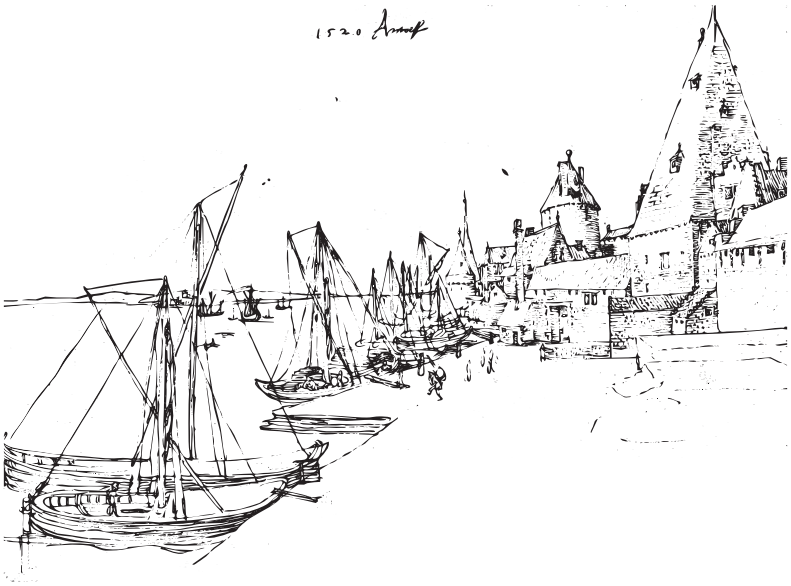
Workflow of statistical data analysis



Oliver Kirchkamp

Workflow of empirical work may seem obvious. It is not. Small initial mistakes can lead to a lot of hard work afterwards. In this course we discuss some techniques that hopefully facilitate the organisation of your empirical work.

This handout provides a summary of the slides from the lecture. It is not supposed to replace a book.

Many examples in the text are based on the statistical software R. I urge you to try these examples on your own computer.

As an attachment of this PDF you find a file `wf.zip` with some raw data. You also find a file `wf.Rdata` with some R functions and some data already in R's internal format.

The drawing on the previous page is Albercht Dürer's "Der Hafen von Antwerpen" — an example for workflow in a medieval city.

Contents

1 Introduction

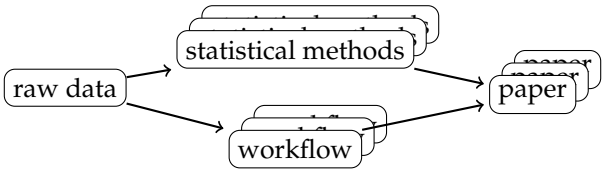
1.1 Motivation

Literature: Surprisingly, there is not much literature about workflow of statistical data analysis:

- J. Scott Long; The Workflow of Data Analysis Using Stata, Stata Press, 2009
- Friedrich Leisch; Sweave User Manual
- Nicola Sartori; Sweave = R · LaTeX2
- Scott Chacon; Pro Git
- Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato; Version Control with Subversion

What is workflow:

- A sequence of operations.
- A pattern of actions that can be documented and learned.



- We spend a lot of time explaining statistical methods to students.
- We do not tell students how to apply these methods (how to integrate methods into a "workflow")
- Why?
- Is "workflow" obvious? — I do not think so.

  Is the wrong workflow not costly? — On the contrary.

  - Mistakes in the statistical method can always be cured.
  - Mistakes in the workflow can render the entire project invalid — no cure possible (e.g. loss of data, loss of understanding the data, loss of methods applied)
- Isn't it sufficient to simply store and backup everything?

  - unfortunately not — statistical analysis tends to create a lot of data.→ storing everythings means hiding everything very well from ourselves and from others.

1.2 Structure of a paper

- Describe the research question

  Which model do we use to structure this question?

- Describe the sample

  How many observations, means, distributions of main variables, key statistics

  Is there enough variance in the independent variables to test what we want to test?

- Test the model

  possibly different variants of the model (increasing complexity)

- Discuss the model, robustness checks

1.3 Aims of statistical data analysis

- Limit work and time
- Get interesting results
- Replicability
  - for us, to understand our data and our methods after we get back to work after a short break
  - for our friends (coauthors), so that they can understand what we are doing
  - for our enemies — we should always (even years after) be able to prove our results exactly
- If statistical analysis was a straightforward procedure, then there would be no problem:

  - Store the raw data. All methods we applied are obvious and trivial.

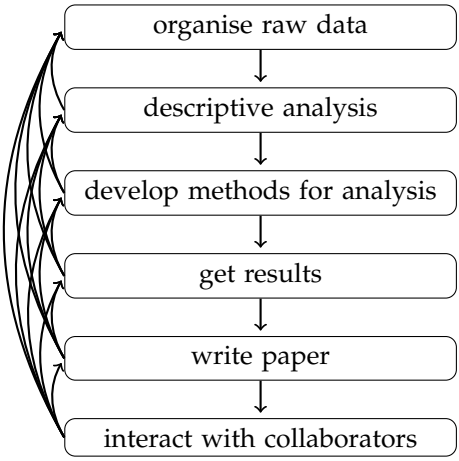
- In the real world our methods are far from obvious:
  - We think quite a lot about details of our statistical analysis
- Assume we have another look at our paper (and our analysis) after a break of 6 month:
  - What does it mean if `sex==1` ?
  - For the variable `meanContribution`: was the mean taken with respect to all players and the same period, or with respect to the same player and all periods, or ...
  - What is the difference between `payoff` and `payoff2`...
  - Do the tables and figures in version 27 of the paper ...
    * ...refer to all periods of the experiment or only to the last 6 periods?
    * ...do they include data from the two pilot experiments we ran?
    * ...do they refer to the "cleaned" dataset, or to the "cleaned dataset in long form" (where we eliminated a few outliers)
    * Do all tables and figures and p-values and t-tests... actually refer to the same data? (or do some include outliers, some not,...)

Assume we take only 10 not completely obvious decisions between two alternatives during our analysis (which perhaps took us 1 week), $\rightarrow$ we will have to explore $2^{10} = 1024$ variants of our analysis (= 1024 weeks) to recover what we actually did.

Often we take more than 10 not completely obvious decisions.

$\rightarrow$ we should follow a workflow that facilitates replicability.

This is not obvious, since workflow is (unfortunately) not linear:



During this process we create a lot of intermediate results. How can we organise these results?

Solutions and restrictions:

- Store everything — not feasible
- We want to be creative, take shortcuts, we want to explore things, play with different representations of a solution…
- During this phase we can not document everything.

1.4 Creativity and chaos

Living two lives:

- creative (undocumented)
- permanent (documented)

Let our computer(s) reflect these two lives:

```
.../projectXYZ/
              /permanent/
                       /rawData
                       /cleanData
                       /R
                       /Paper
                       /Slides
              /creative/
                       /cleanData
                       /R
                       /Paper
                       /Slides
```

You might need more directories for your work.

(In terms of version control, which we will cover later, "permament" could be a *trunk*, while "creative" could be a *branch*)

Rules

1. Anything that we give to other people (collaborators, journals,...) must come entirely from permanent
2. Never delete anything from permanent
3. Never change anything in permanent
4. We must be able trace back everything in permanent clearly to our raw data.

Since we give things to other people more than once (first draft, second draft,..., first revision, ..., second revision,...), we must be able to replicate each of these instances.

Consequences — permanent data has versions (Below we will discuss the advantages of a version control system (git, svn). Let us assume for a moment that we have to do everything manually.)

- We will accumulate versions in our permanent life (do not delete them, do not change them)

```
cleaned_data_110721.Rdata
cleaned_data_110722.Rdata
cleaned_data_110722b.Rdata
⋮
preparingData_110721.R
preparingData_110722.R
descriptives_110722.R
econometrics_110723.R
⋮
paper_110724.Rnw
paper_110725.Rnw
paper_110727.Rnw
⋮
```

What it the optimal workflow? The optimal workflow is different for each of us

Aims

- Exactness (allow clear replication)
- Efficiency
- We must like it (otherwise we don't do it)
- Whatever we do, we should do it in a systematic way
  - Follow a routine in our work (all projects should follow similar conventions)
  - Let the computer follow a routine (a mistake made in a routine will show up "routinely", a hand coded mistake is harder to detect).

    Use functions, try to make them as general as possbible.

  - Prepare for the unexpected! We should not assume that our data will always look the way it looks at the moment.

More on routines Example:

- Probability to make a mistake: 0.1
- Probability to discover (and fix) a mistake: 0.8

Now you solve two related problems, A and B:

- Both problems are solved independently:
  - Probability of (undiscovered) mistake A: $0.1 \cdot 0.2$
  - Probability of (undiscovered) mistake B: $0.1 \cdot 0.2$
  - Probability of some undiscovered mistake: $1 - .98^2 \approx 0.04$
- Both problems are solved with the same routine (one function in your code):
  - Probability of some undiscovered mistake: $0.1 \cdot 0.2^2 = 0.004$

Producing your results with the help of identical (and computerised) routines makes it much easier to discover mistakes.

1.5 Making the analysis reproducible

Here are again the steps in writing a paper:

1. organise raw data
2. descriptive analysis (figures, descriptive tables…)
3. develop methods for analysis
4. get results (run program code)

5. write paper (mix results with text and explanations)
6. interact with collaborators

- All these tasks require decisions.
- All these decisions should be documented.
- When is our documentation sufficient? — If a third person, without our help, can find out what we were doing in all the above steps. If we want to have another look at our data in one year's time we will be in the same position as an outsider today.
- We keep a log where we document the above steps for a given project on a daily basis (research log) (nobody wants to keep logs, so this must be easy)

1.6 Preserve raw data

- If our raw data comes from z-Tree experiments: We better keep all programs (the current version can always be found as `@1.ztt`,... in the working directory).
- If our raw data includes data from a questionnaire:
  - We need a codebook
    - variable name — question number — text of the questions
    - branching in the questionnaire
    - levels (value labels) used for factors
    - missing data, how was it coded?
    - cleaned data, how was it cleaned? (if we have no access to the raw data)

1.7 Interaction with coauthors

- Clear division of labour
  - the "experimenter" decides how the experiment is actually run
  - the "empiricist" decides what statistics and graphs are produced
  - the "writer" decides how to present the text
  - help, do not interfere
- In your communication: concentrate on the essentials:
  - exchange one file
  - make only essential changes to this file
  - clearly explain why these changes are necessary

2 Digression: R

For the purpose of the course we take R as an example for one statistical language. Even if you use other languages for your work, you will find that the concepts are similar.

If you want to know how R's popularity compares with related software, you can read Robert A. Muenchen's article on The Popularity of Data Analysis Software.

2.1 Installation of R

On the Homepage of the R Projekt you find in the menu on the left a link Download / CRAN. This link leads to a choice of "mirrors". If you are in Jena, the GWDG Mirror in Göttingen might be fast. There you also find instructions how to install R on your OS.

Installation of Libraries If the command *library* complains about not being able to find the required library, then the library is most likely not installed. The command

```
install.packages("Ecdat")
```

installs the library *Ecdat*. Some installations have a menu "Packages" that allows you to install missing libraries. Users of operating systems of Microsoft find support at the FAQ for Packages.

