Züchtungslehre - Einführung in R

Peter von Rohr

2016-09-16

Einführung

R (https://www.r-project.org/) ist ein sehr populäres System im Bereich der Datenanalyse. Ursprünglich wurde R von den Statistikern Ross Ihaka und Robert Gentlemen kreiert. Da R über die eigene Programmiersprache erweiterbar ist, wird es aktuell in sehr verschiedenen Gebieten eingesetzt. Das System wird als Open Source vertrieben und kann gratis für die gängigen Betriebssystem heruntergeladen werden. Eine Vielzahl von Dokumentationen zu R ist unter https://cran.r-project.org/manuals.html erhältlich. Unter https://cran.r-project.org/other-docs.html sind auch Anleitungen in Deutsch und vielen weiteren Sprachen erhältlich.

Dieses Dokument basiert auf einigen der oben genannten Quellen und versucht die wichtigsten Punkte von R für diese Vorlesung zusammenzufassen.

Voraussetzung für die Verwendung von R

R muss zuerst heruntergeladen und dann installiert werden. Unter den sogenannten `Comprehensive R Archive Network` (CRAN)-Mirror Webseiten ist R als ausführbares Programm für die Betriebssystem `Windows`, `Mac OSX` und `Linux` erhältlich.

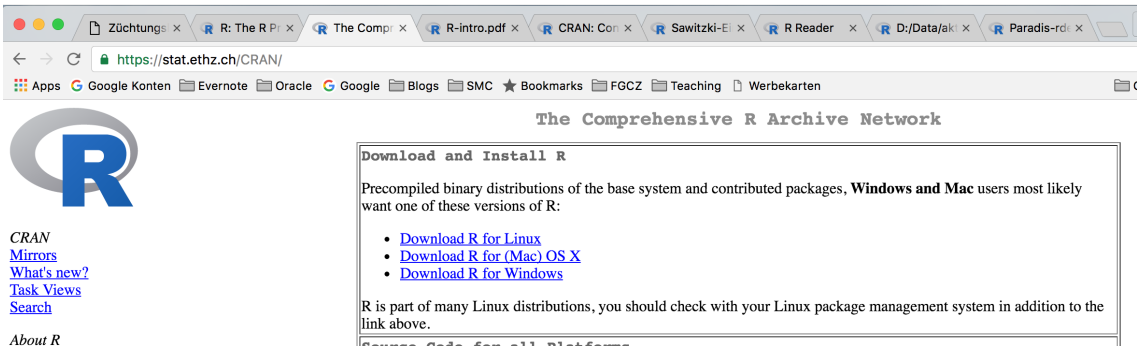


Figure 1: ScreenShotCRAN

Zusätzlich zu R empfielt es sich, die Entwicklungsumgebung RStudio (https://www.rstudio.com/) zu verwenden.

Interaktiver Modus

R kann wie ein Taschenrechner verwendet werden. Man spricht dann auch vom so genannten **interaktiven** Modus, d.h. die/der BenutzerIn gibt einen Befehl ein und bekommt eine Antwort zurück. In Rstudio werden die Befehle in das Fenster, welches mit dem Tab namens `Console` überschrieben ist, eingegeben. Beispiele für mögliche Eingaben im interaktiven Modus sind

