Ökonomisches Arbeiten in R

AUTHOR
Jonathan Wenger

Dieses Skript soll euch dabei helfen, besser und effizienter in R zu arbeiten. Dabei geht es um:

- Updates und package-Auswahl
- short-cuts
- Schleifen und deren Alternativen

1. R auf dem neuesten Stand halten

Klingt trivial, kann aber große Vorteile mit sich bringen:

- bug fixes in jeder neuen Version
- bessere Stabilität und Performance (gerade relevant, wenn euer PC nicht der neueste ist)
- Manche packages funktionieren nur mit den neuesten R Versionen

Um R zu updaten, empfiehlt sich das `installr` package. Mit diesem lassen sich auch packages in die neueste Version übertragen und andere Dinge (wie z.B. rtools) installieren.

```
library(installr)
updateR()
```

RStudio könnt ihr einfach über **Help –> Check for Updates** innerhalb der IDE updaten.

1.2 Packages updaten

Auch hier lohnt es sich, auf dem aktuellen Stand zu bleiben. Nicht nur wird die Performance häufig verbessert, sondern es finden sich auch hier wichtige bug-fixes in den Updates.

```
update.packages()
```

Alternativ auch möglich über den Reiter **Packages -> Update**. Hier empfiehlt es sich jedoch, vor einem größeren Projekt (z.B. der Masterarbeit) einmal alles zu updaten und dann während des Projekts keine Updates durchzuführen. So seid ihr auf der sicheren Seite, dass sich nichts wichtiges für euch ändert, während ihr daran arbeitet.

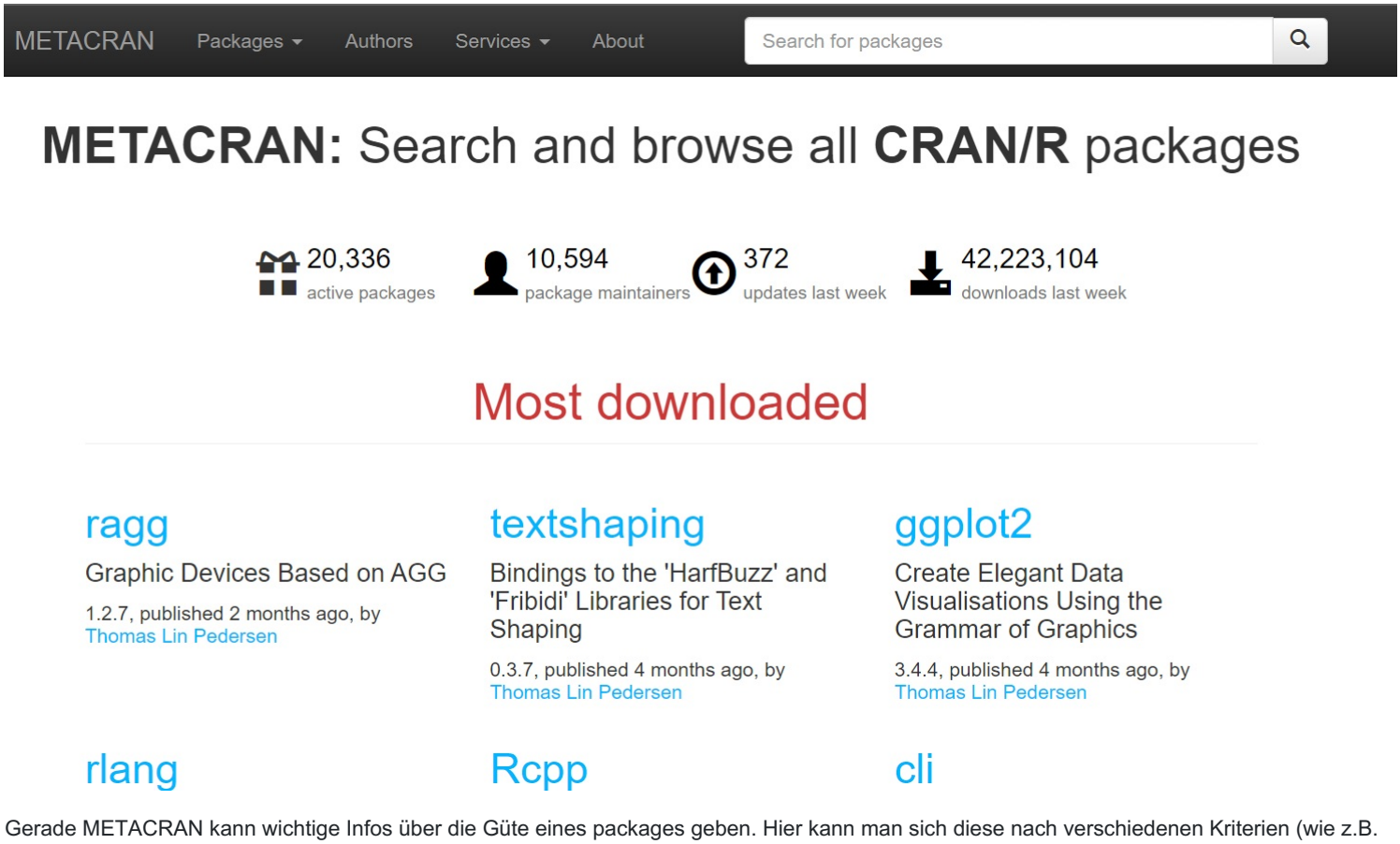
Wenn ihr dennoch automatisch alle packages updaten wollt, könnt ihr das tun, indem ihr update.packages(ask = FALSE) in eurer .Rprofile file hinzufügt (wenn ihr nicht wisst, was das ist, dann lasst am besten die Finger davon).

