Introduction into the R programming language

IOS Regensburg

Christoph Rust

January 07/08 2020

Copyright

Prerequisites

- Laptop with R and RStudio installed
- We will make use of the following directories:
  - R_Code/
  - R_Data/
  - R_Graphics/
  - R_Tables/

Aim of this course

- Basic understanding of the R language (and how it differs from using Stata)
- Enable you to find help on your own
- How to prepare data with `dplyr`
- Data visualization with `ggplot2` and base graphics
- Econometric/statistical models in R
- Making use of spatial data
- We will work on several problem sets throughout the course

Agenda

1. General infos
2. Introduction
3. Objects in R and language elements
4. Input/Output
5. Data preparation (dplyr)
6. Data visualization (ggplot)
7. Econometric models in R
8. Further topics

References

Some textbooks

- Ligges, U. (2008), Programmieren mit R, Springer.
- Kleiber, C. & Zeileis, A. (2008), Applied Econometrics with R, Springer.
- Braun, J. & Murdoch, D.(2007), A first course in statistical programming with R
- Hadley Wickham, Advanced R
- Slides: www.christophrust.de/R-intro/slides
- R Website

1. General infos

General intro

R is...

- a free implementation of the programming languate S, purposes of S were:
  - working interactively with data
  - let the user easily become a programmer
  - nice graphics for data visualisation
  - make code reusable
- an interpreted language (the REPL evaluates expressions)
- a *functional* language (functions are first-level objects)
- an *objectoriented* language (has classes and methods)

- a *vectorized* language (objects are internally represented as vectors)

- development of R started in 1992, version 2.0 released in 2005

- influenced by Scheme (a Lisp variant)
- R itself developed by the R core team, many user-contributed packages

Advantages of R

- a free (open-source, GPL2/3) software, full code can be viewed and checked
- very close to (statistical) research
- easily extendable with packages
- runs on almost all platforms

Disadvantages of R

- no graphical user interface (but R-Studio)
- no interactive graphics (but Shiny)
- interpreted language, therefore sometimes slow compared to compiled languages. Compiled code (C/C++, Fortran, Rust) can be included to get around this.

2. Introduction

How to work with R

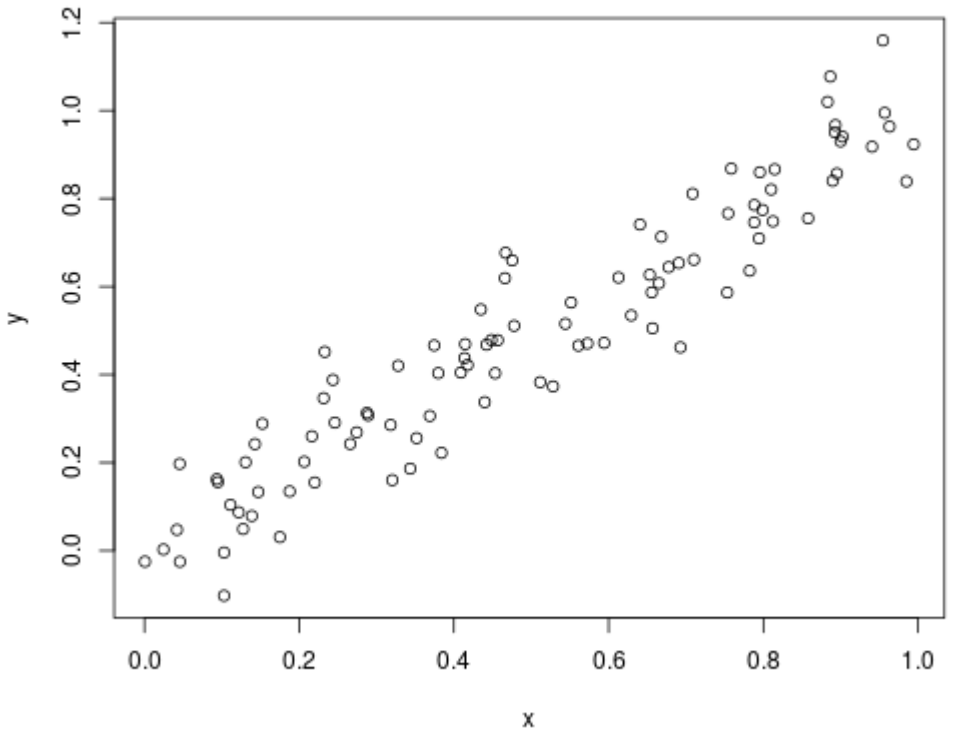
- The program R itself is an interpreter for the R language
- Similar to the Stata console, "commands" are entered and evaluated when *Return* is hit (newline character)
- "commands" are *expressions* (a symbolic description for what to do) which return a *value* when evaluated
- Interpreter stores a history, accessible via arrow keys *(↑, ↓)*

```
> sin(0)
> 2 - 1
> 0/0         # -> NaN (no a number)
> Inf-Inf     # -> NaN (no a number)
> 2 + 3*4     # PEMDAS (paranthes, exponents, multiplicate/divide,
>             #         add/substract)
> 2 +
+   1         # interpreter evaluates only when expression complete
```

```
## [1] 0
## [1] 1
## [1] NaN
## [1] NaN
## [1] 14
## [1] 3
```

Simple graphics:

```
set.seed(123)
x <- runif(100)
y <- x + rnorm(100, sd = 0.1)
plot(x,y)
```



R scripts

There are a lot of simple calculations possible on the command line, but as things get more complicated, *scripts* should be used.

A script is a text file (usually ending with '.R') containing R code.

Example:

```
## change working directory
setwd("C:/Users/Max/R-Code")

## load some date
myData <- read.table("all_important_data.csv",
                     sep = ";", header = TRUE)

## Summary
summary(myData)
```

Editors

Text files are edited with text editors, there are some editors which make the work with R easier:

- R-Studio (used in this course)
    - Closest to a GUI
    - Plot, overview of objects and packages in one window
    - Extra click functions like loading data
    - Highly recommended especially for beginners
- Notepad++ in conjunction with npptor (Notepad++ to R)
    - Flexible editor for all possible programming languages, txt files etc.
    - Flexibly extensible

Editors

- (X)Emacs with ESS
  - Extremely versatile and powerful editor that requires some training
  - Freely configurable
  - Runs on all platforms
  - Editor is suitable for all possible use cases (LaTeX, email, git, ...)

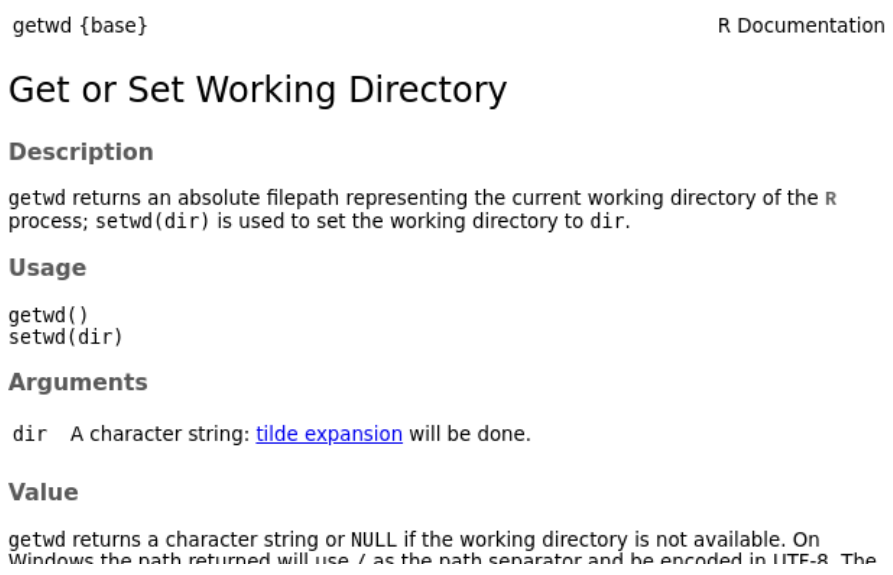
RStudio

- Structure: 4 windows
  - top left: code editor, data.frame view (Ctrl + 1)
  - lower left: R console (Ctrl + 2)
  - top right: Display of objects in the global environment (Ctrl + 8), History (Ctrl + 4)
  - lower right: Help (Ctrl + 3), Plots (Ctrl + 6), Packages (Ctrl + 7),...

Find help to a specific topic

To get help on a *known* function, you can either search for the function in the RStudio Help tab or enter the following in the R console

```
?getwd
```



getwd {base} R Documentation

Get or Set Working Directory

Description

getwd returns an absolute filepath representing the current working directory of the R process; setwd(dir) is used to set the working directory to dir.

Usage

```
getwd()
setwd(dir)
```

Arguments

dir	A character string: tilde expansion will be done.

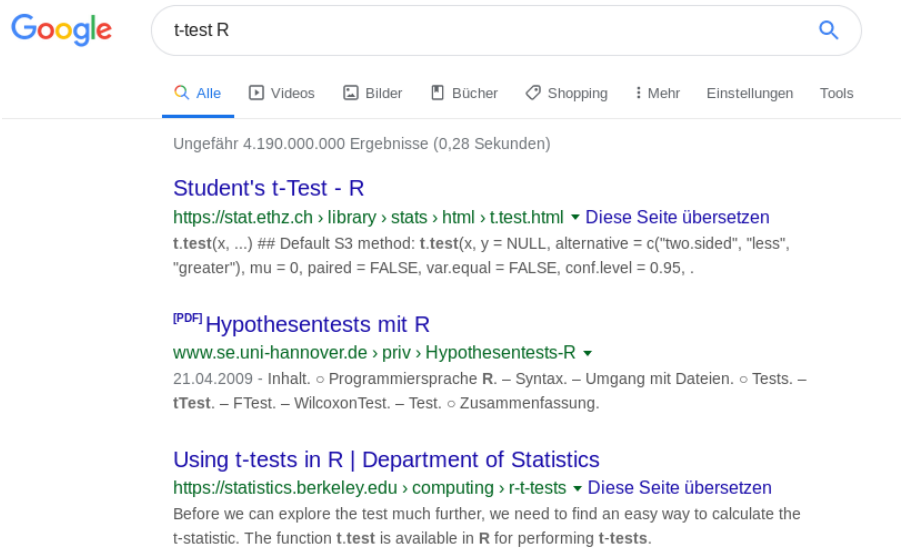
Value

getwd returns a character string or NULL if the working directory is not available. On Windows the path returned will use / as the path separator and be encoded in UTF-8. The

Find help to a specific topic

For example, if you are looking for a *unknown* function (e.g. a function that performs the t-test) then it is best to use Google.

Example:



Find help to a specific topic

- Each package has its own documentation (contains the help files), often vignettes (more detailed explanations)
- stackoverflow.com
- Mailing list *R-help*

R as a calculator

```
1 + 2           # -> 3
1 + (2 * 4)     # -> 9
a <- 3
b <- 3 * a      # -> 9
sqrt(b)         # -> 3
```

Basic operations	+,-,*,/,^
Integer division/modulo	%/%, %%
Extreme and absolute values	max(), min(), abs()
Square root	sqrt()
Round	round(), floor(), ceiling()
Trigonometric functions	sin(), cos(), tan(), asin(), acos(), atan()
Logarithms, exp. function	log(), log10(), log2(), exp()
Sum and product	sum(), prod()

Functions

Allmost all *expressions* make use of *functions*. Operators, Assignments, flow control,... are functions

- Functions are called by entering the function name followed by its arguments in curved brackets, seperated by a comma:
  `functionname(argument1 = value1, argument2 = value2,...)`
- Arguments not neccessarily have to be passed by name:
  `functionname(value1,value2,...)`
- But then, the correct ordering is mandatory
- Very often, functions have default arguments which only have to be specified if one wants to pass a different than the default value

Assignment

The symbol `<-` is used to assign values to variables (symbolic description for objects stored in memory, not variables of a data frame)

- the value of the rhs is stored into the variable named on the lhs
- any object (data structure, function) can be bount to a name by assignment

```
a <- (3 * 4)
(3 * 4) -> a    # the same but not recommended!
a = (3 * 4)     # can also be used but is not equivalent to '<-'
a<-(3*4)        # not so easy to read
```

- For readability, there should always be spaces around the assignment operator.
- Variable names must begin with a letter, but may also contain numbers, periods, and underscores
- It is recommended that the objects be given a uniform naming scheme, more on this later

Environments

- Every variable name is bound to an environment, a data structure on its own that powers *scoping* (next slide)
- Most often we will assign variables in the global environment (`.GlobalEnv`)
- During runtime, functions have their own environment

Scoping

- A set of rules describing *how* (not when) R looks up variables (values of a given symbols)
- `search()` returns the search path, a set of environments. If a variable is not found in the current environment, it is looked up in the next one in the search path

```
f <- function(x) x + z
f(1)     # z not found (neither in function's env nor .GlobalEnv)
z <- 2
f(2)     # now found in .GlobalEnv
```

- More examples to come

Logical operators

Comparison	==, !=, >, >=, <, <=
Operators	! (negation), xor() (exclusive or), &, | (and/or, also vectorized), &&, || (and/or, does not evaluate all operands operands when result already clear)

Examples:

```
TRUE & FALSE   # FALSE
TRUE & TRUE    # TRUE
TRUE | FALSE   # TRUE
!TRUE | FALSE  # FALSE
FALSE && TRUE  # zweites TRUE wird nicht ausgewertet
TRUE && TRUE   # zweites TRUE wird ausgewertet
TRUE || FALSE  # zweites FALSE wird nicht ausgewertet
```

Logical operators

More examples:

```
c(TRUE, FALSE) & c(TRUE, TRUE)  # [1] TRUE FALSE -> vectorized
c(TRUE, FALSE) && c(TRUE, TRUE) # [1] TRUE       -> not vectorized
```

Quantors:

```
a <- c(TRUE, FALSE, TRUE)
b <- c(TRUE, TRUE, TRUE)
any(a)          # [1] TRUE
all(a)          # [1] FALSE
all(b)          # [1] TRUE
```

Logical operators

It is not possible to test `x == NA`, if one wants to check for `NA` one has to use the function `is.na()`

Examples:

```
a <- c(TRUE, NA, TRUE)
a == NA           # [1] NA NA NA
is.na(a)          # [1] FALSE TRUE FALSE
```

Some usefule functions

- `ls()` shows the existing objects in the global environment
- `str()` shows the structure of an object
- `rm()` deletes an object from the global workspace
- `getwd()` shows the current working directory
- `setwd()` changes the working directory
  - Under Windows the path separator is either `/` or `\\`
  - Under Linux/Mac always `/`
- `save()` saves objects as a `.RData` file.
- `load()` loads objects into the global environment
- `list.files()` displays files in the specified directory
- `source()` executes an R-script

Extensibility via R packages

On CRAN there are a lot of different packages for all possible applications. Thus the (relatively small) basic system can be extended at will. R is delivered with some standard packages but for specific topics, packages have to be installed later.

```
install.packages("AER")  # installs not yet available package
library(AER)             # loads namespace of package (exportet objects can
                         # be accessed)
data(CASchools)          # e.g. data frame "CASChools"
?ivreg                   # help for function ivreg
```

- `search()` shows the search path, this includes loaded packages:

```
search()
```

```
##  [1] ".GlobalEnv"             "package:dplyr"          "package:ggplot2"
##  [4] "package:R.utils"        "package:R.oo"           "package:R.methodsS3"
##  [7] "package:RNetCDF"        "package:xaringanthemer" "package:stats"
## [10] "package:graphics"       "package:grDevices"      "package:utils"
## [13] "package:datasets"       "package:methods"        "Autoloads"
## [16] "package:base"
```

If you use Linux, it is likely that R-packages are packaged for your system. Debian, for instance, in the testing branch has most of the CRAN packages, less (and more outdated) are available in the stable branch. I always do

```
sudo apt install -t testing r-cran-{pkgname}
```

for this to work, you have to add testing to your sources list and adjust apt settings:

```
echo 'APT::Default-Release "stable";' \
   | sudo tee -a /etc/apt/apt.conf.d/99defaultrelease
echo 'deb http://ftp.de.debian.org/debian/ testing main contrib non-free' \
   | sudo tee -a /etc/apt/sources.list.d/testing.list
echo 'deb-src http://ftp.de.debian.org/debian/ testing main contrib non-free' \
   | sudo tee -a /etc/apt/sources.list.d/testing.list
sudo apt update
```

Differences to Stata

- Stata (as a language) is more procedural, R functional
- In Stata, you can only work with one data frame in the same time, in R one can create as many objects as one wants to (data frames, estimation objects,...)
- Stata has macros (`` `local' `` (available in do file), and `$global`), R has variables bound to environments (availability has nothing to to with the text files where they were created)
- R is more close to Mata

Exercise 2.1

Create a file `test.R` in your codes folder. This file shall contain a script that assigns to the object `x` the number `5`, and the object `y`, occupied by the number `6`, is created. Before you call this `test.R` file with the `source` function, delete your entire workspace. After that, look at the workspace, calculate the product of the two numbers, then delete the object `x`, save the rest of the workspace in ".RData" format in the R_Data folder.

Note: In general it is smart not to move the work directory back and forth, but only apply the source command to the explicit code folder.

Exercise 2.2*

Test R as a calculator:

1. calculate the value of the sine function at the position 0
2. define `x` as the number 2 and calculate the double of the third power of x

Exercise 2.3

Look for an R package that provides functions to test linear hypotheses in the multiple regression model. Install the package and display the help for a function.

Packages used in this course

To be able to run all the code in these slides, install the following packages:

```
pkglist <- c("AER", "car", "dplyr", "ggplot2", "lmtest",
             "sandwich", "plm", "rgdal", "RNetCDF", "R.utils",
             "readxl", "tidyr", "wbstats", "tmap", "stargazer")
install.packages(pkglist)
```

If you are on debian stable and have testing in your sources list, you can do something like

```
sudo apt install -t testing\
    $(for v in aer car dplyr ggplot2 lmtest sandwich plm rgdal\
    rnetcdf r.utils readxl sf units raster rcolorbrewer\
    viridislite classint htmltools lwgeom; do echo r-cran-$v; done)

R -e 'install.packages(c("tidyr","tmap", "wbstats", "stargazer"))'
```

2. Objects in R and language elements

> To understand computations in R, two slogans are helpful:
>
> - Everything that exists is an object.
> - Everything that happens is a function call.
>
> --- John Chambers

Probably the most important object: function

A function is a program construct that executes a procedure on provided objects and returns a result.

Several function call types are available in R:

- prefix: the majority of functions in R like `sum(a, b)`
- infix: all operators are infix functions, e.g. `2 + 3` or `10^2`
- replace: these functions modify their argument, e.g. `names(x) <- c("first", "second")`
- special: special language elements like `if`, `for`, `while`, `[[`,...

For every function, there is a prefix variant.

```
log(2.3)
sin(2)
2 + 3    # infix
`+`(2,3) # prefix
```

Functions

There are functions with and without side effect:

- Functions without side accept objects and perform an operation on them, and return the result (and nothing else). Example: `log()`
- Functions with side effect also change objects in the global workspace. Example: `setwd()`, `'<-'()` (assignment is also a function!)

⇒ When developing functions, side effects should be avoided if possible (unless they are explicitly desired)

How to define a function

```
## simple function
product1 <- function(x1, x2) x1 * x2

## default arguments
product2 <- function(x1 = 1, x2 = 2) x1 * x2

## curly brackets are useful with more lines of code
product3 <- function(x1 = 1, x2 = 2){
  x1 * x2
}
```

- In the above example, curly braces are not necessary, but as soon as the function performs several operations, they should be used
- Functions either return the last evaluated expression or the argument of the `return()` function.

Function call

```
f1 <- function(x1, x2) x1 + 2* x2
f1                      # print the function definition
                        #    (print-method on the function)
f1()                    # function call, but f1 requires 2 args

f1(x1=2, x2=5)
f1(x1=2, x2=5, y=5)     # three args is too much
```

What happens when you call the function?

`f1(x1 = 1,x2 = 2)`

During the runtime of the function a new environment is created, in which two variables are available, `x1` here with the value 1, `x2` here with the value 2. This is used to calculate the product which is finally returned.

Not all functions require arguments, for example `getwd()`

```
f2 <- function() x1 * x2
x1 <- 2
x2 <- 3
f2              # show function definition
f2()            # function call
f2(x1=2)        # produces an error, function does
                # not accept any argument
f2(2)
```

`f2` has no arguments. This also means that at runtime in the generated environment also the variables `x1` and `x2` are not available. Scoping rules now define that these objects are looked up in the next higher environment. If `x1` and `x2` do not exist in any environment, an error is shown:

```
rm(x1,x2)
f2()            # error
```

Example for a somewhat more complicated function

```
f3 <- function(x1,x2) {
    z <- x1 + x2
    abc <- x1/z
    return(abc)   # return explicitly the value abc and
                  # terminate function
    abc <- 123    # not evaluated any more
}
f3(1,2)             # -> 0.3333333
```

Three dots ellipsis

Something one finds relatively often is the so called *three dots ellipsis* (`...`). It means that the function is designed to take any number of named or unnamed arguments and passes to inner function calls:

```
f4 <- function(x, ...) {
    log(x, ...)^2
}
```

All arguments not named in the function definition of `f4` are passed to `log()` via the `...`:

```
f4(5)
log(5, base=10)^2             # log() has an argument "base"
f4(5, base = 10)              # is passed here
f4(5, base = 10, arg3 = "a")  # arg3 also not an arg of log()...
```

Methods

There are also functions that call methods, especially `print()`, `summary()`, `plot()`. These perform different operations depending on the class of the passed object.

```
plot(cos, -1, 1)  # 'plot' called on object 'cos'
x <- 1:5
y <- 6:10
plot(x, y)        # 'plot' called on two vectors 'x' and 'y'
```

There are two (now also three) class systems in R (S3, S4, and reference classes, the latter closest to OOP).

