Modul Statistische Aspekte der Analyse molekularbiologischer und genetischer Daten

R-Blatt 1: Grundlagen von R

Janne Pott

WS 2021/22

In dieser Übung wird in die Statistik-Software R eingeführt. R ist eine frei verfügbare Software, die unter Windows, MacOS X und Linux zur Verfügung steht. Zusammen mit der einfachen grafischen Oberfläche RStudio können damit auch Einsteiger recht komfortabel grafische Darstellungen und statistische Auswertungen von Daten erstellen. Insbesondere bei der Verarbeitung von Genom-Daten ist es eines der Standardwerkzeuge. Informationen dazu findet man bei Bioconductor.

In dieser ersten Übung sollen Sie zunächst einige Grundfunktionen von R kennenlernen bzw. wiederholen. Im Anschluss zu jedem Kapitel gibt ein paar Aufgaben, die Sie bitte lösen sollen.

Erstellen Sie abschließend ein PDF- oder HTML-Output, indem Sie das Skript knittern.

Hinweise zur Syntax von RMarkdown finden Sie hier bzw. hier.

Grundsätzlich ist es sinnvoll, zunächst das Arbeitsverzeichnis festzulegen (Speicherort des Skripts) und anzugeben wo relevante **R-Pakete** liegen (ggf. abh. von R Version oder Schreibrechten!), zum Beispiel:

```
setwd("D:/Lehre/WS2021_Statistik/uebung/RUebungen/")
.libPaths("C:/Program Files/R/R-3.6.0/library")

# Hier sollen alle notwendigen Pakete stehen die im Laufe der Uebung genutzt werden
library(knitr)
library(foreach)
library(doParallel)
library(data.table)
library(readxl)
library(lubridate)
library(MASS)
library(nlme)
library(ggplot2)
library(meta)
library(qqman)
#library(ivpack)
#library(MendelianRandomization)

knitr::opts_chunk$set(echo = TRUE)
```

R als Taschenrechner

Beispiele

Die folgenden Beispiele illustrieren, wie R als Taschenrechner genutzt werden kann: Sie geben einfach den Ausdruck ein, der ausgewertet werden soll und drücken die Eingabetaste.

```
27+34
```

```
## [1] 61
```

```
113*6-98
```

```
## [1] 580
```

```
2^6
```

```
## [1] 64
```

```
2**6
```

```
## [1] 64
```

```
67/9
```

```
## [1] 7.444444
```

```
(37.4+56)/5
```

```
## [1] 18.68
```

```
5/0
```

```
## [1] Inf
```

R gibt beim Dividieren durch 0 also keine Fehlermeldung aus sondern den Wert *Inf* für unendlich (engl.: infinite). Das entspricht nicht der üblichen mathematischen Konvention, ist aber auch nicht ganz sinnlos.

In R stehen viele Ihnen schon aus der Schule bekannte Funktionen zur Verfügung wie **sin**, **cos**, **tan**, **exp** (e-Funktion), **log** (natürlicher Logarithmus), **sqrt** (Quadtatwurzel) oder **factorial** (Fakultät) usw. Hier sind Beispiele:

```
pi
```

```
## [1] 3.141593
```

```
sin(90)
```

```
## [1] 0.8939967
```

```
sin(pi/2)
```

```
## [1] 1
```

```
factorial(5)
```

```
## [1] 120
```

```
exp(0)
```

```
## [1] 1
```

```
log(exp(1))
```

```
## [1] 1
```

```
log(-1)
```

```
## Warning in log(-1): NaNs wurden erzeugt
```

```
## [1] NaN
```

```
log10(10)
```

```
## [1] 1
```

```
abs(-7)
```