1.3 Package Management

Wenn ihr an einem größeren R Projekt sitzt, werdet ihr vermutlich viele verschiedene packages benutzen. Um da nicht den Überblick zu verlieren, kann es hilfreich sein, diese gesammelt am Start des Dokuments zu laden.

Bei der Auswahl der richtigen packages können die folgenden Seiten helfen:

- https://www.rdocumentation.org/
- https://www.r-pkg.org/



Gerade METACRAN kann wichtige Infos über die Güte eines packages geben. Hier kann man sich diese nach verschiedenen Kriterien (wie z.B. Häufigkeit der downloads) ausgeben lassen.

Die Suche nach dem richtigen package kann am Anfang zwar nervig und unnötig erscheinen, aber es ist häufig sinnvoll, sich einmal am Anfang länger damit auseinanderzusetzen, damit man nicht im Laufe des Projektes auf einmal merkt, es hätte ja noch ein viel besseres package für das gegeben, was man gerne machen würde.

2. Effiziente Skriptbearbeitung

Das vielleicht wichtigste tool, um wirklich effizient und ökonomisch in R arbeiten zu können. Auch wenn RStudio einem mit seiner graphischen Benutzeroberfläche viel abnehmen (möchte), kann der richtige Umgang mit short-cuts einem das Leben um einiges erleichtern.

2.1 Auto-Vervollständigung

`Tab` und `Enter` auto-verollständigen den Code. So reicht es z.B. aus, “micr” einzugeben und `Tab` zu drücken, um `microbenchmark` entstehen zu lassen. Wenn R mehrere Möglichkeiten angibt, so kann man mit den Pfeiltasten die passende auswählen und dann ebenfalls mit `Tab` ausführen. Sollte R nicht automatisch Vorschläge geben, kann man sich diese mit `Tab` anzeigen lassen. Das ist besonders dann hilfreich, wenn man nur die ersten 1-2 Buchstaben eingeben hat.

Diese Auswahl funktioniert auch mit Spalten in einem dataframe. In dem unteren Datensatz kann man so leicht, nachdem man das `$` eingefügt hat, eine der vier Spalten mit den Pfeiltasten auswählen und mit `Tab` ausgeben lassen. Also: `Tab` ist euer bester Freund, gerade auch beim Bearbeiten von längeren Datensätzen.

```
head(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
```

