technische universität dortmund

Fakultät Statistik

Programmierung mit
Teil II

Uwe Ligges

TU Dortmund, Sommersemester 2020

1.0 Organisation

	Datum / Zeit	Raum
Vorlesung	Freitags 12:15 Uhr	ONLINE, später M / E28
Übungen	wird bekanntgegeben (Beginn ab 24.04.2020)	ONLINE, später CDI / 121
Hauptklausur	20.07.2020 / 15:00 Uhr	HG II / HS 6
Nachklausur	29.09.2020 / 12:00 Uhr	HG II / HS 5

- Webseite der Vorlesung: `https://moodle.tu-dortmund.de/course/view.php?id=20625`
- Einteilung der Übungsgruppen nach Anmeldung unter o.g. URL
- Anmeldefrist: 26.04.2020
- Zur Vorlesungszeit/Übungszeit stehen Dozent/Übungsgruppenleiter elektronisch zur Verfügung.
- Zum Vorlesungskript wird es auch Ton geben.

1.0 Organisation

- Abgabe von Übungen elektronisch über Moodle, Rückgabe analog.
- Für alle, die etwas Lernen und einen Leistungsnachweis möchten:
  - **Selbständiges Bearbeiten der Übungszettel!**
  - **Klausur am Ende der Semesters.**
    Hilfsmittel zur Klausur: Bücher, Skripte etc. in gehefteter Form, jedoch keine elektronischen Hilfsmittel.
- Aufgrund von online Lehre und gekürztem Semester wird evtl. entsprechend an den letzten Kapiteln gekürzt und die Reihenfolge einiger Kapitel noch umgestellt.

1.0 Was passiert heute, am 24.04.2020, besonderes?

R

R-4.0.0 erscheint!

1.1 Inhalt

Kapitel

1 Einführung und Wiederholung
1.1 Inhalt

- Einführung und Wiederholung
- Funktionen 2 (Wiederholung, Scoping Rules, Workspace, Debugging)
- effiziente Programmierung (hinsichtlich Speicherplatz, Rechenzeit)
- Verteilungen und Stichproben
- Algorithmik und Simulation
- R Pakete verwenden
- Statistische Verfahren und das Formel-Interface
- Objektorientiertes Programmieren (S3, S4)
- Parallele Programme schreiben
- Erweiterbarkeit (Erstellen von Paketen, externer (C) Code, Namespaces)
- Literatur

1.2 Wiederholung

Kapitel

1 Einführung und Wiederholung
1.2 Wiederholung

Es wird davon ausgegangen, dass folgendes bekannt ist:

- Einführung (Hintergrund, Geschichte und Ausblick, Literatur)
  - „R als Taschenrechner“
  - Verwendung des Hilfesystems
  - Zuweisungen
  - Logische Werte und Operatoren

1.2 Wiederholung

- Datenstrukturen
  - Datentypen (logical, numeric, character, etc.)
  - Datenstrukturen (Vektoren, Matrizen, Listen, data.frame, etc.)
  - Verwendung des Hilfesystems
  - Indizierung bei verschiedenen Datenstrukturen
- Import und Export (Daten, Programmcode, ...)
  - Textdateien
  - Datenbanken
  - SPSS-, Excel-Daten
  - Umgang mit Datum und Zeit

1.2 Wiederholung

- Grafik
  - Devices
  - Highlevel Grafikfunktionen
  - Lowlevel Grafikfunktionen
- „Rechnen“ auf Zeichenketten
- Konstrukte, elementare Funktionen
  - Fallunterscheidungen (bedingte Anweisungen)
  - Schleifen
  - ...apply()-Funktionen
- Literatur

2.1 Funktionen

Kapitel

2 Funktionen
2.1 Einführung

Bisher sind uns schon einige Funktionen begegnet, darunter einfache wie `sin()`, aber auch komplexere wie `read.table()`, die verschiedene Argumente akzeptieren.

- Jegliches Arbeiten geschieht mit Funktionen!

2.1 Funktionen

- Ein Funktionsaufruf hat die Form
  `funktionsname(Argument1 = Wert1, Argument2 = Wert2, usw.)`,
  dabei kann die Benennung der Argumente u.U. weggelassen werden.
  - Es gibt spezielle Funktionen mit Kurzformen, z.B. +.
    Der Ausdruck `3 + 5` würde in voller Form lauten: `"+"(3, 5)`.
    Der Name steht hier in Anführungszeichen, da es kein regulärer Name ist (solche beginnen mit Buchstaben!).
  - Auch die Zuweisung ist eine Funktion: `"<-"(x, 3)`.
- Es gibt Argumente, die Voreinstellungen (defaults) haben.
  - Ein Argument ohne default *muss* beim Funktionsaufruf angegeben werden.
  - Ein Argument mit Voreinstellung *kann* beim Funktionsaufruf geändert werden.
  - Funktionsaufrufe müssen nicht immer Argumente besitzen, wie z.B. `ls()`.

2.1 Funktionen

Eigene Funktionen sind immer dann sinnvoll, wenn eine Folge von anderen Funktionsaufrufen (unter einem Namen) zusammengefasst werden soll, z.B. für mehrmaliges Ausführen mit verschiedenen Parametern.

Die allgemeine Form einer Funktionsdefinition ist:
`Meine.Funktion <- function(Argumente){ Befehlsfolge }`,
wobei die `Argumente` mit oder ohne Voreinstellung angegeben werden. Beim Aufruf der Funktion werden die `Argumente` an die `Befehlsfolge` weitergereicht.

Nicht nur bei der Definition von Funktionen, sondern auch bei allen anderen Konstruktionen (`for()`, `if()`), können Befehlsfolgen, solange sie in geschweiften Klammern stehen, aus mehreren Zeilen bestehen.

2.1 Funktionen

Eine typische Definition einer Funktion könnte wie folgt aussehen:

```
median <- function(x, na.rm = FALSE)
{
     # ... viel code! ...
    sort(x, partial = half)[half]
}
```

- Diese Funktion hat zwei Argumente: `x, na.rm`.
- Nur das zweite hat einen default, nämlich `FALSE`.
- Die letzte Zeile der Funktion gibt den Wert der Funktion an. Sollen mehrere Werte (als Liste) zurückgegeben werden, benutzt man `return()`.

2.1 Funktionen

- Sei `a` ein Vektor, dann sind z.B. folgende Aufrufe sinnvoll:
  - `median(a)` (`na.rm` muss nicht angegeben werden – default)
  - `median(a, TRUE)`
    (In richtiger Reihenfolge müssen Argumente nicht benannt werden.)
  - `median(na.rm = TRUE, x = a)`
    (Benannte Argumente dürfen in beliebiger Reihenfolge stehen.)

2.1 Funktionen

Man muss also zwischen den *formal* in der Funktion definierten Argumenten und den *tatsächlich* beim Funktionsaufruf gegebenen Argumenten unterscheiden. Die Regeln dazu werden in der folgenden Reihenfolge angewandt (Beispiel `median()`):

- Alle Argumente mit vollständigem Namen werden zugeordnet (`x = 1:10`).
- Argumente mit teilweise passendem Namen werden den übrigen *formalen* Argumenten zugeordnet (`na = TRUE`).
- Alle unbenannten Argumente werden der Reihe nach den übrigen *formalen* Argumenten zugeordnet.
- Übrige Argumente werden dem evtl. vorhandenen formalen Argument `...` zugeordnet (s.u.).

Auf das Fehlen eines formalen Arguments kann man innerhalb einer Funktion mit `missing()` testen.

2.1 Funktionen

Beispiele:

```
a <- c(1, NA, 5, 3)

## zunächst unbenannte Argumente:
median(a)
median(a, TRUE)
median(TRUE, a)               # das macht keinen Sinn!

## nun zumindest teilweise benannte Argumente:
median(x = a, na.rm = TRUE)
median(na.rm = TRUE, a)
median(n = TRUE, a)
```


2.1 Funktionen

In der Definition einer Funktion ist der Einsatz des formalen „Drei–Punkte Arguments“ `...` möglich. Alle nicht zugeordneten *tatsächlich* gegebenen Argumente werden durch `...` aufgenommen und können innerhalb der Funktion weiterverarbeitet oder an andere Funktionen durch den Aufruf mittels `...` weitergegeben werden. Es dient also meist zum Durchreichen von Argumenten.

Beispiele:

```
Punkte <- function(x, ...){
    x <- x - 2
    median(x, ...)
}
x <- log(-1:100)
Punkte(x)
Punkte(x, na.rm = TRUE)
```

2.1 Faule Funktionen

In manchen Situationen fällt auf, dass Argumente in Funktionsaufrufen der *lazy evaluation* unterliegen, d.h. Ausdrücke in Argumenten werden erst ausgewertet, wenn das Argument in der Funktion zum ersten Mal benutzt wird.

Beispiele:

```
lazy <- function(x, rechnen = TRUE) {
    if(rechnen) x <- x+1
    print(a)
}
lazy((a <- 3), rechnen = FALSE)
lazy(a <- 3)
label <- function(x)
    return(list(Aufruf = substitute(x), Wert = x))
label(1+2)
```


2.2 Scoping Rules

Kapitel

2 Funktionen
2.2 Scoping Rules

Eine wichtige Frage im Zusammenhang mit Funktionen ist, wie in vielen Programmiersprachen, wann welche Objekte existieren bzw. sichtbar sind.

Wenn man direkt in der Kommandozeile arbeitet, werden standardmäßig alle neu erzeugten Objekte im Workspace abgelegt.

2.2 Scoping Rules

In (komplexeren) Funktionen werden jedoch sehr viele Objekte erzeugt, die nur vorübergehend benötigt werden. Daher macht es Sinn, Funktionen in einer eigenen Umgebung auszuführen, so dass nicht alle (zum größten Teil überflüssige) Objekte im Workspace landen und (a) zur Unübersichtlichkeit und (b) zur Speicherverschwendung beitragen.

Zuweisungen innerhalb einer Funktion werden also nicht im Workspace gespeichert. Ebenso sollten Objekte aus dem Workspace, die innerhalb einer Funktion benötigt werden, der Funktion aus (nicht nur) ästhetischen Gründen als Argumente übergeben werden.

2.2 Scoping Rules

- Es gibt *environments*, die alle im Hauptspeicher stehen.
- Standardmäßig werden Objekte, die im *top–level* erzeugt werden, im Workspace („.GlobalEnv“) gespeichert, dem die Nummer 0 zugeordnet ist.
- Es gibt einen „Suchpfad“, in den weitere Umgebungen (in erster Linie packages, um weitere Funktionen nutzen zu können; und data.frames, um direkt auf deren Elementen arbeiten zu können) hinzugefügt werden können. Vor allem sind darin „.GlobalEnv“ (der Workspace selbst als erstes), das base package (zuletzt) und eingehängte Objekte (mit `library()` oder `attach()` werden Objekte an Stelle 2 eingehängt) dazwischen.
- Es werden beim Aufruf von Funktionen neue *environments* (angefangen von Nummer 1) kreiert.
- Falls in einer verschachtelten Funktion auf ein Objekt zugegriffen wird, so werden alle darunterliegenden Environments durchsucht.

