Ludwig-Maximilians-Universität München

Fakultät für Mathematik, Informatik und Statistik

Institut für Statistik

Bachelorarbeit

über das Thema

„Ein Vergleich der Programmiersprachen R und Julia“

---

vorgelegt von

Thi Thuy Pham

Prüfer und Betreuer: Hr. Prof. Dr. Christian Heumann

Bearbeitungszeit: 07.05.2020 – 28.07.2020

Abstract

Das Ziel der vorliegenden Bachelorarbeit war es die zwei Programmiersprachen R und Julia zu vergleichen. Es wurde hierbei überprüft, ob die junge Programmiersprache Julia mit der beliebten Programmiersprache R im Bereich der statistischen Datenanalyse mithalten oder diese sogar übertreffen kann. Diese Abschlussarbeit führt zunächst grundlegend in die beiden Sprachen ein, also das grundlegende Arbeiten mit R und Julia und anschließend wird das Resultat einer kleinen durchgeführten statistischen Analyse am bekannten R-Datensatz iris vorgestellt. Die Bachelorarbeit ist für all diejenigen interessant, die auf der Suche nach einer guten Statistiksoftware sind und sich in der finalen Entscheidung nicht zwischen R und Julia entscheiden können, aber auch für alle Nutzer der Programmiersprache R, die sich fragen, ob es sich lohnt eine weitere Programmiersprache wie Julia zu erlernen. Um die Bachelorarbeit komplett zu verstehen wird die Kenntnis über die Grundlagen der Statistik vorausgesetzt.

Inhaltsverzeichnis

1. Einleitung

In der Statistik ist das Arbeiten mit Statistik-Softwares nicht mehr wegzudenken, denn mit ihrer Hilfe können große Datenmengen mit teilweise rechenintensiven Methoden analysiert werden. Ganze Teilbereiche der Statistik haben den Einzug in die Datenanalyse neuer Softwares zu verdanken [13]. Die gängigen Statistik-Softwares unterscheiden sich hinsichtlich ihrer Stärken und Schwächen wie auch in ihrer Handhabung [12]. Unter den vielen angebotenen Programmen hat sich neben der Programmiersprache Python auch R vor allem unter den Statistikern als sehr beliebt bewährt (Abb.1+2). Eine junge und aufstrebende Perspektive stellt die Programmiersprache Julia dar (Abb.1+2). Diese wurde mit dem Ziel erschaffen, so einfach für Statistik zu sein wie R es ist [7]. Genauer gesagt versucht sie die Zugänglichkeit und Produktivität der Statistik-Software R mit der Performance der kompilierten Sprache C zu verbinden [11]. Stellt diese vielversprechende Sprache eine Alternative zu R im Bereich der statistischen Analyse und graphischen Darstellung von Daten dar? Um diese Fragestellung zu beantworten, werden diese beiden Sprachen in dieser Arbeit genauer unter die Lupe genommen, indem sie unter den verschiedensten Aspekten miteinander verglichen werden.

Abb 1: Ranking der verschiedenen Programmiersprachen weltweit (Stand: Juli 2020)

(Quelle: http://pypl.github.io/PYPL.html)

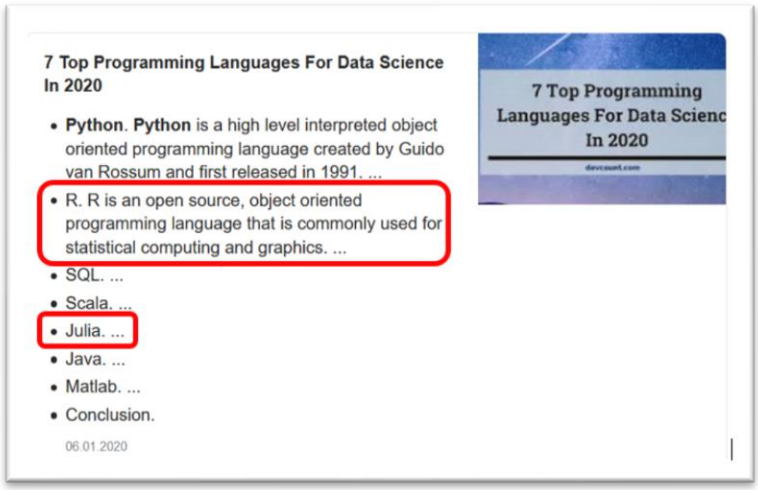
Worldwide, Jul 2020 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	31.73 %	+3.9 %
2		Java	17.13 %	-2.7 %
3		Javascript	7.98 %	-0.3 %
4		C#	6.67 %	-0.6 %
5	↑	C/C++	5.93 %	+0.1 %
6	↓	PHP	5.64 %	-1.1 %
7		R	4.14 %	+0.3 %
8		Objective-C	2.61 %	-0.1 %
9		Swift	2.29 %	-0.1 %
10	↑	TypeScript	1.91 %	+0.2 %
11	↓	Matlab	1.74 %	-0.1 %
12		Kotlin	1.62 %	+0.2 %
13	↑↑	Go	1.37 %	+0.2 %
14		VBA	1.27 %	-0.0 %
15	↓↓	Ruby	1.26 %	-0.1 %
16		Scala	0.99 %	-0.1 %
17		Visual Basic	0.83 %	-0.2 %
18	↑	Rust	0.81 %	+0.3 %
19	↑↑↑↑↑	Dart	0.52 %	+0.2 %
20	↑↑↑	Ada	0.47 %	+0.1 %
21	↑	Lua	0.46 %	+0.1 %
22	↓↓	Abap	0.46 %	-0.1 %
23	↓↓	Groovy	0.43 %	-0.1 %
24	↓↓↓↓↓	Perl	0.42 %	-0.2 %
25		Cobol	0.41 %	+0.1 %
26	↑↑	Julia	0.37 %	+0.1 %
27	↓	Haskell	0.29 %	+0.0 %
28	↓	Delphi	0.25 %	-0.0 %

© Pierre Carbonnelle, 2020

Abb.2**:** Ranking der Programmiersprachen insbesondere für Data Science (Stand: 06.01.2020)

(Quelle: https://hackr.io/blog/best-programming-languages-to-learn-2020-jobs-future)



2. Vorstellung der Programmiersprachen

Bevor die Programmiersprachen R und Julia miteinander verglichen werden, sollten diese erstmals vorgestellt werden. Was ist R beziehungsweise was ist Julia und warum lohnt es sich einen Blick auf diese beiden Sprachen zu werfen?

2.1. R

2.1.1. Überblick

Die Programmiersprache R[1] wurde im Jahr 1992 von den Statistikern Ross Ihaka und Robert Gentlman an der Universität Auckland entwickelt, im Jahr 1993 veröffentlicht, seit Mitte der 90er Jahre von einem Entwickler-Kollektiv (R-Core) betreut und ursprünglich für Statistiker entwickelt. Sie ist eine freie und kostenlose Software-Umgebung zur computergestützten statistischen Datenverarbeitung mit über 14000 gelisteten Zusatzpaketen auf R's größtem Open-Source-Paketarchiv CRAN. Dabei bezeichnet R sowohl das Programm selbst als auch die Sprache, in der die Auswertungsbefehle geschrieben werden. Der Name der Sprache ist auf den Anfangsbuchstaben der Vornamen der Entwickler zurückzuführen und ist in Anlehnung an der Sprache S entstanden. Sie ist eine der am häufigsten verwendeten Programmiersprachen für Datenanalyse und Machine Learning. Die aktuelle Version (Stand:22.06.2020) ist R-4.0.2.

2.1.2. Vor- und Nachteile

Wie jede Programmiersprache, hat auch R seine Stärken und Schwächen
Zu ihren Vorteilen zählen unter anderem:

- Open Source: Die Sprache ist eine Open Source Software und kostenlos zu erwerben. Dementsprechend sind keine Lizenzgebühren zu zahlen. Dadurch ist es möglich, Anpassungen gemäß den Anforderungen vorzunehmen.
- Schnelle Entwicklung: Die befehlsgesteuerte Arbeitsweise erhöht durch die Wiederverwendbarkeit von Befehlssequenzen für häufig wiederkehrende Arbeitsschritte langfristig die Effizienz. Zudem steigt die Zuverlässigkeit von Analysen, wenn bewährte Bausteine immer wieder verwendet und damit Fehlerquellen ausgeschlossen sind [8].
- Leichte Erweiterbarkeit mit Paketen: Für R wurde eine Vielzahl an Zusatzpaketen erstellt, die es einem ermöglichen, auch spezielle Aufgaben zu lösen. R ist somit in jeder Branche verwendbar, aufgrund seines leistungsstarken Paket-Ökosystems.

---

1. https://www.google.com/search?client=firefox-b-d&q=r+download

- Keine Einschränkung bzgl. der Betriebssysteme: R ist auf (fast) jedem Betriebssystem lauffähig und nahtlos ausführbar
- Transparenz: Die Weitergabe von Befehlssequenzen an Dritte kann die Auswertung für andere transparent und überprüfbar machen. Auf diese Weise lassen sich Auswertungsergebnisse jederzeit exakt selbst und durch andere reproduzieren, wodurch die Auswertungsobjektivität steigt [8].
- Große Online-Community: Unterstützung erhält man bei Ratlosigkeit durch eine mittlerweile große aktive Online-Community um R, die sich in den letzten Jahren entwickelt hat.

Allerdings hält R für Anwender auch Hürden bereit, insbesondere für Einsteiger:

- GUI fehlt: R hat keine vollständige graphische Benutzeroberfläche und muss daher auf externe anwendungsfreundlichere Programme zurückgreifen.
- Nicht nur nützliche Pakete: Viele Pakete und einige Funktionen gibt es mehrfach, davon sind ein paar Pakete qualitativ unterdurchschnittlich.
- Hohe Lernbereitschaft: Um in R Daten umfassend auswerten zu können, muss zunächst ein Grundverständnis für die Arbeitsabläufe und ein gewisser Wortschatz häufiger Befehle geschaffen werden. Dies erfordert eine Bereitschaft zur Einübung ihrer Syntax.
- Keine Sicherheitsfunktionen: Befehle sollten aktiv in Erinnerung bleiben, da keine Gedächtnisstützen zur Verfügung stehen.
- Fehlerintolerante Befehle: Die Befehlssteuerung ist nicht fehlertolerant, das heißt man muss die Befehle exakt eingeben, damit diese ausgeführt werden können. Ansonsten liefert R Fehlermeldungen, mit denen man zumindest auf die Ursachen zurückschließen kann. Durch das regelmäßige Arbeiten mit R treten solche Syntaxfehler immer seltener auf.
- Schlechte Speicherverwaltung: Bei der Analyse von sehr großen Datenmengen, muss R Datensätze zur Bearbeitung im Arbeitsspeicher vorhalten, was die Größe von praktisch auswertbaren Datensätzen einschränkt [8].

2.1.3. IDE: RStudio

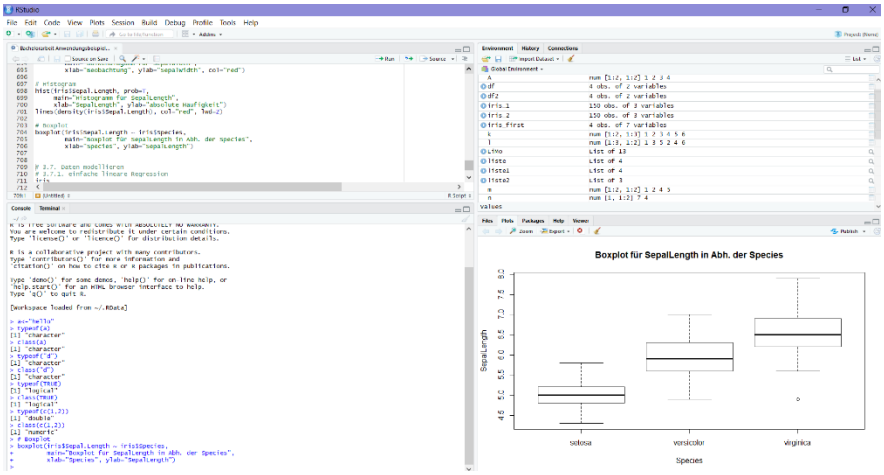
Zusätzlich zur Basis-Oberfläche von R bietet das Unternehmen RStudio[2], Inc außerdem die kostenlose integrierte Entwicklungsumgebung (Integrated Development Environment, kurz: IDE) und grafische Benutzeroberfläche (Graphical User Interface, kurz GUI) RStudio für die R-Programmierung an. Diese setzt dabei voraus, dass R selbst bereits installiert wurde. Sie läuft unter mehreren Betriebssystemen

---

2. https://rstudio.com/products/rstudio/download/

mit einer konsistenten Oberfläche und erleichtert häufige Arbeitsschritte (z.B. Syntax Markierung, Klammerpaare zählen, Einrückung, Verbindung zum R Prozess, wodurch sich die Benutzerfreundlichkeit von R erhöht. Außerdem unterstützt RStudio besonders gut die Möglichkeiten, Dokumente mit eingebetteten R-Auswertungen zu erstellen. Andere Texteditoren, die ebenfalls empfehlenswert sind, sind Tinn-R[3] (kostenfreier Editor, der Syntax-Highlighting und eine Anbindung an R anbietet) und Plugins (R-Plugins für Notepad++, Emacs, Vim usw.).

Abb.3: Oberfläche der Entwicklungsumgebung RStudio



2.2. Julia

2.2.1. Überblick

Die Entwicklung der Programmiersprache Julia[4] fing 2009 durch Stefan Karpinski, Viral B. Shah, Jeff Bezanson am Massachusetts Institute of Technology (MIT) an und sie wurde im Februar 2012 als Open-Source-Software veröffentlicht. Sie wurde vor allem für numerisches und wissenschaftliches Programmieren entwickelt. Ziel der Entwickler war es, die gesammelten Erfahrungen in der Arbeit mit unterschiedlichen Programmiersprachen in eine neue Sprache umzuwandeln, welche die besten Elemente von C, Matlab, Java, Ruby, Perl, Python und R – also die Vorteile dynamischer und statischer Sprachen – miteinander vereint. Die aktuelle Version (Stand: 23. Mai 2020) ist Julia 1.4.2 und sie ist unter allen gängigen Betriebssystemen nutzbar.

---

3. https://tinn-r.soft112.com/

4. https://julialang.org/downloads/

2.2.2. Vor- und Nachteile

Vorteile

- Eine Sprache nicht nur für die Wissenschaft: Julia kann ähnlich wie C, C++ und Python mittlerweile auch als General Purpose Language verwendet werden, das heißt sie eignet sich auch für allgemeine Programmieraufgaben. Dadurch hebt sie sich von anderen wissenschaftlichen Sprachen wie MATLAB oder R erheblich ab.
- Hohe Geschwindigkeit und gute Performance: Ein herausragendes Merkmal von Julia ist die hohe Ausführungsgeschwindigkeit, welche mit die der Programmiersprachen C, C++ und Fortran vergleichbar ist. Dies könnte ein Vorteil für die Statistik sein, da man hier auf gute Perfomance angewiesen ist. Gründe für die Schnelligkeit liegen unter anderem in der Arbeitsweise mit einem Just-in-Time Compiler und der Möglichkeit Programme parallel und verteilt auszuführen.
- Sie erfordert wie Python einen geringen Programmieraufwand.
- Schnell erlernbar: Aufbauend auf wenigen grundlegenden Sprachelementen und einer leicht verständlichen Syntax können in kurzer Zeit produktive Programme entwickelt werden. Dadurch steht einer steilen Lernkurve nichts im Wege.

Nachteile

- Unerfahrenheit: Julia ist eine noch relativ neue Sprache, dementsprechend ist sie noch nicht so weit entwickelt wie manch andere Sprachen wie Python, die sich bereits in der Programmierwelt etablieren konnten.
- Noch unbekannt: Auch ist die Sprache noch nicht so weit verbreitet. Aus diesem Grund ist die Community rund um Julia noch relativ klein.
- Startschwierigkeiten: Es herrscht zudem ein Mangel an Online-Lehr- und Lernressourcen. Dies erschwert Einsteigern das Erlernen der Sprache und sich mit dieser ausreichend auseinander zu setzen.

2.2.3. IDE: Juno (Atom)

Juno[5] ist eine leistungsfähige integrierte Entwicklungsumgebung für Julia, die auf Atom[6], einem Texteditor, der von GitHub bereitgestellt wird, basiert. Sie bietet leistungsstarke Werkzeuge, die einem bei der Code-Entwicklung unterstützen. Juno besteht sowohl aus Julia- als auch aus Atom-Paketen, um Julia-spezifische Verbesserungen hinzuzufügen, wie z.B. Syntax-Highlighting, ein Plot-Fenster,

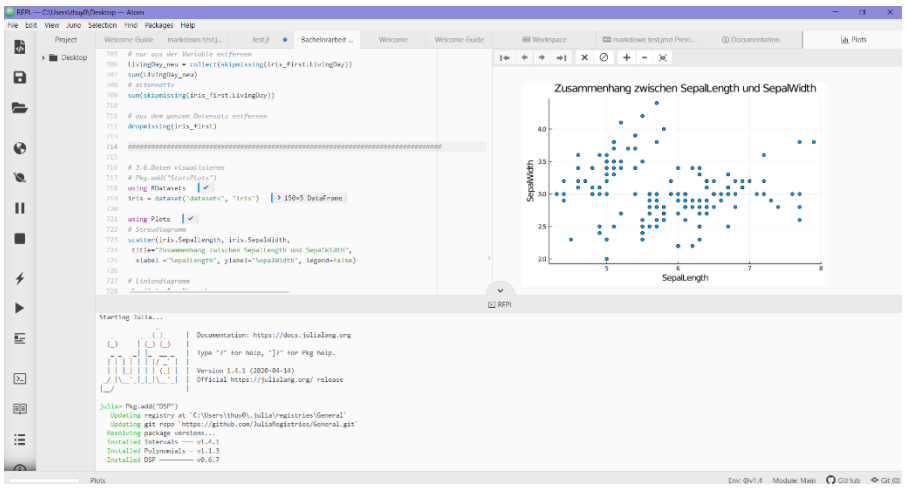
---

5. https://junolab.org/

6. https://atom.io/

Integration mit Julias Debugger, eine Konsole zum Ausführen von Code und vieles mehr [14]. Nennenswerte Texteditoren, die auch häufig genutzt werden, sind unter anderem VS Code[7], Jupyter[8] und Vi/Vim[9].

Abb.4: Oberfläche der Entwicklungsumgebung Juno in Atom mit dem One Light Theme



7. https://code.visualstudio.com/download

8. https://jupyter.org/install

9. https://www.vim.org/download.php

3. Vergleich der beiden Programmiersprachen

Abb.5: R versus Julia

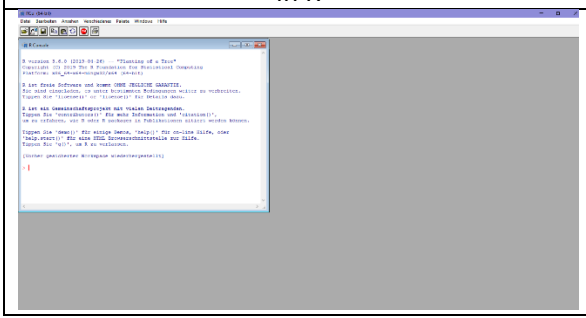
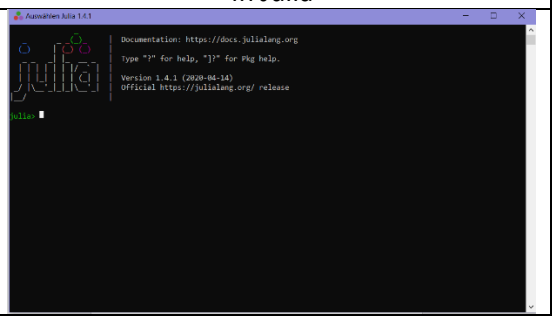
(Quelle: https://www.r-project.org/logo/, https://github.com/JuliaLang/julia-logo-graphics, https://www.123rf.com/photo_89935202_stock-vector-boxing-gloves-fight-icon-blue-and-red-boxing-glove-icon-fighting-concept-isolated-on-white-backgroun.html)

3.1. Erste Schritte

3.1.1. Grundlegende Elemente

Starten: Unter Windows lassen sich R und Julia über die bei der Installation erstellte Verknüpfung auf dem Desktop beziehungsweise im Startmenü starten. Hierdurch öffnen sich in R zwei Fenster: ein großes, das Programmfenster, und darin ein kleineres, die Konsole. In Julia dagegen öffnet sich nur ein Fenster. Unter MacOS und Linux ist die R-Konsole das einzige sich öffnende Fenster und für Julia öffnet sich wie unter Windows auch nur ein Fenster (Abb.6).

Abb.6: Konsole unter Windows

in R	in Julia
	

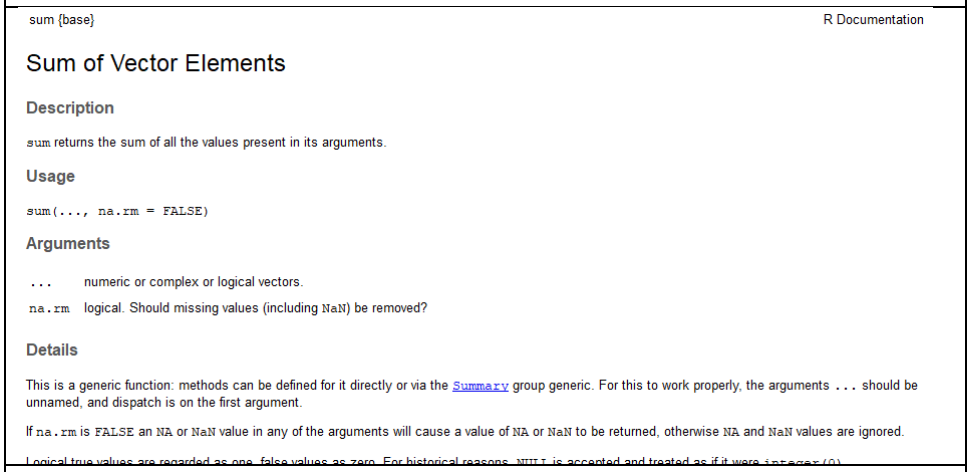
Bedienung: Zur Eingabe ist die Groß- und Kleinschreibung wesentlich. Die interaktiven Eingaben an der Eingabeforderung „>“ in R bzw. „julia>“ in Julia werden mit Enter abgeschlossen und dann ausgeführt. Mit den Cursor-Tasten kann man zusätzlich in R Befehle aus der Command-Line-History holen und bearbeiten.

Hilfe: Hilfe zu einem Befehl erhält man in R mit „?*befehl*“. Auch in Julia kann man mit demselben Ausdruck Hilfe erhalten, wobei nach dem Eingeben des Fragezeichens erst „help?>“ erscheint und dann kann man erst den Befehl eingeben, zu dem man Hilfe benötigt (Abb.7). Außerdem kann man in R noch mit „help.search(„*Stichwort*“)“ in der Hilfe suchen und die Online-Dokumentation startet man mit „help.start()“ im Webbrowser.

Abb.7: Hilfe rufen	
In R	In Julia
`> ?sum()`	`help?> sum()`

Abb.8: Ausschnitt des Hilfedokuments zu einem Befehl

In R



sum {base} R Documentation

Sum of Vector Elements

Description

`sum` returns the sum of all the values present in its arguments.

Usage

```
sum(..., na.rm = FALSE)
```

Arguments

`...`	numeric or complex or logical vectors.
`na.rm`	logical. Should missing values (including `NaN`) be removed?

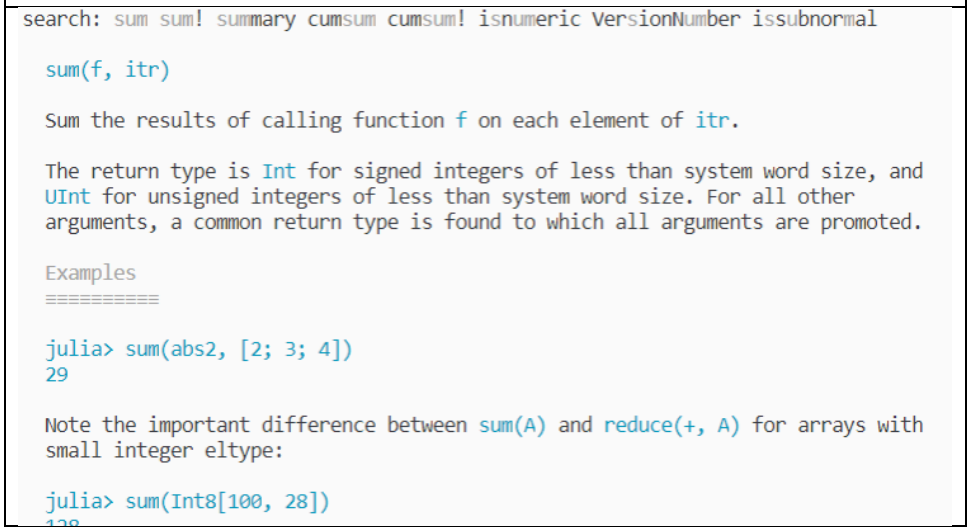
Details

This is a generic function: methods can be defined for it directly or via the `Summary` group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

If `na.rm` is `FALSE` an `NA` or `NaN` value in any of the arguments will cause a value of `NA` or `NaN` to be returned, otherwise `NA` and `NaN` values are ignored.

Logical true values are regarded as one, false values as zero. For historical reasons, `NULL` is accepted and treated as if it were `integer(0)`.