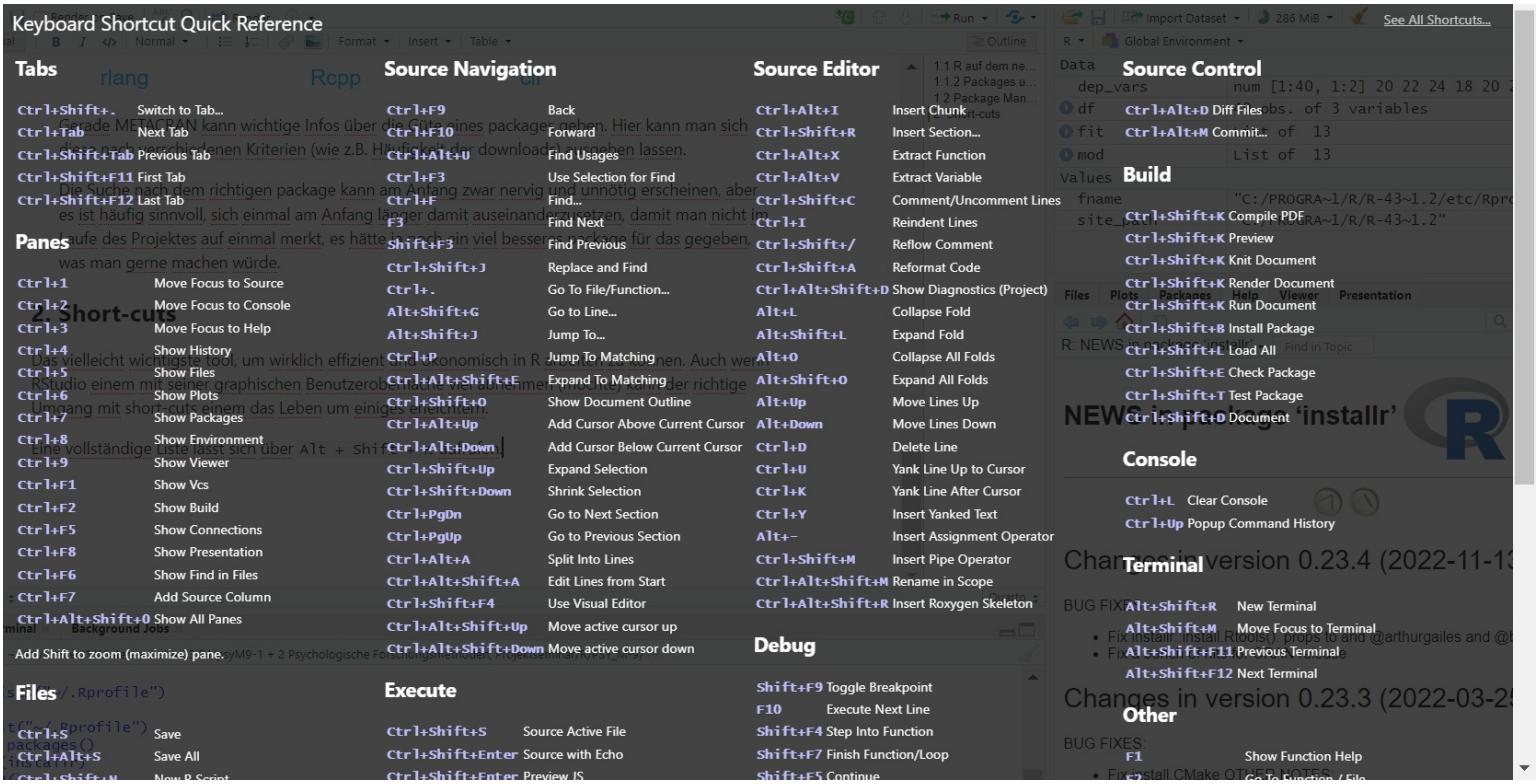
```
#iris$ Hier wird ein drip-down-Menü auftauchen, aus welchem ihr eine Spalte auswählen könnt
iris$Petal.Length
```

```
  [1] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 1.4 1.1 1.2 1.5 1.3 1.4
 [19] 1.7 1.5 1.7 1.5 1.0 1.7 1.9 1.6 1.6 1.5 1.4 1.6 1.6 1.5 1.5 1.4 1.5 1.2
 [37] 1.3 1.4 1.3 1.5 1.3 1.3 1.3 1.6 1.9 1.4 1.6 1.4 1.5 1.4 4.7 4.5 4.9 4.0
 [55] 4.6 4.5 4.7 3.3 4.6 3.9 3.5 4.2 4.0 4.7 3.6 4.4 4.5 4.1 4.5 3.9 4.8 4.0
 [73] 4.9 4.7 4.3 4.4 4.8 5.0 4.5 3.5 3.8 3.7 3.9 5.1 4.5 4.5 4.7 4.4 4.1 4.0
 [91] 4.4 4.6 4.0 3.3 4.2 4.2 4.2 4.3 3.0 4.1 6.0 5.1 5.9 5.6 5.8 6.6 4.5 6.3
```

```
[109] 5.8 6.1 5.1 5.3 5.5 5.0 5.1 5.3 5.5 6.7 6.9 5.0 5.7 4.9 6.7 4.9 5.7 6.0
[127] 4.8 4.9 5.6 5.8 6.1 6.4 5.6 5.1 5.6 6.1 5.6 5.5 4.8 5.4 5.6 5.1 5.1 5.9
[145] 5.7 5.2 5.0 5.2 5.4 5.1
```

2.2 Short-Cuts

Eine vollständige Liste lässt sich über `Alt + Shift + K` (Mac (`Options+ Shift + K)`) aufrufen:



Diese short-cuts sind wirklich nicht zu unterschätzen. Ihr könnt euch einiges an Zeit und Arbeit ersparen, indem ihr die wichtigsten lernt und beherrscht:

Short-Cut (Windows & Linux)	Short-Cut (Mac)	Funktion
`Strg + Z`	`Cmd + Z`	Rückgängig machen
`Strg + Enter`	`Cmd + Return`	Die Zeile ausführen / Den ausgewählten Code ausführen
`Strg + Alt + R`	`Cmd + Option + R`	Den kompletten Code ausführen
`Strg + Alt + B`	`Cmd + Option + B`	Führe den Code vom Anfang bis zur jetzigen Zeile aus
`Strg + Alt + E`	`Cmd + Option + E`	Führt den Code von jetzt bis zum Ende aus
`Alt + Shift + Up/Down`	`Ctrl + Option + Shift + Up/Down`	Die Zeile drüber oder drunter kopieren
`Strg + D`	`Cmd + D`	Die Zeile löschen
`Strg + Left/Right`	`Option + Left/Right`	Navigiert durch den Code Wort für Wort
`Alt + -`	`Option + -`	<-
`Strg + Shift + M`	`Cmd + Shift + M`	%>%

Das ist nur eine Auswahl. Schaut in der Liste der short-cuts nach, was ihr für eure Arbeiten gebrauchen könntet und versucht euch diese short-cuts anzueignen, wenn ihr schneller arbeiten wollt.

2.3 Kommentare

Kommentiert am besten alles, was ihr tut. Auch wenn es nervig ist, kann es einem im späteren Verlauf helfen, was man sich bei dem Code gedacht hat. Gerade, wenn ihr den Code mit anderen Personen teilt, um euch bspw. Hilfe einzuholen, kann dies sehr hilfreich sein. Vergleicht die beiden unteren Code-Blöcke um zu sehen, wie hilfreich Kommentare für ein schnelles Verständnis des Codes sind.

```
load("erstis.RData")
library(lavaan)
library(semPlot)
```

```
cfa.1.txt <- 'f =~ stim1 + stim4 + stim8 + stim11'

cfa.1.fit <- cfa(cfa.1.txt, data = erstis)

summary(cfa.1.fit, fit = TRUE)

cfa.2.txt <- 'f1 =~ stim1 + stim8
              f2 =~ stim4 + stim11'

cfa.2.fit <- cfa(cfa.2.txt, data = erstis)

summary(cfa.2.fit, fit = TRUE)

vergleich <- anova(cfa.1.fit, cfa.2.fit)
```

```
## Konfirmatorische Faktorenanalyse ##
# Datensatz laden
load("erstis.RData")
library(lavaan)
library(semPlot)

#Modelkonstruierung
cfa.1.txt <- 'f =~ stim1 + stim4 + stim8 + stim11'

#Modelschätzung
cfa.1.fit <- cfa(cfa.1.txt, data = erstis)
summary(cfa.1.fit, fit = TRUE)


#Alternative Modellkonstruierung
cfa.2.txt <- 'f1 =~ stim1 + stim8
              f2 =~ stim4 + stim11'


#Modelschätzung
cfa.2.fit <- cfa(cfa.2.txt, data = erstis)
summary(cfa.2.fit, fit = TRUE)


#Modellvergleich
vergleich <- anova(cfa.1.fit, cfa.2.fit)
```

3. Schleifen

Schleifen können ein hilfreiches tool in R sein, wenn sie richtig eingesetzt werden. Schauen wir uns dazu mal das untere Beispiel an:

```
n <- 1000000

method1 <- function(n){
  x <- NULL
  for (i in 1:n) {
    x[i] <- sqrt(i)
  }
}
```

Hier wurde eine Funktion geschrieben, die einem die Wurzel jeder Zahl aus einem Datenvektor ausgeben soll. Allerdings ist diese Funktion nicht ganz optimal, da das in der Schleife wachsende Objekt “x” zu Beginn nicht vollständig initialisiert wurde. Besser wäre es so:

```
method2 <- function(n){
  x <- numeric(n)
  for (i in 1:n) {
    x[i] <- sqrt(i)
  }
}
```

Diese beiden Funktionen lassen sich nun in Bezug auf ihre Durchführungsgeschwindigkeit vergleichen:

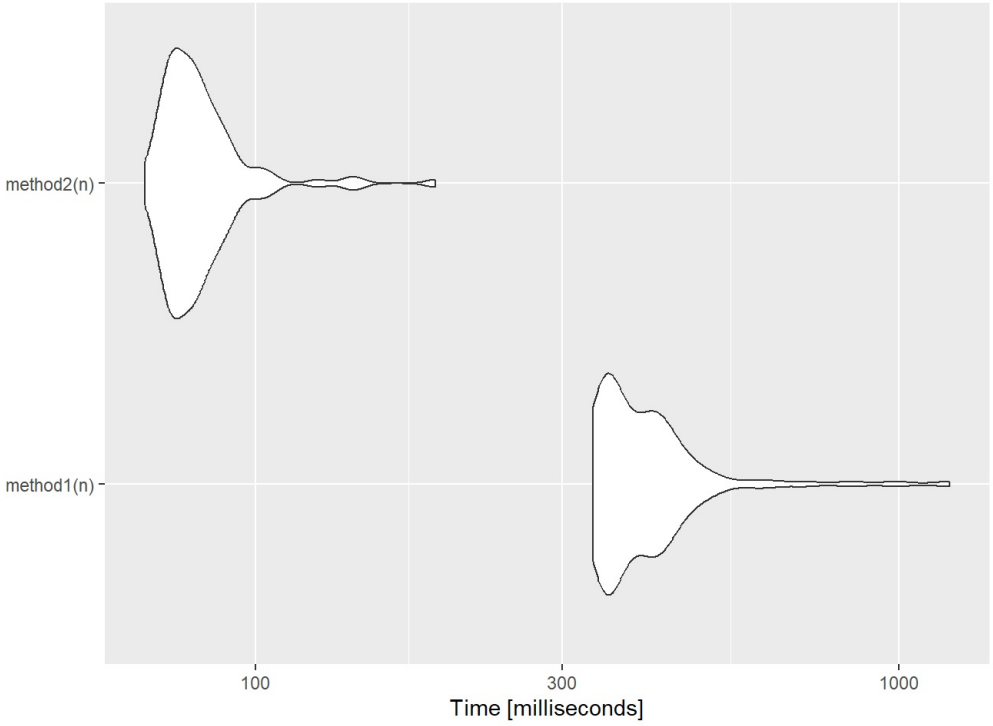
```
library(tidyverse)
```

```
── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
✔ dplyr     1.1.4     ✔ readr     2.1.5
```

```
✔ forcats   1.0.0     ✔ stringr   1.5.1
✔ ggplot2   3.4.4     ✔ tibble    3.2.1
✔ lubridate 1.9.3     ✔ tidyr     1.3.1
✔ purrr     1.0.2
── Conflicts ─────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

```r
library(microbenchmark)
vergleich <- microbenchmark(times = 100, unit = "s",
               method1(n), method2(n))

autoplot(vergleich)
```



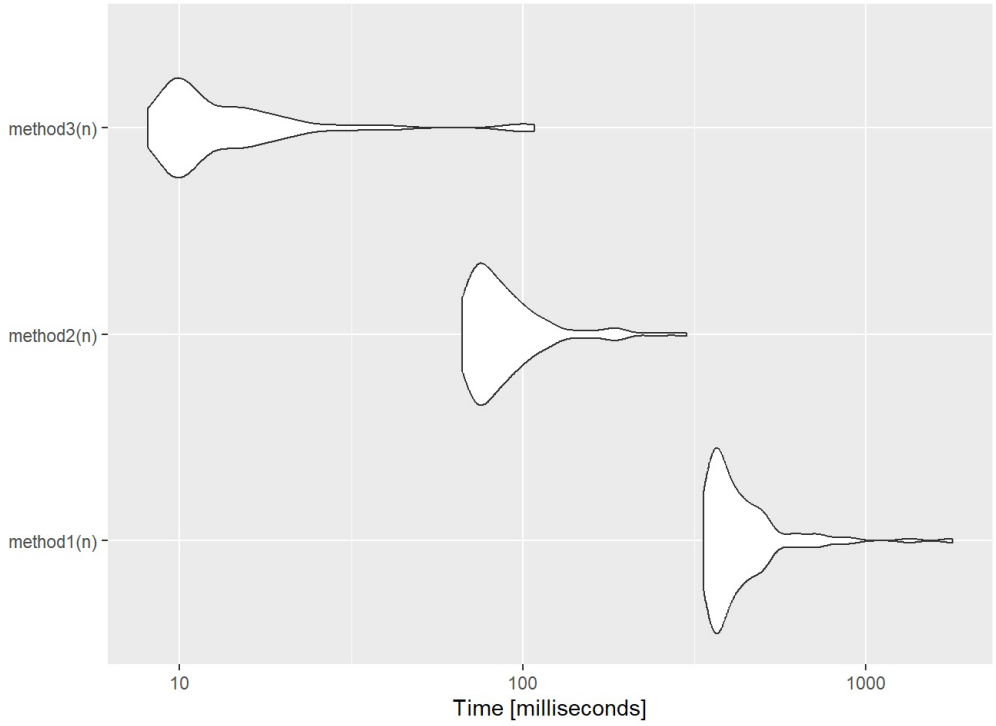
Wie man sehen kann, ist die zweite Methode durchaus um einiges schneller. Es geht aber noch einfacher; man kann sich die ganze Schleife sparen, indem man einfach vektorisiert:

```r
method3 <- function(n){
x <- sqrt(1:n)
}
```

Der Vergleich aller drei Methoden zeigt hier, dass das Vektorisieren die schnellste und effizienteste Methode ist:

```r
vergleich <- microbenchmark(times = 100, unit = "s",
               method1(n), method2(n), method3(n))

autoplot(vergleich)
```



3.1 Alternativen zu Schleifen

Eine gute Art zu Vektorisieren ist über die Funktionen der `apply`-Familie möglich:

Funktion	Ziel	Input	Output
`apply`	Eine Funktion auf die Zeilen oder Spalten (oder beides) anwenden	data frame oder Matrix	Vektor, Liste, Array
`lapply`	Eine Funktion auf alle Elemente eines inputs anwenden	Liste, Vektor oder data frame	Liste
`sapply`	Eine Funktion auf alle Elemente eines inputs anwenden	Liste, Vektor oder data frame	Vektor oder Matrix
`tapply`	Eine Funktion auf eine Teilmenge eines inputs anwenden	Vektor	Vektor

Die Auswahl der passenden Funktion ist also davon abhängig, was das Ziel ist und in welchem output-Format ihr euer Ergebnis gerne haben würdet. Nachfolgend gibt es jeweils ein kurzes Beispiel für jede der vier Funktionen.

3.1.1 apply

```
#Datensatz erstellen
n <- 10
df <- as.data.frame(matrix(sample(1:10, n*9, TRUE), n, 9))
df
```

```
   V1 V2 V3 V4 V5 V6 V7 V8 V9
1   5 10  2  2  4  6  9  3  2
2   6  9  8  1  8  5  6  1  5
3   3  5  8  9  2 10  3  5  2
4   2  3 10  1  4  3  7  7  2
5  10  6  2  1  5  2 10  7  9
6   1  1  9  2  2  1  2  1  9
7   7  9  6  8  7 10  1  3  7
8   3  2  2  9  6  1  3  4  9
9   6  1  6 10  5  4  7  2 10
10  1  9  5  3  2  9  8  3  4
```

```
#Die Summe jeder Zeile berechnen
apply(df, 1, sum)
```

```
 [1] 43 49 47 39 52 28 58 39 51 44
```

```
#Die Summe jeder Spalte berechnen
apply(df, 2, sum)
```

```
V1 V2 V3 V4 V5 V6 V7 V8 V9
```

```
44 55 58 46 45 51 56 36 59
```

3.1.2 lapply

```
#Den Mittelwert jeder Spalte als Liste ausgeben lassen
lapply(df, mean)
```

```
$V1
[1] 4.4

$V2
[1] 5.5

$V3
[1] 5.8

$V4
[1] 4.6

$V5
[1] 4.5

$V6
[1] 5.1

$V7
[1] 5.6

$V8
[1] 3.6

$V9
[1] 5.9
```

```
#Den Wert in jeder Spalte mit 5 multiplizieren und als Liste ausgeben
lapply(df, function(df) df*5)
```

```
$V1
 [1] 25 30 15 10 50  5 35 15 30  5

$V2
 [1] 50 45 25 15 30  5 45 10  5 45

$V3
 [1] 10 40 40 50 10 45 30 10 30 25

$V4
 [1] 10  5 45  5  5 10 40 45 50 15

$V5
 [1] 20 40 10 20 25 10 35 30 25 10

$V6
 [1] 30 25 50 15 10  5 50  5 20 45

$V7
 [1] 45 30 15 35 50 10  5 15 35 40

$V8
 [1] 15  5 25 35 35  5 15 20 10 15

$V9
 [1] 10 25 10 10 45 45 35 45 50 20
```

Das Ganze funktioniert auch als Liste:

```
#Liste erstellen
x <- list(a = 5:6, b = 1:5, c = sample(1:25))

