodgerblue4","white"))(30)
```

```
colorRampPalette(c("dodgerblue4","white","firebrick"))(30)
```



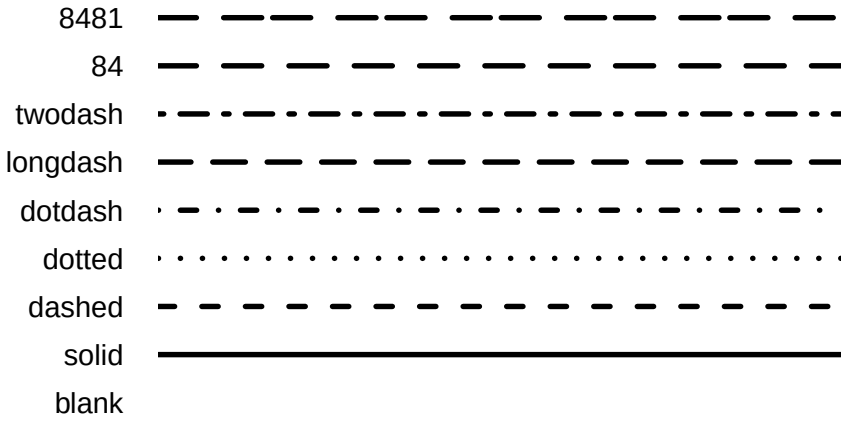
4.4 Symbol- und Linientypen

In der unten stehenden Grafik sind alle in `R` verfügbaren Symbole abgebildet. Sie können über den Schalter `pch=Symbolnummer` ausgewählt werden. Die Nummer für ein bestimmtes Symbol kann aus nachfolgender Übersicht leicht ermittelt werden: Die Symbolnummer ergibt sich aus dem Produkt von Zeilennummer und Spaltennummer. So kann ein ausgefülltes Quadrat beispielsweise mit der Nummer $5 \cdot 3 = 15$ ausgewählt werden. Buchstaben können auch einfacher angegeben werden: `pch="M"`.

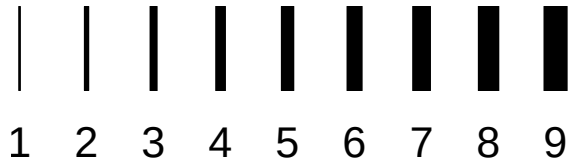


Die Größe der Symbole wird über den Parameter `cex` gesteuert. Der Standardwert ist `cex=1`, ein Wert von 0.5 halbiert die Größe der Symbole und ein Wert von 2 verdoppelt die Symbolgröße.

Linienstile werden über die Option `lty` ausgewählt. Die Grafik unten zeigt die standardmäßig verfügbaren Stile, die am einfachsten über ihren Namen ausgewählt werden können: `lty="dashed"`. Linienstile können auch auf einfache Weise selbst definiert werden. Die ersten beiden Linien der Grafik zeigen selbstdefinierte Linientypen. Die erste Linie wird mit `lty="8481"` erstellt. Die Bedeutung der Zahlen ist einfach zu erkennen: Die erste 8 steht für einen acht Einheiten langen Strich, die 4 zeichnet eine vier Einheiten lange "Pause", die nächste 8 wieder eine Linie der entsprechenden Länge und die 1 erzeugt eine kurze Pause. Danach wird diese Abfolge für den Reste der zu zeichnenden Linie wiederholt.



Die Einheiten für die Linienstile entsprechen dem Wert für `lwd`, der Linienstärke. Dieser Parameter wird genauso wie `cex` behandelt und ist standardmäßig 1.



4.5 Schriften

Schriften unter Windows

Die in den Grafiken verwendeten Schriften lassen sich wie andere Standardeinstellungen ebenfalls ändern. In `R` sind drei Schriftschnitte direkt verfügbar: `serif`, `sans` und `mono`. Diese können über die Option `family` ausgewählt werden.

Unter Windows lassen sich zusätzlich auf einfache Weise im System installierte Truetype-Schriften nutzen. Dazu können mit der Funktion `windowsFonts()` Schriftfamilien definiert werden. Dazu muss jeweils der Name der neuen Schriftfamilie angegeben werden (im ersten Beispiel unten „A“) und die ausgewählte Schriftart (im Beispiel `windowsFont("Arial Black")`):

```
windowsFonts(A=windowsFont("Arial Black"),          # Schriftfamilie A
             B=windowsFont("Bookman Old Style"),    # mit der Schrift
             C=windowsFont("Comic Sans MS"),        # Arial Black usw.
             S=windowsFont("Symbol"))
```

```
> plot(1:10,1:10,type="n") #Plot erstellen
```

Folgende Syntax zeigt, wie die neu definierten Schriften in einer Grafik verwendet werden können:

```
> text(3,3,              "Standard")          # Text in Standardschrift
> text(4,4, family="A", "Arial Black")       # Text in Arial Black
> text(5,5, family="B", "Bookman Old Style")
> text(6,6, family="C", "Comic Sans MS")
> text(7,7, family="S", "Symbol")
```

Schriften mit Cairo

Georgia Regular
Georgia Bold
Georgia Italic
Georgia BoldItalic
Γεοργια Σψμβολε

Das in Abschnitt 2.1 vorgestellt Ausgabegerät `Cairo` ermöglicht es, unter allen unterstützen System individuelle Schriften zu nutzen. Das Erweiterungspaket kann mit `library(Cairo)` geladen werden. Das Ausgabegerät muss mit `CairoX11()` explizit gestartet werden, da ansonsten weiterhin das standardmäßige Ausgabegerät genutzt wird.

Der Befehl `CairoFontMatch(":", sort=T)` liefert eine Übersicht über alle nutzbaren Systemschriften. Die Definition der Standardschriften für die Grafikausgabe erfolgt über die Funktion `CairoFonts`. Dabei muss für jeden der fünf Schriftschnitte sowohl Schriftart und -schnitt angeben werden.

```
> CairoFonts(
            regular="Georgia:style=Regular",
            bold="Georgia:style=Bold",
            italic="Georgia:style=Italic",
            bolditalic="Georgia:style=BoldItalic",
            symbol="Georgia:style=Regular")
```

Die neu definierte Schrift wird automatisch im richtigen Schnitt zur Beschriftung der Plots genutzt, beispielsweise wird der fette Schnitt für die Überschrift genutzt. Die manuelle Auswahl der Schriftschnitte funktioniert mit dem `font`-Parameter, der sowohl im `plot`-Aufruf als auch bei einzelnen Zeichenfunktionen wie `text` oder `axis` genutzt werden kann.

Der folgende Beispiel-Code erzeugt eine Übersicht der Schnitte der Schriftart Georgia, welche auf der vorhergehenden Seite abgebildet ist:

```
> # Leeren Plot erstellen
> plot(1:6,1:6, type="n", bty="n", axes=F, xlab="", ylab="")
> # Text einzeichnen
> text(2,5, pos=4, font=1, "Georgia Regular")    # Standardformatierung
> text(2,4, pos=4, font=2, "Georgia Bold")       # Fett
> text(2,3, pos=4, font=3, "Georgia Italic")     # Kursiv
> text(2,2, pos=4, font=4, "Georgia BoldItalic") # Fett-Kursiv
> text(2,1, pos=4, font=5, "Georgia Symbole")    # Symbole
```

5 Plots kombinieren

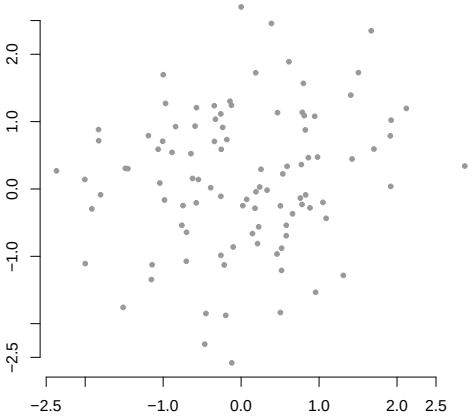
Neben den bereits vorgestellten high-level Plots lassen sich in R auf einfache Weise verschiedene Grafikfunktionen zu neuen Grafiken kombinieren. Auch wenn mit der Funktion `plot.new()` mit einem vollkommen leeren Plot begonnen werden kann, ist es oft sinnvoll, einen der Standardplots als Ausgangspunkt zu nehmen. Im folgenden Beispiel werden ausgehend von einem Streudiagramm die wichtigsten Funktionen vorgestellt.

Für das Streudiagramm werden zuerst die Variablen `x`, `y` und `z` mit jeweils 100 normalverteilten Zufallszahlen erzeugt.

```
> set.seed(19)
> x <- rnorm(100)          # Zufallszahlen erzeugen
> y <- rnorm(100)
> z <- rnorm(100) * 72     # größere Zufallszahlen
```

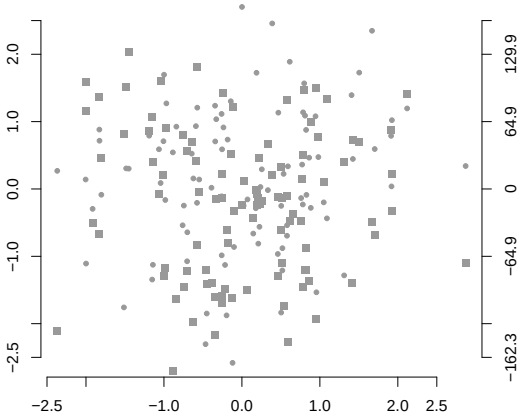
Das Streudiagramm wird zunächst ohne Achsen und weiterer Beschriftungen erstellt.