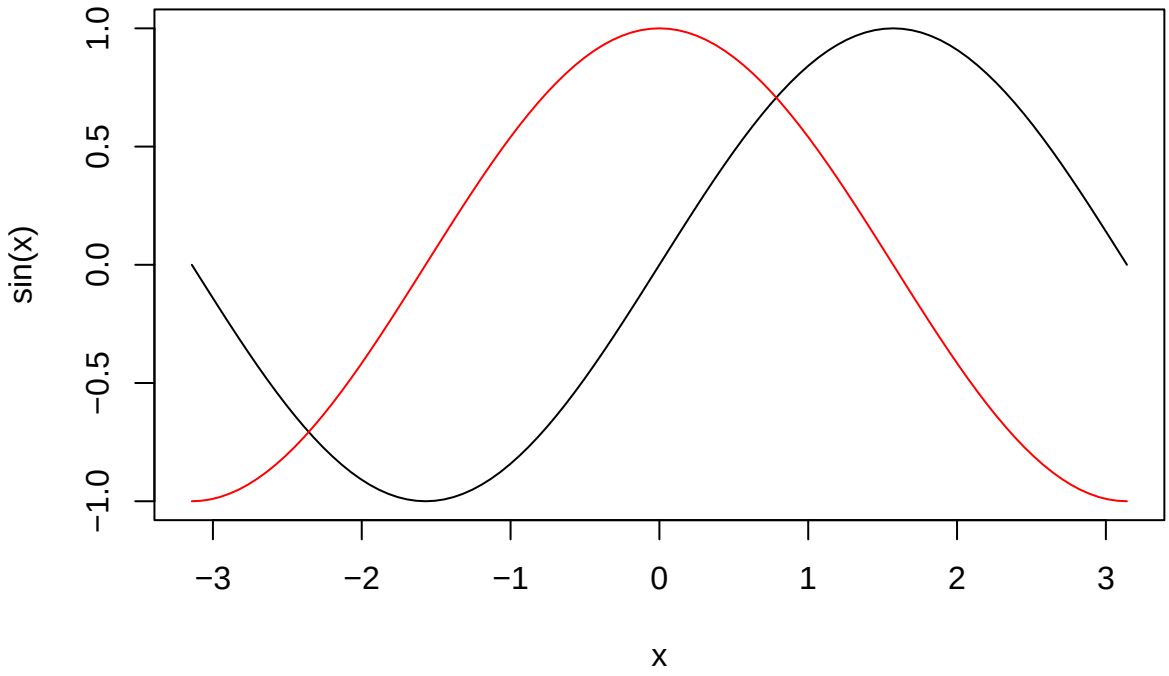
```
## [1] 7
```

In R werden die Winkel nicht in Grad, sondern in Bogenlänge eines Kreises mit Radius 1 gemessen. Da 360 Grad einem vollem Kreisumfang 2π entspricht, muss statt 90 Grad die Bogenlänge $\pi/2$ verwendet werden.

Da der Logarithmus nur für positive Argumente definiert ist, gibt R hier ein *NaN* (not a number) aus.

Zum Abschluss verschaffen wir uns noch einen ersten kleinen Eindruck von den graphischen Fähigkeiten von R.

```
curve(sin,-pi,pi)
curve(cos,add=TRUE,col="red")
```



Im Graphikfenster erkennen Sie die Sinuskurve von π bis π. Die Koordinatenachsen sind am Bildrand gezeichnet. Ohne den Zusatz *add=TRUE* bei der Cosinus-Funktion wäre ein ganz neues Bild gezeichnet worden.

Aufgaben

Berechnen Sie folgende Terme:

a) $|3^5 - 2^{10}|$

b) $sin(\frac{3}{4}\pi)$

c) $\frac{16!}{5!11!}$

d) $\sqrt{37-8} + \sqrt{11}$

e) $e^{-2.7}/0.1$

f) $2.3^8 + \ln(7.4) - \tan(0.3\pi)$

g) $\log_{10}(27)$

h) $\ln(\pi)$

i) $\ln(-1)$

Variablen und Folgen

Beispiele

Auch wenn es nicht der offiziellen R-Terminologie entspricht, so will ich doch Buchstaben (oder aus Buchstaben und anderen Zeichen zusammengesetzte Namen), denen man einen Wert zuweisen kann, als Variablen bezeichnen. Wenn man also das Ergebnis einer Rechnung einer Variablen zuweist, so steht es danach unter deren Namen zur Verfügung.

Die Zuweisung kann also entweder mit <- oder mit = erfolgen. Der darauf folgende Aufruf der Variablen veranlasst erst die Ausgabe des Werts am Bildschirm. Will man das Ergebnis einer Zuweisung sofort sehen, schreibt man die Zuweisung in Klammern:

```
x<-37/5
x
```

```
## [1] 7.4
```

```
y=3^3
y
```

```
## [1] 27
```

```
(y=3^3)
```

```
## [1] 27
```

Mit Variablen, denen Werte zugewiesen wurden, kann man rechnen:

```
y-x
```

```
## [1] 19.6
```

```
(z=5/0)
```

```
## [1] Inf
```

```
3*z
```

```
## [1] Inf
```

```
-z
```

```
## [1] -Inf
```

```
z-300
```

```
## [1] Inf
```

```
z+z
```

```
## [1] Inf
```

```
z-z
```

```
## [1] NaN
```

Im letzten Beispiel weigert sich R den Ausdruck auszuwerten und gibt *NaN* aus. Denken sie über den Umgang von R mit dem Wert unendlich etwas nach, insbesondere über die beiden letzten Ergebnisse!