In Julia

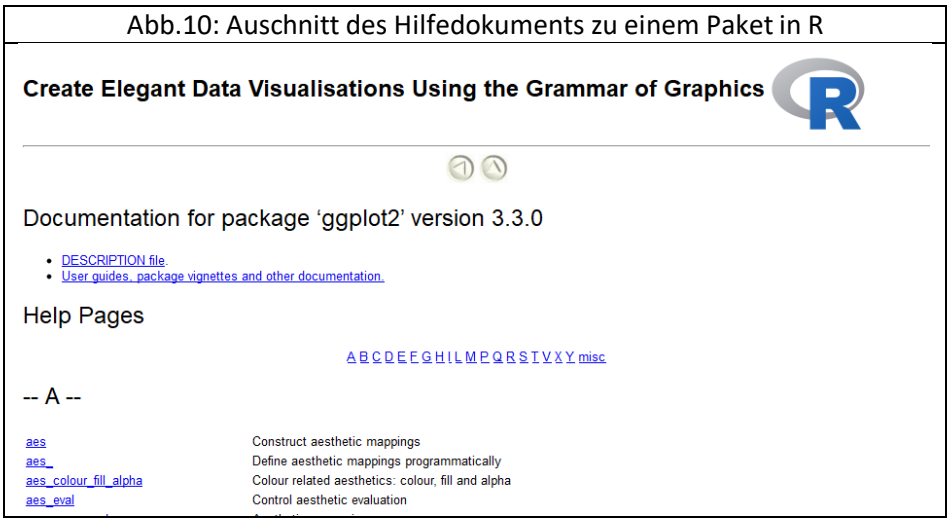


```
search: sum sum! summary cumsum cumsum! isnumeric VersionNumber issubnormal

  sum(f, itr)

  Sum the results of calling function f on each element of itr.

  The return type is Int for signed integers of less than system word size, and
  UInt for unsigned integers of less than system word size. For all other
  arguments, a common return type is found to which all arguments are promoted.

  Examples
  ==========

  julia> sum(abs2, [2; 3; 4])
  29

  Note the important difference between sum(A) and reduce(+, A) for arrays with
  small integer eltype:

  julia> sum(Int8[100, 28])
```

Hilfe zu einem Paket kann man nur in R über den Befehl „help(*Paketname*)“ aufrufen (Abb.9). In Julia gibt es für diese Funktion leider noch keinen Befehl.

Abb.9: Hilfe zu einem Paket in R rufen
`> help(package="ggplot2")`

Abb.10: Auschnitt des Hilfedokuments zu einem Paket in R



Sollte man den exakten Namen einer gesuchten Funktion nicht kennen, so kann man sowohl in R als auch in Julia mit dem Befehl „apropos()“ mit dem Stichwort im Argument nach Hilfe suchen (Abb.11).

Abb.11: Hilfe anhand von Stichworten suchen

In R

```
> apropos("sum")
 [1] ".colSums"                ".rowSums"                ".rs.callSummary"         ".rs.summarizeDir"
 [5] ".tryResumeInterrupt"     "colSums"                 "contr.sum"               "cumsum"
 [9] "format.summaryDefault"   "obj_sum"                 "print.summary.table"     "print.summary.warnings"
[13] "print.summaryDefault"    "rowsum"                  "rowsum.data.frame"       "rowsum.default"
[17] "rowSums"                 "stat_sum"                "stat_summary"            "stat_summary_2d"
[21] "stat_summary_bin"        "stat_summary_hex"        "stat_summary2d"          "StatSum"
[25] "StatSummary"             "StatSummary2d"           "StatSummaryBin"          "StatSummaryHex"
[29] "sum"                     "summarise"               "summarise_"              "summarise_all"
[33] "summarise_at"            "summarise_coord"         "summarise_each"          "summarise_each_"
[37] "summarise_if"            "summarise_layers"        "summarise_layout"        "summarize"
[41] "summarize_"              "summarize_all"           "summarize_at"            "summarize_each"
[45] "summarize_each_"         "summarize_if"            "summary"                 "Summary"
[49] "summary.aov"             "summary.connection"      "summary.data.frame"      "Summary.data.frame"
[53] "summary.Date"            "Summary.Date"            "summary.default"         "Summary.difftime"
[57] "summary.factor"          "Summary.factor"          "summary.glm"             "summary.lm"
[61] "summary.manova"          "summary.matrix"          "Summary.numeric_version" "Summary.ordered"
[65] "summary.POSIXct"         "Summary.POSIXct"         "summary.POSIXlt"         "Summary.POSIXlt"
[69] "summary.proc_time"       "summary.srcfile"         "summary.srcref"          "summary.stepfun"
[73] "summary.table"           "summary.warnings"        "summaryRprof"            "tbl_sum"
[77] "tbl_sum"                 "type_sum"                "type_sum"
```

In Julia

```
julia> apropos("sum")
Base.schedule
Base.digits
Base.checkbounds_indices
Base.readuntil
Base.fieldoffset
Base.sum
Base.IteratorSize
Base.release
```

Beenden: R wird über den Befehl „q()“ und Julia über „exit()“ in der Konsole beendet. Alternativ lassen sich beide Sprachen auch durch das Schließen des Programmfensters beenden. Zuvor erscheint die Frage, ob man den workspace, also alle erstellten Daten, im aktuellen Arbeitsverzeichnis speichern möchte.

Tab.1: grundlegende Elemente

Bedeutung	In R	In Julia
Schnellhilfe	• ?*Befehl,* help(*Befehl*)	• ?*Befehl* • apropos(“*Teil des Befehlsnamen*“)

	• apropos(“*Teil des Befehlsnamen*“)	
Beenden	q()	exit()
Kommentar	# *Kommentar*	# *Kommentar*
Zuweisungsoperator	<-	=

3.1.2. Zusatzpakete verwenden

R und Julia lassen sich über eigenständig entwickelte Zusatzpakete, mit denen man spezielle Fragestellungen lösen kann, modular erweitern. Diese müssen zunächst in der Konsole installiert werden. In R erfolgt dies über den Befehl „install.packages()“ und in Julia über den Befehl „Pkg.add()“ (Abb.12). In Julia ist es aber notwendig zuallererst das Paket „Pkg“ zu laden, bevor man irgendwelche Zusatzpakete installieren kann. In den beiden Programmiersprachen lassen sich die bereits installierten Pakete über „installed.packages()“ in R und über „Pkg.status()“ in Julia aufrufen (Abb.12).

Abb.12: Pakete installieren und bereits installierte Pakete auflisten

In R

```
> # Paket(e) installieren
> install.packages("psych")                    # 1 Paket installieren
Installing package into 'C:/Users/thuy0/Documents/R/win-library/3.6'
(as 'lib' is unspecified)
trying URL 'https://cran.rstudio.com/bin/windows/contrib/3.6/psych_1.9.12.31.zip'
Content type 'application/zip' length 3801000 bytes (3.6 MB)
downloaded 3.6 MB

package 'psych' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\thuy0\AppData\Local\Temp\RtmpqgoJ2G\downloaded_packages
> install.packages(pkgs=c("ggplot2", "dplyr", tidyr))      # mehrere Pakete gleichzeitig installieren
Error in install.packages : object 'tidyr' not found
>
> # installierte Pakete auflisten
> installed.packages()
             Package          LibPath                                    Version       Priority
abind        "abind"          "C:/Users/thuy0/Documents/R/win-library/3.6" "1.4-5"      NA
acepack      "acepack"        "C:/Users/thuy0/Documents/R/win-library/3.6" "1.4.1"      NA
ade4         "ade4"           "C:/Users/thuy0/Documents/R/win-library/3.6" "1.7-13"     NA
AER          "AER"            "C:/Users/thuy0/Documents/R/win-library/3.6" "1.2-8"      NA
agricolae    "agricolae"      "C:/Users/thuy0/Documents/R/win-library/3.6" "1.3-1"      NA
AlgDesign    "AlgDesign"      "C:/Users/thuy0/Documents/R/win-library/3.6" "1.2.0"      NA
arm          "arm"            "C:/Users/thuy0/Documents/R/win-library/3.6" "1.10-1"     NA
arules       "arules"         "C:/Users/thuy0/Documents/R/win-library/3.6" "1.6-4"      NA
arulesViz    "arulesViz"      "C:/Users/thuy0/Documents/R/win-library/3.6" "1.3-3"      NA
```

In Julia

```
julia> using Pkg

julia> Pkg.add("LinearAlgebra")
   Updating registry at `C:\Users\thuy0\.julia\registries\General`
   Updating git-repo `https://github.com/JuliaRegistries/General.git`
  Resolving package versions...
   Updating `C:\Users\thuy0\.julia\environments\v1.4\Project.toml`
 [no changes]
   Updating `C:\Users\thuy0\.julia\environments\v1.4\Manifest.toml`
 [no changes]

julia> Pkg.status()
Status `C:\Users\thuy0\.julia\environments\v1.4\Project.toml`
  [0825541b] ANOVA v0.1.0
  [c52e3926] Atom v0.12.18
  [324d7699] CategoricalArrays v0.8.1
  [a93c6f00] DataFrames v0.21.4
  [31c24e10] Distributions v0.23.5
  [da1fdf0e] FreqTables v0.4.0
```

Aber auch das Aktualisieren dieser Zusatzpakete ist möglich. In R wird dies über den Befehl „update.packages()“ aufgerufen und in Julia über „Pkg.update()“ (Abb.13). Um die installierten Pakete nun zu verwenden, werden sie in R über „library()“und in Julia über „using“ aufgerufen (Abb.13).

Abb.13: installierte Pakete aktualisieren und Pakete verwenden

In R

```
> # installiertes Paket aktualisieren
> update.packages("psych")
>
> # Pakete verwenden
> library(psych)

Attache Paket: 'psych'

The following objects are masked from 'package:ggplot2':

    %+%, alpha

Warning message:
Paket 'psych' wurde unter R Version 3.6.3 erstellt
```

In Julia

```
julia> Pkg.rm("LinearAlgebra")
   Updating `C:\Users\thuy0\.julia\environments\v1.4\Project.toml`
  [37e2e46d] - LinearAlgebra
   Updating `C:\Users\thuy0\.julia\environments\v1.4\Manifest.toml`
 [no changes]

julia> using LinearAlgebra
```

Tab2.: Pakete verwenden		
Bedeutung	In R	In Julia
Paket installieren	install.packages(*Paketname*)	Pkg.add(*Paketname*)
Installierte Pakete auflisten	installed.packages()	Pkg.status()
Installierte Paket aktualisieren	update.packages(“*Paketname*“)	Pkg.update(“*Paketname*“)
Paket entfernen	remove.packages(*Paketname*)	Pkg.rm(„*Paketname*“)
Paket laden	library(*Paketname*)	using *Paketname*

3.1.3. Datenstrukturen

Um mit R bzw. Julia arbeiten zu können, muss man sich zunächst ein Grundverständnis zu den verschiedenen Datenobjekten in diesen beiden Sprachen aufbauen. Diese Datenobjekte repräsentieren Informationen und können einzelne Werte, eine Menge von Werten, mehrere Variablen und Ergebnisse statistischer Analysen enthalten. In R bzw. in Julia berechnete Zwischenergebnisse sollten immer einem Datenobjekt zugewiesen werden, um bei weiterführenden Berechnungen immer auf diese zurückgreifen zu können. Dies hat den Vorteil, dass Berechnungen in der Konsole übersichtlich gestaltet werden können. Zudem wird dadurch der Rechenaufwand des Computers reduziert und komplizierte Berechnungen müssen nur einmal durchgeführt werden [8]. In R werden Zwischenergebnisse mit einem Pfeil „<-“ Datenobjekten zugewiesen und in Julia mit „=“. Sowohl in R als auch in Julia können Datenobjekte beliebig oft überschrieben werden, d.h. man kann einem Datenobjekt immer wieder neue Ergebnisse zuordnen. Julia hat im Vergleich zu R noch die nützliche Funktion bestimmte Zuweisungen kürzer zu gestalten, z.B. kann statt „x = x + 5“ auch die

folgende Schreibweise „x += 5“ verwendet werden. Auch bietet Julia die Möglichkeit auf das zuletzt ausgewertete Ergebnis zurückzugreifen, das unter dem Befehl „ans“ gespeichert ist (Abb.14).

Abb.14: Zuweisung an Datenobjekten

In R	In Julia

`> x <- 3`
`> x`
`[1] 3`
`> x <- 7`
`> x`
`[1] 7`
`> x <- x + 8`
`> x`
`[1] 15`
	`julia> (5+3)^2 / 4 - 6`
	`10.0`
	
	`julia>`
	
	`julia> ans`
	`10.0`
	
	`julia> x = 3`
	`3`
	
	`julia> x = 7`
	`7`
	
	`julia> x = x + 8`
	`15`
	
	`julia> x += 5`
	`20`

In der Programmiersprache unterscheidet man zwischen verschiedenen Datenobjekttypen. Diese sind im Wesentlichen Vektoren, Matrizen, Arrays, Faktoren, Tupel, Listen, Datensätze und Funktionen. Variablen mit Nominalskalenniveau werden als Faktoren gespeichert und Variablen mit Ordinalskalenniveau oder höher werden als Vektoren gespeichert, wobei ein Vektor eine Menge von Elementen darstellt. Matrizen sind Objekte mit mehreren Zeilen und Spalten, die Daten von nur einem Datenobjekttyp darstellen. Arrays sind mehrdimensionale Matrizen, d.h. dieser Datenobjekttyp besitzt neben den Zeilen und den Spalten noch mindestens eine weitere Dimension. Ähnlich zu den Matrizen sind die Datensätze oder auch Data Frames genannt. Diese entstehen durch die Kombination von verschiedenen Datenobjekttypen wie Faktoren und Variablen. In den Zeilen eines Datensatzes stehen die Beobachtungen und in den Spalten die erhobenen Variablen. Auch der Datenobjekttyp Liste kann aus mehreren Objekten bestehen, die unterschiedlich viele Elemente enthalten. Matrizen und Datensätze dagegen können nur Objekte enthalten, die alle dieselbe Anzahl an Elementen besitzen. Die verschiedenen Datenobjekttypen, die in R und Julia auftauchen, können aus der folgenden Tabelle (Tab.3) entnommen werden.

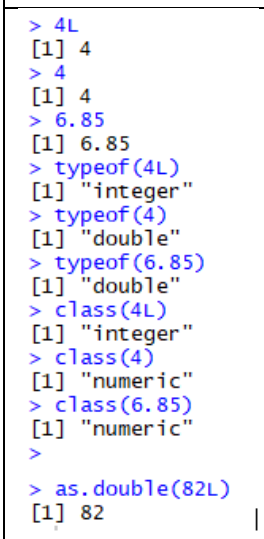
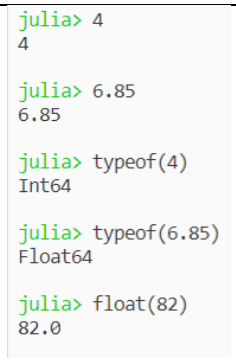
Tab.3: Datenobjekttypen und genauere Unterscheidungen

in R			
Beschreibung	Datenobjekttyp	exakter Datenobjekttyp	Beispiel
Ganze Zahlen	numeric	integer	5
Reelle zahlen	numeric	double	4,736
Zeichen und Zeichenketten	character		d Text
Array	array		
Matrix	matrix		$\begin{pmatrix} 5 & 4 \\ 2 & 7 \end{pmatrix}$
Vektor	numeric		
Datensatz	dataframe		
Liste	list		
Faktoren	factor		
Komplexe Zahlen	complex		2+3i
Logische Werte	logical		TRUE, FALSE (T,F)
Spezielle Typen: 1. Fehlende Werte 2. Leere Menge 3. Nicht definierte Werte 4. Unendlich	1. NA 2. NULL 3. NaN 4. +Inf, -Inf		

in Julia		
Beschreibung	exakter Datenobjekttyp	Beispiel
Ganze Zahlen	Int8 UInt8 Int16 Uint16 Int32 Uint32 Int64 Uint64 Int128 Uint128	5
Reelle Zahlen	Float16 Float32 Float64	4,736
Zeichen	character	d
Zeichenketten	string	Text
Array (mehr-dim. Array)	array	
Matrix (2-dim. Array)	array	$\begin{pmatrix} 5 & 4 \\ 2 & 7 \end{pmatrix}$
Vektor (1-dim. Array)	array	
Datensatz	dataframe	
Tupel	tuple	
Faktoren	categoriacal	
Komplexe Zahlen	complex	2+3im
Logische Werte	bool	true, false
Spezielle Typen: 1. Fehlende Werte 2. Nicht definierte Werte 3. Unendlich	1. missing 2. NaN 3. +Inf, -Inf	

In R kann der Datenobjekttyp eines Datenobjekts mit dem Befehl „class()“ abgefragt werden und der exakte Datenobjekttyp kann mit „typeof()“ abgefragt werden. In Julia kann man nur den exakten Datenobjekttyp über „typeof()“ abfragen (Abb.15).

Abb.15: Beispiel zur Abfrage ausgewählter Datenobjekttypen

In R	in Julia
	

3.2. Grundlagen

3.2.1. Rechnen mit Operationen und mathematischen Funktionen

Beide Programmiersprachen können als simplen Taschenrechner genutzt werden, denn die Verwendung der einfachen Arithmetik wie Operatoren (Abb.16), Funktionen (Abb.17) und Konstanten (Abb.18) ist in beiden Sprachen möglich. Es gilt dabei die PO-KLAPS-Regel (d.h. Potenzen vor Klammern vor Punkt- vor Strichrechnungen). Die Befehle für die grundlegenden Rechenoperationen wie Addition, Differenz, Multiplikation und Division, aber auch das Potenzieren und die Befehle für diverse Konstanten ist in beiden Sprachen gleich. Bei den arithmetischen Funktionen unterscheiden sich lediglich die Befehle für die Modulo-Division (in R: a%%b, in Julia: mod(a,b)), die Schreibweise für die Argumente der (kumulierten) Summen- und Produktfunktionen (sum(), prod(), cum(), cumprod()) und dem Befehl für das Bilden des Binomialkoeffizienten (in R: choose(n, k), in Julia: binomial(n, k)). Auffallend ist zudem noch, dass die Ergebnisse in Julia viel mehr Nachkommastellen anzeigen als in R.

Abb.16: grundlegende Rechenoperationen

In R	In Julia

`> 2+3`
`[1] 5`
`> 9-4`
`[1] 5`
`> 3*5`
`[1] 15`
`> 9/3`
`[1] 3`
`>`
`> 2^4`
`[1] 16`
`>`
`> (5+3)^2 / 4 - 6`
`[1] 10`
	`julia> 2+3`
	`5`
	
	`julia> 9-4`
	`5`
	
	`julia> 3*5`
	`15`
	
	`julia> 9/3`
	`3.0`
	
	`julia>`
	
	`julia> 2^4`
	`16`
	
	`julia>`
	
	`julia> (5+3)^2 / 4 - 6`
	`10.0`

Abb.17: arithmetische Funktionen

In R

```
> 9 %% 4
[1] 1
>
> sum(1,2,3,4)
[1] 10
> prod(1,2,3)
[1] 6
>
> cumsum(c(1,2,3,4))
[1]  1  3  6 10
> cumprod(c(2,4,3))
[1]  2  8 24
>
> sign(-4)
[1] -1
> sign(0)
[1] 0
> sign(8)
[1] 1
>
> abs(-5)
[1] 5
> abs(0)
[1] 0
> abs(9)
[1] 9
>
> sqrt(16)
[1] 4
>
> round(4.6254)
[1] 5
> round(4.6254, digits=2)
[1] 4.63
> floor(4.6254)
[1] 4
> ceiling(4.6254)
[1] 5
> trunc(4.6254)
[1] 4
> log(3)
[1] 1.098612
> log10(3)
[1] 0.4771213
> log(3, base = 4)
[1] 0.7924813
>
> exp(6)
[1] 403.4288
> exp(1)
[1] 2.718282
>
> sin(2)
[1] 0.9092974
> cos(5)
[1] 0.2836622
> tan(4)
[1] 1.157821
>
> factorial(3)
[1] 6
>
> choose(5,3)
[1] 10
```

In Julia

```
julia> mod(9,4)
1

julia> sum([1,2,3,4])
10

julia> prod([1,2,3])
6

julia> cumsum([1,2,3,4])
  1
  3
  6
 10

julia> cumprod([2,4,3])
3-element Array{Int64,1}:
  2
  8
 24

julia>
julia> sign(-4)
-1
julia> sign(0)
0

julia> sign(8)
1

julia>
julia> abs(-5)
5

julia> abs(0)
0

julia> abs(9)
9

julia> sqrt(16)
4.0

julia> round(4.6254)
5.0

julia> round(4.6254, digits=2)
4.63

julia> floor(4.6254)
4.0

julia> ceil(4.6254)

julia> trunc(4.6254)
4.0

julia> log(3)
1.0986122886681098

julia> log10(3)
0.47712125471966244

julia> exp(6)
403.4287934927351

julia> MathConstants.e
ℯ = 2.7182818284590...

julia>
julia> sin(2)
0.9092974268256817

julia> cos(5)
0.28366218546322625

julia> tan(4)
1.1578212823495777

julia> factorial(3)
6

julia> binomial(5,3)
10
```

Abb.18: Konstanten

In R

```
> pi
[1] 3.141593
>
> Inf
[1] Inf
> -Inf
[1] -Inf
>
> 0/0
[1] NaN
> NaN
[1] NaN
>
> NA
[1] NA
```

In Julia

```
julia> pi
π = 3.1415926535897...

julia> π
π = 3.1415926535897...

julia> Inf
Inf

julia> -Inf
-Inf

julia> 0/0
NaN

julia> NaN
NaN
```

Tab.4: Erste Schritte: Operatoren, arithmetische Funktionen, Konstanten

Bedeutung	In R	In Julia
Operatoren		
Operationen (komponentweise)	+,-,*,/	+,-,*,/
Potenzieren	^	^
Ganzzahlige Division	%/%	
Modulo Division (von a und b)	a%%b	mod(a,b)
Summenzeichen Σ	sum()	sum()
Produktzeichen Π	prod()	prod()
Kummulierte Summe/Partialsumme	cumsum()	cumsum()
Kumuliertes Produkt	cumprod()	cumprod()
Arithmetische Funktionen		
Vorzeichen	sign()	sign()
Absolutbetrag	abs()	abs()
Quadratwurzel	sqrt()	sqrt()
runden	round(*Zahl*, digits=*Anzahl der Dezimalstellen*)	round(*Zahl*, digits=*Anzahl der Dezimalstellen*)
Auf nächste Ganzzahl 1. abrunden 2. aufrunden 3. tranchieren	1. floor() 2. ceiling() 3. trunc()	1. floor() 2. ceil() 3. trunc()
1. Natürlicher Logarithmus 2. Logarithmus zur Basis 10 3. Logarithmus zur bel. Basis	1. log() 2. log10() 3. log(*Zahl*, base=*Basis*)	1. log() 2. log10()
1. Exponentialfunktion 2. Eulersche Zahl e	1. exp() 2. exp(1)	1. exp() 2. MathConstants.e
Trigonometr. Funktionen und deren Umkehrfunktionen	sin(), cos(), tan(), asin(), acos(), atan()	sin(), cos(), tan(), asin(), acos(), atan()
Fakultät	factorial(n)	factorial(n)
Binomialkoeffizient	choose(n,k)	binomial(n,k)
Konstanten		
Kreiszahl π	pi	pi
$-\infty, \infty$	-Inf, Inf	-Inf, Inf
Nicht definiert (z.B. 0/0)	NaN	NaN
Fehlende Werte	NA	missing

Leere Menge	NULL	

3.2.2. Zeichenketten

Zeichenketten tauchen zum Beispiel bei der Datenauswertung als Ausprägungen von kategorialen Variablen auf. In R werden sowohl einzelne Zeichen als auch Zeichenketten als Datenobjekt „character“ betrachtet. Dahingegen wird in Julia zwischen diesen beiden unterschieden. Einzelne Zeichen wie beispielsweise Buchstaben sind in Julia vom Datenobjekttyp „character“, während Zeichenketten vom Datenobjekttyp „string“ sind. In R werden die „character“-Objekte erzeugt, indem einzelne Zeichen oder Zeichenketten in Anführungszeichen (sowohl die einfachen als auch die doppelten Anführungszeichen sind verwendbar) geschrieben werden. In Julia dagegen werden die einzelnen Zeichen nur als „character“-Objekte anerkannt, wenn sie in einfachen Anführungszeichen geschrieben werden. Und für die Erzeugung von „string“-Objekten müssen die Zeichen bzw. Zeichenketten in doppelten Anführungszeichen geschrieben werden (Abb.19).

Abb.19: Zeichenketten

In R

```
> "a"
[1] "a"
> "Hallo"
[1] "Hallo"
> 'Hallo'
[1] "Hallo"
> "Das ist eine Zeichenkette"
[1] "Das ist eine Zeichenkette"
```

In Julia

```
julia> 'a'
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> "a"
"a"

julia> "Hallo"
"Hallo"

julia> "Das ist eine Zeichenkette"
"Das ist eine Zeichenkette"
```

Mit dem Befehl „length()“ wird in R die Anzahl der Zeichenketten ausgegeben. Dabei wird eine Zeichenkette als ein Element wahrgenommen. Um sich die Anzahl der Zeichen in einer Zeichenkette auszugeben, verwendet man in R den Befehl „nchar()“. Der Befehl „length()“ hat in der Julia-Sprache eine ganz andere Funktion. Sie gibt nämlich in dieser Sprache die Anzahl der Zeichen in einer Zeichenkette aus und ist deshalb mit dem Befehl „nchar()“ in R zu vergleichen. Ein Befehl zur Ausgabe der Zeichenkette als ein Element so wie in R existiert in Julia nicht. Sollte man ein Datenobjekt mit mehreren Zeichenketten haben, so kann man in R sich die Anzahl der Zeichenketten ausgeben lassen. In Julia ist dies leider noch nicht möglich (Abb.20).

Abb.20: Länge von Zeichenketten

In R

```
> a <- "Hallo"
> length(a)
[1] 1
> nchar(a)
[1] 5
> b <- "Das ist eine Zeichenkette"
> length(b)
[1] 1
> nchar(b)
[1] 25
```

In Julia

```
julia> a = "Hallo"
"Hallo"

julia> length(a)
5

julia> b = "Das ist eine Zeichenkette"
"Das ist eine Zeichenkette"

julia> length(b)
25
```

Um auf die einzelnen Komponenten einer Zeichenkette zu greifen, wird in R der Befehl „substr()“ genutzt. Der erste Argument ist die Zeichenkette, auf die zugegriffen werden soll. Die zwei letzten Argumente geben an, von welcher Stelle aus auf die Zeichenkette zugegriffen werden soll und ab welcher Stelle der Zugriff enden soll. In Julia wird der Zugriff auf die Zeichenketten über die eckigen Klammern gewährt. Auch hier kann angegeben werden, an welcher Stelle der Zugriff beginnt und an welcher er endet (Abb.21). Der Zugriff auf einzelne Komponenten über die eckige Klammern ist nicht nur bei Zeichenketten möglich, sondern auch bei Vektoren, Matrizen und Datensätzen schlägt sich diese Schreibweise durch (das kann man später in den nächsten Kapiteln sehen).