```
> plot(x=x,                 # X-Variable
       y=y,                 # Y-Variable
       pch=20,              # Symbol für Datenpunkte
       col=gray(0.6),       # Farbe für Datenpunkte
       xaxt="n",            # keine X-Achse
       yaxt="n",            # keine Y-Achse
       xlab="",             # keine Beschriftung
       ylab="",
       bty="n")             # keine Box
```



Als nächstes werden die X- und Y-Achse eingezeichnet. Die Anordnung der Achsen wird mittels `side=` gesteuert, jede Seite des Plots entspricht dabei einer Nummer: 1 unten, 2 links, 3 oben und 4 rechts. Es gibt zahlreiche Optionen, um dass Aussehen der Achsen zu beeinflussen. Im Beispiel unten wurden die Positionen für die Achsenmarkierungen, die Farbe und der Linientyp festgelegt. Einen Überlick über weitere Optionen liefert wie immer `help(axis)`.

```
> axis(side=1,           # 1 ist die X-Achse
       at=c(-2.5, -2,    # Vektor mit den Punkten,
            -1, 0,       # an denen eine Markierung
             1, 2, 2.5), # gesetzt werden soll
       col="black",      # Farbe der Achse
       lty="solid")      # Linientyp
> yat <- c(-2.5,-2,-1, 0,1,2,2.5)
> axis(side=2,           # 2 ist die Y-Achse
       at=yat,
       col="black",
       lty="solid" )
```



Der Plot kann auch um eine zweite X- oder Y-Achse ergänzt werden, um bspw. eine Variable mit einer unterschiedlichen Skalierung in einer Grafik anzuzeigen. Die Skalierung der zusätzlichen Variable muss manuell durchgeführt werden und kann über einen einfachen Drei-Satz erreicht werden:

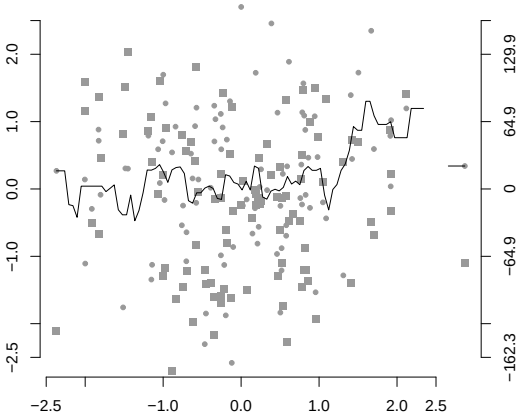
```
> zfactor <- max(abs(y))/max(abs(z))
> zskal <- zfactor * z
```

Die Funktion `points` zeichnet Punkte in das bestehende Koordinatensystem ein. Für die Gestaltung der Datenpunkte stehen alle für Streudiagramme vorgesehenen Optionen zur Verfügung:

```
> points(x=x, y=zskal, pch=15, col=gray(0.6))
```

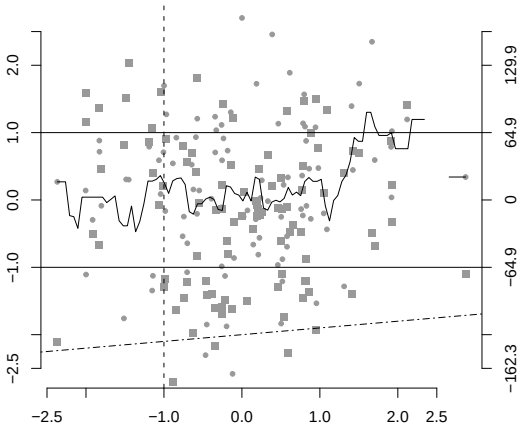
Die rechte Y-Achse kann wiederum mit dem `axis`-Befehl eingezeichnet werden. Die ursprüngliche Skalierung der Variable `z` wird nun in der Beschriftung der Markierungen berücksichtigt. Gleichzeitig werden die Achsenpositionen (`at`) der ersten Y-Achse übernommen und unter Berücksichtigung der vorgenommenen Skalierung von `z` die entsprechenden Label berechnet.

```
> y4lab  <- c(-2.5,-2,-1,0,1,2,2.5)/zfactor
> y4lab  <- round(y4lab, digits=1)
> axis(side=4,          # 2 ist die Y-Achse
      at=yat,           # Position der Markierungen (vorherige Seite)
      labels=y4lab)     # Beschriftung der Markierungen
```



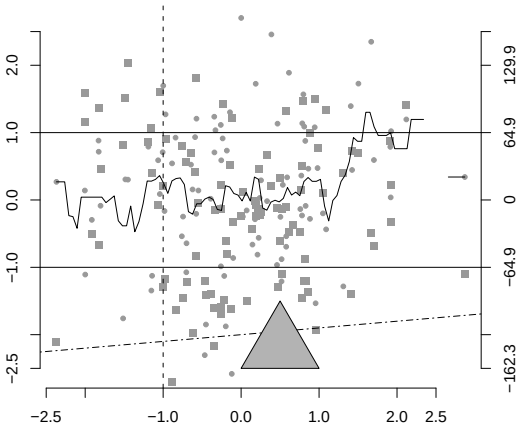
In ein Streudiagramm können problemlos Liniendiagramme eingefügt werden. Im folgenden Beispiel wird eine lokale Regressionsfunktion für `x` und `y` berechnet und in die Grafik eingezeichnet.

```
> sm <- ksmooth(x=x, y=y)      # Lokale Regressionsfunktion berechnen
> lines(x=sm$x,                # X-Koordinaten der Linienpunkte
        y=sm$y,                # Y-Koordinaten der Linienpunkte
        lwd=1,                 # Linienstärke
        lty="solid")
```



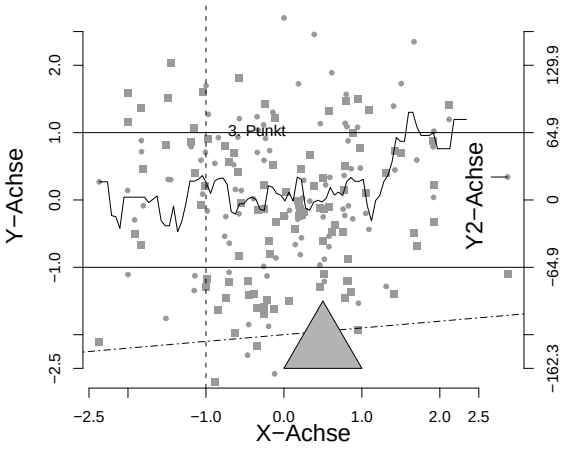
Um ein Linienraster oder andere Markierungen einzufügen, eignet sich die Funktion `abline`. Mit ihr ist es einfach horizontale, vertikale und beliebige andere Geraden durch den Zeichnungsbereich zu führen:

```
> abline(v=-1, lty="dashed")           # vertikale Linie an der Stelle 1
> abline(h=c(-1,1), lty="solid")       # horizontale Linie bei -1 und 1
> abline(a=-2, b=0.1, lty="twodash")   # Linie g(x)=a+b*x
```



Polygone, also von einer Linie umschlossene Flächen, lassen sich ebenso einfach zeichnen. Die Form des Polygons wird durch die Koordinaten der Eckpunkte bestimmt. Die farbliche Gestaltung wird über die Parameter `col` für die Füllfarbe und `border` für die Umrandungsfarbe gesteuert.

```
> polygon(x=c(0,0.5,1),          # Koordinaten der Ecken
          y=c(-2.5,-1.5,-2.5),   # 1. Ecke (0,-2.5)
          col=gray(0.7),         # Füllfarbe
          border="black")        # Randfarbe
```



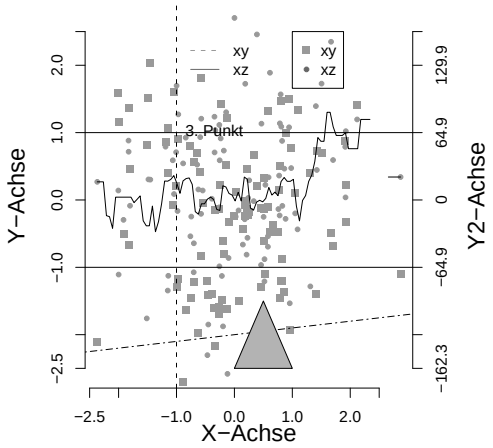
Die noch fehlende Achsenbeschriftung lässt sich mit dem Befehl `mtext` erzeugen. Die Positionierung erfolgt dabei mit den gleichen Parametern wie bei den Achsen selbst:

```
> par(mar=c(5,4,4,8)+0.1) # Platz für rechte Beschriftung schaffen
> mtext(side=1, "X-Achse", line=2, cex=1.5)
> mtext(side=2, "Y-Achse", line=3, cex=1.5)
> mtext(side=4, "Y2-Achse",line=3, cex=1.5)
```

Parallel dazu kann mit der Funktion `text()` an beliebige Positionen Text eingefügt werden.[1]