Reguläre Folgen lassen sich in R sehr leicht erzeugen:

```
a=11:20
a
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
b=seq(5,14)
b
```

```
##  [1]  5  6  7  8  9 10 11 12 13 14
```

```
c=seq(1,10,2)
c
```

```
## [1] 1 3 5 7 9
```

```
d=seq(0,5,0.5)
d
```

```
##  [1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Mit a:b erzeugt man also eine Folge ganzer Zahlen von a bis b, während der Befehl *seq()* eine sehr flexible Folgenerzeugung ermöglicht: seq(a,b,d) erzeugt eine Folge von Zahlen mit Abstand d, die bei a beginnt und höchstens bis b läuft.

R kann nun mit Folgen sehr leicht rechnen, auch wenn dabei die korrekte mathematische Notation manchmal etwas arg strapaziert wird. Schauen Sie sich die Ergebnisse an und versuchen sie sich klar zu machen, was tatsächlich berechnet wird.

```
3*a
```

```
##  [1] 33 36 39 42 45 48 51 54 57 60
```

```
a^2
```

```
##  [1] 121 144 169 196 225 256 289 324 361 400
```

```
2^a
```

```
##  [1]    2048    4096    8192   16384   32768   65536  131072  262144  524288
## [10] 1048576
```

```
a*b
```

```
##  [1]  55  72  91 112 135 160 187 216 247 280
```

```
a*c
```

```
##  [1]  11  36  65  98 135  16  51  90 133 180
```

```
a*d
```

```
## Warning in a * d: Länge des längeren Objektes
##       ist kein Vielfaches der Länge des kürzeren Objektes
```

```
##  [1]  0  6 13 21 30 40 51 63 76 90 55
```

Eine nicht reguläre Folge kann man auf zwei Weisen erzeugen: mit der Funktion *c()* oder mit dem Befehl *scan()*, der Daten direkt von der Tastatur liest (ist im Markdown aber nicht praktisch)

```
u=c(1,7,-2,11,0.1)
u
```

```
## [1]  1.0  7.0 -2.0 11.0  0.1
```

```
#v=scan()
```

Aufgaben

Erzeugen Sie folgende $a_1, ..., a_{10}$:

a) $a_n = 3^n$

b) $a_n = e^{-n}$

c) $a_n = (1 + \frac{1}{n})^n$

d) $a_n = \sin(n\frac{\pi}{10})$

Funktionen

Beispiele

Viele mathematische und statistische Funktionen sind in R bereits fest installiert. Man kann aber auch Funktionen selbst definieren, wie man hier am Beispiel der Funktion $f(x) = \exp(-x^2/2)$ sieht:

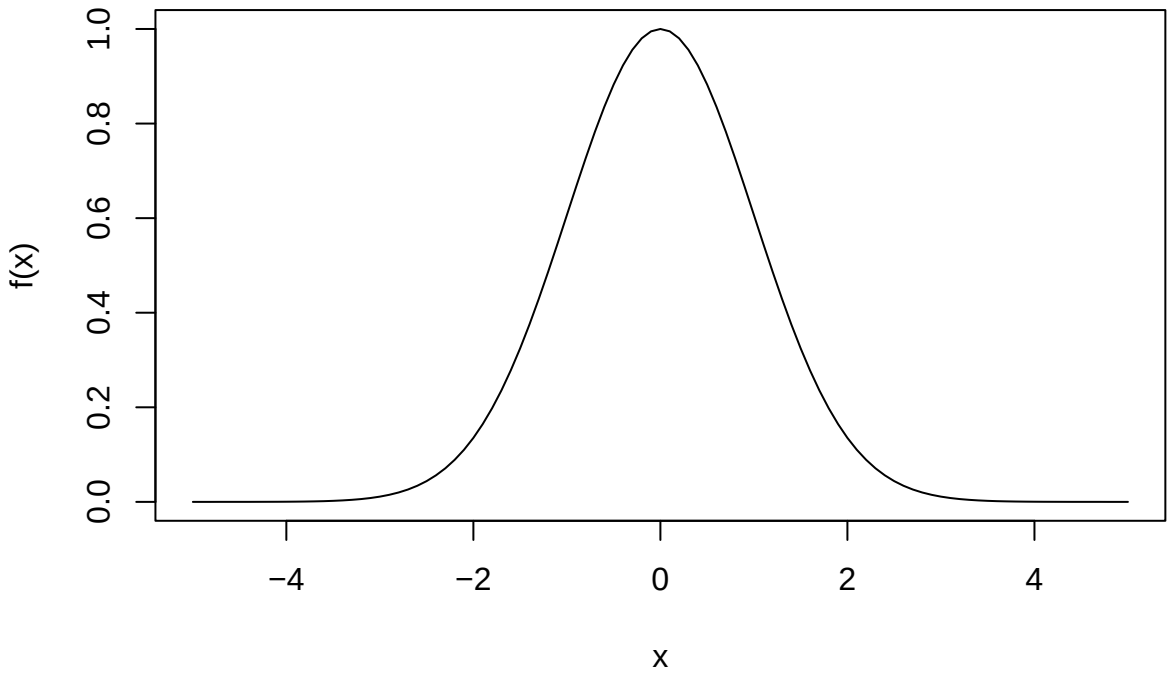
```
f=function(x){exp(-x^2/2)}
f(3)
```