Look up source code of functions

R is not a black box, see also the article (p. 43) by Uwe Ligges:

> "When looking at R source code, sometimes calls to one of the following functions show up:
>
> `.C()`, `.Call()`, `.Fortran()`, `.External()`, or `.Internal()` and `.Primitive()`. These functions are calling entry points in compiled code such as shared objects, static libraries or dynamic link libraries. Therefore, it is necessary to look into the sources of the compiled code, if complete understanding of the code is required. ... The first step is to look up the entry point in file `$R HOME/src/main/names.c`, if the calling R function is either `.Primitive()` or `.Internal()`."

Exercise 3.1

1. Create a function named "getSquaredSum", which, when entered of two numbers calculates and returns the squared sum of the numbers. Test this function in two ways. Now set the default value of the second argument to zero.

2. 1st version: create a function where in the function body another function is defined and called.
   2nd version: now define the "inner" function outside the outer. Which version do you like better? Check that everything works and gives the same values!

Atomic types

Atomic data types are the building blocks of more complex data structures (vectors, matrices, lists,...)

- `NULL`: empty set
- `logical`: `TRUE`/`FALSE`
- `integer`: -2,-1,0,1,2,...
- `numeric`: real numbers (double precision)
- `complex`: complex numbers
- `character`: character strings
- see also `?typeof`

```
typeof(2.3)    # numeric
typeof(TRUE)   # logical
typeof("abc")  # character
typeof(log)    # special
```

To test for a specific type, use `is.{type}()`, for instance `is.numeric(1.23` returns `TRUE` and to convert (if possible) to a specific type, use `as.{type}()`; `as.character(1.23)` returns the character `"1.23"`.

Vectors

Vectors are the basic structure in R and consist of several elements of an atomic data type. With the function `c()` vectors can be generated:

```
x <- c(1, 2.3, 5, 1)
x <- c(2, 2, 2, x)      # increase vector x
y <- c("Test", "Hallo")
y <- c(y, x)            # everything is converted to 'character',
                        # the lowest type
x <- c(x, NA)           # but NA does not change type
```

Construction of vectors

Several possibilities to construct vectors

- integer sequences: `1:5`, alternatively: `seq(1,5)`
- any sequence: `seq(start, end, by)`
- repetitions: `rep()`

```
2:4               # -> 2,3,4
seq(2,8,2)        # -> 2,4,6,8
rep(2, 4)         # -> 2,2,2,2
x <- 1:3
rep(x, 2)         # -> 1,2,3,1,2,3
rep(x, each = 2) # -> 1,1,2,2,3,3
```

It is possible to give names to the elements of vectors:

```
x <- c(one = 2.4, two = 3, three = 4, last = 2)
```

Vectorized operations

All basic mathematical operations operate *vectorized*

```
c(1,2,3) + c(1,1,1)  # -> 2,3,4
c(1,2) * c(1,4)      # -> 1,8
```

Take care: if both operands in a vectorized operation are not of same lenght, R recycles the shorter to the lenght of the longer vector. Sometimes, you get a warning

```
c(1,2,4) * 2         # 2,4,6 -> second obj is recycled to c(2,2,2)
c(1,2,4) * c(2,3)    # 2,6,8 -> warning
c(1,2,4,8) * c(2,3)  # 2,6,8,24 -> works fine
```

Indexing of vectors

To access elements of vectors, one has several possibilities:

- numerical indexing (works also vectorized): `x[c(2,3,7)]` returns 2nd, 3rd and 7th element of vector `x`
- logical indexing: `x[c(TRUE, FALSE, TRUE, FALSE, FALSE)]` returns 1st und 3rd element from `x` (given that `x` contains 5 elements)
- elements of named vectors can also be accessed by its name: `x["one"]` returns element with name "one"
- `x[-1]` returns all elements but first
- `x[ x > 2 ]` returns all elements larger than 2 (given `x` is of type numeric)

Some useful vector functions

sort()	sort elements increasingly
rev()	revert ordering
rank()	vector with rank of elements
order()	vector with indices that can be used to sort the arg
unique()	removes duplicates
duplicated()	returns logical vector indicating occurrences of duplicates
which()	test which elements fulfill a certain condition

Exercise 3.2

Write a function `vectorSum` that calculates the sum of two vectors and returns as a value (not as print output) the character "The sum is [value]". Test with the vectors:

```
xy <- c(1,2,3)
yx <- c(4,5,6)
vectorSum(xy,yx)   # -> "The sum is [5,7,9]"
```

Note: see the help for the `paste()` function.

Matrices

Matrices are vectors with an additional dimension info:

- construction `matrix(data= NA, ncol=1, nrow =1, byrow = FALSE)`

```
matrix(data = 1:9, ncol = 3, nrow = 3) # column-major
```

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

If `data` is not long enough, it is again recycled:

```
matrix(data = 1, ncol = 3, nrow = 3)
```

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

Matrices - Indexing

Indexing is analogous to indexing of vectors, only that we need column and row index

```
x <- matrix(data = 1:9, ncol = 3, nrow = 3, byrow = TRUE)
x[1,3]    # returns element in 1st row, 3rd column
x[1,]     # returns first row as vector
x[,2]     # returns second column as vector
x[,2, drop = FALSE] # 2nd column as 3 x 1 matrix
x[x>2]    # returns elements of x larger than 3 (as vector)
```

- With `cbind()` and `rbind()` one can stack matrices column- and row-wise
- `dim()` returns the dimensions of the matrix

Some useful functions for matrices

%*%	matrix multiplication
chol()	Cholesky decomposition
diag()	get and set diagonal values of matrix
dimnames()	row and column names
eigen()	Eigen decomposition
qr()	QR decomposition
solve()	invert matrix
svd()	singular value decomposition
t()	transpose matrix

Arrays

Matrices are two-dimensional objects, there are also multi-dimensional objects (arrays): `array(data, dim)`, where `dim` is a vector specifying how large the array is in each dimension.

```
array(1:30, dim = c(3,3,5))
```

is an array of dimension $3 \times 3 \times 5$.

- indexing is the same as with matrices
- basic mathematical operations operate elementwise:

```
A <- matrix(1:6, nrow = 2)
A^2
```

```
##      [,1] [,2] [,3]
## [1,]    1    9   25
## [2,]    4   16   36
```

Exercise 3.3

Calculate for the matrix `A = Mat_A`

```
Mat_A <- matrix(1:9, ncol = 3)
```

and the vector `b = vec_b`

```
vec_b <- 12:14
```

the matrix product $A \cdot b$ and the component product. Explain the differences.

Exercise 3.4

Enter the objects $y = (3, 5, 2, 8, 6, 4, 7)'$ and the matrix X, whose first column consists of ones and whose second column contains the entries $(4, 3, 7, 1, 3, 7, 5)'$, in R.

1. Print for 'X' and 'y' respectively the 3rd observation
2. Calculate the quantities $X'X$, $X'y$, $(X'X)^{-1}$ and the OLS estimator of the linear regression model.
3. Create a function which, given arguments y and X, outputs a vector with the KQ estimate.

Lists

Vectors and matrices have one restriction, namely, all elements have to be of the same atomic type.

A more flexible object is a list. Lists may contain anything. They can be created with the function `list()`.

```
L1 <- list(
  a = 1:3,
  A = matrix(1:9,3,3),
  w = "Hallo!"         # named list with different atomic types
)
```

Lists again may contain lists:

```
L2 <- list(
  a = 1:3,
  l1 = L1 # list in another list
)
```

Lists - indexing

In order to access elements of a list, a pair of two brackes `[[]]` is neccessary, one pair (`[]`) would return only a sublist. Again, indexing can be done numerically, logically or by name:

```
L1[[1]]    # 1,2,3     -> vector
L1[1]      # list(1:3) -> still a list (sublist of L1)
L1[["w"]]  # "Hallo!"
```

Elements of named lists can also be accessed via `$` operator

```
L1$w       # "Hallo!"
```

Exercise 3.5

- Create a named list with three different elements.
- Access the elements by different ways.
- Use the function `str()` on your list
- Now create a list containing a list which again contains a list.

Dataframes

Important data structure: `data.frame`:

- a list with the restriction that all elements have to be of the same length
- also understands matrix indexing, if given two dimensional index
- created with `data.frame()`:

```
Customers <- data.frame(
  FirstName = c("Patrick","Mario","Claudio","Mario"),
  LastName =  c("Meyer","Schröder","Müller","Schmidt"),
  DateBirth = c("1994-03-03","1986-05-21","1978-10-03","1985-07-10"),
  Age = c(25,33,41,34),
  Childs = c(2,3,0,1),
  stringsAsFactors = FALSE      # names shall be represented as
                                # character, not "factor"
)
```

Dataframes

Indexing:

```
names(Customres)          # variable names
Customres$FirstName       # indexing via list-"language"
Customres[ ,1]            # numeric indexing like matrix
Customres[ ,"FirstName"] # by name
Customers[Customers$LastName=="Müller" , ] # logical
```

Add variables to the data frame:

```
Customers$Gender <- c("m", "m", "m", "m")
```

Dataframes

For working with data frames (data preparation) we will use the package `dplyr`.

However, some useful functions for working with `data.frames`:

- `summary()`: summary statistics for every variable
- `head()`: prints first rows of data frame
- `tail()`: prints last rows of data frame
- `attach()`: makes variables accessible in the global environment
- `split()`: splits the data frame into tho parts (horizontally)
- `merge()`: merges two data frames containing different information on the same individuals

Exercise 3.6

Use the just created data.frame "Customers" and do the following:

1. change the column name "FirstName" to "Prename"
2. access the birth data in two different ways (see data.frame-matrix-list similarity).
3. use logical indexing to extract all customers who have 2 or more children.
4. add a new observation (row) (for example yourself) with arbitrary data
5. add a new variable (column) to the data frame
6. select all customers who have more than one child and are younger than 32 years old.

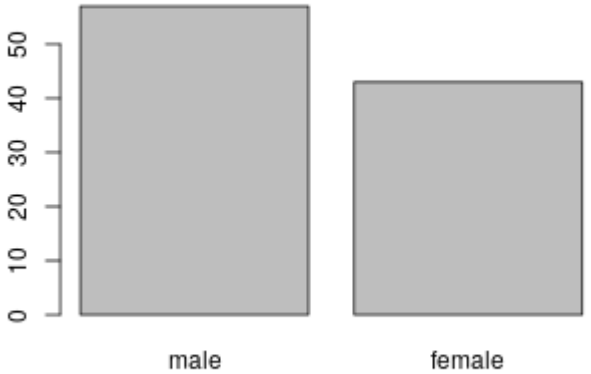
Special data objects useful for real world data

factor

- useful for categorical data
- technically the same as *labelled numeric* in Stata
- useful in regressions (automatic dummy expansion)
- some methods implemented like `summary()`, `plot()`
- created with `factor()`:

```
x <- rep(c(1,2), 4)
factor(x, labels=c("female", "male"))
Z <- c("Yes", "Yes", "No", "Yes", "Maybe", "No",
       "Maybe", "Yes", "Yes", "No")
factor(Z)
```

```r
x <- sample(x=1:2, size=100, replace = TRUE)
x <- factor(x, labels = c("male", "female"))
plot(x)
```




```r
summary(x)
```

```
##   male female
##     57     43
```

Data objects not covered in this curse:

- ordered factors, see `?ordered()`
- date and time, see `?as.Date()` and `?DateTimeClasses`
- time series data, for example see `?ts()` or package `zoo`

Flow control

An essential part of a programming language are constructs to allow conditional evaluation, perform similar tasks many times.

In R, theses constructs are

- conditional evaluation (`if`, `else`)
- loops (`for`, `while`)
- instead of loops, functions of the `apply()` family often offer a more convienient way to repeat some code

Conditional evaluation

```
if ( expr ) {
    ## some code evaluated if as.logical(expr) == TRUE
} else {
    ## some other code
}
```

the whole construct is an expression itself and, therefore, also has a value which is the last evaluated expression inside that construct:

```
result <- if ( 2 > 1 ) {
              2       ## last evaluated expression
          } else {
              1
          }   ## result has value 2
```

The expression in the condition of the if construct should be of length one, otherwise only the first element is taken into account and a warning is shown

```
y <- c(5, 3, 2)
y > 3
if(y > 0) "Look!"   ## evaluates only first entry of y > 0
```

If one wants to check that all entries of a vector fulfil a certain condition, then the quantors `all()` and `any()` are helpful

```
if ( all(y > 0) )  ## all() evaluates to "TRUE", if all entries are TRUE
{
    print("All entries of y are larger than 0")
}
else {
    print("At least one element of y is equal to 0 or smaller!")
}
```

More than two different possibilities:

```
stepfunction <- function(x){
  if ( x <= 0 ) {
      0
  } else if (x <= 5 ) {
      4
  } else {
      6
  }
}
```

see also `?switch()`.

There is also a vectorized `ifelse()` returning a vector

```
x <- c(3, NA, 2)
ifelse( is.na(x), "Missing", "Not Missing")

## absolut value
x <- c(-3, 5, -8, 2)
ifelse( x < 0, -x, x)
```

Exercise 3.7

Write a function `if_test` which gets two arguments, `x` and `y`, and checks whether `x` is numeric and `y` is character and returns "super!" if both conditions are fulfilled and otherwise prints which of the ojects `x`/`y` does not fulfil the required property.

Test your function with

```
if_test(5, "char")

if_test("abc", 2)

if_test(list(1,2), matrix("a", 3, 3))
```

Loops

R has three different loops and two control commands:

- `repeat {code}`: repeats the evaluations of the expression `code` until `break` is called
- `while (cond){ code }`: `code` is evaluated as long as condition `cond` evaluates to `TRUE`
- `for (v in values){ code }`: `code` is evaluated as many times as the number of entries in the object `values`. In the i-th iteration, `v` has the value of the i-th entry of `values`

`break` stops a loop and `next` directly jumps into the next loop run

Examples:

```
vec <- c("One","Two","Three")
for (v in vec) print(v)

for (i in 1:10) {
  print(i + 2)
}

## list with functions:
flist <- list(function(x) x,
              function(x) x+1,
              function(x) x+2)

for (f in flist) print(f(10))
```

Exercise 3.8

Compute the matrix product of the matrices A and B

```
A <- matrix(1:12,4,3)
B <- matrix(1:9,3,3)
```

using loops and check whether the result is equivalent to `A %*% B`

Apply constructs

Loops in R are relatively slow because R is an interpreted language (machine code is generated during run time). Most of the operations performed using loops can be done faster by using an `apply` construct (exception: recursive operations).

- `apply()`: perform operations on subdimensions of matrices or arrays
- `lapply()`: perform operations on vectors or lists and return a list
- further versions of the above two: `mapply()`, `sapply()`, `vapply()`

Example: apply( )

```
A <- matrix(c(2, 6, 3, 4, 5, 7, 8, 4, 1), ncol = 3)
 ## maximum over each row
apply(X = A,            ## array or matrix
      MARGIN = 1,       ## dimension where operation is applied on
                        ##  (1-> rows, 2-> columns)
      FUN = max         ## function to be applied
)

X <- matrix(rnorm(1000), ncol = 10)

apply(X, 2, var)        ## variances of columns
```

Example: `lapply()`

```
List <- list(a=c(4,8,7),
  b=seq(0,100,5),
  c=c(TRUE,TRUE,FALSE,TRUE)
)

ListSum <- lapply(List, sum)
class(ListSum)       ## again a list
```

Sometimes, the result of such an operation is relatively simple and can be saved in a more simple data structure:

```
ListSum <- sapply(List, sum)
class(ListSum)         ## "numeric"
```

If the function applied requires additional arguments, they can be passed via the `...`:

```
## function with several args
funnyFun <- function(x, m, std)
    sum(x)/( 2 * rnorm(1, mean=m, sd=std))

## 'm' und 'std' can by passed to funnyFun via '...':
sapply(X=List, FUN=funnyFun, m=2, std=5)

## any function can be applied, e.g. plot()
par(mfrow=c(length(List),1))
lapply(List, plot, main="Title", type="l", lwd=2)
```

Any operation is a function:

In Stata you propably did create macro names dynamically, depending on the context

```
for var in varlist _all {
    local mean`var` = mean(`var`)
}
```

In R, you can do the same using `assign()` and `get()`:

```
## equivalent assignments:
x <- 2
assign("x", 2)
assign(objectName, 2)

## get the value of object "x"
get(objectName)   ## -> 2

for (var in names(data)){ ## var is a character
    assign(paste0("mean", var), mean(data[,var]))
}
```

Error handling

Sometimes, expressions may raise an error and you do not want that the execution of your script stops:

```
value <- tryCatch({
  ## some code to be evaluated
  expr
}, warning = function(w){
  ## code evaluated if evaluating of expr leads to a warning
  print(w)
}, error = function(e){
  ## code evaluated if evaluating of expr leads to an error
  print(w)
})
```

4. Input/Output

Reading data into R workspace

Several opportunities, depending of the available file format:

- if data available as R-image (`.RData`), the funtion `load()` loads the objects in the image into the global environment
- very often, data sets are available as structured text files, for instance, the data frame `Customers`from above could be saved in a text file with the content

```
"FirstName";"LastName";"DateBirth";"Age";"Childs"
"Patrick";"Meyer";"1994-03-03";25;2
"Mario";"Schröder";"1986-05-21";33;1
"Claudio";"Müller";"1978-10-03";41;0
"Mario";"Schmidt";"1985-07-10";34;1
```

To read such text files, the functions `read.table()`, `read.csv()`, `read.csv2()`, `read.delim()` and `read.delim2()` are available, the latter are wrappers of the first one, each with differnt defaults

Reading data into R workspace

- `xls`/`xlsx`-files: package`readxl`has functions `read_xls()` and `read_xlsx()`
- to read files of other statistical software (Stata, SPSS, Eviews, SAS,...), there are the packages `foreign` and `haven` with appropriate functions

Example

```
## download zip archive with some data examples
curl::curl_download(url = "http://www.christophrust.de/example_data.zip",
                    destfile = "example_data.zip")
## unpack zip
res <- unzip("example_data.zip")

install.packages("readxl", "haven") ## install packages readxl and haven
library(readxl)

excel_data <- read_xls("Africa.xls", skip = 7,
                       col_names = c("country","pop", "larea","pop_dens",
                                     "gdpppp", "gdppcppp","gdpgrwth"))

dax <- read.csv("dax.csv") ## correct defaults for sep, dec
schools_treat <- haven::read_dta("TreatmentSchools.dta") ## namespace via ::
```

Reading data into R workspace

- for reading spatial data (ESRI shape,...), the package `rgdal` has the function `readORG()`:

```
install.packages("rgdal")
library(rgdal)

kreise <- readOGR("vg2500_krs.shp")

class(kreise)
str(kreise, max = 2)
```

Writing data from R workspace to file

- for almost every function that reads data, there is a function for the reverse direction: `write.csv()`, `write.table()`, `haven::write_dta()`,...

More input/output

- there are also more low-level function for interacting with text/binary files: `readLines()`/`writeLines()`, `scan()`/`write()`,...
- for connecting to data bases, there is the `odbc` package
- network ressources can be accessed with `url()`, compressed files `unz()`,...

See also `?connections()`

5. Data preparation (dplyr)

tidyverse

For working with data, a collection of several R packages is very useful: tidyverse. The following packages are part of the tidyverse:

- dplyr: "Grammar of Data Manipulation"
- ggplot2: "Grammar of Graphics"
- readr: "fast and friendly way to read rectangular data"
- tibble: "A tibble, or tbl_df, is a modern reimagining of the data.frame"
- tidyr: "create tidy data. Tidy data is data where:

    1. Every column is [a] variable.
    2. Every row is an observation..
    3. Every cell is a single value."

- purrr: "enhance R's functional programming toolkit"

Pipe operator

R is a functional language and many operations consist of compositions of different functions (nested function call):

```
f <- function(x) x * 2
g <- function(x) x + 10
h <- function(x) x ^ 4
a <- 2
h(g(f(a)))         # (2*a + 10)^4 -> 38416
```

The proplem with that is bad readability. Therefore, `dplyr` makes heavy use of the pipe operator `%>%`. In particular for data preparation this often is very useful.

The composition from above can be written using `%>%` as

```
library(dplyr)

a %>% f %>% g %>% h
```

Pipe operator

The pipe operator forwards the result (or value of the expression) on the LHS (by default) as first argument of the function (or expression) on the RHS.

The value of the LHS can also be accessed via the expression `.`:

```
2 %>%
    log(16, base = .)
```

The RHS can also be an expression using `.`:

```
2 %>% {
    a <- .^2
    a * 5
}
```

dplyr

`dplyr` provides several functions for manipulating data frames:

- `mutate()`: add new variables to data frame
- `select()`: select (extract) columns (variables) of the data frame (stata: `keep varlist`)
- `filter()`: extract rows (observations) from data frame (stata `keep if`)
- `arrange()`: sort observations
- `summarise()`: condense values of a variable to a single value (stata `collapse`)
- `group_by()`: perform operations on groups (stata: `by`)
- `join()`: merge two data frames

For most of these functions there are additional helper functions (e.g. for selecting variables by regex)

Tidy data

For working with packages from the `tidyverse`, the data should be *tidy* (every obs is a row, every variable a column) or *long*. Very often however, data is published in *wide* format.