```
> text(x=x[3],                  # X-Koordinate des 3. Punktes von x
       y=y[3],                  # X-Koordinate des 3. Punktes von y
       labels="3. Punkt",       # Der Text
       adj=c(0,0),              # Verschiebung (X-Richtung, Y-Richtung)
       pos=1)                   # Text 1=unten, 2=links, 3=oben,
```

[1] Tipp: mit `locator` können Koordinaten mit der Maus ausgewählt werden. Einfach `locator()` ohne Argumente aufrufen, mit einem Linksklick eine oder mehrere Stellen im Plot auswählen und mit Rechtsklick beenden. Die Koordinaten der einzelnen Linksklicks werden angezeigt. Ruft man die Funktion mit `kor <-locator()` auf, werden die Koordinaten in einem Objekt gespeichert und können in den Grafikfunktionen weiterverwendet werden.



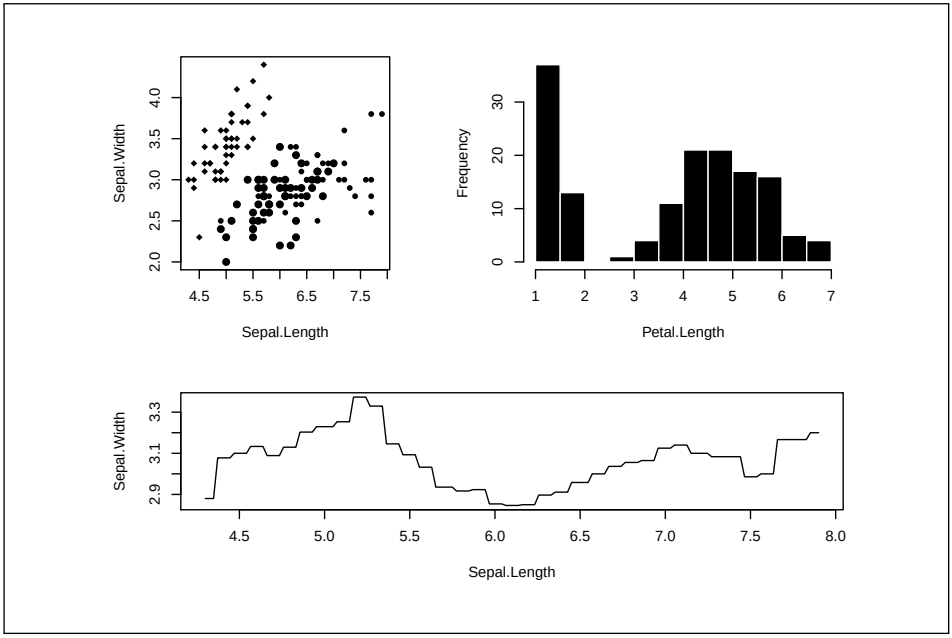
Abschließend kann die Grafik noch um eine Legende erweitert werden. R stellt dazu die Funktion `legend` bereit. Die erste Legende nutzen wir zur Erläuterung der Darstellung der Punktwolke.

```
> legend(x=1,                          # Koordinaten für die Legende
         y=2.5,
         legend=c("xy", "xz"),         # Vektor mit dem Text
         pch=c(15,20),                 # Zum Text passende Symbole
         col=c(gray(0.6),gray(0.4)))   # Farbe der Symbole
```

Die zweite Legende zeigt beispielhaft die notwendigen Optionen zur Beschriftung von Linien.

```
> legend(x=-1,                         # Koordinaten für die Legende
        y=2.5,
        legend=c("xy", "xz"),          # Vektor mit dem Text
        lty=c("dashed","solid"),       # Zum Text passende Linientypen
        col=c(gray(0.6),gray(0.4)),    # Farbe der Symbole
        bty="n")                       # ohne Rahmen
```

5.1 Mehrfach-Plots



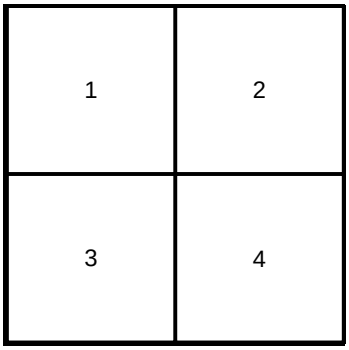
Um mehrere Plots in eine Grafikumgebung zu zeichnen, bestehen mehrere Möglichkeiten: a) die `par`-Option `mfrow`, b) die Funktion `split.screen` und c) die Funktion `layout`. Letztere ist in der Bedienung recht einfach und wird im Weiteren behandelt.

Mit `layout()` wird das Layout für den Mehrfachplot festgelegt. Die Funktion erfordert als Argument eine Matrix, welche die Anordnung und Reihenfolge für die einzelnen Grafiken enthält. Eine solche Matrix kann mit dem Befehl `rbind` erstellt werden. Die Funktion bietet sich an, da mit ihr Zeilenvektoren zu eine Matrix zusammengefügt werden und so die Struktur der Matrix schon in der Syntax deutlich wird.

```
> l <- rbind(c(1,2),
             c(3,4))
> l
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

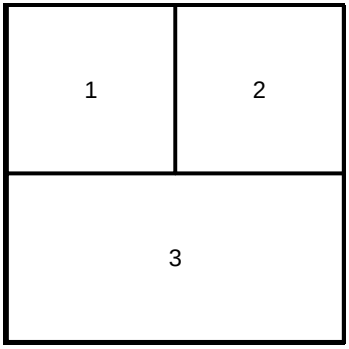
Der Befehl `layout(l)` „aktiviert“ das Layout entsprechend der soeben definierten

Matrix „l“. Im Folgenden werden die „Kacheln“ entsprechend der Nummerierung mit Grafiken gefüllt. Nach jedem Aufruf von `plot` wird zur Kachel mit der nächstgrößeren Zahl gewechselt, so dass beim nachfolgenden `plot`-Aufruf diese gefüllt wird. Sind alle Kacheln gefüllt, wird bei einem erneuten Aufruf von `plot` die erste Kachel überschrieben. Eine Vorschau auf die Aufteilung der Zeichenfläche liefert die Funktion `layout.show`, die als Parameter die Anzahl der einzelnen Grafiken benötigt.




```
> layout(l)
> layout.show(4)
```

Einzelne Grafiken können sich auch über mehrere Kacheln erstrecken, dazu wird die entsprechende Nummer einfach in mehrere Positionen der Matrix eingetragen.

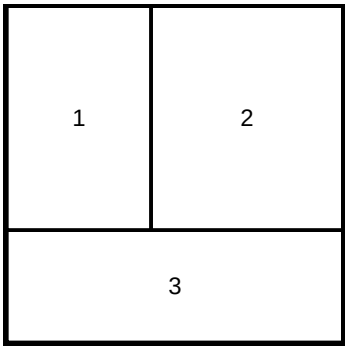



```
> l <- rbind(c(1,2),
             c(3,3))
> l
     [,1] [,2]
[1,]    1    2
[2,]    3    3
> layout(l)
> layout.show(3)
```

Ohne Angabe weiterer Optionen wird der zur Verfügung stehende Raum gleichmäßig zwischen den einzelnen Grafiken aufgeteilt. In der `layout`-Funktion kann über die Optionen `widths` und `heights` für jede Kachel eine bestimmte Breite und Höhe angegeben werden. Dazu wird für Höhe und Breite jeweils ein Vektor erstellt, der das Verhältnis der einzelnen Kacheln zueinander ausdrückt.

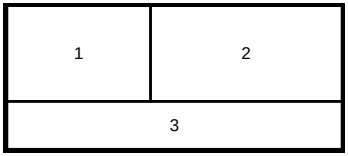
Der Vektor `c(2,4,6)` im unten stehenden Beispiel lässt sich wie folgt lesen: Die Kachel 1 soll die Breite 2 haben, die Kachel 2 soll doppelt so breit sein und die Ka-

chel 3 soll so breit sein wie die anderen Kacheln zusammen (also 6 Einheiten). Die Reihenfolge der Angaben für die Breite erfolgt nach folgendem Schema: 1. Spalte/ 1. Zeile, 2. Spalte/1.Zeile, 1. Spalte/2. Zeile usw. Die Angaben zur Höhe der einzelnen Zeilen folgen dementsprechend dem Muster: 1. Zeile, 2. Zeile usw.




```
> l <- rbind(c(1,2),
             c(3,3))
> layout(l,
        heights=c(2,1),
        widths=c(3,4,7))
> layout.show(3)
```

In der Standardeinstellung wird bei der Breiten- und Höhenangabe jeweils nur das Verhältnis der Zahlen betrachtet. Sollen die absoluten Unterschiede zwischen den Angaben berücksichtigt werden, muss die Option `respect=T` gesetzt werden. Diese sorgt dafür, dass die numerischen Werte für `height` und `widths` dieselbe Einheit aufweisen.




```
> layout(l,
        heights=c(2,1),
        widths=c(3,4,7),
        respect=T)
> layout.show(3)
```

Soll zwischen den einzelnen Grafiken ein Leerraum eingefügt werden, so muss dies schon in der Layoutmatrix definiert werden. Im Beispiel unten wurde eine leere Zeile zwischen den Grafiken 1/2 und 3 eingefügt, indem in der Matrix eine Zeile mit Nullen eingefügt wurde. Alle mit Null besetzten Positionen in der Matrix werden später nicht mit einer Grafik besetzt. Die Höhe und Breite dieser leeren Kacheln lässt sich genauso über `widths` und `heights` steuern.


```
> l <- rbind(c(1,2),
             c(0,0),
             c(3,3))
> l
     [,1] [,2]
[1,]    1    2
[2,]    0    0
[3,]    3    3
> layout(l,
        heights=c(2,0.5,1),
        widths=c(2,4,6),
        respect=T)
> layout.show(3)
```

6 Lattice

Lattice ist ein R-Paket zur Visualisierung multivariater Daten. Es basiert auf dem grid-Paket, einem alternativem Grafiksubsystem von R. Die Nutzung der enstsprechenden Funktionen für Standardgrafiken unterscheidet sich jedoch kaum von denen des Basis-Grafiksystems, da die Unterschiede vornehmlich bei den low-level Funktionen von grid sichtbar werden. lattice selbst beinhaltet keine low-level Befehle, stattdessen müssen die Funktionen des Grafiksystems genutzt werden. Eine Übersicht über die Zeichenobjekte im grid-System bietet help.search("grid") oder die Homepage des Hauptenwicklers Paul Murrel[1].