```
## [1] 0.011109
```

```
f(d)
```

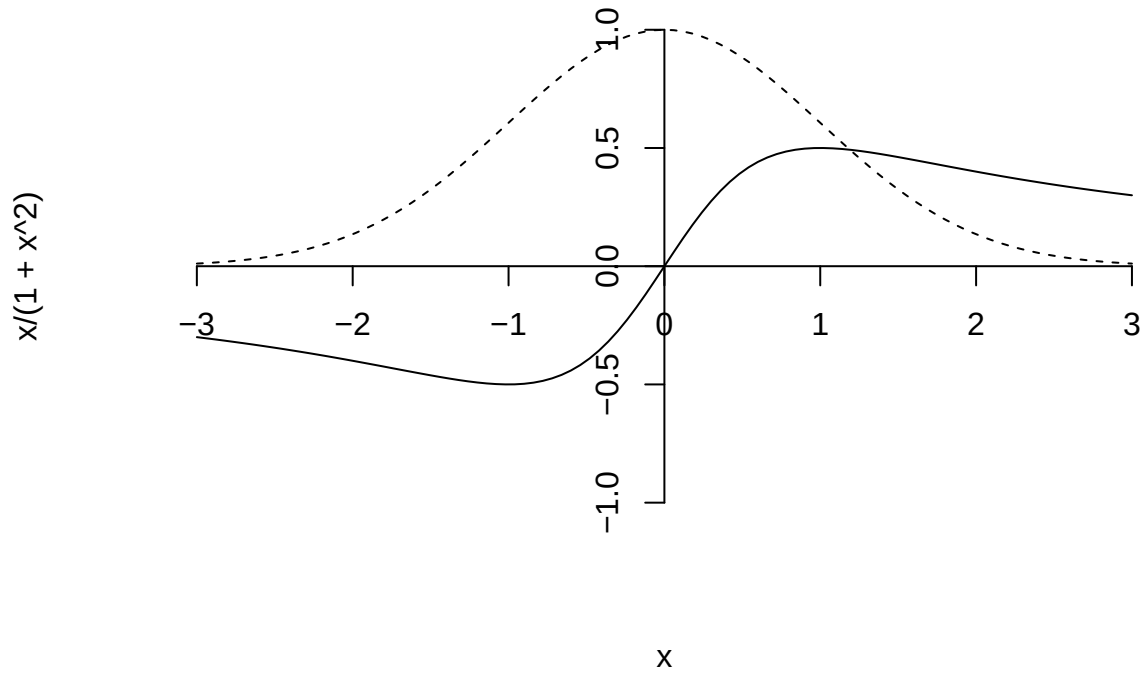
```
##  [1] 1.000000e+00 8.824969e-01 6.065307e-01 3.246525e-01 1.353353e-01
##  [6] 4.393693e-02 1.110900e-02 2.187491e-03 3.354626e-04 4.006530e-05
## [11] 3.726653e-06
```

```
curve(f,-5,5)
```



Man kann auch beliebige Funktionsterme direkt plotten:

```
curve(x/(1+x^2),-3,3,ylim=c(-1,1),axes=FALSE)
axis(1,pos=0)
axis(2,pos=0)
curve(f,add=TRUE,lty="dashed")
```



Um Minima und Maxima zu bestimmmen, kann man einfach *min()* und *which.min()* verwenden.

```
g=function(x){x/(1+x^2)}
X<-seq(-3,3,0.1)
Y<-g(X)
min(Y);which.min(Y)
```

```
## [1] -0.5
```

```
## [1] 21
```

```
max(Y);which.max(Y)
```

```
## [1] 0.5
```

```
## [1] 41
```

```
X[which.min(Y)];min(Y)
```

```
## [1] -1
```

```
## [1] -0.5
```

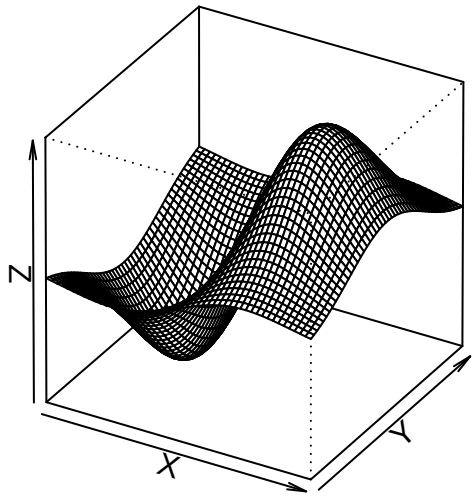
```
X[which.max(Y)];max(Y)
```