2.2 Types and assignments

R knows about different types of data. We will meet some types in this chapter. To assign a number (or a value, or any object) to a variable, we use the operator <-

```
x <- 4
```

R stores the result of this assignment as *double*

```
typeof(x)

[1] "double"
```

Now we can use *x* in our calculations:

```
2 * x

[1] 8

sqrt(x)

[1] 2
```

Often our calculations will not only involve a single number (a scalar) but several which are connected as a vector. Several numbers are connected with *c*

```
x <- c(21,22,23,24,25,16,17,18,19,20)
x

 [1] 21 22 23 24 25 16 17 18 19 20
```

When we need a long list of subsequent numbers, we use the operator `:`

```
21:30

 [1] 21 22 23 24 25 26 27 28 29 30

y <- 21:30
```

Subsets We can access single elements of a variable with `[]`

```
x[1]

[1] 21
```

When we want to access several elements at the same time, we simply use several indices (which are connected with `c`). We can use this to change the sequence of values (e.g. to sort).

```
x[c(3,2,1)]

[1] 23 22 21

x[3:1]

[1] 23 22 21
```

```
x

 [1] 21 22 23 24 25 16 17 18 19 20
```

(to sort a long vector we would use the function `order`).

```
order(x)

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

```
x[order(x)]

 [1] 16 17 18 19 20 21 22 23 24 25
```

Negative indices drop elements:

```
x[-1:-3]

[1] 24 25 16 17 18 19 20
```

Logicals Logicals can be either *TRUE* or *FALSE*. When we compare a vector with a number, then all the elements will be compared (this results from the recycling rule, see below):

```
x < 20

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

We can use logicals as indices, too:

```
x [ x < 20 ]

[1] 16 17 18 19
```

Characters Not only numbers, also character strings can be assigned to a variable:

```
x <- "Mary"
```

We can also work with vectors of character strings:

```
x <- c("John","Mary","Jane")
x[2]

[1] "Mary"

x[3]<-"Lucy"
x

[1] "John" "Mary" "Lucy"
```

Factors Often it is clumsy to store a string of characters again and again if this string appears in the dataset several times. We might, e.g., want to store whether an observation belongs to a man or a woman. This can be done in an efficient way by storing 2 for "male", and 1 for "female".

```
x <- as.factor(c("male","female","female","male"))
levels(x)

[1] "female" "male"

x[2]

[1] female
Levels: female male

as.numeric(x)

[1] 2 1 1 2
```

Usually the first level in a factor is the level that comes first on the alphabet. If we do not want this, we can *relevel* a factor:

```
x<-relevel(x,"male")
x

[1] male   female female male
Levels: male female

as.numeric(x)

[1] 1 2 2 1
```

Note that the *meaning* of the values remains unchanged.

Sometimes, when we have more than only two levels, we want to order levels of a factor along a third variable. This is done by *reorder*.

```
y <- c(12,7,8,11)
reorder(x,y)
```

```
[1] male   female female male
attr(,"scores")
  male female
  11.5    7.5
Levels: female male
```

2.3 Functions

R knows many built-in functions:

```
mean(x)
median(x)
max(x)
min(x)
length(x)
unique(c(1,2,3,4,1,1,1))
```

When we need more, we can write our own:

```
square <- function(x) {
  x*x
}
```

The last expression in a function (here `x*x`) is the return value. Now we can use the function.

```
square(7)

[1] 49
```

When we want to apply a function to many numbers, *sapply* helps:

```
range <- 1:10
sapply(range,square)

 [1]   1   4   9  16  25  36  49  64  81 100
```

With *sapply* we do not have to define a name for a function:

```
sapply(range,function(x) x*x)

 [1]   1   4   9  16  25  36  49  64  81 100
```

2.4 Random numbers

Random numbers can be generated for rather different distributions. R calculates pseudo-random numbers, i.e. R picks numbers from a very long list that appears random. Where we start in this long list is determined by *set.seed*:

```
set.seed(123)
```

10 pseudo-random numbers from a normal distribution can be obtained with

```
rnorm(10)

 [1] -0.56048 -0.23018  1.55871  0.07051  0.12929  1.71506  0.46092 -1.26506
 [9] -0.68685 -0.44566
```

We get the same list when we initialise the list with the same starting value:

```
set.seed(123)
rnorm(10)

 [1] -0.56048 -0.23018  1.55871  0.07051  0.12929  1.71506  0.46092 -1.26506
 [9] -0.68685 -0.44566
```

This is very useful, when we want to replicate the same "random" results.

10 uniformly distributed random numbers from the interval $[100, 200]$ can be obtained with

```
runif(10,min=100,max=200)

 [1] 189.0 169.3 164.1 199.4 165.6 170.9 154.4 159.4 128.9 114.7
```

Often we use random numbers when we simulate (stochastic) processes. To replicate a process we use the command *replicate*. E.g.

```
replicate(10,mean(rnorm(100)))

 [1]  0.016749 -0.024756  0.061321 -0.028206  0.087712 -0.025113 -0.141044
 [8]  0.123990  0.109293 -0.002743
```

takes 10 times the mean of each 100 pseudo-normally distributed random numbers.

2.5 Example Datasets

We just saw that the command *c* allows us to describe the elements of a vector. For long datasets this is not very convenient. R contains already a lot of example datasets. These datasets are, similar to statistical functions, organised in libraries. To save space and time R does not load all libraries initially. The command *library* allows us to load a library with a dataset at any time.

The library *Ecdat* provides a lot of interesting economic datasets. The library *memisc* gives access to some interesting functions that help us organising our data.

When we need a specific function and we do not know in which library to look for this function we can use the command *RSiteSearch* or the R Site Search Extension for Firefox.

The dataset *BudgetFood* is, e.g., contained in the library *Ecdat*.

```
data(BudgetFood,package="Ecdat")
```

To really *see* the numbers, we can use the command *fix*:

```
fix(BudgetFood)
```

Usually we do not want to see many numbers. Instead we want to derive (in a structured way) a few numbers (parameters, confidence intervals, p-values,...)

The command *help* aids us in finding out the meaning of the numbers of the diferent columns of a dataset.

```
help(BudgetFood)
```

An important command to get a summary is *summary*

```
summary(BudgetFood)
```

How can we access specific columns from our dataset? Since R may have several datasets at the same time in its memory, there are several possibilities. One possibility is to append the name of the dataset *BudgetFood* with a *$* and then the name of the column.

```
BudgetFood$age

 [1] 43 40 28 60 37 35 40 68 43 51 43 48 51 58 61 53 58 64 50 50 47 76 49 44 49
[26] 51 56 63 30 70 29 60 50 56 36 46 43 32 45 34
 [ reached getOption("max.print") -- omitted 23932 entries ]

#$
```

This is helpful when we work with several different datasets at the same time.

The example also shows that R does not flood our screen with long lists of numbers. Instead we only see the first few numbers, and then the text "*omitted ... entries*".

When we want to use only one dataset, then the command *attach* is helpful.

```
attach(BudgetFood)
age

 [1] 43 40 28 60 37 35 40 68 43 51 43 48 51 58 61 53 58 64 50 50 47 76 49 44 49
[26] 51 56 63 30 70 29 60 50 56 36 46 43 32 45 34
 [ reached getOption("max.print") -- omitted 23932 entries ]
```

From now on, all variables will first be searched in the dataset *BudgetFood*. When we no longer want this, then we say

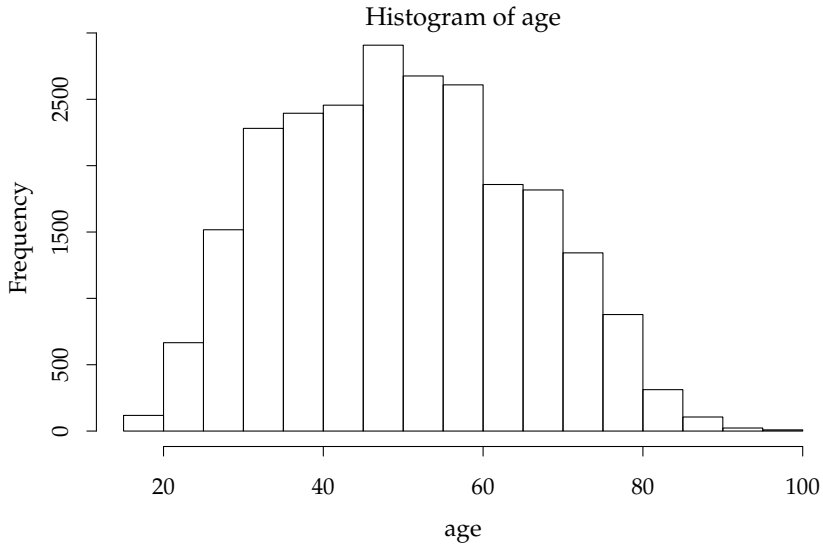
```
detach(BudgetFood)
```

A third possibility is the command *with*:

```
with(BudgetFood,age)

 [1] 43 40 28 60 37 35 40 68 43 51 43 48 51 58 61 53 58 64 50 50 47 76 49 44 49
[26] 51 56 63 30 70 29 60 50 56 36 46 43 32 45 34
 [ reached getOption("max.print") -- omitted 23932 entries ]
```

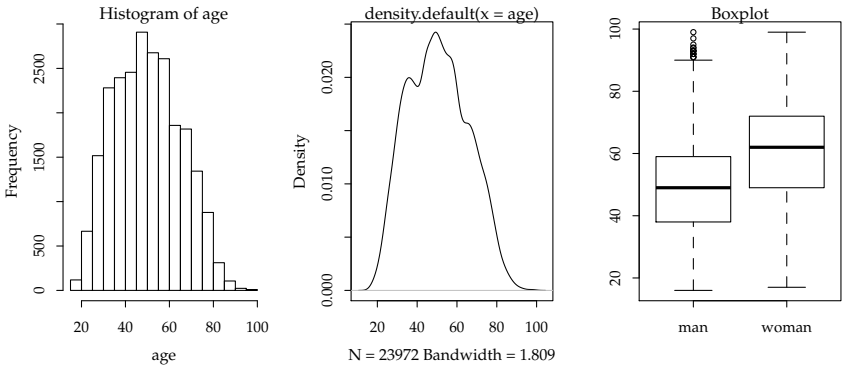
We often use *with* when we use a function and want to refer to a specific dataset in this function. E.g. *hist* shows a histogram:

```
with(BudgetFood,hist(age))
```



Most commands have several options which allow you to fine-tune the result. Have a look at the help-page for *hist* (you can do this with *help(hist)*). Perhaps you prefer the following graph:

```
with(BudgetFood,hist(age,breaks=40,xlab="Age [years]",
     col=gray(.7),main="Spain"))
```



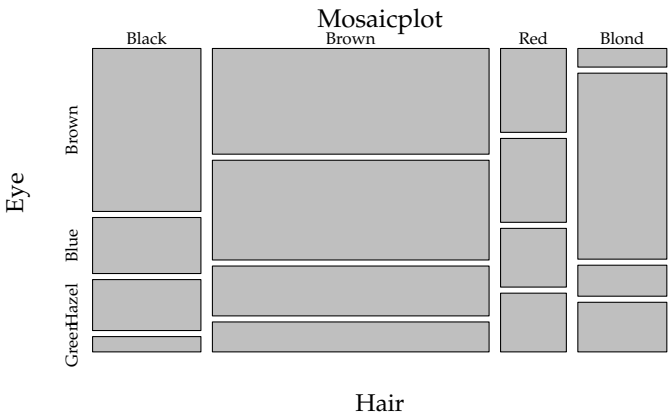
2.6 Graphs

There is more than one way to represent numbers as graphs.