To get from `wide` to `long` there is the function `pivot_longer()` from `tidyr` (>=1.0.0):

```
library(tidyr) ## at least version 1.0.0
data_wide <- data.frame(id = c(1,2,3,4),
                        wage90 = c(12,13,14,11),
                        wage95 = c(14,16,13,18))

data_long <- data_wide %>%
  pivot_longer(cols = starts_with("wage"), # columns with values
               names_to = "year",          # name of new variable
               values_to = "wage")         # name of variable with values

class(data_long) # tbl_df, tbl, data.frame
```

Real data example

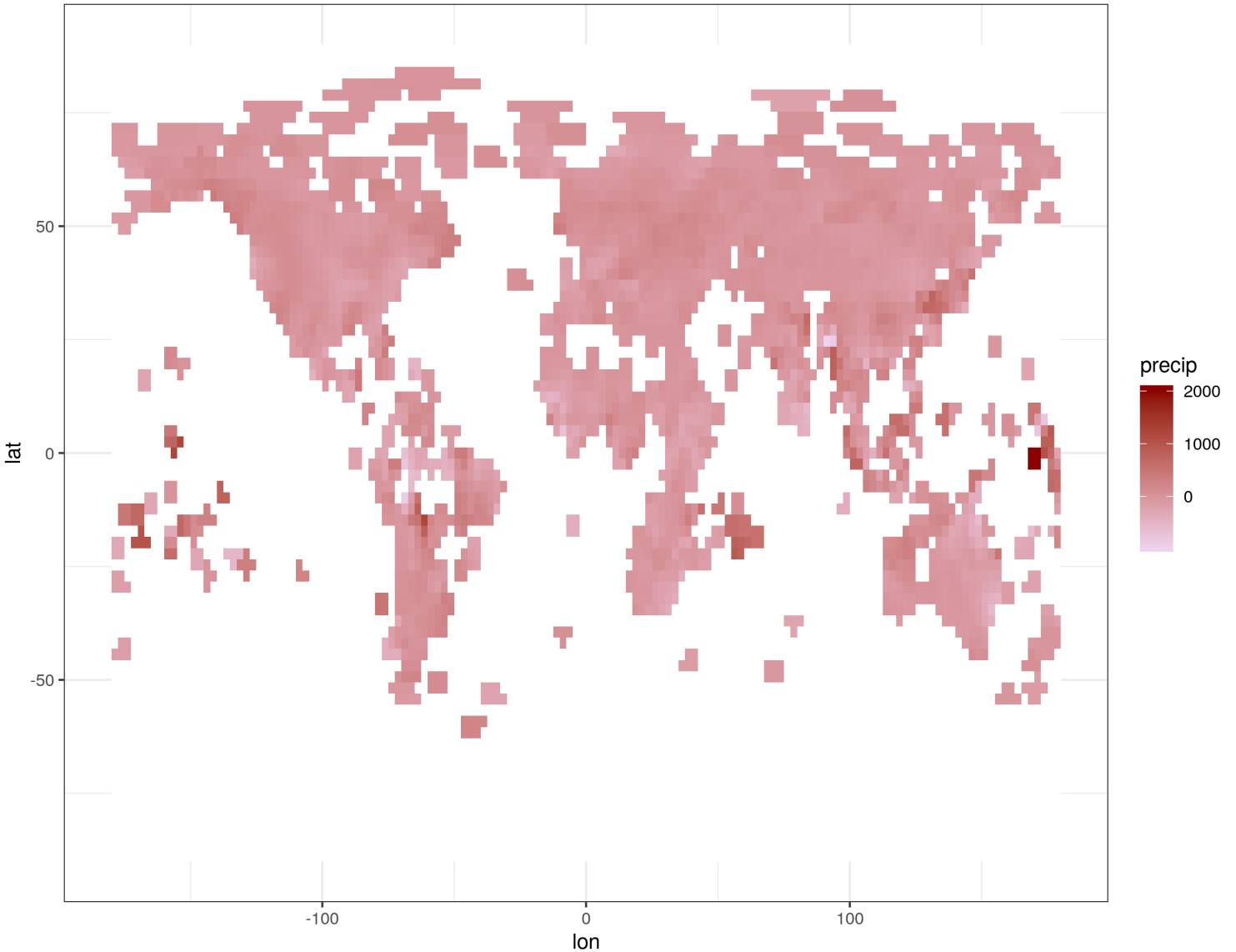
```
library(tidyr)

county_elections <-
    read.table("btw_gemeinden.csv", skip = 10,
               colClasses = c("character", "integer","character",
                              rep("numeric", 10 )),
               sep = ";", dec = ",", na.strings = c("-", "DG"),
               quote = "",  nrows = 41654) %>%
    setNames(c("date", "ags_gem", "name", "electorate", "participation",
             "vote_tot", "CDU/CSU", "SPD", "GRÜNE",
             "FDP", "DIE LINKE", "AfD", "others")) %>%
    pivot_longer(cols = c("CDU/CSU", "SPD", "GRÜNE",
                          "FDP", "DIE LINKE", "AfD", "others"),
                 names_to = "party", values_to = "votes") %>%
    filter(ags_gem > 1000000 & ags_gem < 17000000) %>%
    mutate(ags_kkz = trunc(ags_gem / 1e3),
           date = as.Date(date, format = "%d.%m.%Y"),
           pary = factor(party)) %>%
    mutate(year = as.numeric(format(date, "%Y"))) %>%
    group_by(party, year, ags_kkz) %>%
    summarize(votes_share = sum(votes, na.rm = TRUE)/sum(vote_tot, na.rm = TRUE))
```

Get GDP data for German Kreise

```
county_gdp <-
    read.table("gdp_counties.csv", skip = 10, sep = ";",
               dec = ",", na.strings = c("-", "DG", "."),
               quote = "",
               colClasses = c("integer", "integer", "character", "numeric",
                              "NULL", "numeric", rep("NULL", 8))) %>%
    setNames(c("year", "ags_kkz", "name", "gdp", "gdppc")) %>%
    filter(ags_kkz > 1000 & ags_kkz < 20000) %>%
    mutate(year = as.numeric(year))

## now join both data sources
county_data <- county_elections %>%
    left_join(county_gdp)
```

Data from NASA (NetCDF)

```
library(RNetCDF)
library(R.utils)
library(ggplot2)

url <- "https://data.giss.nasa.gov/pub/precipdai/precip1900-1988.nc.gz"
curl::curl_download(url,
                    destfile = "precip1900-1988.nc.gz")
gunzip("precip1900-1988.nc.gz")

p <- (res <- read.nc(open.nc("precip1900-1988.nc")) %>% {

    ## make a long data from the included array
    df <- data.frame(expand.grid(.$lon, .$lat, .$month, .$year),
                     as.vector(.$precip))

    colnames(df) <- c("lon", "lat", "month", "year", "precip")
    df

}) %>%
    mutate(precip = na_if(precip, -99999)) %>%
    filter(year == 1980 & month == 13) %>%     ## year average
    ggplot(aes(lon, lat, fill = precip)) +
    geom_tile(na.rm = TRUE) +
    scale_fill_continuous(low="thistle2", high="darkred", na.value="white") +
    theme_bw()
p
```


lat
50
0
-50
-100
0
100
lon
precip
2000
1000
0

World Bank Data

For loading data from, for instance WDI, there is the package `wbstats`:

```
library(wbstats)

## Population, total
wb(indicator = "SP.POP.TOTL",
   country = c("DEU", "FRA", "GBR"), startdate = 2000, enddate = 2015) %>%
    head()
```

```
##   iso3c date    value indicatorID         indicator iso2c country
## 1   DEU 2015 81686611 SP.POP.TOTL Population, total    DE Germany
## 2   DEU 2014 80982500 SP.POP.TOTL Population, total    DE Germany
## 3   DEU 2013 80645605 SP.POP.TOTL Population, total    DE Germany
## 4   DEU 2012 80425823 SP.POP.TOTL Population, total    DE Germany
## 5   DEU 2011 80274983 SP.POP.TOTL Population, total    DE Germany
## 6   DEU 2010 81776930 SP.POP.TOTL Population, total    DE Germany
```

Exercise 5.1

Use the data frame CPS1985 available as csv at
`www.christophrust.de/cps1985.csv` and

1. add to the data frame a variable log_wage by using `mutate()` and the pipe operator
2. compute for the interaction of the groups `gender` and `occupation` average log wages
3. filter for both men and women the observation with highes wage

6. Data visualization (ggplot)

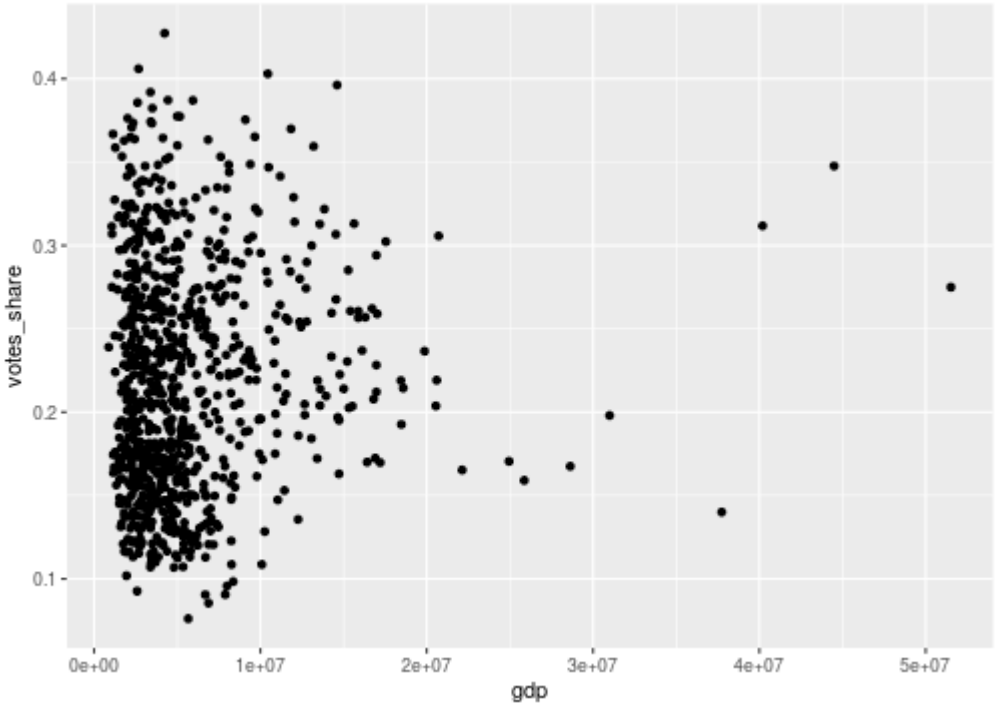
Grammar of Graphics

To build up a graph for visualizing data, the following generic components can be used:

- data, provided as *tidy* data.frame
- aesthetic: which variables do you want to plot and what is their role
- geom: what do you want to plot? points, lines, polygons,...
- scale: how are data values graphically represented? colors, axis scaling,...
- stat: is the data to be transformed?
- facet: plot small subfigures for grouping variable

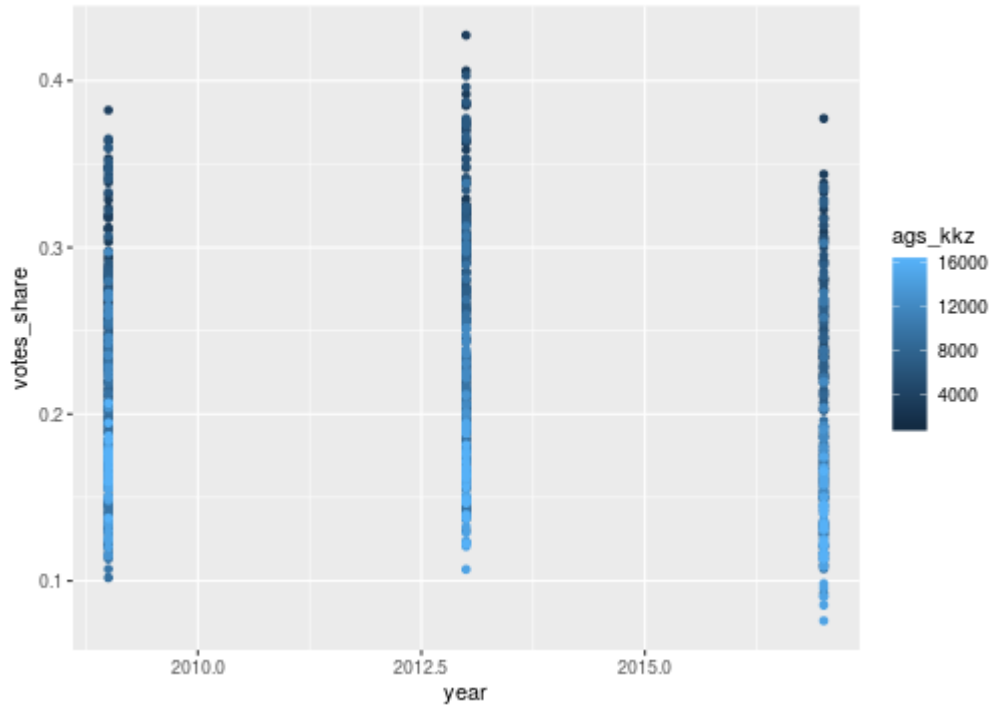
Let's use this:

```
library(ggplot2)
library(dplyr)

county_data %>%
    filter(party == "SPD") %>%
    ggplot(aes(x = gdp, y = votes_share)) +
    geom_point()
```



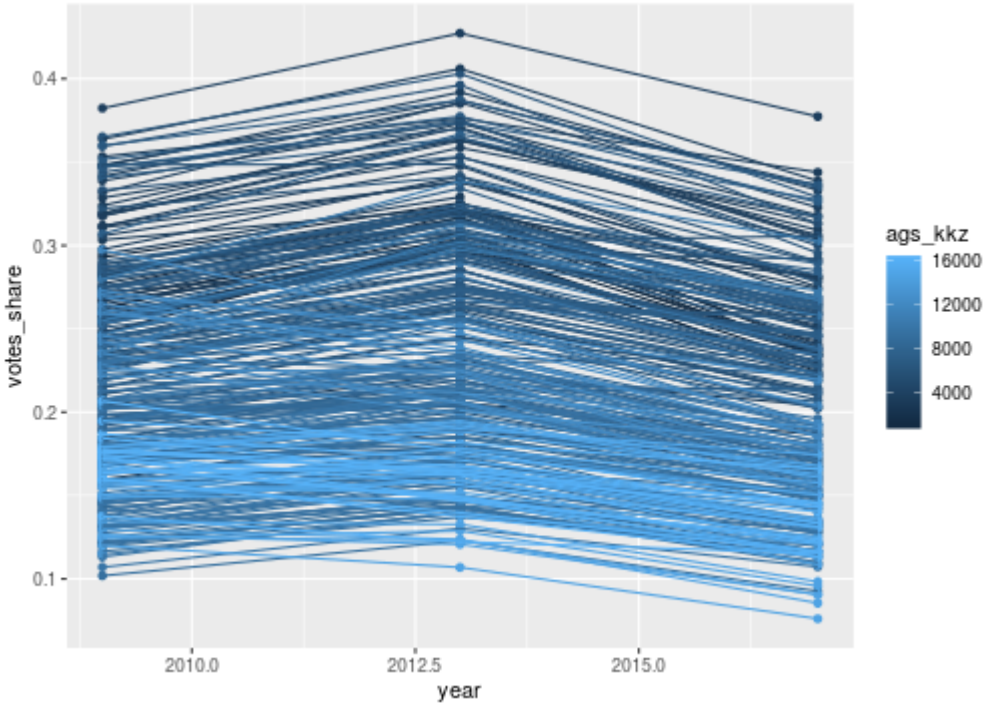
Let's use this:

```
library(ggplot2)
library(dplyr)

county_data %>%
    filter(party == "SPD") %>%
    ggplot(aes(y = votes_share,
               x = year, group = ags_kkz, colour = ags_kkz)) +
    geom_point()
```



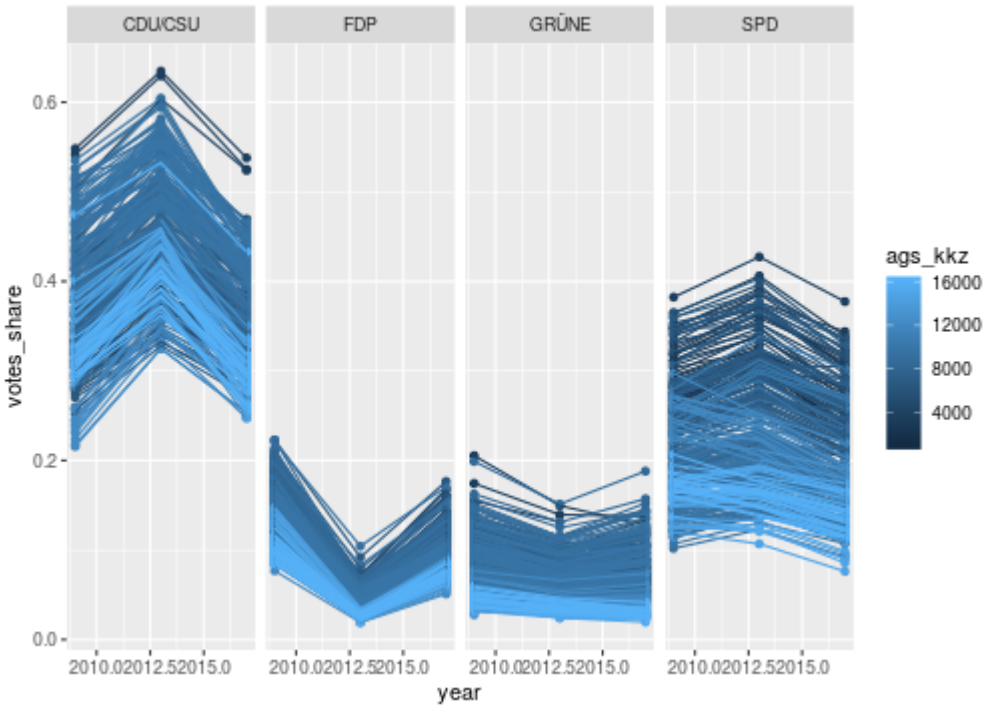
Let's use this:

```
library(ggplot2)
library(dplyr)

county_data %>%
    filter(party == "SPD") %>%
    ggplot(aes(y = votes_share,
               x = year, group = ags_kkz, colour = ags_kkz)) +
    geom_point() +
    geom_line()
```



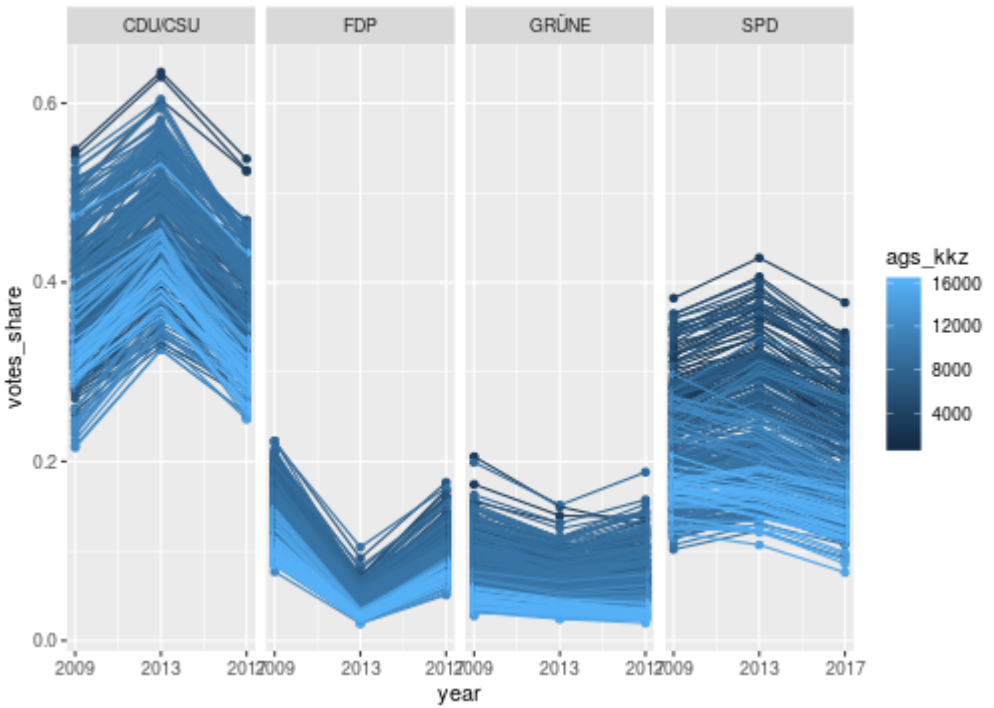
Let's use this:

```
library(ggplot2)
library(dplyr)

county_data %>%
    filter(party %in% c("SPD", "CDU/CSU", "FDP", "GRÜNE")) %>%
    ggplot(aes(y = votes_share,
               x = year, group = ags_kkz, colour = ags_kkz)) +
    geom_point() +
    geom_line() +
    facet_grid(.~party)
```