```
## [1] 1
```

```
## [1] 0.5
```

R kann auch mit Funktionen von mehreren Variablen umgehen:

```
h=function(x,y){x*exp(-(x^2+y^2)/2)}
X=Y=seq(-2,2,length=50)
Z=outer(X,Y,h)
persp(X,Y,Z,theta = 30,phi = 30,d=100)
```



Wir haben zuerst die Funktion $h(x, y) = x \exp(-(x^2 + y^2)/2)$ definiert und Gitter X und Y aus je 50 x- bzw. y-Werten zwischen -2 und 2 erzeugt. Mit der Funktion *outer()* wurde für jeder Kombination von X- und Y-Werten der Funktionswert von h berechnet und in der Matrix Z gespeichert. Anschließend wurden die Tripel aus X,Y, und Z-Werten dreidimensional geplottet (Funktionsgraph von h als Fläche über der x-y-Ebene). Theta und Phi legen der Blickwinkel fest und d bestimmt den Grad der perspektivischen Verzerrung.

Diese zunächst etwas kompliziert erscheinende Art, eine Funktion von zwei Variablen zu plotten, erklärt sich dadurch, dass die x, y- und z-Werte in der Statistik nicht als mathematische Funktion gegeben, sondern aus beobachteten Daten bestimmt werden, die dann als Vektoren oder Matrizen (wie X, Y und Z) zur Verfügung stehen.

Eine andere Darstellungsform benutzt Höhenlinien, evtl. noch unterstützt durch Farben (gleiche Farben repräsentieren gleiche Z-Werte).

```
image(X,Y,Z)
contour(X,Y,Z)
image(X,Y,Z);contour(X,Y,Z,add=T)
image(X,Y,Z,col=heat.colors(100));contour(X,Y,Z,add=T)
image(X,Y,Z,col=gray((30:100)/100));contour(X,Y,Z,add=T)
```

Aufgaben

a) Definieren Sie die Funktion $h(x) = \sin(\sqrt{x})$ und werten Sie sie an den Stellen 0, 0.1, 0.2, ..., 0.9, und 1 aus.

b) Definieren Sie die Funktionen $g_1(a,b,c) = \frac{a*b}{a*b+(1-c)*(1-a)}$ und $g_2(a,b,c) = \frac{c*(1-a)}{c*(1-a)+(1-b)*a}$ und werten Sie sie für $a \in [0,1]$, $b = 0.7$ und $c = 0.95$ aus.

c) Erstellen Sie einen Plot von beiden Funktionen g_1 und g_2 für $a \in [0,1]$, $b = 0.7$ und $c = 0.95$.

Vektoren und Matrizen

Beispiele

Vektoren sind Objekte, in denen einzelne Werte **mit gleichem Modus** zusammengefasst sind. Man kann auf einzelne Elemente eines Vektors zugreifen, oder eine mathematische Funktion auf einen Vektor anwenden - dann wird jede Komponente des Vektors durch die Funktion transformiert.

```
v=c(2.3,4.1,-0.5,2,-7)
w=seq(1.1,3.1,length=5)
is.vector(v)
```

```
## [1] TRUE
```

```
mode(v)
```

```
## [1] "numeric"
```

```
class(v)
```

```
## [1] "numeric"
```

```
v+w
```

```
## [1]  3.4  5.7  1.6  4.6 -3.9
```

```
2*v+3*w
```

```
## [1]  7.9 13.0  5.3 11.8 -4.7
```

```
c(v,w)
```

```
##  [1]  2.3  4.1 -0.5  2.0 -7.0  1.1  1.6  2.1  2.6  3.1
```

```
v[3]; v[2:3]; v[2:length(v)]; c(v,w)[4:9]
```

```
## [1] -0.5
```

```
## [1]  4.1 -0.5
```

```
## [1]  4.1 -0.5  2.0 -7.0
```

```
## [1]  2.0 -7.0  1.1  1.6  2.1  2.6
```

```
exp(w)
```

```
## [1]  3.004166  4.953032  8.166170 13.463738 22.197951
```

```
min(v);max(v);mean(v);median(v);sum(v);var(v)
```

```
## [1] -7
```

```
## [1] 4.1
```

```
## [1] 0.18
```

```
## [1] 2
```

```
## [1] 0.9
```

```
## [1] 18.797
```

```
summary(v)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   -7.00   -0.50    2.00    0.18    2.30    4.10
```

Matrizen lassen sich sehr flexibel erzeugen, indem man zunächst sämtliche Koeffizienten der Matrix als eine lange Folge bereitstellt und dann angibt, wieviele Zeilen oder Spalten die daraus zu formende Matrix haben soll.