2.7 Basic Graphs

Here are three basic graphs:

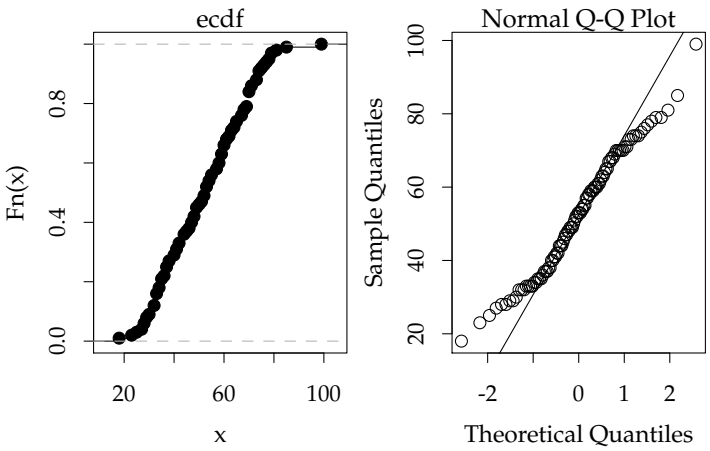
```
with(BudgetFood, {
  hist(age)
  plot(density(age))
  boxplot(age ~ sex,main="Boxplot")
})
```



Two further helpful plots are `ecdf` and `qqnorm`:

```
x <- sample(BudgetFood$age,
            100)
plot(ecdf(x),main="ecdf")
```

```
qqnorm(x)
qqline(x)
```

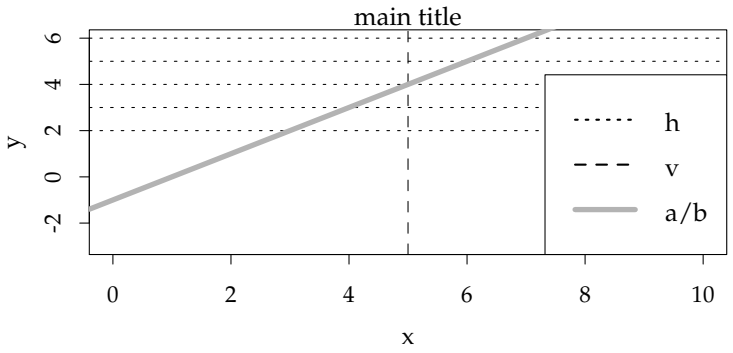


- Sometimes it is obvious how to prepare our data for these functions. Sometimes it is more complicated. Then other commands help and calculate an object that can be plotted (with *plot*)
  - *density*, *ecdf*, *xyplot*...
- Some commands then plot whatever we have prepared:
  - *plot*, *hist*, *boxplot*, *barplot*, *curve*, *mosaicplot*,...
- Yet other commands add something to an existing plot:
  - *points*, *text*, *lines*, *abline*, *qqline*...

2.7.1 Plotting functions

We can plot functions of *x* with *curve*.

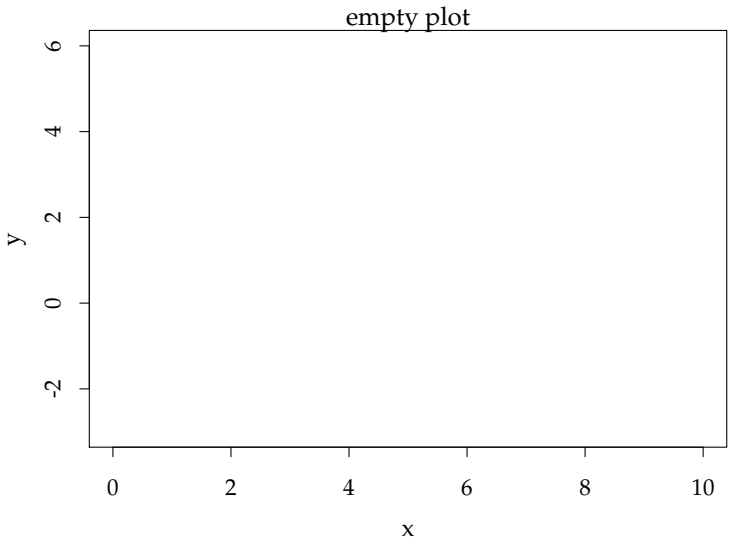
```
curve(dchisq(x,3),from=0,to=10)
```



2.7.2 Empty plots

Sometimes it is helpful to start with an empty plot. Then we have to help *plot* a little bit. Usually, *plot* can guess from the data the limits and labels of the axes. With an empty plot we have to specify them explicitely.

```
plot(NULL,xlim=c(0,10),ylim=c(-3,6),xlab="x",ylab="y",main="empty plot")
```

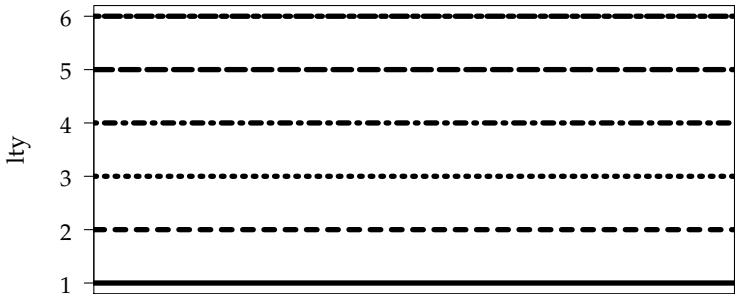


2.7.3 Line type

Almost all commands that draw lines follow the following conventions:

- *lty* linetype (*"dashed"*, *"dotted"*, or simply a number)

```
plot(NULL,ylim=c(1,6),xlim=c(0,1),xaxt="n",ylab="lty",las=1)
sapply(1:6,function(lty) abline(h=lty,lty=lty,lwd=5))
```

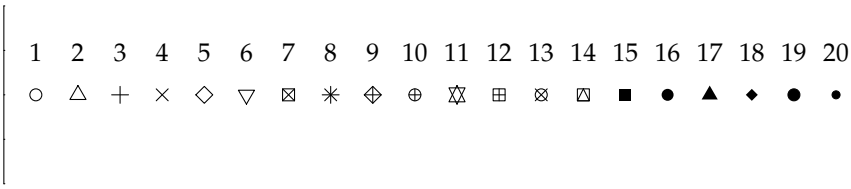


- *lwd* linewidth (a number)
- *col* colour (*"red"*, *"green"*, *gray(0.5)*)

2.7.4 Points

The character used to draw points is determined with *pch*.

```
range=1:20
plot(range,range/range,pch=range,frame=FALSE)
text(range,range/range+.2,range)
```

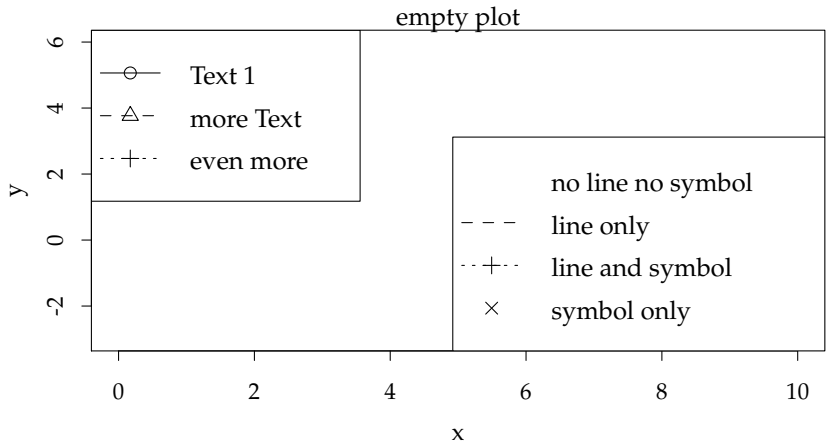


2.7.5 Legends

When we use more than one line or more than one symbol in our plot we have to explain their meaning. This is done in a legend.

Usually *legend* gets as an option a vector of linetypes *lty* and symbols *pch*. They will be used to construct example lines and symbols next to the actual text of the legend. If the *lty* or *pch* is *NA*, then no line or point is drawn.

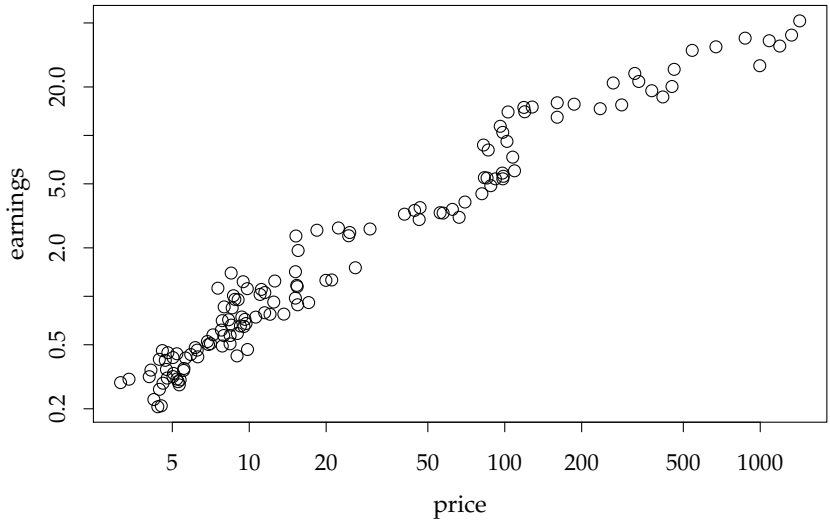
```
plot(NULL,xlim=c(0,10),ylim=c(-3,6),xlab="x",ylab="y",main="empty plot")
legend("topleft",c("Text 1","more Text","even more"),lty=1:3,pch=1:3)
legend("bottomright",c("no line no symbol","line only","line and symbol",
              "symbol only"),lty=c(NA,2,3,NA),pch=c(NA,NA,3,4),bg="white")
```



2.7.6 Auxiliary lines

The command *abline* allows us to add auxiliary lines to a plot.

```
plot(NULL,xlim=c(0,10),ylim=c(-3,6),xlab="x",ylab="y",main="main title")
abline(h=2:6,lty="dotted")
abline(v=5,lty="dashed")
abline(a=-1,b=1,lwd=5,col=grey(.7))
legend("bottomright",c("h","v","a/b"),lty=c("dotted","dashed","solid"),
       col=c("black","black",grey(.7)),lwd=c(2,2,5),bg="white")
```



`abline` knows the following important parameters:

- `h=` for horizonal lines
- `v=` for vertical lines
- `a=..., b=...` for lines with intercept `a` and slope `b`

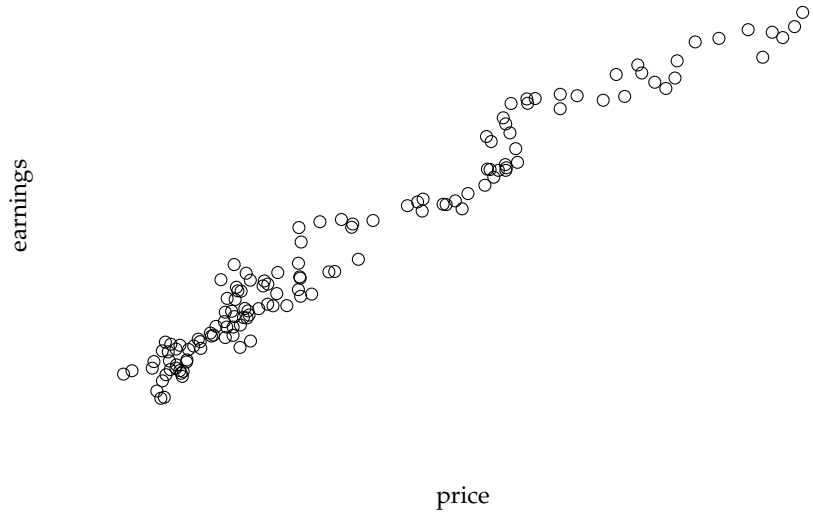
Note, that these arguments can be vectors if we want to draw several lines at the same time.