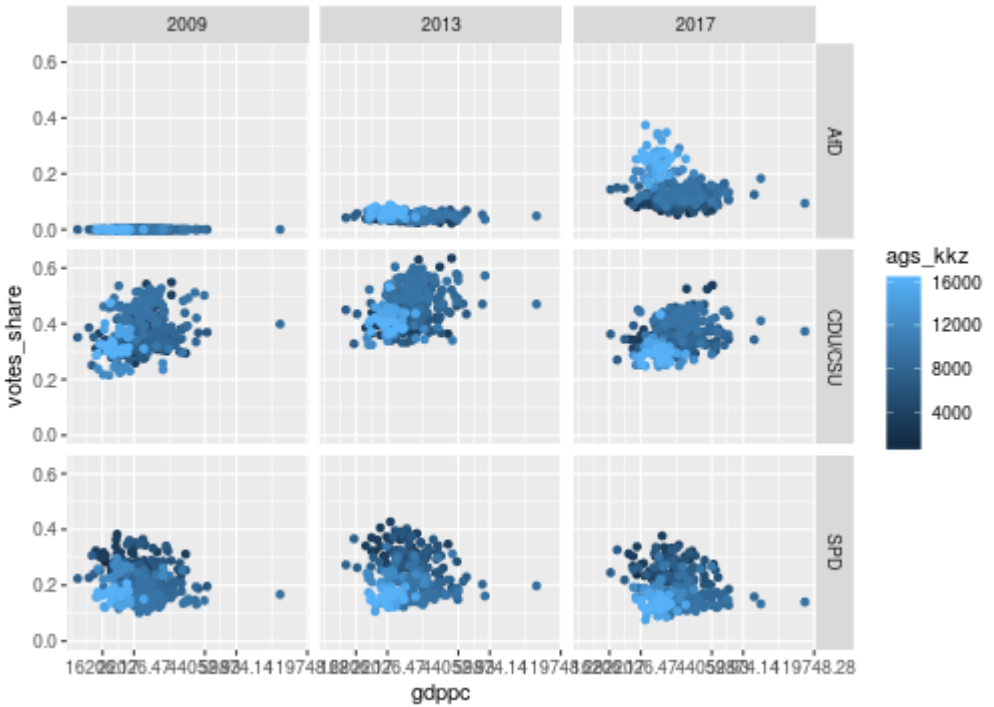
Let's use this:

```
library(ggplot2)
library(dplyr)

county_data %>%
    filter(party %in% c("SPD", "CDU/CSU", "FDP", "GRÜNE")) %>%
    ggplot(aes(y = votes_share,
               x = year, group = ags_kkz, colour = ags_kkz)) +
    geom_point() +
    geom_line() +
    facet_grid(.~party) +
    scale_x_continuous(breaks = c(2009,2013, 2017))
```



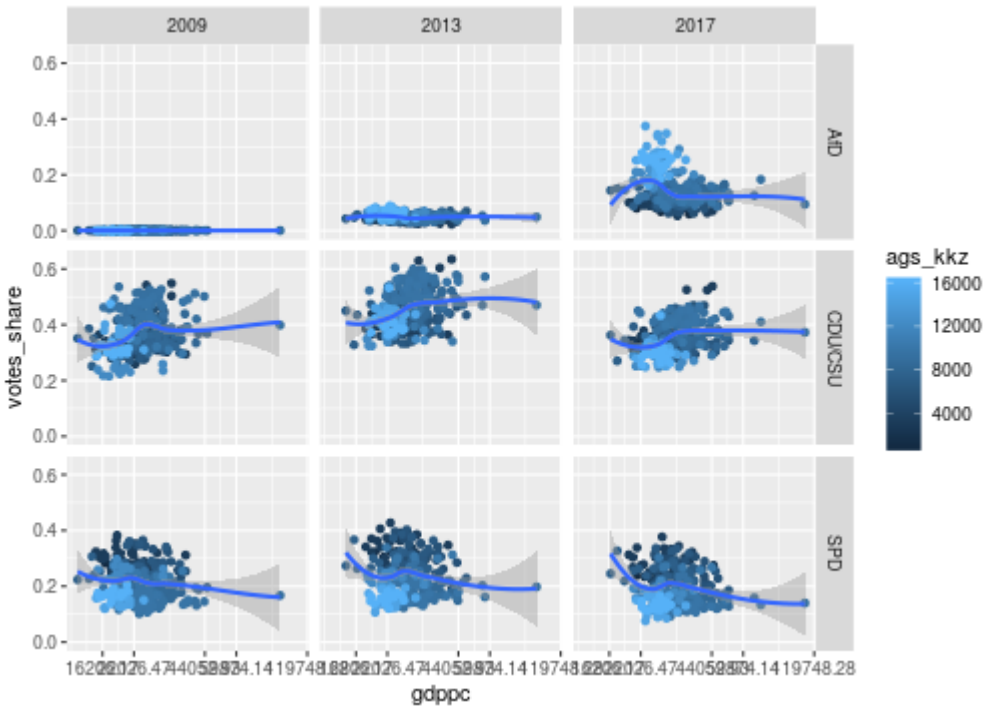
Let's use this:

```
library(ggplot2)
library(dplyr)

county_data %>%
    filter(party %in% c("SPD", "CDU/CSU", "AfD")) %>%
    ggplot(aes(y = votes_share,
               x = gdppc, group = ags_kkz, colour = ags_kkz)) +
    geom_point() +
    facet_grid(party~year) +
    scale_x_continuous(trans="log")
```

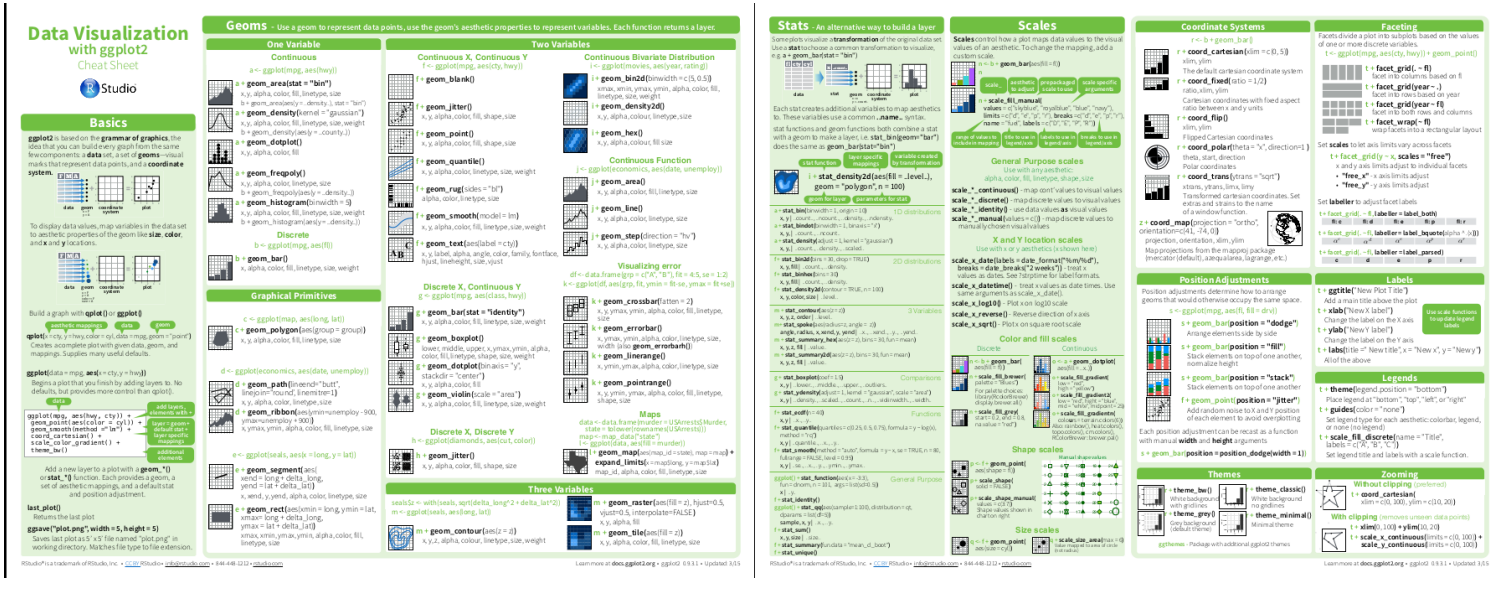


Let's use this:

```
library(ggplot2)
library(dplyr)

county_data %>%
    filter(party %in% c("SPD", "CDU/CSU", "AfD")) %>%
    ggplot(aes(y = votes_share,
               x = gdppc)) +
    geom_point(aes(colour = ags_kkz)) +
    geom_smooth() +
    facet_grid(party~year) +
    scale_x_continuous(trans="log")
```



For a more systematic overview over different geoms, stats, scales, see the ggplot2 cheatsheet



Save plots

```
library(ggplot2)
library(dplyr)

county_data %>%
    filter(party %in% c("SPD", "CDU/CSU", "AfD")) %>%
    ggplot(aes(y = votes_share,
               x = gdppc)) +
    geom_point(aes(colour = ags_kkz)) +
    geom_smooth() +
    facet_grid(party~year) +
    scale_x_continuous(trans="log")

ggsave("R_Graphics/votes_gdp_by_party.pdf")
```

Exercise 6.1

Use the data `btw2017.RData` available in the zip archive `example_data.zip` which contains for all German "Wahlkreise" the results (Zweitstimmen) of the Bundestagswahl 2017 for all parties that entered into the Bundestag. With that data do the following:

1. Plot a bar chart for a Wahlkreis of your choice. Hint: Check out the differences between `geom_bar()` and `geom_col()`.
2. Change the color of the bars.
3. Draw a scatter plot for the variables `AnteilStimmen` and `ProKopfBIP`
4. Draw the above scatter plot for each of the Parties using a `facet`
5. Change the theme of the plot.
6. Draw a bar chart depiciting how many times each party performed best in all "Wahlkreise".

7. Econometric models in R

formula objects

One of R's design goals was to make estimation of statistical models very simple. The technical implementation for that interface are `formula` objects.

A `formula` object is a symbolic description of a statistical model.

Example: the econometric model

$$y_i = \beta_0 + \beta_1 x_{1,i} + \beta_2 x_{2,i} + u_i$$

can be represented as a `formula` object by

```
y ~ x1 + x2
```

Important symbols

- `~`: defines what is modeled (lhs) by what (rhs): `y ~ predictors`
- `+`: `predictors` is given as a group of terms (variables, transformations,...), seperated by a `+`
- `:`: generates interactions
- `*`: `a*b` shorthand for `a + b + a:b`
- `^`: `(a + b)^2` is equivalent to `(a + b)* (a + b)`
- `|`: multiple models or instrumentation (in `ivreg()` from package `AER`)

- Attention: `y ~ x1 + x2^2` does not represent the model $y_i = \beta_0 + \beta_1 x_{1,i} + \beta_2 x_{2,i}^2 + u_i$!

- for aritmetic tranformations one has to put it inside the brackets of `I()`: `y ~ x1 + I(x2^2)`

- factor variables are automatically expanded into dummy variables

Estimate linear models

The workhorse for estimating linear models is the function `lm()`. The value of this function is an `lm` object with several methods: `summary()`, `plot()`, `predict()`,...

`-1` in a formula removes the constant term:

```
county_data %>%
    lm(votes_share ~ -1 + gdp + party, data = .) %>%
    summary()
```

Output

```
Call:
lm(formula = votes_share ~ -1 + gdp + party, data = .)

Residuals:
      Min        1Q    Median        3Q       Max
-0.177643 -0.039212 -0.009715  0.028975  0.311092

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
gdp            -3.150e-12  1.608e-10   -0.02    0.984
partyAfD        6.314e-02  2.185e-03   28.89   <2e-16 ***
partyCDU/CSU    3.933e-01  2.185e-03  179.96   <2e-16 ***
partyDIE LINKE  9.279e-02  2.185e-03   42.46   <2e-16 ***
partyFDP        9.625e-02  2.185e-03   44.04   <2e-16 ***
partyGRÜNE      7.634e-02  2.185e-03   34.93   <2e-16 ***
partyothers     6.101e-02  2.185e-03   27.91   <2e-16 ***
partySPD        2.175e-01  2.185e-03   99.51   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.059 on 6110 degrees of freedom
  (623 observations deleted due to missingness)
Multiple R-squared:  0.9056,    Adjusted R-squared:  0.9054
F-statistic:  7323 on 8 and 6110 DF,  p-value: < 2.2e-16
```

More realistic model:

Transformation `log()` not neccessarily has to go into `I()`:

```
county_data %>%
    filter(year == 2017) %>%
    mutate(BuLa = factor(trunc(ags_kkz/1000))) %>%
    lm(votes_share ~ -1+ log(gdppc)*party + BuLa:party, data = .) %>%
    summary()
```

Access regression results

- several functions available to access the results of the regression, `coefficients()`, `fitted()`, `residuals()`,...
- the objects `lm` and `summary.lm` are lists and we can look at them with `str()` and extract whatever we want by using list indexing

```
suLm <-
    (lmObj <- county_data %>%
         filter(year == 2017) %>%
         mutate(BuLa = factor(trunc(ags_kkz/1000))) %>%
         lm(votes_share ~ -1+ log(gdppc)*party + BuLa:party, data = .)) %>%
    summary()

str(lmObj)    ## structure of lm object
str(suLm)     ## structure of summary.lm object

# elements in lm object
lmObj$coefficients
lmObj$residuals
lmObj$fitted

# summary.lm offers more stuff:
suLm$coefficients      ## matrix with estimate, std-dev, t-stat, pvalue in colums
suLm$sigma             ## estimate of residual standard error
suLm$adj.r.squared     ## R^2 of regression
```

Exercise 7.1

Use the growth data frame available here and

1. After inspecting the data frame, specify several models for economic growth
2. Take one of your models and use apply the `summary()` method on it
3. Access estimated coefficients, standard errors, p-values, sum of squared residuals and adjusted R^2.

Model diagnosis und tests

Heteroskedasticity

graphically:

```
uhat_AIC <- resid(lmObj)
yhat_AIC <- fitted(lmObj)
plot(yhat_AIC,uhat_AIC^2, main="Squared Residuals vs. Fitted Values")
```

statistical test (Breusch-Pagan, White):

```
# Breusch - Pagan
bptest(lmObj)

# White Test
summary(lm(I(uhat_AIC^2)~ yhat_AIC + I(yhat_AIC^2))
        )$r.squared * length(yhat_AIC) %>%
  pchisq(df = 2, lower.tail = FALSE)
```

If heteroskedastic errors, use robust standard errors for tests

```
library(sandwich)
library(lmtest)  ## for coeftest

vcov.robust <- vcovHC(lmObj, type = "HC")
coeftest(lmObj,vcov=vcov.robust)
```

Further diagnosis plots

```
plot(lmObj)
```

Hypothesis tests

- a general function is `linearHypothesis()` from the package `car`

```
library(car)

## symbolically:
linearHypothesis(lmObj, "log(gdppc)=0")

## Another representation: R %*% beta = r
R <- numeric(length(lmObj$coef))
R[1] <- 1    # 1st entry select 1st coefficient
r <- 0       # test whether equal to zero
linearHypothesis(lmObj, hypothesis.matrix = R, rhs = r)

## Heteroscedasticity robust standard errors:
linearHypothesis(lmObj, "log(gdppc)=0",vcov=vcov.robust)
```

Several hypothesis at the same time:

```
## symbolically:
linearHypothesis(lmObj, c("log(gdppc)=0", "partyFDP = 0"))
```

Confidence intervals

```
confint(lmObj,parm=c("log(gdppc)"),level=0.9)
```

Prediction

```
## New observation
X0 <- data.frame( party = "SPD",
                  year = 2017,
                 gdppc = 20000,
                 BuLa = "9")

rownames(X0) <- c("Land 1", "Land 2")

## Prediction plus prediction interval
predict(lmObj, newdata=X0, se.fit=TRUE, interval="prediction")
```

Several extensions

Dynamic regression

The package `dynlm` has the function `dynlm()` which is a similar function like `lm()` but allows to specify lags/leads in the formula:

Philipps curve example:

```
data.us <- read.csv("data_us.csv")[,-1]
PhilData <-ts( subset(data.us,select=c("inf","ur")), start=c(1948,1), frequency=4)

philCurve <- dynlm( inf ~ ur + L(ur,3) + L(inf,1:3) ,data=PhilData)
summary(philCurve)
```

Generalized linear models

The function `glm()` is a quite general function which can estimate several types of generalized linear models, such like:

```
## import some data
diabetes <- read.table("diabetes2.csv", header = TRUE, sep= ",", dec = ".")

## logit
logitFit <- glm(Outcome ~ ., data = diabetes, family = binomial(link = "logit"))
summary(logitFit)

## probit
probitFit <- glm(Outcome ~ ., data = diabetes, family = binomial(link = "probit"))
summary(probitFit)
```

Further possibilities: see `?family()`

Panel models

To work with panel data models, the package **plm** is a natural starting point, see also the Section *Panel data models* in the CRAN Taskview Econometrics.

```
library(plm)

data("Grunfeld")

## index (i, t) in first two colums, nothing to say to plm
## about panel struture

## Random effects model (individual FE)
pm_re <- plm(inv~value+capital, data=Grunfeld, model="random")
summary(pm_re)

## Fixed effects model (time FE)
pm_fe <- plm(inv~value+capital, data=Grunfeld, model="within", effects = "time")
summary(pm_fe)

## Further models: see
?plm
```

See also vignette Panel data econometrics in R.

Export regression tables

Several packages can do that. I mostly use `stargazer`:

```
library(stargazer)

county_data %>% {
    m1 <<- lm(votes_share ~ log(gdppc), data = .)
    m2 <<- lm(votes_share ~ log(gdppc) + party + factor(year), data = .)
}

stargazer(m1, m2, out = "R_Tables/mytable.tex")
```

8. Further topics

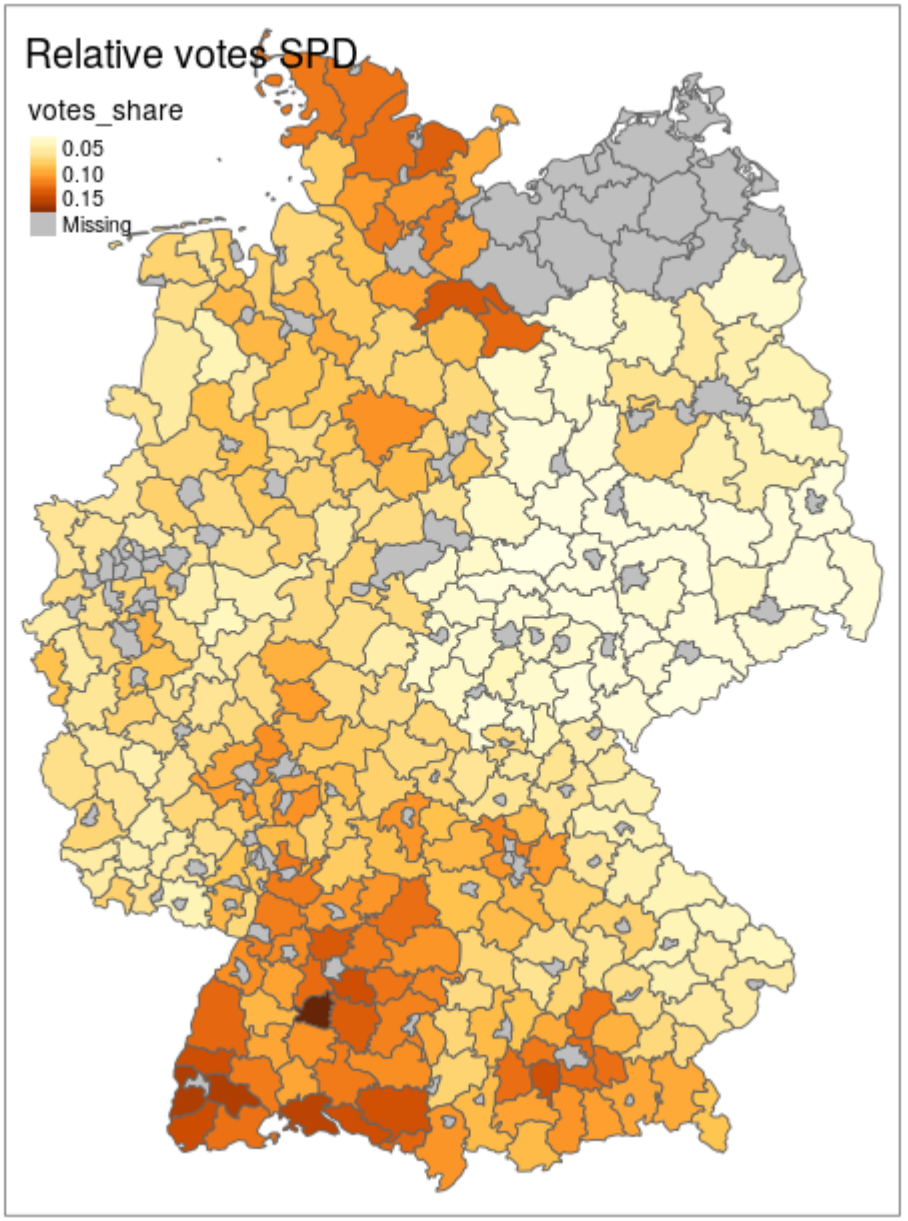
Spatial data

We have already touced a spatial data set in the section Input/Output, where we imported an ESRI shape file.