2.2 Scoping Rules

Das bedeutet: Ein in Funktion 2 erzeugtes x wird verwendet, auch wenn es im Workspace ein Objekt x gibt. `search()` liefert die Liste:

```
-8 package:base
-7 Autoloads
... ... ... ... ...
-2 package:stats
-1 package:methods
 0 .GlobalEnv       # Workspace
 1 environment 1    # Funktion 1
 2 environment 2    # Funktion 2
 3 environment 3    # Funktion 3
```

Es gibt die Funktionen `assign()` und `get()` für Zuweisungen und Abfragen, mit denen man direkt *environments* ansprechen kann.

2.2 Scoping Rules

R beherrscht sogenanntes *Lexical Scoping* (Gentleman, R. und Ihaka, R., 2000).

Das bedeutet u.a., dass Funktionen, die in einer bestimmten *environment* erzeugt wurden, und dann z.B. einer anderen zugewiesen wurden, weiterhin die Objekte der ursprünglichen *environment* kennen. Eine *environment* wird also nur dann wirklich gelöscht, wenn die zugehörige beendete Funktion keine Funktion zurückgegeben hat, die einer anderen *environment* zugewiesen wurde.

2.2 Scoping Rules

Im Falle von Verwirrung empfiehlt sich ein Blick in Venables, W.N. und Ripley, B.D. (2000).

Ausnahmen zu den hier beschriebenen Regeln werden durch s.g. Namespaces geschaffen, die am Ende des Kurses beschrieben werden.

2.2 Scoping Rules

Beispiele:

```
scope <- function()
{
    x <- 3
    innen <- function()
        print(x)
    innen()
}
scope()     # 3
x <- 5
scope()     # 3
```

2.2 Scoping Rules

Beispiele:

```
l.scope <- function()
{
    nur.hier <- 2
    neu <- function()
        print(nur.hier)
    return(neu)
}
Ausgabe <- l.scope()

Ausgabe()   # 2
nur.hier <- 4
Ausgabe()   # 2
```


2.3 Rekursion

Kapitel

2 Funktionen
2.3 Rekursion

Eine in einigen höheren Programmiersprachen übliche Konstruktion ist die Rekursion.

- Sinn der Rekursion: Eine Funktion kann sich selbst aufrufen.
- Beispiel: Fibonacci–Zahlen

2.3 Rekursion

Wegen der Scoping–Rules erzeugt jeder Funktionsaufruf eine neue, zusätzliche Umgebung (*environment*), denn die Funktionsaufrufe werden erst am Ende der Rekursion geschlossen.

Resultat:

- Hoher Speicherverbrauch — viele Variablen werden mehrfach im Speicher abgelegt.
- Langsam — Erzeugung von *environments* und Speicherallokation.

Man sollte also nur Probleme rekursiv programmieren, die eine überschaubare Rekursionstiefe haben. Wenn möglich, ist iteratives Programmieren angebrachter, gerade bei Funktionen, die häufiger verwendet werden sollen.

2.3 Rekursion

Als **Beispiel** folgt eine rekursive Version der Fakultät.

```
factorial <- function(n){
    if(n > 0)
        return(n * factorial(n - 1))
    else
        return(1)
}
factorial(10)

prod(1:10)  # BESSER!
gamma(11)   # NOCH BESSER!
```

2.4 Batch Betrieb

Kapitel

2 Funktionen
2.4 Batch Betrieb

Gerade bei lange dauernden Rechnungen, wie etwa tage- oder wochenlangen Simulationen, auf Unix Maschinen ist es wünschenswert, nicht innerhalb einer Sitzung zu bleiben, sondern das Programm im s.g. Batch Betrieb laufen lassen zu können (z.B. um sich abmelden zu können).

```
R CMD BATCH Programmdatei.R
```

Es empfiehlt sich hier, die Hilfe genau zu studieren, wie welches Programm unter welchem Betriebssystem die Ausgabe gestaltet. Üblich ist die Ausgabe in Dateien oder auf die Konsole.

2.4 Batch Betrieb

Unter Unix empfiehlt es sich im Fall langer Simulationen, **R** wie folgt zu starten:

```
nohup R CMD BATCH Code.R &
```

Damit wird **R** so im Hintergrund gestartet, dass man sich abmelden kann, ohne dass der Prozess beendet wird.

2.5 Debugging

Kapitel

2 Funktionen
2.5 Debugging

- Wer eigene Funktionen schreibt, macht Fehler!
- Bei sehr kurzen Funktionen findet man den / die Fehler meist schnell.
- In komplizierten Funktionen kann Fehlersuche auch für Experten zum Nervenzusammenbruch führen!
- **R** bietet einige Werkzeuge, um die Fehlersuche einfacher zu machen.

2.5 Debugging

Als nächstes wird gezeigt, wie und mit welchen Mitteln man nach Fehlern sucht.

- In allen Programmiersprachen ist die Ausgabe kurzer Texte üblich, die anzeigen, welche Stellen die Funktion bereits fehlerfrei passiert hat, so dass der Fehler eingegrenzt werden kann.
- Auch die Ausgabe von Objekten, bei denen man den Fehlerverursacher vermutet, ist nützlich.

2.5 Ausgabe auf die Konsole

- Zur Ausgabe von Informationen auf die Konsole oder in Text–Dateien eignet sich die Funktion `cat()`.
  Mit `cat()` können für die Ausgabe verschiedene Textelemente und Zahlen, auch als Variablen, kombiniert werden, insbesondere sind die Sonderzeichen `\n` (neue Zeile) und `\t` (Tabulator) nützlich.
- Für die Ausgabe von Objekten in Text–Form auf die Konsole: `print()`.
- Wenn Objekte aus einer Funktion zurückgegeben werden sollen, die wieder anderen Objekten zugewiesen werden können, so ist unbedingt `return()` zu verwenden!

Beispiele:

```
cat("Ein", 0, 8, 15, "\tHallo Welt Beispiel\nin zwei Zeilen!\n")
# Ein 0 8 15      Hallo Welt Beispiel
# in zwei Zeilen!
```


2.5 Debugging mit Werkzeugen

Da die Ausgabe von Informationen auf die Konsole nicht immer ausreicht, gibt es einige Werkzeuge, die die Fehlersuche vereinfachen:

- `traceback()` zeigt an, welche Funktion den letzten Fehler verursacht hat, und den „Pfad“ der Funktionsaufrufe bis dorthin. So kann in verschachtelten Strukturen der Schuldige gefunden werden.
- `debug(foo)`: Ab sofort wird die Funktion `foo` immer im *Browser* (s.u.) ausgeführt (bis zu einem `undebug(foo)`).
- `browser()` startet an dieser Stelle in einer Funktion den *Browser*.
- `recover()` und `options(error = recover)`: Im Falle eines Fehlers wird der *Browser* so gestartet, dass man die zu „browsende“ Umgebung wählen kann.

2.5 Debugging mit Werkzeugen

Beispiele:

```
foo1 <- function(x){  |foo1 <- function(x){   |foo1 <- function(x){
 foo2 <- function(x,s)| foo2 <- function(x,s){| foo2 <- function(x,s){
   x[[s]] + 5         |   browser()           |   print(x)
 y <- x + 1           |   x[[s]] + 5          |   x[[s]] + 5
 foo2(y, s = -5)      | }                     | }
}                     | y <- x + 1            | y <- x + 1
                      | foo2(y, s = -5)       | foo2(y, s = -5)
                      |}                      |}
                      |                       |
foo1(1:5)             |foo1(1:5)              |foo1(1:5)
traceback()           |                       |options(error = recover)
                      |                       |foo1(1:5)
```


2.5 Debugging mit Werkzeugen

Beispiele:

```
foo1 <- function(x){
 foo2 <- function(x,s)
   x[[s]] + 5
 y <- x + 1
 foo2(y, s = -5)
}

debug(foo1)
foo1(1:5)
undebug(foo1)
```

2.6 Algorithmik

Kapitel

2 Funktionen
2.6 Algorithmik

Mit der Entwicklung von Algorithmen beschäftigen sich in erster Linie Informatiker. Aber auch Statistiker werden nicht umhin kommen, Algorithmen für eigene Methoden zu entwickeln. Auch die Zusammenstellung einer eigenen Simulation kann bereits als Algorithmus bezeichnet werden.

2.6 Algorithmik

Definieren wir einen **Algorithmus**:
Ein Algorithmus ist ein Verfahren, mit dessen Hilfe man die Antwort auf Fragen eines gewissen Fragenkomplexes nach einer vorgeschriebenen Methode erhält. Er muss bis in die letzten Einzelheiten eindeutig angegeben sein. Insbesondere muss die Vorschrift, die den Algorithmus angibt, durch einen Text (und Formeln) endlicher Länge gegeben sein.

Ein Algorithmus heißt **abbrechend**, wenn er für jede Frage des betrachteten Fragenkomplexes nach endlich vielen Schritten eine Antwort liefert. Andernfalls heißt der Algorithmus **nicht abbrechend**.

2.6 Algorithmik

- Das Ziel des Algorithmus ist die Beantwortung einer konkreten Fragestellung ...
- ... in möglichst kurzer Zeit (wenige Rechenschritte – Effizienz).
- Ein Algorithmus muss absolut präzise beschrieben sein. Das Wort „vielleicht“ kommt durch die Erzeugung von Zufallszahlen ins Spiel.
- Ein Algorithmus kann auch als Abfolge von Teil-Algorithmen beschrieben werden. Z.B. kann man als Teil eines Algorithmus zur Bestimmung des Medians sagen, dass die Daten sortiert werden sollen, ohne einen Sortieralgorithmus vollständig anzugeben.
- Durch diese Modularität können so immer größere, aber dennoch übersichtliche Algorithmen entstehen.
- Man sollte sich immer Gedanken über den Algorithmus machen, bevor man wild mit der Programmierung von Funktionen beginnt und es später wieder verwirft! **R** verzeiht hier sehr viel, dennoch entstehen nach reiflicher Überlegung bessere Programme.

2.6 Algorithmik — Flussdiagramme

Zur strukturierten und übersichtlichen Darstellung von Algorithmen werden Flussdiagramme verwendet, diese erleichtern

- die Erstellung der notwendigen Abläufe im Algorithmus,
- die strukturierte Programmierung von Algorithmen,
- die Reduktion der Arbeitsschritte auf Ablaufformen (z.B. Anweisungen, Fallunterscheidungen (if/else), Schleife, ...).

Symbole für Flussdiagramme
nach DIN 66001, die durch Kanten sinngemäß verbunden werden müssen, werden an der Tafel gezeigt.

Damit fällt oft auch die Umsetzung in eine Funktion (oder beliebigen Code anderer Programmiersprachen) leicht.