Ein beispielhafter Aufruf einer lattice-Funktion zeigt einen sichtbaren Unterschied zu den traditionellen Grafikfunktionen:

```
>   data(iris)
>   histogram( ~ Petal.Length + Petal.Width | iris$Species,
              data=iris)
```

Im Aufruf werden die zu plottenden Variablen in Form eines formular-Objekts angegeben: xyplot(y ~ x) entspricht dem Aufruf plot(x,y). Bei Funktionen mit nur einem Argument muss ebenfalls die Tilde genutzt werden: histogram(~ x) entspricht hist(x). Zusätzlich können auch weitere Variablen mit einem + aufgenommen werden. Bei kategorialen Variablen wird für jede Kategorie einfach ein weiterer Plot erzeugt (siehe folgende Abbildung).

Bei numerischen Merkmalen werden die Werte der unterschiedlichen Variablen beispielsweise farblich markiert:

```
xyplot(Sepal.Width + Sepal.Length ~
            Petal.Width + Petal.Length,
       data=iris,
       auto.key=T)
```

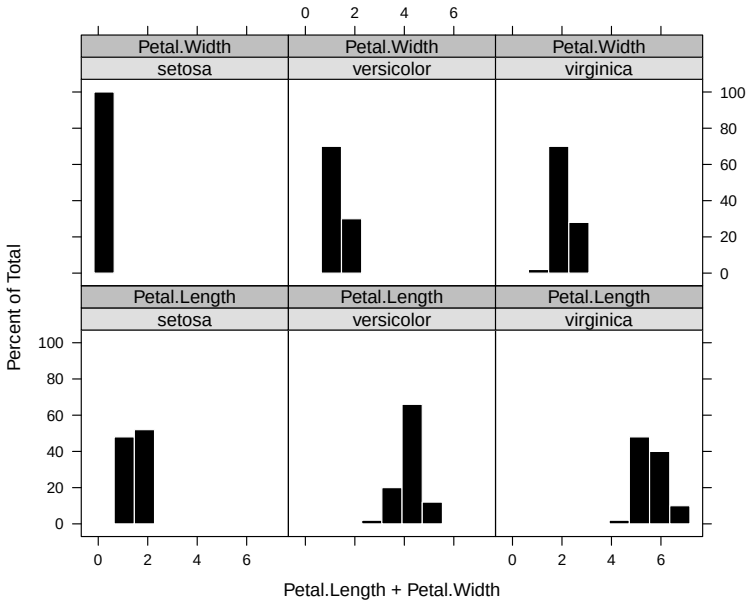
Im Gegensatz zur Definition von Regressionsmodellen wird bei der Aufnahme weiterer Variablen nicht zwischen * und + unterschieden.

[1] http://www.stat.auckland.ac.nz/~paul/grid/grid.html

Eine Besonderheit im `lattice`-Paket ist der Hochstrich: `xyplot(y ~ x|z)`, mit dessen Hilfe die Grafikausgabe auf eine oder mehrere Variablen konditioniert werden kann. Ein Beispiel liefert die folgende Grafik mit Daten des `iris`-Datensatzes, bei der die Kron- und Kelchblattlänge konditioniert auf die jeweilige Spezies dargestellt werden. Für jede Spezies und Variable wird ein eigener Plot erstellt.

Die Aufteilung der Plots in Spalten und Zeilen kann über den Paramter `layout=c(spalten,zeilen)` gesteuert werden:

```
xyplot(Sepal.Width ~ Petal.Length | Species,
       data=iris,
       layout=c(1,3),    # Anordnung der Plots
       aspect=c(0.5))    # Seitenverhältnis der Plots
```



Die Einstellungen für Farben, Linienstärke oder Symbole lassen sich zentral für die gesamte Sitzung definieren. Dabei ist es hilfreich, unterschiedliche Themen zu definieren. Dazu wird eine Liste erstellt, die die Namen der Parameter und deren Einstellungen enthält:

```
> druckthema <- list(
     plot.line = list(col = "black"),              # Farbe der Linien
     plot.polygon = list(col = grey(0/8),          # Farbe der Balken
                         border="white"),
```

```
    plot.symbol = list(col = grey(4/8)),         # Farbe der Symbole
    dot.symbol = list(col = grey(0/8)),          # Symbole beim Dotchart
    box.rectangle = list(col = grey(0/8)),       # Farbe Boxplot (box)
    box.umbrella  = list(col = grey(0/8)),       #  und (whisker)
    strip.background = list(col = grey(7:1/8))) # Farbe Überschriften
```

Eine Übersicht über alle Parameter und deren aktuelle Standardeinstellungen liefert `trellis.par.get`. Eine grafische Darstellung der Einstellungen erhält man mit `show.settings`.

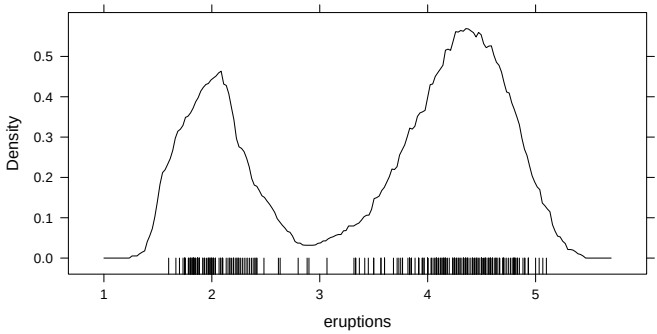
Um ein Thema zu aktivieren, kann entweder mit `standard.theme()` ein Theme für alle Ausgabegeräte gesetzt werden oder mit der Funktion `trellis.device()` die Themen für einzelne Ausgabegeräte angepasst werden:

```
> trellis.device(pdf, color=T, theme=druckthema)
```

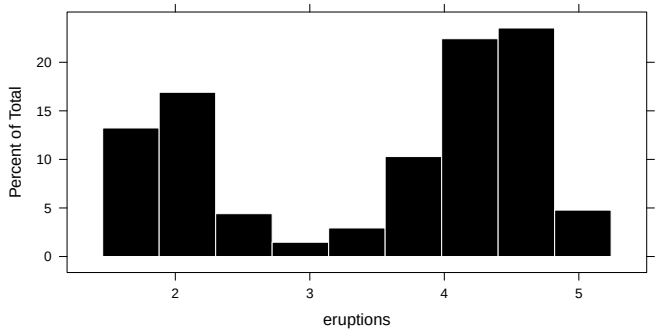
6.1 Lattice Standarddiagramme

Auf den folgenden Seiten werden die bereits bekannten Standarddiagramme mit ihren entsprechenden `lattice`-Aufrufen aufgeführt. Wichtige Optionen werden in der Syntax erläutert, falls sie sich von der Funktion im Basissystem unterscheiden.

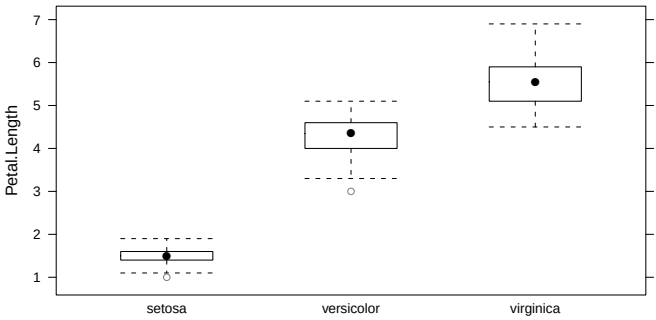
```
> data(faithful)
> densityplot(~ eruptions,
              data = faithful,
              kernel="rect", bw=0.2,
              plot.points="rug", n=200,
              col="black")
```



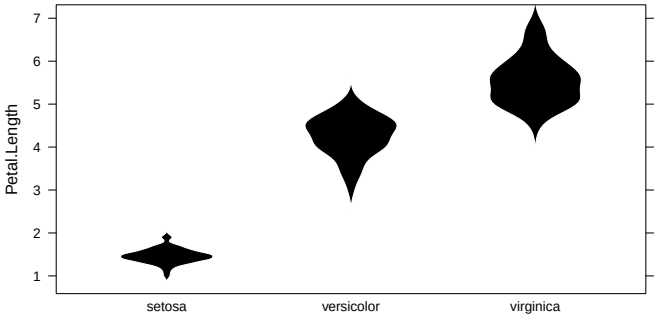

```
> histogram(~ eruptions, data = faithful)
```



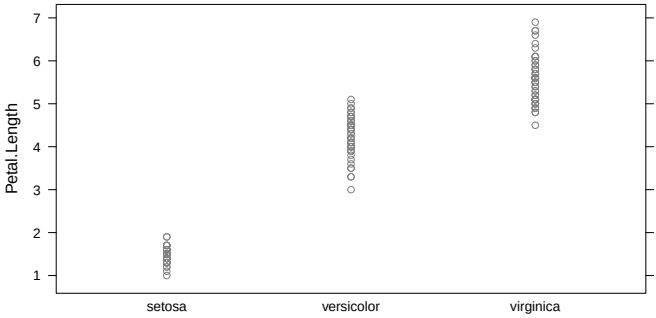
```
> bwplot(Petal.Length ~ Species, data = iris)
```