Abb.21: Zugriff auf Zeichenketten

In R

```
> a
[1] "Hallo"
> substr(a,2,2)
[1] "a"
> substr(a,2,4)
[1] "all"
> b
[1] "Das ist eine Zeichenkette"
> substr(a,2,2)
[1] "a"
> substr(a,2,4)
[1] "all"
```

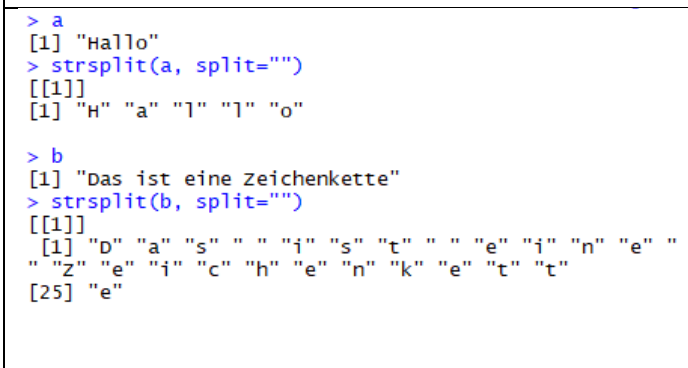
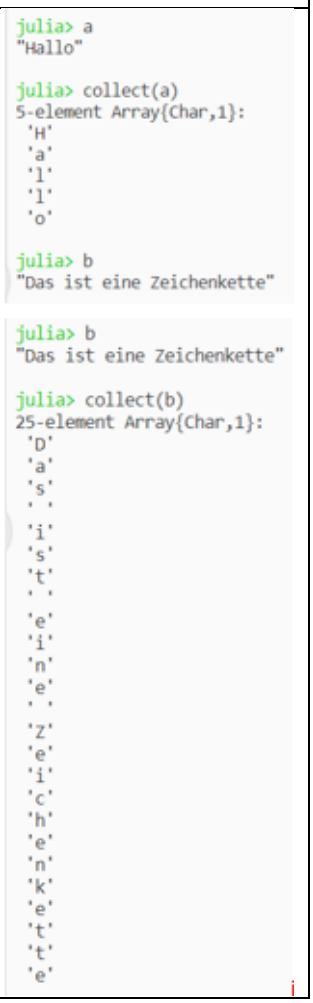
In Julia

```
julia> a
"Hallo"

julia> a[2]
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> a[2:4]
"all"

julia> a[4:end]
"lo"

julia> b
"Das ist eine Zeichenkette"

julia> b[2]
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> b[2:4]
"as "

julia> b[4:end]
" ist eine Zeichenkette"
```

Des Weiteren kann man Zeichenketten in seine einzelnen Komponenten zerlegen. In R geschieht dies mit dem Befehl „strsplit()“. Das erste Argument davon gibt an, welche Zeichenkette zerlegt werden soll und das zweite Argument gibt an, bei welchem Zeichen die Zerlegung durchgeführt werden soll (Abb.22). Auch kann mit diesem Befehl die Zeichenkette in einzelne Wörter zerlegt werden (Abb.23). Hierfür muss lediglich das zweite Argument angepasst werden. In Julia gibt es zwei Befehle zur Zerlegung von Zeichenketten. Für die Zerlegung in einzelne Zeichen, verwendet man den Befehl

„collect()“. Diesem Befehl muss nur die Zeichenkette übergeben werden, um eine Zerlegung durchzuführen (Abb.22). Und die Zerlegung der Zeichenkette in einzelne Wörter kann mit dem Befehl „split()“ erreicht werden (Abb.23). In R kann man nur mit einem Befehl bestimmen, wie eine Zeichenkette zerlegt werden soll, während man in Julia zwei Befehle kennen muss, je nachdem, ob man die Zeichenkette in seine einzelne Komponenten oder nur in einzelne Wörter zerlegen möchte.

Abb.22: Zeichenkette zerlegen (1)

In R	In Julia
	

```
> a
[1] "Hallo"
> strsplit(a, split="")
[[1]]
[1] "H" "a" "l" "l" "o"

> b
[1] "Das ist eine Zeichenkette"
> strsplit(b, split="")
[[1]]
 [1] "D" "a" "s" " " "i" "s" "t" " " "e" "i" "n" "e" "
" "Z" "e" "i" "c" "h" "e" "n" "k" "e" "t" "t"
[25] "e"
```

```
julia> a
"Hallo"

julia> collect(a)
5-element Array{Char,1}:
 'H'
 'a'
 'l'
 'l'
 'o'

julia> b
"Das ist eine Zeichenkette"

julia> b
"Das ist eine Zeichenkette"

julia> collect(b)
25-element Array{Char,1}:
 'D'
 'a'
 's'
 ' '
 'i'
 's'
 't'
 ' '
 'e'
 'i'
 'n'
 'e'
 ' '
 'Z'
 'e'
 'i'
 'c'
 'h'
 'e'
 'n'
 'k'
 'e'
 't'
 't'
 'e'
```

Abb.23: Zeichenkette zerlegen (2)

In R

```
> a
[1] "Hallo"
> strsplit(a, split=" ")
[[1]]
[1] "Hallo"

> b
[1] "Das ist eine Zeichenkette"
> strsplit(b, split=" ")
[[1]]
[1] "Das"          "ist"          "eine"         "Zeich
enkette"
```

In Julia

```
julia> a
"Hallo"

julia> split(a)
1-element Array{SubString{String},1}:
 "Hallo"

julia> b
"Das ist eine Zeichenkette"

julia> split(b)
4-element Array{SubString{String},1}:
 "Das"
 "ist"
 "eine"
 "Zeichenkette"

julia> split(b, " ")
4-element Array{SubString{String},1}:
 "Das"
 "ist"
 "eine"
 "Zeichenkette"

julia> c = "Das_ist_eine_Zeichenkette"
"Das_ist_eine_Zeichenkette"

julia> split(c, "_")
4-element Array{SubString{String},1}:
 "Das"
 "ist"
 "eine"
 "Zeichenkette"
```

Es besteht auch die Möglichkeit mehrere Zeichenketten miteinander zu verknüpfen. Mit dem Befehl „paste0()“ können in R zwei Zeichenketten ohne ein Trennungszeichen dazwischen verknüpft werden. Ähnliches Ergebnis kann mit dem Befehl „paste()“ erreicht werden. In diesem Befehl kann man zusätzlich angeben, mit welchem Trennungszeichen die Zeichenketten verknüpft werden. In Julia erfolgt die Verknüpfung über den Befehl „string()“. Das erste und dritte Argument stellt dabei die Zeichenketten dar, die verknüpft werden sollen. Und mit dem zweiten Argument kann man bestimmen, mit welchem Trennungszeichen verknüpft werden soll (Abb.24). Die Argumente in dem Julia-Befehl sind dabei so angeordnet, wie man sich die Zusammensetzung der Zeichenkette vorstellt.

Abb.24: Zeichenketten verknüpfen

In R

```
> d <- "Haus"
> e <- "Tier"
> paste0(d, e)
[1] "HausTier"
> paste(d, e, sep="")
[1] "HausTier"
> paste(d, e)
[1] "Haus Tier"
> paste(d, e, sep=" ")
[1] "Haus Tier"
> paste(d, e, sep="_")
[1] "Haus_Tier"
```

In Julia

```
julia> d = "Haus"
"Haus"

julia> e = "Tier"
"Tier"

julia> d * e
"HausTier"

julia> string(d, "", e)
"HausTier"

julia> string(d, " ", e)
"Haus Tier"

julia> string(d, "_", e)
"Haus_Tier"
```

Möchte man in Zeichenketten nach einem bestimmten Muster suchen, so kann dies in R mit dem Befehl „grep()“ erreicht werden. Hierfür müssen lediglich das gesuchte Muster und die Zeichenkette, die untersucht werden soll, übergeben werden. In Julia gibt es mehrere Befehle zur Durchführung dieser Aufgabe. Zum einem kann man mit dem Befehl „findfirst()“ die Stelle angeben lassen, an dem der erste Treffer erfolgt ist. Man kann aber auch mit dem Befehl „findnext()“ bestimmen, ab welcher Stelle in der Zeichenkette die Mustersuche beginnen soll. Dieser Befehl liefert aber auch nur die Stelle des ersten Treffers. Um sich alle Treffer ausgeben zu lassen, verwendet man den Befehl „findall()“. In Julia kann man sich insgesamt viel mehr Trefferstellen ausgeben lassen als in R, denn in R kann immer nur die erste Trefferstelle ausgegeben werden (Abb.25).

Abb.25: Trefferstelle bei Mustersuche

In R	In Julia

`> f <- "Banane"`
`> grep("a", f)`
`[1] 1`
`> grep("an", f)`
`[1] 1`
	`julia> f = "Banane"`
	`"Banane"`
	
	`julia> findfirst("a", f)`
	`2:2`
	
	`julia> findfirst("an",f)`
	`2:3`
	
	`julia> findfirst("j",f)`
	
	`julia> findnext("a", f, 3)`
	`4:4`
	
	`julia> findnext("an", f, 3)`
	`4:5`
	
	`julia> findall("a", f)`
	`2-element Array{UnitRange{Int64},1}:`
	` 2:2`
	` 4:4`
	
	`julia> findall("an", f)`
	`2-element Array{UnitRange{Int64},1}:`
	` 2:3`
	` 4:5`

Möchte man generell nur wissen, ob überhaupt ein Treffer bei der Mustersuche gelandet wurde, so benutzt man in R den Befehl „grepl()“ und in Julia den Befehl „occursin()“ (Abb.26).

Abb.26: Treffererfolg bei Mustersuche

In R	In Julia

`> grepl("e", f)`
`[1] TRUE`
`> grepl("j", f)`
`[1] FALSE`
	`julia> occursin("e", f)`
	`true`
	
	`julia> occursin("j", f)`
	`false`

Bestimmte Zeichen in einer Zeichenkette können in R mit dem Befehl „gsub()“ und „sub()“ durch ein anderes Muster ersetzt werden. Dabei wird zunächst nach dem bestimmten Muster gesucht und anschließend werden die Treffer durch das neue Muster ersetzt. Mit „gsub()“ werden alle Treffer durch das neue Muster ersetzt und mit „sub()“ wird nur der erster Treffer ersetzt. In Julia erfolgt die Musterersetzung über den Befehl „replace()“. Bei diesem Befehl kann man sogar entscheiden, wie viele Treffer durch das neue Muster ersetzt werden sollen. Bei dieser Aufgabe punktet definitiv Julia, denn die Sprache kommt mit nur einem Befehl aus im Vergleich zur R (Abb.27).

Abb.27: Mustersuche und Ersetzung

In R

```
> f
[1] "Banane"
> gsub("a", "e", f)
[1] "Benene"
> sub("a", "e", f)
[1] "Benane"
> b
[1] "Das ist eine Zeichenkette"
> gsub("e", "o", b)
[1] "Das ist oino Zoichonkotto"
> sub("e", "o", b)
[1] "Das ist oine Zeichenkette"
```

In Julia

```
julia> f
"Banane"

julia> replace(f, "a" => "e")
"Benene"

julia> replace(f, "a" => "e", count=1)
"Benane"

julia> b
"Das ist eine Zeichenkette"

julia> replace(b, "e" => "o")
"Das ist oino Zoichonkotto"

julia> replace(b, "e" => "o", count=1)
"Das ist oine Zeichenkette"

julia> replace(b, "e" => "o", count=3)
"Das ist oino Zoichenkette"
```

Auch ist das Umwandeln von Zeichenketten, die nur aus numerischen Elementen bestehen, in Zahlen möglich und das in beiden Sprachen. In R erfolgt dies mit den Befehlen „as.integer()“ und „as.numeric()“, je nachdem ob die Ausgabe in einer ganzen oder reellen Zahl erwünscht ist. Gleiches Ergebnis erhält man in Julia mit dem Befehl „parse()“. Dabei wird im ersten Argument angegeben, was für einen exakten Datentyp man erhalten möchte und im zweiten Argument gibt man die Zeichenkette an, die man umwandeln möchte. Das Umwandeln von reellen Zahlen in ganze Zahlen ist nur in R möglich. In Julia erhält man dagegen eine Fehlermeldung (Abb.28).

Abb.28: Zeichenketten in Zahlen umwandeln

In R

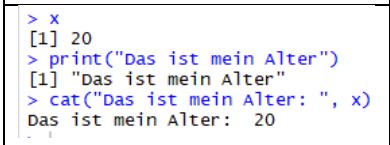
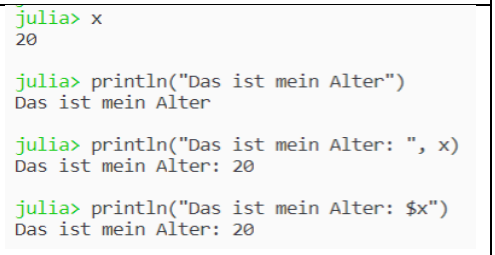
```
> as.integer("15")
[1] 15
> as.integer("17.8")
[1] 17
> as.numeric("441")
[1] 441
> as.numeric("45.6")
[1] 45.6
```

In Julia

```
julia> parse(Int64, "15")
15

julia> parse(Int64, "17.8")
ERROR: ArgumentError: invalid base 10 digit '.' in "17.8"
Stacktrace:
 [1] tryparse_internal(::Type{Int64}, ::String, ::Int64, ::Int64, ::Int64, ::Bool)
 at .\parse.jl:132
 [2] parse(::Type{Int64}, ::String; base::Nothing) at .\parse.jl:238
 [3] parse(::Type{Int64}, ::String) at .\parse.jl:238
 [4] top-level scope at none:0

julia> parse(Float64, "441")
441.0

julia> parse(Float64, "45.6")
45.6
```

Die Ausgabe von Zeichenketten erfolgt in R über den Befehl „print()“. Möchte man ein numerisches Ergebnis und einen dazu entsprechenden Text ausgeben, muss man den Befehl „cat()“ benutzen. In Julia kann dies mit nur einem Befehl, nämlich „println()“, erfolgen. Mit diesem Befehl kann in Julia ein Text und ein numerischer Wert gleichzeitig ausgegeben werden (Abb.29).

Abb.29: Zeichenketten ausgeben

In R	In Julia
	

Tab.5: Zeichen und Zeichenketten		
Bedeutung	In R	In Julia
Zeichen	*“x“*	‘x‘
Zeichenkette	*“Zeichenkette“*	*“Zeichenkette“*
Zugriff auf Zeichenkette (i-te Stelle)	*“Zeichenkette“*[i]	*“Zeichenkette“*[i]
1. Anzahl der Zeichen in einer Zeichenkette **2.** Anzahl der Elemente in einer Zeichenkette	1. nchar(*“Zeichenkette“*) 2. length(*“Zeichenkette“*)	1.1. length(*“Zeichenkette“*) 1.2. endof(*“Zeichenkette“*)) 1.3. sizeof(“Zeichenkette“[i])
Zugriff auf Zeichenketten	substr(*Zeichenkette*, *Stelle*)	*Zeichenkette*[*Stelle*]
Zeichenkette zerlegen	strsplit(*Zeichenkette*, split=“*Trennzeichen*“)	• collect(*Zeichenkette*) • split(*Zeichenkette*, “*Trennzeichen*“)
Zeichenketten verknüpfen	• paste(*Zeichenkette*, *Datenobjekt*, sep=“*Verbindungszeichen*“, collapse) • paste0() (Abk. für paste(, sep=““))	• *“Zeichenkette1“ * “ Zeichenkette2“* • string(*“Zeichenkette1“*, *“Trennzeichen“*, *“Zeichenkette2“*)
Muster-Suche 1. Trefferstelle **2.** Treffer gelandet?	3. grep(*“Muster“*, *Zeichenkette*) 4. grepl(*“Muster“*, *Zeichenkette*)	1. findfirst(*“Muster“*, *“Zeichenkette”*), findnext(*“Muster“*, *“Zeichenkette”*), findall(*“Muster“*, *“Zeichenkette"*) 2. occursin(*“Muster“*, *Zeichenkette*),

Muster-Suche und Ersetzung 1. Nur der erste Treffer 2. Alle Treffer	1. sub("$Muster_{alt}$", "$Muster_{neu}$", *Zeichenkette*) 2. gsub("*Muster*", " ", *Zeichenkette*)	1. replace(Zeichenkette, "$Muster_{alt}$" => "$Muster_{neu}$", count=1) 2. replace(Zeichenkette, "$Muster_{alt}$" => "$Muster_{neu}$")
Zeichenketten in Zahlen umwandeln	• as.integer(„15“) • as.numeric(„15“)	• parse(Int64, „15“) • parse(Float64, „15“)
Zeichenketten ausgeben	print(), cat()	println()

3.2.3. Vektoren, Folgen und Matrizen

Zur Vektorerzeugung in R werden die Vektoreinträge in „c()“ geschrieben. Einzelne Elemente werden dabei mit einem Komma getrennt. Der Vektor wird anschließend in einer Zeile ausgegeben. In Julia werden Vektoren mit den eckigen Klammern erstellt. Der Output hierbei wird in einer Spalte ausgegeben. In beiden Sprachen ist es möglich, ein Vektor aus verschiedenen Datenobjekttypen zu erstellen (Abb.30).

Abb.30: Vektoren erstellen

In R

```
> c(1,5,4,9)
[1] 1 5 4 9
> c('d', 'y','e')
[1] "d" "y" "e"
> c("Apfel", "Banane", "Melone")
[1] "Apfel"  "Banane" "Melone"
> c("Melone", 5, NA, "d")
[1] "Melone" "5"      NA       "d"
```

In Julia

```
julia> [1,5,4,9]
4-element Array{Int64,1}:
 1
 5
 4
 9

julia> ['d', 'y', 'e']
3-element Array{Char,1}:
 'd'
 'y'
 'e'

julia> ["Apfel", "Banane", "Melone"]
3-element Array{String,1}:
 "Apfel"
 "Banane"
 "Melone"

julia> ["Melone", 5, missing, 'd']
4-element Array{Any,1}:
  "Melone"
 5
  missing
  'd'
```

Das Erstellen von Folgen, also sequentielle Vektoren, kann in R mit „a:b“ erfolgen, wobei a den Anfangswert und b den Endwert darstellt. Mit dieser Schreibweise kann eine Folge mit der Standardschrittweite +1 und sogar -1 erzeugt werden. Möchte man Folgen mit einer anderen Schrittweite erzeugen, so verwendet man in R den Befehl „seq()“. Mit diesem Befehl lässt sich neben der Schrittweite auch die Länge der Folge festlegen. In Julia kann auch mit dem Ausdruck „a:b“ eine Folge erzeugt werden, jedoch ist der Output nicht so, wie man ihn in R kennt. Außerdem kann man mit dieser Schreibweise in Julia keine absteigende Folge erstellen. Für einen schöneren Output verwendet

man den Befehl „collect()“. Bei diesem Befehl kann man ebenfalls die Schrittweite und die Länge der zu erzeugenden Folge bestimmen. Zusätzlich ist es möglich, einen Wert anzugeben, bis zu welcher die Folge erzeugt werden soll (Abb.31).

Abb.31: Folgen erstellen

In R

```
> # mit Standardschrittweite +1 und -1
> 2:8
[1] 2 3 4 5 6 7 8
> 6:2
[1] 6 5 4 3 2
> seq(from=2, to=8)
[1] 2 3 4 5 6 7 8
> seq(from=8, to=2)
[1] 8 7 6 5 4 3 2
>
> # mit verschiedenen Schrittweiten
> seq(from=2, to=10, by=3)
[1] 2 5 8
> seq(from=8, to=1, by=-2)
[1] 8 6 4 2
>
> # mit vorgegebener Vektorlänge (Abbruchstelle nicht
 vorhanden)
> seq(from=1, by=3, length=4)
[1]  1  4  7 10
```

In Julia

```
julia> # mit Standardschrittweite +1 (-1 funktioniert nicht)

julia> 2:8
2:8

julia> 6:2
6:5

julia> collect(2:8)
7-element Array{Int64,1}:
 2
 3
 4
 5
 6
 7
 8

julia> collect(6:2)
0-element Array{Int64,1}

julia> # mit verschiedenen Schrittweiten

julia> collect(2:3:10)
3-element Array{Int64,1}:
 2
 5
 8

julia> collect(8:-2:1)
4-element Array{Int64,1}:
 8
 6
 4
 2

julia> # mit vorgegebener Vektorlänge oder Abbruchstelle

julia> collect(range(1, step=3, length=4))
4-element Array{Int64,1}:
  1
  4
  7
 10

julia> collect(range(1, step=3, stop=9))
3-element Array{Int64,1}:
 1
 4
 7
```

In beiden Sprachen kann man sich die Länge eines Vektors mit dem Befehl „length()“ ausgeben lassen Abb.32).

Abb.32: Vektorlänge berechnen

In R

```
> g <- c("Melone", 5, NA, "d")
> length(g)
[1] 4
```

In Julia

```
julia> g = ["Melone", 5, missing, 'd']
4-element Array{Any,1}:
  "Melone"
 5
  missing
  'd'

julia> length(g)
4
```

Der Zugriff auf Vektoreinträge kann über die eckigen Klammern erreicht werden (Abb.33).

Abb.33: Vektorzugriff

In R

```
> g
[1] "Melone" "5"       NA        "d"
> g[3]
[1] NA
> g[1:3]
[1] "Melone" "5"       NA
```

In Julia

```
julia> g
4-element Array{Any,1}:
  "Melone"
 5
  missing
  'd'

julia> g[3]
missing

julia> g[1:3]
3-element Array{Any,1}:
  "Melone"
 5
  missing
```

Die Vektoreinträge können mit dem Befehl „sort()“ alphabetisch oder in der Zahlenfolge aufsteigend sortiert werden. In R lassen sich auch Vektoren mit verschiedenen Datenobjekttypen sortieren. Enthält der Vektor fehlende Werte, so wird er bei der Sortierung einfach weggelassen. In Julia dagegen lassen sich nur Vektoren mit gleichen Datenobjekttypen sortieren (Abb34).

Abb.34: Vektor sortieren

In R

```
> h <- c(9, 1, 5)
> sort(h)
[1] 1 5 9
>
> i <- c("d", "y", "e")
> sort(i)
[1] "d" "e" "y"
>
> j <- c("Banane", "Melone", "Apfel")
> sort(j)
[1] "Apfel"  "Banane" "Melone"
> g
[1] "Melone" "5"       NA        "d"
> sort(g)
[1] "5"       "d"       "Melone"
```

In Julia

```
julia> h = [9, 1, 5]
3-element Array{Int64,1}:
 9
 1
 5

julia> sort(h)
3-element Array{Int64,1}:
 1
 5
 9

julia> i = ['d', 'y', 'e']
3-element Array{Char,1}:
 'd'
 'y'
 'e'

julia> sort(i)
3-element Array{Char,1}:
 'd'
 'e'
 'y'

julia> j = ["Banane", "Melone", "Apfel"]
3-element Array{String,1}:
 "Banane"
 "Melone"
 "Apfel"
```

In R kann man einem Vektor beliebige neue Vektoreinträge oder andere Vektoren anhängen, indem man den Befehl „c()“ verwendet (Abb.35+36). In Julia können nur Vektoreinträge des gleichen

Datenobjekttyps mit dem Befehl „push!()“ angehängt werden (Abb.35). Und mehrere Vektoren können nur mit dem Befehl „append!()“ verknüpft werden, wenn alle dieselbe Datenstruktur besitzen. Besteht der Vektor zum Beispiel nur aus „character“-Objekten, so können diesem Vektor auch nur weitere „character“-Objekte angehängt werden (Abb.36).

Abb.35: neuen Vektoreintrag anhängen

In R

```
> h
[1] 9 1 5
> c(h, 4)
[1] 9 1 5 4
> c(h, "Kiwi")
[1] "9"    "1"    "5"    "Kiwi"
>
> j
[1] "Banane" "Melone" "Apfel"
> c(j, 5)
[1] "Banane" "Melone" "Apfel"  "5"
> c(j, "Kiwi")
[1] "Banane" "Melone" "Apfel"  "Kiwi"
>
> g
[1] "Melone" "5"      NA       "d"
> c(g, 7)
[1] "Melone" "5"      NA       "d"
> c(g, "Kiwi")
[1] "Melone" "5"      NA       "d"
```

In Julia

```
julia> h
3-element Array{Int64,1}:
 9
 1
 5

julia> push!(h, 4)
4-element Array{Int64,1}:
 9
 1
 5
 4

julia> push!(h, "Kiwi")
ERROR: MethodError: Cannot `convert` an object of type String to an obj
 Int64
Closest candidates are:
  convert(::Type{T}, ::T) where T<:Number at number.jl:6
  convert(::Type{T}, ::Number) where T<:Number at number.jl:7
  convert(::Type{T}, ::Ptr) where T<:Integer at pointer.jl:23
  ...
Stacktrace:
 [1] push!(::Array{Int64,1}, ::String) at .\array.jl:913
 [2] top-level scope at none:0

julia> j
3-element Array{String,1}:
 "Banane"
 "Melone"
 "Apfel"
```

```
julia> push!(j, 5)
ERROR: MethodError: Cannot `convert` an object of type Int64 to an object
String
Closest candidates are:
  convert(::Type{T}, ::T) where T<:AbstractString at strings/basic.jl:209
  convert(::Type{T}, ::AbstractString) where T<:AbstractString at strings/
:210
  convert(::Type{T}, ::T) where T at essentials.jl:171
Stacktrace:
 [1] push!(::Array{String,1}, ::Int64) at .\array.jl:913
 [2] top-level scope at none:0

julia> push!(j, "Kiwi")
4-element Array{String,1}:
 "Banane"
 "Melone"
 "Apfel"
 "Kiwi"

julia> g
4-element Array{Any,1}:
  "Melone"
 5
  missing
  'd'
```

```
julia> push!(g, 7)
5-element Array{Any,1}:
  "Melone"
 5
  missing
  'd'
 7

julia> push!(g, "Kiwi")
6-element Array{Any,1}:
  "Melone"
 5
  missing
  'd'
 7
  "Kiwi"
```

Abb.36: mehrere Vektoren verknüpfen

In R

```
> h
[1] 9 1 5
> j
[1] "Banane" "Melone" "Apfel"
> g
[1] "Melone" "5"      NA       "d"
> c(h, c(2,8,4))
[1] 9 1 5 2 8 4
> c(j, c("Kiwi", "Traube"))
[1] "Banane" "Melone" "Apfel"  "Kiwi"   "Traube"
> c(g, g)
[1] "Melone" "5"      NA       "d"      "Melone"
[6] "5"      NA       "d"
```

In Julia

```
julia> h
4-element Array{Int64,1}:
 9
 1
 5
 4

julia> j
4-element Array{String,1}:
 "Banane"
 "Melone"
 "Apfel"
 "Kiwi"

julia> g
6-element Array{Any,1}:
  "Melone"
 5
  missing
  'd'
 7
  "Kiwi"

julia> append!(h,[2,8,4])
7-element Array{Int64,1}:
 9
 1
 5
 4
 2
 8
 4

julia> append!(j, ["Kiwi", "Traube"])
6-element Array{String,1}:
 "Banane"
 "Melone"
 "Apfel"
 "Kiwi"
 "Kiwi"
 "Traube"

julia> append!(g,g)
12-element Array{Any,1}:
  "Melone"
 5
  missing
  'd'
 7
  "Kiwi"
  "Melone"
 5
  missing
  'd'
 7
  "Kiwi"
```

Einem Vektor in R kann ein Element über die eckigen Klammern entfernt werden. So kann man zum Beispiel mit dem Ausdruck g[-i] das i-te Element des Vektors g entfernen. Nach jeder Entfernung erhält man die übrig gebliebenen Vektorelemente als Output. Um diesen gekürzten Vektor weiter zu behandeln, muss man ihn wieder neu zuweisen. In Julia dagegen gibt es mehrere Befehle zur Entfernung von Vektorelementen. Mit „popfirst!()“ kann das erste Vektorelement entfernt werden, mit „splice!()“ das i-te Element und mit „pop!()“ das letzte Element. Nach jeder Entfernung erhält man als Output das entfernte Element. In Julia werden die gekürzten Vektoren automatisch dem altem Objektnamen zugewiesen (Abb.37).