2.6 Algorithmik — Flussdiagramme

Beispiele zur Konstruktion von Algorithmen mittels Flussablaufdiagrammen:

- Median
- Fakultät
- Fibonacci-Zahlen

2.6 Simulationen

Algorithmen und deren Strukturierung und Laufzeitoptimierung sind gerade für Simulationen wichtig. Simulationen werden vom Statistiker sehr häufig eingesetzt:

- Struktur und Verhalten des zu untersuchenden Systems werden in geeignetem Abstraktionsgrad in einem Modell nachgebildet.
- Bei Simulation zeigt das (korrekte) Modell hoffentlich das Verhalten des realen Systems
  $\longrightarrow$ Überprüfung eines statistischen Modells
- Oft ist reales System nicht analysierbar, so dass Simulationen notwendig werden (Messungen zu teuer oder unmöglich, falls z.B. Menschenleben in Gefahr sind).
- Simulationen sind meist zeitdiskret und ereignisorientiert
- Als Statistiker verwendet man eher stochastische als deterministische Simulationen (bzw. deren Mischformen).

2.6 Simulationen

Eigenschaften von stochastischer Simulation:

- Simulation wird durch Pseudozufallszahlen gesteuert,
- Eingangsdaten und Ergebnisse sind Zufallsvariablen,
- Durch Mitführen von Auswertevariablen an beliebige Stellen im Simulationsmodell wird eine Möglichkeit zur detaillierteren Analyse geschaffen.
- Notwendig bei stochastischer Simulation:
  - Berechnung der Konfidenzintervalle der Ergebnisse,
  - Planung und Berechnung der Anzahl und der Dauer von Simulationsläufen.

3.1 Effizienz – Einführung

Kapitel

3 Effizienz

3.1 Einführung

Der Begriff der *Effizienz* wird im Bereich der Programmierung oft anders verwendet als in der Mathematik.

Mit Effizienz wird meist ein Algorithmus oder eine Implementierung gemeint, bei denen eine besonders geringe Laufzeit zu erwarten ist oder ein möglichst geringer Speicherverbrauch.

Der Begriff bleibt aber verschwommen und man sollte auch eine möglichst geringe Arbeitsbelastung und das „Blood Pressure Theorem“ beachten.

3.2 Programmierstil

Kapitel

3 Effizienz
3.2 Programmierstil

Je mehr man programmiert, desto mehr verbessert sich der Stil. Beachtet werden kann direkt:

- *Wiederverwendbarkeit*
- *Nachvollziehbarkeit*
- *Lesbarkeit*
- *Effizienz*

3.2 Wiederverwendbarkeit

Eine Funktion sollte möglichst allgemein geschrieben sein, also nicht nur auf dem aktuellen Datensatz, sondern auf beliebigen anwendbar sein.

Wenn man eine Funktion schreibt, liegt meist eine sehr konkrete Problemstellung vor. Oft wird man mit einem aktuell vorliegenden Datensatz Berechnungen anstellen wollen.

Beispiel: Sehr einfache Würfelsimulation
Ich muss ganz schnell mal simulieren, wie häufig die Zahl 8 bei 100 Würfen eines Würfels mit 10 Seiten fällt (schlechter Stil!):

```
wuerfel <- function(){
    x <- sample(1:10, 100, replace = TRUE)
    sum(x == 8)
}
wuerfel()
```


3.2 Nachvollziehbarkeit

Eine Funktion sollte so viele Kommentare wie möglich enthalten, am besten sogar eigene Dokumentation wie Hilfeseiten, die mit dem Packaging System von **R** einfach erzeugt werden können.
Nur so kann man selbst (nach einigen Wochen) und vor allem können andere (Kollegen) den Code noch verstehen.

3.2 Lesbarkeit

Zwischen den beiden folgenden Punkten zur *Kompaktheit* von Code muss ein Kompromiss gefunden werden:

- Sehr kompakt geschriebenen Code kann man sehr schlecht verstehen, es macht daher wenig Sinn zu versuchen, ganze Programme in einer Zeile unterzubringen.
- Auch zu wenig kompakter Code ist schlecht zu verstehen, da man die Übersicht bei zu vielen Variablen verliert.

3.2 Lesbarkeit

Auch das *Schriftbild* sollte gut lesbar sein:

- Je nach Kontext mit Einrückungen arbeiten, so dass z.B. bei Funktionen, bedingten Anweisungen und Schleifen sofort klar wird, welche Code Teile wozu gehören.
- Mit Leerzeichen für Übersichtlichkeit sorgen, insbesondere um Zuweisungszeichen und Gleichheitszeichen herum, sowie nach Kommata.
- Zeilen kürzer als etwa 66 Zeichen halten (R Core Standard).
- Nebenbei: Im Ausdruck sollte eine Schrift fester Zeichenbreite (z.B. *Courier New* unter Windows) verwendet werden.

3.3 Effizienz – Laufzeit und Speicherverbrauch

Kapitel

3 Effizienz

3.3 Laufzeit und Speicherverbrauch

Funktionen, die man sehr häufig verwenden will, veröffentlichen will, oder die Bestandteil längerer Simulationen sind, sollte man hinsichtlich ihrer Geschwindigkeit und ihres Speicherverbrauchs optimieren:

3.3 Effizienz – Laufzeit und Speicherverbrauch

- Eine simple Verdoppelung der Geschwindigkeit bedeutet z.B. einen statt zwei Tagen (Wochen, Monate) Rechenzeit.
- Eine Verringerung des Speicherverbrauchs bringt Geschwindigkeit, vor allem, wenn vermieden werden kann, virtuellen Speicher zu verwenden.
- Eine Verringerung des Speicherverbrauchs kann das lösen komplexer Probleme erst möglich machen, wenn sonst z.B. nicht genügend Hauptspeicher zu Verfügung stünde.
- Einige einfache Regeln für effizientes Programmieren kann man gleich von Anfang an beachten, andere muss man je nach Gegebenheit testen, wenn eine Optimierung nötig erscheint.

3.3 Effizienz – Laufzeit und Speicherverbrauch

Regeln für effizientes Programmieren:

- Vektorisiere! Keine Schleifen benutzen, wenn man mit Vektor– und Matrix–Operationen weiter kommt (vgl. Kapitel zu Schleifen und `apply()`).
- Benutze bereits implementierte Funktionen, wie etwa `optim()`. Diese greifen häufig auf vorhandenen schnellen *C* oder *Fortran* Code zurück.
- Initialisiere langsam wachsende Objekte in Schleifen komplett (Beispiel folgt!).
- Überprüfung von Fehlern nicht in Schleifen! Überprüfung von Argumenten vor, und von Ergebnissen nach der Schleife.
- Führe keine Berechnung, die nur einfach nötig sind, mehrfach aus, vor allem nicht in Schleifen (Beispiel folgt!).

3.3 Effizienz – Laufzeit und Speicherverbrauch

- Benutze Datenbanken im Falle großer Datensätze! **S** Implementierungen müssen Datensätze, auf denen operiert wird, im Hauptspeicher halten, was bei moderaten Datensätzen heute kein Problem darstellt (einfache PCs können heute leicht auf 2GB RAM aufgerüstet werden). Bei großen Datensätzen kann durch die Benutzung von Datenbanken durch das Laden von Teildatensätzen das RAM Problem umgangen werden.
- Sehr zeitkritische Teile des Codes kann man, nachdem sie auf Richtigkeit und Funktionalität überprüft sind, in *C* oder *Fortran* Code auslagern, und so sehr schnelle kompilierte Bibliotheken erstellen.
- Kaufe einen größeren Computer (schneller, mehr Speicher)!
- Benutze ein Rechencluster, Multiprozessor-Rechner oder Linux Cluster aus vielen Rechnern, um viele Teile der nötigen Berechnungen in separaten Prozessen gleichzeitig laufen zu lassen, falls parallelisierbar.

3.3 Effizienz – Laufzeit und Speicherverbrauch

- Für die sehr häufig verwendeten Matrix–Operationen der zeilen– bzw. spaltenweisen Summen– und Mittelwertberechnung gibt es die sehr performanten Funktionen `rowSums()`, `colSums()`, `rowMeans()`, `colMeans()`.

Man braucht ein Werkzeug, mit dem die Geschwindigkeit von Code gemessen werden kann, damit mehrere Varianten zur Lösung eines Problems verglichen werden können: `system.time()`.

3.3 Effizienz – Laufzeit und Speicherverbrauch

```
time0 <- function(n){             | time3 <- function(n){
  a <- NULL                       |   (1:n)^2
  for(i in 1:n) a <- c(a, i^2)    | }
  a                               |
}                                 |
time1 <- function(n){             |
  a <- NULL                       |
  for(i in 1:n) a[i] <- i^2       |
  a                               |
}                                 |
time2 <- function(n){             | system.time(a <- time0(1e5)) #!
  a <- numeric(n)                 | system.time(a <- time1(1e6))
  for(i in 1:n) a[i] <- i^2       | system.time(a <- time2(1e6))
  a                               | system.time(a <- time3(1e6))
}                                 |
```


3.3 Effizienz – Laufzeit und Speicherverbrauch

Beispiele:

```
a <- 1:50000

system.time({
    for(i in seq(along=a))
        a[i] <- 2 * n * pi * a[i]^2
})

system.time({
    for(i in seq(along=a))
        a[i] <- a[i]^2
    a <- 2 * n * pi * a
})
```

3.3 Effizienz – Laufzeit und Speicherverbrauch

Ein weiteres (professionelles) Werkzeug zur Überprüfung von Effizienz ist das s.g. *Profiling*.

Damit wird zu einer Funktion ein Profil erstellt, das aussagt, wieviel Zeit bei welcher Sub-Prozedur verwendet wird. Sollte eine Sub–Prozedur einen Großteil der Zeit verbrauchen, kann dort optimiert werden.

Details zum *Profiling* findet man im ersten **R** Newsletter in der *Programmer's Niche*.

Beispiele:

```
Rprof()          # log für Profiling wird geschrieben
example(glm)     # ein Beispiel mit sehr komplexem Code
Rprof(NULL)      # Anhalten der Aufzeichnung
summaryRprof()   # Auswertung, wer der Schuldige ist ...
```


4.1 Verteilungen – Stichproben ziehen

Kapitel

4 Verteilungen

4.1 Stichproben

Ein zentrales Gebiet bei der Programmierung in einer für Statistiker entwickelten Programmiersprache ist natürlich

- das Ziehen von Stichproben,
- die Erzeugung von Zufallszahlen gemäß gewisser Verteilungen,
- aber auch das Berechnen von Dichtefunktion, Verteilungsfunktion und Quantilen solcher Verteilungen.

4.1 Verteilungen – Stichproben ziehen

Mit der Funktion `sample()` lassen sich sehr einfach Stichproben ziehen bzw. zufällig Elemente eines Vektors auswählen. Die vollständige Syntax lautet:

`sample(x, size, replace = FALSE, prob = NULL)`.

Damit wird aus dem Vektor `x` eine Stichprobe der Größe `size` ohne Zurücklegen (`replace = FALSE`) gezogen. Sollen die Auswahlwahrscheinlichkeiten der einzelnen Elemente nicht gleich sein, so kann zusätzlich `prob` spezifiziert werden.