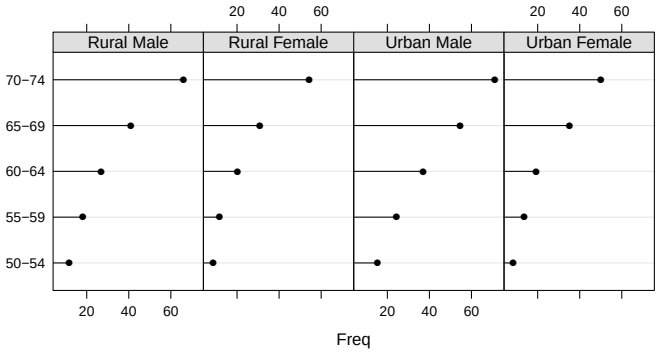

```
> bwplot(Petal.Length ~ Species,
        data = iris,
        panel=panel.violin)
```



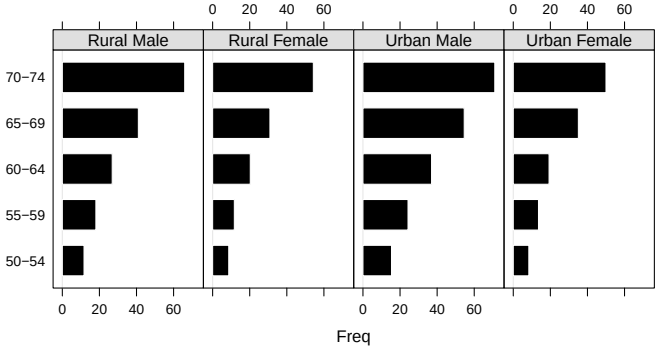

```
> stripplot(Petal.Length ~ Species,
        data = iris,
        jitter.data=F)
```



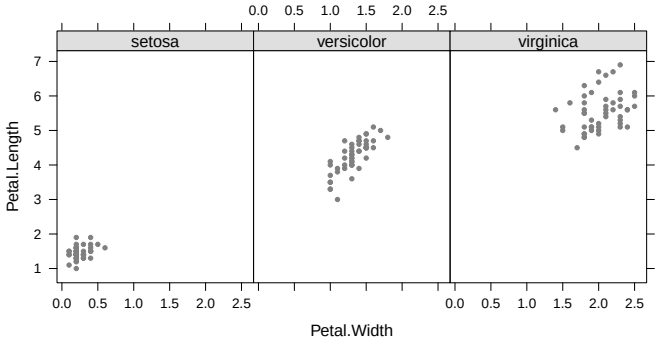
```
> dotplot(VADeaths, groups=FALSE,
          type=c("p","h"))
```



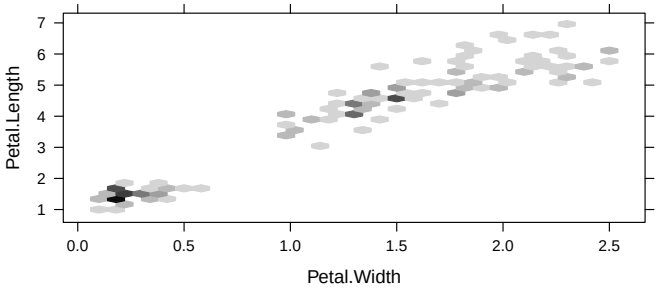

```
> barchart(VADeaths, groups=FALSE)
```




```
> xyplot(Petal.Length ~ Petal.Width | Species,
          data=iris, pch=20)
```

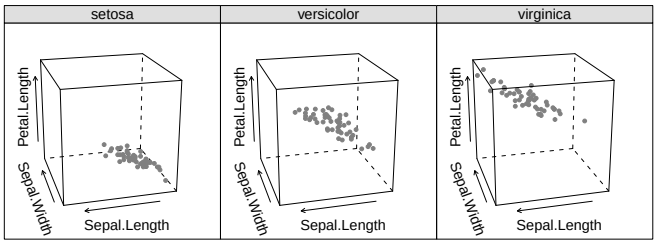


```
> library(hexbin)
> xyplot(Petal.Length ~ Petal.Width, data=iris,
         panel= panel.hexbinplot, pch=20)
```



Isometrische Darstellungen lassen sich mit der Funktion `cloud` erstellen. Die Z-Achse steht vor der Tilde und die X- und Y-Achse dahinter. Die isometrischen Darstellung lässt sich mit einigen neuen Optionen beeinflussen. Mit dem Parameter `screen` lässt sich die Beobachterposition festlegen. Im Beispiel unten wurden zusätzlich die Achsenbeschriftungen rotiert. Dazu muss die Achsenbeschriftung als Liste angegeben werden, die an erster Stelle den Beschriftungstext und den Parameter `rot` mit dem Rotationswinkel enthält.

```
> cloud(Petal.Length ~ Sepal.Width + Sepal.Length | Species,
         data=iris, pch=20,
         screen=list(z=105, x=-70, y=0),
         zlab=list("Petal.Length", rot=90),
         xlab=list("Sepal.Width", rot=-70),
         ylab=list("Sepal.Length")          )
```



Nicht nur Punktwolken auch Ebenen können isometrisch dargestellt werden. Für das folgende Beispiel werden aus den Werten von X und Y mit der Funktion $log(2 \cdot (x^2 + y^2))$ die Z-Werte einer gekrümmten Ebene berechnet.

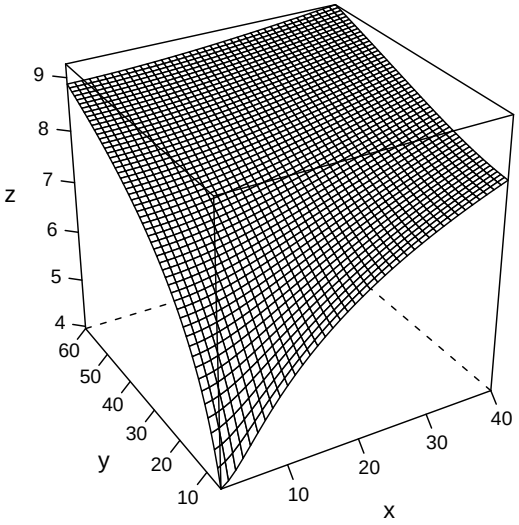
```
> g <- expand.grid(x = 1:40, y = 5:60)
> g$z <- log(2*(g$x^2 + g$y^2))
```

Um die Grafik freizustellen, können Achsen und Panel versteckt werden.

```
> par.set <- list(
            axis.line=list(col="transparent"), # Rahmen und Achsen
            clip=list(panel="off"))            # Kein Panel erzeugen
```

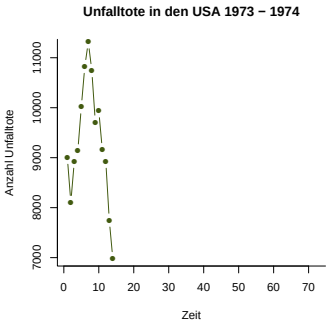
Mit dem Befehl `wireframe` lässt sich die Ebene schließlich anzeigen. Die soeben erzeugte Liste mit den Darstellungsoptionen wird über die Option `par.settings` aktiviert.

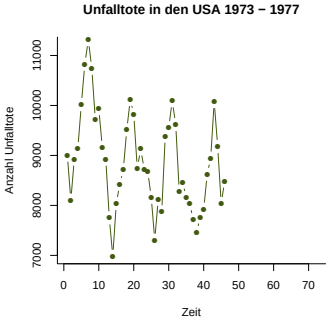
```
> wireframe(z ~ x * y, data = g,
          par.settings=par.set,              # Obige Einstellungen
          shade=F,                           # Schattierung
          scales = list(col="black",         # Farbe der Achsen
                        arrows = FALSE),     # Keine Pfeilspitzen
          drape = FALSE,                     # Höhenabhängige Färbung
          colorkey = FALSE,
          screen = list(z = 30, x = -60))
```

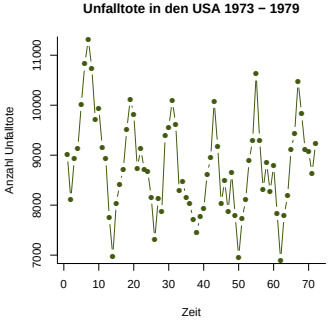


7 Animation

Animationen sind nützliche Hilfsmittel, um zusätzliche Informationen in einer Grafik darzustellen und sind insbesondere zur Visualisierung von Veränderungen über den Zeitverlauf nützlich. Animationen können einfach in Form eines Daumenkinos erstellt werden. Man beginnt mit dem ersten Plot, fügt etwas hinzu oder verschiebt einen Punkt und speichert das Bild ab. Dieser Vorgang wird solange wiederholt, bis die Grafik ihre endgültige Form erhalten hat.
Dies lässt sich in R komfortabel mit Hilfe eine Schleife erledigen. Das gewünschte Ausgabegerät wird geöffnet (hier `png`) und in der Option `file` ein Zähler angegeben (siehe S. 13). Innerhalb einer Schleife wird der `plot`-Befehl aufgerufen und mit jedem Aufruf wird eine neue Datei geschrieben.Wichtig ist, dass `dev.off` außerhalb der Schleife steht, da ansonsten nur der erste Frame gespeichert werden würde.
Die Animation selbst wird mit dem Programm `convert` aus dem Programmpaket ImageMagick erzeugt.[a] Die Nutzung eines Kommandozeilenprogramms hat den Vorteil, dass das Zusammenfügen der Einzelbilder direkt aus dem `R`-Script aufgerufen werden kann; hierzu dient der Befehl `system`, der Programmaufrufe an die Windows-Eingabeaufforderung bzw. unter MacOS X und Linux an die Shell weiterleitet.