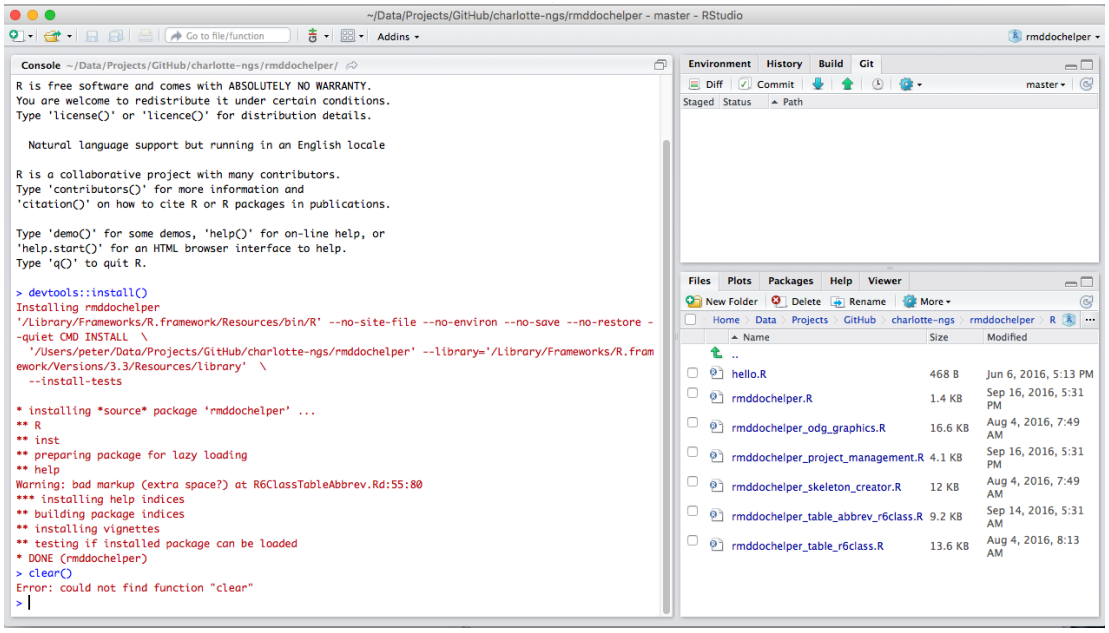


Figure 2: ScreenShotRstudio

```
43-15
```

```
## [1] 28
```

```
76/8
```

```
## [1] 9.5
```

R befolgt die in der Arithmetik üblichen Rechenregeln, so gilt beispielsweise, dass Klammern vor Punkt vor Strich

```
65 - 18 / 3
```

```
## [1] 59
```

```
(65 - 18) / 3
```

```
## [1] 15.66667
```

Arithmetische Operationen

Die folgende Tabelle enthält die Liste mit den in R verwendeten arithmetischen Operationen

Operator	Operation
+	Addition

Operator	Operation
−	Subtraktion
∗	Multiplikation
/	Division
^	Potenz
e	Zehnerpotenz

Logische Operationen

Abgesehen von arithmetischen Rechenoperationen lassen sich auch logische Operationen durchführen. Mithilfe dieser Operationen lassen sich Vergleiche machen oder Beziehungen überprüfen. Das Resultat eines solchen Vergleichs ist immer entweder `TRUE` (wahr) oder `FALSE` (falsch). (Im Abschnitt zu den Datentypen werden wir sehen, dass es für die Vergleichsresultaten einen speziellen Datentyp - den `BOOLEAN` Datentyp - gibt.) Die folgende Liste zeigt die Vergleichsoperatoren in R.

Operator	Operation
==	ist gleich
! =	ist ungleich
>	ist grösser
<=	ist kleiner gleich
&	und
|	oder

Die folgenden Beispiele zeigen, wie logische Vergleiche ausgewertet werden

```
3 == 3
```

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

```
3 == 4
```

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

```
3 != 4
```

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

```
3 <= 4 & 3 == 3
```

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

```
3 <= 4 | 3 == 4
```

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

```
(3 <= 4 | 3 == 3) & 3 == 4
```

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

Variablen - Objekte

Die Begriffe `Variablen` und `Objekte` werden hier als Synonyme verwendet. Das Rechnen im Taschenrechner-Modus ist sicher sehr nützlich, aber wir wollen auch in der Lage sein, bestimmte Grössen an speziellen Orten im Speicher abzulegen. Dazu verwenden wir Variablen. Damit wir besser verstehen, was bei der Verwendung bei Variablen passiert, schauen wir uns das so genannte `Von Neumann`-Modell eines Computers an.

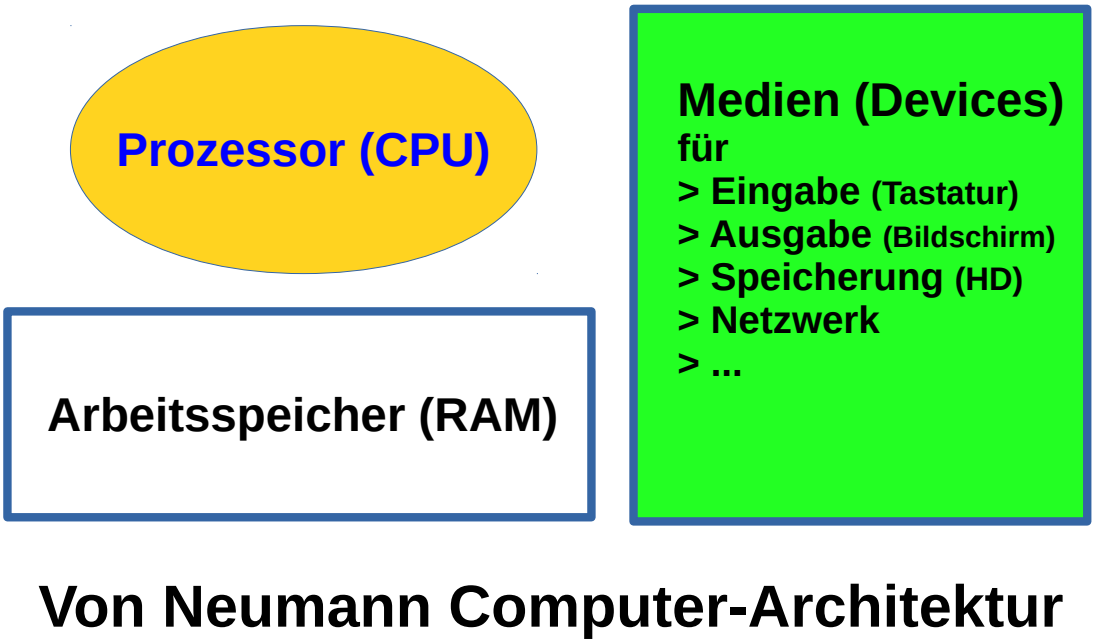
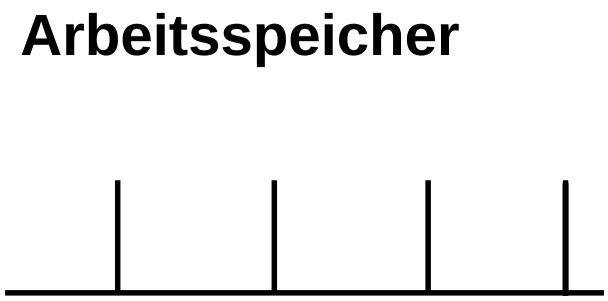


