# Robustheit und Voraussetzungstests

Max Brede und Johannes Andres

2021-10-05

# Contents

**1 Vorwort** **5**

**2 Lehrplan** **7**
2.1 Semesterplan . . . . . 7

**3 Programmieren in R** **11**
3.1 Funktionen . . . . . 11
3.2 Geschachtelte Funktionen . . . . . 15
3.3 Optionale Argumente und if-Statements . . . . . 16
3.4 Zufallswerte . . . . . 19
3.5 Schleifen . . . . . 21

# Chapter 1

# Vorwort

Dieses mit `bookdown` erstellte Dokument ist das Skript zum Seminar "psyM9-1: Psychologische Forschungsmethoden. Projektseminar I" und "PSY_B_20_e-1: Forschungsorientierte Vertiefung: Forschungsmethoden" der CAU zu Kiel.

# Chapter 2

# Lehrplan

## 2.1 Semesterplan

| Sitzung | Datum | Sitzungstitel | Lernziele |
|---|---|---|---|
| | | | Die Studierenden… |
| 1 | 2021-10-25 | Rste Codeschnipsel | können Funktionen in R definieren, deren Bestandteile nennen und diese benutzen.<br>wissen, was if-Statements sind und können diese in R einsetzen |
| 2 | 2021-11-01 | Iterationen und Zufallszahlen | wissen, was for- und while-Schleifen sind und können diese in R einsetzen<br>können Werte von Zufallszahlen in R erzeugen |
| 3 | 2021-11-08 | Listen und Matrizen | wissen, wie eine „list“ in R aufgebaut ist und können diese benutzen<br>können Matrizen in R benutzen |
| 4 | 2021-11-15 | Welche Iterationen? | kennen verschiedene Arten von Rs iterativen „functionals“ und können diese benutzen<br>können basales microbenchmarking einsetzen um Funktionsperformance zu evaluieren |
| 5 | 2021-11-22 | Erste Simulation I | können geschachtelte Schleifen und iterative functionals nutzen |
| 6 | 2021-11-29 | Erste Simulation II | |
| 7 | 2021-12-06 | Beispiele | |
| 8 | 2021-12-13 | Beispiele | |
| 9 | 2021-12-20 | | |
| 10 | 2022-01-10 | | |

| | |
|---|---|
| 11 | 2022-01-17 |
| 12 | 2022-01-24 |
| 13 | 2022-01-31 |
| 14 | 2022-02-07 |
| 15 | 2022-02-14 |

# Chapter 3

# Programmieren in R

## 3.1 Funktionen

Wir kennen Funktionen ja schon aus den EDV-Veranstaltungen.

Zum Beispiel macht die `sum`-Funktion mit einem Vektor das, was der Name sagt:

```
sum(c(1,2,3))
```

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

```
1 + 2 + 3
```

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

Funktionen können wir auch selbst definieren. Eine Funktion, die uns begrüßt könnte zum Beispiel wie folgt aussehen:

```
greet_me <- function(){
  return('Hello! Nice to see you!')
}
```

Wenn wir jetzt die `greet_me`-Funktion aufrufen, sehen wir:

```
greet_me()
```

```
## [1] "Hello! Nice to see you!"
```

In der Funktionsdefinition können wir ein paar Teile wiederkennen. Zum Einen ist da der `greet_me <-`-Teil, den wir ja schon als Objektzuweisung kennen. Wir weisen also dem Ergebnis eines Ausdrucks den Namen `greet_me` zu.

Funktions-*Objekte* werden also genauso wie Datensätze und Vektoren als eine Kombination von Namen und zugehörigem Inhalt definiert.

Dabei erstellt die Funktion `function()` einen Objektinhalt, der aus *body* und *formals* besteht und in einem *environment* definiert ist.

Der *body* ist der Teil der Funktion, der definiert, was passieren soll und wird in R in geschweiften Klammern hinter der `function`-Funktion definiert In unserem Beispiel besteht der body aus dem Aufruf, einen Text zurückzugeben:

```
body(greet_me)
```

```
## {
##     return("Hello! Nice to see you!")
## }
```

Die *formals* sind die Argumente, die bei der Ausführung des *body*s genutzt werden sollen.