Of course, it is possible to use these geometries to draw some information into maps. The package `tmap` provides thematic mapping optionally based on the Grammar of Graphics.

Thematic map example

```
library(rgdal)
library(tmap)

kreise <- readOGR("vg2500_krs.shp")

## add information about elections to data frame
kreise@data %<>%
    mutate(ags_kkz = as.numeric(as.character(RS))/1e7) %>%
    left_join({
        county_data %>%
            filter(year == 2017 & party == "GRÜNE")
    })

tm_shape(kreise) +
    tm_borders() +
    tm_fill(col = "votes_share", style = "cont") +
    tm_layout(title = "Relative votes SPD")
```


Relative votes SPD
votes_share
0.05
0.10
0.15
Missing

Where to look further:

There are so many options how to draw spatial information that it is not possile to look here into every detail.

Some references:

- Chapter 8 of the book Geocomputation with R
- tmap: get started vignette
- see also Github page of tmap

Efficiency of R

- R is an interpreted language and, sometimes, it might seem that evaluation is "slow"
- However, most of the numerical computations are implemented in compiled code (C/C++, Fortran), to make use of them, consider to do the following
  - vectorize computations (often, automatic recycling of objects is very helpful here)
  - replace loops by `apply` calls whenever possible
- Some functions offer high flexibility at the cost of execution time, example: `lm()` vs. `lm.fit()` vs. `.lm.fit()` (useful in simulations)
- avoid to grow objects, always initialize them with full size and incrementally fill them
- R has several profiling opportunities to find out, where time is consumed

General remarks

- if projects start to become more complex (several functions), consider to use R package structure. Benefits: functions are easily imported (-> `devtools`), documentation (-> `roxygen2`) and tests (`testthat`) are pretty much standardized
- use version control like git to track how the code basis changed over time
- R markdown is a very easy-to-use markup language which can also include R code to produce smaller reports and convert to `pdf`, `docx`, `html`,...
- It is worthwile to think about a consistent coding style (naming scheme for objects, code formatting,...). Examples: tidyverse style guide or Google's R Style Guide

Thank you!

Slides created via the R package **xaringan**

Some additional material (in german)

Grafiken

- In diesem Kapitel behandeln wir Grafikfunktionen aus dem `graphics`-Paket
- Es gibt auch andere Grafik-Pakete, z.B. `ggplot2` (siehe Dazu Buch von Hadley Wickham), `plotly`, `Rgnuplot`,...

Zum Erzeugen von Plots mit den R Standard `graphics`-Paket existieren sogenannte *high-level*- und *low-level*-Plot-Funktionen.

- *high-level*-Funktionen erzeugen eine eigene Grafik (und öffnen ein *device*)
- *low-level*-Funktionen fügen einer bereits bestehenden Grafik weitere Elemente hinzu

Einige *devices* gibt es? Auswahl:

- `windows()`/`x11()`/`quartz()`: Bildschirmgrafiken (windows, Unix, Mac), (wird default)
- `pdf()`: Adobe PDF (leicht in LaTeX einzubinden)
- `svg()`: Scalable Vector Graphics (Häufig in Webseiten genutzt)
- `png()`, `jpeg()`, `tiff()`, `bmp()`: unterschiedliche Bitmaps-Formate

Grafik erzeugen:

```
pdf("myPlot.pdf")                    ## plot-device öffnen
plot(y = rnorm(100), x = 1:100)      ## Grafik erzeugen, mit high-level funktion plot
dev.off()                            ## plot-device schließen
```

`plot()` ist eine Funktion, die unterschiedliche Methoden für unterschiedliche Objekte aufruft:

```
function (x, y, ...)
UseMethod("plot")
<bytecode: 0x55ef0a3b5ec8>
<environment: namespace:graphics>
```

Diese Methoden sind die unterschiedlichen zur Verfügung stehenden high-level-Plotfunktionen

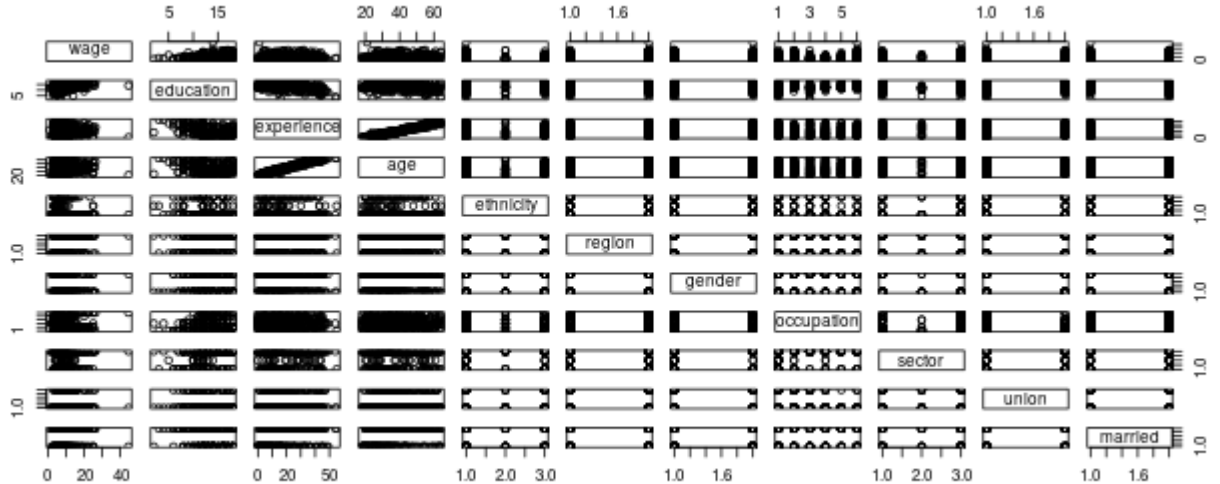
Arten von Plots (High-Level Plots)

- `barplot()`: Säulen- bzw. Balkendiagramm
- `pie()`: Kuchendiagramm (package plotrix mit pie3d)
- `boxplot()`: Boxplot
- `contour()`: Gut geeignet für Höhenkarten, d.h. f: R^2 -> R; filled.contour für farbibe Höhenkarten
- `coplot()`: conditioning plot: Für einen factor mehrfach ausgegebene Plots
- `curve()`: Linienplot, aber einfache Funktionsübergabe durch curve(f(x)= ..., x=, from=, to)
- `dotchart()`: auch mit Scatterplot erzeugbar; besonders sinnvoll für sehr viele Faktoren mit einer Ausprägung
- `hist()`: Histogramm
- `mosaicplot()`: Mosaikplot; sinnvoll für Zusammensetzungen in der Zeit
- `pairs()`: Sinnvoll zur Korrelationsüberprüfung
- `image()`: Zum Zeichnen...
- `persp()`: Wieder zum Zeichen von 3d Graphen, inklusive Koordinatenkreuzwahl
- `scatterplot3d()`: das gleiche
- `qqplot()`: Quantile-Quantile-Plot

```
load(url("https://www.uni-regensburg.de/wirtschaftswissenschaften/vwl-tschernig/medi
class(CPS1985)
```

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

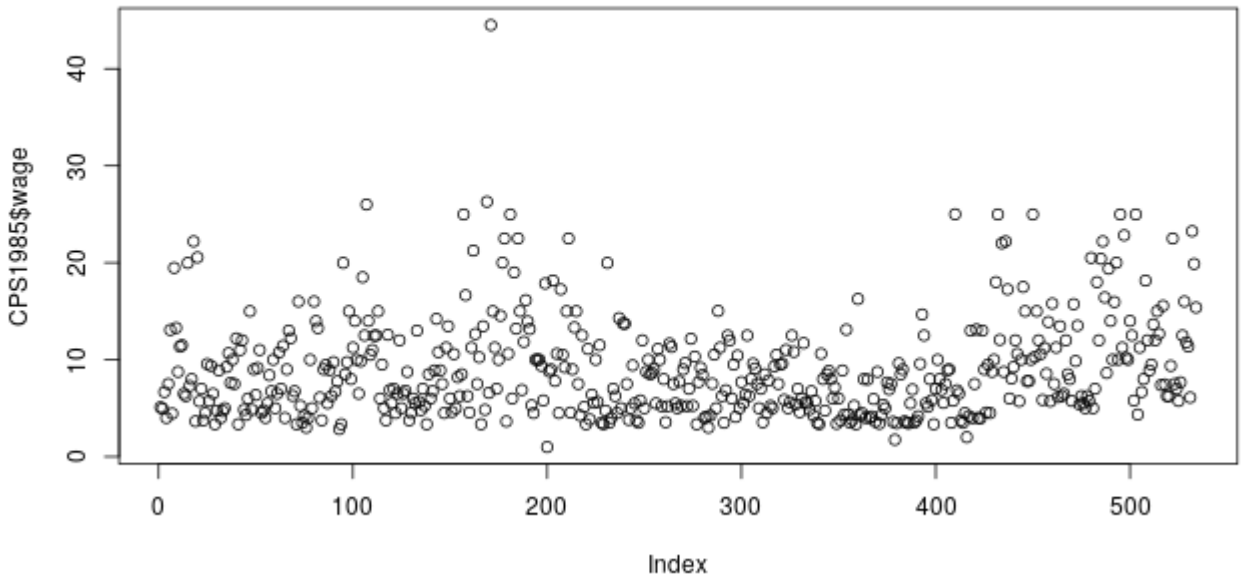
```
plot(CPS1985)              ## plot auf data.frame ruft pairs() auf
```



```
class(CPS1985$wage)
```

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

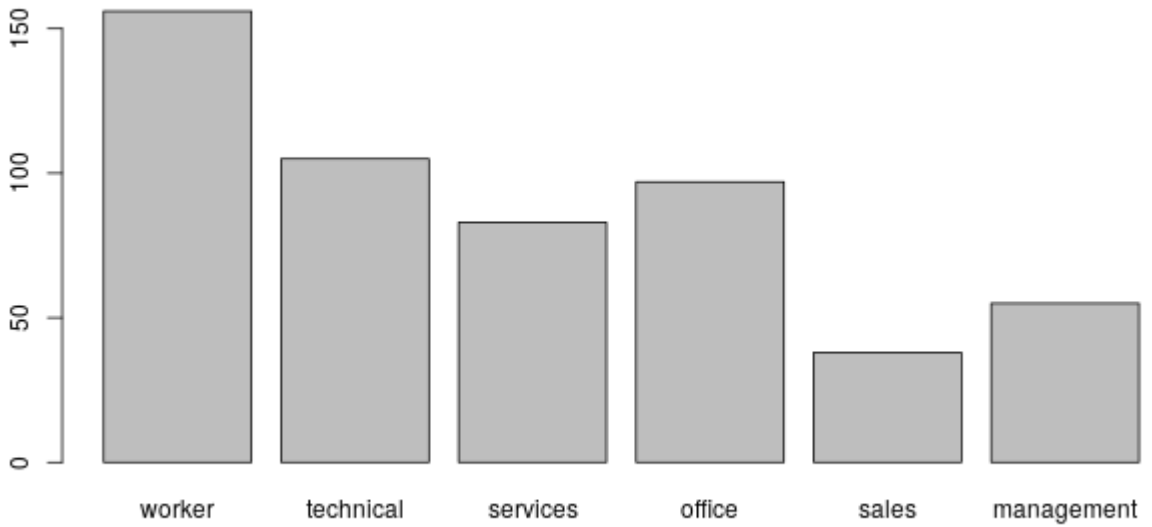
```
plot(CPS1985$wage)       ## plot auf numeric ruft plot.default() auf
```



```
class(CPS1985$occupation)
```

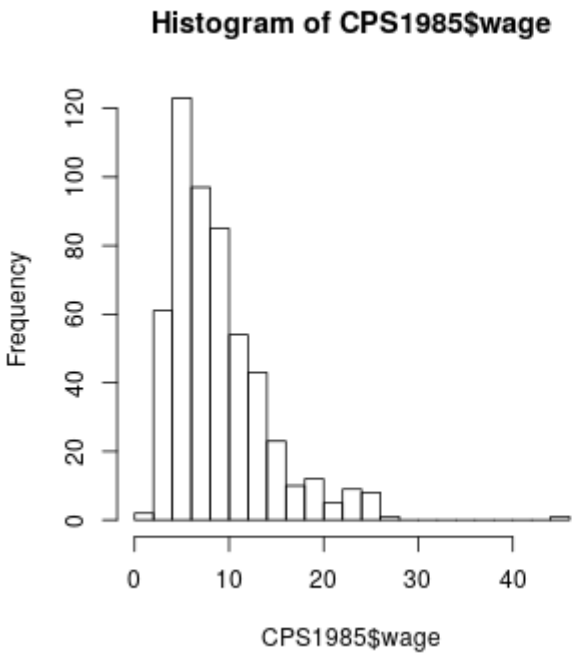
```
## [1] "factor"
```

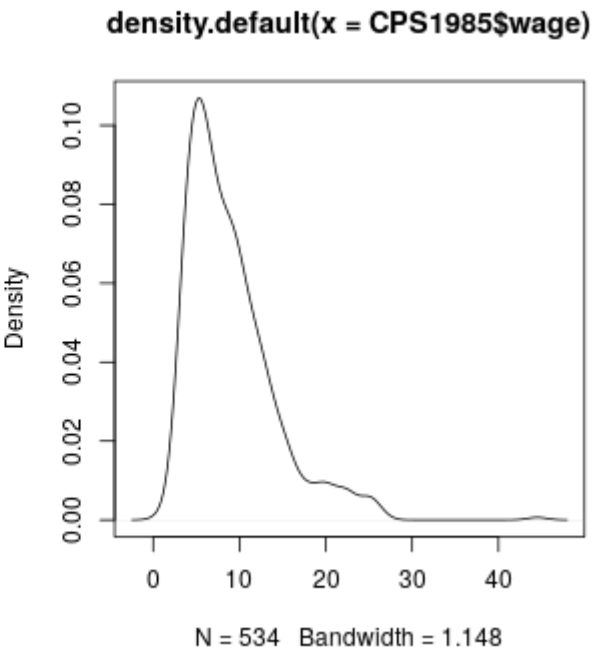
```
plot(CPS1985$occupation)       ## plot auf factor ruft barplot() auf
```



Eine stetige Variable

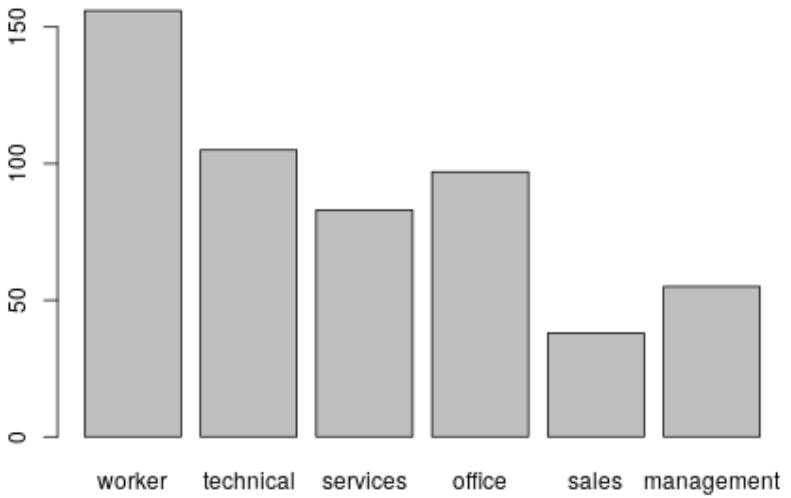
```
hist(CPS1985$wage , breaks = 20)
plot(density(CPS1985$wage))
```

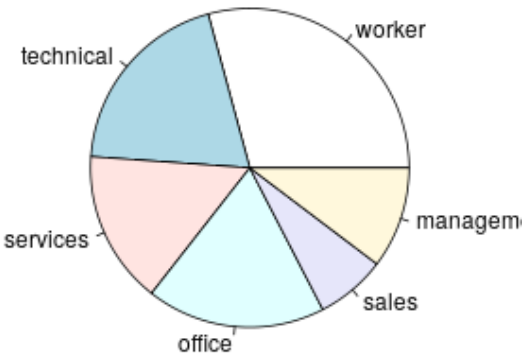




Eine diskrete Variable

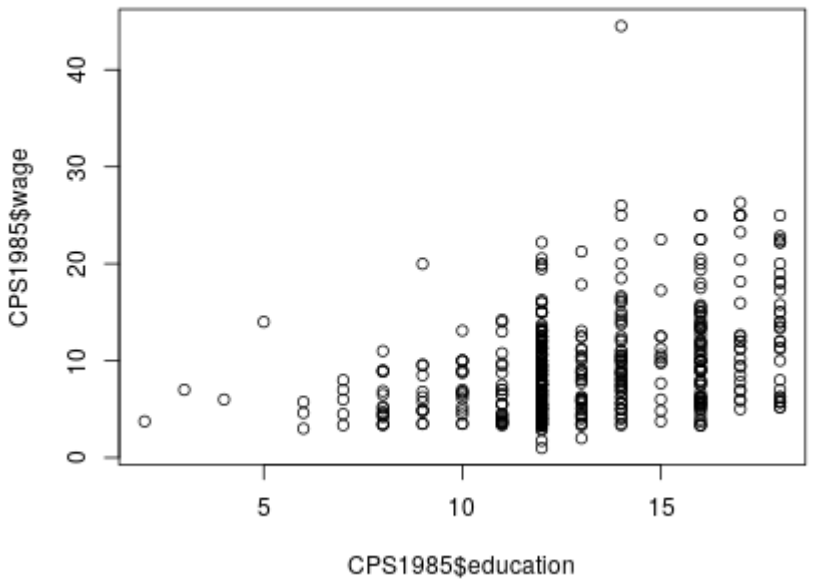
```
plot(CPS1985$occupation)         ## erzeugt barplot(table(CPS1985$occupation))
pie(table(CPS1985$occupation))   ## gleiche Information, aber nicht unbedingt zu empf
```





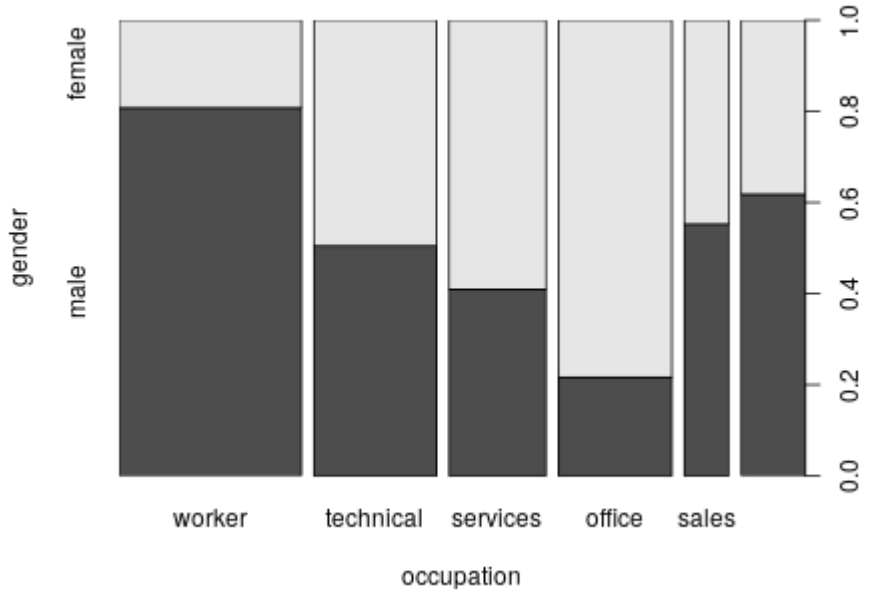
Zwei stetige Variablen

```
## Scatterplot als Ausgangspunkt der meisten Überlegungen:
plot(x = CPS1985$education, y = CPS1985$wage)
```



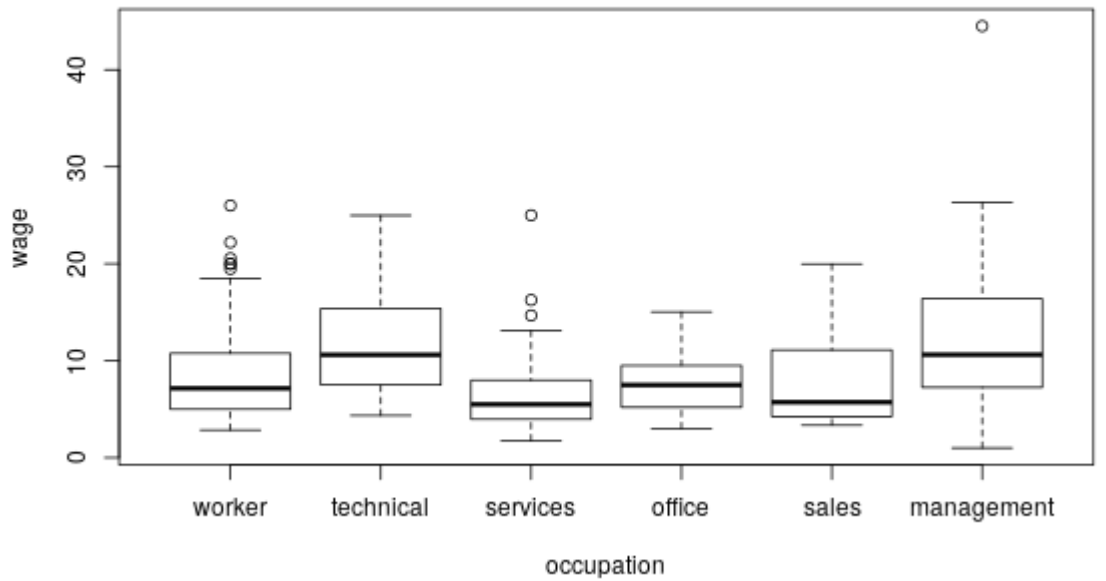
Zwei diskrete Variablen

```
## Scatterplot als Ausgangspunkt der meisten Überlegungen:
plot(gender ~ occupation, data = CPS1985)
```