```
(A=matrix(c(1,3,2,0,-2,5),nrow=2))
```

```
##      [,1] [,2] [,3]
## [1,]    1    2   -2
## [2,]    3    0    5
```

```
(B=matrix(c(-2,-1,3,0,2,4),nrow=2))
```

```
##      [,1] [,2] [,3]
## [1,]   -2    3    2
## [2,]   -1    0    4
```

```
(C=matrix(c(2,5,1,-3,-1,0,0,4,2,4,3,-2),nrow=3))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2   -3    0    4
## [2,]    5   -1    4    3
## [3,]    1    0    2   -2
```

```
(D=matrix(c(1,3,4,6,7,8),ncol=3))
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    3    6    8
```

```
(E=matrix(c(1,3,4,6,7,8),nrow=4))
```

```
## Warning in matrix(c(1, 3, 4, 6, 7, 8), nrow = 4): Datenlänge [6] ist kein Teiler
## oder Vielfaches der Anzahl der Zeilen [4]
```

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    3    8
## [3,]    4    1
## [4,]    6    3
```

```
(M=matrix(c(1,3,4,6,7,8),nrow=4,ncol=6))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    7    4    1    7    4
## [2,]    3    8    6    3    8    6
## [3,]    4    1    7    4    1    7
## [4,]    6    3    8    6    3    8
```

```
(N=matrix(c(w,v),ncol=2))
```

```
##      [,1] [,2]
## [1,]  1.1  2.3
## [2,]  1.6  4.1
## [3,]  2.1 -0.5
## [4,]  2.6  2.0
## [5,]  3.1 -7.0
```

```
max(N); which.max(N)
```

```
## [1] 4.1
```

```
## [1] 7
```

```
N[7]; N[2,2]
```

```
## [1] 4.1
```

```
## [1] 4.1
```

```
M
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    7    4    1    7    4
## [2,]    3    8    6    3    8    6
## [3,]    4    1    7    4    1    7
## [4,]    6    3    8    6    3    8
```

```
is.matrix(M)
```

```
## [1] TRUE
```

```
M[3,2]
```

```
## [1] 1
```

```
M[,c(2,4)]
```

```
##      [,1] [,2]
## [1,]    7    1
## [2,]    8    3
## [3,]    1    4
## [4,]    3    6
```

```
B
```

```
##      [,1] [,2] [,3]
## [1,]   -2    3    2
## [2,]   -1    0    4
```

```
t(B)
```

```
##      [,1] [,2]
## [1,]   -2   -1
## [2,]    3    0
## [3,]    2    4
```

```
A*B
```

```
##      [,1] [,2] [,3]
## [1,]   -2    6   -4
## [2,]   -3    0   20
```

Das ist **keine** Matrizenmultiplikation! Hier wurden zwei gleich große Matrizen einfach koeffizientenweise multipliziert. Das Zeichen für Matrizenmultiplikation ist %*%:

```
A%*%C
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   10   -5    4   14
## [2,]   11   -9   10    2
```

```
t(A)%*%B
```

```
##      [,1] [,2] [,3]
## [1,]   -5    3   14
## [2,]   -4    6    4
## [3,]   -1   -6   16
```

Aufgaben

a) Erzeugen Sie einen Vektor A mit den Quadratzahlen 1, 4, 9, ..., 400 als Einträgen.

b) Bilden Sie zwei Vektoren B und C aus den ersten bzw. letzten zehn Einträngen von A. Erzeugen Sie daraus einen Vektor D mit 50 Einträgen, in dem zunächst einmal die Elemente von A, zweimal die von C und einmal die von B auftreten.

c) Erzeugen Sie aus dem Vektor D die 10x5 Matrix M.