In unserem Beispiel haben wir noch keine Argumente berücksichtigt:

```
formals(greet_me)
```

```
## NULL
```

Wir könnten die Funktion aber neu definieren, so dass sie einen gegebenen Namen begrüßt. Dafür geben wir der `function`-Funktion ein Argument, das der Name des erwarteten Arguments sein soll:

```
greet_someone <- function(name){
  return(paste0('Hello ',name,'! Nice to see you!'))
}

greet_someone('Marvin')
```

```
## [1] "Hello Marvin! Nice to see you!"
```

Wenn wir uns jetzt nochmal die `formals` ausgeben lassen, sehen wir, dass ein Argument vorgesehen ist:

```
formals(greet_someone)
```

```
## $name
```

Der letzte Teil einer Funktionsdefinition ist das *environment*. Damit ist der Namensraum gemeint, auf den die Funktion zugreifen kann:

```
environment(greet_someone)
```

```
## <environment: R_GlobalEnv>
```

In unserem Beispiel wurde die Funktion in der laufenden R-Session definiert (wie die allermeisten Funktionen, die wir dieses Semester nutzen werden). Der Output `R_GlobalEnv` heißt also, dass die Funktion auf Objekte in der Haupt-R-Session zurückgreifen kann.

Praktisch heißt das, dass wir zum Beispiel Konstanten einmal außerhalb einer Funktion definieren können, die diese dann nutzen kann:

```
n <- 10

greet_someone_n_times <- function(name) {
  return(rep(paste0('Hello ',
                    name,
                    '! Nice to see you!'),
             n))
}

greet_someone_n_times('Marvin')
```

```
##  [1] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
##  [3] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
##  [5] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
##  [7] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
##  [9] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
```

Was das aber nicht heißt ist, dass die Funktion eine externe Variable ändern kann:

```
greet_someone_n_times <- function(name) {

  n <- 3

  return(rep(paste0('Hello ',
                    name,
                    '! Nice to see you!'),
             n))
}

greet_someone_n_times('Marvin')
```

```
## [1] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
## [3] "Hello Marvin! Nice to see you!"
```

```
n
```

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

Das liegt daran, dass die Funktion beim Aufrufen eine Kopie des globalen Environments als eigene Variablenumgebung zugewiesen bekommt und diese Kopie auch verändern kann. Die Kopie wird aber bei Beenden der Funktion verworfen, so dass Änderungen in der Funktion nicht erhalten bleiben.

Dabei werden die Argumente der Funktion dem Environment der Funktion hinzugefügt, wobei im Zweifelsfall gleichnamige Objekte für die Funktion überschrieben werden:

```
x <- 1:10
y <- 11:20

my_function <- function(x,y){
  return(c(x,y))
}

my_function(21,22)
```

```
## [1] 21 22
```

### 3.1.1 Aufgabe

1. Erstelle eine Funktion mit dem Namen `my_mean`, die den Mittelwert eines gegebenen Vektors berechnet.
2. Was ergibt der folgende Aufruf:

```
dummy_function <- function(a,b){
  a * b
}

c <- dummy_function(1,2)

formals(dummy_function)
```

3. Was ist am Ende des folgenden Aufrufs in `number` abgelegt?

```
number <- 'a number'

important_calculation <- function(number){
  number <- 42 / number * number
  return(number)
}

important_calculation(5)
```

```
## [1] 42
```

Antworten

**1.**

```
my_mean <- function(x){
  return(sum(x)/length(x))
}
```

**2.**

```
## $a
```

```
##
##
## $b
```

**3.**

```
'a number'
```

## 3.2 Geschachtelte Funktionen

Um den Code übersichtlicher zu halten, können Teile von Funktionen in andere Funktionen ausgegliedert werden. Dabei nutzen wir, dass unsere Funktionen auch auf das global Environment zugreifen und Funktionen ja auch nur Arten von Objekten sind.