2.7.7 Axes

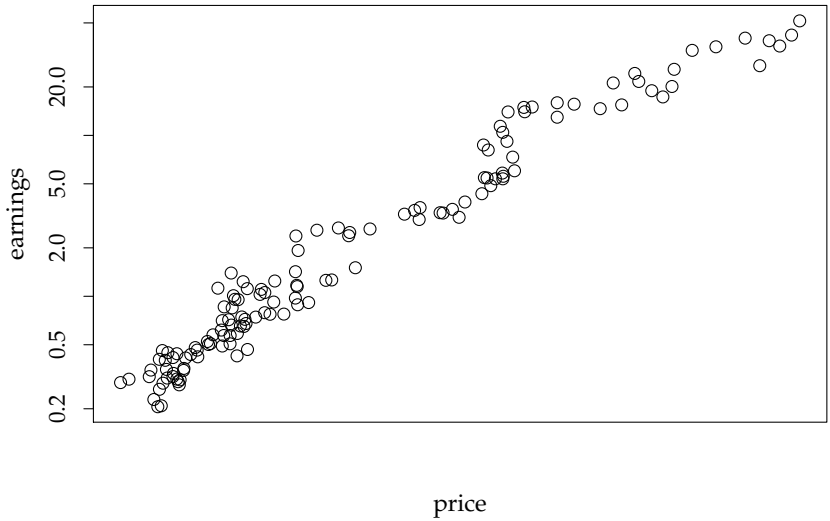
The options `log='x'`, `log='y'` or `log='xy'` determine whether which axis is shown in a logarithmic style.

```
data(PE,package="Ecdat")
xx<-as.data.frame(PE)
attach(xx)
```

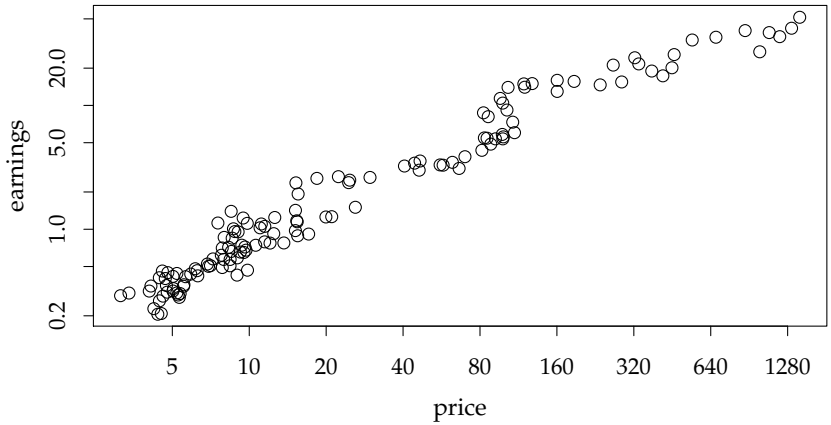
```
plot(price, earnings)
```



```
plot(price, earnings,log="x")
```

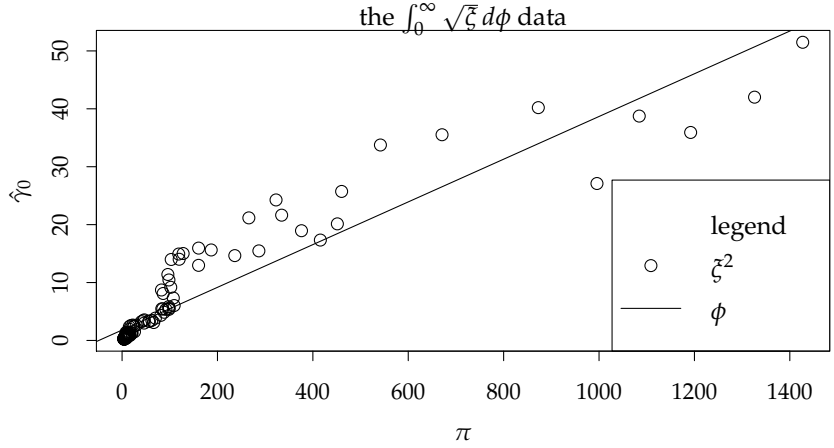


```
plot(price, earnings,log="xy")
```

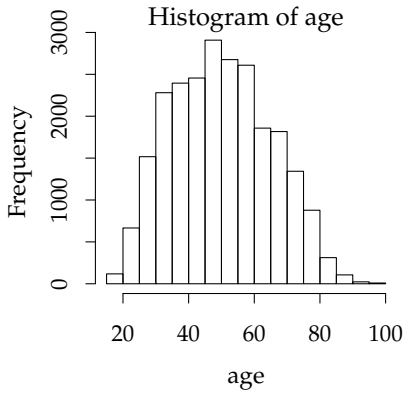


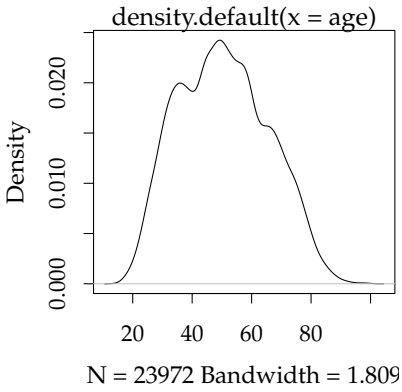
To gain more flexibility `axis` can draw a wide range of axes. Before using `axis` the previous axes can be removed entirely (`axes=FALSE`) or suppressed selectively (`xaxt="n"` or `yaxt="n"`).

```
plot(price, earnings,log="xy",axes=FALSE)
```

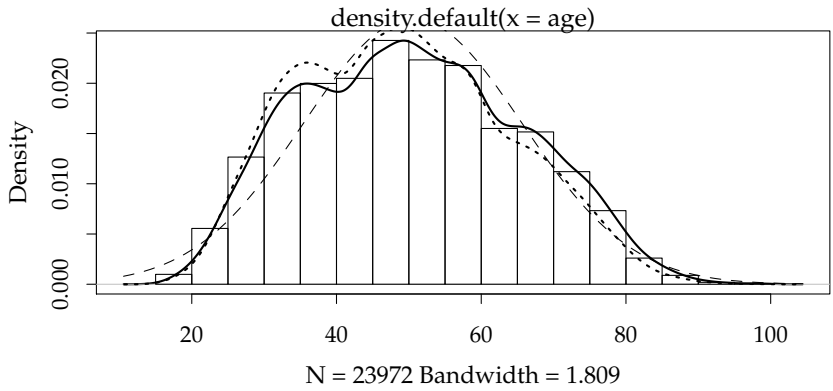


```
plot(price, earnings,log="xy",xaxt="n")
```





```
plot(price, earnings,log="xy",xaxt="n")
axis(1,at=c(5,10,20,40,80,160,320,640,1280))
```

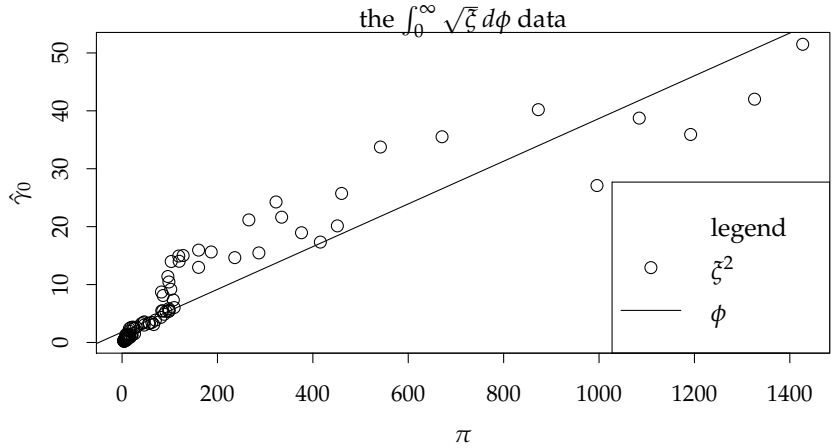


If we specify a lot of axes labels, as in the example above, R does not print them all if they overlap.

2.8 Fancy math

R can also render more than only textual labels with *plotmath*. When we use the tikz-device in Sweave we can also use LaTeX notation for mathematics.

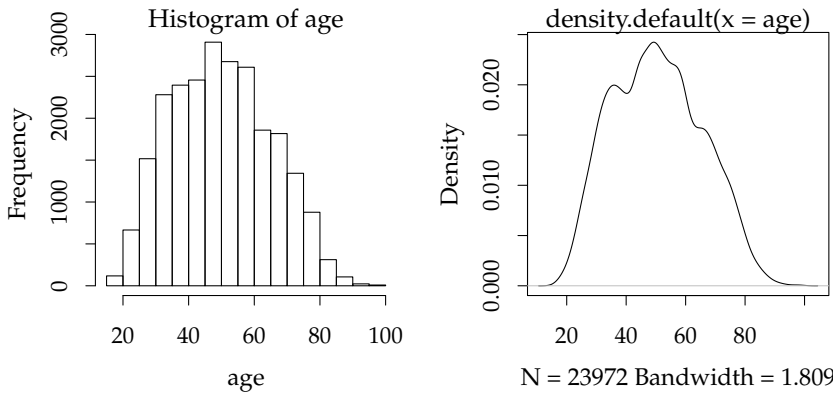
```
plot(price, earnings,xlab="$\\pi$",ylab="$\\hat{\\gamma}_0$",main="the $\\int_0^\\in
abline(lm(earnings~price))
legend("bottomright",c("legend","$\\xi^2$","$\\phi$"),pch=c(NA,1,NA),lty=c(NA,NA,1))
```



2.8.1 Several diagrams

Diagrams side by side To put several diagrams on one plot side by side we can call *par(mfrow=c(...))* or *layout* or *split.screen*.

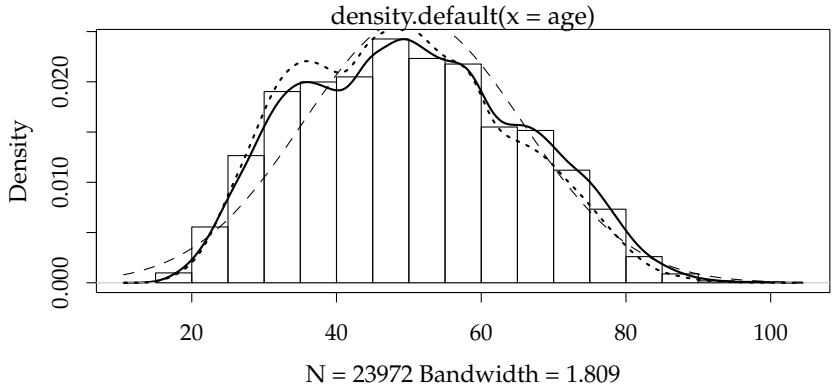
```
par(mfrow=c(1,2))
with(BudgetFood, {
  hist(age)
  plot(density(age))
})
```



Superimposed graphs

- Anything that can create *lines* or *points* (like *density* or *ecdf*) can immediately be added to an existing plot.
- Plot-objects that would otherwise create a new figure (like *plot*, *hist*, or *curve*) can be added to an existing plot with the optional parameter *add=TRUE*.

```
with(BudgetFood, {
  plot(density(age),lwd=2)
  lines(density(age[sex=="man"],na.rm=TRUE),lty=3,lwd=2)
  hist(age,freq=FALSE,add=TRUE)
  curve(dnorm(x,mean(age),sd(age)), add = TRUE,lty=2)
})
```



Coplots We will discuss coplots in section **??**.