Abb.37: Vektorelement entfernen

In R

```
> g
[1] "Melone" "5"       NA        "d"
> g[-1]
[1] "5" NA  "d"
> g[-2]
[1] "Melone" NA        "d"
> g[-length(g)]
[1] "Melone" "5"       NA
```

In Julia

```
julia> g
4-element Array{Any,1}:
  "Melone"
 5
  missing
  'd'

julia> popfirst!(g)
"Melone"

julia> g
3-element Array{Any,1}:
 5
  missing
  'd'

julia> splice!(g, 2)
missing

julia> g
2-element Array{Any,1}:
 5
  'd'

julia> pop!(g)
'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)

julia> g
1-element Array{Any,1}:
 5
```

Datenobjekte können in R mit dem Befehl „rep()“ repliziert werden. Dabei kann mit dem Argument „times“ festgelegt werden, wie oft das Datenobjekt repliziert werden soll und mit dem Argument „each“ kann festgelegt werden, wie oft jedes Element nacheinander repliziert werden soll. In Julia werden die Datenobjekte mit dem Befehl „repeat()“ repliziert. Das Argument „outer“ ist mit dem Argument „times“ und „inner“ mit „each“ aus R vergleichbar. Zudem können mit „repeat()“ keine einfachen numerischen Objekte repliziert werden (Abb.38).

Abb.38: Datenobjekte replizieren

In R

```
> rep(6, times=4)
[1] 6 6 6 6
> rep(c(2,7), times=3)
[1] 2 7 2 7 2 7
>
> a
[1] "Hallo"
> rep(a, times=3)
[1] "Hallo" "Hallo" "Hallo"
>
> j
[1] "Banane" "Melone" "Apfel"
> rep(j, times=2)
[1] "Banane" "Melone" "Apfel"  "Banane" "Melone"
[6] "Apfel"
>
> rep(c(2,7), each=2)
[1] 2 2 7 7
```

In Julia

```
julia> repeat(6, 4)
ERROR: MethodError: no method matching repeat(::Int64, ::Int64)
Closest candidates are:
  repeat(::AbstractArray{T,1} where T, ::Integer) at abstractarraymath.jl:290
  repeat(::Union{AbstractArray{T,2}, AbstractArray{T,1}} where T, ::Integer) at ab
stractarraymath.jl:276
  repeat(::Union{AbstractArray{T,2}, AbstractArray{T,1}} where T, ::Integer, ::Int
eger) at abstractarraymath.jl:276
  ...
Stacktrace:
 [1] top-level scope at none:0

julia> repeat([2,7], outer=3)
6-element Array{Int64,1}:
 2
 7
 2
 7
 2
 7

julia> a
"Hallo"

julia> repeat(a, 3)
"HalloHalloHallo"

julia> j
6-element Array{String,1}:
 "Banane"
 "Melone"
 "Apfel"
 "Kiwi"
 "Kiwi"
 "Traube"

julia> repeat(j, outer=2)
12-element Array{String,1}:
 "Banane"
 "Melone"
 "Apfel"
 "Kiwi"
 "Kiwi"
 "Traube"
 "Banane"
 "Melone"
 "Apfel"
 "Kiwi"
 "Kiwi"
 "Traube"

julia> repeat([2,7], inner=2)
4-element Array{Int64,1}:
 2
 2
 7
 7
```

Die Erstellung von Matrizen erfolgt in R mit dem Befehl „matrix()“. In diesem Befehl kann man festlegen, wie viele Zeilen und Spalten die Matrix enthalten soll und wie die Matrixelemente angeordnet sind. In

Julia können Matrizen wie bei den Vektoren mit eckigen Klammern erstellt werden. Über das Semikolon „;“ werden die Matrixzeilen festgelegt (Abb.39).

Abb.39: Matrix erstellen

In R

```
> matrix(c(1,2,3,4,5,6), nrow=2, ncol=3)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> matrix(c(1,2,3,4,5,6), nrow=3, ncol=2)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> matrix(c(1,2,3,4,5,6), nrow=3, ncol=2, byrow=TRUE)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

In Julia

```
julia> [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> [1 2; 3 4; 5 6]
3×2 Array{Int64,2}:
 1  2
 3  4
 5  6
```

In R erhält man die Matrixdimension über den Befehl „dim()“. Und aus diesem entstandenen Output kann wieder auf die Zeilen- oder Spaltenanzahl der Matrix zugegriffen werden. Unabhängig von der Dimension kann man auch über die Befehle „nrow()“ und „ncol()“ an die Zeilen- bzw. Spaltenzahl kommen. Mit dem Befehl „size()“ kann in Julia die Matrixdimesion, die Zeilen- oder die Spaltenanzahl ausgegeben werden (Abb.40).

Abb.40: Matrixdimension, Zeilen- und Spaltenanzahl

In R

```
> k <- matrix(c(1,2,3,4,5,6), nrow=2, ncol=3)
>
> # Matrixdimension
> dim(k)
[1] 2 3
>
> # Zeilenanzahl
> dim(k)[1]
[1] 2
> nrow(k)
[1] 2
>
> # Spaltenzahl
> dim(k)[2]
[1] 3
> ncol(k)
[1] 3
```

In Julia

```
julia> k = [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> size(k)
(2, 3)

julia> size(k)[1]
2

julia> size(k, 1)
2

julia> size(k)[2]
3

julia> size(k, 2)
3
```

Der Matrixzugriff erfolgt in beiden Sprachen über die eckigen Klammern. Soll nur die i-te Matrixzeile ausgegeben werden, so gebe in R den Ausdruck „A[i,]“ ein, wobei A die Matrix darstellt. In Julia muss man hierfür den Ausdruck „A[i,:]“ verwenden. Für die Matrixspalten gilt dies analog (Abb.41).

Abb.41: Matrixzugriff

In R

```
> k
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> k[1,3]
[1] 5
> k[1:2, 2:3]
     [,1] [,2]
[1,]    3    5
[2,]    4    6
> k[2,]
[1] 2 4 6
> k[,1]
[1] 1 2
```

In Julia

```
julia> k
2x3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> k[1,3]
3

julia> k[1:2, 2:3]
2x2 Array{Int64,2}:
 2  3
 5  6

julia> k[2,:]
3-element Array{Int64,1}:
 4
 5
 6

julia> k[:,1]
2-element Array{Int64,1}:
 1
 4
```

Für die speziellen Matrizen wie Einheitsmatrix, Nullmatrix und Einsmatrix gibt es in R keine eigenen Befehle. In R kann die Einheitsmatrix mit dem Befehl „diag()“ erstellt werden. Die Null- und die Einsmatrix wird über den üblichen Befehl „matrix()“ erzeugt. In Julia dagegen gibt es für diese Matrizen eigene Befehle. So wird eine Einheitsmatrix über den Befehl „Matrix{}()“ erstellt. Der Ausdruck in der geschweiften Klammer legt dabei den Datenobjekttyp der Matrixelemente fest und über die Ausdrücke in den runden Klammern werden die Zeilen- und Spaltenzahl der Matrix festgelegt. Die Nullmatrix kann mit „zeros()“ und die Einsmatrix mit „ones()“ erstellt werden (Abb.42).

Abb.42: spezielle Matrizen

In R

```
> diag(1, nrow=3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
> matrix(0, nrow=2, ncol=5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
> matrix(1, nrow=3, ncol=2)
     [,1] [,2]
[1,]    1    1
[2,]    1    1
[3,]    1    1
>
```

In Julia

```
julia> Matrix{Int}(I,2,2)
2x2 Array{Int64,2}:
 1  0
 0  1

julia> zeros(2,5)
2x5 Array{Float64,2}:
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0

julia> ones(3,2)
3x2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0
 1.0  1.0
```

Das Transponieren von Matrizen erfolgt in R über den Befehl „t()“ und in Julia über „Transpose()“. Alternativ kann in Julia auch durch „ ‘ “ eine Matrix transponiert werden (Abb.43).

Abb.43: Matrix transponieren

In R

```
> k
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> t(k)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

In Julia

```
julia> k
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> Transpose(k)
3×2 Transpose{Int64,Array{Int64,2}}:
 1  4
 2  5
 3  6

julia> k'
3×2 Adjoint{Int64,Array{Int64,2}}:
 1  4
 2  5
 3  6
```

Die allgemeine Matrixmultiplikation wird in R mit „%*%“ durchgeführt und in Julia mit „*“. Die komponentenweise Matrixmultiplikation dagegen wird in R mit „*“ durchgeführt und in Julia mit „ .*“ (Abb.44).

Abb.44: Matrixmultiplikation

In R

```
> k
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> l <- t(k)
> k %*% l
     [,1] [,2]
[1,]   35   44
[2,]   44   56
> k * k
     [,1] [,2] [,3]
[1,]    1    9   25
[2,]    4   16   36
```

In Julia

```
julia> k
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> l = Transpose(k)
3×2 Transpose{Int64,Array{Int64,2}}:
 1  4
 2  5
 3  6

julia> k * l
2×2 Array{Int64,2}:
 14  32
 32  77

julia> k .* k
2×3 Array{Int64,2}:
  1   4   9
 16  25  36
```

Mit dem Befehl „det()“ lassen sich in beiden Sprachen die Determinante einer Matrix berechnen (Abb.45).

Abb.45: Determinante einer Matrix

In R

```
> m <- matrix(c(1,2,4,5), nrow=2, ncol=2)
> det(m)
[1] -3
```

In Julia

```
julia> m = [1 2; 4 5]
2×2 Array{Int64,2}:
 1  2
 4  5

julia> det(m)
-3.0
```

Matrizen werden in R mit „solve()“ invertiert und in Julia mit „inv()“ (Abb.46).

Abb.46: Inverse einer Matrix

In R

```
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
> solve(m)
           [,1]       [,2]
[1,] -1.6666667  1.3333333
[2,]  0.6666667 -0.3333333
```

In Julia

```
julia> m
2×2 Array{Int64,2}:
 1  2
 4  5

julia> inv(m)
2×2 Array{Float64,2}:
 -1.66667   0.666667
  1.33333  -0.333333
```

Mit „solve()“ kann man in R lineare Gleichungssysteme lösen. In Julia ist das mit „\“ auch möglich (Abb.47).

Abb.47: lineares Gleichungssystem lösen

In R

```
> A <- matrix(c(1,2,3,4), nrow=2, ncol=2)
> b <- c(6,8)
> solve(A,b)
[1] 0 2
```

In Julia

```
julia> A = [1 3; 2 4]
2×2 Array{Int64,2}:
 1  3
 2  4

julia> b = [6, 8]
2-element Array{Int64,1}:
 6
 8

julia> A \ b
2-element Array{Float64,1}:
 0.0
 2.0
```

Sowohl in R als auch in Julia lassen sich Eigenwerte und Eigenvektoren über den Befehl „eigen()“ berechnen. Über diesen Output kann man dann in R nochmals separat auf die Eigenwerte oder die Eigenevektoren zugreifen. Unabhängig von dem Befehl „eigen()“ kann in Julia auch mit „eigenvals()“ und „eigenvecs()“ die Eigenwerte und Eigenvektoren berechnet werden (Abb.48).

Abb.48: Eigenwerte und Eigenvektoren

In R

```
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
> eigen(m)
eigen() decomposition
$values
[1]  6.4641016 -0.4641016

$vectors
           [,1]       [,2]
[1,] -0.5906905 -0.9390708
[2,] -0.8068982  0.3437238

> eigen(m)$values
[1]  6.4641016 -0.4641016
> eigen(m)$vectors
           [,1]       [,2]
[1,] -0.5906905 -0.9390708
[2,] -0.8068982  0.3437238
```

In Julia

```
julia> m
2×2 Array{Int64,2}:
 1  2
 4  5

julia> eigen(m)
Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}
values:
2-element Array{Float64,1}:
 -0.4641016151377544
  6.464101615137754
vectors:
2×2 Array{Float64,2}:
 -0.806898  -0.343724
  0.59069   -0.939071

julia> eigvals(m)
2-element Array{Float64,1}:
 -0.4641016151377544
  6.464101615137754

julia> eigvecs(m)
2×2 Array{Float64,2}:
 -0.806898  -0.343724
  0.59069   -0.939071
```

Vektoren und Matrizen der gleichen Dimension können in R über „cbind()“ spaltenweise miteinander verknüpft werden und über „rbind()“ zeilenweise miteinander verknüpft werden. In Julia ist der Befehl „hcat()“ mit „cbind() und „vcat()“ mit „rbind()“ zu vergleichen (Abb.49).

Abb.49: Vektoren und Matrizen der gleichen Dimension spalten- oder zeilenweise verknüpfen

In R

```
> h
[1] 9 1 5
> l
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
> n = matrix(c(7,4), nrow=1, ncol=2)
> cbind(h,h)
     h h
[1,] 9 9
[2,] 1 1
[3,] 5 5
> cbind(h,l)
     h
[1,] 9 1 2
[2,] 1 3 4
[3,] 5 5 6
> rbind(h,h)
  [,1] [,2] [,3]
h    9    1    5
h    9    1    5
> rbind(l,n)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    4
```

In Julia

```
julia> h
3-element Array{Int64,1}:
 9
 1
 5

julia> l
3×2 Transpose{Int64,Array{Int64,2}}:
 1  4
 2  5
 3  6

julia> n = [7 4]
1×2 Array{Int64,2}:
 7  4

julia> hcat(h,h)
3×2 Array{Int64,2}:
 9  9
 1  1
 5  5

julia> hcat(h,l)
3×3 Array{Int64,2}:
 9  1  4
 1  2  5
 5  3  6

julia> vcat(h,h)
6-element Array{Int64,1}:
 9
 1
 5
 9
 1
 5

julia> vcat(l,n)
4×2 Array{Int64,2}:
 1  4
 2  5
 3  6
 7  4
```

Tab.6: Vektoren und Matrizen

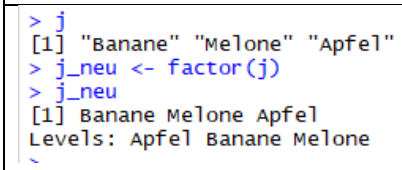
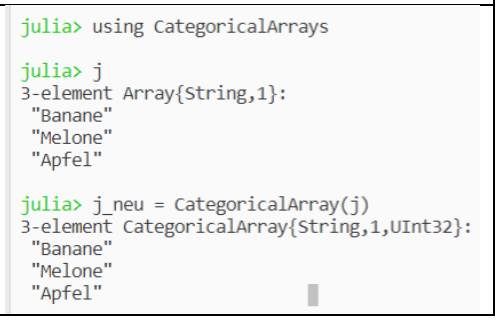
Bedeutung	In R	In Julia
Vektor	c(1,2,3)	[1,2,3]
sequentielle Vektoren mit Standardschrittweite +1 oder -1 erzeugen	*Startwert*: *Endwert*	collect(*Startwert*:*Endwert*)
Flexiblere sequentielle Vektoren erzeugen	seq(from=*Startwert*, to=*Endwert*, by=*Schrittweite*, length=*Länge*)	• [*Startwert*:*Schrittweite*:*Endwert*;], • collect(*Startwer*:*Schrittweite*:*Endwert*) • collect(range(*Startwert*, step=*Schrittweite*, length=*Länge*, stop=*Endwert*))
Länge eines Vektors	length(*Vektor*)	length(*Vektor*)
Vektoreintrag extrahieren	c(9,7,3)[2]	[9,7,3][2]
Vektor sortieren	sort()	sort()
Neuer Vektoreintrag anhängen	c(*Vektor*, *neuer Eintrag*)	push!(*Vektor*, *neuer Eintrag)*
Mehrere Vektoren verknüpfen	c($Vektor_1$, $Vektor_2$)	append!($Vektor_1$, $Vektor_2$)
Vektorelement entfernen	*Vektor*[-*Stelle*]	• popfirst!(*Vektor*) • splice!(*Vektor*, *Stelle*) • pop!(*Vektor*)

Datenobjekt replizieren	rep(*Datenobjekt*, times=*Anzahl der Wiederholungen*, each=*Anzahl der aufeinanderfolgenden Wiederholungen*)	repeat(*Datenobjekt*, outer= *Anzahl der Wiederholungen*, inner= *Anzahl der aufeinanderfolgenden Wiederholungen*)
Matrix erstellen	matrix(data=*Vektor*, nrow=*Anzahl der Zeilen*, ncol=*Anzahl der Spalten*) Bsp : matrix(c(1,2,3,4), nrow=2,ncol=3) => 1 3 5 2 4 6	[1 2 3 ; 4 5 6] => 1 2 3 4 5 6
Matrixdimension	dim()	size()
Anzahl der Zeilen	nrow()	size(A)[1], size(A,1)
Anzahl der Spalten	ncol()	size(A)[2], size(A,2)
Matrixelemente extrahieren	A[i,j]	A[i,j]
i-te Matrixzeile extrahieren	A[i,]	A[i,:]
j-te Matrixspalte extrahieren	A[,j]	A[:,j]
mxn-Nullmatrix	matrix(0,m,n)	zeros(m,n)
nxn-Einheitsmatrix	diag(1, nrow=n)	Matrix{Datentyp}(I,n,n)
mxn-Matrix aus Einsen	matrix(1,m,n)	ones(m,n)
Transponierte Matrix	t(A)	• transpose(A) • A‘
Matrixmultiplikation	A %*% B	A*B
Elementw. Matrixmultiplikation	A*B	A.*B
Äußeres Produkt	%o%, outer()	
Kreuzprodukt X‘X einer Matrix X	crossprod(*Matrix*) (Abk. für t(A) %*% A)	
Determinante	det()	det()
Inverse einer Matrix	solve()	inv()
Ax=b lösen	solve(A,b)	A\b
Eigenwerte und Eigenvektoren	eigen()	eigen() eigvals(), eigvecs()
Zeilenweise Vektoren und Matrizen der gl. Dimension verbinden	rbind()	vcat()
Spaltenweise Vektoren und Matrizen der gl. Dimension verbinden	cbind()	hcat()

3.2.4. Behandlung von kategorialen Variablen

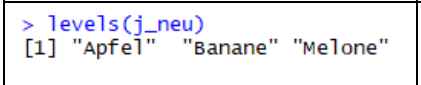
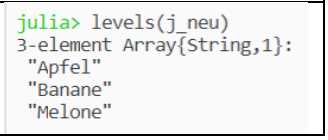
In R werden die kategorialen Variablen mit dem Befehl „factor()“ in Faktorvariablen umkodiert. In Julia muss man zunächst das Zusatzpaket „CategoricalArrays“ laden. Erst dann kann die kategoriale Variable mit dem Befehl „CategoricalArray()“ in eine Faktorvariable umgewandelt werden (Abb.50).

Abb.50: Umkodieren von kategorialen Variablen

In R	In Julia
	

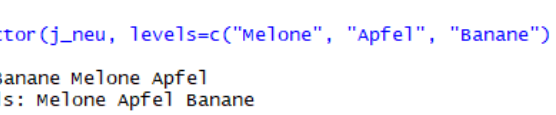
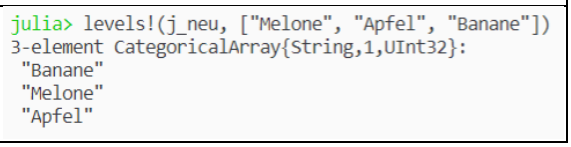
Um die Faktorlevels anzusehen, gibt man in beiden Sprachen den Befehl „levels()“ ein (Abb.51).

Abb.51: Levels ansehen

In R	In Julia
	

Um die Reihenfolge der Faktorlevels manuell zu bestimmen, verwendet man in R den Befehl „factor()“ und in Julia den Befehl „levels!()“ (Abb.52).

Abb.52: Levels umordnen

In R	In Julia
	

Außerdem ist es möglich, in beiden Sprachen die Levels nach dem Alphabet oder aufsteigend in der Zahlenfolge zu sortieren. Dazu verwendet man in beiden Sprachen den Befehl „sort()“ (Abb.53).

Abb.53: Levels sortieren

In R

```
> j_neu1 <- cbind(j,j)
> j_neu1 <- sort(j_neu1)
> j_neu1
[1] "Apfel"  "Apfel"  "Banane" "Banane" "Melone"
[6] "Melone"
```

In Julia

```
julia> j_neu1 = vcat(j,j)
6-element Array{String,1}:
 "Banane"
 "Melone"
 "Apfel"
 "Banane"
 "Melone"
 "Apfel"

julia> sort(j_neu1)
6-element Array{String,1}:
 "Apfel"
 "Apfel"
 "Banane"
 "Banane"
 "Melone"
 "Melone"
```

Tab.7: Behandlung von kategorialen Variablen

Bedeutung	In R	In Julia
Faktorumwandlung	factor()	• CategoricalArray() • categorial!()
Faktorlevels abfragen	levels()	levels!()
Faktorlevels umordnen	sort()	sort()

3.2.5. Häufigkeitstabellen

Um in R eine Kontingenztafel mit absoluten Häufigkeiten zu erstellen, verwende den Befehl „table()“ . In Julia erhält man ebenfalls solch eine Tabelle entweder über den Befehl „countmap()“ aus dem Zusatzpaket „StatsBase“ oder über den Befehl „freqtable()“ aus dem Zusatzpaket „FreqtTables“. Zweiteres kommt hinsichtlich des Outputs dem Befehl „table()“ in R am nächsten (Abb.54). Für die Ausgabe von relativen Häufigkeitstabellen, gibt man in R „prob.table(table())“ in die Konsole ein. Für die Erstellung von relativen Häufigkeitstabellen gibt es zurzeit in Julia noch keinen Befehl (Abb.55).

Abb.54: absolute Häufigkeitstabellen

In R

```
> g
[1] "Melone" "5"      NA       "d"
> j_neu1
[1] "Apfel"  "Apfel"  "Banane" "Banane" "Melone"
[6] "Melone"
> o <- c(g, j_neu1)
> table(o)
o
     5  Apfel Banane      d Melone
     1      2      2      1      3
```

In Julia

```
julia> using StatsBase

julia> g
9-element Array{Any,1}:
 5
  'd'
 7
  "Kiwi"
  "Melone"
 5
  missing
  'd'
 7

julia> j_neu1
6-element Array{String,1}:
 "Banane"
 "Melone"
 "Apfel"
 "Banane"
 "Melone"
 "Apfel"

julia> o = vcat(g, j_neu1)
15-element Array{Any,1}:
 5
  'd'
 7
  "Kiwi"
  "Melone"
 5
  missing
  'd'
 7
  "Banane"
  "Melone"
  "Apfel"
  "Banane"
  "Melone"
  "Apfel"

julia> countmap(o)
Dict{Any,Int64} with 8 entries:
  7       => 2
  "Melone" => 3
  'd'     => 2
  "Banane" => 2
  5       => 2
  "Kiwi"  => 1
  missing => 1
  "Apfel" => 2

julia> using FreqTables

julia> freqtable(o)
8-element Named Array{Int64,1}
Dim1    │
────────┼──
7       │ 2
Melone  │ 3
'd'     │ 2
Banane  │ 2
5       │ 2
Kiwi    │ 1
missing │ 1
Apfel   │ 2
```

Abb.55: relative Häufigkeitstabellen

in R

```
> prop.table(table(o))
o
        5     Apfel    Banane         d    Melone
0.1111111 0.2222222 0.2222222 0.1111111 0.3333333
```

in Julia

Tab.8: Häufigkeitstabellen		
Bedeutung	In R	In Julia
Absolute Häufigkeitstabelle	table()	• countmap() • freqtable()
Relative Häufigkeitstabelle	prop.table(table())	

3.2.6. Logische Werte und logische Operatoren

In beiden Programmiersprachen existieren die logischen Werte. In R werden diese mit „TRUE“ und „FALSE“ bezeichnet. Die Ausdrücke „T“ und „F“ liefern dasselbe Ergebnis. In Julia werden diese Werte mit „true“ und „false“ bezeichnet. Die logischen Werte werden in beiden Sprachen automatisch zu 1 und 0 umgewandelt, sobald man diese Werte mit numerischen Werten berechnet (Abb.56).