Wenn wir unsere Funktionen `greet_someone` und `greet_someone_n_times` von vorhin nochmal angucken sehen wir, dass ein Teil des Codes in beiden auftaucht:

```
print(greet_someone)
```

```
## function(name){
##   return(paste0('Hello ',name,'! Nice to see you!'))
## }
```

```
print(greet_someone_n_times)
```

```
## function(name) {
##
##   n <- 3
##
##   return(rep(paste0('Hello ',
##                     name,
##                     '! Nice to see you!'),
##              n))
## }
```

Wir können `greet_someone_n_times` jetzt so umschreiben, dass sie `greet_someone` benutzt:

```
greet_someone_n_times <- function(name){
  n <- 3

  return(rep(greet_someone(name),
             n))
}


greet_someone_n_times('Marvin')
```

```
## [1] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
## [3] "Hello Marvin! Nice to see you!"
```

### 3.2.1 Aufgabe

1. Erstelle eine `my_var` Funktion, die die unkorrigierte Varianz eines Vektors ($S^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - M_X)^2$) berechnet. Benutze für den Mittelwert deine Mittelwerts-Funktion aus dem letzten Aufgaben-Block.

Antworten

**1.**

```
my_var <- function(x){
  m_x <- my_mean(x)

  x <- (x - m_x)^2

  return(1/length(x) * sum(x))
}
```

## 3.3 Optionale Argumente und if-Statements

Unsere `greet_someone_n_times`-Funktion sieht ja im Moment wie folgt aus:

```
print(greet_someone_n_times)
```

```
## function(name){
##   n <- 3
##
##   return(rep(greet_someone(name),
##              n))
## }
```

Ungünstig daran ist noch, dass das `n` bei jedem Aufruf als Konstante neu definiert ist.

Um dieses n anpassen zu können, können wir es als zweites Argument übergeben, das beim Aufruf festgelegt werden kann:

```
greet_someone_n_times <- function(name, n){
  return(rep(greet_someone(name),
             n))
}

greet_someone_n_times('Marvin',3)
```

```
## [1] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
## [3] "Hello Marvin! Nice to see you!"
```

Da wir uns aber das Tippen sparen wollen und nicht jedes Mal unseren Standard-Fall (n=3) explizit machen wollen, können wir dem Argument einen Standardwert zuweisen.

Dadurch machen wir aus dem `n` ein *optionales Argument*, wie wir es ja auch schon aus anderen Situationen kennen:

```
greet_someone_n_times <- function(name, n=3){
  return(rep(greet_someone(name),
             n))
}

greet_someone_n_times('Marvin')
```

```
## [1] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
## [3] "Hello Marvin! Nice to see you!"
```

```
greet_someone_n_times('Marvin', 5)
```

```
## [1] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
## [3] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
## [5] "Hello Marvin! Nice to see you!"
```

Als letzten Schritt wollen wir besonders höflich zu speziellen Nutzern sein. So soll der Name 'Justus'[1] ausschweifender gegrüßt werden, als alle anderen Namen.

Um das umsetzten zu können, müssen wir irgendwie einen Test einfügen, ob der gegebene Name gleich 'Justus' ist.

Dafür können wir ein so genanntes `if...else`-Statement nutzen. Die erste Hälfte, das `if`-Statement ermöglicht es uns, besonderes Verhalten auszulösen, wenn eine logische Bedingung erfüllt ist. Das `else` danach können wir nutzen um jeden anderen Fall zu definieren:

```
greet_someone <- function(name){
  if(name == 'Justus'){
    return(paste0('Hello ',name,'! How extraordinarily nice to see you! I hope you are doing well
  }else{
    return(paste0('Hello ',name,'! Nice to see you!'))
  }
}
```

Das funktioniert zwar:

```
greet_someone('Justus')
```

```
## [1] "Hello Justus! How extraordinarily nice to see you! I hope you are doing well!"
```

[1] Nicht, dass er uns verklagt.

```
greet_someone('Jonas')
```

```
## [1] "Hello Jonas! Nice to see you!"
```

Und bemerkensweiterweise auch in der geschachtelten Form:

```
greet_someone_n_times('Justus')
```

```
## [1] "Hello Justus! How extraordinarily nice to see you! I hope you are doing well!"
## [2] "Hello Justus! How extraordinarily nice to see you! I hope you are doing well!"
## [3] "Hello Justus! How extraordinarily nice to see you! I hope you are doing well!"
```

Aber wirklich gut lesbar ist das nicht unbedingt. Ein gängiger Ansatz, um die Lesbarkeit einer solchen Funktion mit verschiedenen Outputs zu verbessern, ist, nur ein `return`-statement ans Ende der Funktion zu stellen und die Teil-Änderung durch das `if`-statement in einem Objekt abzulegen, das im gemeinsamen `return` genutzt wird. So könnten wir den letzten Teil des Grußes zum Beispiel in einem Objekt namens `text` ablegen und je nach Nutzer anpassen:

```
greet_someone <- function(name){
  if(name == 'Justus'){
    text <- '! How extraordinarily nice to see you! I hope you are doing well!'
  }else{
    text <- '! Nice to see you!'
  }
  return(paste0('Hello ',name,text))
}
```