Figure 3: vonNeumannComputerArch

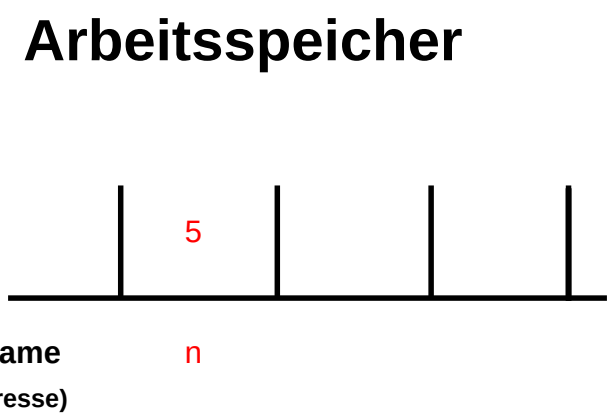
Schauen wir uns den Arbeitsspeicher etwas genauer an, dann können wir uns den als eine Art Setzkasten, in dem man verschiedene Objekte ablegen kann, vorstellen.



Wenn wir in der Console von Rstudio als Beispiel der Variablen `n` den Wert 5 zuweisen, dann wird das mit folgender Anweisung gemacht

```
n <- 5
```

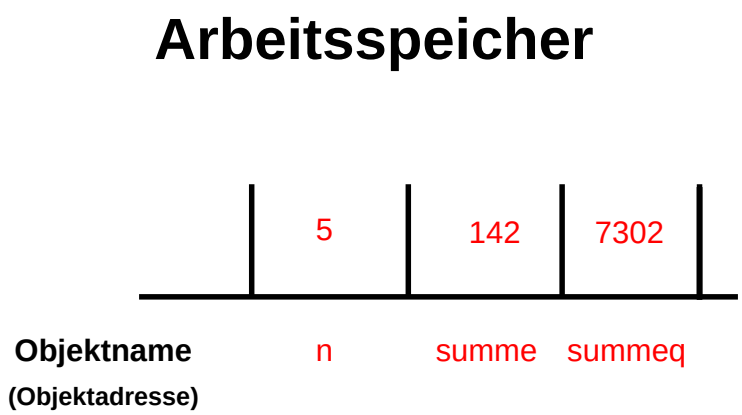
Das folgende Diagramm zeigt, wie sich durch diese Anweisung der Arbeitsspeicher verändert.



Berechnung und Zuweisung zu Variablen können kombiniert werden.

```
summe <- 15 + 9 + 8 + 34 + 76
summeq <- 15^2 + 9^2 + 8^2 + 34^2 + 76^2
```

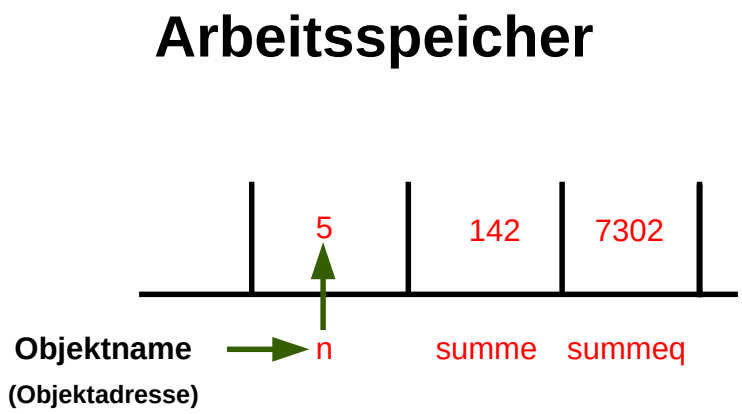
Die obige Berechnung und Zuweisung ist im nachfolgenden Diagramm dargestellt.