Abb.56: logische Werte

In R

```
> TRUE
[1] TRUE
> FALSE
[1] FALSE
> TRUE + 3
[1] 4
> 4 - FALSE
[1] 4
```

In Julia

```
julia> true
true

julia> false
false

julia> true + 3
4

julia> 4 - false
4
```

Die logischen Operatoren unterscheiden sich in den beiden Sprachen nicht. Sowohl für die Vergleiche als auch für die Verknüpfungen wird dieselbe Schreibweise verwendet (Abb.57).

Abb.57: logische Operatoren

In R

```
> # Vergleiche
> 2 < 5
[1] TRUE
> 6 > 9
[1] FALSE
> 3 <= 3
[1] TRUE
> 4 >= 7
[1] FALSE
> 6 == 8
[1] FALSE
> 9 != 2
[1] TRUE
>
> # Verknüpfungen
> (2 < 5) && (9 != 2)
[1] TRUE
> (6 == 8) || (3 <= 3)
[1] TRUE
```

In Julia

```
julia> # Vergleiche

julia> 2 < 5
true

julia> 6 > 9
false

julia> 3 <= 3
true

julia> 4 >= 7
false

julia> 6 == 8
false

julia> 9 != 2
true

julia> # Verknüpfungen

julia> (2 < 5) && (9 != 2)
true

julia> (6 == 8) || (3 <= 3)
true
```

Tab.9: Logische Werte und logische Operatoren		
Bedeutung	In R	In Julia
logische Werte	TRUE, FALSE	true, false
Relationen	==, !=, <, <=, >, >=, %in%	==, != , <, <=, >, >=
logische Operatoren	&, &&, |, ||, xor()	&&, ||

3.2.7. Funktionen, Bedingungen, Verzweigungen und Schleifen

In R werden eigene Funktionen über den Befehl „function(){}“ erzeugt. Dabei werden in den runden Klammern die Parameter übergeben, welche beim Funktionsaufruf zur Verfügung stehen und in den geschweiften Klammern stehen die Anweisungen, also die eigentliche Funktionsschreibweise. In Julia verzichtet man komplett auf die geschweiften Klammern. Entweder werden die Funktionen so aufgestellt, wie man es in der Mathematik aus der Schule kennt. Oder aber es wird die Schreibweise verwendet, bei dem die Anweisungen zwischen den Ausdrücken „function“ und „end“ stehen. Damit ist es einfacher und schneller in Julia Funktionen aufzustellen als in R (Abb.58).

Abb.58: Funktionen erstellen

In R

```
> f <- function(x){
+   x^2
+ }
> f(4)
[1] 16
```

In Julia

```
julia> f(x)=x^2
f (generic function with 1 method)

julia> function f2(x)

           x^3

       end
f2 (generic function with 1 method)

julia> f(4)
16

julia> f2(2)
8
```

Mit Schleifen können bestimmte Anweisungen einer Funktion beliebig oft wiederholt werden und erspart einem das ständige Erstellen von Funktionen. Sowohl in R als auch in Julia können die „for“- und die „while“-Schleifen über die Befehle „for“ und „while“ aufgerufen werden. In Julia wird wieder bewusst auf die runden und geschweiften Klammern verzichtet und erleichtert einem dadurch den Programmieraufwand (Abb.59+60).

Abb.59: for- Schleife erstellen

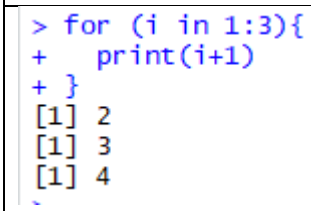
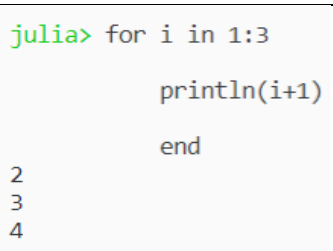
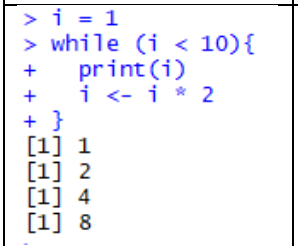
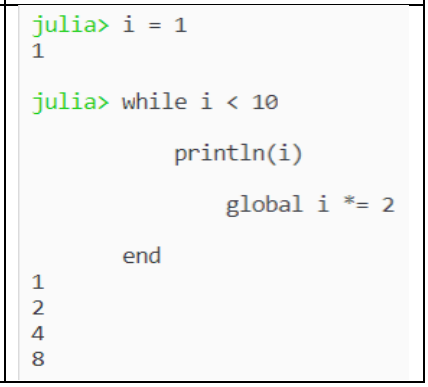
In R	In Julia
	

Abb.60: while-Schleife erstellen

In R	In Julia
	

Bedingte Ausführungen bzw. Verzweigungen können mit den Befehlen „if“, „if-else“ und „if-elseif“ aufgerufen werden. Ähnlich zu Funktionen und Schleifen werden in R die Variablenbedingungen in runden Klammern geschrieben und die Anweisungen innerhalb den geschweiften Klammern. In Julia dagegen verzichtet man auf jegliche Art von Klammern. Wie man in den Abbildungen 61-63 sehen kann, sind die Verzweigungen in Julia wesentlich schneller zu programmieren und es lässt sich viel besser lesen.

Abb.61: if- Bedingung erstellen

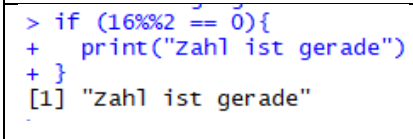
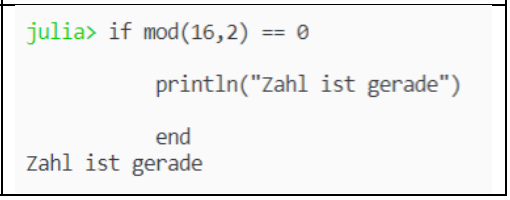
In R	In Julia
	

Abb.62: if-else-Verzweigung erstellen

In R

```
> if (2 == 3){
+   print("die Werte stimmen überein")
+ } else {
+   print("die Werte stimmen nicht überein")
+ }
[1] "die Werte stimmen nicht überein"
```

In Julia

```
julia> if 2 == 3
           println("die Werte stimmen überein")
       else
           println("die Werte stimmen nicht überein")
           end
die Werte stimmen nicht überein
```

Abb.63: if-elseif-else-Verzweigung erstellen

In R

```
> if (5 > 4){
+   print("5 ist größer als 4")
+ } else if (5 < 4){
+   print("5 ist kleiner als 4")
+ } else {
+   print("5 ist gleich 4")
+ }
[1] "5 ist größer als 4"
```

In Julia

```
julia> if 5 > 4
           println("5 ist größer als 4")
       elseif 5 < 4
           println("5 ist kleiner als 4")
       else
           println("5 ist gleich 4")
       end
5 ist größer als 4
```

Tab.10: Funktionen, Verzweigungen und Schleifen

Bedeutung	In R	In Julia
Funktion	f <- function(x) { x^2}	f(x) = x^2 function f(x) Anweisung end

if- Verzweigung	if (Bedingung) {
	Anweisung
	}
		if Bedingung
		Anweisung
		end

if-else-Verzweigung	if (Bedingung) {
	Anweisung
	}
	else {
	Anweisung
	}
		if Bedingung
		Anweisung
		else
		Anweisung
		end

if-elseif- Verzweigung	if (Bedingung) { Anweisung } e	if Bedingung Anweisung elseif Bedingung Anweisung end

for-Schleife (i=1,…,n)	for (i in 1:n) {
	Anweisung
	}
		for i in 1:n
		Anweisung
		end

while-Schleife	while (Bedingung) {
	Anweisung
	}
		while Bedingung
		Anweisung
		end

3.2.8. Zufallsgrößen, Dichtefunktionen, Verteilungsfunktionen und Quantilsfunktionen

Beide Programmiersprachen bieten die Möglichkeit Zufallszahlen zu generieren oder auch häufig benötigte Dichte-, Verteilungs- und Quantilsfunktionen von bestimmten Verteilungen an beliebigen Stellen zu berechnen. Beispiele hierfür liefert die Abbildung (Abb.64).

Abb.64: Seed setzen, Zufallsstichproben ziehen, Zufallzahlen einer bestimmten Verteilung generieren, Dichte-. Verteilungs- und Quantilsfunktion berechnen

In R

```
> # Seed setzen zur Reproduzierbarkeit
> set.seed(123)
>
> # Zufallsstichprobe ziehen ..........................
..................................................
> sample(1:20, 4)
[1] 15 19 14  3
> sample(1:5, size=10, replace=TRUE)
 [1] 2 2 3 5 4 1 2 3 5 3
> sample(1:5, size=3, prob=c(0.2, 0.1, 0.4, 0.1, 0.2))
[1] 2 4 5
>
> # Zufallszahlen aus best. Verteilungen generieren, z.B.
 Stichprobe vom Umfang n=5 aus N(0,1)_Verteilung
> rnorm(5)
[1]  0.3598138  0.4007715  0.1106827 -0.5558411
[5]  1.7869131
>
> # Bsp: Dichte der N(0,1)-Verteilung an der Stelle x=0 .
..................................................
> dnorm(0)
[1] 0.3989423
>
> # Bsp: Verteilungsfunktion der N(0,1)-Verteilung an der
 Stelle q=0 oder q=1.96 .........................
> pnorm(0)
[1] 0.5
> pnorm(1.96)
[1] 0.9750021
>
> # Bsp: p-Quantil der N(0,1)-Verteilung z_p mit p=0.95 .
..................................................
> qnorm(0.95)
[1] 1.644854
```

In Julia

```
julia> using Random

julia> Random.seed!(123)
MersenneTwister(UInt32[0x0000007b], Random.DSFMT.DSI
3116007, 222134151, 1073120226, -290652630, 1072956
32671753, 1073438661  …  138346874, 1073030449, 104
3, 1597138926, -775229811, 32947490, 382, 0]), [0.0
 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
0000000000000000000000000000000, 0x0000000000000000
0000000000000000000, 0x0000000000000000000000000000
0000000000, 0x0000000000000000000000000000000, 0x00
, 0x0000000000000000000000000000000, 0x0000000000000
0000000000000000000000  …  0x0000000000000000000
0000000000000000000, 0x0000000000000000000000000000
00000000, 0x0000000000000000000000000000000, 0x000
 x0000000000000000000000000000000, 0x0000000000000
0000000000000000000000000, 0x0000000000000000000000

julia> # Zufallsstichprobe ziehen

julia> using StatsBase

julia> sample(1:20, 4)
4-element Array{Int64,1}:
  5
 15
 19
 18

julia> using Distributions

julia> # Bsp: Stichprobe vom Umfang n=5 aus N(0,1)_Verteilung

julia> rand(Normal(), 5)
5-element Array{Float64,1}:
  1.5239161501048377
 -1.7777276640658903
 -2.9330595087256466
  0.7822577631437713
  2.313579637440314

julia> # Bsp: Dichte der N(0,1)-Verteilung an der Stelle x=0

julia>  pdf(Normal(), 0)
0.3989422804014327

julia> # an der Stelle q=0 oder q=1.96

julia> cdf(Normal(), 0)
0.5

julia> cdf(Normal(), 1.96)
0.9750021048517795

julia> # Bsp: p-Quantil der N(0,1)-Verteilung z_p mit p=0.95

julia> quantile(Normal(), 0.95)
1.6448536269514717

julia> # Bsp: Verteilungsfunktion der N(0,1)-Verteilung
```

Die Namen dieser verschiedenen Funktionen sind in beiden Programmiersprachen unterschiedlich aufgebaut. Tabelle 11 gibt eine Übersicht zur Berechnung dieser Funktionen in R und Julia. In Julia muss zur Berechnung dieser Werte, die Zusatzpakete „Random“ und „Distributions“ geladen werden.

Tab.11: Zufallsgrößen, Dichte-, Verteilungs- und Quantilsfunktionen		
Bedeutung	in R	in Julia
Seed setzten	set.seed()	Random.seed!()
Ziehen einer Stichprobe	sample()	sample()
Zufallszahlen, die der 1. Normalverteilung 2. Binomialverteilung 3. χ^2-Verteilung 4. Exponentialverteilung 5. Poissonverteilung 6. Cauchyverteilung 7. Gleichverteilung 8. Geom. Verteilung 9. Hypergeom. Verteilung genügen/folgen	1. rnorm() 2. rbinom() 3. rchisq() 4. rexp() 5. rpois() 6. rcauchy() 7. runif() 8. rgeom() 9. rhyper()	1. randn() (Paket Random), rand(Normal()) (Paket Distributions) 2. rand(Binomial()) 3. rand(Chisq()) 4. randexp() (Paket Random), rand(Exponential()) (Paket Distributions) 5. rand(Poisson()) 6. rand(Cauchy()) 7. rand() (Paket Random), rand(Uniform()) 8. rand(Geometric()) 9. rand(Hypergeometric())
Dichte der 1. Normalverteilung 2. Binomialverteilung 3. χ^2-Verteilung 4. Exponentialverteilung 5. Poissonverteilung 6. Cauchyverteilung 7. Gleichverteilung 8. Geom. Verteilung 9. Hypergeom. Verteilung	1. dnorm() 2. dbinom() 3. dchisq() 4. dexp() 5. dpois() 6. dcauchy() 7. dunif() 8. dgeom() 9. dhyper()	1. pdf(Normal()) 2. pdf (Binomial()) 3. pdf (Chisq()) 4. pdf (Exponential()) 5. pdf (Poisson()) 6. pdf (Cauchy()) 7. pdf (Uniform()) 8. pdf (Geometric()) 9. pdf (Hypergeometric())
Verteilungsfunktion der 1. Normalverteilung 2. Binomialverteilung 3. χ^2-Verteilung 4. Exponentialverteilung 5. Poissonverteilung 6. Cauchyverteilung 7. Gleichverteilung 8. Geom. Verteilung 9. Hypergeom. Verteilung	1. pnorm() 2. pbinom() 3. pchisq() 4. pexp() 5. ppois() 6. pcauchy() 7. punif() 8. pgeom() 9. phyper()	1. cdf (Normal()) 2. cdf (Binomial()) 3. cdf (Chisq()) 4. cdf (Exponential()) 5. cdf (Poisson()) 6. cdf (Cauchy()) 7. cdf (Uniform()) 8. cdf (Geometric()) 9. cdf (Hypergeometric())
Quantilsfunktion der 1. Normalverteilung 2. Binomialverteilung 3. χ^2-Verteilung 4. Exponentialverteilung 5. Poissonverteilung 6. Cauchyverteilung 7. Gleichverteilung 8. Geom. Verteilung 9. Hypergeom. Verteilung	1. qnorm() 2. qbinom() 3. qchisq() 4. qexp() 5. qpois() 6. qcauchy() 7. qunif() 8. qgeom() 9. qhyper()	1. quantile (Normal()) 2. quantile (Binomial()) 3. quantile (Chisq()) 4. quantile (Exponential()) 5. quantile (Poisson()) 6. quantile (Cauchy()) 7. quantile (Uniform()) 8. quantile (Geometric()) 9. quantile(Hypergeometric())

3.3. Daten importieren

In beiden Sprachen können bereits erstellte Datensätze aus dem aktuellen Arbeitsverzeichnis in der Konsole eingelesen werden. Das Arbeitsverzeichnis kann man in R mit „getwd()“ und in Julia mit

„pwd()“ abgefragt werden. Es kann auch ein bestimmtes Arbeitsverzeichnis eingestellt werden, indem man in R die Funktion „setwd()“ benutzt und in Julia „cd()“. Wichtig hierbei ist, dass der Pfad in einer bestimmten Form geschrieben ist. Hierfür gibt es in R und in Julia zwei Möglichkeiten: Zum Beispiel kann eine txt-Datei über den Pfad “C:\\Users\\…\\datei.txt“ eingelesen werden oder über “C:/Users/…/datei.txt“ eingelesen werden. Daten können in unterschiedlichen Formaten vorliegen. Wie welche Art von Datei eingelesen werden kann, kann aus der folgenden Tabelle 12 entnommen werden.

Tab.12: Daten importieren		
Bedeutung	In R	In Julia
aktuelles Arbeitsverzeichnis abfragen	getwd()	pwd()
neues Arbeitsverzeichnis festlegen	setwd()	cd()
Daten in Textform einlesen	read.table(“*datei.txt*“, header=TRUE)	readtable(“*datei.txt*“)
csv-Daten einlesen	• read.csv(“*datei.csv*”) • read.csv2(“*datei.csv*”)	readcsv(“*datei.csv*”)
Datei mit Tabulatoren als Wertetrennung einlesen	read.delim(„d*atei.tsv*“)	readddlm(„d*atei.tsv*“, delim=`\t`)
Daten im Excel-Format einlesen	read.xlsx(„*datai.xlsx*“))	XLSX.readxlsc(„*datai.xlsx*“)
Daten im SPSS-Format einlesen	read.spss(„*datei.dta*“)	DataFrame(load(„*datei.dta*“)) (Paket StatFiles, DataFrames)

3.4. Daten aufbereiten

In diesem Unterkapitel werden zunächst die Datenobjekttypen Liste und Tupel gegenübergestellt. Es wird darauf eingegangen, wie man mit diesen Typen arbeitet und wie mit denen umgegangen wird. Anschließend wird kurz erklärt, wie man eigene Datensätze in den beiden Programmiersprachen erstellt. Zum Schluss wird am bekannten R-Datensatz „iris“ erkärt, wie man generell Daten aufbereitet und diese analysiert.

3.4.1. Listen vs. Tupel

Das Datenobjekt Liste in R ist eine Menge von Elementen unterschiedlicher Datenobjekttypen. Listen existieren in Julia nicht, aber vergleichbar sind die Listen mit den Datenobjekt Tupel in Julia. In R werden Listen mit dem Befehl „list()“ erstellt. Im entsprechenden Output sind die Listenelementen spaltenweise aufgelistet. In Julia können Tupel auf zwei Arten erstellt werden. Entweder wird ein Tupel über die runden Klammern erzeugt oder über den Befehl „tuple()“. Der Output der Tupelelemente wird hier zeilenweise ausgegeben (Abb.65).

Abb.65: Listen bzw. Tupel erstellen

In R	In Julia

`> list(1, 5, "hello", "bye")`
`[[1]]`
`[1] 1`
 	
`[[2]]`
`[1] 5`
 	
`[[3]]`
`[1] "hello"`
 	
`[[4]]`
`[1] "bye"`
	`julia> (1, 5, "hello", "bye")`
	`(1, 5, "hello", "bye")`
	
	`julia> tuple(1, 5, "hello", "bye")`
	`(1, 5, "hello", "bye")`

Der Zugriff auf die Listen- bzw. Tupelelemente kann in R über die Klammern „[[]]“ erreicht werden. Man kann hierbei über die Listennamen auf das Listenelement greifen oder über den Listenindex. In Julia kann man nur über die rechteckigen Klammern auf das Tupelelement zugreifen. Der Zugriff über den Tupelnamen funktioniert hier leider nicht (Abb.66).

Abb.66: Listen- bzw. Tupelzugriff

In R	In Julia

`> liste <- list(1, 5, "hello", "bye")`
`> liste1 <- list(a=1, b=5, c="hello", d="bye")`
`> liste[[2]]`
`[1] 5`
`> liste[[1:3]]`
`Error in liste[[1:3]] : recursive indexing failed at level 2`
`> liste[c(1,4)]`
`[[1]]`
`[1] 1`
 	
`[[2]]`
`[1] "bye"`
 	
`> liste1$b`
`[1] 5`
`> liste1[["b"]]`
`[1] 5`
`> liste1[c("b", "d")]`
`$b`
`[1] 5`
 	
`$d`
`[1] "bye"`
	`julia> tupel = (1, 5, "hello", "bye")`
	`(1, 5, "hello", "bye")`
	
	`julia> tupel1 = (a=1, b=5, c="hello", d="bye")`
	`(a = 1, b = 5, c = "hello", d = "bye")`
	
	`julia>`
	
	`julia>`
	
	`julia> tupel[2]`
	`5`
	
	`julia> tupel[1:3]`
	`(1, 5, "hello")`
	
	`julia> tupel[[1,4]]`
	`(1, "bye")`
	
	`julia> tupel1[:b]`
	`5`
	
	`julia> tupel1["b"]`
	`ERROR: MethodError: no method matching getindex(::NamedTuple{(:a, :b, :c, :d),Tuple{Int64,Int64,String,String}}, ::String)`
	`Closest candidates are:`
	`  getindex(::NamedTuple, ::Int64) at namedtuple.jl:94`
	`  getindex(::NamedTuple, ::Symbol) at namedtuple.jl:95`
	`Stacktrace:`
	` [1] top-level scope at none:0`
	
	`julia> tupel1[[:b, :d]]`
	`ERROR: MethodError: no method matching getindex(::NamedTuple{(:a, :b, :c, :d),Tuple{Int64,Int64,String,String}}, ::Array{Symbol,1})`

Es ist möglich, die Listen bzw. die Tupel nachträglich um weitere Elemente zu erweitern oder sogar diese mit anderen Listen bzw. Tupel zu verknüpfen. In R benutzt man hierfür den Befehl „c()“ mit dem man auch Vektoren erzeugt. In Julia dagegen verwendet man die runden Klammern, mit denen man eigentlich Tupel erzeugt (Abb.67+68).

Abb.67: neues Listen- bzw. Tupelelement anhängen

In R	In Julia

```
> liste
[[1]]
[1] 1

[[2]]
[1] 5

[[3]]
[1] "hello"

[[4]]
[1] "bye"

> c(liste, "K")
[[1]]
[1] 1

[[2]]
[1] 5

[[3]]
[1] "hello"

[[4]]
[1] "bye"

[[5]]
[1] "K"

> c(liste, "K", NA)
[[1]]
[1] 1

[[2]]
[1] 5

[[3]]
[1] "hello"

[[4]]
[1] "bye"

[[5]]
[1] "K"

[[6]]
[1] NA
```

```
julia> tupel
(1, 5, "hello", "bye")

julia> (tupel, 'K')
((1, 5, "hello", "bye"), 'K')

julia> (tupel, 'K', missing)
((1, 5, "hello", "bye"), 'K', missing)
```

Abb.68: mehrere Listen miteinander verknüpfen

In R	In Julia

`> liste`
`[[1]]`
`[1] 1`
 	
`[[2]]`
`[1] 5`
 	
`[[3]]`
`[1] "hello"`
 	
`[[4]]`
`[1] "bye"`
 	
`> liste2 <- list(9, NA, "f")`
`> c(liste, liste2)`
`[[1]]`
`[1] 1`
 	
`[[2]]`
`[1] 5`
 	
`[[3]]`
`[1] "hello"`
 	
`[[4]]`
`[1] "bye"`
 	
`[[5]]`
`[1] 9`
 	
`[[6]]`
`[1] NA`
 	
`[[7]]`
`[1] "f"`
	`julia> tupel`
	`(1, 5, "hello", "bye")`
	
	`julia> tupel2 = (9, missing, 'f')`
	`(9, missing, 'f')`
	
	`julia> (tupel, tupel2)`
	`((1, 5, "hello", "bye"), (9, missing, 'f'))`

Tab.13: Daten aufbereiten: Listen		
Bedeutung	In R	In Julia
Liste bzw. Tupel	list(1, 5, “hello“, “bye“)	(1, 5, “hello“, “bye“), tuple(1, 5, “hello“, “bye“)
Listen-/ Tupelzugriff (i-te Stelle)	• list()[[i]] • list()$Variablename	()[i]
neues Listenelement anhängen	c(*liste*, *neuer Eintrag*)	(*tupel*, *neuer Eintrag*)
Liste anhängen	c($liste_1$, $liste_2$)	(*tupel1*, *tupel2*)

3.4.2. Data Frames

Sowohl in R als auch in Julia kann man eigene Datensätze auf zwei Arten erzeugen. In R kann man während der Erstellung das Datensatz mit „data.frame()“ bereits in den Klammern die Variablen definieren. Alternativ erstellt man erst die Variablen und kombiniert diese erst dann zu einem Datensatz. In Julia muss für die Erzeugung der Datensätze zunächst das Paket „DataFrames“ geladen werden. Anschließend können wie in R, während der Erzeugung des Datensatzes mit dem Befehl „DataFrame()“ die Variablen definiert werden. Alternativ erstellt man erst ein Datensatz mit leeren Einträgen und definiert dann die Variablen, die in diesem Datensatz enthalten sein sollen (Abb.69).

Abb.69: Data Frames erstellen

In R

```
> df <- data.frame(a = 1:4, b = c("h", "d", "e", "s")
)
> df
  a b
1 1 h
2 2 d
3 3 e
4 4 s
> df$a
[1] 1 2 3 4
> df$b
[1] h d e s
Levels: d e h s
>
> # alternativ
> a <- 4:7
> b <- c("Hase", "Maus", "Hund", "Katze")
> df2 <- data.frame(a, b)
> df2
  a     b
1 4  Hase
2 5  Maus
3 6  Hund
4 7 Katze
```

```
> df <- data.frame(a = 1:4, b = c("h", "d", "e", "s")
)
> df
  a b
1 1 h
2 2 d
3 3 e
4 4 s
> df$a
[1] 1 2 3 4
> df$b
[1] h d e s
Levels: d e h s
>
> # alternativ
> a <- 4:7
> b <- c("Hase", "Maus", "Hund", "Katze")
> df2 <- data.frame(a, b)
> df2
  a     b
1 4  Hase
2 5  Maus
3 6  Hund
4 7 Katze
```

In Julia

```
julia> using DataFrames

julia> df1 = DataFrame(a = 1:4, b = ['h', 'd', 'e', 's'])
4×2 DataFrame
| Row | a     | b    |
|     | Int64 | Char |
|-----|-------|------|
| 1   | 1     | 'h'  |
| 2   | 2     | 'd'  |
| 3   | 3     | 'e'  |
| 4   | 4     | 's'  |

julia> df1.a
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> df1."a"
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> # alternativ

julia> df2 = DataFrame()
0×0 DataFrame


julia> df2.a = 4:7
4:7

julia> df2.b = ["Hase", "Maus", "Hund", "Katze"]
4-element Array{String,1}:
 "Hase"
 "Maus"
 "Hund"
 "Katze"

julia> df2
4×2 DataFrame
| Row | a     | b      |
|     | Int64 | String |
|-----|-------|--------|
| 1   | 4     | Hase   |
| 2   | 5     | Maus   |
| 3   | 6     | Hund   |
| 4   | 7     | Katze  |
```

Im Folgenden wird mit dem bekannten R-Datensatz „iris“ (Abb.70) weiter gearbeitet. Der Datensatz „iris“ liefert Informationen von 150 Beobachtungen der Schwertlilie, an dem 4 Variablen erhoben wurden. Gemessen wurden die Länge und Breite des Kelchblatts (Sepalum) sowie desKronblatts (Petalum) in Zentimeter. Des Weiteren ist für jede Beobachtung die Art der Schwertlilie (Iris setosa, Iris virginica oder Iris versicolor) angegeben, für die je 50 Beobachtungen vorliegen [15]. Mit diesem Datensatz soll nun verglichen werden, wie man in R bzw. in Julia mit einem realen Datensatz arbeitet, wichtige Informationen daraus filtert, auf bestimmte Zeilen oder Spalten des Datensatzes zugreift, den Datensatz erweitert und wie man mit fehlenden Werten darin umgeht.