Schleifen

Beispiele

Die wichtigsten Programmierbefehle in R sind: **for**, **if** und **while**. Falls man in der Schleife nur einen Befehl ausführt, kann man die geschweiften Klammern auch weglassen.

```
for (i in 1:10){
  #i=1
  message("Working on Index ", i)
  if(i %% 2 == 0){ message("Index ",i," ist gerade!") }else cat("Index ",i," ist ungerade!\n")
}
```

```
## Working on Index 1

## Index  1  ist ungerade!

## Working on Index 2

## Index 2 ist gerade!

## Working on Index 3

## Index  3  ist ungerade!

## Working on Index 4

## Index 4 ist gerade!

## Working on Index 5

## Index  5  ist ungerade!

## Working on Index 6

## Index 6 ist gerade!

## Working on Index 7

## Index  7  ist ungerade!

## Working on Index 8

## Index 8 ist gerade!

## Working on Index 9

## Index  9  ist ungerade!

## Working on Index 10

## Index 10 ist gerade!
```

```
i=1
while(i<4){print(i);i=i+1;print(i)}
```

```
## [1] 1
## [1] 2
## [1] 2
## [1] 3
## [1] 3
## [1] 4
```

```
c("Ende",i)
```

```
## [1] "Ende" "4"
```

Um auch parallele Prozesse zu erlauben, wurde die Schleife **foreach** entwickelt, die ein leicht abgeänderte Syntax hat. In diesem Beispiel wird nur %do% verwendet, d.h. keine Parallelisierung. Stattdessen müsste man %dopar% verwenden, und vorher die Anzahl der zu verwendenden Kerne angeben

```
#setup parallel backend to use 8 processors
# cl<-makeCluster(8)
# registerDoParallel(cl)

dumTab<-foreach(i=1:10)%do%{
  # dumTab<-foreach(i=1:10)%dopar%{
  #i=1
  message("Working on Index ", i)
  if(i %% 2 == 0){
    dummy<-paste0("Index ",i," ist gerade!")
    }else dummy<-paste0("Index ",i," ist ungerade!")
  tab<-data.frame(Index=i,Comment=dummy)
  tab
}
```

```
## Working on Index 1

## Working on Index 2

## Working on Index 3

## Working on Index 4

## Working on Index 5

## Working on Index 6

## Working on Index 7

## Working on Index 8

## Working on Index 9

## Working on Index 10
```

```
dumTab<-rbindlist(dumTab)
dumTab
```

```
##     Index               Comment
##  1:     1 Index 1 ist ungerade!
##  2:     2   Index 2 ist gerade!
##  3:     3 Index 3 ist ungerade!
##  4:     4   Index 4 ist gerade!
##  5:     5 Index 5 ist ungerade!
##  6:     6   Index 6 ist gerade!
##  7:     7 Index 7 ist ungerade!
##  8:     8   Index 8 ist gerade!
##  9:     9 Index 9 ist ungerade!
## 10:    10  Index 10 ist gerade!
```

```
class(dumTab)
```

```
## [1] "data.table" "data.frame"
```

Aufgaben

Erstellen Sie eine Schleife in einer Schleife!

a) Erstellen Sie einen Vektor **iters** für Anzahl der Iterationen, beginnend bei 10, endend bei 100, und in 10er Schritten.

b) Erstellen Sie einen Outputvektor **times**, in dem die Zeit eingetragen werden soll.

c) Definieren Sie die erste *for*-Schleife von 1 bis zur Länge von **iters**, die

- sich die Anzahl der gewünschten Iterationen aus **iters** zieht
- die Zeitmessung startet (x=Sys.time())
- pro Iteration eine normalverteilte Zufallsvariable mit n=10000 Ziehungen erstellt (dummy=rnorm(1e5), zweite Schleife) und die Summary davon bestimmt (dummy2<-summary(dummy), entspricht Min., Max., Quantile)
- die Zeit in der Variablen **times** abspeichert

d) Plotten Sie **iters** gegen **times**!

Dateneingabe

Beispiele

Hier werden nur die gängigsten Dateitypen vorgestellt: .txt, .csv und .xlsx. CSV und TXT Dateien können recht einfach mit dem Befehl *read.table* in R eingelesen werden. Für Excel-Tabellen oder große TXT Dateien empfiehlt es sich, Funktionen aus dem Paket *data.table* und *readxl* zu nutzen.