Der Zugriff auf die Werte, welche unter einer Variablen abgelegt sind, erfolgt durch die Eingabe des Variablennamens an der Console.

```
n
```

```
## [1] 5
```



Bezüglich der Namengebung von Variablen müssen einige Regeln eingehalten werden. Variablennamen können Buchstaben, Zahlen und Zeichen wie "-""," _" oder "." enthalten. Sie sollen aber nicht mit einer Zahl beginnen. Es wird empfohlen Namen von schon existierenden Funktionen nicht als Variablen zu verwenden. Allgemeine Hinweise zu Style, Namen und Notationen sind unter http://r-pkgs.had.co.nz/style.html erhältlich.

Mit Variablen kann wie mit Zahlen gerechnet werden. Für die Berechnung werden die im Arbeitsspeicher unter dem entsprechenden Variablennamen abgelegten Werte verwendet. Wollen wir als Beispiel aus den Werten, welche wir unter den Variablennamen `summe` und `summeq` abgelegt haben, den Mittelwert und die Standardabweichung berechnen, dann sieht das wie folgt aus.

```
m <- summe / n
s <- sqrt((summeq - summe^2/n)/(n-1))
```

Bei einer normalen Zuweisung wird kein Output generiert. Für die Anzeige der Resultate muss der Variablenname eingeben oder die Funktion `print()` verwendet werden.

```
m
```

```
## [1] 28.4
```

```
print(s)
```

```
## [1] 28.58846
```

Datentypen

Bis anhin hatten wir in R mit Zahlen gerechnet. Abgesehen von Zahlen gibt es noch die folgenden Datentypen in R.

Datentyp	Beschreibung
numeric	reelle Zahlen
integer	ganze Zahlen
complex	Quadratwurzel aus negativen Zahlen
character	Buchstaben, Zeichen
factor	Datentyp für lineare Modelle

Wichtige Punkte zu Datentypen

R kennt keine strenge Prüfung von Datentypen, d.h. R erlaubt es der gleichen Variablen einmal eine Zahl und dann ein Character zuzuweisen. Falls nötig und möglich macht R eine automatische Umwandlung zwischen Datentypen. Diese Umwandlung wird als **coersion** bezeichnet.

Hilfreiche Funktionen im Zusammenhang mit Datentypen sind:

- `class()` gibt den Typ eines Objekts zurück
- `is.<data.type>()` prüft, ob Objekt vom Datentype `<data.type>` ist. Als Beispiel überprüft `is.integer(5)`, ob 5 eine ganze Zahl ist.
- `as.<data.type>()` kann für explizite Umwandlungen verwendet werden. Zum Beispiel wandelt `as.character(12)` die Zahl 12 in den String "12" um.

Vektoren

Prinzipiell behandelt R jede Variable oder jedes Objekt als einen Vektor. Das ist für den Gebrauch in dieser Vorlesung nicht von grosser Bedeutung. Vektoren als Sequenzen von Komponenten des gleichen Datentyps sind für uns viel wichtiger. Diese werden mit der Funktion `vector()` erzeugt und mit der Funktion `c()` erweitert.

```r
vecNum <- vector(mode = "numeric", length = 2)
vecNum[1] <- 5
vecNum[2] <- -4
print(vecNum)
```

```
## [1]  5 -4
```

Analog dazu kann eine Vektor auch direkt ohne den Umweg über die Funktion `vector()` direkt erzeugt werden.

```r
vecNum <- c(5,-4)
print(vecNum)
```

```
## [1]  5 -4
```

Vektoren können mit der Funktion `c()` auch erweitert werden.

```r
vecNum <- c(vecNum, 43,-2)
print(vecNum)
```

```
## [1]  5 -4 43 -2
```

```r
vecChar <- c("aa", "ba")
vecChar <- c(vecChar, vecNum)
print(vecChar)
```

```
## [1] "aa" "ba" "5"  "-4" "43" "-2"
```

Das zweite Beispiel, in welchem wir den Vektor `vecChar` um den Vektor `vecNum` erweiterten, zeigt, dass bei der Erweiterung eines Character-Vektors um einen Zahlen-Vektor, die Zahlen automatisch in Strings umgewandelt werden. Umgekehrt, ist es nicht möglich einen Vektor von Zahlen um einen Character-Vektor zu erweitern, da Character nicht eindeutig in Zahlen verwandelt werden können.