Abb.70: Datensatz iris (Ausschnitt)

```
> iris
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1            5.1         3.5          1.4         0.2     setosa
2            4.9         3.0          1.4         0.2     setosa
3            4.7         3.2          1.3         0.2     setosa
4            4.6         3.1          1.5         0.2     setosa
5            5.0         3.6          1.4         0.2     setosa
6            5.4         3.9          1.7         0.4     setosa
```

Um in Julia auf R-Datensätze zuzugreifen, muss das Zusatzpaket „RDatasets“ geladen werden. Danach kann mit dem Befehl „dataset()“ ein bestimmter R-Datensatz aufgerufen werden (Abb.71).

Abb.71: R-Datensätze in Julia verwenden

```
julia> using RDatasets

julia> iris = dataset("datasets", "iris")
150×5 DataFrame. Omitted printing of 1 columns
| Row | SepalLength | SepalWidth | PetalLength | PetalWidth |
|     | Float64     | Float64    | Float64     | Float64    |
|-----|-------------|------------|-------------|------------|
| 1   | 5.1         | 3.5        | 1.4         | 0.2        |
| 2   | 4.9         | 3.0        | 1.4         | 0.2        |
| 3   | 4.7         | 3.2        | 1.3         | 0.2        |
| 4   | 4.6         | 3.1        | 1.5         | 0.2        |
| 5   | 5.0         | 3.6        | 1.4         | 0.2        |
| 6   | 5.4         | 3.9        | 1.7         | 0.4        |
⋮
| 144 | 6.8         | 3.2        | 5.9         | 2.3        |
| 145 | 6.7         | 3.3        | 5.7         | 2.5        |
| 146 | 6.7         | 3.0        | 5.2         | 2.3        |
| 147 | 6.3         | 2.5        | 5.0         | 1.9        |
| 148 | 6.5         | 3.0        | 5.2         | 2.0        |
| 149 | 6.2         | 3.4        | 5.4         | 2.3        |
| 150 | 5.9         | 3.0        | 5.1         | 1.8        |
```

Bei der Analyse des Datensatzes „iris“ kann man sich zunächst anschauen, welche und wie viele Variablen erhoben wurden und wie viele Beobachtungen betrachtet wurden. Mit dem Befehl „dim()“ in R bzw. . „size()“ in Julia kann man sich die Größe des Datensatzes ausgeben lassen (Abb.72). Welche Variablen erhoben wurden, kann man in beiden Sprachen mit „names()“ herausfinden. In R gibt der Befehl „colnames()“ denselben Output aus wie „names()“. Zusätzlich kann man in R sich die Zeilennamen des Datensatzes ausgeben (Abb.73).

Abb.72: Dimension eines Data Frames

In R	In Julia
`> dim(iris)` `[1] 150   5`	`julia> size(iris)` `(150, 5)`

Abb.73: Variablen- und Zeilennamen eines Data Frames

In R

```
> names(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length"
[4] "Petal.Width"  "Species"
> colnames(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length"
[4] "Petal.Width"  "Species"
>
> # Zeilennamen des Datensatzes ausgeben
> rownames(iris)
  [1] "1"   "2"   "3"   "4"   "5"   "6"   "7"
  [8] "8"   "9"   "10"  "11"  "12"  "13"  "14"
 [15] "15"  "16"  "17"  "18"  "19"  "20"  "21"
 [22] "22"  "23"  "24"  "25"  "26"  "27"  "28"
 [29] "29"  "30"  "31"  "32"  "33"  "34"  "35"
 [36] "36"  "37"  "38"  "39"  "40"  "41"  "42"
 [43] "43"  "44"  "45"  "46"  "47"  "48"  "49"
 [50] "50"  "51"  "52"  "53"  "54"  "55"  "56"
 [57] "57"  "58"  "59"  "60"  "61"  "62"  "63"
 [64] "64"  "65"  "66"  "67"  "68"  "69"  "70"
 [71] "71"  "72"  "73"  "74"  "75"  "76"  "77"
 [78] "78"  "79"  "80"  "81"  "82"  "83"  "84"
 [85] "85"  "86"  "87"  "88"  "89"  "90"  "91"
 [92] "92"  "93"  "94"  "95"  "96"  "97"  "98"
```

In Julia

```
julia> names(iris)
5-element Array{String,1}:
 "SepalLength"
 "SepalWidth"
 "PetalLength"
 "PetalWidth"
 "Species"
```

Der nächste Schritt in einer Datenanalyse ist es, sich einen Überblick über die Daten zu verschaffen. Dies kann in R mit den Befehlen „str()“ und „summary()“ durchgeführt werden. Mit „str()“ erfährt man, um was für Datenobjekte es sich bei den erhobenen Variablen handelt, also was für eine Datenstruktur die Variablen haben. Und mit „summary()“ erhält man die Information, in welchen Wertebereich die metrischen Variablen liegen und weitere wichtige statistische Kennzahlen zu den einzelnen Variablen. In Julia kann man einen ähnlichen Output nur mit dem Befehl „describe()“ erreichen (Abb.74).

Abb.74: Überblick über die Daten verschaffen

In R

```
> str(iris)
'data.frame':   150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4
 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2
.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5
 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2
 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicc
lor",..: 1 1 1 1 1 1 1 1 1 1 ...
> summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length
 Min.   :4.300   Min.   :2.000   Min.   :1.000
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600
 Median :5.800   Median :3.000   Median :4.350
 Mean   :5.843   Mean   :3.057   Mean   :3.758
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100
 Max.   :7.900   Max.   :4.400   Max.   :6.900
  Petal.Width          Species
 Min.   :0.100   setosa    :50
 1st Qu.:0.300   versicolor:50
 Median :1.300   virginica :50
 Mean   :1.199
 3rd Qu.:1.800
 Max.   :2.500
```

In Julia

```
julia> describe(iris)
5x8 DataFrame. Omitted printing of 3 columns
| Row | variable    | mean     | min    | median | max       |
|     | Symbol      | Union…   | Any    | Union… | Any       |
|-----|-------------|----------|--------|--------|-----------|
| 1   | SepalLength | 5.84333  | 4.3    | 5.8    | 7.9       |
| 2   | SepalWidth  | 3.05733  | 2.0    | 3.0    | 4.4       |
| 3   | PetalLength | 3.758    | 1.0    | 4.35   | 6.9       |
| 4   | PetalWidth  | 1.19933  | 0.1    | 1.3    | 2.5       |
| 5   | Species     |          | setosa |        | virginica |
```

Möchte man nun auf bestimmte Zeilen des Datensatzes greifen, so kann man dies sowohl in R als auch in Julia über die Befehle „head()“ die ersten sechs Zeilen des Datensatzes ausgeben lassen. In Julia

existiert zudem die Funktion „first()“, mit der man bestimmen kann, welche ersten Zeilen man ausgeben möchte. Die letzten sechs Zeilen des Datensatzes kann man in R mit „tail()“ und in Julia mit “last()“ ausgeben lassen (Abb.75). Um auf spezifische Zeilen oder Spalten zuzugreifen, verwendet man in beiden Sprachen wieder die Schreibweise mit den eckigen Klammern, in denen man den Zeilen- bzw-den Spaltenindex angibt (Abb.75+76). Der Zugriff auf die Zeilen kann auch über eine Bedingung ermöglicht werden. Dabei ist zu beachten, dass in Julia vor den Relationen ein Punkt steht, damit die Bedingung für jede Zeile geprüft werden kann (Abb.75). Außerdem kann man in R und in Julia auch über den Variablenamen auf bestimmte Spalten zugreifen. In R muss der Variablenname dann in Anführungszeichen gesetzt werden, während man in Julia vor den Variablennamen einen Doppelpunkt „:“ schreibt (Abb.76). Ein Unterschied zwischen R und Julia ist, dass wenn man zum Beispiel die ganze i-te Zeile des Datensatz „iris“ ausgeben möchte, dann muss man in R an der Stelle, wo der Spaltenindex angegeben werden kann, leer lässt, während man in Julia ein Doppelpunkt an dieser Stelle reinschreiben muss (Abb.75+76).

Abb.75: Data Frame-Zugriff über die Beobachtungen

In R

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          5.1         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
3          4.7         3.2          1.3         0.2
4          4.6         3.1          1.5         0.2
5          5.0         3.6          1.4         0.2
6          5.4         3.9          1.7         0.4
  Species
1  setosa
2  setosa
3  setosa
4  setosa
5  setosa
6  setosa
> tail(iris)
    Sepal.Length Sepal.Width Petal.Length
145          6.7         3.3          5.7
146          6.7         3.0          5.2
147          6.3         2.5          5.0
148          6.5         3.0          5.2
149          6.2         3.4          5.4
150          5.9         3.0          5.1
    Petal.Width   Species
145         2.5 virginica
146         2.3 virginica
147         1.9 virginica
148         2.0 virginica
149         2.3 virginica
150         1.8 virginica
> iris[3,]
  Sepal.Length Sepal.Width Petal.Length Petal.Width
3          4.7         3.2          1.3         0.2
  Species
3  setosa
> iris[5:8,]
  Sepal.Length Sepal.Width Petal.Length Petal.Width
5          5.0         3.6          1.4         0.2
6          5.4         3.9          1.7         0.4
7          4.6         3.4          1.4         0.3
8          5.0         3.4          1.5         0.2
  Species
5  setosa
6  setosa
7  setosa
8  setosa
> iris[iris$Sepal.Length > 3.9,]
   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1           5.1         3.5          1.4         0.2     setosa
2           4.9         3.0          1.4         0.2     setosa
3           4.7         3.2          1.3         0.2     setosa
4           4.6         3.1          1.5         0.2     setosa
5           5.0         3.6          1.4         0.2     setosa
6           5.4         3.9          1.7         0.4     setosa
7           4.6         3.4          1.4         0.3     setosa
```

In Julia

```
julia> first(iris,3)
3x5 DataFrame. Omitted printing of 1 columns
| Row | SepalLength | SepalWidth | PetalLength | PetalWidth |
|     | Float64     | Float64    | Float64     | Float64    |
|-----|-------------|------------|-------------|------------|
| 1   | 5.1         | 3.5        | 1.4         | 0.2        |
| 2   | 4.9         | 3.0        | 1.4         | 0.2        |
| 3   | 4.7         | 3.2        | 1.3         | 0.2        |

julia> head(iris)
┌ Warning: `head(df::AbstractDataFrame)` is deprecated, use `first(df, 6)` instead.
│   caller = top-level scope at none:0
└ @ Core none:0
6x5 DataFrame. Omitted printing of 1 columns
| Row | SepalLength | SepalWidth | PetalLength | PetalWidth |
|     | Float64     | Float64    | Float64     | Float64    |
|-----|-------------|------------|-------------|------------|
| 1   | 5.1         | 3.5        | 1.4         | 0.2        |
| 2   | 4.9         | 3.0        | 1.4         | 0.2        |
| 3   | 4.7         | 3.2        | 1.3         | 0.2        |
| 4   | 4.6         | 3.1        | 1.5         | 0.2        |
| 5   | 5.0         | 3.6        | 1.4         | 0.2        |
| 6   | 5.4         | 3.9        | 1.7         | 0.4        |

julia> last(iris, 2)
2x5 DataFrame. Omitted printing of 1 columns
| Row | SepalLength | SepalWidth | PetalLength | PetalWidth |
|     | Float64     | Float64    | Float64     | Float64    |
|-----|-------------|------------|-------------|------------|
| 1   | 6.2         | 3.4        | 5.4         | 2.3        |
| 2   | 5.9         | 3.0        | 5.1         | 1.8        |

julia> iris[3, :]
DataFrameRow. Omitted printing of 1 columns
| Row | SepalLength | SepalWidth | PetalLength | PetalWidth |
|     | Float64     | Float64    | Float64     | Float64    |
|-----|-------------|------------|-------------|------------|
| 3   | 4.7         | 3.2        | 1.3         | 0.2        |

julia> iris[5:8, :]
4x5 DataFrame. Omitted printing of 1 columns
| Row | SepalLength | SepalWidth | PetalLength | PetalWidth |
|     | Float64     | Float64    | Float64     | Float64    |
|-----|-------------|------------|-------------|------------|
| 1   | 5.0         | 3.6        | 1.4         | 0.2        |
| 2   | 5.4         | 3.9        | 1.7         | 0.4        |
| 3   | 4.6         | 3.4        | 1.4         | 0.3        |
| 4   | 5.0         | 3.4        | 1.5         | 0.2        |

julia> iris_filter = iris[iris.SepalWidth .> 3.9, :]
4x5 DataFrame. Omitted printing of 1 columns
| Row | SepalLength | SepalWidth | PetalLength | PetalWidth |
|     | Float64     | Float64    | Float64     | Float64    |
|-----|-------------|------------|-------------|------------|
| 1   | 5.8         | 4.0        | 1.2         | 0.2        |
| 2   | 5.7         | 4.4        | 1.5         | 0.4        |
| 3   | 5.2         | 4.1        | 1.5         | 0.1        |
| 4   | 5.5         | 4.2        | 1.4         | 0.2        |
```

Abb.76: Data Frame-Zugriff über die Variablen

In R

```
> iris[,"Sepal.Length"]
  [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4
 [12] 4.8 4.8 4.3 5.8 5.7 5.4 5.1 5.7 5.1 5.4 5.1
> iris[, c(1,5)]
    Sepal.Length    Species
1            5.1     setosa
2            4.9     setosa
3            4.7     setosa
```

In Julia

```
julia> iris[:, :SepalLength]
150-element Array{Float64,1}:
 5.1
 4.9
 4.7

julia> iris[:, [1,5]]
150x2 DataFrame
| Row | SepalLength | Species |
|     | Float64     | Cat…    |
|-----|-------------|---------|
| 1   | 5.1         | setosa  |
| 2   | 4.9         | setosa  |
| 3   | 4.7         | setosa  |
```

Möchte man den Datensatz nach einer Variablen sortieren, so kann man dies in R über den Befehl „order()“ und der Schreibweise mit den eckigen Klammern erreichen und in Julia über „sort!()“ (Abb.77).

Abb.77: Data Frame sortieren

In R

```
> iris_first <- iris[1:4,]
> iris_first[order(iris_first$Petal.Length),]
  Sepal.Length Sepal.Width Petal.Length Petal.Width
3          4.7         3.2          1.3         0.2
1          5.1         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
4          4.6         3.1          1.5         0.2
  Species
3  setosa
1  setosa
2  setosa
4  setosa
```

In Julia

```
julia> iris_first = first(iris, 4)
4×5 DataFrame. Omitted printing of 1 columns
| Row | SepalLength | SepalWidth | PetalLength | PetalWidth |
|     | Float64     | Float64    | Float64     | Float64    |
|-----|-------------|------------|-------------|------------|
| 1   | 5.1         | 3.5        | 1.4         | 0.2        |
| 2   | 4.9         | 3.0        | 1.4         | 0.2        |
| 3   | 4.7         | 3.2        | 1.3         | 0.2        |
| 4   | 4.6         | 3.1        | 1.5         | 0.2        |

julia> sort!(iris_first, :PetalLength, rev=false)
4×5 DataFrame. Omitted printing of 1 columns
| Row | SepalLength | SepalWidth | PetalLength | PetalWidth |
|     | Float64     | Float64    | Float64     | Float64    |
|-----|-------------|------------|-------------|------------|
| 1   | 4.7         | 3.2        | 1.3         | 0.2        |
| 2   | 5.1         | 3.5        | 1.4         | 0.2        |
| 3   | 4.9         | 3.0        | 1.4         | 0.2        |
| 4   | 4.6         | 3.1        | 1.5         | 0.2        |
```

Um neue Variablen in den Datensatz aufzunehmen, geht man in R und in Julia so vor, wie wenn man einen eigenen Datensatz erzeugt (Abb.78). Dies wurde auf Seite 57f erklärt.

Abb.78: Hinzufügen neuer Variablen

In R

```
> iris_first
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          5.1         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
3          4.7         3.2          1.3         0.2
4          4.6         3.1          1.5         0.2
  Species
1  setosa
2  setosa
3  setosa
4  setosa
> status <- c("fresh", NA, "fresh", "wilted")
> LivingDay <- c(6, 2, NA, 14)
> iris_first <- data.frame(iris_first, status, LivingDay)
> iris_first
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          5.1         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
3          4.7         3.2          1.3         0.2
4          4.6         3.1          1.5         0.2
  Species status LivingDay
1  setosa  fresh         6
2  setosa   <NA>         2
3  setosa  fresh        NA
4  setosa wilted        14
>
```

In Julia

```
julia> iris_first
4×5 DataFrame. Omitted printing of 1 columns
| Row | SepalLength | SepalWidth | PetalLength | PetalWidth |
|     | Float64     | Float64    | Float64     | Float64    |
|-----|-------------|------------|-------------|------------|
| 1   | 5.1         | 3.5        | 1.4         | 0.2        |
| 2   | 4.9         | 3.0        | 1.4         | 0.2        |
| 3   | 4.7         | 3.2        | 1.3         | 0.2        |
| 4   | 4.6         | 3.1        | 1.5         | 0.2        |

julia> iris_first.status = ["fresh", missing, "fresh", "wilted
"]
4-element Array{Union{Missing, String},1}:
 "fresh"
 missing
 "fresh"
 "wilted"

julia> iris_first.LivingDay = [6, 2, missing, 14]
4-element Array{Union{Missing, Int64},1}:
  6
  2
   missing
 14

julia> iris_first
4×7 DataFrame. Omitted printing of 3 columns
| Row | SepalLength | SepalWidth | PetalLength | PetalWidth |
|     | Float64     | Float64    | Float64     | Float64    |
|-----|-------------|------------|-------------|------------|
| 1   | 5.1         | 3.5        | 1.4         | 0.2        |
| 2   | 4.9         | 3.0        | 1.4         | 0.2        |
| 3   | 4.7         | 3.2        | 1.3         | 0.2        |
| 4   | 4.6         | 3.1        | 1.5         | 0.2        |
```

Tab.14: Daten aufbereiten: Data Frames

Bedeutung	In R	In Julia

Data Frames	data.frame(Var1, Var2, …)	DataFrame(Var1, Var2, …)
Data Frame Dimension	dim()	size()
Namen von Vektoren/Listen-Elemente abfragen	names()	names()
Zeilen-/Variablenname in Data Frame	• rownames() colnames(), names()	names()
Überblick über den Data Frame	summary()	describe()
Zugriff auf Data Frame (die ersten oder letzten Zeilen)	• head() • tail()	• head(), first() • taill(), last()
Data Frame-Zugriff über die Zeilen	*Daten*[i,]	*Daten*[i, :]
Zugriff auf Variablen in data.frame 1. Listenzugriff mit Variablennamen 2. Matrixzugriff mit Variablennamen 3. Matrixzugriff mit bek. Variablenindex 4. Matrixzugriff mit unbek. Variablenindex	1. *Daten*$*Variable* 2. *Daten*[, "*Variable*"] 3. *Daten*[, *Zahl*] 4. *Daten*[*Daten*$*Variable* == *Ausprägung*]	1. *Daten.Variable* 2. *Daten*[:, :*Variable*], *Daten*[:, "*Variablenname*"] 3. *Daten*[:, *Zahl*] 4. Daten[*Daten.Variable* .== *Ausprägung*]
Data Frame sortieren	*Daten*[order(*Var*),]	sort!(*Daten*, :*Var)*

3.4.3. Behandlung von fehlenden Werten

Häufig kommt es vor, dass in dem Datensatz fehlende Werte enthalten sind. Wenn man mit Objekten rechnen möchte oder Funktionen darauf anwenden möchte, die fehlende Werte beinhalten, dann erhält man als Output wieder den fehlenden Wert, d.h. in R wird „NA" ausgegeben und in Julia „missing". Damit R und Julia ein qualitatives Ergebnis liefern können, kann man die Daten erst nach den fehlenden Werten prüfen und diese dann rauswerfen. Mit dem Befehl „is.na()" in R bzw. mit „ismissing()" in Julia kann nach diesen fehlenden Werten gesucht werden (Abb.79).

Abb.79: nach fehlenden Werten prüfen

In R

```
> iris_first$LivingDay
[1]  6  2 NA 14
> sum(iris_first$LivingDay)
[1] NA
>
> # nach fehlende Werten prüfen
> is.na(iris_first$LivingDay)
[1] FALSE FALSE  TRUE FALSE
```

In Julia

```
julia> iris_first.LivingDay
4-element Array{Union{Missing, Int64},1}:
  6
  2
   missing
 14

julia> sum(iris_first.LivingDay)
missing

julia> ismissing.(iris_first.LivingDay)
4-element BitArray{1}:
 0
 0
 1
 0
```

Die fehlenden Werte kann man dann in R auf drei verschiedenen Arten entfernen, die entweder die Funktionen „!is.na()“, „complete.cases()“ oder „na.omit()“ verwenden. In Julia entfernt man die fehlenden Werte in Variablen mit „collect(skipmissing())“ und in Datensätzen mit „dropmissing()“ (Abb.80).

Abb.80: fehlende Werte entfernen

In R

```
> # nur aus den Variablen entfernen
> LivingDay_neu <- iris_first$LivingDay[!is.na(iris_first$LivingDay)]
> LivingDay_neu <- iris_first$LivingDay[complete.cases(iris_first$LivingDay)]
> LivingDay_neu <- na.omit(iris_first$LivingDay)
> LivingDay_neu
[1]  6  2 14
attr(,"na.action")
[1] 3
attr(,"class")
[1] "omit"
> sum(LivingDay_neu)
[1] 22
>
> # aus dem ganzen Datensatz entfernen
> iris_first[complete.cases(iris_first),]
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species status LivingDay
1          5.1         3.5          1.4         0.2  setosa  fresh         6
4          4.6         3.1          1.5         0.2  setosa wilted        14
> na.omit(iris_first)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species status LivingDay
1          5.1         3.5          1.4         0.2  setosa  fresh         6
4          4.6         3.1          1.5         0.2  setosa wilted        14
>
```

In Julia

```
julia> # nur aus der Variable entfernen

julia> LivingDay_neu = collect(skipmissing(iris_first.LivingDay))
3-element Array{Int64,1}:
  6
  2
 14

julia> sum(LivingDay_neu)
22

julia> # alternativ

julia> sum(skipmissing(iris_first.LivingDay))
22

julia> # aus dem ganzen Datensatz entfernen

julia> dropmissing(iris_first)
2×7 DataFrame
| Row | SepalLength | SepalWidth | PetalLength | PetalWidth | Species | status | LivingDay |
|     | Float64     | Float64    | Float64     | Float64    | Cat…    | String | Int64     |
|-----|-------------|------------|-------------|------------|---------|--------|-----------|
| 1   | 5.1         | 3.5        | 1.4         | 0.2        | setosa  | fresh  | 6         |
| 2   | 4.6         | 3.1        | 1.5         | 0.2        | setosa  | wilted | 14        |
```

Tab.15: Behandlung von fehlenden Werten

Bedeutung	In R	In Julia
Daten auf fehlende Werte prüfen	is.na(x)	ismissing()
Fehlende Werte entfernen • In Variblen • In Data Frames	na.omit()	• collect(skipmissing()) • dropmissing()

3.5. Daten explorieren

In der deskriptiven Analyse stellen die statistischen Kennzahlen und die graphische Darstellung der Daten einen wichtigen Teilbereich der Statistik dar. Mit diesen Maßzahlen lassen sich komplexe Sachverhalte übersichtlich darstellen. Beide Programmiersprachen stellen für die Berechnung aller gängigen statistischen Kennwerte eigene Funktionen bereit, die man auf die Daten anwenden kann, vorausgesetzt, diese sind in Vektoren gespeichert. In diesem Unterkapitel werden ausgewählte Lage- und Streuungsparameter sowie die Extremwerte angeschaut. In Julia muss zur Berechnung dieser Kennzahlen das Paket „Statistics“ geladen werden (Abb.81).

3.5.1. Lageparameter

Das arithmetische Mittel, auch empirischer Mittelwert genannt, eines Vektors lässt sich in beiden Programmen mit der Funktion „mean()“ berechnen. Diese Zahl gibt Auskunft über die zentrale Lage einer Verteilung der Daten. Der Median, also der Wert der die Stichprobedaten in genau zwei Hälften teilt, lässt sich mit der Funktion „median()“ berechnen (Abb.81).

3.5.2. Extremwerte

Zur Bestimmung der Extremwerte können das Minimum und das Maximum ausgerechnet werden. In R kann mit „range()“ sowohl der kleinste als auch der größte Wert ausgegeben werden. Mit „min()“ und „max()“ werden die Extremwerte separat ausgegeben. In Julia wird das Minimum und das Maximum mit „minimum()“ und „maximum()“ berechnet. Zu beachten ist, dass in Julia auch die Funktionen „min()“ und „max()“ existieren, jedoch sind sie nicht mit „minimum()“ und „maximum()“ gleich zusetzen. Außerdem kann in R mit der Funktion „summary()“ die wichtigsten Kennzahlen auf einen Blick ausgeben werden. Dazu zählen das Minimum, das 25%-Quartil, der Median, der Mittelwert, das 75%-Quartil und das Maximum. Eine ähnliche Funktion existiert auch in Julia, nämlich „describe()“. Dieser kann jedoch nur auf Datensätzen angewendet werden (Abb.81).

3.5.3. Streuungsparameter

Mit „var()“ lassen sich in beiden Sprachen die empirische Stichprobenvarianz berechnen. Dies ist ein Maß für die Streuung der Wahrscheinlichkeitsdichte um ihren Schwerpunkt, d.h. sie gibt die Größe der Abweichung zum Mittelwert wider. Die Standardabweichung dagegen ist die mittlere Abweichung zum Mittelwert und lässt sich in R mit „sd()“ berechnen und in Julia mit „std()“ (Abb.81).

Abb.81: statistische Kennzahlen berechnen

In R