Achtung: Falls `x` ein numerischer Skalar (Länge 1) ist mit $x \geq 2$, dann wird eine Stichprobe aus der Sequenz `1:floor(x)` gezogen!

4.1 Verteilungen – Stichproben ziehen

Beispiele:

```
# Zufällig vier Zahlen zwischen 1 und 10 auswählen
# also eine Stichprobe aus 1:10 der Größe 4 ziehen:
sample(1:10)                       # alle ohne Zurücklegen
sample(1:10, 4, replace = TRUE)    # vier  mit Zurücklegen

x <- LETTERS[1:20]
sample(x, 5)                       # nicht nur integer

sample(5)                          # wie sample(1:5)
```

4.2 Funktionen für Verteilungen

Kapitel

4 Verteilungen
4.2 Funktionen für Verteilungen

Für die wichtigsten Verteilungen sind bereits Funktionen implementiert.

4.2 Funktionen für Verteilungen

Es gibt dabei meistens vier Typen von Funktionen, denen bei allen Verteilungen jeweils derselbe Buchstabe vorangestellt ist:

`d`	(density) für Dichtefunktionen
`p`	(probability) für Verteilungsfunktionen
`q`	(quantiles) für die Berechnung von Quantilen
`r`	(random) für das Erzeugen von Zufallszahlen

Nach diesen „magischen“ Buchstaben folgt dann der Name der Verteilung bzw. dessen Abkürzung, z.B. `norm` für die Normalverteilung oder `unif` (uniform) für die Rechteckverteilung.

`rnorm()` erzeugt somit normalverteilte Zufallszahlen, während `punif()` die Verteilungsfunktion der Rechteckverteilung berechnet.

4.2 Funktionen für Verteilungen

Hier einige implementierte Verteilungen, wobei der erste Buchstabe der Funktion gemäß der Tabelle auf der vorherigen Folie zu ersetzen ist:

Funktion	Verteilung	Funktion	Verteilung
`xbeta()`	Beta-	`xlnorm()`	Lognormal-
`xbinom()`	Binomial-	`xnbinom()`	negative Binomial-
`xcauchy()`	Cauchy-	`xnorm()`	Normal-
`xexp()`	Exponential-	`xpois()`	Poisson-
`xf()`	F-	`xt()`	t-
`xgamma()`	Gamma-	`xunif()`	Rechteck-
`xgeom()`	Geometrische-	`xweibull()`	Weibull-
`xhyper()`	Hypergeometrische-	`xwilcox()`	Vtlg. d. Wilcoxon-Stat.
`xlogis()`	Logistische-	`xchisq()`	χ^2–

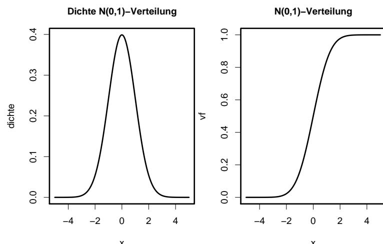
4.2 Funktionen für Verteilungen

Beispiele:

```
# 10 Zufallszahlen einer R[3,5]-Verteilung:
 runif(10, min = 3, max = 5)
# 0.25-Quantil der R[3,5]-Verteilung:
 qunif(0.25, min = 3, max = 5)

 x <- seq(-5, 5, 0.05)
# Dichte an den Stellen x einer N(0,1)-Verteilung:
 dichte <- dnorm(x, mean = 0, sd = 1)
 plot(x, dichte, main = "Dichte einer N(0,1)-Verteilung",
      type = "l")
# Verteilungsfunktion ...:
 vf <- pnorm(x, mean = 0, sd = 1)
 plot(x, vf, main="N(0,1)-Verteilung", type = "l")
```

4.2 Funktionen für Verteilungen



4.3 Zufallszahlen

Kapitel

4 Verteilungen

4.3 Zufallszahlen

Es wurde bereits gezeigt, wie Zufallszahlen verschiedener Verteilungen erzeugt werden können. Auch das Ziehen von Stichproben beruht natürlich auf der Erzeugung von Zufallszahlen.

- ZZ werden in Programmiersprachen von s.g. *Zufallszahlengeneratoren* erzeugt.
- ZZ sollen möglichst (fast) *keine Regelmäßigkeiten* enthalten.
- ZZ sollen *schnell* erzeugt werden können.
- ZZ sollen *reproduzierbar* sein, um z.B. eine Simulation wiederholen zu können und Ergebnisse nachvollziehen und bestätigen zu können.

4.3 Zufallszahlen

Vom Rechner erzeugte ZZ sind jedoch alles andere als zufällig, denn Sie müssen ja (durch Funktionen) berechnet werden. Außerdem sollen sie ja auch reproduzierbar sein.

Leider gibt es nicht den „optimalen" Zufallszahlengenerator. In **R** ist ein Zufallszahlengenerator Standard, der einen Kompromiss eingeht, so dass möglichst wenig Regelmäßigkeiten in den ZZ bei hoher Geschwindigkeit des Generators auftreten.

Andere Generatoren wählt man mit der Funktion `RNGkind()` aus.

Der ZZ–Generator wird normalerweise mit der Systemuhr initialisiert. Wenn man aber reproduzierbare Ergebnisse haben will, kann man einen Startwert mit der Funktion `set.seed()` definieren.

4.3 Zufallszahlen

Beispiele:

```
set.seed(1234)          # Startwert definieren
rnorm(2)                # A
rnorm(2)                # B

set.seed(1234)          # Startwert re-definieren
rnorm(2)                # wieder A

## anderen ZZ Generator auswählen:
RNGkind("Mersenne-Twister")
rnorm(2)                # nicht B
```

5.1 Statistische Verfahren

Kapitel

5 Verfahren
5.1 Überblick

Es gibt eine ganze Reihe von statistischen Verfahren in **R** . Hier wollen wir auf die wesentlichsten Funktionen eingehen, die in den Standard-Paketen enthalten sind und von elementarer Bedeutung für die tägliche Arbeit eines Statistikers sein können.

- Für einen Überblick empfehle ich erneut Venables und Ripley (2002).
- Für Beispiele und Details zum Funktionsaufruf bitte die Hilfe benutzen.

5.1 Statistische Verfahren – Lagemaße

Lagemaße

Funktion	Bedeutung
`mean(), median()`	$\bar{x}$, $\tilde{x}_{0.5}$
`quantile()`	Quantile
`summary()`	deskriptive Zusammenfassung eines Objekts

Streumaße

Funktion	Bedeutung
`range()`	Spannweite
`var(), cov()`	Varianz, Kovarianz
`cor()`	Korrelation
`mad()`	Median Absolute Deviation
`cor.test()`	KK nach Spearman, Kendall's τ

5.1 Statistische Verfahren – mathematische Hilfsfunktionen

Nützliche mathematische Hilfsfunktionen

Funktion	Bedeutung
`optimize()`	(lokale) Extrema einer stetigen Funktion
`uniroot()`	Nullstellen einer reellwertigen stetigen Funktion innerhalb eines Intervalls
`polyroot()`	Komplexe Nullstellen von Polynomen
`deriv()`	Partielle Differenzierung von Funktionen
`optim()`	Sammlung von Optimierungsverfahren, z.B. Nelder–Mead (nicht nur univariat!)

5.1 Testverfahren

Tests bei NV-Annahme und Anpassungstests

Funktion	Bedeutung
`t.test()`	t-Test
`var.test()`	F-Test
`ks.test()`	Kolmogorov-Smirnov-Test

5.1 Testverfahren

Nichtparametrische Tests

Funktion	Bedeutung
`binom.test()`	Binomialtest
`wilcox.test()`	Wilcoxontest
`friedman.test()`	Friedman-Test (RST)
`kruskal.test()`	Kruskal-Wallis-Test (RST)

Tests für Kontingenztafeln

Funktion	Bedeutung
`chisq.test()`	χ^2-Test
`fisher.test()`	Exakter Test von Fisher
`mcnemar.test()`	Test nach McNemar

5.2 Lineare Modelle

Kapitel

5 Verfahren

5.2 Lineare Modelle und das Formel Interface

- Lineare Modelle (in ihrer allgemeinen Form) sind eigentlich Stoff des 4. Semesters.
- Lineare Modelle sind von *wesentlicher Bedeutung* in der Statistik.
- Mindestens ein Spezialfall eines linearen Modells ist bereits aus dem 1. Semester bekannt: Lineare Regression einer erklärenden und einer abhängigen Variablen (Stichwort KQ-Schätzung).

5.2 Lineare Modelle

Elementare Funktionen für lineare Modelle:

- `lm()` rechnet ein lineares Modell.
- `glm()` steht für generalisierte lineare Modelle zur Verfügung.
- `anova()` rechnet eine Varianzanalyse zu einem (g)lm Objekts.
- `summary()` gibt viel mehr Details zu einem (g)lm Objekt.
- `plot()` zeichnet Grafiken zur Residualanalyse.
- `predict()` sagt neue Werte mit zuvor geschätztem Modell vorher und bestimmt Konfidenzintervalle.

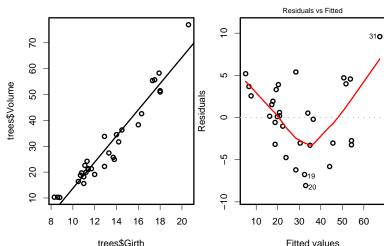
5.2 Lineare Modelle

Beispiel:
Schauen wir uns einen Datensatz mit Baumdaten an. Die Variablen „Volume" und „Girth" (Umfang) scheinen einen nahezu linearen Zusammenhang zu haben.

```
plot(trees)                        # Scatterplotmatrix ...
plot(trees$Girth, trees$Volume) # ... etwas größer

lm(trees$Volume ~ trees$Girth)  # lineare Regression, oder:
Baum.lm <- lm(Volume ~ Girth, data = trees)
Baum.lm                         # Parameterschätzung zeigen
abline(Baum.lm)                 # Regressionsgerade einzeichnen
summary(Baum.lm)                # viel mehr Details, u.a. R^2
plot(Baum.lm)                   # Oh je!
# Das ist noch ein schlechtes Modell, Verbesserung in der Übung!
```

5.2 Lineare Modelle



5.2 Lineare Modelle

Gründe für Schätzung statistischer Modelle:

- Versuch, Zusammenhänge zu verstehen.
- Aufgrund neuer Messwerte das Ergebnis (z.B. das Baumvolumen) vorhersagen.

```
## Datensatz basteln:
neueBaeume <- data.frame(Girth = c(11.95, 12.05))
## Aus Modell mit der Schätzung neue Daten vorhersagen:
predict(Baum.lm, neueBaeume)
```

Auch Konfidenz- und Prognoseintervalle sind sehr interessant. Hier möchte man den Grad der Sicherheit für die Schätzung (Konfidenz) bzw. die Sicherheit für die Vorhersage (Prognose) neuer Werte kennenlernen:

5.2 Lineare Modelle

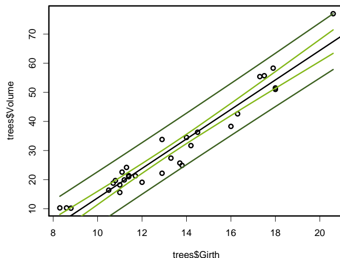
Beispiele:

```
## Grafik wiederherstellen:
plot(trees$Girth, trees$Volume)
abline(Baum.lm)

## Konfidenzintervall berechnen und zeichnen:
int <- predict(Baum.lm, interval ="confidence")
lines(trees$Girth, int[, "lwr"], col="blue")
lines(trees$Girth, int[, "upr"], col="blue")

## Prognoseintervall berechnen und zeichnen:
int <- predict(Baum.lm, interval ="prediction")
lines(trees$Girth, int[, "lwr"], col="red")
lines(trees$Girth, int[, "upr"], col="red")
```


5.2 Lineare Modelle



5.2 Lineare Modelle

Die Syntax für die Modellspezifikation:

Nr.	Zeichen	Beispiel	Bedeutung
1)	`~`	`y ~ x1`	*y* ist abhängig von *x*
2)	`+`	`y ~ x1 + x2`	Hinzufügen weiterer erklärender Variablen
3)	`*`	`y ~ x1 * x2`	Hinzufügen weiterer erklärenden Variablen inkl. Interaktionen
4)	`:`	`y ~ x1 + x2 + x1:x2`	Direkte Spezifikation von Interaktionen (= Modell 3)
5)	`-`	`y ~ x1 * x2 - x1:y2`	Herausnahme von Termen (= Modell 2)
6)	`-1`	`y ~ x1 - 1`	Herausnahme des Intercept (Achsenabschnitt)
7)	`I()`	`y ~ I(x1 + x2)`	As is: Im Modell mathematisch rechnen
8)	`.`	`y ~ .`	Alle Variablen im data.frame einschließen

5.2 Lineare Modelle (Beispiel)

```
R> lm.obj <- lm(Volume ~ Girth + Height, data = trees)
R> summary(lm.obj)
### Das ist noch ein schlechtes Modell, Verbesserung in der Übung!
Call: lm(formula = Volume ~ Girth + Height, data = trees)

Residuals:
    Min      1Q  Median      3Q     Max
-6.4065 -2.6493 -0.2876  2.2003  8.4847

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -57.9877     8.6382  -6.713 2.75e-07 ***
Girth         4.7082     0.2643  17.816  < 2e-16 ***
Height        0.3393     0.1302   2.607   0.0145 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.882 on 28 degrees of freedom
Multiple R-Squared: 0.948,      Adjusted R-squared: 0.9442
F-statistic:   255 on 2 and 28 DF,  p-value: < 2.2e-16
```


5.3 Zeitreihen

Kapitel

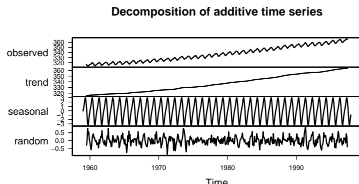
5 Verfahren

5.3 Zeitreihen

Auch Zeitreihen sind in Statistik 1 behandelt worden, bekannte Funktionalität bietet z.B. die Funktion `decompose()`. Damit werden Trend, Saisonfigur etc. „erkannt" und dargestellt.

5.3 Zeitreihen

```
m <- decompose(co2)
plot(m)
```



5.4 Klassifikationsverfahren

Kapitel

5 Verfahren

5.4 Klassifikationsverfahren

In diesem letzten Abschnitt zu statistischen Verfahren in R folgt eine Zusammenstellung einiger Klassifikationsverfahren.

5.4 Klassifikationsverfahren

Zusammenstellung einiger Klassifikationsverfahren:

- `lda(), qda()` für lineare bzw. quadratische Diskriminanzanalyse in Paket **MASS**
- `rda()` für regularisierte Diskriminanzanalyse in Paket **klaR**
- `knn()` für *k*-nächste Nachbarn in Paket **e1071**
- `naiveBayes()` für naive Bayes Klassifikation in Paket **e1071**
- `rpart()` für rekursive Partitionierung (Regressions- und Klassifikationsbäume) in Paket **rpart**
- `ctree()` für Conditional Inference Trees in Paket **party**
- `randomForest()` für random Forests in Paket **randomForest**
- `svm()` für Support Vector Machines in Paket **e1071**
- `svmlight()` für andere Support Vector Machines in Paket **klaR**
- `predict()` stets für die Vorhersage aus gelernten Modellen.

5.4 Klassifikationsverfahren

Beispiele:

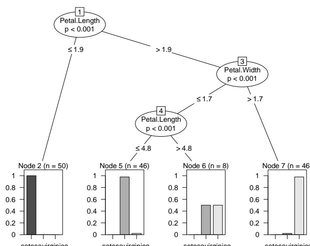
```
library("MASS")
irislda <- lda(Species ~ ., data = iris)
predict(irislda)$class

library("klaR")
partimat(Species ~ ., data = iris, method = "lda")

library("rpart")
irisrpart <- rpart(Species ~ ., data = iris)
plot(irisrpart);    text(irisrpart)

library("party")
irisct <- ctree(Species ~ ., data = iris)
plot(irisct)
```


5.4 Klassifikationsverfahren



6.0 Pakete

Kapitel

6 Pakete 1

Dieser Abschnitt soll eine Einführung zu den vielen R Paketen auf CRAN und deren Benutzung sowie deren Administration geben.

6.0 Pakete

- **Paket**: strukturierte, standardisierte Einheit. Diese besteht aus **R** Code, Dokumentation, Daten, externem Quellcode usw.
- Pakete werden mit `library("Paketname")` geladen und mit `detach()` entladen.
- Hilfe zu Paketen (anstelle von Funktionen) gibt es mit `library(help = "Paketname")`.
- Auf CRAN sind Pakete erhältlich, für alle (un)denkbaren Teilgebiete der Statistik.
- Das BioConductor Projekt betreibt eigene Paket-Repositories
- Bei Standard–Installation werden die Pakete *base, datasets, graphics, grDevices, methods, stats* und *utils* beim Starten geladen.
- Neben *base* werden wichtige weitere Pakete gleich installiert (Details auf den nächsten Folien).

6.0 zusätzliche „standard packages“

Paket	Beschreibung
base	das Basis Paket
compiler	Byte Code Compiler
datasets	Sammlung von Beispieldatensätzen
graphics	Grafik Funktionen
grDevices	Grafik Devices
grid	ein Re-Design für Grafik-Layout
methods	**S** 4 Methoden nach Chambers (1998)
parallel	Parallelisierung von R Programmen
splines	splines
stats	allgemeine Statistik Funktionen (Tests usw.)
stats4	auf **S** 4 Klassen basierende Statistik Funktionen
tcltk	Programmierung von GUI mit / Interface zu *tcl/tk*
tools	Werkzeuge zur Paketentwicklung und -verwaltung
utils	Sammlung nützlicher Hilfsfunktionen

6.0 zusätzliche „recommended packages“

Paket	Beschreibung
boot	Bootstrap Verfahren nach Davison und Hinkley (1997)
cluster	Clusterverfahren nach Rousseeuw et al.
codetools	Codeanalyse
foreign	Routinen für Import und Export von Daten der Statistikpakete Minitab, S, SAS, SPSS, Stata, ...
KernSmooth	Kerndichteschätzung, -glättung nach Wand & Jones (1995)
lattice	Implementierung von Trellis Grafik nach Cleveland (1993)
Matrix	Matrix Klassen, z.B. zum effizienten Umgang mit sparseness
mgcv	Generalisierte Additive Modelle
nlme	(Nicht-) Lineare Modelle mit gemischten Effekten nach Pinheiro und Bates (2000)
rpart	Rekursive Partitionierung
survival	Überlebenszeitanalyse (z.B.: Hazardfunktionen, Cox Modell, Zensierung)

6.0 Pakete von V&R

class	Klassifikation
MASS	Funktionen-Sammlung von Venables and Ripley (2002): „Modern Applied Statistics with S“
nnet	Neuronale Netze (feed-forward) mit einer versteckten Schicht — und multinomiale log-lineare Modelle.
spatial	Spatial Statistik

6.0 Pakete aus Bibliotheken laden

- Installierte Pakete liegen in einer Bibliothek (=library), einem Verzeichnis und werden daraus geladen mit:
  `library("Paketname", lib.loc = Pfad_zur_Bibliothek)`
- `.libPaths()` zeigt an, welche Bibliotheken automatisch durchsucht werden
- Bibliothek entweder mit `.libPaths()` zum Suchpfad hinzufügen
- oder vor dem Start von **R** in der Umgebungsvariablen `R_LIBS` eintragen, z.B. mit der Datei `.Renviron`:
  `R_LIBS=/home/user/myR/myLib;/home/user/myR/develLib`
- Basis-Pakete und empfohlene Pakete sind in der Haupt-*Library* im Verzeichnis `R_HOME/library`
- `R_HOME` ist das Verzeichnis zur entsprechenden Version von **R** , z.B. `/usr/local/lib/R` oder `c:\Programme\R-x.y.z`.
- Als Voreinstellung werden neue Pakete in die erste Library von `.libPaths()` installiert.

6.0 Pakete aus Bibliotheken laden

Beispiele:

```
library(help = "survival")  # Übersicht, Hilfe
library("survival")         # Laden des Pakets
detach("package:survival")  # Entfernen aus dem Suchpfad
.libPaths("c:/temp")        # Setzen einer Library
.libPaths()
```


6.0 Sinn von Bibliotheken

Sinn mehrerer Bibliotheken:

- Strukturierung der Pakete
- Aufteilung in Entwickler / Benutzerversionen
- zentrale Installation (ohne Schreibrecht für Benutzer) vs. lokale Bibliothek eigener Pakete

Beispiele:

- Zentrale Bibliothek der Standardpakete, z.B. `n:\software\R-x.y.z\library`,
- zentrale Bibliothek der CRAN Pakete, z.B. `n:\software\Rlibs\CRAN`,
- lokale eigene Benutzerbibliothek, z.B. `d:\meins\meineRlibs\work`,
- lokale eigene Entwicklerbibliothek, z.B. `d:\meins\meineRlibs\devel`.

6.0 Sourcepakete

Source-Pakete sind i.d.R. unabhängig von der verwendeten Plattform (Hardware, Betriebssystem)

- Voraussetzungen für Installation eines Source-Pakets: Perl, evtl. C(++) Compiler, Fortran compiler, ...
- CRAN akzeptiert ausschließlich Source-Pakete
- Standard Form der Paketverteilung für Unix-artige Systeme (z.B. Linux, Solaris, ...), da hier meist bereits Voraussetzungen erfüllt sind.