2.9 Tables

Tables of frequencies The command *table* calculates a table of frequencies. Here we show only the first 16 columns:

```
with(BudgetFood,table(sex ,age ))[,1:16]

       age
sex     16 17 18 19 20 21 22  23  24  25  26  27  28  29  30  31
  man    3  6 21 21 36 37 87 100 132 201 210 248 254 329 367 363
  woman  0  2  7  9 12 21 19  21  22  26  18  28  10  25  28  12
```

Other statistics The command *aggregate* groups our data by levels of one or several factors and applies a function to each group. In the following example the factor is *sex*, the function is the *mean* which is applied to the variable *age*.

```
library(memisc)
with(BudgetFood,aggregate(mean(age) ~ sex))

    sex mean(age)
1   man     49.09
4 woman     59.47
```

2.10 Regressions

Simple regressions can be estimated with *lm*. The operator ~ allows us to describe the regression equation. The dependent variable is written on the left side of ~, the independent variables are written on the right side of ~.

```
lm (wfood ~ totexp,data=BudgetFood)


Call:
lm(formula = wfood ~ totexp, data = BudgetFood)

Coefficients:
 (Intercept)        totexp
 0.495039722  -0.000000135
```

The result is a bit terse. More details are shown with the command *summary*.

```
summary(lm (wfood ~ totexp,data=BudgetFood))


Call:
lm(formula = wfood ~ totexp, data = BudgetFood)

Residuals:
    Min      1Q  Median      3Q     Max
-0.4931 -0.0937 -0.0100  0.0862  1.0618

Coefficients:
                 Estimate     Std. Error t value Pr(>|t|)
(Intercept)  0.49503972250  0.00156181913   317.0   <2e-16 ***
totexp      -0.00000013485  0.00000000146   -92.4   <2e-16 ***
```

```
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.142 on 23970 degrees of freedom
Multiple R-squared:  0.263,Adjusted R-squared:  0.263
F-statistic: 8.54e+03 on 1 and 23970 DF,  p-value: <2e-16
```

Of course we can also pretty-print these results:

```
library(xtable)
xtable(summary(lm (wfood ~ totexp,data=BudgetFood)))
```

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	0.4950	0.0016	316.96	0.0000
totexp	-0.0000	0.0000	-92.41	0.0000

2.11 Starting and stopping R

Whenever we start R, the program attempts to find a file *.Rprofile*, first in the current working directory, then in the home directory. If the file is found, it is "sourced", i.e. all R commands in this file are executed. This is useful when we want to run the same commands whenever we start R. The following line

```
options(browser = "/usr/bin/chromium-browser")
```

in *.Rprofile* makes sure that the help system of R always uses chromium.

Also when we quit R with the command *q()*, the application tries to make our life easier.

```
q()
```

R first asks us

```
Save workspace image? [y/n/c]:
```

Here we have the possibility to save all the data that we currently use (and that are in our workspace) in a file `.Rdata` in the current working directory. When we start R for the next time (from this directory) R automatically reads this file and we can continue our work.

3 Organising work

3.1 Scripts

Most of the practical work in data analysis and statistics can be see as a sequence of commands to a statistical software.

How can we run these commands?

- Execute a command in the command window (or with mouse and dialog boxes)
  - clumsy
  - hard to repeat actions
  - hard to replicate what we did and why we did it (logs don't really help).
  - hard to find mistakes (structure of the mistake is easy to overlook).
- Write a file (`.R` or `.do`) and execute single lines (or small regions) from the file while editing the file.
  - great way to creatively develop code line by line. Not reproducible since the file changes permanently.
  - one window with the file another window with mainly the R output

- Write a source file (`.R` or `.do`), open it in an editor and then always execute the entire file (while editing the file).
  - great way to creatively develop larger parts of code
- Source "public" files (`.R` or `.do`) from a "master file"

```
source("read_data_130715.R")
source("clean_data_130715.R")
source("create_figures_130715.R")
```

  This is the first step to reproducible research. When our script seems to do what it is supposed to do, we make it "public", give it a unique name, and never change it again.

- From a master file, first source a file which defines functions. Then call these functions.

```
source("functions_XYZ_130715.R")
read_data()
clean_data()
create_figures()
```

  Advantages of using functions:
  - functions can take parameters.
  - several functions can go in one file (still do not harm each other). Systematic changers are easier with only one file.
  - Regardless whether we divide our work into source files or into functions: This division allows us to save time. Some of these steps take a lot of time. Once they work, we do not have to do them over and over again.

Advantages of using source files (with or without functions):

- We keep a record of our work.

- We can work incrementally, fix mistakes and introduce small changes (if we refer to a public file, we should work on a copy of this file with a new name).
- We can use the editor of our choice (Emacs is a nice editor)

3.1.1 Robust scripts

How can we make our scripts robust? Remember:

- the structure of the data may change over time
  - new variables might come with new treatments of our experiment
  - new treatments might require that we code variables differently
- the scripts may not only run on our computer
- the scripts are not always sourced in the same context
- our random number generator may start from different seeds

3.1.2 Robustness towards different computers

we better use relative pathnames
assume that on my computer the script is stored in

```
/home/oliver/projectXYX/R
```

next to it we have

```
/home/oliver/projectXYX/data/munich/1998/test.Rdata
```

From the script I might call either

```
load(file="/home/oliver/projectXYX/data/munich/1998/test.Rdata")
```

or

```
load(file="../data/munich/1998/test.Rdata")
```

The latter assumes that there is a file
../data/munich/1998/test.Rdata
next to the script. But it does not assume that everything is in
/home/oliver/projectXYZ

Hence, the latter works even if my coauthor has stored everything as

```
C:/users/eva/PhD/projectXYX/R
C:/users/eva/PhD/projectXYX/data/munich/1998/test.Rdata
```

If a lot happens in *../data/munich/1998/* anyway, use the *setwd* command

```
setwd("../data/munich/1998/")
...
load(file="test.Rdata")
```

(and remember to make the *setwd* relative, i.e. avoid the following:

```
setwd("/home/oliver/projectXYZ/data/munich/1998/")
...
```

).

3.1.3 Robustness towards changes in context

assume we have the following two files

```
# script1.R
load("someData.Rdata")
# now two variables, x and y are defined
source("script2.R")
```

```
# script2.R
est <- lm ( y ~ x)
```

In this example *script2.R* assumes that variables *y* and *x* are defined. As long as *script2.R* is called in this context, everything is fine.

Changing *script1.R* might have unexpected side effects since we transport variables from one script to the other. The call

```
source("script2.R")
```

does not reveal how *y* and *x* are used by the script.

3.1.4 Functions increase robustness

```
# script1.R
source("script2.R")
load("someData.Rdata")
myFunction(y,x)
```

```
# script2.R
# defines myFunction
myFunction <- function(y,x) {
  est <<- lm ( y ~ x)
  }
```

Now *script2.R* only defines a function. The function has arguments, hence, when we use it in *script1.R* we realise which variable goes where.

Note that the function takes arguments. This is more elegant (and less risky) than to write functions like this one:

```
# script2.R
# defines myFunction
myFunction <- function() {
  est <<- lm ( y ~ x)
  }
```

and then say

```
# script1.R
source("script2.R")
load("someData.Rdata")
x <- ...
y <- ...
myFunction()
```

It will still work, but later it will be less clear to us that the assignments before the function call are essential for the function.

```
myFunction <- function(y,x) {
  est <<- lm ( y ~ x)
  }
```

This function has a side effect. It changes a variable *est* outside the function. Often it is less confusing to define functions with *return values* and no side effects.

```
myFunction <- function(y,x) {
  lm ( y ~ x)
  }
```

When we call this function later as

```
est <- myFunction(y,x)
```

it is clear where the result of the function goes.

Recap

- Functions which use global variables — risky
- Functions with side effects — risky
- Functions which only use arguments and return values — often better

Note: If we replace functions by scripts: Scripts must use global variables and can only produce side effects. — Scripts are more likely to lead to mistakes than functions.

→ replace scripts by functions (with arguments) whenever possible.

3.2 Calculations that take a lot of time

If a sequence of functions takes a lot of time to run, let it generate intermediate data.

Our master-R-file could look like this:

```
set.seed(123)
...
source("projectXYZ_init_130715.R")
getAndCleanData() # takes a lot of time
save(cleanedData,file="cleanedData130715.Rdata")

load("cleanedData130715.Rdata")
doBootstrap() # takes a lot of time
save(bsData,file="bsData130715.Rdata")

load("cleanedData130715.Rdata")
load("bsData130715.Rdata")
doFigures()
...
```

3.3 Nested functions

If our functions become long and complicated, we can divide them into small chuncs.

```
...
doAnalysis <- function () {
  firstStepAnalysis()
  secondStepAnalysis()
  thirdStepAnalysis()
  ...
}

firstStepAnalysis <- function() {
  ...
}

secondStepAnalysis <- function() {
  ...
}
...
```

Actually, if we need some functions only within a specific other function then we can define them within this function:

```
...
doAnalysis <- function () {
  firstStepAnalysis <- function() {
  ...
  }
  secondStepAnalysis <- function() {
  ...
  }
  firstStepAnalysis()
  secondStepAnalysis()
  thirdStepAnalysis()
  ...
}
```

- Advantage: These function are only visible from within *doAnalysis* and can not do any harm elsewhere (where we, perhaps, defined functions with the same name that do different things).

Nesting of functions has three advantages:

- it structures our work
- it facilitates debugging
- it facilitates exchange with our coauthors
  "…there is a problem in *thirdStepAnalysis*…"

3.4 Reproducible randomness

```
set.seed(123)
```

Random numbers affect our results:

- Simulation
- Bootstrapping

- Approximate permutation tests
- Selection of training and confirmation samples
- ...

3.5 Recap — writing scripts and using functions

- If there is a systematic structure in our problem, then we can exploit it
- If we make mistakes, let us make them systematically!

```
N <- 100
profit88 <- rnorm(N)
profit89 <- rnorm(N)
profit98 <- rnorm(N)
myData <- as.data.frame(cbind(profit88,profit89,profit98))
```

Compare

```
t.test(profit88,data=myData)$p.value
t.test(profit89,data=myData)$p.value
t.test(profit98,data=myData)$p.value
```

with

```
sapply(grep("profit",colnames(myData),value=TRUE),function(x)
       t.test(myData[,x])$p.value)
```

The first looks simpler.
The second is more robust against

- a change in the dataset
- a change in the names of the variables
- adding another *profit*-variable
- typos

3.6 Human readable scripts

- Weaving and knitting → we do this later
- Comments at the beginning of each file

```
# scriptExample130715.R
#
# the purpose of this script is to illustrate the use of
#                                                 comments
#
# first version: 130715
# this version:  130715
# last change by: Oliver
# requires:      test130715.Rdata, someFunctions130715.R
# provides:      ...
#
set.seed(123)
```

- Comments at the beginning of each function

```
#
# exampleFun transforms two vectors into an example
# side effects: ...
# returns: ...
#
exampleFun <- function(x,y) {
  ...
}
```

- Comment non-obvious steps

```
#
# to detect outliers we use lrt-method.
# We have tried depth.trim and depth.pond
# but they produce implausible results...
outl <- foutliers(data,method="lrt")
```

- Document your thoughts in your comments

```
  ...
# 13/07/21: although I thought that age should not affect
#           profits, it does here! I also checked
#           xyz-specification and it still does.
#           Perhaps age is a proxy for income.
#           Unfortunately we do not have data on
#           income here.
...
```

- Formatting
  Compare

```
lm ( s1 ~ trust + ineq + sex + age + latitude )
lm ( otherinvestment ~ trust + ineq + sex + age + latitude )
```

  with

```
lm ( s1              ~ trust + ineq + sex + age + latitude )
lm ( otherinvestment ~ trust + ineq + sex + age + latitude )
```

Insert linebreaks Compare

```
lm ( otherinvestment ~ trust + ineq + sex + age + latitude, data=trustExp, subset=se
```

with

```
lm ( otherinvestment ~ trust + ineq + sex + age + latitude,
                       data=trustExp,
                       subset=sex=="female" )
```

- Variables names
  short but not too short

```
lm ( otherinvestment ~ trustworthiness + inequalityaversion + sexOfProposer + a
lm ( otherinvestment ~ trust + ineq + sex + age + latitude)
lm ( oi ~ t + i + s + a + l1 + l2)
lm ( R100234 ~ R100412 + R100017 + R100178 + R100671 + R100229 )
```

We will say more about variable names in section 6.3.