```
> iris_first
  Sepal.Length Sepal.Width Petal.Length
1          5.1         3.5          1.4
2          4.9         3.0          1.4
3          4.7         3.2          1.3
4          4.6         3.1          1.5
  Petal.Width Species status LivingDay
1         0.2  setosa  fresh         6
2         0.2  setosa   <NA>         2
3         0.2  setosa  fresh        NA
4         0.2  setosa wilted        14
>
> # Mittelwert
> mean(iris_first$Sepal.Length)
[1] 4.825
>
> # Median
> median(iris_first$Sepal.Length)
[1] 4.8
>
> # Extremwerte
> min(iris_first$Sepal.Length)
[1] 4.6
> max(iris_first$Sepal.Length)
[1] 5.1
> summary(iris_first$Sepal.Length)
   Min. 1st Qu.  Median    Mean 3rd Qu.
  4.600   4.675   4.800   4.825   4.950
   Max.
  5.100
>
> # Varianz
> var(iris_first$Sepal.Length)
[1] 0.04916667
>
> # Standardabweichhung
> sd(iris_first$Sepal.Length)
[1] 0.2217356
>
> # Korrelation (solte eigentlich nicht funk
tionieren, da nur eine Variable)
> cor(iris_first$Sepal.Length)
Error in cor(iris_first$Sepal.Length) :
  'x' und 'y' oder ein matrixähnliches 'x' m
üssen angegeben werden
```

In Julia

```
julia> iris_first
4×7 DataFrame. Omitted printing of 5 columns
| Row | SepalLength | SepalWidth |
|     | Float64     | Float64    |
|-----|-------------|------------|
| 1   | 5.1         | 3.5        |
| 2   | 4.9         | 3.0        |
| 3   | 4.7         | 3.2        |
| 4   | 4.6         | 3.1        |

julia> using Statistics

julia> mean(iris_first.SepalLength)
4.824999999999999

julia> median(iris_first.SepalLength)
4.800000000000001

julia> minimum(iris_first.SepalLength)
4.6

julia> maximum(iris_first.SepalLength)
5.1

julia> describe(iris_first.SepalLength)
ERROR: MethodError: no method matching describe
(::Array{Float64,1})
Closest candidates are:
  describe(::AbstractDataFrame; cols) at C:\Use
rs\thuy0\.julia\packages\DataFrames\ht7zm\src\a

julia> var(iris_first.SepalLength)
0.04916666666666666

julia> std(iris_first.SepalLength)
0.22173557826083448

julia> cor(iris_first.SepalLength)
1.0
```

Tab.16: Daten explorieren

Bedeutung	In R	In Julia
Mittelwert	mean()	mean()
Median	median()	median()
Stichproben-Varianz	var()	var()
Stichproben-Standardabweichung	sd()	std()
Extremwerte	min(), max(), range()	minimum(), maximum()
Wichtige deskr. Kennzahlen	summary()	describe() (nur auf Datensätzen anwendbar!!!)
Spannweite	diff(range())	maximum() - minimum()

3.6. Daten visualisieren

3.6.1. Streudiagramme

Erste Zusammenhänge zwischen den Variablen kann man an verschiedenen Graphiken erkennen. Zum Beispiel könnte man einen Streudiagramm verwenden, um zu schauen, ob zwei stetige Variablen voneinander abhängen. Zur Erzeugung dieses Diagramms kann man die Befehle „plot()“ in R und „scatter()“ in Julia aus dem R-Base bzw. dem Julia-Base verwenden. Es fällt auf, dass der Aufbau dieser beiden Befehle recht ähnlich ist. Beim dem Streudiagramm von Julia wurde die Punktewolke automatisch mit Farbe erzeugt. In R müsste man dies mit dem Argument „col“ festlegen (Abb.82+83).

Abb.82: Streudiagramm erzeugen

In R

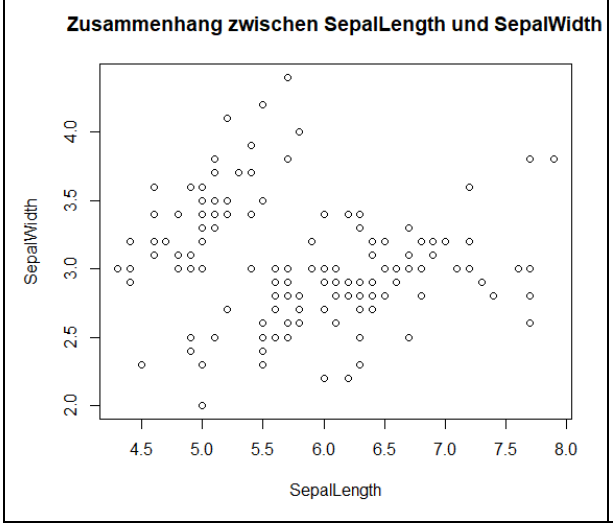
```
> plot(iris$Sepal.Length, iris$Sepal.width,
+      main="Zusammenhang zwischen SepalLength und SepalWidth",
+      xlab="SepalLength", ylab="SepalWidth")
```

In Julia

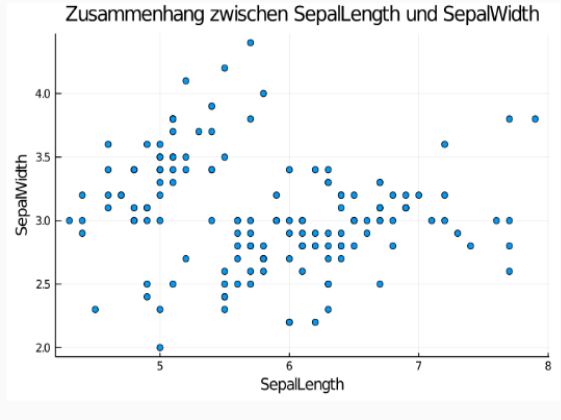
```
julia> scatter(iris.SepalLength, iris.SepalWidth,

       title="Zusammenhang zwischen SepalLength und SepalWidth",

         xlabel ="SepalLength", ylabel="SepalWidth", legend=false)
```

Abb.83: erzeugtes Streudiagramm

In R



In Julia



3.6.2. Liniendiagramme

Weiterhin könnte man mit einem Liniendiagramm sich anschauen, welche Werte einer metrischen Variablen, die Beobachtungen annehmen. Dies wurde beispielhaft an den Variablen „SepalLength“ und „SepalWidth“ durchgeführt. In beiden Sprachen ist es möglich. die zwei erzeugten Liniendiagramme in einem Fenster zu plotten. In R wird für den zweiten Plot der Befehl „lines()“ verwendet. In Julia kann dies erreicht werden, indem man nach dem regulären Plotbefehl ein „!“ hinzuschreibt, d.h. man

verwendet den Befehl zum Beispiel „plot!()“, um den zweiten Graphen in das gleiche Fenster zu plotten wie der erste Graph. Man erkennt, dass „SepalLength“ (siehe blaue Kurve) und „SepalWidth“ (siehe rote Kurve) nicht denselben Verlauf haben. „SepalLength“ hat über alle Beobachtungen hinweg immer größere Werte als „SepalWidth“. Es lässt sich anhand dieser Graphik erkennen, dass vermutlich kein Zusammenhang zwischen diesen Variablen existiert (Abb.84+85).

Abb.84: Liniendiagramm erzeugen

In R

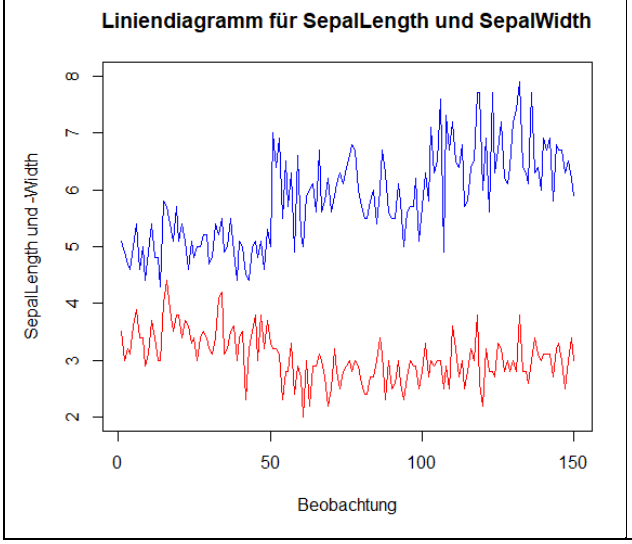
```
> plot(iris$Sepal.Length,
+      type="l", col="blue",
+      main="Liniendiagramm für SepalLength und Sepalwidth",
+      xlab="Beobachtung", ylab="SepalLength und -Width",
+      ylim=c(2,8))
> lines(iris$Sepal.Width, type="l", col="red")
```

In Julia

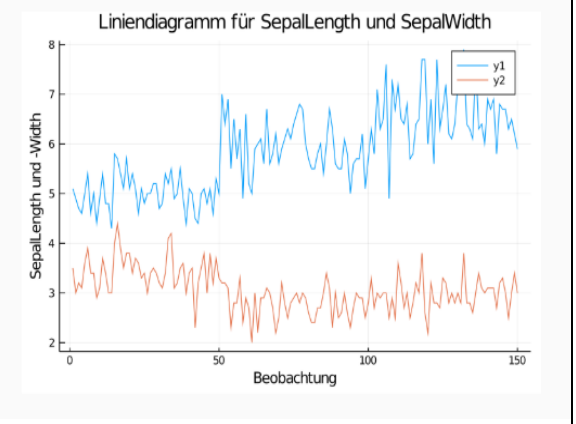
```
julia> plot(iris.SepalLength,

       title="Liniendiagramm für SepalLength und SepalWidth",

       xlabel="Beobachtung", ylabel="SepalLength und -Width")

julia> plot!(iris.SepalWidth)
```

Abb.85: erzeugte Liniendiagramme

In R



In Julia



3.6.3. Flächendiagramme

Ein gleiches Resultat erhält man, wenn man sich das Flächendiagramm für die Variablen „SepalLength“ und „SepalWidth“ plotten lässt. Im Grunde genommen liefert diese Graphik keine neue Information zu den beiden Variablen, denn die erzeugten Flächendiagramme ähneln den zuvor erzeugten Liniendiagrammen. Es wurde nur zusätzlich die Fläche unter den Kurven bis zur x-Achse hin ausgefüllt. Um ein Flächendiagramm zu erzeugen, kann man in R nicht mehr auf die Funktionen des R-Base zurückgreifen. Es muss zusätzlich das Paket „ggplot2“ geladen werden, mit dem man viel mehr

Möglichkeiten hat, Graphiken jeglicher Art zu erzeugen. Bevor man den Graphen erstellen kann, muss man die Daten in ein Long-Format überführen. Erst dann kann man mit „ggplot()“ ein Flächendiagramm über alle Beobachtungen hinweg erzeugen. Diese Aufgabe hat Julia eleganter gelöst, denn in der Julia-Base gibt es die Funktion „areaplot()“, mit dem man ohne viel Aufwand das Flächendiagramm plotten kann. Das Hinzufügen einer weiteren Graphik in dasselbe Fenster hat auch problemlos und flott funktioniert (Abb.86+87).

Abb.86: Flächendiagramm erzeugen

In R

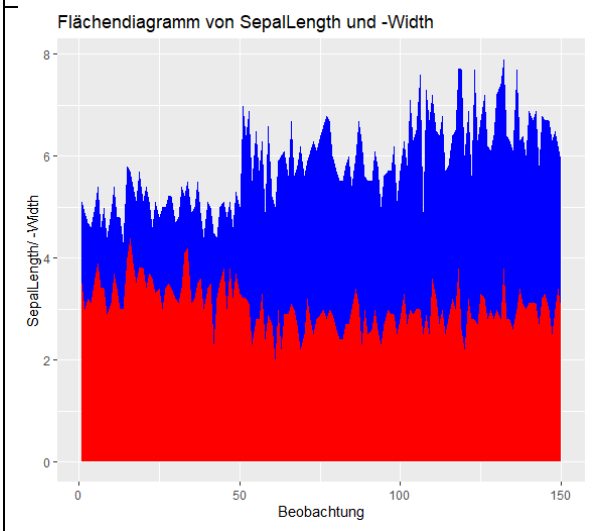
```
> library(ggplot2)
> library(dplyr)
> library(reshape2)
>
> iris_1 <- iris %>%
+   mutate(id = row_number()) %>%
+   select(id, Sepal.Length) %>%
+   melt(id.vars="id", measure.vars="Sepal.Length")
> iris_2 <- iris %>%
+   mutate(id = row_number()) %>%
+   select(id, Sepal.Width) %>%
+   melt(id.vars="id", measure.vars="Sepal.Width")
> ggplot() +
+   geom_area(aes(x=iris_1$id, y=iris_1$value), fill="blue") +
+   geom_area(aes(x=iris_2$id, y=iris_2$value), fill="red") +
+   ggtitle("Flächendiagramm von SepalLength und -Width") +
+   labs(x="Beobachtung", y="SepalLength/ -Width")
```

In Julia

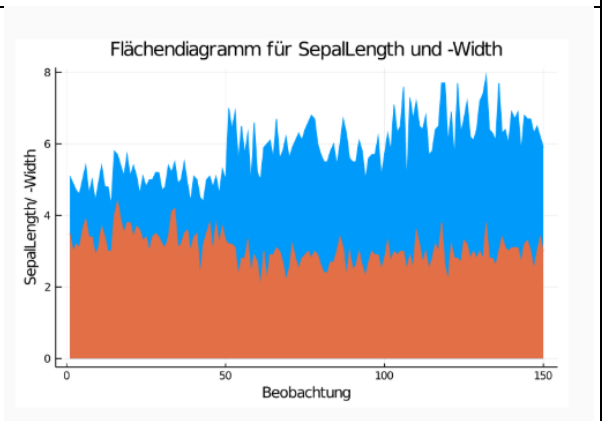
```
julia> areaplot(iris.SepalLength,

       title="Flächendiagramm für SepalLength und -Width",

       xlabel="Beobachtung", ylabel="SepalLength/ -Width", legend=false)

julia> areaplot!(iris.SepalWidth)
```

Abb.87: erstellte Boxplots

In R



In Julia



3.6.4. Balkendiagramme

Statt einem Flächendiagramm kann man sich auch ein Balkendiagramm mit denselben Informationsgehalt erzeugen lassen. Die Erstellung dieses Diagramms kann in beiden Sprachen schnell

erledigt werden. Man muss hierfür in R nur den Befehl „barplot()“ eingeben und in Julia den Befehl „bar()“. Auch hier ist das Plotten von mehreren Graphiken in einem Fenster in Julia möglich. In R dagegen klappt es nicht, auch wenn man den Befehl „lines()“ verwendet. Um in R zwei Balkendiagramme in einem Fenster zu plotten, müsste man mit dem Befehlen aus dem „ggplot2“-Paket arbeiten (Abb.88+89).

Abb.88: Balkendiagramm erzeugen

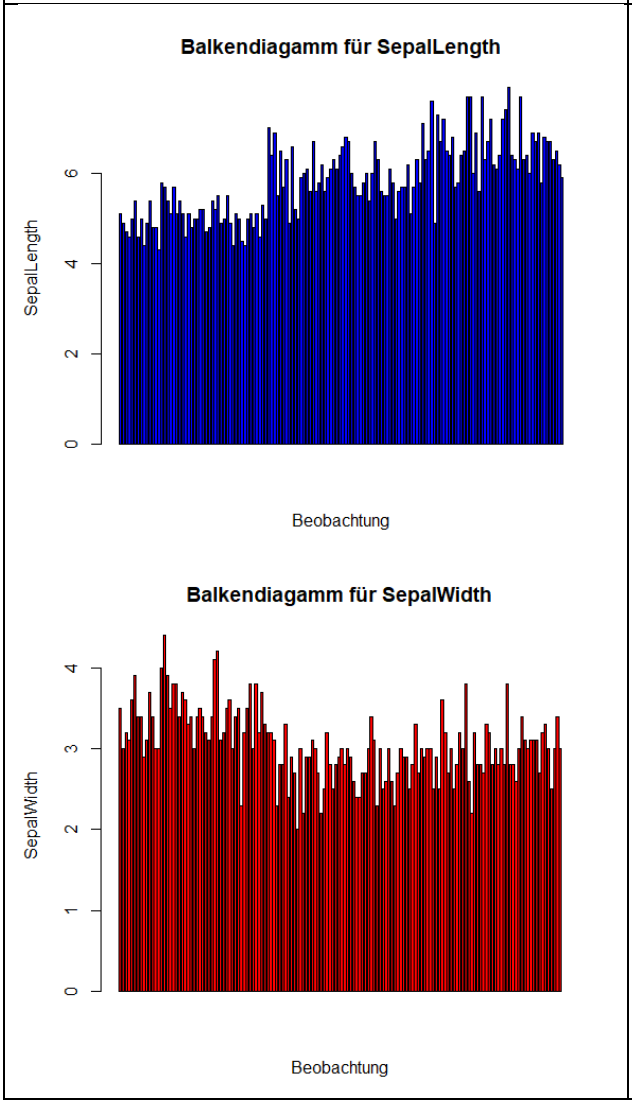
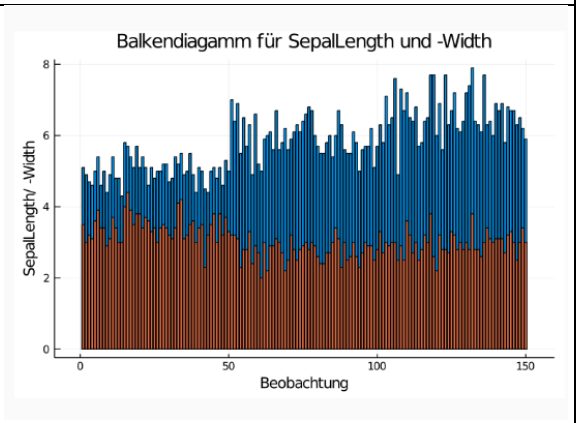
In R

```
barplot(iris$Sepal.Length,
        main="Balkendiagamm für SepalLength",
        xlab="Beobachtung", ylab="SepalLength",
        col="blue")
barplot(iris$Sepal.Width,
        main="Balkendiagamm für SepalWidth",
        xlab="Beobachtung", ylab="SepalWidth", col="red")
```

In Julia

```
julia> bar(iris.SepalLength,

       title="Balkendiagamm für SepalLength und -Width",

        ylabel="SepalLength/ -Width", xlabel="Beobachtung", legend=false)

julia> bar!(iris.SepalWidth)
```

Abb.89: erstellte Balkendiagramme

In R	In Julia
	

3.6.5. Histogramme und Kerndichteschätzer

Mit Histogrammen kann die Verteilung einer Variablen graphisch darstellt werden. Darüber hinaus kann eine stetige Kurve über das Histogramm gelegt werden. In R kann man solche Graphen mit den Befehlen „hist()“, „density()“ und „lines()“ erzeugen. Genauso einfach lassen sich diese Graphen in Julia erzeugen mit Hilfe von den Befehlen „histogram“ und „density!“ aus dem Zusatzpaket „StatsPlots“. In der erstellten Graphik sieht man, dass die stetige Kurve der Variable „SepalLength“ einer Glockenkurve ähnelt. Daraus könnte man vermuten, dass „SepalLength“ eine normalverteilte Zufallsgröße ist (Abb.90+91).

Abb.90: Histogramm und Kerndichteschätzer erzeugen

In R

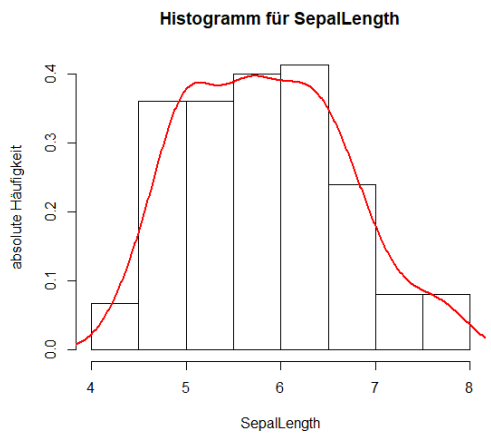
```
> hist(iris$Sepal.Length, prob=T,
+      main="Histogramm für SepalLength",
+      xlab="SepalLength", ylab="absolute Häufigkeit")
> lines(density(iris$Sepal.Length), col="red", lwd=2)
```

In Julia

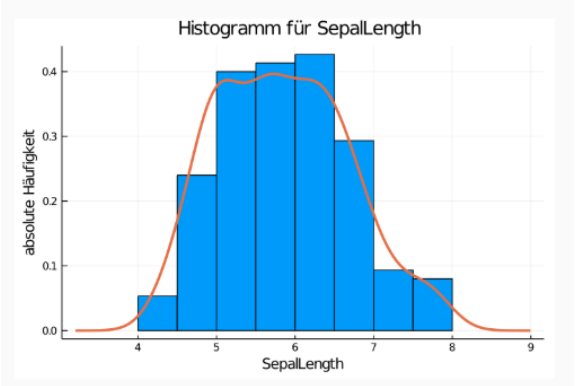
```
julia> using StatsPlots

julia> histogram(iris.SepalLength, normed=true,

       title="Histogramm für SepalLength ",

        xlabel="SepalLength",   ylabel="absolute Häufigkeit", legend=false)

julia> density!(iris.SepalLength, lw=3)
```

Abb.91: erstellte Histogramme und Kerndichteschätzer

In R



In Julia



3.6.6. Boxplots

Als letztes soll noch verglichen werden, wie R und Julia Boxplots erstellen. In R lässt sich ein Boxplot gruppiert nach den Kategorien über den Befehl „boxplot()“ erzeugen und in Julia über den Befehl „groupedboxplot()“. In Julia muss hierfür das Zusatzpaket „StatsPlots“ geladen. In den Graphen sieht man, dass die Mittelwerte der verschiedenen Spezien der Iris unterschiedlich sind. Daraus könnte man sich erschließen, dass die „Species“ einen möglichen Einfluss auf die „SepalLength“ hat (Abb.92+93).

Abb.92: Boxplot erzeugen

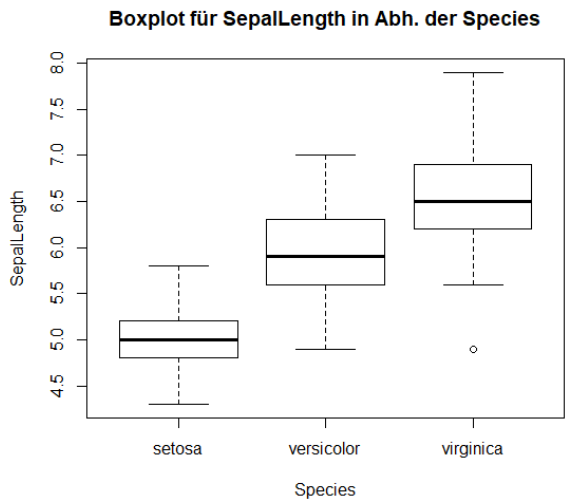
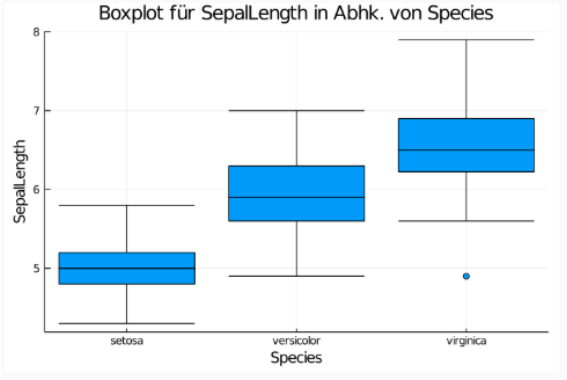
In R

```
> boxplot(iris$Sepal.Length ~ iris$Species,
+         main="Boxplot für SepalLength in Abh. der Species",
+         xlab="Species", ylab="SepalLength")
```

In Julia

```
julia> groupedboxplot(iris.Species, iris.SepalLength,

       title="Boxplot für SepalLength in Abhk. von Species",

         xlabel="Species", ylabel="SepalLength", legend=false)
```

Abb.93: erstellte Boxplots

In R	In Julia
	

Tab.17: Daten visualisieren (Graphiken)		
Bedeutung	In R	In Julia
Basis-Graphik	plot()	plot()
Streudiagramm	points()	scatter
Weitere Graphen im gleichen Fenster erzeugen	lines()	plot!(), scatter!(), bar!(), …
Histogramm	hist()	histogram()
Boxplot	boxplot()	boxplot()
Barplot	barplot()	bar()

3.7. Daten modellieren: Regressionsmodelle aufstellen

Mit der Regressionsanalyse wird der Zusammenhang zwischen einer oder mehreren unabhängigen Variablen und einer abhängigen Variable erfasst. In diesem Unterkapitel soll verglichen werden, wie die Programmiersprachen R und Julia ausgewählte Regressionen durchführen.

3.7.1. Einfache lineare Regression

Wie man vorhin schon in den Boxplots sehen konnte, lässt sich ein Zusammenhang zwischen den „Species“ und den „SepalLength“ vermuten. Um zu sehen, ob die Vermutung stimmt, kann man eine Regressionsanalyse durchführen. Hierfür eignet sich die einfache lineare Regression, mit der man den linearen Zusammenhang zwischen einer metrischen abhängigen Variable Y und einer unabhängigen Variable X ermitteln kann. In R wird eine lineare Regression mit dem Befehlen „lm()“ und „summary()“ durchgeführt. In Julia muss man zunächst das Zusatzpaket „GLM“ laden. Danach kann man ebenfalls über den Befehl „lm()“ eine lineare Regression durchführen. In beiden Sprachen kann mit dem Befehl „coef()“ die geschätzten Modellparameter aus dem Output extrahiert werden. Die

Modellgüte kann in R am Determinationskoeffizient R^2 (Multiple R-squared) im Output abgelesen werden. In Julia dagegen kann man sich diesen Wert mit dem Befehl „r2()“ ausgegeben werden. Fehlende Werte werden in beiden Sprachen bei der Bestimmung der Schätzwerte ignoriert. Es werden also nur Beobachtungen berücksichtigt, bei denen kein fehlender Wert vorkommt. In Julia können die fehlenden Werte nur ignoriert werden, wenn die verwendeten Variablen aus einem Datensatz stammen. Die Schätzung eines Regressionsmodells mit selbsterzeugten Variablen kann in Julia nicht durchgeführt werden. Sollte man es dennoch versuchen, so wird Julia eine Fehlermeldung ausgeben (Abb.94).

Abb.94: lineares Modell aufstellen

In R