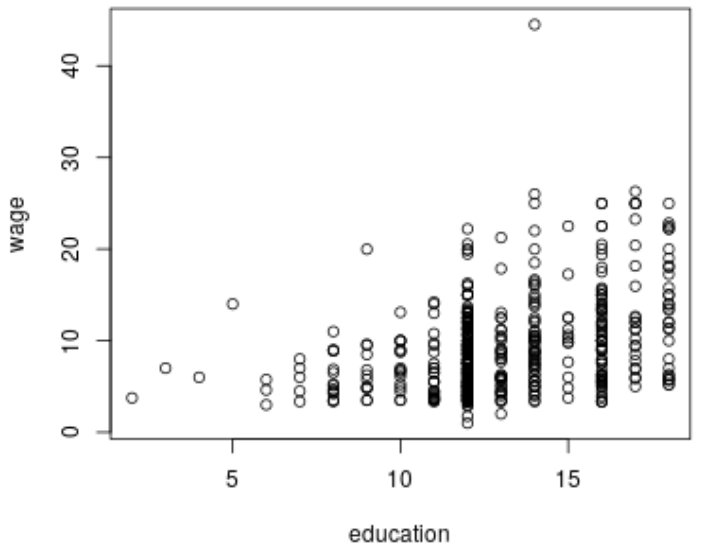
Stetige Variable in Abhängigkeit von einer diskreten Variable

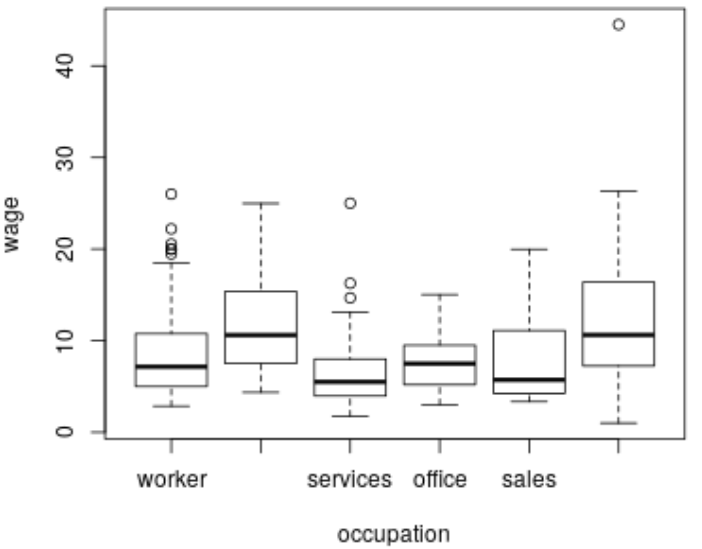
```
plot(wage ~ occupation, data = CPS1985)
```



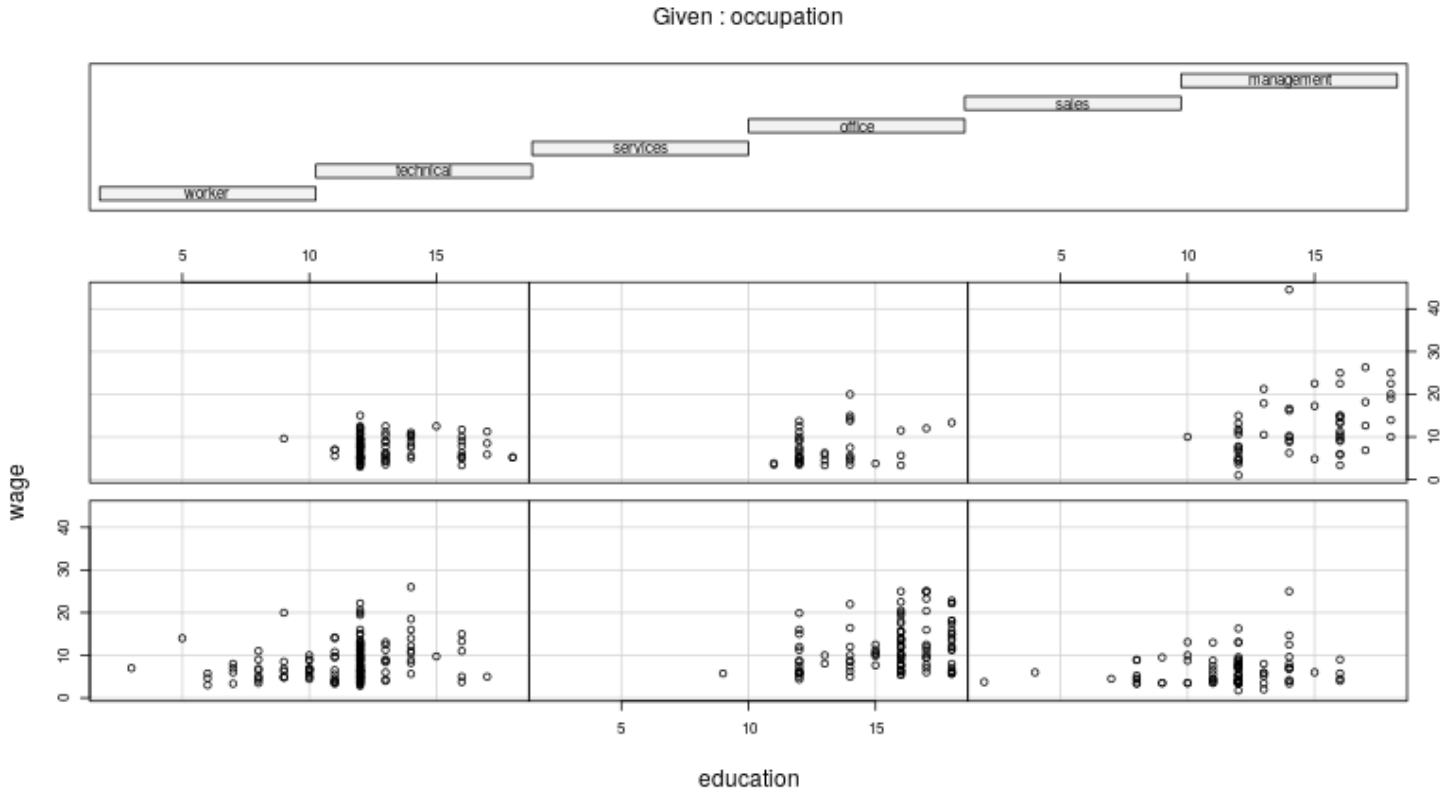
drei oder mehr Variablen

```
## nacheinander
plot(wage ~ education + occupation, data = CPS1985)
```

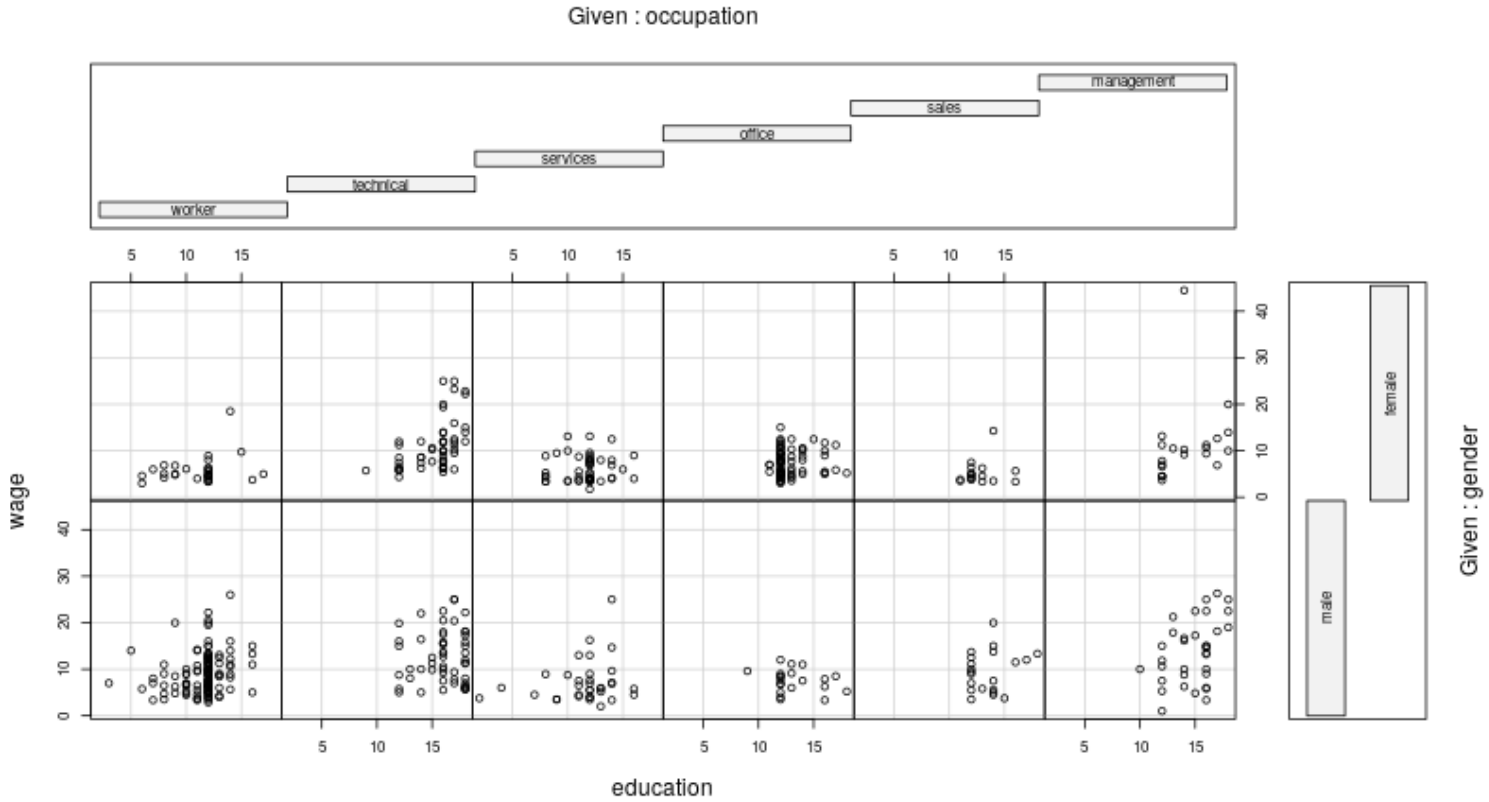




```
## in einem plot
coplot(wage ~ education | occupation, data = CPS1985)
```

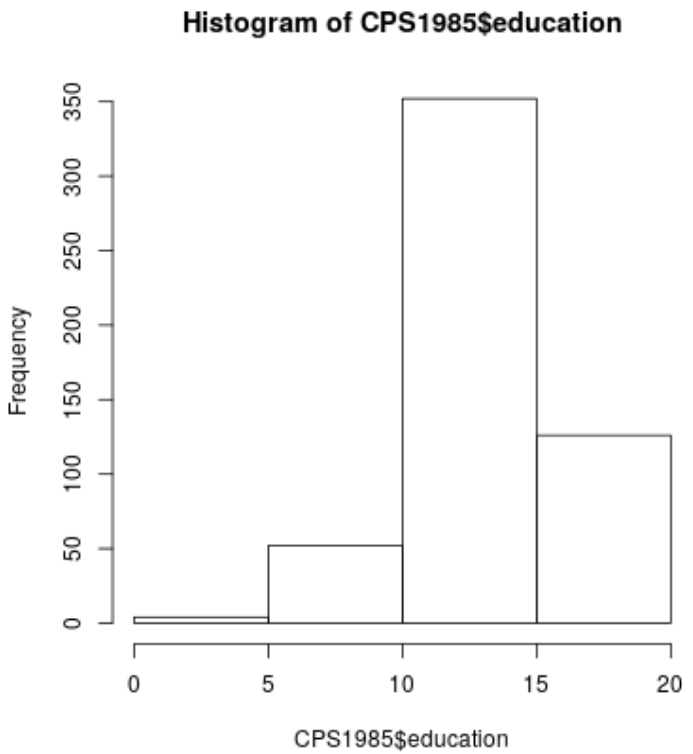


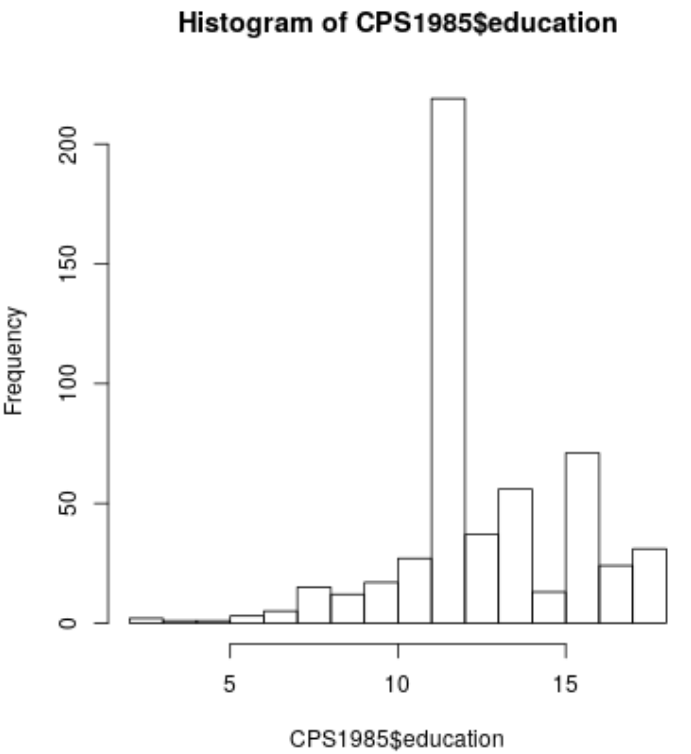
```
## vier variablen in einem plot
coplot(wage ~ education | occupation + gender, data = CPS1985)
```



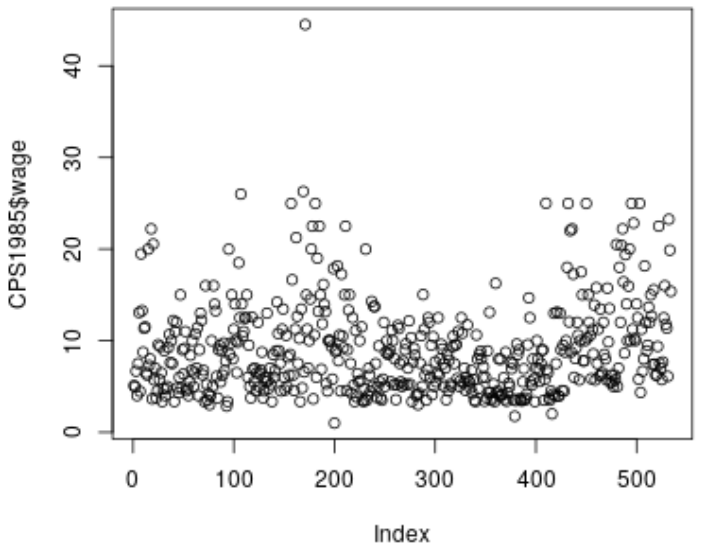
Grafiken anpassen

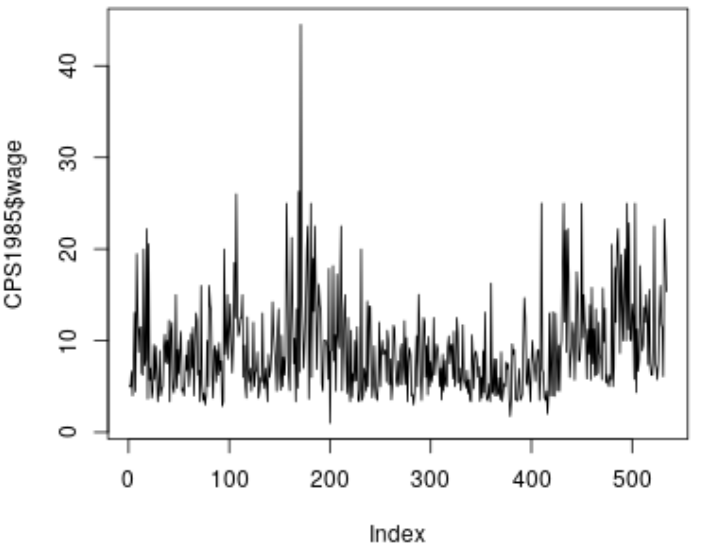
```
hist(CPS1985$education, breaks = 3)  ## wie 'fein' soll das Histogramm sein
hist(CPS1985$education, breaks = 16)
```



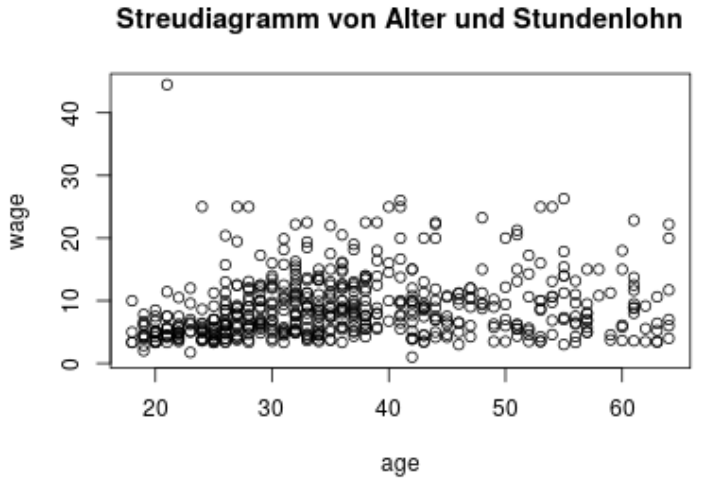


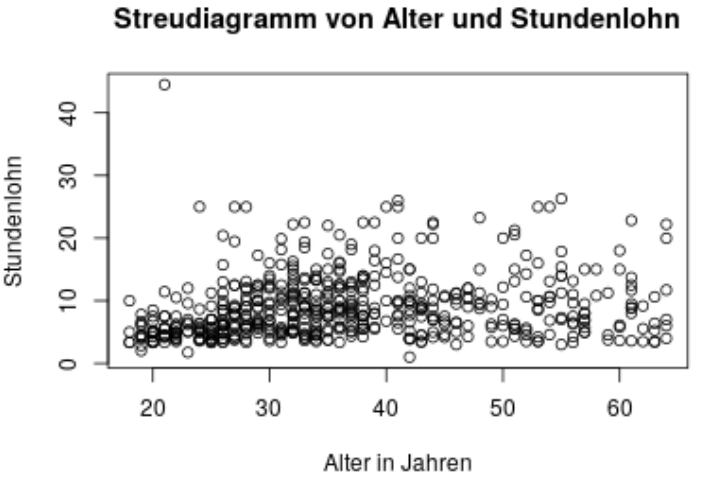
```
plot(CPS1985$wage, type = "p") ## points
plot(CPS1985$wage, type = "l") ## lines
```





```r
# Überschrift und Achsenbeschriftung
with(CPS1985,{        ## siehe Hilfe zu with()
  plot(x = age, y = wage,
     main = "Streudiagramm von Alter und Stundenlohn")
  plot(age, wage,
     xlab = "Alter in Jahren", ylab = "Stundenlohn",
     main = "Streudiagramm von Alter und Stundenlohn")
})
```

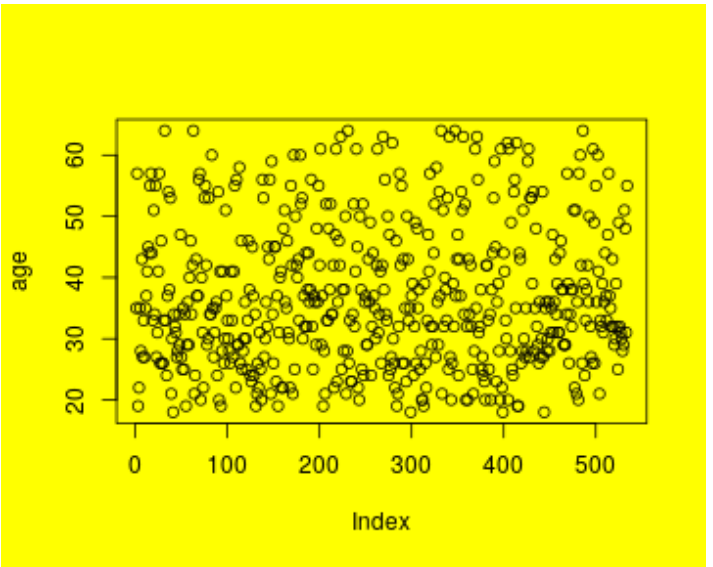




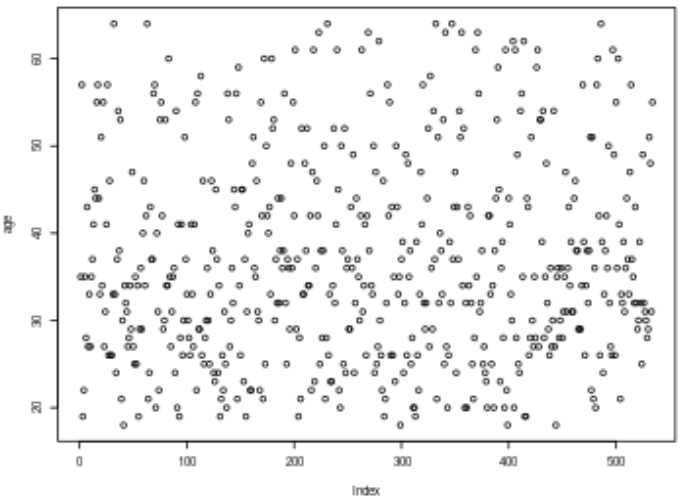
Feintuning

Nicht alles lässt sich über optionale Argumente in high-level-Funktionen anpassen, dann häufig über `par()`:

```
par(bg = "yellow")
with(CPS1985,
     plot(age)
)
```




```
par(bg = "transparent", cex=0.5)
with(CPS1985,
     plot(age)
)
```