- Abbreviations in scripts

  R (and other languages too) allows you to refer to parameters in functions with names:

```
qnorm(p=.01,lower.tail=FALSE)
```

  To save space, you can abbreviate these names:

```
qnorm(p=.01,low=FALSE)
```

4 Some programming techniques

4.1 Debugging functions

general strategies: debug the function with a simple example
take a subsample of the data

```
library(Ecdat)
data(Kakadu)
```

```
sqMean <- function (x) {
   z <- mean(x)
   z^2
}
sqMean(Kakadu$lower)
xx <- sample(Kakadu$lower,10)
xx
sqMean(xx)
```

Assume that we still do not trust the function. *debug* allows us to debug a function. *ls* allows us to list the variables in the current environment.

```
debug(sqMean)
sqMean(xx)
undebug(sqMean)
```

If the function returns with an error, it helps to set

```
options(error=recover)
```

In the following function we refer to the variable *xxx* which is not defined. The function will, hence, fail. With *options(error=recover)* we can inspect the function at the time of the failure.

```
sqMean <- function (x) {
   z <- mean(xxx)
   z^2
}
sqMean(xx)
```

4.2 Lists of variables

To make the analysis more consistent.

Whenever things repeat, we define them in variables at the top of the paper:

```
model1 <- "income"
model2 <- "income + age + sex"
model3 <- "income + age + sex + conservation + vparks"
```

(We use here character strings to represent parts of formulas. Alternatively, we could also store objects of class *formula*. However, manipulating these objects is not always to obvious. To keep things simple, we will use character strings here.) Later in the paper we compare the different models:

```
mylm <- function (model) lm(paste("as.integer(answer) ~ ",model),data=Kakadu)
lm1 <- mylm (model1)
lm2 <- mylm (model2)
lm3 <- mylm (model3)
```

```
mtable(Model1=lm1,Model2=lm2,Model3=lm3)
mylogit <-function(model) glm(paste("answer=='yy' ~ ",model),
                              data=Kakadu,family=binomial(link=logit))
```

```
est1 <-mylogit(model1)
est2 <-mylogit(model2)
est3 <-mylogit(model3)
mtable(Model1=est1,Model2=est2,Model3=est3)
```

Similarly, we might define at the beginning of the paper…

- lists of random effects
- lists of variables to group by
- palettes for plots

4.3 Return values of functions

Most functions do not only return a number (or a vector) but rather complex objects. In R *str()* helps us to learn more about the structure of these objects. (In Stata similar return values are provided by *return*, *ereturn*, and *sreturn*)

```
lm1 <- mylm (model1)
str(lm1)
```

There are at least two ways to extract data from these objects:

- Extractor functions

  ```
  coef(lm1)
  effects(lm1)
  fitted.values(lm1)
  residuals(lm1)
  vcov(lm1)
  hccm(lm1)
  logLik(lm1)
  ```

  (the equivalent in Stata are postestimation commands)

- Whatever is a list item can also be accessed directly:

```
lm1$coefficients
lm1$residuals
lm1$fitted.values
lm1$residuals
```

Note: Some interesting values are not provided by the *lm*-object itself. These can often be accessed as part of the *summary*-object.

```
slm1 <- summary(lm1)
slm1$r.squared
slm1$adj.r.squared
slm1$fstatistic
```

4.4 Repeating things

Looping The simplest way to repeat a command is a loop:

```
for (i in 1:10) print(i)
```

If the command is a sequence of expressions, we have to enclose it in braces.

```
for (i in 1:10) {
  x <- runif(i)
  print(mean(x))
}
```

Avoiding loops In R loops should be avoided. It is more efficient (faster) to apply a function to a vector.

```
sapply(1:10,print)
```

Or, the more complex example:

```
sapply(1:10,function(i) {
  x <- runif(i)
  mean(x)
})
```

Note that *sapply* already returns a vector which is in many cases what we want anyway.

In the above examples we applied a function to a vector. Sometimes we want to apply functions to a matrix.

Applying a function along one dimension of a matrix In the following example we apply a function along the second dimension of the dataset *Kakadu*.

```
apply(Kakadu,2,function(x) mean(as.integer(x)))
```

Rectangular and ragged arrays Rectangular array:

wide

	a	b	c
A	1	2	3
B	4	5	6

long

hor	vert	x
a	A	1
b	A	2
c	A	3
a	B	4
b	B	5
c	B	6

Ragged array:

wide

	a	b	c
A		2	3
B	4	5	

long

hor	vert	x
b	A	2
c	A	3
a	B	4
b	B	5

In R ragged arrays can be represented as datasets grouped by one or more factors. These variables describe which records belong together (e.g. to the same person, year, firm,...)

In the following example we use the dataset *Fatality*. This dataset contains for each state of the United States and for each year in 1982 to 1988 the trafic fatality rate.

```
data(Fatality)
```

```
by(Fatality,list(Fatality$year),function(x) mean(x$mrall))
by(Fatality,list(Fatality$state),function(x) mean(x$mrall))
```

by does not return a vector but an object of class *by*. If we actually need a vector we have to use *c* and *sapply*.

In the following example we let by actually return two values.

```
byObj <- by(Fatality,list(Fatality$year),
            function(x) c(fatality=mean(x$mrall),
                          meanbeertax=mean(x$beertax)))
sapply(byObj,c)

              1982   1983   1984   1985   1986   1987   1988
fatality    2.0891 2.0078 2.0171 1.9737 2.0651 2.0607 2.0696
meanbeertax 0.5303 0.5324 0.5296 0.5169 0.5087 0.4951 0.4798
```

We can do more complicated things in *by*. In the following example we calculate a regression. To get only the coefficients from the regression (and not fitted values, residuals, etc.) we use the extractor function *coef*.

```
byObj <- by(Fatality,list(Fatality$year),function(x)
            lm(mrall ~ beertax + jaild, data=x))
sapply(byObj,coef)

              1982   1983   1984   1985   1986   1987   1988
(Intercept) 1.9080 1.7504 1.6768 1.6567 1.7109 1.7188 1.7412
beertax     0.1824 0.2992 0.4067 0.4058 0.4945 0.4920 0.4509
jaildyes    0.4501 0.3625 0.4283 0.3430 0.3286 0.3369 0.3843
```

Applying a function to each element of a ragged array *by* is very powerful. It offers the entire subset of the dataframe, as defined by the index variable, to the function. The function can then combine these values in any way. Sometimes we want simply to apply the same function to each column of a ragged array.

```
aggregate(Fatality,list(year=Fatality$year),mean)
```

Again, the function (which was *mean* in the previous example) can be defined by us:

```
aggregate(Fatality,list(year=Fatality$year),
          function(x) sd(x)/mean(x))
```

5 Data manipulation

5.1 Subsetting data

There are several ways to access only a part of a dataset:

- Many functions take an option *...,subset=...*
- The *subset()* function
- The first index of the dataset

```
lm(Offer ~ sex, data=trustGS$subjects, subset = Period == 6)
subset(trustGS$subjects, Date == "130716_0601" & Subject == 1 )
trustGS$subjects[ trustGS$subjects$Date=="130716_0601" , ]
```

5.2 Merging data

- Appending two datasets (outer join)

  ```
  merge(x,y,all=TRUE)
  ```

  (works, as long as the two datasets have a variable that is different for both, e.g. date, otherwise use *rbind.fill* from the *plyr* library).

  (In Stata this is done by *append*)

- Matching two datasets (inner join)

```
merge(x,y)
```

  (In Stata this is done by *merge*)

- Joining two datasets (left join)

```
merge(x,y,all.x=TRUE)
```

  (In Stata this is done by *joinby*)

Appending In the following example we first split the data from an experiment into two parts. Merge helps us to append them to each other.

```
load("130716_060x.Rdata")
experiment1 <- subset(trustGS$subjects,Date=="130716_0601")
experiment2 <- subset(trustGS$subjects,Date=="130716_0602")
dim(experiment1)

[1] 108  14

dim(experiment2)

[1] 108  14

dim(merge(experiment1,experiment2,all=TRUE))

[1] 216  14
```

merge tries to find common variables, but does not find any matches (i.e. no records which have the same *Date*, *Subject*, etc. in both datasets). Without the *all=TRUE* option, *merge* would simply return an empty dataset. With this option, *merge* keeps records from both datasets, even if they are not matched (which in this case they are not supposed to be).

Joining A frequent application for a join are tables in z-Tree that have something to do with each other. E.g. the globals and the subjects tables both provide information about each period. Common variables in these tables are *Date*, *Treatment*, and *Period*.

By merging globals with subjects, *merge* looks up for each record in the subjects table the matching record in the globals table and adds the variables which are not already present in subjects.

In the following example we simply get two more variables in the dataset (*NumPeriods* and *RepeatTreatment*). With more variables in globals we would, of course, also get more variables.

```
dim(trustGS$global)

[1] 24  5

dim(trustGS$subject)

[1] 432  14

dim(merge(trustGS$global,trustGS$subject))

[1] 432  16
```

Joining aggregates A common application for a join is a comparison of our individual data with aggregated data. Let us come back to the Fatalities example. We want to compare the traffic fatility rate *mrall* for each state with the average values for each year.

```
merge(Fatality,aggregate(c(avgMrall=mean(mrall)) ~ year,data=Fatality))[1:8,]

  year state mrall beertax mlda jaild comserd vmiles unrate perinc avgMrall
1 1982     1 2.128  1.5394   19    no       no  7.234   14.4  10544    2.089
2 1982    30 3.155  0.3464   19   yes       no  8.284    8.6  12033    2.089
3 1982    10 2.033  0.1730   20    no       no  7.652    8.5  14264    2.089
 [ reached getOption("max.print") -- omitted 5 rows ]
```

merge has joined the two datasets, the large Fatality one, and the small aggregated one, on the two variables *year* and *state*.

5.3 Reshaping data

Sometimes we have different observations of the same (or similar) variable in the same row (e.g. *profit.1* and *profit.2*), sometimes we have them stacked in one column (e.g. as *profit*). We call the first format wide, the second long.

For the long case we need a variable that distinguishes the different instances of this variable (*profit.1* and *profit.2*) from each other. Such a variable is called *timevar* (Stata call them *j*).

We also need one or more variables that tells us, which observations actually belonged to one row in the wide format. We call these variables *idvar* (Stata call this variable *i*).

Let us look at a part of our trust dataset

```
trustLong <- trustGS$subjects[,c("Date","Period","Subject","Pos",
                                 "Group","Offer")]
trustLong[1:4,]

         Date Period Subject Pos Group Offer
1 130716_0601      1       1   2     1 0.000
2 130716_0601      1       2   2     4 0.000
3 130716_0601      1       3   1     5 0.495
4 130716_0601      1       4   2     2 0.000

trustWide <- reshape(trustLong,v.names=c("Offer","Subject"),
                     idvar=c("Date","Period","Group"),timevar="Pos",
                     direction="wide")
trustWide[1:4,]

         Date Period Group Offer.2 Subject.2 Offer.1 Subject.1
1 130716_0601      1     1       0         1  0.5100        13
2 130716_0601      1     4       0         2  0.5580         5
3 130716_0601      1     5       0         7  0.4950         3
4 130716_0601      1     2       0         4  0.8422         8
```

```
reshape(trustWide,direction="long")[1:4,]

                          Date Period Group Pos Offer Subject
130716_0601.1.1.2 130716_0601      1     1   2     0       1
130716_0601.1.4.2 130716_0601      1     4   2     0       2
```

```
130716_0601.1.5.2 130716_0601      1     5   2     0       7
130716_0601.1.2.2 130716_0601      1     2   2     0       4
```

↑ Reshaping back returns more or less the orignal data. The ordering has changed and rows have got names now.