Eine wichtige Eigenschaft eines Vektors ist seine **Länge**. Hier hat Länge nicht eine geometrische Bedeutung, sondern hier ist Länge die Anzahl Komponenten im Vektor gemeint. Die Funktion `length()` ermittelt die Anzahl Elemente in einem Vektor.

```r
length(vecChar)
```

```
## [1] 6
```

Logische Vergleiche können direkt auf Vektoren angewendet werden. Als Resultat erhalten wir einen Vektor mit Boolschen Komponenten der gleichen Länge, wie der ursprüngliche Vektor. Als Beispiel können wir für alle Komponenten eines numerischen Vektors testen, ob die Komponenten grösser als ein Bestimmter Wert sind.

```r
vecNum > 5
```

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

Die **arithmetischen Operationen** mit Vektoren werden alle komponenten-weise ausgeführt.

```r
x <- c(3,5,13,-2)
y <- c(2,6,-3,19)
x+y
```

```
## [1]  5 11 10 17
```

```r
x-y
```

```
## [1]   1  -1  16 -21
```

```r
x*y
```

```
## [1]   6  30 -39 -38
```

```r
x/y
```

```
## [1]  1.5000000  0.8333333 -4.3333333 -0.1052632
```

Das Skalarprodukt zweier Vektoren berechnen wir mit der Funktion `crossprod()`

```r
crossprod(x,y)
```

```
##      [,1]
## [1,]  -41
```

Der **Zugriff auf ein bestimmtes Elementes** i eines Vektors x geschieht mit dem Ausdruck $x[i]$.

```r
x[2]
```

```
## [1] 5
```

Setzen wir den Index i in $x[i]$ auf einen Wert $i < 0$, dann erhalten wir den Vektor, in welchem das Element i fehlt.

```r
x[-3]
```

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

Mit einem Bereich von Indices kann ein Teil des Vektors angesprochen werden.

```r
x[2:4]
```

```
## [1]  5 13 -2
```

Matrizen

Matrizen sind ihrem mathematischen Vorbild nachempfunden und somit in Zeilen und Kolonnen organisiert. Alle Komponenten einer Matrix müssen vom gleichen Datentyp sein. Eine Matrix in R wird mit der Funktion `matrix()` erstellt.

```
matA <- matrix(c(5,3,4,-6,3,76), nrow = 2, ncol = 3, byrow = TRUE)
print(matA)
```

```
##      [,1] [,2] [,3]
## [1,]    5    3    4
## [2,]   -6    3   76
```

Falls die Matrix zeilenweise aufgefüllt werden soll, dann ist die Option `byrow = TRUE` wichtig. Andernfalls wird die Matrix kolonnenweise aufgefüllt.

Eine grundlegende Eigenschaft einer Matrix ist ihre Dimension. Sie entspricht der Anzahl Zeilen und Kolonnen der Matrix und wird mit der Funktion `dim()` bestimmt.

```
dim(matA)
```

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

Der **Zugriff auf Elemente** einer Matrix erfolgt analog zum Zugriff bei den Vektoren. Aber für die Spezifikation eines einzelenen Elementes braucht es bei den Matrizen zwei Indices.

```
matA[2,1]
```

```
## [1] -6
```

Es ist aber auch möglich eine ganze Zeile oder eine ganze Spalte zu selektieren. Das Resultat ist in beiden Fällen ein Vektor.

```
matA[1,]
```

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

Was etwas verwirrend aussieht ist, dass beim Zugriff auf eine Spalte, scheinbar auch eine Zeile ausgegeben wird, das hängt aber damit zusammen, dass das Resultat ein Vektor ist und Vektoren werden immer auf einer Zeile ausgegeben.

```
matA[,2]
```

```
## [1] 3 3
```

Bestehende Matrizen können auch **erweitert** werden. Dies kann auf zwei Arten passieren. Entweder stapeln wir Matrizen aufeinander oder wir stellen sie nebeneinander. Die beiden Fälle sind in den nachfolgenden Statements gezeigt.

```
matB <- matrix(c(3,-1,90,1,1,4), nrow = 2, ncol = 3, byrow = TRUE)
rbind(matA, matB)
```

```
##      [,1] [,2] [,3]
## [1,]    5    3    4
## [2,]   -6    3   76
## [3,]    3   -1   90
## [4,]    1    1    4
```

Alternativ können wir die Matrizen auch in Kolonnenrichtung erweitern.

```
cbind(matA, matB)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    5    3    4    3   -1   90
## [2,]   -6    3   76    1    1    4
```