Übersicht: Häufig genutzte Argumente von Grafikfunktionen

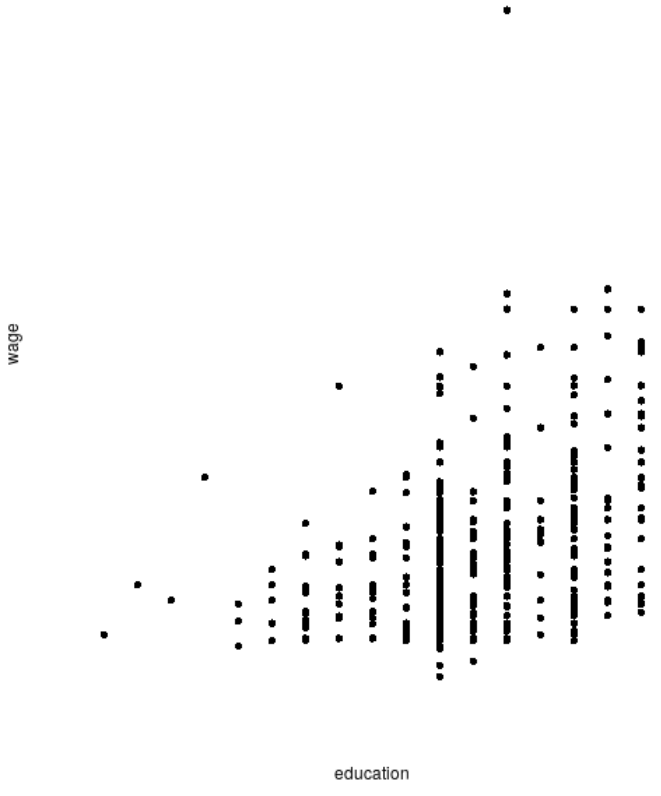
- `axes`: Achsenangabe
- `bg`: Backgroundfarbe
- `cex`: Faktor, der die Vergrößerung zum Standard angeben soll
- `col`: Plotfarbe
- `log`: xlog und ylog für logarithmische Skalen
- `lty`, `lwd`: Linientyp und -dicke
- `mai`: 4-Vektor über die Ränder unten, links, oben, rechts
- `main`, `sub`: Titel, Untertitel
- `mar`: Rand
- `mfcol`, `mfrow`: mehrere Plots in ein Grafikfenster (spaltenweise/zeilenweise)
- `pch`: Pointcharakter (1-16)
- `usr`: Die Extremalstellen für einen plot
- `xlab`, `ylab`: x und y Beschriftung
- `xlim`, `ylim`: x und y Begrenzung

Siehe `?par` für alle Grafikoptionen

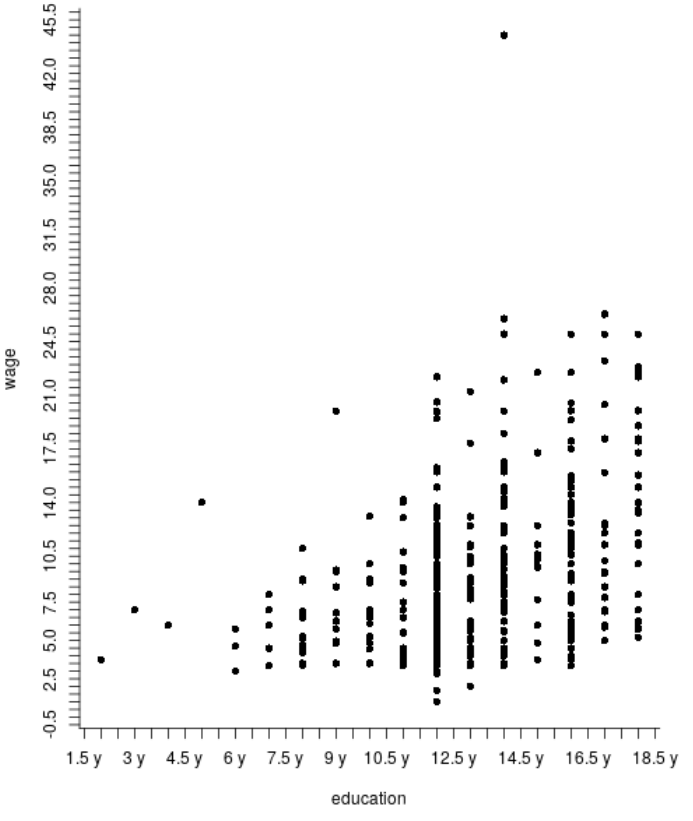
Low-level Grafik-Funktionen

- `lines`: Linien einzeichnen
- `abline`: Schnell für horizontale, vertikale Geraden und Geraden in Geradengleichung $y = bx + a$
- `points`: Punkte
- `arrows`: Pfeile
- `polygon`: Beliebige Vielecke
- `segments`: Unausgemalte Vielecke
- `axis`: Achsen
- `grid`: Gitter
- `rug`: "Dichteteppich"
- `title`: Titel
- `legend`: Legende
- `text`: Text durch (x, y) Koordinaten
- `mtext`: Text durch Positionsangebae wie `side=1,...,4`

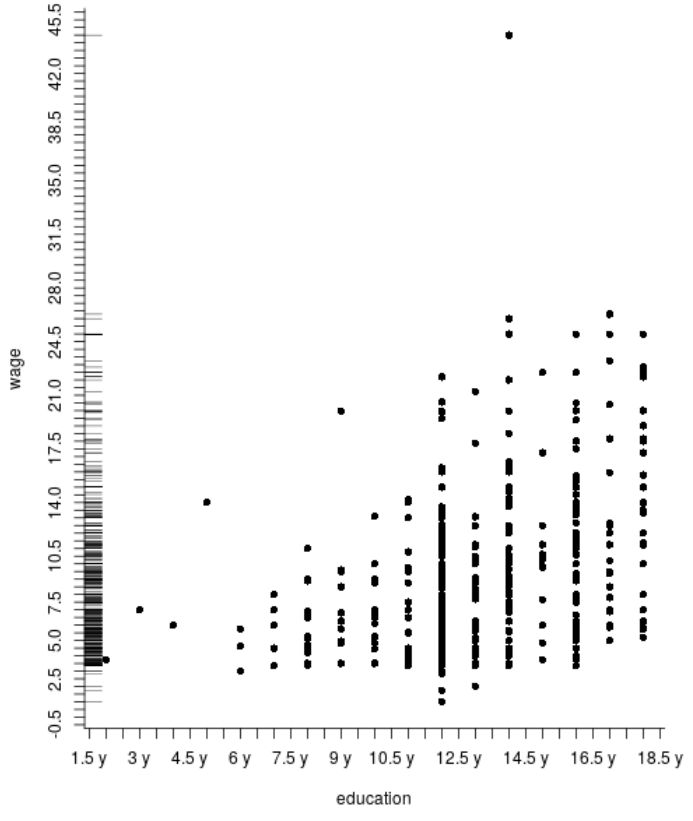
```r
attach(CPS1985) ## siehe ?attach
plot(x = education, y = wage,
     pch = 16, axes = FALSE)
```



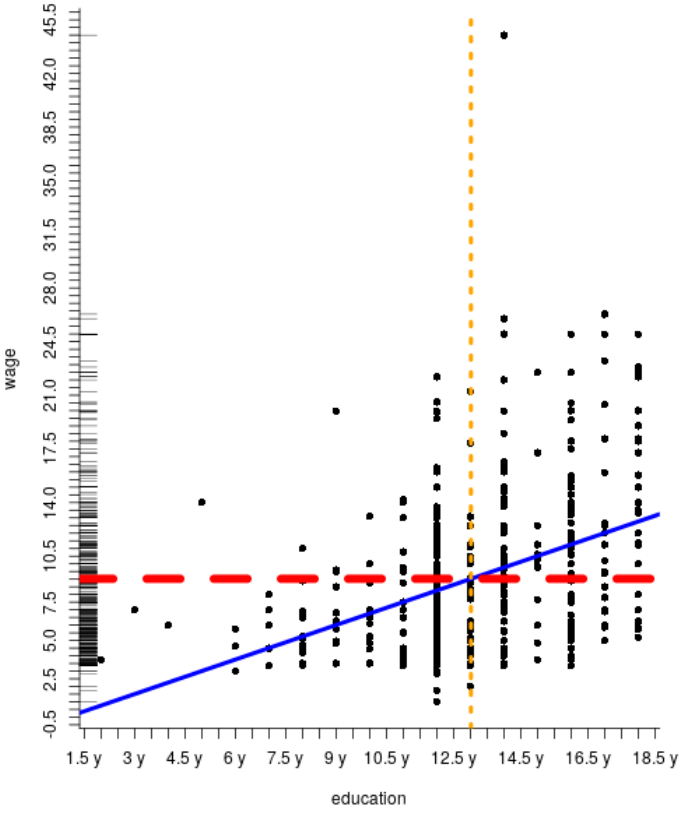
```r
attach(CPS1985) ## siehe ?attach
plot(x = education, y = wage,
     pch = 16, axes = FALSE)


# Achsen hinzufügen
ticks <- seq(-100, 100, 0.5)
axis(side = 1, at = ticks,
     labels = paste(ticks,"y"))

axis(side = 2, at = seq(-100,100, 0.5))
```



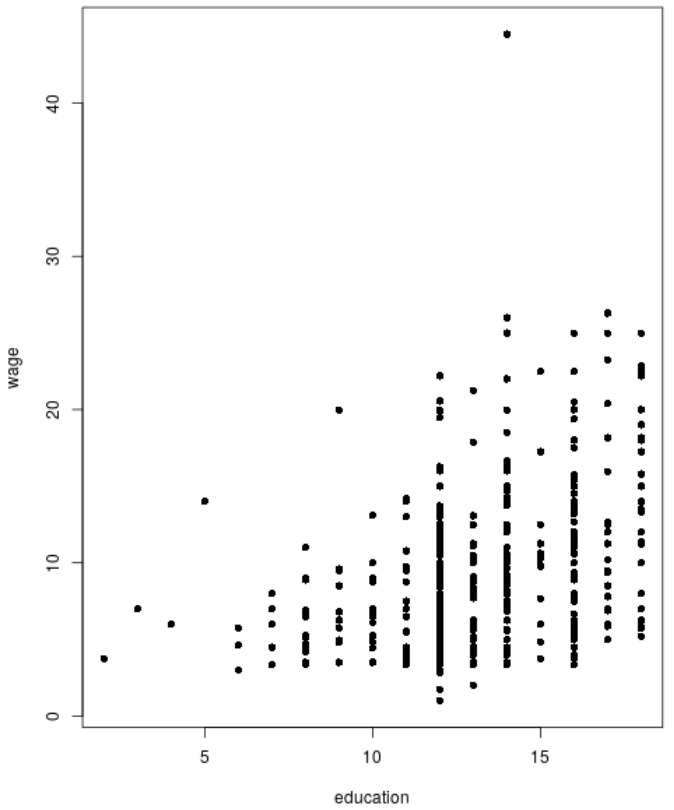
```r
attach(CPS1985) ## siehe ?attach
plot(x = education, y = wage,
     pch = 16, axes = FALSE)


# Achsen hinzufügen
ticks <- seq(-100, 100, 0.5)
axis(side = 1, at = ticks,
     labels = paste(ticks,"y"))

axis(side = 2, at = seq(-100,100, 0.5))

# Striche wo Datenpunkte
rug(wage, side = 2)
```



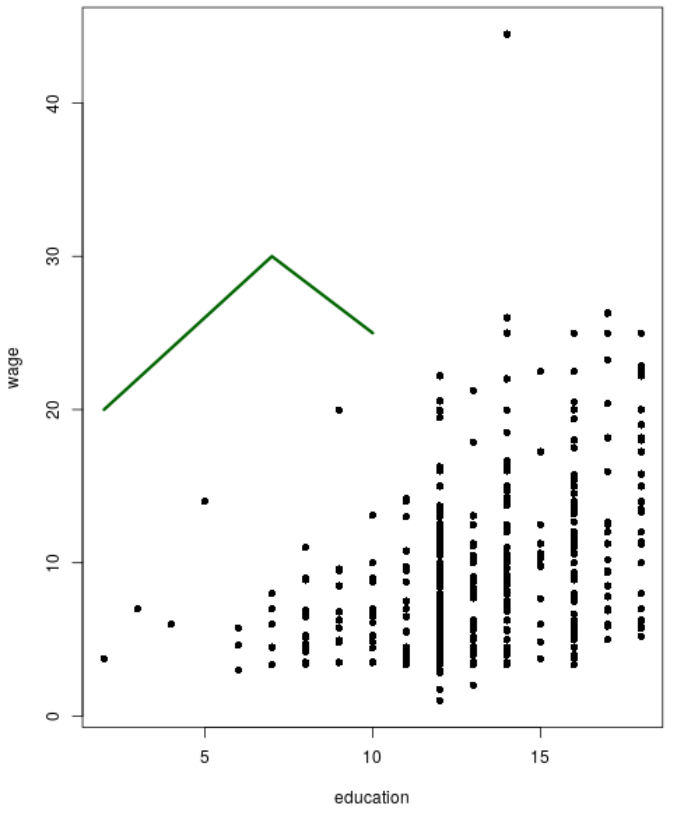
```r
attach(CPS1985) ## siehe ?attach
plot(x = education, y = wage,
     pch = 16, axes = FALSE)


# Achsen hinzufügen
ticks <- seq(-100, 100, 0.5)
axis(side = 1, at = ticks,
     labels = paste(ticks,"y"))

axis(side = 2, at = seq(-100,100, 0.5))

# Striche wo Datenpunkte
rug(wage, side = 2)

# Achsenabschnitt und Steigung angeben
abline(a = -0.7460, b=0.7505, col="blue
abline(h = mean(wage), lty=2,
       col = "red", lwd = 8)
abline(v = mean(education), lty=3,
       col = "orange", lwd = 4)
```



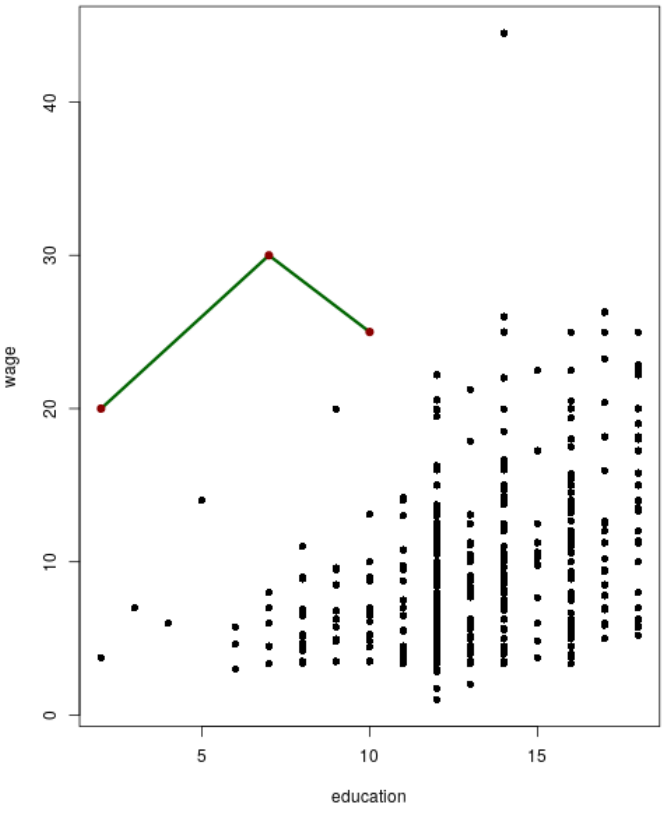
```
plot(x = education, y = wage,
     pch = 16)
```



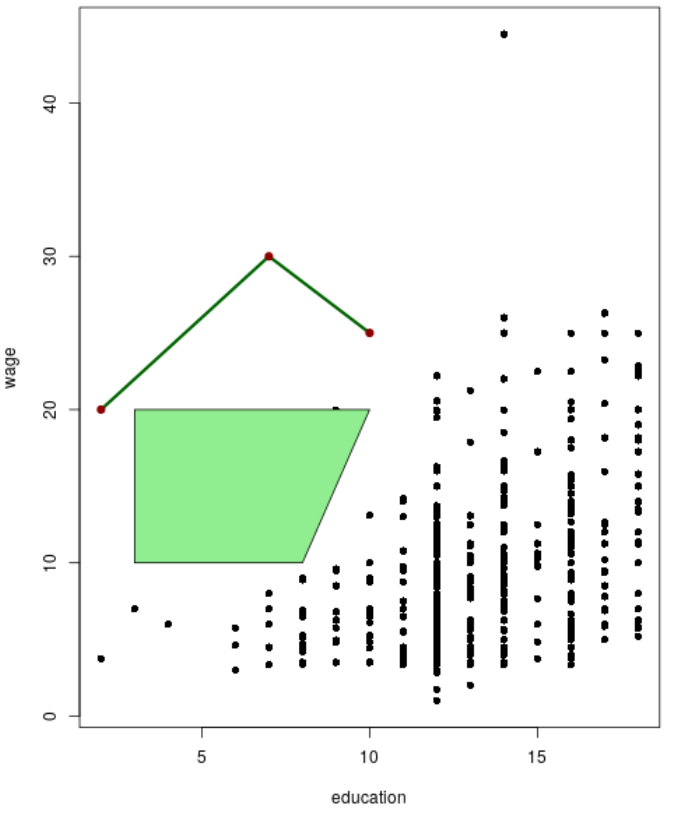
```
plot(x = education, y = wage,
     pch = 16)
lines(x = c(2, 7, 10),
      y = c(20, 30, 25),
      col="darkgreen", lwd = 3)
```



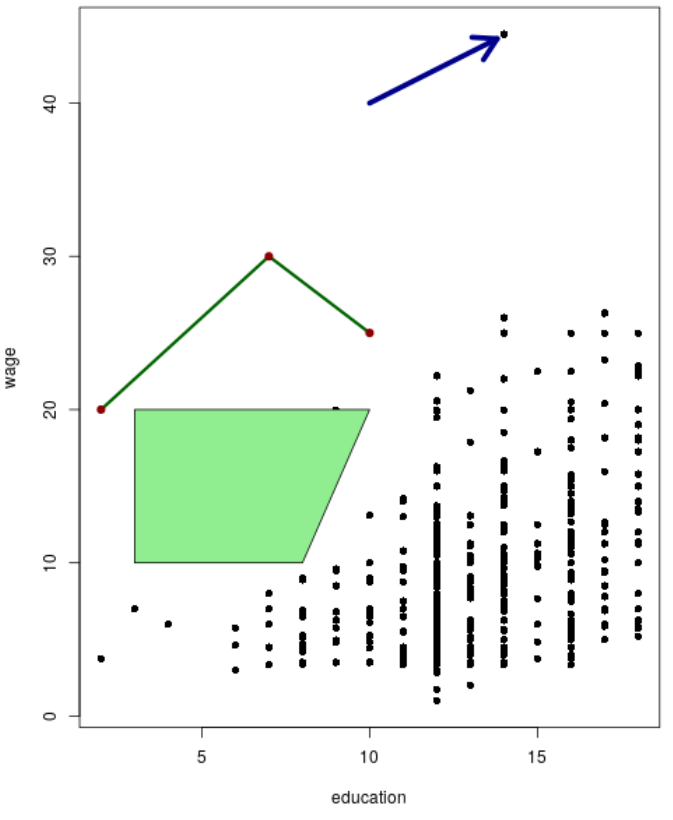
```r
plot(x = education, y = wage,
     pch = 16)
lines(x = c(2, 7, 10),
      y = c(20, 30, 25),
      col="darkgreen", lwd = 3)
points(x = c(2, 7, 10),
      y = c(20, 30, 25),
      col="darkred", pch = 19)
```