WICHTIG: **data.table** nutzt automatisch alle vorhandenen Kerne (weil viel in parallelisierten Prozessen verwendet)! Daher lieber prüfen und auf Bedarf einschränken!

```
myTab1<-read.table("data/Beispiel1.csv",header=T)
myTab1
```

```
##   Laenge Gewicht
## 1   7.56   0.055
## 2  11.92   0.213
## 3  16.40   0.564
## 4  24.83   1.894
## 5  29.03   3.012
```

```
class(myTab1)
```

```
## [1] "data.frame"
```

```
myTab2<-read.table("data/Beispiel1.txt",header=T,sep="\t")
myTab2
```

```
##   Laenge Gewicht
## 1   7.56   0.055
## 2  11.92   0.213
## 3  16.40   0.564
## 4  24.83   1.894
## 5  29.03   3.012
```

```
class(myTab2)
```

```
## [1] "data.frame"
```

```
library(data.table)
getDTthreads()
```

```
## [1] 4
```

```
setDTthreads(1)
library(readxl)
myTab3<-data.table(read_excel("data/Beispiel1.xlsx",sheet = 1))
myTab3
```

```
##    Laenge Gewicht
## 1:   7.56   0.055
## 2:  11.92   0.213
## 3:  16.40   0.564
## 4:  24.83   1.894
## 5:  29.03   3.012
```

```
class(myTab3)
```

```
## [1] "data.table" "data.frame"
```

```
myTab4<-fread("data/Beispiel1.txt")
myTab4
```

```
##    Laenge Gewicht
## 1:   7.56   0.055
## 2:  11.92   0.213
## 3:  16.40   0.564
## 4:  24.83   1.894
## 5:  29.03   3.012
```

```
class(myTab4)
```

```
## [1] "data.table" "data.frame"
```

Es können auch Daten aus R Paketen geladen werden:

```
data("mtcars")
df <- mtcars
head(df)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Was ist der Unterschied zwischen data.frame und data.table?

Data Frame: Spaltenbasiert, man kann Zeilen und Spalten auswählen. Der Grundaufbau ist DF[i,j], wobei i die Zeile und j die Spalte angibt. Daher sind Weitere Funktionen recht umständlich zu definieren

```
id<-c("b","a","a","c","c","b")
val<-c(4,2,3,1,5,6)
X<-data.frame(id,val)
X
```

```
##   id val
## 1  b   4
## 2  a   2
## 3  a   3
## 4  c   1
## 5  c   5
## 6  b   6
```

```
X[X$id!="a",]
```

```
##   id val
## 1  b   4
## 4  c   1
## 5  c   5
## 6  b   6
```

```
X[,"val"]
```

```
## [1] 4 2 3 1 5 6
```

```
X[X$id!="a","val"]
```

```
## [1] 4 1 5 6
```

```
DF<-read.table("data/Beispiel2.txt",header = T)
DF
```

```
##   id code valA valB
## 1  1  abc  0.1   11
## 2  1  abc  0.6    7
## 3  1  abd  1.5    5
## 4  2  apq  0.9   10
## 5  2  apq  0.3   13
```

```
# Mit Werten aus den Spalten rechnen
sum(DF[DF$code != "abd", "valA"])
```

```
## [1] 1.9
```

```
# In Gruppen sortieren
aggregate(cbind(valA,valB) ~ id, DF[DF$code != "abd",],sum)
```

```
##   id valA valB
## 1  1  0.7   18
## 2  2  1.2   23
```

```
# Updates
DF[DF$code == "abd", "valA"] <- NA
```

Data Table: Spaltenbasiert, aber flexibler. Der Grundaufbau ist DT[i,j,by], wobei i die Zeile festlegt, j was zu tun ist und by die Gruppe definiert.

```
Y<-setDT(X)
Y[id!="a",mean(val)]
```

```
## [1] 4
```

```
DF[DF$code == "abd", "valA"] <- 1.5
DT<-setDT(DF)
DT[code!="abd",sum(valA)]
```

```
## [1] 1.9
```

```
DT[code!="abd",.(sum(valA),sum(valB)),id]
```

```
##    id  V1 V2
## 1:  1 0.7 18
## 2:  2 1.2 23
```

```
DT[code=="abd",valA:=NA]
DT[code=="abd",valA:=3.0]
DT
```

```
##    id code valA valB
## 1:  1  abc  0.1   11
## 2:  1  abc  0.6    7
## 3:  1  abd  3.0    5
## 4:  2  apq  0.9   10
## 5:  2  apq  0.3   13
```

```
B<-fread("data/Beispiel3.txt")
class(B)
```

```
## [1] "data.table" "data.frame"
```

```
DT[B, on=.(id,code),valA:=valA*mul]

DT[,lapply(.SD,sum),by=.(id,code),.SDcols="valA"]
```

```
##    id code valA
## 1:  1  abc  0.7
## 2:  1  abd  6.0
## 3:  2  apq  0.6
```

Aufgaben

a) Laden Sie den Datensatz *iris*.

b) Ändern Sie die Klasse von *data.frame* zu *data.table*.

c) Wie viele Einträge sind pro Spezies vorhanden?

d) Wie lang und breit sind im Mittel die Blätter pro Spezie? Nutzen Sie dazu die Funktion *lapply()*.

e) Definieren Sie eine neue Spalte als Produkt der Kelchblattlänge und -breite.

f) Wie groß ist die mittlere Differenz der Blattlänge (Kelch - Blüte) in der Spezies *setosa*?