6.0 Binärpakete

Binärpakete sind plattform-spezifisch und können auch abhängig von der verwendeten **R** Version sein

- Binärpakete können aber ohne spezielle zusätzliche Werkzeuge installiert werden: „shared object files“ und DLL, Hilfeseiten und Meta-Informationen sind in Binärpaketen bereits vor-kompiliert
- CRAN enthält Binärpakete für die aktuellsten **R** Versionen für verschiedene Plattformen, z.B. Windows und MacOS X. Binärpakete für Windows werden meist innerhalb von zwei Tagen nach den Sources bereitgestellt

6.0 Pakete installieren und verwalten

Dokumentation:

- Handbuch „R Installation and Administration“
- „The R FAQ“ und „R for Windows FAQ“
- „R Help Desk: Package Management“ in *R News 3(3)*

Repositories:

- CRAN (+ CRAN extras unter Windows), BioConductor
- `setRepositories()` oder `options("repos" = ...)` zur Auswahl des Repositories
- `chooseCRANmirror()` wählt den Spiegelserver aus

6.0 Installation von Paketen

```
install.packages("Paket", lib = "/Pfad/zur/library")
```

- automatisches Herunterladen der aktuellsten Version aus dem Repository und Installieren
- Spezifikation der *Library* kann unterbleiben (dann erste Library im Suchpfad)
- Das Argument `dependencies = TRUE` installiert alle anderen Pakete, für die Empfehlungen und Abhängigkeiten vom Paket deklariert sind.

6.0 Update von Paketen

```
update.packages()
```

- installiert neuere Versionen aus den Repositories
- mit `checkBuilt = TRUE` wird auch nach Update von **R** ein Neukompilieren aller Pakete veranlasst

7.1 OOP

Kapitel

7 OOP
7.1 Einführung

OOP! Die Kurzform für *Objekt–Orientiertes Programmieren*.

Wir wissen schon:

- *Alles* ist ein Objekt (jegliche Art Daten und Funktionen)!
- **R** ist eine *objektorientierte Sprache*!

7.1 OOP

Und was bedeutet *objektorientiert*?

- Ein Objekt kann ein Attribut „`class`“ (*Klasse*), oder eine Liste mehrerer haben (z.B. „numeric“, „matrix“ oder „lm“).
- Eine Funktion kann für verschiedene Klassen jeweils angepasste *Methoden* mitbringen. Eine solche Funktion nennt man *generisch*.
- Vorteil: Der Benutzer muss nicht viele Funktionen für verschiedene Objektklassen kennen, sondern kann einfach die generische Funktion benutzen.

7.1 OOP

- Eine typische *generische* Funktion ist `plot()`. Um zu erfahren, welche *Methoden* es gibt, fragt man: `methods(plot)`.
- Methoden in Paketen mit Namespaces (Erklärung am Ende des Kurses) sind versteckt, auf sie kann man z.B. mit dem `:::` Operator zugreifen: `Namespacename:::Methodenname`
- Die neuen **S4** Methoden werden im grünen Buch (Chambers, 1998), aber auch in *S Programming* (Venables und Ripley, 2000) ausführlich beschrieben.
- Der Vorteil der neuen Methoden ist, dass die Syntax und die zur Verfügung stehenden Tools wesentlich konsistenter sind. Den wahren *objektorientierten* Programmierer wird das freuen. Der doch höhere Aufwand des Programmierens einer neuen Klasse gehört jedoch nicht zur schnellen täglichen Arbeit des Ausprobierens.
- Die **S3** Methoden sind weiterhin unverzichtbar und bilden die Basis von **R** .

7.2 OOP – S3

Kapitel

7 OOP
7.2 S3

Zunächst eine Zusammenfassung wichtiger Funktionen:

`attributes()`	Erfragen und Setzen aller Attribute eines Objekts
`attr()`	Erfragen und Setzen konkreter Attribute
`class()`	Erfragen und Setzen des Klassen–Attributs
`inherits()`	Von einer anderen Klasse Merkmale *erben*
`methods()`	Alle zu einer generischen Funktion gehörenden Methoden
`NextMethod()`	Verweis auf die nächste in Frage kommende Methode
`UseMethod()`	Verweis von der generischen Funktion an die Methode

7.2 OOP

Beispiele:

```
x1 <- rnorm(30)
x2 <- factor(sample(1:3, size = 30, replace = TRUE))
y <- rnorm(30)
z <- lm(y ~ x1)

plot(x1)
plot(x2)
plot(z)     # Für jeden Plot wird die richtige Methode gewählt!
methods(plot)

summary(x)  # Ebenso für das Erstellen der Zusammenfassung.
summary(y)
summary(z)
```


7.2 OOP – S3 (Anwendungsbeispiel)

```
## bekannt aus dem Teil zu linearen Modellen:
Baum.lm <- lm(Volume ~ Girth, data = trees)
## neues Objekt (und Objektklasse) konstruieren:
object <- list(data = trees[, c("Girth", "Volume")],
               lm.obj = Baum.lm)
class(object) <- "nurSoEinName"
print.nurSoEinName <- function(x, ...){
    cat("Data:\n=====\n\n")
    print(x$data, ...)
    cat("\n\nEstimated model:\n===============\n")
    print(x$lm.obj, ...)
    invisible(x)
}
object
```


7.2 OOP – S3 (Anwendungsbeispiel)

```
plot.nurSoEinName <- function(x,
  main = "Regression, Konfidenz- und Prognoseintervalle", ...){

    plot(x$data, main = main, ...)
    ## Konfidenzintervall berechnen und zeichnen:
    int <- predict(x$lm.obj, interval ="confidence")
    lines(x$data[,1], int[, "lwr"], col="blue")
    lines(x$data[,1], int[, "upr"], col="blue")
    ## Prognoseintervall berechnen und zeichnen:
    int <- predict(x$lm.obj, interval ="prediction")
    lines(x$data[,1], int[, "lwr"], col="red")
    lines(x$data[,1], int[, "upr"], col="red")
}
plot(object)
```

7.3 OOP – S4

Kapitel

7 OOP

7.3 S4

Hier eine Liste von Funktionen für das OOP nach „neuem“ Stil (S4):

`setClass()` `getClass()`	Definition einer neuen Klasse und deren Abfrage
`setValidity()`	Definition einer Validitäts–Prüfung
`new()`	Erzeugen eines Objekts einer Klasse
`getSlots()` `slotNames()`	Arbeiten mit Slots
`setGeneric()` `setMethod()`	Definition einer generischen Funktion und deren Methode(n)
`selectMethod()`	zur expliziten Methodenwahl
`oldClass()`, `setOldClass()`	zum Arbeiten mit alten Klassen

7.3 OOP – S4

Ein Blick in S4 Klassen anhand eines längeren **Beispiels**, beginnend mit der Klassendefintion:

```
## Definiere Klasse "Wave":
setClass("Wave",
    slots = c(left = "numeric", right = "numeric",
        stereo = "logical", samp.rate = "numeric",
        bit = "numeric"),
    prototype = prototype(stereo = TRUE, samp.rate = 44100,
        bit = 16))

## konkretes Objekt erstellen:
waveobj <- new("Wave")
waveobj@stereo <- FALSE
str(waveobj)
```


7.3 OOP – S4

Validitätsüberprüfung ist minimal bereits dadurch gegeben, dass die Slots einen vordefinierten Typ haben (character, numeric, ...).

In unserem Beispiel möchte man z.B. zusätzlich sichergehen, dass es im rechten wie im linken Kanal gleich viele Samples gibt:

7.3 OOP – S4

```
WaveValid <- function(object) {
  len.r <- length(object@right)
  if(len.r != 0 && len.r != length(object@left))
    return("'left' and 'right' slots must have the same length")
  if(length(object@stereo) != 1)
    return("'stereo' must have length 1")
  sr <- object@samp.rate
  if(length(sr) != 1 || sr <= 0)
    return("'samp.rate' must be a positive number")
  if(length(object@bit) != 1 || !(object@bit %in% c(8, 16)))
    return("'bit' must be one of 8 or 16 (of length 1)")
  ## else : all is fine
  TRUE
}
setValidity("Wave", WaveValid)  # deklariere Validitätsprüfung
```

7.3 OOP – S4

- Anstelle von `print()` wird bei S4 Objekten auch `show()` zur Ausgabe des Objekts auf der Konsole benutzt (und wird von `print()` eigentlich aufgerufen).
- Für echte Fans als Übung: Eine `plot()` Methode schreiben, die für Stereo Waves beide Kanäle in eine Grafik ausgibt.

7.3 OOP – S4

```
showWave <- function(object) {
    l <- length(object@left)
    cat("\nWave Object")
    cat("\n\tNumber of Samples:      ", l)
    cat("\n\tDuration (seconds):     ",
        round(l / object@samp.rate, 2))
    cat("\n\tSamplingrate (Hertz):   ", object@samp.rate)
    cat("\n\tChannels (Mono/Stereo):",
        if(object@stereo) "Stereo" else "Mono")
    cat("\n\tBit (8/16):             ", object@bit, "\n\n")
}
## Setze showWave() als S4 Methode für "Wave" Objekte
## zur generischen Funktion show():
setMethod("show", signature(object = "Wave"), showWave)
```


8.1 Parallel

Kapitel

8 Parallel
8.1 Vorüberlegungen

Zunächst einige Vorüberlegungen zu parallelem Rechnen, u.a. Wahl der Rechner und Beachtung der Kosten.

8.1 Parallele Programme: Vorüberlegungen

Kosten für High Performance Computing am Beispiel der *Amazon Elastic Compute Cloud (Amazon EC2)* (Stand 24.09.2017):

- Virtuelle Maschine (1 core, 2 Gb RAM, 160 Gb Plattenplatz): ca. **0.025$** pro Stunde
- Datentransfer: **0.09$** pro Gb
- Datenstorage: **0.11$** pro Gb und Monat plus 0.11$ pro 10^6 I/O requests
- Rechnung: 100 Cores für 3 Tage Simulationen: $100 \cdot 3 \cdot 24 \cdot 0.025 = 180\$$ (ohne Speicher/Datentransfers), davon ca. Stromkosten:
  0.2 kW Verbrauch pro Rechner bei 0.25 EUR Stromkosten pro kWh und 8 Cores im Rechner, mal zwei wegen Kühlung:
  $0.25 \cdot 0.2 \cdot (100/8) \cdot 3 \cdot 24 \cdot 2 = 90$ EUR.

8.1 Parallele Programme: Vorüberlegungen

Zur Wahl der Plattform:

- Was ist eine **Plattform**?
  'Platform – the computer hardware and operating system software that runs application software.'
  (`http://nces.ed.gov/pubs98/tech/glossary.asp`)
- R ist verfügbar und verbreitet benutzt unter verschiedenen Plattformen, insbesondere:
  - OS: AIX, HP-UX, Linux, Mac OS, Solaris, Windows usw.
  - CPU: Itanium, PowerPC, (Ultra)Sparc, ix86, x86-64 usw.