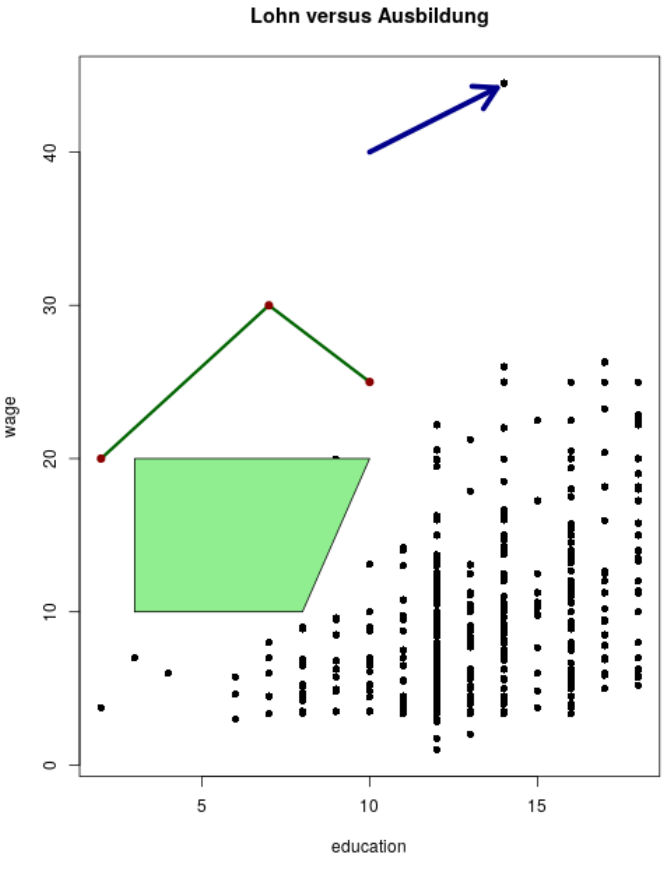
```
plot(x = education, y = wage,
     pch = 16)
lines(x = c(2, 7, 10),
      y = c(20, 30, 25),
      col="darkgreen", lwd = 3)
points(x = c(2, 7, 10),
      y = c(20, 30, 25),
      col="darkred", pch = 19)
polygon(x = c(3, 3, 10, 8),
        y = c(10,20,20,10),
        col="lightgreen")
```



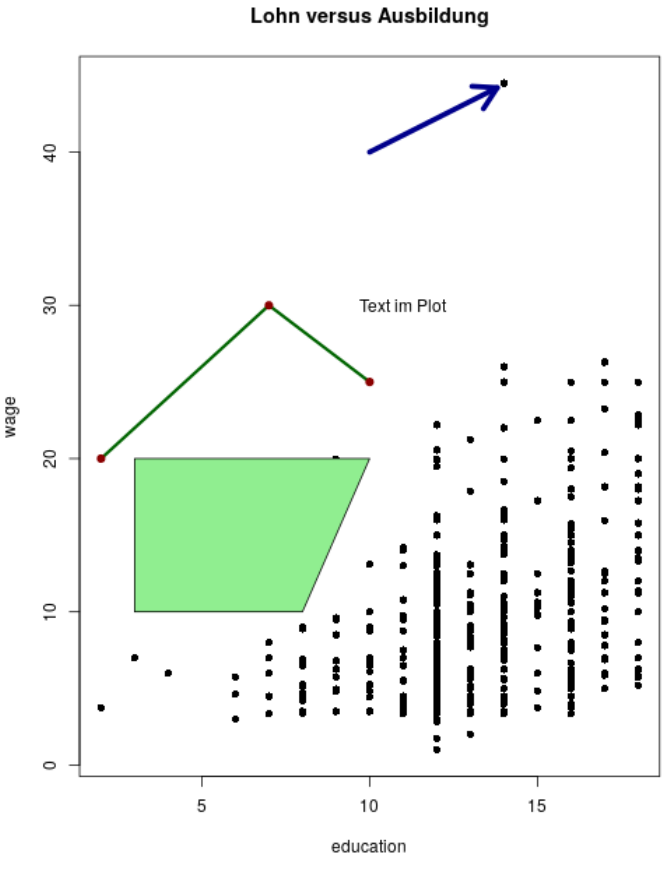
```
plot(x = education, y = wage,
     pch = 16)
lines(x = c(2, 7, 10),
      y = c(20, 30, 25),
      col="darkgreen", lwd = 3)
points(x = c(2, 7, 10),
      y = c(20, 30, 25),
      col="darkred", pch = 19)
polygon(x = c(3, 3, 10, 8),
        y = c(10,20,20,10),
        col="lightgreen")
arrows(x0 = 10, y0 = 40, x1 = 13.8,
       y1 = 44.2, lwd = 5,
       col = "darkblue")
```



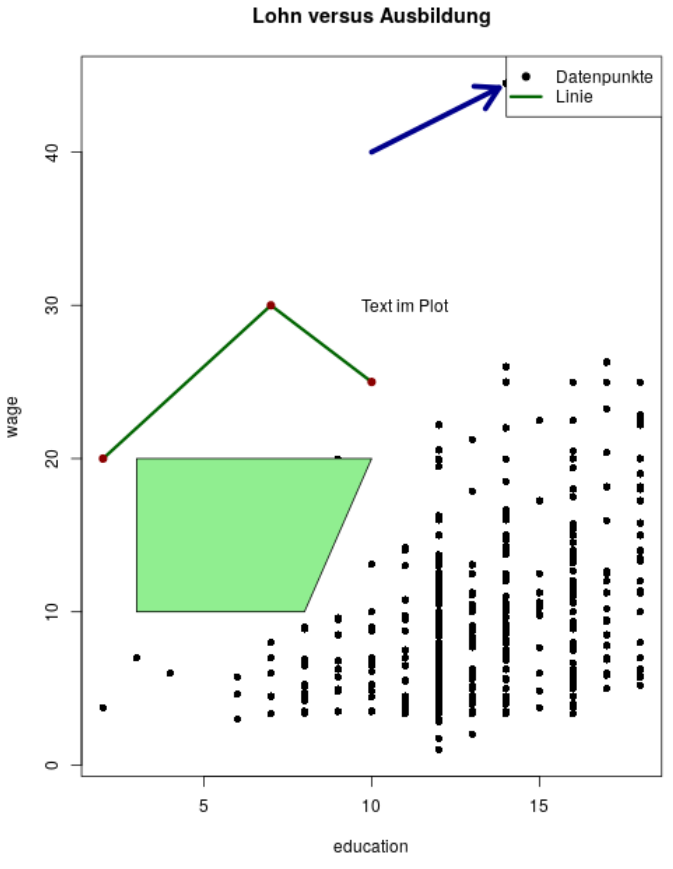
```r
plot(x = education, y = wage,
     pch = 16)
lines(x = c(2, 7, 10),
      y = c(20, 30, 25),
      col="darkgreen", lwd = 3)
points(x = c(2, 7, 10),
      y = c(20, 30, 25),
      col="darkred", pch = 19)
polygon(x = c(3, 3, 10, 8),
        y = c(10,20,20,10),
        col="lightgreen")
arrows(x0 = 10, y0 = 40, x1 = 13.8,
       y1 = 44.2, lwd = 5,
       col = "darkblue")
title("Lohn versus Ausbildung")
```



```r
plot(x = education, y = wage,
     pch = 16)
lines(x = c(2, 7, 10),
      y = c(20, 30, 25),
      col="darkgreen", lwd = 3)
points(x = c(2, 7, 10),
      y = c(20, 30, 25),
      col="darkred", pch = 19)
polygon(x = c(3, 3, 10, 8),
        y = c(10,20,20,10),
        col="lightgreen")
arrows(x0 = 10, y0 = 40, x1 = 13.8,
       y1 = 44.2, lwd = 5,
       col = "darkblue")
title("Lohn versus Ausbildung")
text(x = 11, y = 30, "Text im Plot")
```



```r
plot(x = education, y = wage,
     pch = 16)
lines(x = c(2, 7, 10),
      y = c(20, 30, 25),
      col="darkgreen", lwd = 3)
points(x = c(2, 7, 10),
      y = c(20, 30, 25),
      col="darkred", pch = 19)
polygon(x = c(3, 3, 10, 8),
        y = c(10,20,20,10),
        col="lightgreen")
arrows(x0 = 10, y0 = 40, x1 = 13.8,
       y1 = 44.2, lwd = 5,
       col = "darkblue")
title("Lohn versus Ausbildung")
text(x = 11, y = 30, "Text im Plot")
legend(x="topright",
  legend = c("Datenpunkte","Linie"),
  lty = c(NA, 1), pch = c(19, NA),
  lwd = c(1, 3), col = c(1, "darkgreen"
```



Aufgabe 3.1

Zeichnen Sie einen Plot der Funktion $f(x) = e^x + 2$ im Bereich von -2 bis +2, Linienfarbe grün Erstellen Sie eine Legende oben links mit dem Eintrag e^2 und der Linie.

Hinweis: Um mathematische Beschriftungen in eine Legende zu machen, benötigen Sie die "mathematical annotations", die hier zu finden sind. Mit `paste()` und `expression()` dürfte es danach kein Problem mehr sein.

Aufgabe 3.2

Erzeugen Sie einen Scatterplot von `wage` gegen `education` (aus dem Datensatz `CPS1985`), bei dem Frauen als rote und Männer als blaue Punkte erscheinen. Erzeugen Sie eine Legende, die dies illustriert.

Aufgabe 3.3

Versuchen Sie einen Christbaum zu zeichnen (z. B. mit `polygon()`). Speicheren Sie diesen als `.pdf` im "graphs"-Ordner ab.

Numerische Optimierung

In VWL, Ökonometrie und Statistik ist häufig Maximierung oder Minimierung von Funktionen gefordert. Bei einigen Funktionen (statistische Zielfunktionen wie Likelihood oder Residuenquadratsumme nichtlinearer Regressionen) ist das Optimierungsproblem analytisch nicht lösbar, es existiert keine (algebraische) Formel für den Schätzer (wie etwa bei OLS).

Ausweg: Es ist möglich, die Funktion **numerisch** bzw. allgemeiner **iterativ** (in mehreren Schritten) zu optimieren. Dazu muss i.A. nur der Funktionswert evaluierbar sein.

Achtung: Bei numerischen Algorithmen muss auf Eindeutigkeit der Nullstelle, des Optimums, des Fixpunkts usw geachtet werden. U. a. können unterschiedliche Startwerte zu unterschiedlichen Ergebnissen führen (s. unten)

In R gibt es verschiedene Funktionen zur numerischen Optimierung

- `optimize()`: relativ einfach zu bedienen, kann aber nur univariate Funktionen optimieren
- `nlm()`: nichtlineare Optimierung
- `optim()`: allgemeines Interface zu mehreren Optimierungsalgorithmen

$\Rightarrow$ wir werden hier nur `optim()` behandeln

Zur Minimierung benötigt `optim()` folgende Argumente:

- `par`: Vektor mit Startwerte von den für die zu minimierende Funktion relevanten Parameter
- `fn`: die zu minimierende Funktion

optional kann außerdem spezifiziert werden:

- `method`: welcher Optimierungsalgorithmus soll verwendet werden?
  - `"Nelder-Mead"`: etwas langsamer aber dafür relativ robust, auch auf nicht differenzierbare Funktionen anwendbar
  - `"BFGS"`: benötigt zusätzlich eine erste Ableitung (entweder übergeben mit `gradient` ansonsten numerisch), schneller als `"Nelder-Mead"`
  - `"CG"`: etwas schneller als `"BFGS"` aber etwas stabiler
  - `"L-BFGS-B"`: erlaubt Parametereinschränkungen durch obere und untere Grenzen für die Parameter
  - `"SANN"`: Simulated Annealing, siehe dazu Eintrag in Wikipedia
  - `"Brent"`: Für eindimensionale Optimierung, wird auch in `optimize()` verwendet

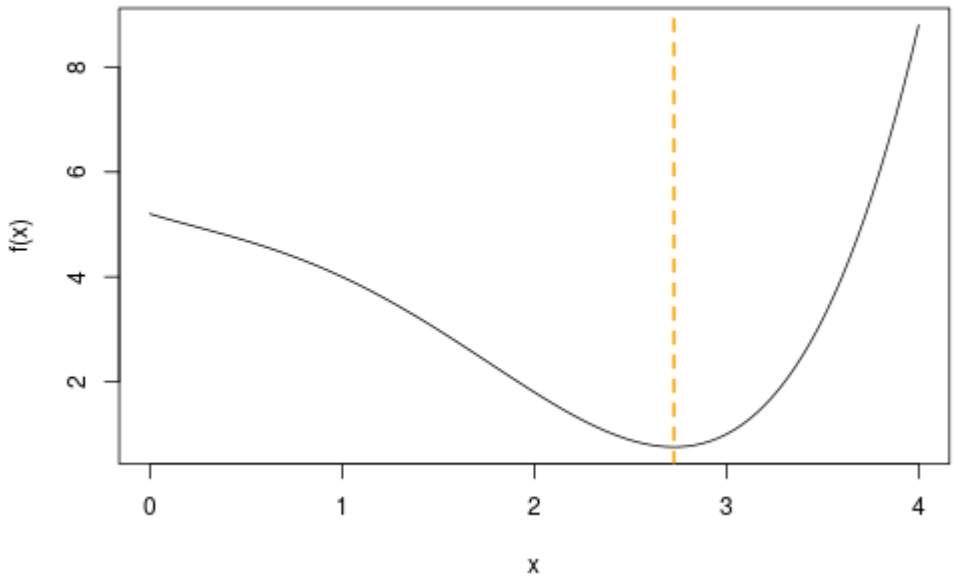
- `lower/upper`: falls `method = "L-BFGS-B"` oder `method = "Brent"`, kann hiermit der Parameterraum eingeschränkt werden
- `control`: weitere Kontrollparameter übergeben in einer Liste (z.B. maximale Anzahl der Iterationsschritte)
- `hessian`: soll die Hessematrix der Funktion im berechneten Optimum mit ausgegeben werden?

Der Rückgabewert der Funktion `optim()` ist eine Liste mit den Einträgen

- `par`: Parameterwerte im Optimum
- `value`: Funktionswert im Optimum
- `counts`: Anzahl der Funktions- und Gradientenaufrufe bis zum Finden des Optimums
- `convergence`: 0 falls Optimierung erfolgreich, ansonsten verschiedene Fehlermeldungen

Einfaches Beispiel

```r
f <- function(x) 0.2*(x-1)^4-0.7*x^2-0.3*x+5
curve(f, from = 0, to = 4)

## für eindimensionale Optimierung wähle "Brent"
opt <- optim(par=NA, fn=f, lower=-10, upper=10, method="Brent")
abline(v = opt$par, col = "orange", lwd = 2, lty = 2)
```



Aufgabe 8.1

Modifizieren Sie die Funktion f, so dass vor der Ausgabe des Funktionswerts mit der print-Funktion der x-Wert und der Funktionswert ausgegeben wird. Führen Sie nun die Optimierung durch. Wie erklären Sie sich das?

Weiteres Beispiel: Nichtlineare Regression (siehe Fortgeschrittene Ök.)

```r
## Regressionsmodell mit zwei Parametern a1 und a2
a <- c( 1 , 0.6 )
n <- 200

x <- rchisq(n,df=3)
u <-  rnorm(n,sd=0.5)

# Nichtlineares Modell y = a x^b+u
y <-  a[1]*x^a[2] + u

# Meist verwendet man stattdessen log(y) = c0 + c1 log(x) + e
DATA <- data.frame(y,x)
plot(DATA$x,DATA$y)


## Residuenquadratsumme zu minimieren
RSS <- function(a,DATA){
  u_tilde <- DATA$y - a[1]*DATA$x^a[2]
  return(sum(u_tilde^2))
}
```

```r
## Zielfunktion anschauen für verschiedene Werte von a
length.grid <- 50
a1_grid <- seq(-1,3,length.out=length.grid)
a2_grid <- seq(0,1.5,length.out=length.grid)
a_grid <- as.matrix(expand.grid(a1=a1_grid,a2=a2_grid))

RSS_grid <- apply(a_grid, 1, RSS, DATA = DATA)

RSS_grid <- matrix(RSS_grid, length.grid, length.grid)

# Nicht viel zu sehen
contour(a1_grid,a2_grid,RSS_grid,xlab="a1",ylab="a2",nlevels=100)
# -> Logarithmieren
contour(a1_grid,a2_grid,log(RSS_grid),xlab="a1",ylab="a2",nlevels=100)
# 3D-Plot
persp(RSS_grid, xlab = "a1", ylab = "a2", phi = 10, theta = -40, ticktype = "detaile

# damit das Minimum später auch Sinn macht:
persp(log(RSS_grid), xlab = "a1", ylab = "a2", phi = 10, theta = -40, ticktype="deta
```

Numerische Minimierung

```
est <- optim(par = c(1,1), fn = RSS, DATA = DATA)

points(x = est$par[1], y = est$par[2] , col = "red" , pch = 20)
```

Fazit:
Am Tripel $(0.99, 0.61, 45.14) = (a_1, a_2, f(a_1, a_2))$ liegt das Minimum der Residuenquadratsumme, da am Punkt (a_1, a_2) die Funktion RSS das Minimum, den geschätzten Wert 45.14 annimmt

Allgemein:
Hat man eine Funktion $f : \mathbb{R}^n \to \mathbb{R}^n$, so hat der Graph Dimension $n + m$. In diesem Fall: $\mathbb{R}^m = \mathbb{R}^2$ und $\mathbb{R}^n = \mathbb{R}$, also ist der Graph dreidimensional.

Beim vorigen Beispiel ging es um die Berechnung eines nichtlinearen Kleinst-Quadrate-Schätzers, hierfür gibt es in R eine eigene Funktion:

```
est_nls <- nls(y ~ a1 * x^a2, start = c(a1 = 1,a2 = 1), data = DATA)
summary(est_nls)
```

Startwerten haben einen Einfluss auf das Optimum:

```
# Ziel: Maximieren der Funktion f
f <- function(x) (2*cos(x[1]/0.5)-0.3*x[1]^2)+cos(x[2]^2)-0.5*(x[2]-2)^2

# max statt min: control-Argument benutzen (oder g=-f minimieren)
optim(par=c(1,1),f,control=list(fnscale=-1))
optim(par=c(-0.5,3),f,control=list(fnscale=-1))   # Andere Startwerte...
```

```
# Funktion betrachten
l.grid <- 50
grid.x <- as.matrix(expand.grid(x1 = seq(-5,5,length.out=l.grid),
                                x2 = seq(-5,5,length.out=l.grid)))
grid.f <-apply(grid.x, 1, f)
matrix.grid.f <- matrix(grid.f ,l.grid)

image(matrix.grid.f,xlab="x1",ylab="x2")
contour(matrix.grid.f,nlevels=100,add=TRUE)

persp(x = seq(-5,5,length.out=l.grid),
      y = seq(-5,5,length.out=l.grid),
      z = matrix.grid.f,
      xlab = "x1", ylab = "x2", theta = 60, ticktype = "detailed")

# Und Startwerte geschickt wählen
(x.start <- grid.x[which.max(grid.f),])
optim(par=x.start,f,control=list(fnscale=-1))
```

Aufgabe 8.2

Maximieren Sie die Funktion f

```
f <- function(x) (2 * cos( x[1]/0.5 ) - 0.3*x[1]^2) + cos(x[2]^2) - 0.5 * (x[2] - 2)
```

$$x_t = h_t \cdot \varepsilon_t$$

$$\varepsilon_t \sim IID(0,1) \text{ (White Noise)}$$

$$\text{Var}(x_t|\mathcal{I}_{t-1}) = h_t^2$$

Bedingte Varianz ist dabei von der Vorperiode abhängig:

$$h_t^2 = \alpha + \gamma_0 h_{t-1}^2 + \cdots + \gamma_q h_{t-q}^2 + \beta_1 \varepsilon_{t-1}^2 + \cdots + \beta_p \varepsilon_{t-p}^2$$

Wir brauchen nun eine Likelihood-Funktion, die von den Parametern `theta` (Vektor, der alle Parameter des Modells zusammenfasst) und von den Daten `x` (univariate Zeitreihe) abhängt.

```r
LogLik.GARCH <- function(theta, x){    ## GARCH(1,1)

  c <- theta[1]    # Konstante in GARCH Gleichung
  a <- theta[2]    # ARCH Parameter
  g <- theta[3]    # GARCH Parameter
  # Unzulässige Parameterwerte ausschließen (-> Inf als Wert)
  if (sum(a) + sum(g) > 1) return(NA) # Instabil -> nicht zugelassen
  if (any(theta < 0)) return(NA)      # Nichtnegativität verletzt -> nicht zugelasse

  n <- length(x)        # Stichprobengröße
  h_sq <- rep(var(x),n) # Vektor der bedingten Varianzen (am Anfang: unbedingte Var
  ll <- numeric(n)      # Vektor der (negativen) Log Likelihoods pro Beobachtung

  # Iterativ Bedingte Varianz und negative Likelihood berechnen
  for (i in 2:n){
    h_sq[i] <- c + a * x[i-1]^2 + g * h_sq[i-1]
    ll[i] <- - 0.5 * (log(2*pi) + log(h_sq[i]) + x[i]^2/h_sq[i])     # negative Like
  }

  # Funktionswert = (negative) Log Likelihood

 # Funktionswert = (negative) Log Likelihood
  return(sum(ll))
}
```

Speichere hierzu zunächst Datei wheat.txt lokal unter **`wheat.txt`** ab.

```
wheat <- read.table("wheat.txt",header=TRUE) %>%
  na.omit()

## Startwerte festlegen
par_start <- c(var(wheat$Change)*0.05 , 0.5 , 0.5)

## Test Funktionsaufruf:
LogLik.GARCH( par_start , x = wheat$Change)

## Optimum berechnen
opt <- optim( par_start , LogLik.GARCH , x = wheat$Change ,
              control = list(fnscale = -1 , maxit = 10000) )
```

Thanks!

Slides created via the R package **xaringan**.

The chakra comes from remark.js, **knitr**, and R Markdown.