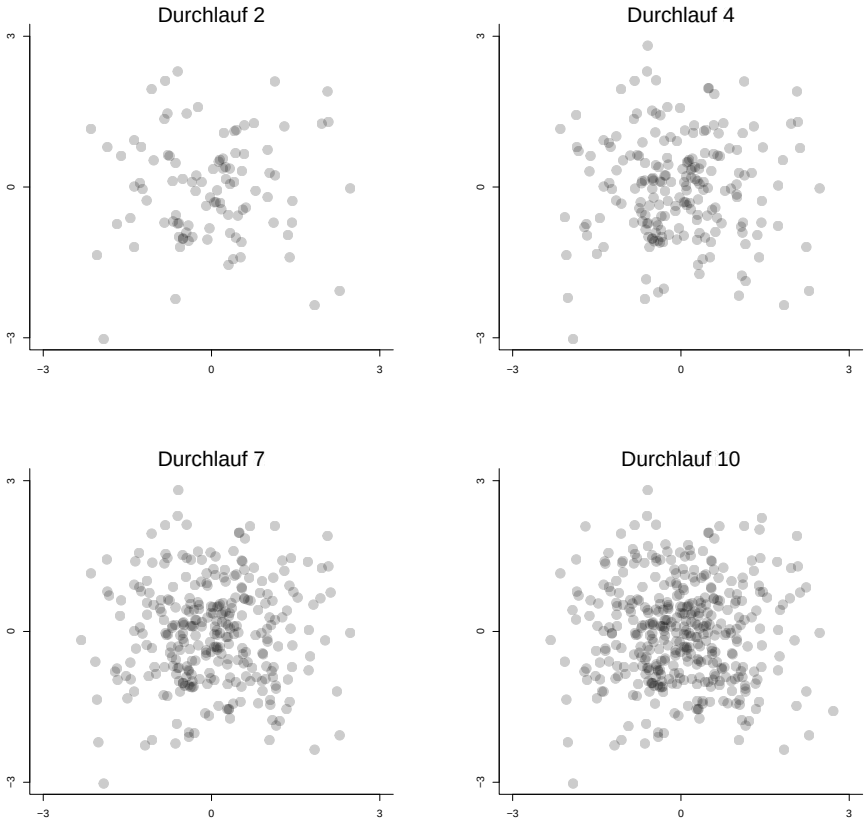






[a] ImageMagick ist freie Software und kann unter folgender URL für die meisten Betriebssysteme heruntergeladen werden: `http://www.imagemagick.org/`

```
> data(USAccDeaths)
> png(file="anihi%03d.png",       # Aufruf mit automatischem Zähler
                                  #  mit jedem plot()-Aufruf
                                  #  eine neue Datei
     width=500, height=400)       # Höhe und Breite in Pixeln

> for (i in 1:72){
     plot(USAccDeaths[1:i],       # Nur Werte 1 bis i zeichnen
          xlim=c(1,72),           # X-Achse festlegen
          ylim=c(7000,11200),     # Y-Achse festlegen
          type="b",
          pch=16,
          cex=1,
          col="#425a10",
          bty="l",
          xlab="Zeit",
          ylab="Anzahl Unfalltote",
          main=paste("Unfalltote in den USA 1973 - ",
                     round(1973+(i/12)),
                     sep="")
        )
    }

> dev.off()

> system("convert -delay 9 -loop 0 anihi*.png animated1.gif")
```



Soll ein Plot schrittweise ergänzt werden oder aus anderen Gründen nicht für jeden Frame `plot` aufgerufen werden, muss ein wenig anders vorgegangen werden. Zuerst wird mit `plot.new` und `plot.window` ein Grafikfenster geöffnet. Anschließend werden alle Elemente hinzugefügt, die in allen Frames erscheinen sollen, wie Achsen, Umrandungen, Legenden etc.

Die einzelnen Frames werden innnerhalb der Schleife mit `dev.copy` gespeichert. Die Funktion `paste` ermöglicht es, Texte aus einzelnen Fragmenten zusammenzusetzen und dient im Beispiel rechts dazu, den Dateinamen zu generieren.

```
> plot.new()
> plot.window(xlim=c(-3,3),     # Das Koordinatensystem definieren
            ylim=c(-3,3))

> axis(1, at=c(-3,0,3))         # X-Achse einzeichnen
> axis(2, at=c(-3,0,3))         # Y-Achse einzeichnen
> box(, bty="l")                # Box um den Plotbereich

> fnum  <- 0                    # Zähler für die Dateinamen

> for (i in 1:10){              # Schleife mit 10 Durchläufen
   fnum <- fnum+1               # Jeder Durchlauf erhöht den
                                #  Zähler um 1

   points(rnorm(100),rnorm(100),      # 100 zufällige Punkte
          pch=20, cex=3, lwd=0,
          col=rgb(0,0,0,0.2))         # Schwarz, 80% Transparenz

   mtext(side=3,  cex=2,              # Titel mit Zähler
         paste("Durchlauf", fnum, sep=" "))

   dev.copy(png,                          # Plot in PNG-Datei
            file=paste("anilow",          # Dateiname
                 sprintf("%03d",fnum),    # Zahlen formatieren
                                          # hier: 001, 002 usw.
                 ".png",                  # Dateinendung
                 sep=""                   # Nichts zwischen den
                                          #  Teilstücken einfügen
                 ))
    dev.off()

    # Titel mit weißer Schrift überschreiben
    for (z in 1:10){
              mtext(side=3, paste("Durchlauf", fnum, sep=" "),
                    cex=2, col="white") }
   }

> system("convert -delay 9 -loop 0 anilow*.png animated2.gif")
```

8 Karten

Die eingängiste Form der Darstellung von räumlichen Zusammenhängen sind Karten. Die Darstellung räumlicher Bezüge in Tabellen (z.B. durch Entfernungsangaben oder Angaben zur Region) oder andere Formen sind auch möglich, diese haben jedoch den Nachteil, dass die räumlichen Beziehungen nicht in gewohnter Form sichtbar sind und die räumlichen Bezüge erst vom Rezipienten gedanklich nachvollzogen werden müssen. Die Analyse und Datenhaltung von räumlichen Daten erfolgt traditioneller Weise in Geo-Informations-Systemen (GIS).

Das Paket `sp` erweitert `R` um Klassen und Methoden zur Analyse und Visualisierung von räumlichen Daten. Zum Import von Karten wird das Paket `maptools` benötigt. Das Paket `rgdal` ist notwendig, um mit Projektionen zu arbeiten.[1]

8.1 Typen von räumlichen Daten

Es können grob vier Arten von räumlichen Daten unterschieden werden (Bivand u. a. 2008: S. 8):

Punkte Die Daten enthalten einzelne Punktkoordinaten wie sie etwas GPS-Empfänger speichern können oder geokodierte Adressen, meist in Längen- und Breitengrade angegeben.

Linien Eine Liste mit Punkten, die durch eine Linie verbunden werden können. Beispielsweise zur Darstellung von Straßenverläufen oder Flüssen.

Polygone Polygone sind Linien, die eine Fläche umschließen und werden beispielsweise genutzt, um Kreis- oder Landesgrenzen darzustellen.

Raster (Grid) Raster teilen eine Region in einzelne rechteckige Zellen auf, für jede Zelle des Rasters wird ein numerischer Wert gespeichert. Höhenprofile werden oftmals in Form von Rastern gespeichert.

[1] Da die Erdoberfläche geometrisch die Oberfläche einer Kugel ist, kann sie nicht verzerrungsfrei auf einer Ebene abgebildet werden. Es wurden daher verschiedene Projektionsverfahren entwickelt, mit denen die Oberfläche auf einer zweidimensioanlen Fläche dargestellt werden kann. Eine grafische Übersicht verschiedener Projektionsverfahren liefert die deutsche Wikipedia: `http://de.wikipedia.org/wiki/Kartennetzentwurf`

8.2 Kartenmaterial

Mittlerweile finden sich zahlreiche Quellen für digital aufbereitetes Kartenmaterial, welches für die nicht-kommerzielle Nutzung kostenlos ist:

- OpenStreetmap.org: `http://download.geofabrik.de/osm/`
- OpenGeoDB pflegt eine Datenbank, die Postleitzahlenbereiche mit Geokoordinaten verbindet: `http://opengeodb.giswiki.org/wiki/OpenGeoDB`
- GADM Database of Global Administrative Areas bietet Karten mit administrativen Grenzen der meisten Länder weltweit: `http://www.gadm.org/country`
- DIVAS-GIS stellt zahlreiche Geo-Daten zum Donwload bereit: `http://www.diva-gis.org/Data`
- NASA Blue Marble liefert Sattelitenbilder der Erde: `http://earthobservatory.nasa.gov/Features/BlueMarble/`
- Digital Chart of the World Server: `http://www.maproom.psu.edu/dcw/`
- U.S. Census Bureau (US-Karten): `http://www.census.gov/geo/www/tiger/tgrshp2009/tgrshp2009.html`
- Das R-Paket mapdata enthält hochauflösende Weltkarten mit nationalen Grenzen (CIA World Data Bank II), Flüsse weltweit sowie detaillierte Karten von China und Neuseeland

8.3 Datenimport

Vielfach liegen die Karten im Shapefile-Format vor. Eine Karte umfasst normalerweise mindestens drei Dateien: eine mit der Dateiendung `.shp`, die Punkte, Linien oder die Karte als Polygon enthält, eine mit der Dateiendung `.dbf`, die Sachdaten im dBASE-Format enthält und drittens eine `.shx`-Datei, die die Sachdaten mit den Geometriedaten verknüpft (*ESRI Shapefile Technical Description* 1998).