Ein letzter Trick, um diese Funktion leichter lesbar zu gestalten, ist den Regelfall *vor* das `if`-statement zu stellen und sich so das `else` zu sparen. Da die Anweisung im `if`-statement nur evaluiert wird, wenn der Test positiv aufgeht, ist das Ergebnis der folgenden Funktion äquivalent:

```
greet_someone <- function(name){
  text <- '! Nice to see you!'

  if(name == 'Justus'){
    text <- '! How extraordinarily nice to see you! I hope you are doing well!'
  }

  return(paste0('Hello ',name,text))
}
```

### 3.3.1 Aufgabe

1. Erweitere die `my_var`-Funktion um ein optionales `corrected` Argument, das standardmäßig auf `TRUE` gesetzt ist. Die Funktion soll, wenn dieses Argument auf `TRUE` gesetzt ist, die korrigierte($s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - M_X)^2$);

wenn es auf `FALSE` gesetzt wird die unkorrigierte ($S^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - M_X)^2$) Stichprobenvarianz ausgeben.

Antworten

**1.**

```
my_var <- function(x, corrected=TRUE){
  m_x <- my_mean(x)

  x <- (x - m_x)^2

  factor <- 1 / (length(x) - 1)

  if(!corrected){
    factor <- 1 / (length(x))
  }

  return(factor * sum(x))
}
```

## 3.4 Zufallswerte

Mit R kann man sehr einfach quasi-zufällige[2] Wertreihen erstellen. Aus der Veranstaltung letztes Semester kennen wir ja schon die Funktion `sample`, die aus einer vorgegebenen Urne ziehen kann.

Für unsere Simulations-Probleme gibt es noch eine andere Reihe an Funktionen, die für eine feste Verteilungsklasse zufällige Wertfolgen erstellen können.

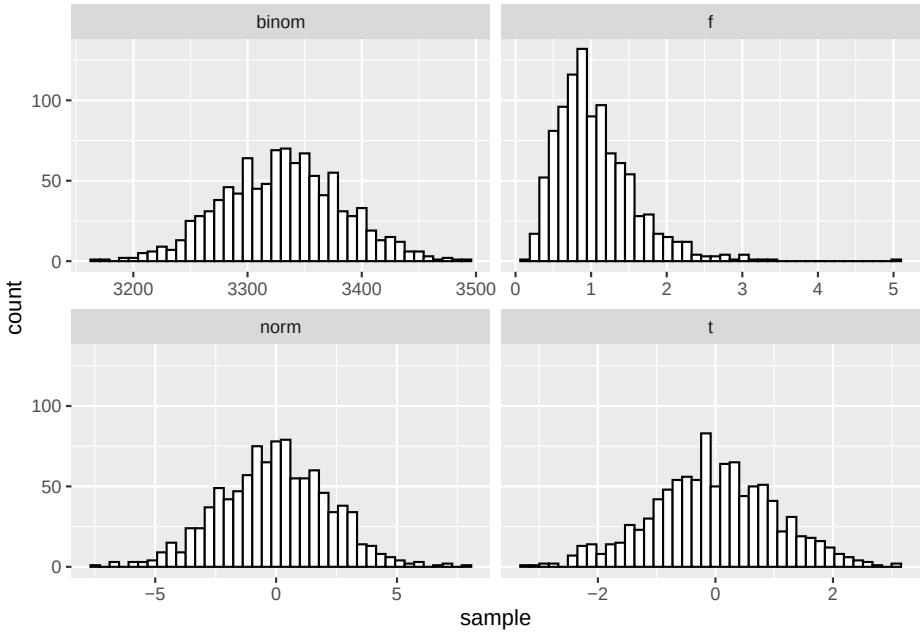
Diese Funktionen werden mit einem `r` für *random* und der Verteilungsklasse aufgerufen. Eine Übersicht der implementierten Verteilungen kann man mit `?Distributions` aufrufen.