6 Preparing Data

- read data
- check structure (names, dimension, labels)
- check values
- create new data:
  - recode variables
  - rename variables
  - label variables
  - eliminate outliers
  - reshape data

6.1 Reading data

6.1.1 Reading z-Tree Output

The function
zTreeTables(...vector of filenames...[,vector of tables])
reads zTree .xls files and returns a list of tables. Here we use *list.files* to find all files that match the typical z-Tree pattern. If we ever get more experiments our command will find them and use them.

```
setwd("rawdata/Trust/")
files <- list.files(pattern = "[0-9]{6}_[0-9]{4}.xls",recursive=TRUE)
trustGS <- zTreeTables(files)
setwd("../..")
```

As long as we need only a single table, we can access, e.g. the subjects table with *$subjects* .

If we need, e.g. the globals table together with the subjects table, we can merge them:

```
with(trustGS,merge(globals,subjects))
```

6.1.2 Reading and writing R-Files

If we want to save one or more R objects in a file, we use *save*

```
save(trustGS,zTreeTables,file="130716_060x.Rdata")
```

To retrieve them, we use *load*

```
load("130716_060x.Rdata")
```

Advantages:

- Rdata is very compact, files are small
- All attributes are saved together with the data
- We can save functions together with data

6.1.3 Reading Stata Files

There are two commands that help you reading Stata files:

One is *read.dta* which is part of *library(foreign)*

```
library(foreign)
sta <- read.dta("130716_060x.dta")
```

The other is *Stata.file* which is part of *library(memisc)*

```
sta2 <- Stata.file("130716_060x.dta")
```

The main difference is that internal Stata information is stored in different places. When we use *read.dta* all additional information is stored as attributes of the data frame and not together with the variable.

```
str(sta)
attributes(sta)
```

Stata.file stores variable labels as attributes of the variables:

```
codebook(sta2)
str(sta2)
attributes(sta2)
```

Very often this is more intuitive. Some packages are, however, confused by these attributes.

6.1.4 Reading CSV Files

CSV-Files (Comma-Separated-Value) Files are in no way always comma separated. The term is rather used to denote any table with a constant separator. Some of the parameters that always change are:

- Separators: , ; TAB
- Quoting of strings: " ' —
- Headers: with / without

As a result, the *read.table* has many parameters.

```
csv <- read.csv("130716_060x.csv",sep="\t")
str(csv)
```

The advantage of CSV as a medium to exchange data is: CSV can be read by any software.

The disadvantage is: No extra information (variable labels, levels of factors, . . .) can be stored.

6.1.5 Filesize

For our example we obtain the following sizes:

Format	Size / Bytes
xlsx	96048
xls	30856
dta	19468
csv	17791
Rdata	31176

6.2 Checking Values

```
load("130716_060x_C.Rdata")
```

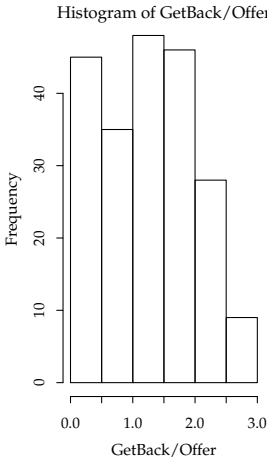
6.2.1 Range of values

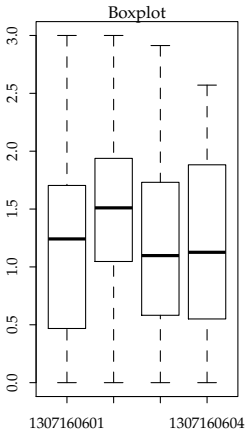
```
codebook(data.set(trustC))
```

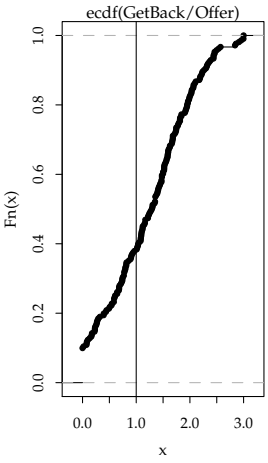
6.2.2 (Joint) distribution of values

Basic plots

```
with(trustC,hist(GetBack/Offer))
boxplot(GetBack/Offer ~ sub("_","",Date),data=trustC,main="Boxplot")
with(trustC,plot(ecdf(GetBack/Offer)))
abline(v=1)
```

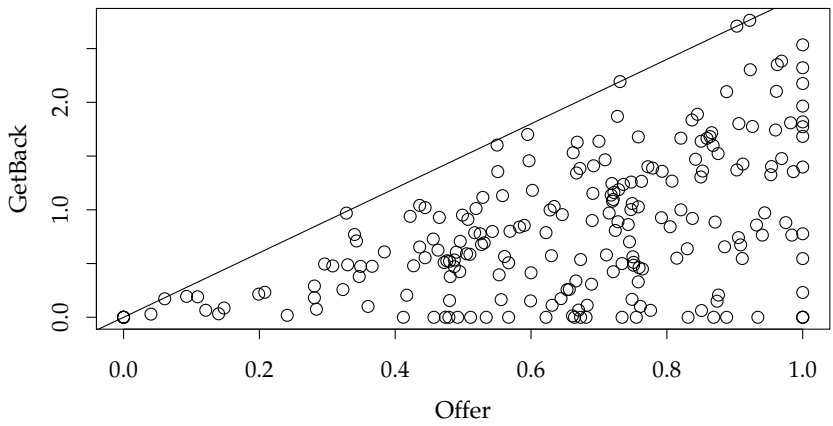






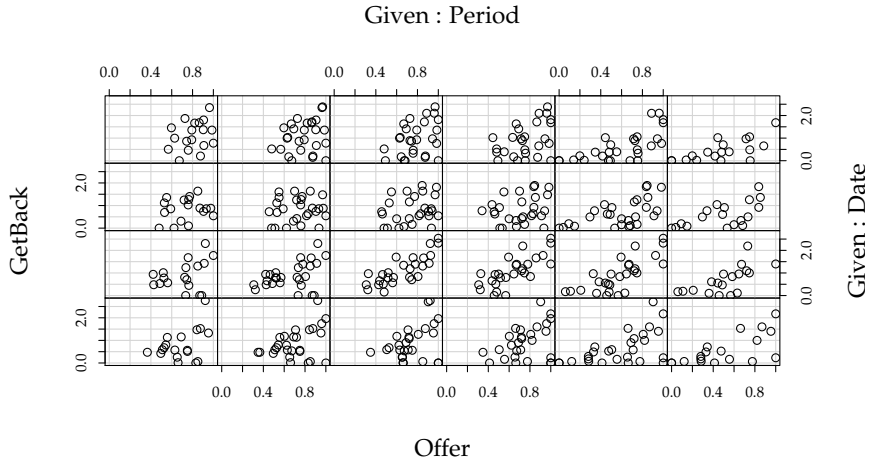
Joint distributions First pool all data:

```
plot(GetBack ~ Offer ,data=trustC)
abline(a=0,b=3)
```



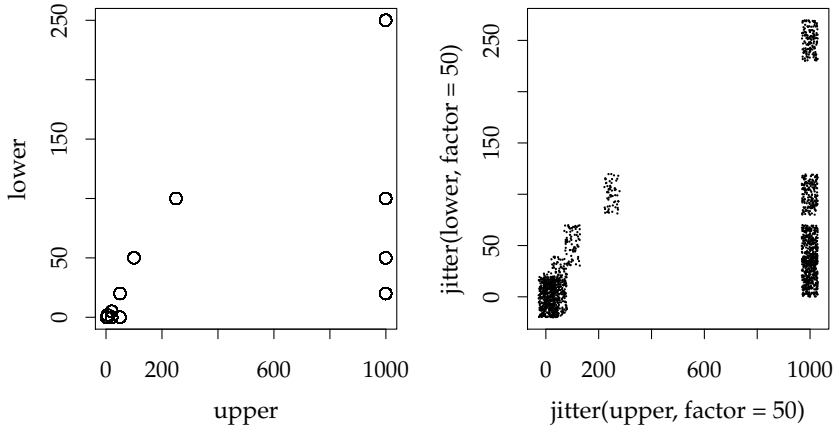
If something is suspicious (which does not seem to be the case here) plot the data for subgroups:

```
coplot(GetBack ~ Offer | Period + Date,data=trustC,show.given=FALSE)
```



When our data falls into a small number of categories a simple scatterplot is not too informative. The right graph shows a scatterplot with some jitter added.

```
data(Kakadu)
plot(lower ~ upper,data=Kakadu)
plot(jitter(lower,factor=50) ~ jitter(upper,factor=50),cex=.1,
     data=Kakadu)
```



With such a large number of observations, and such a small number of categories, a table might be more informative

```
with(Kakadu,table(lower,upper))

     upper
lower   2   5  20  50 100 250 999
  0   129 147 156 176   0   0   0
  2     0   9   0   0   0   0   0
  5     0   0  63   0   0   0   0
  20    0   0   0  69   0   0 321
  50    0   0   0   0  76   0 281
 [ reached getOption("max.print") -- omitted 2 rows ]
```

6.2.3 (Joint) distribution of missings

- Do we expect any missings at all?
- Are missings where they should be?
  - e.g. number of siblings=0, age of oldest sibling=NA ☑

– e.g. number of siblings=NA, age of oldest sibling=25 ↯

The discussion of value labels in section 6.5 contains more details on missings.

6.2.4 Checking signatures

How can we make sure that we are working on the "correct dataset"?

Assume you and your coauthors work with what you think is the same dataset, but you get different results.

Solution: compare checksums.

```
library(tools)
md5sum("130716_060x.Rdata")

                  130716_060x.Rdata
"a98b6b8677b8a093580702838b622bf5"
```

It might be worthwile to include in the draft version of your paper the checksum of your datasets.

6.3 Naming variables

We already mentioned variable names in section 88.

- short but not too short

```
lm ( otherinvestment ~ trust + ineq + sex + age + latitude + longitude)
lm ( R100234 ~ R100412 + R100017 + R100178 + R100671 + R100229 + R100228 )
lm ( otherinvestment ~ trustworthiness + inequalityaversion + sexOfProposer + a
lm ( oi ~ t + i + s + a + l1 + l2)
```

- changing existing variables creates confusion, better create new ones

- Keep related variables alphabetically together.

  `... ProfitA ProfitB ProfitC ...`

  and not

  `... AProfit BProfit CProfit ...`

- How do we order variable names anyway?

```
trustC[,sort(names(trustC))]
```

6.4 Labeling (describing) variables

- Variable names should be short...
- but after a while we forget the exact meaning of a variable
  - What was the difference between *Receive* and *GetBack* ?
  - Did we code *male=1* and *female=2* or the opposite?
- Labels provide additional information.

```
load("130716_060x.Rdata")
trust <- within(with(trustGS,merge(globals,subjects)), {
  description(Pos)<- "(1=trustor, 2=trustee)"
  description(Offer)<- "trustor's offer"
  description(Receive)<- "amount received by trustee"
  description(Return)<- "amount trustee sends back to
                                              trustor"
  description(GetBack)<- "amount trustor receives back
                                          from trustee"
  description(country)<- "country of origin"
  description(sex)<- "participant's sex (1=male, 2=female)"
  description(siblings)<- "number of siblings"
  description(age)<- "true age"
  })
codebook(data.set(trust))
attr(trust,"annotation")<-"Note: 130716_0601 was a pilot,..."
annotation(trust)["note"]="Note: This is not a real dataset..."
```

- labels can be long, but they should be meaningful even if they are truncated.