Die aus der lineare Algebra bekannten **Operationen** können wir auch hier in R anwenden. Als erstes können wir eine gegebene Matrix **transponieren**.

```
t(matA)
```

```
##      [,1] [,2]
## [1,]    5   -6
## [2,]    3    3
## [3,]    4   76
```

Wir können also einfach überprüfen, dass die Transponierte Matrix der Transponierten wieder der ursprünglichen Matrix entspricht.

```
t(t(matA))
```

```
##      [,1] [,2] [,3]
## [1,]    5    3    4
## [2,]   -6    3   76
```

Die **arithmetischen** Rechenoperationen $+$, $-$, $*$ und $/$ werden alle Element-weise ausgeführt.

```
matA + matB
```

```
##      [,1] [,2] [,3]
## [1,]    8    2   94
## [2,]   -5    4   80
```

```
matA - matB
```

```
##      [,1] [,2] [,3]
## [1,]    2    4  -86
## [2,]   -7    2   72
```

```
matA * matB
```

```
##      [,1] [,2] [,3]
## [1,]   15   -3  360
## [2,]   -6    3  304
```

```
matA / matB
```

```
##           [,1] [,2]        [,3]
## [1,]  1.666667   -3  0.04444444
## [2,] -6.000000    3 19.00000000
```

Die **Matrixmultiplikation**, welche wir aus der linearen Algebra kennen, muss entweder mit dem speziellen Operator `%*%` oder mit der Funktion `crossprod()` berechnet werden.

```
matA %*% t(matB)
```

```
##      [,1] [,2]
## [1,]  372   24
## [2,] 6819  301
```

```
crossprod(matA, matB)
```

```
##      [,1] [,2] [,3]
## [1,]    9  -11  426
## [2,]   12    0  282
## [3,]   88   72  664
```

Wir sehen hier dass die beiden Matrixmultiplikationen `matA %*% t(matB)` und `crossprod(matA, matB)` nicht das gleiche Resultat ergeben. Aus der Linearen Algebra wissen wir auch, dass die Matrizen `matA` und `matB` so nicht kompatibel sind für die Matrixmultiplikation. Die Funktion `crossprod()` transponiert automatisch die erste Matrix und somit wird die folgende Berechnung durchgeführt.

```
t(matA) %*% matB
```

```
##      [,1] [,2] [,3]
## [1,]    9  -11  426
## [2,]   12    0  282
## [3,]   88   72  664
```

Die **Inverse** einer Matrix erhalten wir als Resultat der Funktion `solve()`.

```
matC <- matA %*% t(matB)
(matCinv <- solve(matC))
```

```
##              [,1]          [,2]
## [1,] -0.005823853  0.0004643603
## [2,]  0.131936383 -0.0071975853
```

Als Kontrolle berechnen wir das Produkt der Inversen und der ursprünglichen Matrix und müssen als Resultat die Einheitsmatrix bekommen.

```
matCinv %*% matC
```

```
##              [,1]          [,2]
## [1,] 1.000000e+00 -2.775558e-17
## [2,] 7.105427e-15  1.000000e+00
```

Listen

In Listen kann man Sammlungen von Objekten, welche nicht den gleichen Datentyp haben, abspeichern. Die Definition einer Liste erfolgt über die Angabe von Schlüssel-Werte-Paaren. Was das bedeutet, wird im folgend Beispiel gezeigt.

```
lstA <- list(nZahlen = c(5,-2,7),
             sNamen = c("Fred","Mary"),
             lBool = c(FALSE,TRUE))
```

Der **Zugriff** auf die Elemente mit Indices oder die Angabe von Schlüsselnamen gibt als Resultat wieder eine Liste zurück mit dem entsprechenden Schlüssel-Werte-Paar, welches selektiert wurde.

```
lstA[2]
```

```
## $sNamen
## [1] "Fred" "Mary"
```

Ein Element kann aber auch mit einem Namen selektiert werden.

```
lstA["nZahlen"]
```

```
## $nZahlen
## [1]  5 -2  7
```

Die Liste aller Namen erhalten wir mit der Funktion `names()`.

```
names(lstA)
```

```
## [1] "nZahlen" "sNamen"  "lBool"
```

Wollen wir die Elemente einer Liste so selektieren, dass ein Vektor als Resultat zurückkommt, dann müssen wir doppelte Klammern oder den `$`-Operator verwenden.

```
lstA[[3]]
```