8.1 Parallele Programme: Vorüberlegungen

Speicherverbrauch:

- wenig RAM $\longrightarrow$ virtueller Speicher ist viel langsamer
  - echter RAM: $\approx$ 20 Gb/sec
  - virtueller RAM (=Festplatte/SSD) $\approx$ 0.5 Gb/sec
- 32 bit OS: limitierter Adressraum
  - bis max. 4 Gb
- Windows:
  - Adressraum für einzelnen 32-bit R Prozess in 32-bit Windows: 2 Gb
  - Adressraum für einzelnen 32-bit R Prozess in 64-bit Windows: 3.5 Gb

8.1 Parallele Programme: Vorüberlegungen

Geschwindigkeit:

- Linux gewinnt gegen Windows ($\sim$ 10%)
- Einsatz optimierter (B)LAS (Basic Linear Algebra Subprograms) als Subplattform:
  - viel besser optimierte Matrix Operationen
  - Ausnutzung von Level 1 bis Level 3 Cache in der CPU

8.1 Parallele Programme: Vorüberlegungen

Beispiele für paralleles Rechnen auf mehreren Rechnern:

- Simulation verschiedener Einstellungen eines „Versuchsplans“
- einzelne Läufe einer Kreuzvalidierung
- Bootstrap Läufe
- Verschiedene Ketten einer MCMC Simulation
- **aber NICHT**: einzelne MCMC Kette

Voraussetzung ist **Parallelisierbarkeit**!

8.1 Parallele Programme: Vorüberlegungen

Parallelisieren: Wo und wie stark?

- Parallelisierung i.d.R. auf der höchsten Ebene eines Programms beginnen.
- Auf der niedrigsten Ebene können auf Multi-Core Systemen parallelisierte BLAS verwendet werden.
- Bei m vorhandenen CPUs mindestens m parallele Prozesse erzeugen können.
- Keine Berechnung / Simulation länger als $\frac{1}{2}$ Woche – Rechner könnte herunterfahren oder Fehler unbemerkt bleiben!
- Keine Berechnung / Simulation kürzer als 5 Minuten (als Faustregeln).

8.2 Parallele Programme: Werkzeuge

Nun werden **Werkzeuge zum parallelen Rechnen in R** beschrieben, dabei werden insbesondere Beispiele zum Basis Paket **parallel** gezeigt.

- **multicore** Paket zum parallelen Rechnen auf einer physikalischen Maschine mit mehreren CPU Kernen – geht nicht unter Windows, wichtige Teile sind seit R-2.14.0 im Paket **parallel**.
- **nws** (Network Spaces) Paket für verteiltes Rechnen auf mehreren physikalischen Maschinen bei großen Datenmengen
- **foreach** Paket zur Konstruktion parallel ausgeführter Schleifen

8.2 Parallele Programme: Werkzeuge

- **snow** (Simple Network of Workstations) Paket für die Clusterbildung, dass sowohl mehrere CPU Kerne pro Maschine nutzen kann und dazu verschiedene low-level Lösungen nutzen kann:
  - MPI (Message Passing Interface, Paket Rmpi)
  - nws (Network Spaces)
  - PVM (Parallel Virtual Machine, Paket rpvm)
  - sockets (lokal oder über ssh, einfach aber recht langsam, wird jetzt gezeigt)

  Die wichtigsten Teile sind heute im R Basis Paket **parallel** (seit R-2.14.0).
- weitere Pakete (auch kommerzielle wie ParallelR)
- eine Zusammenfassung gibt es unter `http://cran.r-project.org/web/views/HighPerformanceComputing.html`

8.2 Parallele Programme: Werkzeuge

Es folgt ein kurzes Beispiel mit dem **parallel** Paket, anhand dessen wir einige Probleme und nützliche Funktionen kennenlernen:

- `makeCluster()` zum Erzeugen eines Clusters
- `clusterCall()` oder `clusterEvalQ()` zum Aufrufen einer Funktion auf allen Knoten
- `clusterExport()` zum Verteilen eines Objekts an alle Knoten
- `clusterSetRNGStream()` zum Initialisieren des Zufallszahlengenerators
- `parSapply()` , `parLapply()` und `parApply()` (oder `parRapply`, `parCapply`) als parallele Versionen von `sapply()` und Freunden.
- `clusterApply()` und `clusterApplyLB()` als low level Funktionen zum parallelen Aufruf von Code, letztere mit load balancing
- `stopCluster()` zum Anhalten der Clusterfähigkeit

8.2 Parallele Programme: Werkzeuge

```
R> library("parallel")        # setup the cluster:
R> cl <- makeCluster(rep("localhost", 3), type = "SOCK")
R> clusterCall(cl, runif, 3)  # run on all nodes
  [[1]] [1] 0.6984976 0.8827487 0.1643290
  [[2]] [1] 0.8635299 0.0240543 0.4089040
  [[3]] [1] 0.6984976 0.8827487 0.1643290

R> ## initialize parallel RNG:
R> clusterSetRNGStream(cl)
R> clusterCall(cl, runif, 3)  # again
  [[1]] [1] 0.0142665 0.7493918 0.0073161
  [[2]] [1] 0.8390032 0.8424790 0.8896625
  [[3]] [1] 0.2724742 0.5006002 0.4281134

R> stopCluster(cl)            # stop for now
```


8.2 Parallele Programme: Werkzeuge

10^7 Zufallszahlen – wie schnell wird es?

```
R> ## 100 Mal soll foo() durchgeführt werden:
R> foo <- function(x) rnorm(100000)

R> ### ohne Parallelisierung:
R> system.time(L <- sapply(1:100, foo))
R> ## 4.5 Sek.

R> ### Mit Parallelisierung, 5 Rechner:
R> library("parallel")
R> cl <- makeCluster(c("rao", "kendall", "tukey", ...),
R+                   type = "SOCK")
R> clusterSetRNGStream(cl)

R> system.time(L <- parSapply(cl, 1:100, foo))
R> ## 10 Sek.!

R> stopCluster(cl)
```


8.2 Parallele Programme: Werkzeuge

Aggregation über 10^7 Zufallszahlen

```
R> ## 100 Mal soll foo() durchgeführt werden:
R> foo <- function(x) sum(rnorm(100000))

R> ### ohne Parallelisierung:
R> system.time(L <- sapply(1:100, foo))
R> ## 4.2 Sek.

R> ### Mit Parallelisierung, 5 Rechner:
R> library("parallel")
R> cl <- makeCluster(c("rao", "kendall", "tukey", ...),
R+                   type = "SOCK")
R> clusterSetupRNG(cl)

R> system.time(L <- parSapply(cl, 1:100, foo))
R> ## 1 Sek.!

R> stopCluster(cl)
```


9.1 Erweiterbarkeit

Kapitel

9 Erweiterbarkeit
9.1 Einführung

Wie bereits mehrmals erwähnt, ist **R** sehr auf Erweiterbarkeit durch den Benutzer ausgelegt. So ist es möglich

- einfach eigene Funktionen zu erstellen (bereits gezeigt),
- standardisierte Dokumentation zu den eigenen Funktionen zu erstellen,
- eigenen *C*, *C++* oder *Fortran* Code als *shared library (DLL)* einzubinden, und von **R** aufzurufen,
- packages zu erstellen, die alle (oder einige der) oben genannte(n) Punkte enthalten, und die man einfach installieren, also auch weitergeben kann.

9.1 Warum Packages?

Es gibt einige Punkte, die für solches „Verpacken" von Funktionen, Sourcen und Dokumentation sprechen:

- dynamisches Laden und Entladen des Packages (Speicherplatz sparend)
- einfache Installation und Update von lokalen Datenträgern oder über das Web, innerhalb von **R** oder über die Kommandozeile des Betriebssystems
- einfache Administration – Globale (über Fachbereichsserver) und lokale, eigene Library–Verzeichnisse gleichzeitig nutzbar
- Validierung – **R** bietet Befehle zur groben Überprüfung von Code, Dokumentation und Installierbarkeit, sowie, falls man möchte, Überprüfung von Rechenergebnissen.
- einfache Verteilung der Software an Dritte (Standard Mechanismus)
- Beispieldatensätze können auch ins Package!

9.1 Das S-PLUS (8) Paketsystem und CSAN

Proposed S-PLUS® Packages

- **An S-PLUS® package is a collection of S-PLUS® functions, data, help files and other associated source files that have been combined into a single entity for distribution to other S-PLUS® users.**
- **This package system is modeled after the package system in R.**
- **Insightful Corporation hosts the Comprehensive S-PLUS® Archival Network (CSAN) site at http://csan.insightful.com/ to facilitate S-PLUS® package distribution.**
- **Packages can be downloaded from the CSAN websites in two forms: as raw source code or as Windows binaries.**

Insightful 8 Copyright © 1999 – 2006 Insightful Corporation, All Rights Reserved.

9.2 Struktur von Packages

Kapitel

9 Erweiterbarkeit

9.2 Paketentwicklung

9.2 Struktur von Packages

Ein Package besteht aus einigen Standard–Dateien und Verzeichnissen, die für bestimmte Arten von Dateien gedacht sind (Details im Manual *Writing* **R** *Extensions*):

- `DESCRIPTION` (Datei) mit standardisierten Erläuterungen zu Autor, Lizenz, Titel, Abhängigkeiten etc. Die Datei wird automatisch von Skripten weiterverarbeitet.
- `INDEX` (Datei) kann automatisch generiert werden und enthält den Index aller Daten und Funktionen.
- `man/` (Verzeichnis) enthält Dokumentation, unter **R** im `*.Rd` Format.
- `R/` (Verzeichnis) enthält **R** Code.
- `data/` (Verzeichnis) enthält Datensätze.
- `src/` (Verzeichnis) enthält *C*, *C++* oder *Fortran* Sourcen.
- `tests/` (Verzeichnis) für Software – Validierung.

9.2 Packages: Daten und Funktionen

Ein nützliches Paket wird aber sowohl Dokumentation, als auch mindestens **R** Code oder Datensätze enthalten.

- Jeder Datensatz und jede Funktion stehen i.A. in einer separaten Datei.
- Datensätze können mit `data()` geladen werden und in folgenden Formen im `data/` Verzeichnis vorliegen:
  - als „rechteckige" Textdatei (durch Leerzeichen oder Kommata getrennt),
  - als mit `dump()` exportierter source code und
  - als mit `save()` exportierte **R** Binärdatei.
- Code, der beim Laden eines Packages direkt ausgeführt werden soll, wird per Konvention in der Datei `R/zzz.R` abgelegt.

9.2 Packages: Dokumentation

Mit `prompt()` erstellt man ein Gerüst für die Dokumentation einer Funktion. Solche `*.Rd`–Dateien haben eine LaTeX ähnliche Syntax und enthalten folgende Abschnitte:

`\name`	Name der Hilfeseite (meist = `\alias`)
`\alias`	Name(n) der Funktion(en), die beschrieben werden
`\title`	Überschrift der Hilfeseite
`\description`	Kurze Beschreibung der Funktion
`\usage`	Wie man die Funktion (inkl. aller Argumente) aufruft
`\arguments`	Liste aller Argumente und deren Bedeutung
`\value`	Liste der zurückgegebenen Werte
`\details`	Falls nötig, eine detaillierte Beschreibung der Funktion
`\references`	Hinweise auf Literaturstellen
`\seealso`	Verknüpfungen zu relevanter Dokumentation anderer Funktionen
`\examples`	Beispiele, wie man die Funktion benutzen kann
`\keyword`	ein Standard Keyword, für Indizierung der Funktionen

9.2 Packages: Dokumentation

- Es gibt weitere (und selbst definierbare) Abschnitte, ebenso spezielle Möglichkeiten mathematische Formeln, URLs und Verknüpfungen zu anderen Hilfeseiten einzufügen, sowie vieles mehr ...
- Das **R** Packaging System überprüft beim Erstellen der Dokumentation, dass der `\usage` Part mit der tatsächlichen Definition der Funktion übereinstimmt.
- Die im Abschnitt `\examples` definierten Beispiele werden testweise durch das Packaging System überprüft.
- Dokumentation aus einer `*.Rd`–Datei kann direkt erzeugt werden mit den Befehlen
  - `R CMD Rdconv` zur Konvertierung in LaTeX, HTML und formatierten ASCII Text.
  - `R CMD Rd2dvi` zur Konvertierung in DVI und Adobes PDF Format.