Der Import von Shapefiles wird mit den Funktionen a) `readShapeLines` b) `readShapePoints` c) `readShapePoly` und d) `readShapeSpatial` durchgeführt. Die beiden wichtigsten Argumente sind der Dateiname und der `proj4string`, der die Art der Kartenprojektion beschreibt. Es reicht dabei aus, den Dateinamen des Shapefiles anzugeben, da die dBASE-Datenbank usw. automatisch eingelesen werden.

Für die folgenden Beispiele werden die von GADM bereitgestellten Shapefiles für Deutschland verwendet: `http://www.diva-gis.org/data/adm/DEU_adm.zip`.

```
library(maptools)
library(rgdal)
de <- readShapePoly("DEU_adm1.shp", proj4string=CRS("+proj=longlat
    +datum=NAD27"))
```

Das erzeugte Objekt gehört zur S4-Klasse; dies ist nur in der Weise bedeutsam, dass die einzenlen Bestandteile des Objekt in „Slots“ gespeichert sind, die nicht wie in einer Liste mit $ angesprochen werden sondern mit @.

```
  de@data      # Datentabelle des Shapefiles
  de@data$NAME_1 # Namen der Bundesländer
```

Weitere Daten können problemlos ergänzt werden. Wie immer ist es wichtig, dass die Zeilensortierung übereinstimmt, d.h. der erste Wert ist auf Baden-Württemberg bezogen, der zweite auf Bayern etc.

```
> einwd <- c(301,177,3834,  # Einwohnderdichte der Bundesländer
             86,1640,2344,
             288,72,167,
             528,204,404,
             229,118,180,142)
> mp <- c(1,1,2,  # Parteizugehörigkeit Ministerpräsident
          2,2,1,
          1,2,1,
          1,2,1,
          1,1,1,1)
> mp <- factor(mp, labels=c("CDU/CSU", "SPD"))
> de@data$einwd <- einwd    # In den Kartensatz schreiben
> de@data$mp <- mp
```

Diese Daten können im Weiteren dazu genutzt werden, die Bundesländer entsprechend ihrer Einwohnerdichte einzufärben. Dazu muss zuerst die Farbpalette bestimmt werden. Problematisch ist die schiefe Verteilung der Werte, so dass bei einer Einteilung in gleichgroße Intervalle die Unterschiede zwischen den Flächenländern kaum erkennbar wären. Daher werden im Folgenden die Dezilgrenzen bestimmt und dann später jedem Dezil ein Grauwert zugeordnet:

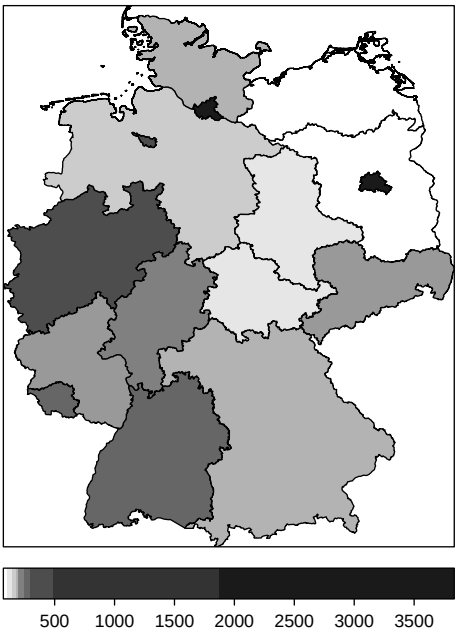
```
> einwd.breaks <- quantile(einwd,     # Dezile bestimmen
                           seq(0,1,    # Dezilgrenzen
                           length.out=10))
```

Da im Plot die Intervallgrenzen in der Form $[a, b[$ angegeben werden müssen, muss die letzte Intervallgrenze um 1 erhöht werden, damit der Maximalwert auch in die letzte Gruppe fällt.

```
> einwd.breaks[10] <- einwd.breaks[10]+1
```

Dieser Vektor kann nun einfach an die `spplot()`-Funktion übergeben werden:

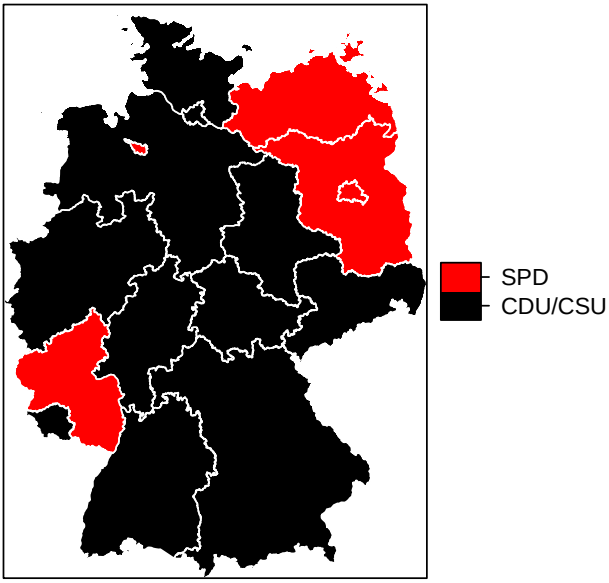
```
> library(sp)
> spplot(de,                                    # Kartenobjekt
        "einwd",                                # Variable
        col.regions = grey(10:1/10),            # Farbpalette
        at=einwd.breaks,
        aspect=1.2,                             # Darstellung
                                                #  strecken > 1
                                                #  oder stauchen <1
        colorkey=list(space="bottom"))          # Farbschlüsselposition
```



Auf die gleiche Weise können auch kategoriale Werte dargestellt werden:

```
> spplot(de,                        # Kartenobjekt
        "mp",                       # Variable
        col="white",                # Farbe Begrenzungslinien
```

```
        col.regions=c("black",    # Farben Kategorien
                      "red"))
```



Karten können auch mit den Basis-Funktionen geplottet werden:

```
> plot(de, col=c("black","red")[unclass(mp)],
      border="white")
```

8.4 Datenexport

Zum Export von Shapefiles gibt es spezielle Funktionen für die verschiedenen Datentypen:

- Linien: `writeLinesShape`
- Punkte: `writePointsShape`
- Polygone: `writePolyShape`
- Allgemein: `writeSpatialShape`
- Rasterkarten: `writeGDAL`

Die Deutschlandkarte mit den eingegebenen Daten lässt sich mit folgendem Aufruf speichern:

```
> writePolyShape(de,
                 "bula",              # Dateiname (ohne .shp)
                 factor2char = TRUE,  # factor als Text oder Nummern
                 max_nchar=254)       # maximale Länge der Textfelder
```

8.5 Weitere Beispiele

Weitere Beispiele sind auf der Homepage des R-Spatial-Projektes zu finden: http://r-spatial.sourceforge.net/gallery/.

Ein interessantes Beispiel zur Darstellung der Arbeitslosenraten auf Kreisebene zeigt Mark Heckmann in seinem Blog: http://ryouready.wordpress.com/2009/11/16/infomaps-using-r-visualizing-german-unemployment-rates-by-color-on-a-map/

9 Graphen und Netzwerke

Die Analyse von von sozialen Netzwerken mit den Mitteln der Graphentheorie hat weite Verbreitung gefunden. R bietet daher ebenfalls Funktionen zur Analyse und Darstellung von Graphen an. Eine Übersicht mit Beispielgraphen findet sich auf den Seiten des dem R-Pakets igraph zugrundeligenden Projekts: http://igraph.sourceforge.net/screenshots2.html.

9.1 Datenaufbereitung

Es gibt zwei grundlegende Arten Graphen zu repräsentieren: durch Kantenlisten und Adjazenmatrizen. Beide Repräsentationsformen können in R genutzt werden.

Kantenliste

In einer Kantenliste wird ein Graph über seine Kanten beschrieben. Jede Kante wird durch den Ausgangsknoten, den Typ der Kante und dem Eingangsknoten definiert.

Die Funktion graph.edgelist() erzeugt einen Graphen aus einer zweispaltigen Matrix. Jede Zeile der Matrix entspricht einer Kante, die vom in der ersten Spalte angegeben Knoten zum Knoten in der zweiten Spalte verläuft. Intern werden Knoten (englisch: vertices) durch ID-Nummern bezeichnet. Werden in der Matrix jedoch symbolische Namen wie im Beispiel unten genutzt, erzeugt die Funktion automatisch ein Namensattribut, das beim Plotten automatisch genutzt wird.

```
elist <- rbind(c("A","C"),
               c("A","E"),
               c("B","A"),
               c("C","B"),
               c("C","D"),
               c("D","D"),
               c("D","E"),
               c("E","B"))
```

```
g <- graph.edgelist(elist,          # Matrix
                    directed=TRUE)  # Ist der Graph gerichtet?
```

Adjazensmatrix

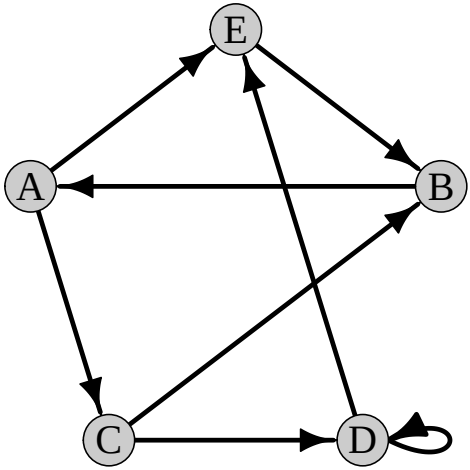
In einer Adjazensmatrix $A[k, l]$ stehen die Knoten des Graphs in Spalten und Zeilen. Eine Kante vom Knoten i zum Knoten j wird durch einen Wert > 0 für $a[i, j]$ definiert. Bei unbewerteten Kanten werden meist die Werte `0` für keine Kante und `1` für bestehende Kanten verwendet. Bei bewerteten Graphen wird oftmals `0` bei fehlender Kante und der Wert des Gewichts bei bestehender Kante verwendet.