Mit diesen Funktionen können wir uns zum Beispiel jeweils 1000 zufällige Werte aus einer $N(0, 5)-$, einer $B(20000, 1/6)-$, einer $t(50)$- und einer $F(12, 36)-$Verteilung ziehen und in Histogrammen darstellen:

```
library(tidyverse)
set.seed(42)
tibble(norm = rnorm(1000,mean = 0,sd = sqrt(5)),
       binom = rbinom(1000, size = 20000,prob = 1/6),
       t = rt(1000,df = 50),
```

[2] Es ist für einen Rechner sehr schwierig, wahren Zufall zu erzeugen. Auf der Hilfeseite `?Random` kann man sich eine Liste der Algorithmen anschauen, die R zur Generation quasi-zufälliger Wertfolgen nutzt. Diese nicht-wirklich-zufällige Natur der Wertfolgen hat aber auch den Vorteil, dass wir "zufällige" Ergebnisse reproduzierbar machen können. mit `set.seed()` können wir einen Startwert für die Zufalls-Generation festlegen, den andere R-Nutzer dann auch wählen können.

```
      f = rf(1000,df1 = 12,df2 = 36)) %>%
  pivot_longer(everything(),
               names_to = 'dist',
               values_to = 'sample') %>%
  ggplot(aes(x = sample)) +
  geom_histogram(fill = 'white',
                 color = 'black',
                 bins = 40) +
  facet_wrap(~dist,scales = 'free_x')
```




### 3.4.1 Aufgabe

Erstelle eine Funktion `gen_distributed_values`, die mit Hilfe von `if`-Statements anhand eines Arguments `distribution` und eines Arguments `n` einen Vektor an Zufallszahlen generiert. Dabei soll `distribution` angeben, welche aus drei möglichen Verteilungen genutzt wird. Denke außerdem an mögliche Verteilungseigenschaften der genutzten Funktionen und füge sie als optionale Argumente zur Funktion hinzu.

Antwort

```
gen_distributed_values <- function(distribution,
                                   n,
                                   df = 1,
                                   lambda = 25,
                                   rate = 5) {
```

```
  if(distribution == 't'){
    return(rt(n, df))
  }else if(distribution == 'pois'){
    return(rpois(n, lambda))
  }else if(distribution == 'exp'){
    return(rexp(n, rate))
  }
  print('Distribution is not implemented, returning zeros')
  return(numeric(n))
}

gen_distributed_values('pois',
                       5,
                       lambda = 3)
```

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

## 3.5 Schleifen

Bei so gut wie allen Simulationsproblemen stehen wir vor der Situation, dass wir eine Operation, zum Beispiel das Generieren einer gewissen Zahl an Zufallswerten und die Berechnung der zugehörigen Teststatistik, mehrere hundert Mal ausführen wollen.

Wir könnten jetzt die Operation mehrere hundert mal in unser Skript schreiben, damit würden wir aber zum Einen Fehler einladen, zum Anderen viel zu viel Lebenszeit verschwenden.

Deswegen gibt es in so gut wie jeder Programmiersprache irgendeine Form von Ausdrücken, die eine *iterative* Wiederholung eines Ausdrucks ermöglichen.

Ein Beispiel für die Anweisung einer solchen iterativen Wiederholung sind Schleifen. In R gibt es davon drei Arten:

- `repeat` - die flexibelste Schleife, die so lange wiederholt bis sie mit `break` unterbrochen wird
- `while` - eine Schleife, die zu Beginn jeder Wiederholung einen logischen Test durchführt und bei Zutreffen die Operation wiederholt
- `for` - die unflexibelste der drei Möglichkeiten. `for` iteriert einen Wert durch einen Vektor oder eine Liste, bis alle Einträge einmal dran waren. Das heißt, dass wir eine feste Laufzeit und damit den einfachsten Umgang haben, weswegen wir auch `for` in diesem Kurs verwenden werden.