9.2 Packages erstellen, prüfen, installieren

Um ein neues Package zu erstellen, gibt es die Funktion `package.skeleton()`:

```
package.skeleton(name = "MeinPaket", Objektliste, path = ".")
```

legt das Gerüst für `MeinPaket` mit den Dateien der Objekte aus `Objektliste` im aktuellen Pfad an. Ebenso wird eine grobe `DESCRIPTION` Datei angelegt und Gerüste der nötigen Hilfedateien angelegt. **R** sagt auch, was man als nächstes zu tun hat.

- Wenn man die Dateien wie erforderlich editiert hat, kann man mit `R CMD build` das Paket zusammenpacken.
- `R CMD check` überprüft die Konsistenz, Installierbarkeit, Dokumentation etc.
- `R CMD INSTALL` installiert das Paket.

9.3 Eigener C, C++ oder Fortran Code

Kapitel

9 Erweiterbarkeit
9.3 Externer Code

Wir beschränken uns auf `.Call()`, obwohl es weitere Möglichkeiten gibt mit den Interfaces `.C()`, `.Fortran()` und `.External()`.

- Eine Menge wichtiger Macros zum Benutzen von Code ist in den Header–Dateien `R.h` und `Rinternals.h` definiert.
- Eine weitere Header–Datei mit Macros ist `Rdefines.h`.

Wenn man den Code nicht innerhalb eines Packages kompilieren und laden lässt, so geschieht das Kompilieren mit `R CMD SHLIB` und das Laden mit `dyn.load(Dateiname)`.

Vorteil von kompiliertem Code: Geschwindigkeit!

9.3 Eigener C, C++ oder Fortran Code

Als einfaches **Beispiel** zeigen wir die Addition zweier reeller Vektoren *a* und *b* durch Benutzung eines `.Call()`.

Datei `c:\test.c`:

```
#include <Rinternals.h>
SEXP addiere(SEXP a, SEXP b)
{
  int i, n;
  n = length(a);
  for(i = 0; i < n; i++)
    REAL(a)[i] += REAL(b)[i];
  return(a);
}
```


9.3 Eigener C, C++ oder Fortran Code

Möglicher **R** Code:

```
dyn.load("c:/test.dll")          # Laden der Library
# oder library("Packagename"), falls in einem Package ....

# Definition einer aufrufenden R Funktion:
addiere <- function(a, b){
    if(length(a) != length(b))
        stop("a und b müssen gleich lang sein!")
    .Call("addiere", as.double(a), as.double(b))
}

addiere(4:3, 8:9)                  # Klappt!
```


9.4 Namespaces

Kapitel

9 Erweiterbarkeit
9.4 Namespaces

Zusätzliche Regeln zu den beschriebenen Scoping Rules (Regeln z.B. zur Suchreihenfolge in verschiedenen Umgebungen) wurden durch die Einführung von *Namespaces* geschaffen.

9.4 Namespaces

- Die Anzahl an Zusatzpaketen steigt, es kommt daher unvermeidlich zu Konflikten zwischen Funktionen gleichen Namens in gleichzeitig benutzten Paketen.
- Namespaces definieren, welche Objekte für den Benutzer und andere Funktionen (im Suchpfad) sichtbar sind und welche nur innerhalb des eigenen Namespaces sichtbar sind.
- Funktionen, die aus einem Paket mit Namespace nicht explizit *exportiert* werden, sind nur für andere Funktionen innerhalb desselben Namespaces sichtbar. Funktionen, die nur zur Strukturiertheit dienen und nicht für Aufruf durch den Benutzer gedacht sind, können so „versteckt“ werden.

9.4 Namespaces

Wer in einem Paket eine Funktion

```
beispiel <- function(x)
    sin(2 * pi * x)
beispiel(1:5)          # erwartetes Ergebnis:
                       # [1] -2.449213e-16 -4.898425e-16 . . .
```

schreibt, erwartet vermutlich, dass die darin verwendeten Objekte `sin()` und `pi` aus dem Paket **base** stammen. Gleichlautende Funktionen in anderen Paketen oder im Workspace würden sie aber wegen der Scoping Rules „überschreiben“, wie z.B. in:

```
sin <- sum
pi <- 0.5
beispiel(1:5)          # Summe der Zahlen (1:5) = 15
```


9.4 Namespaces

- Durch Namespaces wird sichergestellt, dass keine Objekte des Pakets **base** maskiert werden.
- Man kann auch Objekte aus beliebigen anderen Paketen *importieren*, die dann ihrerseits nicht mehr durch neue Objekte, z.B. im Workspace, maskiert werden können. Pakete, die nur durch im Namespace angegebene *Imports* geladen werden, werden nicht in den Suchpfad gehängt.
- Eine in einem Namespace definierte Funktion sucht nach Objekten gemäß dem angegebenen Suchpfad – mit folgender Ausnahme: Zuallererst wird im eigenen Namespace gesucht, dann in den in den Namespace importierten Objekten, danach im **base** Paket und danach erst in dem bereits bekannten Suchpfad.

9.4 Namespaces

- Für expliziten Zugriff auf ein Objekt in einem konkreten Paket mit Namespace kann der Operator „`::`“ benutzt werden, der den Namen des Namespaces von dem Objektnamen trennt. Mit `stats::ks.test` greift man auf das Objekt (Funktion) `ks.test` in Namespace `stats` zu.
- In seltenen Fällen, z.B. um Fehler in einem Paket zu finden oder ein Paket zu entwickeln, möchte man auf eine nicht aus einem Namespace exportierte Funktion zugreifen: `getFromNamespace()`.
- Der Operator „`:::`“ bietet Zugriff auf nicht exportierte Objekte eines Namespaces.
- `fixInNamespace()`: Ändern / Ersetzen einer nicht exportierten Funktion.
- `getS3method()`: direkt auf eine zu einer generischen Funktion gehörenden Methoden-Funktion zuzugreifen, die nicht aus einem Namespace exportiert wird.

9.4 Namespaces

- `getAnywhere()`: alle Objekte innerhalb des Suchpfads und in geladenen Namespaces finden, auch wenn das gesuchte Objekt nicht exportiert wird bzw. der geladene Namespace nicht im Suchpfad eingehängt ist.

Beispiele:

```
library("MASS")                 # MASS laden
lda                             # Funktion lda: generisch
methods(lda)                    # Welche Methoden gibt es?
lda.default                     # lda.default wird nicht exportiert
getS3method("lda", "default")# Trotzdem anschauen ....
getAnywhere("lda.default")
MASS:::lda.default
```


10.0 Literatur — S Definitionen

- Becker, R.A. and Chambers, J.M. (1984): **S**, *an Interactive Environment for Data Analysis and Graphics*, Wadsworth, Belmont. (brown!)
- Becker, R.A., Chambers, J.M. and Wilks, A.R. (1988): *The New* **S** *Language*, Wadsworth & Brooks/Cole, Pacific Grove. (blue!)
- Chambers, J.M. and Hastie, T.J. (1992): *Statistical Models in* **S**, Wadsworth & Brooks/Cole, Pacific Grove. (white!)
- Chambers, J.M. (1998): *Programming with Data – A Guide to the* **S** *Language*. Springer, New York. (green!)

10.0 Literatur — R Definitionen

- Gentleman, R. and Ihaka, R. (2000): Lexical Scope and Statistical Computing. *Journal of Computational and Graphical Statistics 9*, 491–508.
- Ihaka, R. and Gentleman, R. (1996): **R**: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics 5*, 299–314.
- Murrell, P. and Ihaka, R. (2000): An Approach to Providing Mathematical Annotation in Plots. *Journal of Computational and Graphical Statistics 9*, 582–599.

10.0 Literatur — R – Standardwerke I

- Braun, W.J. and Murdoch, D.J. (2007): *A First Course in Statistical Programming with R*, Cambridge University Press, Cambridge.
- Chambers, J.M. (2008): *Software for Data Analysis: Programming with R*, Springer, New York.
- Dalgaard, P. (2008): *Introductory Statistics with R*, 2. Auflage, Springer, New York.
- Everitt, B. and Hothorn, T. (2009): *A Handbook of Statistical Analysis Using R*, 2. Auflage, Chapman & Hall/CRC, Boca Raton.
- Fox, J. (2011): *An R and S–PLUS Companion to Applied Regression*, 2. Auflage, Sage Publications, Thousand Oaks.
- Murrell, P. (2005): *R Graphics*, Chapman & Hall/CRC, Boca Raton.
- Pinheiro, J.C. and Bates, D.M. (2001): *Mixed effects models in S and S–PLUS*, Springer, New York.

10.0 Literatur — R – Standardwerke II

- Venables, W.N. and Ripley, B.D. (2002): *Modern Applied Statistics with S*, 4th ed., Springer, New York. (yellow!)
- Venables, W.N. and Ripley, B.D. (2000): *S Programming*, Springer, New York.

Diesem Kurs liegt das folgende deutsche Buch zu Grunde:

- Ligges, U. (2009): *Programmieren mit R*, 3. Auflage, Springer, Heidelberg.

10.0 Literatur — Online

Unter anderem ist die folgende Literatur im PDF Format frei auf CRAN unter `http://cran.r-project.org/other-docs.html` erhältlich oder zumindest gelinkt (alle über 100 Seiten und als „brauchbar“ befunden):

- Burns, P.J. (1998): *S Poetry*, `http://www.burns-stat.com/pages/spoetry.html`.
- Maindonald, J. (2001): *Using R for Data Analysis and Graphics*.

The R Journal ist Online zu lesen unter `http://journal.r-project.org/`.

10.0 Literatur — Online (R Core Team)

Die folgenden Online–Manuals werden vom R–Project auf CRAN unter `http://CRAN.R-Project.org/manuals.html` publiziert und sind in jeder **R** Version enthalten:

- Venables, W.N., Smith, D.M., and the R Core Team (2017): *An Introduction to R*.
- R Core Team (2020): *R Data Import / Export*.
- Hornik, K. (2020): *R FAQ*.
- R Core Team (2020): *R Installation and Administration*.
- R Core Team (2020): *R Language Definition*.
- R Core Team (2020): *R: A Language and Environment for Statistical Computing*.
- R Core Team (2020): *Writing R Extensions*.