A	B	C	D	E
0	0	1	0	1
1	0	0	0	0
0	1	0	1	0
0	0	0	1	1
0	1	0	0	0

Adjazensmatrizen können mit der Funktion `graph.adjacency()` in ein `igraph` -Objekt umgewandelt werden:

```
g <- graph.adjacency(adjmatrix,
          mode=c("directed"), # oder "undirected"
          weighted=NULL,      # NULL: ungewichtet
                              # TRUE: Matrix-Werte als Gewichte
          diag=TRUE,          # Diagonale der Matrix auswerten?
          add.colnames=NULL,  # NULL: Spaltenüberschriften als
                              #  Knotenbezeichnung
                              # NA: keine Namen
          add.rownames=NA)    # siehe add.colnames
```

9.2 Graphen



Der oben abgebildete Graph lässt sich aus folgender Adjazensmatrix erzeugen:

```
> adjm <- matrix(c(0,1,0,0,0,
                   0,0,1,0,1,
                   1,0,0,0,0,
                   0,0,1,1,0,
                   1,0,0,1,0),
                   ncol=5)
> g <- (graph.adjacency(adjm))
```

Mit Hilfe der `plot`-Funktion wird der Graph gezeichnet. Eine Übersicht über sämtliche Darstellungsoptionen lässt sich über `help(igraph.plotting)` abrufen. Die Darstellungsoptionen können mit der Funktion `igraph.par()` für die gesamte Sitzung festgelegt werden.

```
> plot(g,
      layout=layout,               # Graphenlayout (siehe unten)
      edge.width=3,                # Linienstärke Kanten
      edge.color="black",          # Farbe der Kanten
      edge.arrow.size=1.2,         # Größe der Pfeilspitzen
      vertex.size=25,              # Größe der Knoten
      vertex.color=gray(0.8),      # Farbe der Knoten
      vertex.label.color="black",  # Farbe des Beschriftung
      vertex.label.cex=1.6,        # Schriftgröße Beschriftung
```

```
      vertex.label=c("A",          # Label für die Knoten in
                     "B",          #  Reihenfolge der Vertex IDs
                     "C",
                     "D",
                     "E"))
```

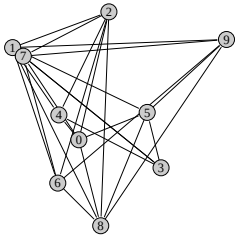
Die Funktionen `tkplot` und `rglplot` erweitern die Möglichkeiten der visuellen Darstellung von Graphen: `tkplot` öffnet ein interaktives Grafikfenster, in dem die Position der einzelnen Knoten manuell angepasst werden kann oder auch direkt verschiedene Layout-Varianten ausprobiert werden können. `rglplot` ermöglicht das Betrachten dreidimensionaler Darstellungen, hier sollte ein Layout gewählt werden, das die dritte Dimension unterstützt, wie etwa `layout.sphere`.

9.2.1 Layouts

Graphen können verschiedene Layouts aufweisen, die in unterschiedlicher Weise Anzahl, Richtung und Gewichte der Kanten berücksichtigen, um entweder eine ausgewogene Darstellung zu erzielen oder Informationen über die Beziehung der Knoten zueinander darzustellen.

```
> set.seed(46)
> g <- random.graph.game(10, 27, "gnm")     # Erzeugt einen zu-
>                                           #  fälligen Graphen
> igraph.par("edge.color","black")          # Farbe der Kanten für die
>                                           #  Sitzung festlegen
> igraph.par("vertex.color",gray(0.8))      # Hintergrundfarbe Knoten
> igraph.par("vertex.label.color","black") # Schriftfarbe Knoten
```

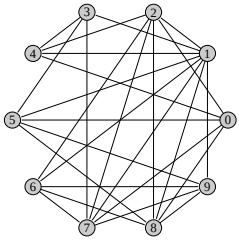
Standardmäßig wird für die Darstellung eine zufällige Anordnung der Knoten gewählt:



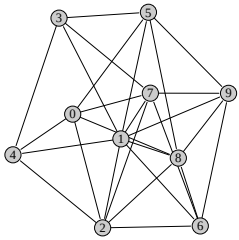

```
plot(g, layout=layout.random(g))
```

Die verschiedenen Layout-Varianten werden beim Aufruf von `plot()` angegeben und enthalten als Argument den zu zeichnenden Graphen. Eine Übersicht über Kon-

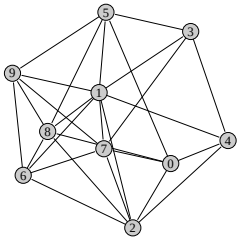
struktionsprinzipien solcher Layoutfunktionen liefert Krempel (2005: Kapitel 4).



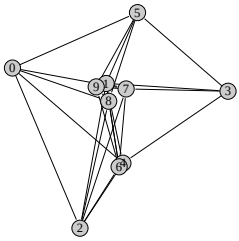
`layout.circle(g)`



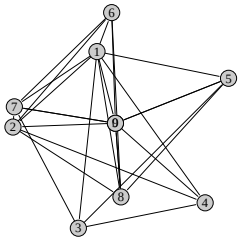
`fruchterman.reingold(g)`



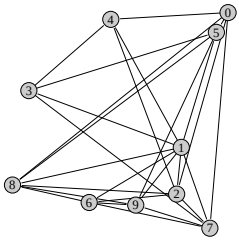
`layout.mds(g)`



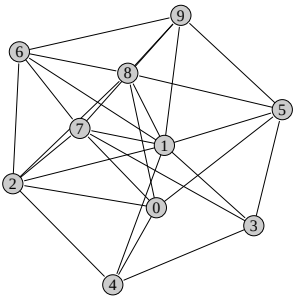
`layout.spring(g)`



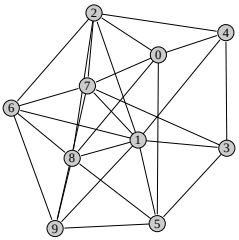
`layout.sphere(g)`



`layout.random(g)`



`layout.kamada.kawai(g)`



`layout.graphopt(g)`

9.2.2 Import und Export von Graphen

Graphen können aus anderen Anwendungen über den Befehl

```
g <- read.graph(file="sweavetmp/test.graphml", format="graphml")
```

importiert werden.

Auf ähnliche Weise lassen sich die `igraph`-Objekte zur weiteren Auswertung in andere Programme exportieren:

```
> E(g)$weigth <- rep(c(1,2,3),9) # Beispielgewichte
> V(g)$eigenschaft <- 1:10       # Beispiel für Knotenattribute
> write.graph(g,
              file="sweavetmp/test.graphml",
              format="graphml")
```

Weitere Ein- und Ausgabeformate sind: `edgelist`, `pajek`, `ncol`, `lgl`, `graphml`, `dimacs`, `gml` und `dot`. Informationen zu den einzelnen Dateiformaten finden sich in der Hilfe zu `write.graph`.

Literatur

Bivand, Roger S. u. a. (2008): *Applied spatial data analysis with R*. Use R! New York, NY: Springer. XIV, 374. isbn: 9780387781709.

Cleveland, William S. und Robert McGill (1984): „Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods“. In: *Journal of the American Statistical Association* 79.387, S. 531–554. issn: 01621459. url: http://www.jstor.org/stable/2288400.

Crawley, Michael J. (2009): *The R book*. Reprint. Chichester u.a.: Wiley. VIII, 942. isbn: 9780470510247.

Dalgaard, Peter (2008): *Introductory statistics with R*. 10. Statistics and computing. New York u.a.: Springer. XV, 267. isbn: 0387954759.

Dudel, Christian und Sebastian Jeworutzki (2010): *Einführung in R*. url: http://www.stat.rub.de.

ESRI Shapefile Technical Description (Juli 1998). Environmental Systems Research Institute, Inc. (ESRI). url: http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf.

Krempel, Lothar (2005): *Visualisierung komplexer Strukturen. Grundlagen der Darstellung mehrdimensionaler Netzwerke*. Frankfurt: Campus. isbn: 3593378132.

Ligges, Uwe (2007): *Programmieren mit R*. 2., überarb. und aktualisierte Aufl. Statistik und ihre Anwendungen. Berlin u.a.: Springer. XII, 247. isbn: 9783540363323. url: http://dx.doi.org/10.1007/978-3-540-36334-7.

Murrel, Paul (2009): *R Graphics Devices*. url: http://www.stat.auckland.ac.nz/~paul/R/devices.html.

Venables, W. N. u. a. (2009): „An Introduction to R“. Version 2.10.0, abrufbar unter www.r-project.org.