```
> LiMo <- lm(formula=Sepal.Length ~ Species, data=iris)
> summary(LiMo)

Call:
lm(formula = Sepal.Length ~ Species, data = iris)

Residuals:
    Min      1Q  Median      3Q     Max
-1.6880 -0.3285 -0.0060  0.3120  1.3120

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)         5.0060     0.0728  68.762  < 2e-16 ***
Speciesversicolor   0.9300     0.1030   9.033 8.77e-16 ***
Speciesvirginica    1.5820     0.1030  15.366  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.5148 on 147 degrees of freedom
Multiple R-squared:  0.6187,    Adjusted R-squared:  0.6135
F-statistic: 119.3 on 2 and 147 DF,  p-value: < 2.2e-16

>
> coef(LiMo)
      (Intercept) Speciesversicolor  Speciesvirginica
            5.006             0.930             1.582
```

In Julia

```
julia> using GLM

julia> LiMo =  lm(@formula(SepalLength ~ Species), iris)
StatsModels.DataFrameRegressionModel{LinearModel{LmResp{Array{

Formula: SepalLength ~ 1 + Species

Coefficients:
──────────────────────────────────────────────────────────────────
                    Estimate  Std.Error   t value  Pr(>|t|)
──────────────────────────────────────────────────────────────────
(Intercept)            5.006  0.0728022  68.7616     <1e-99
Species: versicolor     0.93  0.102958    9.03282    <1e-15
Species: virginica     1.582  0.102958   15.3655     <1e-31
──────────────────────────────────────────────────────────────────

julia> coef(LiMo)
3-element Array{Float64,1}:
 5.006
 0.9299999999999985
 1.5820000000000007

julia> r2(LiMo)
0.6187057307384871
```

3.7.2. Multiple lineare Regression

In der multiplen linearen Regression kann man den linearen Zusammenhang zwischen einer abhängigen Variable Y und mehreren unabhängigen Variablen erfassen. In R, aber auch in Julia kann man diese Art von Regression ebenfalls mit der Funktion „lm()“ durchführen. Es muss hierfür nur die Modellgleichung und der Datensatz übergeben werden, d.h. folgender Ausdruck „lm(y~x1+…+xp, data=D*aten*)” muss in R eingegeben werden und in Julia wird der Ausdruck „lm(@formula(y ~ x1+…+xp), Daten)“ eingegeben werden. Die geschätzten Modellparameter lassen sich ebenfalls wie in der einfachen lineraren Regression mit dem Befehl „coef()“ ausgeben. In Julia kann dann wieder mit „r2()“ der Determinationskoeffizient R^2 ausgegeben werden

3.7.3. Logistische Regression

Auch die logistische Regression, bei dem der Zusammenhang zwischen einer binären abhängigen Variable und den unabhängigen Variablen ermittelt wird, kann in den beiden Programmiersprachen durchgeführt werden. In R muss dafür der Ausdruck „glm(formula=*Modellformel*, family=binomial(link=“logit“), data=*Datensatz*)“ eingegeben werden und in Julia der Ausdruck „glm(@formula(*Modellformel*), shuttle, Binomial(), LogitLink())“. Die Ausgabe der exponierten Koeffizienten erhält man in beiden Sprachen über den Befehl „exp(coef(*glm-Modell*))“. In R kann die multinominale Regression mit dem Ausdruck „vglm(*Modellformel*, family=multinomial(refLevel=1), data=*Datensatz*)“ durchgeführt und die ordinale Regression mit dem Ausdruck „vglm(*Modellformel*, family=propodds, data=*Datensatz*)“.

3.7.4. Multinominale und ordinale Regression

Soll der Zusammenhang zwischen einer kategorialen abhängigen Variable und den unabhängigen Einflussvariablen ermittelt werden, dann kann eine multinominale oder eine ordinale Regression durchgeführt werden. Die multinominale Regression eignet sich dabei für nominale Zielvariablen und die ordinale Regression eignet sich für ordinale Zielvariablen. In R wird die multinominale Regression mit dem Ausdruck „vglm(*Modellformel*, family=multinomial(refLevel=1), data=*Datensatz*)“ und die ordinal Regression mit dem Ausdruck „vglm(*Modellformel*, family=propodds, data=*Datensatz*)“ durchgeführt. In Julia kann die multinominale Regression noch nicht durchgeführt, weil entsprechende Befehle dazu noch fehlen. Aber die ordinale Regression kann mit dem Befehl „polr(@formula(y~x1+…+xp),Daten)“ aus dem Zusatzpaket „OrdinalMultinomialModels“ durchgeführt werden.

Tab.18: Daten modellieren

Bedeutung	In R	In Julia
Einfache lineare Regression	lm(*y~x*, data=*Daten*)	lm(@formula(*y ~ x*), *Daten*)

Multiple lineare Regression	lm(*y~x1*+…+*xp*, data=*Daten*)	lm(@formula(*y ~ x1*+…+*xp*), Daten)
Koeffizienten ausgeben	coef(*lin. Modell*)	coef(*lin. Modell*)
Logit-Modell	glm(formula=*Modellformel*, family=binomial(link="logit"), data=*Datensatz*)	glm(@formula(*Modellformel*), shuttle, Binomial(), LogitLink())
Exponierte Koeffizienten ausgeben	exp(coef(*glm-Modell*))	exp(coef(*glm-Modell*))
Multinomiale Regression	vglm(*Modellformel*, family=multinomial(refLevel=1), data=*Datensatz*)	
Ordinale Regression	vglm(*Modellformel*, family=propodds, data=*Datensatz*)	polr(@formula(y~x1+…+xp), Daten)
Odds Ratios ausgeben	exp(coef(*vlgm-Modell*))	exp(coef(*vlgm-Modell*))

3.8. Daten testen

3.7.1. Varianzanalyse

Die Varianzanalyse ("Analysis of Variance", kurz: ANOVA) testet, ob sich die Mittelwerte mehrerer unabhängiger Gruppen unterscheiden, die durch eine kategoriale unabhängige Variable definiert werden. In R wird der Test über „aov()" aufgerufen und in Julia, wird er mit der Funktion „anova()" aus dem Paket „ANOVA" durchgeführt (Abb.95).

Abb.95: Varianzanalyse

In R

```
> summary(aov(LiMo))
             Df Sum Sq Mean Sq F value Pr(>F)
Species       2  63.21  31.606   119.3 <2e-16 ***
Residuals   147  38.96   0.265
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

In Julia

```
julia> using ANOVA

julia> LiMo
StatsModels.DataFrameRegressionModel{LinearModel{LmResp{Array{Float64,1}},DensePr

Formula: SepalLength ~ 1 + Species

Coefficients:
──────────────────────────────────────────────────────────────────────────
                    Estimate  Std.Error   t value  Pr(>|t|)
──────────────────────────────────────────────────────────────────────────
(Intercept)            5.006  0.0728022  68.7616     <1e-99
Species: versicolor    0.93   0.102958    9.03282    <1e-15
Species: virginica     1.582  0.102958   15.3655     <1e-31
──────────────────────────────────────────────────────────────────────────

julia> anova(LiMo)
2×6 DataFrame
| Row | Source     | DF       | SS       | MSS      | F        | p            |
|     | String     | Abstrac… | Abstrac… | Abstrac… | Abstrac… | AbstractFl… |
|-----|------------|----------|----------|----------|----------|--------------|
| 1   | Species    | 2.0      | 63.2121  | 31.6061  | 119.265  | 1.66967e-31  |
| 2   | Residuals  | 147.0    | 38.9562  | 0.265008 | 0.0      | 0.0          |
2×6 DataFrame

julia> | Source     | DF       | SS       | MSS      | F        | p            |
|     | String     | Abstrac… | Abstrac… | Abstrac… | Abstrac… | AbstractFl… |
|-----|------------|----------|----------|----------|----------|--------------|
| 1   | Species    | 2.0      | 63.2121  | 31.6061  | 119.265  | 1.66967e-31  |
| 2   | Residuals  | 147.0    | 38.9562  | 0.265008 | 0.0      | 0.0          |
```

3.8.2. F-Test

Mit dem F-Test kann die Varianzhomogenität überprüft werden, also ob sich die Varianzen zweier normalverteilten Variablen gleich sind. In R wird Test über die Funktion „var.test()“ aufgerufen und in Julia über „ftest()“ aus dem Paket GLM.

3.8.3. t-Test

Der t-Test ist ein Verfahren zur Prüfung von Mittelwertunterschieden. Man unterscheidet dabei zwischen einem Einstichproben-t-Test und einem Zweistichproben-t-Test. Mit dem Einstichproben-t-Test wird geprüft, ob der Mittelwert eines beliebigen Merkmals einer Stichprobe dem Mittelwert einer Grundgesamtheit gleicht. Der Zweistichproben-t-Test prüft dagegen den Mittelwertsunterschied zweier Gruppen. In R werden beide t-Test-Verfahren mit der Funktion „t.test()“ durchgeführt. In Julia wird der Einstichproben-t-Test mit der Funktion „UnequalVarianceTTest()“ aus dem Zusatzpaket „HypothesisTests“ durchgeführt und der Zweistichproben-t-Test wird mit der Funktion „EqualVarianceTTest()“ aus demselben Paket durchgeführt.

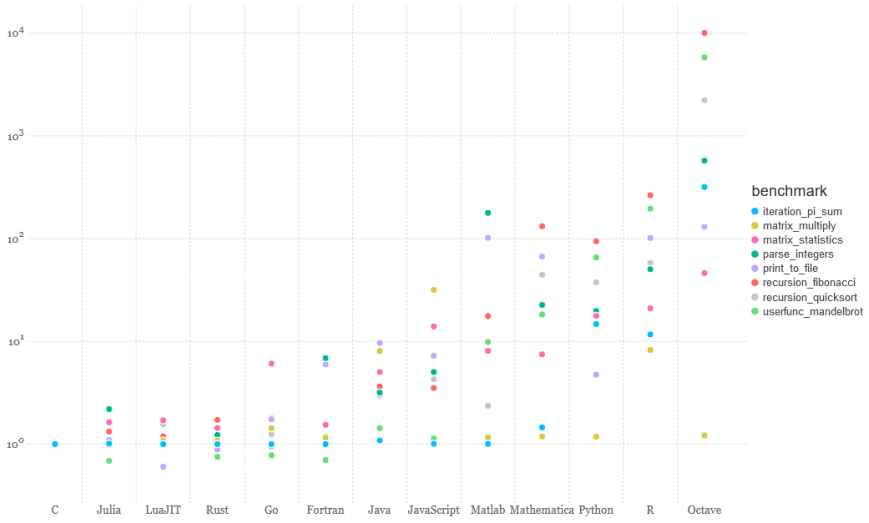
Tab.19: Daten testen		
Bedeutung	In R	In Julia
Einfache Varianzanalyse (ANOVA)	summary(aov(y~x, data=Daten))	anova(fit(LinearModel, @formula(y ~ x), Daten)
t-Test 1. 1 Stichprobe 2. 2 Stichproben	1. t.test(*Var*, *MW*) 2. t.test(Var_1, Var_2, var.equal=TRUE)	1. UnequalVarianceTTest(*Var*, *MW*) 2. EqualVarianceTTest(Var_1, Var_2)
F-Test	var.test(Var_1, Var_2,,)	ftest(Var_1, Var_2)

3.9. Sonstiges

3.9.1. Schnelligkeit

Julia´s größter Vorteil gegenüber R ist seine Geschwindigkeit und seine Performance. Laut dem Benchmark[10], das auf der Julia-Seite angeboten wird, ist Julia hinsichtlich seiner Geschwindigkeit fast auf demselben Level wie die Programmiersprache C (Abb.96). So war Julia für einige Befehle, die getestet wurden, zum Beispiel für die Ausführung einer Matrixmultiplikation mehr als doppelt so schnell im Vergleich zu R.

Abb.96. Ergebnisse eines Micro-Benchmarks



3.9.2. Häufigkeit der Nutzung und Anwendungsbereiche

Laut einer Umfrage[11] der Julia Computing company benutzen die meisten Julia-Anwender die Sprache zu 56% häufig (Abb.97) und das vor allem im Bereich der Statistik, des Data Sciences, des Engineering und des Machine learnings (Abb.98). Auch die Sprache R findet insbesondere in den Gebieten der Statistik

---

10. https://julialang.org/benchmarks/

11. https://julialang.org/images/2019-julia-user-developer-survey.pdf

und Data Science Anwendung, da sie genau dafür entwickelt wurde. Aber mittlerweile kann R in nahezu jeder Branche verwendet werden, aufgrund der Breite des unterstützten Methodenspektrums. Stark verbreitet und intensiv genutzt wird die Sprache an Universitäten, im Forschungs- und im Pharmabereich [12].

Abb.97: Häufigkeit der Nutzung von Julia unter den Julia Anwendern (Quelle: https://julialang.org/images/2019-julia-user-developer-survey.pdf)

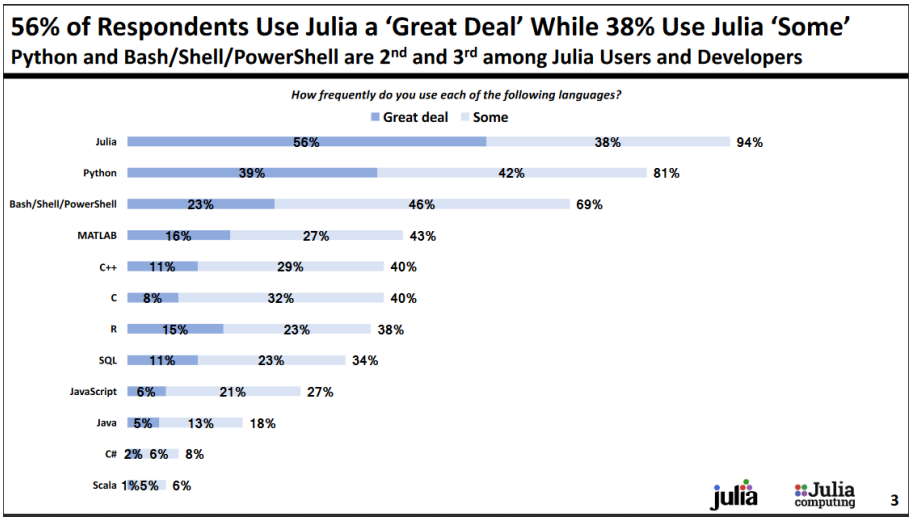
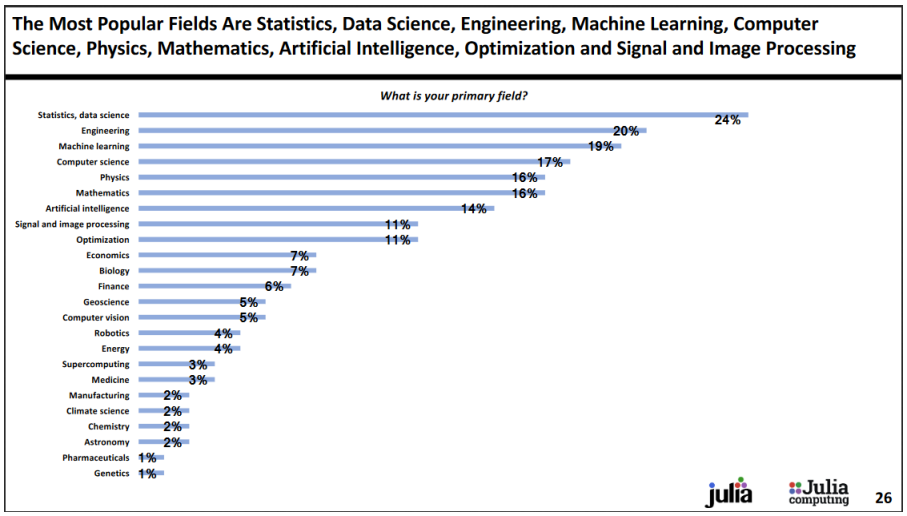


Abb.98: Julia`s Anwendungsbereiche



3.9.3. Syntax

Die Syntax in Julia ist mathematikfreundlich, da die Syntax für mathematische Berechnungen sich an den Matheformeln, die man noch aus der Schule kennt, orientiert. Somit ist die Sprache vor allem für Einsteiger und all diejenigen geeignet, die noch keine Erfahrungen mit dem Programmieren haben. R´s Syntax orientiert sich wie in der Programmvorstellung schon erwähnt, an der Programmiersprache S und hat große Ähnlichkeiten mit der Syntax von C. Sie ist sehr einfach und intuitiv gestaltet, aber nicht

fehlertolerant. Damit eignet sich R zwar auch für Einsteiger, jedoch muss man bereit sein, sich ausgiebig mit der Sprache auseinanderzusetzen und eine gewisse Zeit darin investieren, um sich in die Sprache einzuüben.

3.9.4. Programmieraufwand

Wie man im Laufe der Arbeit anhand der Anwendungsbeispiele sehen kann, ist der Programmieraufwand in Julia etwas geringer als in R. Zum Beispiel konnte man bei der Erstellung von Funktionen, Schleifen und Verzweigungen die Klammern um die Bedingungen und den Anweisungen komplett weglassen. Auch konnte man bei der Erstellung des Flächendiagramms sehen, dass mit weniger Schreibaufwand ein ähnlich gutes Resultat erzielt werden konnte wie in R.

4. Fazit und Ausblick

Wie man an den bisherigen Vergleichen sehen konnte, steht Julia R in nichts nach. Nahezu alles, was R konnte, hat Julia auch meistern können. Julia konnte vor allem mit weniger Programmieraufwand immer ein vergleichbar gutes Ergebnis wie R erzielen. Zum Beispiel wurden bei den Funktionen, den Schleifen und den Verzweigungen die Klammern immer alle Klammern wegelassen. Dadurch konnte man nicht nur schneller programmieren, sondern der Befehl war sogar viel einfacher zu lesen und sah deutlich übersichtlicher aus ohne den ganzen Klammern. Sogar für spezielle Matrizen gibt es in Julia Funktionen, mit denen diese Matrizen schnell erzeugt werden konnten. Bei der Erstellung von Graphiken konnte man sehen, dass Julia die Plots immer automatisch mit Farbe erzeugt hat. Wurde beispielsweise ein zweiter Graph in das gleiche Fenster geplottet, so hat dieser zweite Graph immer automatisch eine andere Farbe angenommen als der erste Graph. In R dagegen, musste man immer die Farben manuell einstellen. Außerdem konnte man in R nicht immer zwei Graphen in einem Fenster plotten, wie das bei den Balkendiagrammen der Fall war. Für Julia war das nie ein Problem gewesen. Positiv auffallend bei Julia war auch, dass ihre Syntax sich meistens an die Schreibweise der Matheformeln orientiert hat. Dies hat den Vorteil, dass auch Einsteiger, die noch keine Erfahrungen im Programmieren haben, dennoch mit Julia arbeiten können. Im Output von Julia wurde immer die Datenstruktur der Objekte mitausgeben, so musste man nie die Struktur abfragen. Das ist natürlich ein Vorteil, da man dann dadurch weniger Befehle eingeben muss. Nicht so erfreulich dagegen ist, dass viele wichtige Funktionen nicht in der Julia-Base enthalten sind. So musste man schon des Öfteren Zusatzpakete laden, um bestimmte Aufgaben erst lösen zu können. Einige Aufgaben konnte Julia nicht lösen. Dazu gehört zum Beispiel die Durchführung einer multinominalen Regression oder die Erstellung einer relativen Häufigkeitstafel. Für diese Aufgaben hat Julia noch keine Befehle beziehungsweise Pakete entwickelt. Das könnte daran liegen, dass Julia eine noch relativ junge Sprache ist und sich noch in der Entwicklung befindet. Ein weiterer negativer Punkt an Julia ist, dass zum Beispiel die Funktionsnamen zur Berechnung der verschiedenen Verteilungen immer relativ lang sind, während diese für R immer kurz und prägnant waren.

Zusammenfassend kann man sagen, dass Julias Stärken vor allem in der numerischen Berechnung liegen, denn in diesem Bereich performt Julia enorm schnell, aufgrund ihrer einfachen und intuitiven Syntax. Auch kleine Datenanalysen kann Julia problemlos durchführen, insbesondere das Plotten von Graphen fällt Julia nicht schwer. Dennoch ist Julia noch sehr unreif, da bestimmte Aufgaben im statistischen Bereich noch nicht von ihr gelöst werden können. Obwohl Julia noch relativ jung ist, kann sie R in manchen Bereichen dennoch übertreffen. Zudem Julia hat in nur kurzer Zeit es geschafft sich einen Platz nahe einem Programmiergiganten wie R zu ergattern und sollte deshalb definitiv im Auge behalten werden.

Anhang

Datensatz iris

- https://archive.ics.uci.edu/ml/datasets/iris
- https://gist.github.com/curran/a08a1080b88344b0c8a7

R-Pakete

- H. Wickham. ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York, 2016.
- Hadley Wickham, Romain François, Lionel Henry and Kirill Müller (2019). dplyr: A Grammar of Data Manipulation. R package version 0.8.3. https://CRAN.R-project.org/package=dplyr
- Hadley Wickham (2007). Reshaping Data with the reshape Package. Journal of Statistical Software, 21(12), 1-20. URL http://www.jstatsoft.org/v21/i12/.
- Revelle, W. (2019) psych: Procedures for Personality and Psychological Research, Northwestern University, Evanston, Illinois, USA, https://CRAN.R-project.org/package=psych Version = 1.9.12.

Julia-Pakete

- Tom Breloff, Plots.jl (https://github.com/JuliaPlots/Plots.jl)
- Random.jl (https://docs.julialang.org/en/v1/stdlib/Random/, https://github.com/JuliaLang/julia/blob/master/stdlib/Random/docs/src/index.md)
- StatsPlots.jl (https://github.com/JuliaPlots/StatsPlots.jl)
- Distributions.jl (https://github.com/JuliaStats/Distributions.jl)
- LinearAlgebra.jl (https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/)
- DataFrames.jl, (https://juliadata.github.io/DataFrames.jl/stable/man/getting_started/, https://github.com/JuliaData/DataFrames.jl)
- FreqTables.jl (https://github.com/nalimilan/FreqTables.jl)
- CategoricalArray.jl (https://github.com/JuliaData/CategoricalArrays.jl)
- ANOVA.jl (https://github.com/marcpabst/ANOVA.jl, https://github.com/ZaneMuir/ANOVA.jl)
- GLM.jl (https://githubcom/JuliaStats/GLM.jl)
- RDatasets.jl (https://github.com/JuliaStats/RDatasets.jl)
- StatsBase (https://github.com/JuliaStats/StatsBase.jl, https://juliastats.org/StatsBase.jl/stable/)

Abbildungsverzeichnis

Tabellenverzeichnis

Quellenverzeichnis

[1] Ben Lauwens, Allen B. Downey. Think Julia – How To Think Like A Computer Scientist. O'Reilly Media. United States of America. 2019

[2] Balbaert Ivo, Sengupta Avik, Sherrington Malcolm. Julia - High Performance Programming. Packt Publishing, 2016 [https://learning.oreilly.com/library/view/julia-high-performance/9781787125704/]

[3] The Julia Language – The Julia Project. April 15,2020. [https://raw.githubusercontent.com/JuliaLang/docs.julialang.org/assets/julia-1.5.0-DEV.pdf]

[4] Introducing Julia. 17 January 2020, at 09:40. [https://en.wikibooks.org/wiki/Introducing_Julia]

[5] Günter Faes. Datenanalyse mit Julia – Einstieg in die Datenanalyse mit der Programmiersprache Julia.BoD – Books on Demand. Norderstedt. 2019

[6] Emmanuel Paradis. R for Beginners. 2005 [https://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf]

[7] R (Programmiersprache). 2020-07-26 10:25. [https://de.wikipedia.org/wiki/R_(Programmiersprache)]

[8] DanielWollschläger. Grundlagen der Datenanalyse mit R – Eine anwenungsorientierte Einführung. 4.Auflage. Springer-Spektrum GmbH Deutschland. 2017

[9] Jürgen Groß. Grundlegende Statistik mit R – Eine anwendungsorientierte Einführung in die Verwendung der Statistik Softwar R. 1. Auflage. Vieweg+Teubner Verlag | Springer Fachmedien Wiesbaden GmbH. 2010

[10] Maike Luhmann. R für Einsteiger – Eine Einführung in die Statistiksoftware für die Sozialwissenschaften. 3. Auflage. Beltz Verlag. Weinheim, Basel. 2013

[11] Christian Ewald. R, Python & Julia in Data Science: Ein Vergleich. Eoda. 2020. 16.09.2019 [https://www.eoda.de/wissen/blog/r-python-julia-data-science-2019]

[12] Amit Ghosh. Statistik-Software: R, Python, SAS, SPSS und STATA im Vergleich. INWT Statistics. 25.04.2018 13:30 [https://www.inwt-statistics.de/blog-artikel-lesen/Statistik-Software-R_Python_SAS_SPSS_STATA_im_Vergleich.html]

[13] Wikipedia. Liste von Statistik-Software. 28. Juni 2020 um 19:44 [https://de.wikipedia.org/wiki/Liste_von_Statistik-Software]

[14] junolab.org [http://docs.junolab.org/stable/]

[15] wanderinformatiker.at [https://www.wanderinformatiker.at/unipages/general/iris.html]

Selbstständigkeitserklärung

Hiermit erkläre ich, Pham Thi Thuy, dass ich meine Bachelorarbeit selbstständig verfasst und keine anderen Quellen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe.

Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

München, den 28.07.2020

……

Unterschrift