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

```
lstA[["lBool"]]
```

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

```
lstA$lBool
```

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

Dataframes

Dataframes sind spezielle Listen in R. Diese erhalten wir als Resultat, wenn wir Daten von Dateien einlesen (siehe nächster Abschnitt). Technisch gesehen sind Dataframes eine Mischung aus Listen und Matrizen. Dies wird klar, wenn wir uns den möglichen Zugriff auf Elemente eines Dataframes anschauen. Zuerst müssen wir aber ein Dataframe erzeugen. Dies geschieht an dieser Stelle mit der Funktion `data.frame()`. Die Option `stringsAsFactors = FALSE` muss angegeben werden, da sonst alle Strings in Faktoren umgewandelt werden.

```
dfA <- data.frame(nZahl = c(-2,15),
                  sZeichen = c("Alice","Bob"),
                  bWahr = c(FALSE,FALSE),
                  stringsAsFactors = FALSE)
```

Der Zugriff auf die einzelnen Elemente kann nun wie bei einer Matrix über die Angaben von Zeilen- und Kolonnenindices sein, oder wie bei einer Liste mit der Angabe eines Schlüsselnamens.

```
dfA[2,1]
```

```
## [1] 15
```

Mit der Angabe eines Schlüsselnamens erhalten wir alle Werte zum entsprechenden Schlüssel.

```
dfA$sZeichen
```

```
## [1] "Alice" "Bob"
```

Einlesen von Daten

Eine Verwendungsart von R ist die statistische Analyse von Daten. Zu diesem Zweck müssen wir die Daten zuerst einlesen. Erst dann können wir sie analysieren. Die wichtigste Funktion um Daten in R einzulesen, heisst `read.table()`. Mit dieser Funktion können Daten aus Files, welche Tabellen-artig organisiert sind, eingelesen werden. Haben die einzulesenden Daten ein spezifischers Format, so wie zum Beispiel `Comma Separated Values` (CSV), so gibt es spezialisierte Funktionen, wie `read.csv2()`, welche Daten im CSV-Format einlesen können. Ein Beipiel dafür sei nachfolgend gezeigt. Die Option `file = "csv/br_gew.csv"` gibt an, wo im Dateisystem das File mit den Daten zu finden ist.

```
dfBrGew <- read.csv2(file = "csv/br_gew.csv")
dim(dfBrGew)
```

```
## [1] 10  2
```

Die Funktion `dim()` gibt die Dimension der eingelesenen Daten. Dies ist eine gute Kontrolle, ob der Einleseprozess auch wirklich funktioniert hat. Als Resultat gibt die Funktion `read.csv2()` ein Dataframe zurück. Die Funktion `head()` liefert die ersten paar Zeilen des eingelesenen Dataframes.

```
head(dfBrGew)
```

```
##   Brustumfang Gewicht
## 1         176     471
## 2         177     463
## 3         178     481
## 4         179     470
## 5         179     496
## 6         180     491
```

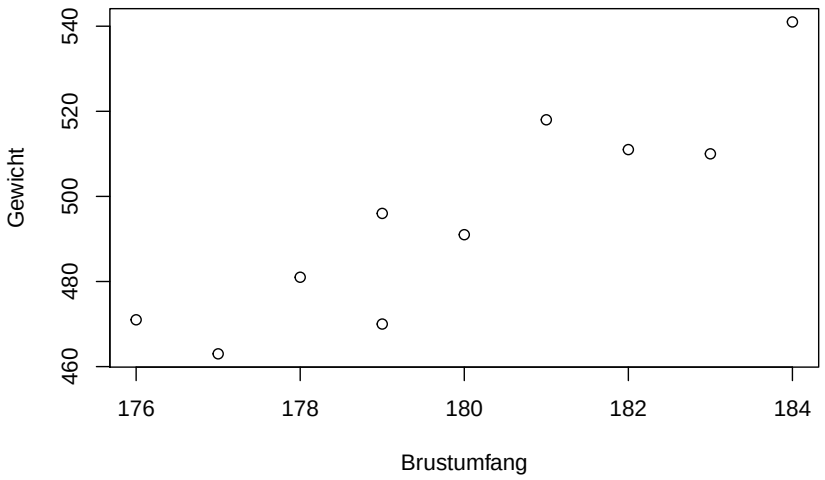
Die umgekehrte Vorgang des Lesens von Daten, das **Schreiben von Daten** in Dateien, kann je nachdem auch wichtig sein. Dazu gibt es die zu `read.table()` analogen Funktionen namens `write.table()`. Sollen die Daten im CSV-Format geschrieben werden dann können wir das mit `write.csv2()` tun. Für den Output von Daten, welche nicht Tabellen-artig organisiert sind, kann auch die Funktion `cat()` verwendet werden.

Plots und Diagramme

Sobald die Daten in R eingelesen sind, ist ein erster Schritt häufig eine Beschreibung der Daten mithilfe von graphischen Hilfsmitteln. R hat eine grosse Auswahl an Möglichkeiten für die graphische Darstellung von Daten. Hier sind nur die einfachsten erwähnt.