#Mittelwert jedes Elements der Liste als Liste ausgeben
lapply(x, mean)
```

```
$a
```

```
[1] 5.5

$b
[1] 3

$c
[1] 13
```

3.1.3 sapply

```
#Standardabweichung jeder Spalte als Vektor ausgeben
sapply(df, sd)
```

```
      V1       V2       V3       V4       V5       V6       V7       V8
2.913570 3.597839 3.011091 3.864367 2.121320 3.541814 3.134042 2.170509
      V9
3.281260
```

```
#Standardabweichung für jedes Element der Liste als Vektor ausgeben
sapply(x, sd)
```

```
        a         b         c
0.7071068 1.5811388 7.3598007
```

3.1.4 tapply

```
#Beispieldatensatz
head(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
```

```
#Mittelwert der Sepal.Length pro Spezies ausgeben
tapply(iris$Sepal.Length, iris$Species, mean)
```

```
    setosa versicolor  virginica
     5.006      5.936      6.588
```

```
#Max der Sepal.Width pro Spezies ausgeben
tapply(iris$Sepal.Width, iris$Species, max)
```

```
    setosa versicolor  virginica
       4.4        3.4        3.8
```

Sind Schleifen also unnötig? Nein. Es gibt sicher einige Situationen, in denen R keine passende Funktion für das habt, was ihr tun wollt. Spezifische Änderungen an einem großen data frame lassen sich z.B. super über Schleifen lösen. Es kann allerdings von Vorteil sein, vorher einmal zu schauen, ob es nicht schon passende Alternativen gibt, die schneller und effizienter sind, als die selbst geschriebene Schleife. Wichtig bei der Erstellung von Schleifen ist:

- Ergebnisse, die nur einmal berechnet werden müssen, vor dem loop abspeichern und nicht immer neu berechnen
- Generell: Alles, was nicht zwingend in der Schleife sein muss, **außerhalb der Schleife verlagern**!
- Objekte, die in der Schleife wachsen, vor der Schleife initialisieren
    - Wenn ihr wisst, wie lang ein Objekt wird, auch die Länge vorher schon abspeichern
- Das Wichtigste: **Schleifen vermeiden, wenn es bessere Alternativen gibt.** R und alle zugehörigen packages bieten einem unfassbar viele Möglichkeiten. Wer vorher länger sucht, hat hinterher vielleicht weniger Arbeit.

3.2 Ein Beispiel für eine schlechte und eine gute Schleife

```
#Für jede Zahl von 1 bis 1000000 soll 15 addiert werden und dann durch 2 geteilt werden
loop1 <- function(i){
  for(i in 1:1000000){
```

```
  N1 <- i
  N2 <- 15
  N3 <- N1 + N2
  N4 <- N3 / 2
  }
}
```

An dieser Schleife kann man schon einiges fehlerhaftes feststellen:

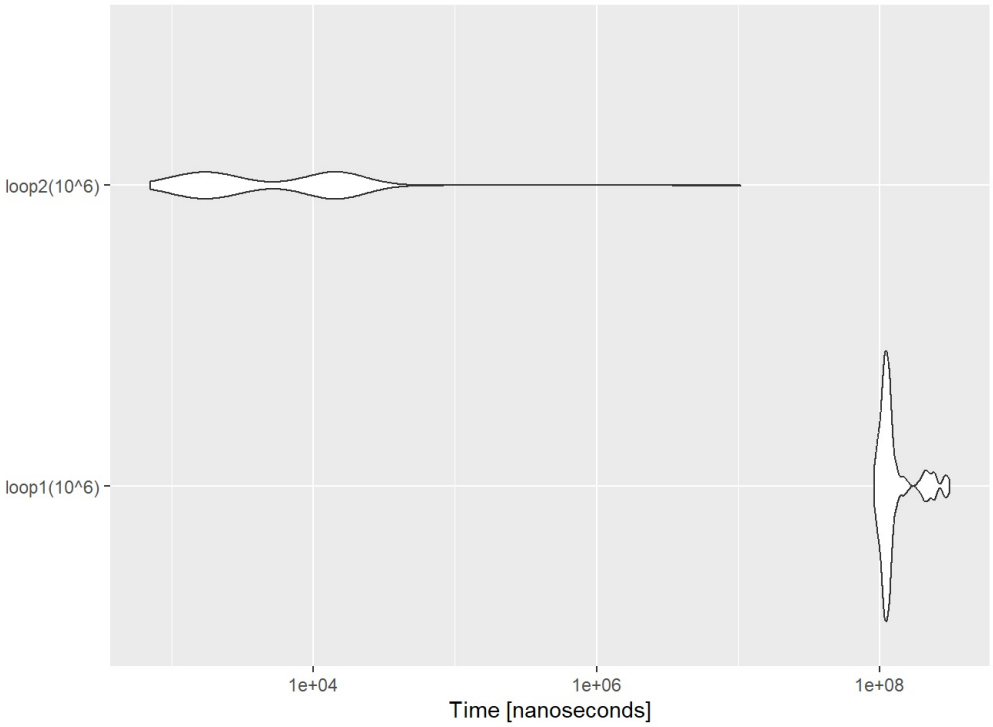
- Es wird alles (auch unnötigerweise) innerhalb der Schleife berechnet
- N1 ist nicht nötig zu deklarieren
- N2 sollte vor der Schleife deklariert werden
- N3 und N4 könnte man zusammenfassen in einem Rechenschritt

Verbessert könnte das ganze so aussehen:

```
N2 = 15
loop2 <- function(rep){
  for(i in rep){
    N3 <- (i + N2) / 2
  }
}
```

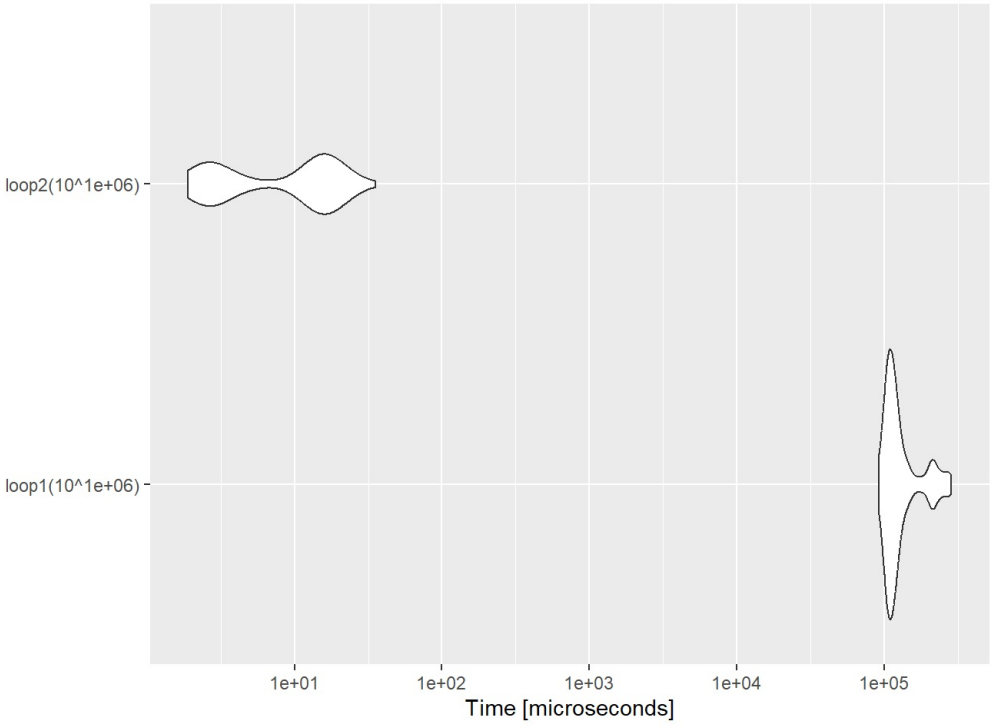
Vergleich zwischen beiden Schleifen:

```
vergleich <- microbenchmark(times = 100, unit = "s",
               loop1(10^6), loop2(10^6))
autoplot(vergleich)
```



Die Unterschiede werden umso “größer”, je mehr Wiederholungen man in der Schleife durchführt. In diesem Beispiel können selbst große Datenmengen keine wirklichen Probleme verursachen.

```
vergleich <- microbenchmark(times = 100, unit = "s",
               loop1(10^1000000), loop2(10^1000000))
autoplot(vergleich)
```



4. Weitere Informationen zum Nachschlagen

- https://cran.r-project.org/other-docs.html
- https://bookdown.org/csgillespie/efficientR/
- https://www.burns-stat.com/pages/Tutor/R_inferno.pdf

Bei dem ersten Link findet ihr eine Liste von verschiedenen Tutorials in R. Die letzten beiden Tutorials und Hilfen sind eher für Fortgeschrittene. Die meisten dort angesprochenen Problematiken und Themen sind vermutlich mehr als das, was in den Masterarbeiten auf einen zu kommt. Bei Interesse an der Materie kann dies aber durchaus hilfreich und interessant sein.