Die `for`-Syntax sieht dabei wie folgt aus:

```
for(value in vector){
  do something
}
```

Um jetzt Beispielsweise alle Werte von eins bis zehn durch zu laufen und jeden Wert auszugeben können wir den folgenden Ausdruck benutzen:

```
for(i in seq_len(10)){
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

Wie man an diesem Beispiel schon sehen kann, können wir in der Schleife auf den `i`-Wert zugreifen. Das können wir zum Beispiel benutzen, um die ersten zehn Zahlen der Fibonacci-Reihe zu berechnen, in der jeder Wert die Summe der zwei vorhergegangenen ist:

```
fib <- numeric(10)
for(i in seq_along(fib)){
  if(i<3){ # die ersten zwei Stellen müssen Einsen sein
    fib[i] <- 1
  }else{
    fib[i] <- fib[i-1] + fib[i-2] # nimm die letzten zwei Einträge und summier sie auf
  }
}
fib
```

```
##  [1]  1  1  2  3  5  8 13 21 34 55
```

In diesem Beispiel ist noch ein weiteres Programmier-Prinzip sichtbar, die *Allokation* des Ergebnis-Vektors vor der Berechnung der Werte. Damit ist einfach gemeint, dass wir den leeren `fib`-Vektor erstellt haben, bevor wir mit der Schleife angefangen haben.

Der Grund für dieses Vorgehen ist, dass jedes Anlegen eines Vektors einer bestimtmen Größe ein bisschen Rechenzeit kostet. Wenn wir von vornherein festlegen, wie lang der Vektor werden soll, müssen wir nur einen Vektor anlegen. Wenn wir stattdessen wie im folgenden Beispiel in jeder Iteration den Vektor vergrößern, erstellt R implizit in jeder Iteration einen neuen Vektor.

```
fib <- 1
for(i in seq_len(10)){
  if(i<3){
    fib[i] <- 1
  }else{
    fib[i] <- fib[i-1] + fib[i-2]
  }
}
fib
```

```
##  [1]  1  1  2  3  5  8 13 21 34 55
```

Bei 10 Stellen ist der Unterschied noch nicht wirklich bemerkbar. Wenn wir jetzt aber das ganze in Funktionen verpacken und für längere Sequenzen laufen lassen und die Laufzeit stoppen sehen wir den Unterschied:

```
fib_alloc <- function(n){
  fib <- numeric(n)
  for(i in seq_along(fib)){
    if(i<3){
      fib[i] <- 1
    }else{
      fib[i] <- fib[i-1] + fib[i-2]
    }
  }
  return(fib)
}


fib_no_alloc <- function(n){
  fib <- 1
  for(i in seq_len(n)){
    if(i<3){
      fib[i] <- 1
    }else{
      fib[i] <- fib[i-1] + fib[i-2]
    }
  }
  return(fib)
}
start <- Sys.time()
a <- fib_alloc(100000)
runtime_alloc <- Sys.time() - start
start <- Sys.time()
b <- fib_no_alloc(100000)
runtime_no_alloc <- Sys.time() - start
```

```
runtime_alloc
```

```
## Time difference of 0.03389716 secs
```

```
runtime_no_alloc
```

```
## Time difference of 0.0829339 secs
```

Mehr als die Hälfte der Zeit geht für das Erstellen des neuen Vektors drauf!

### 3.5.1 Aufgabe

In QM-2 habt Ihr im Rahmen des zentralen Grenzwertsatzes gelernt, dass die Verteilungsfunktion der z-Transformation der n-ten Summe einer Reihe von unabhängigen Zufallsvariablen für wachsendes n schwach gegen die Standardnormalverteilung konvergiert.

Schreibe eine Funktion, die für eine gegebene Anzahl an Summen und eine gegebene Verteilungsklasse (und den entsprechenden Parametern inklusive der Stichprobengröße) einen Vektor mit den entsprechenden Summen zurückgibt. Nutze dafür deine Funktion aus der letzten Aufgabe.

Nutze diese Funktion dann um ein Histogramm mit 5000 dieser z-transformierten Summen zu erstellen.

Antwort

```
gen_central_lim_vec <- function(N,
                                distribution,
                                n,
                                df = 1,
                                lambda = 25,
                                rate = 5) {

  ret_vec <- numeric(N)
  for(i in seq_len(N)){
    ret_vec[i] <- sum(gen_distributed_values(distribution,
                                             n,
                                             df,
                                             lambda,
                                             rate))
  }
  return(ret_vec)
}

tibble(sample = scale(gen_central_lim_vec(5000, 'exp', 1000,rate = 5))) %>%
  ggplot(aes(x = sample)) +
  geom_histogram(binwidth = .1,
```

```
                color = 'grey',
                fill = 'white')
```