Die Funktion `plot()` kann verwendet werden, um einfache zwei-dimensionale Darstellungen zu erzeugen. Wenn wir als Beispiel das Gewicht gegen den Brustumfang aus dem Dataframe `dfBrGew` darstellen wollen, dann geschieht das mit folgendem Statement.

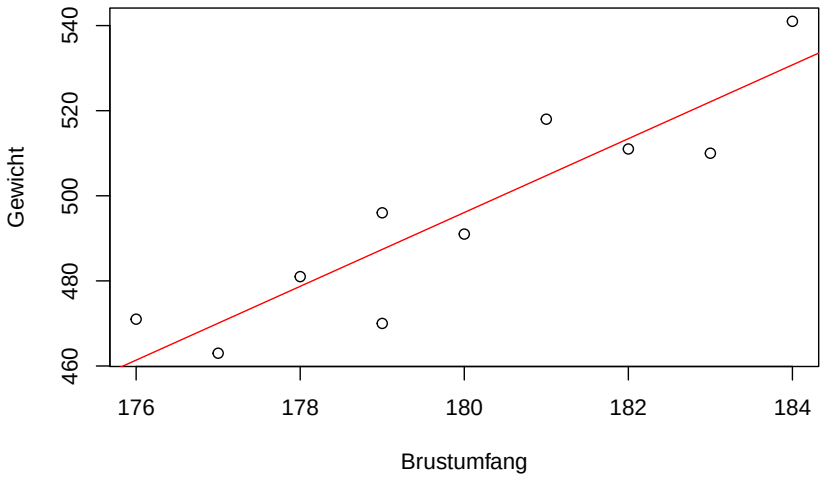
```
plot(dfBrGew$Brustumfang, dfBrGew$Gewicht, xlab = "Brustumfang", ylab = "Gewicht")
```



Einfaches lineares Modell

Zur Überprüfung eines statistischen Zusammenhangs zwischen den Variablen `Gewicht` und `Brustumfang` können wir ein einfaches lineares Modell anpassen. Wie gut das angepasste lineare Modell zu den Daten passt, können wir visuell durch den Vergleich der Daten zur gefundenen Regressionsgeraden beurteilen. Die Modellanpassung und die Erstellung des Plots werden mit folgenden Statements erzeugt.

```
lmBrGew <- lm(Gewicht ~ Brustumfang, data = dfBrGew)
plot(dfBrGew$Brustumfang, dfBrGew$Gewicht, xlab = "Brustumfang", ylab = "Gewicht")
abline(coef = coefficients(lmBrGew), col = "red")
```



Erweiterungen

Die Popularität von R ist auch bedingt durch, dass das System als solches fast beliebig erweitert werden kann. Grundsätzlich sind zwei Arten der Erweiterung denkbar.

1. Packages
2. Benutzer-definierte Funktionen

Packages

Unter einem Package versteht man eine Sammlung von Funktionen, welche von Autoren zur Verfügung gestellt werden. Als BenutzerIn von R können diese Packages mit der Funktion `install.packages()` verwendet werden. Wenn wir als Beispiel das Package namens `pedigreemm` verwenden wollen, dann können wir dieses mit folgendem Befehl installieren.

```
install.packages(pkgs = "pedigreemm")
```

Nach einer erfolgreichen Installation von `pedigreemm` können die Funktionen in `pedigreemm` verwendet werden.

Eigene Funktionen

Müssen gewisse Befehle wiederholt und häufig ausgeführt werden, dann empfielt es sich die Befehle in eine BenutzerIn-definierte Funktion zu verpacken. Nach der Definition der Funktion, kann diese genau wie jede andere R-Funktion aufgerufen werden.

Nehmen wir beispielsweise an, wir möchten Temperaturwerte von der Celsius- auf die Fahrenheit-Skala umrechnen, dann können wir das mit einer Funktion machen.

```
celcius_in_fahrenheit <- function(pnCelsius){
  nResultFahrenheit <- 32 + 9/5 * pnCelsius
  return(nResultFahrenheit)
}
```

Die Umrechnung für einen bestimmten Wert geschieht jetzt über folgenden Aufruf.

```
celcius_in_fahrenheit(pnCelsius = 0)
```

```
## [1] 32
```

```
celcius_in_fahrenheit(pnCelsius = 7)
```

```
## [1] 44.6
```

```
celcius_in_fahrenheit(pnCelsius = 25)
```

```
## [1] 77
```

Abkürzungen

Abbreviation	Meaning
CRAN	Comprehensive R Archive Network
CSV	Comma Separated Values