  The following is not a label but a wording:

```
description(uncondSend) <-  "how much would you send to the
            other player if no binding contract was possible"
description(condSend) <-  "how much would you send to the
 other player if you had the possibility of a binding contract"
```

  Better:

```
description(uncondSend) <-  "how much to send without binding
                                                    contract"
description(condSend) <-  "how much to send with binding
                                                    contract"
wording(uncondSend) <-  "how much would you send to the other
  player if no possibility of a binding contract was possible"
wording(condSend) <-  "how much would you send to the other
     player if you had the possibility of a binding contract"
```

General attributes

description()	short description of the variable	always
wording()	wording of a question	if necessary
annotation()["..."]	e.g. specific property of dataset	if necessary
	how a variable was created	if necessary

6.5 Labeling values

Let us again list some interesting datatypes:

- numbers: 1, 2, 3
- characters: “male”, “female”, . . .
- factors: “male”=1, “female”=2,. . .

  - technically an integer + "levels", often treated as a character
  - can have only one type of missing (is not really a restriction, since the type of missingness could be stored in another variable)

The *memisc*-package provides another type:

- item: "male"=1, "female"=2,...

  technically a number + "levels", often treated as a number

  can have several types of missing. Useful, when we get data from a questionnaire (or from z-Tree).

```
codebook(trustC$sex)

================================================================================

   trustC$sex 'participant's sex (1=male, 2=female)'

--------------------------------------------------------------------------------

   Storage mode: double
   Measurement: nominal
   Missing values: 98, 99

   Values and labels    N    Percent

       1   'male'     174   44.6 40.3
       2   'female'   216   55.4 50.0
      98 M 'refused'   18         4.2
      99 M 'missing'   24         5.6
```

```
table(as.factor(trustC$sex),useNA="always")


  male female   <NA>
   174    216     42

table(as.numeric(trustC$sex),useNA="always")
```

```
   1    2 <NA>
 174  216   42

table(as.character(trustC$sex),useNA="always")


 female    male missing refused    <NA>
    216     174      24      18       0
```

We see that *table* with the option *useNA="always"* allows us to count missings. *mean(is.na())* allows us to calculate the fraction of missings. The result depends on the representation.

```
mean(is.na(trustC$sex))

[1] 0

mean(is.na(as.factor(trustC$sex)))

[1] 0.09722

mean(is.na(as.numeric(trustC$sex)))

[1] 0.09722

mean(is.na(as.character(trustC$sex)))

[1] 0
```

How do we add labels to values? (requires *memisc*)

```
trust <- within(trust,{
  labels(sex)<-c("male"=1,"female"=2,"refused"=98,"missing"=99)
  labels(siblings)<-c("refused"=98,"missing"=99)
  labels(age)<-c("refused"=98,"missing"=99)
  labels(country)<-c("a"=1, "b"=2, "c"=3, "d"=4, "e"=5, "f"=6, "g"=7, "h"=8, "i"=9,
  missing.values(sex)<-c(98,99)
  missing.values(siblings)<-c(98,99)
  missing.values(age)<-c(98,99)
  missing.values(country)<-c(98,99)
  })
```

6.6 Recoding data

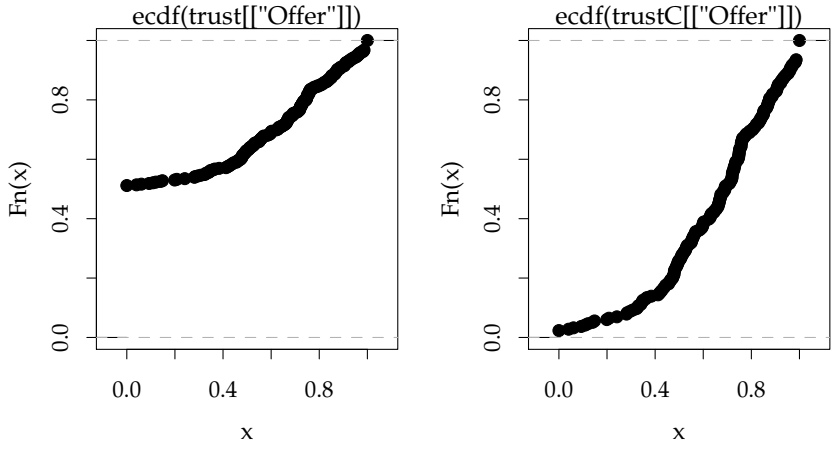
6.6.1 Replacing meaningless values by missings

In our trust game not all players have made all decisions. z-Tree coded these "decisions" as zero. This can be confusing. Better code them as missing.

```
trustC <- within(trust, {
  Offer   [Pos==2 & Offer==0]   <-NA
  GetBack[Pos==2 & GetBack==0]<-NA
  Receive[Pos==1 & Receive==0]<-NA
  Return [Pos==1 & Return==0] <-NA
  })
```

Introducing missings makes a difference. The left graph shows the plot where missings were coded (wrongly) as zero, the right graph shows the plot with missings.

```
plot(ecdf(trust[["Offer"]]))
plot(ecdf(trustC[["Offer"]]))
```



```
mean(trust$Offer)

[1] 0.3268

mean(trustC$Offer,na.rm=TRUE)

[1] 0.6537
```

6.6.2 Replacing values by other values

Sometimes we want to simplify our data. E.g. the siblings variable might be too detailed.

```
trustC <- within(trustC,altSiblings<-recode(siblings,
                  "single child"=0 <- 0,
                  "siblings"    =1 <- range(1,50),
                  "refused"     =98 <- 98,
                  "missing"     =99 <- 99))
```

6.6.3 Comparison of missings

We can not compare *NA*s. The following will fail in R:

```
if(NA == NA) print("ok")
if(7 < NA) print("ok")
```

(Note that the equivalent in Stata, `. == .` and `7 < .`, do not fail but returns TRUE.)

6.7 Creating new variables

- give them new names
- give them labels
- keep the old variables

6.8 Select subsets

(See the remarks on subsetting in section 5.1)

- delete records you will never ever use (in the cleaned data, not in the raw data)

```
trust<-subset(trust,Pos!=2)
```

- generate indicator variables for records you will use in a specific context

```
trust<-within(trust,youngSingle <- age<25 & siblings==0)
with(subset(trust,youngSingle),...)
```

7 Weaving and tangling

- Describe the research question

  Which model do we use to structure this question?

- Describe the sample

  How many observations, means, distributions of main variables, key statistics

  Is there enough variance in the independent variables to test what you want to test?

- Test model

  possibly different variants of the model (increasing complexity)

- Discuss model, robustness checks

7.1 How can we link paper and results?

Lots of notes in the paper, e.g. the following:
In your LaTeX-file...:

```
%
% the following table was created by tableAvgProfits()
%                              from projectXYZ_130621.R
% \begin{table}
% ...
```

Better: Weave (Sweave, knitr)

7.2 A history of literate programming

Donald Knuth: The CWEB System of Structured Documentation (1993)

- CTANGLE: *foo.w* → *foo.c*
- CWEAVE: *foo.w* → *foo.tex*
  (may contain parts of *foo.c*)

What is "literate programming":

- meaningful and readable high-quality documentation
- details that are usually not included in *#comments*
- supposed to be read
- facilitates feedback and reuse of code
- reduces the amount of text one must read to understand the code

Literate programming for empiricists:

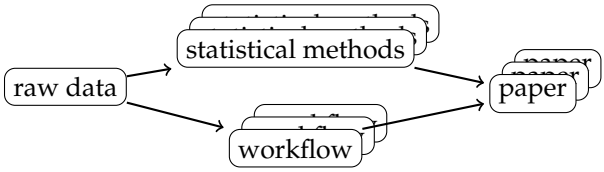
- Tangle (Stangle, knit(..., tangle=TRUE)): `foo.Rnw` → `foo.R`
- Weave (Sweave, knit): `foo.Rnw` → `foo.tex`
  (may contain parts of `foo.R`)

What does `Rnw` mean:

- `R` for the R project
- `nw` for noweb (web for no particular language, or Norman Ramsey's Web)

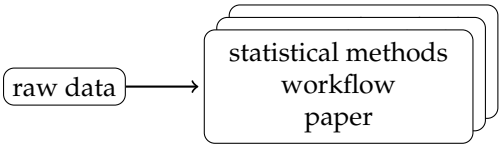
Nonliterate versus literate work

Nonliterate:



Remember: it is easy to confuse the different version of the analysis and their relation to the versions of the paper.

Literate:



With literatate programming in the analysis we avoid one important source of errors: Confusion about which parts of our work do belong together and which do not.

Advantages of literate programming

- Methods are clearly connected with the paper (no more: 'which version of the methods were used for which figure, which table')
- The paper is dynamic
  - More raw data arrives: the new version of the paper writes itself
  - You organise and clean the data differently: the new version of the paper writes itself
  - You change a detail of the method which has implications for the rest of the paper: the new version of the paper writes itself

7.3 An example

Here is a brief *Rnw*-document:

```
\documentclass{article}
\begin{document}
    text that explains what you are doing and why it is
                                                interesting ...
```

```
<<someCalculations,results='asis',echo=FALSE>>=
library(Ecdat)
library(xtable)
library(lattice)
data(Caschool)
attach(Caschool)
est <- lm(testscr ~  avginc)
print(xtable(anova(est)),floating=FALSE)
@
```

```
<<aFigure,echo=FALSE,fig.width=4,fig.height=3>>=
plot(xyplot(testscr ~ avginc,xlab="average income",ylab="testscore",
    type=c("p","r","smooth")))
@
```

```
the correlation between average income and testscore is
\Sexpr{round(cor(testscr,avginc),4)}
more text ...
\end{document}
```

To compile this *Rnw*-file, we can do the following:

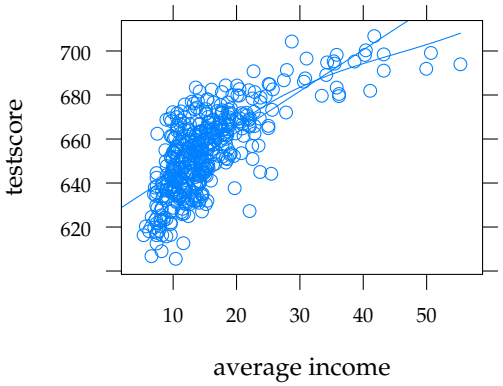
```
library(knitr)
knit("<filename.Rnw>")
system("pdflatex <filename.tex>")
```

... or use a front end like RStudio.

The result, after knitting:

text that explains what you are doing and why it is interesting ...

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
avginc	1	77204.39	77204.39	430.83	0.0000
Residuals	418	74905.20	179.20		



the correlation between average income and testscores is 0.7124.
more text ...

7.4 Text chunks

What we saw:

- The usual LaTeX-text
- "chunks" like this

```
<<>>=
lm(testscr ~  avginc)
@
```

  or "chunks" with parameters:

```
<<fig.height=2.5>>=
plot(est,which=1)
@
```

  more generally

```
<<...parameters...>>=
...R-commands...
@
```

What are these parameters:

- `<<anyName,...>>=`

  not necessary, but identifies the chunk. Also helps recycling chunks, e.g. a figure.

- `<<...,eval=FALSE,...>>=`

  this chunk will not be evaluated

- `<<...,echo=FALSE,...>>=`

  the code of this chunk will not be shown

- `<<...,fig.width=3,fig.height=3,...>>=`

  the chunk will produce a figure of a given width and height (in inches) which should be inserted here.

- `<<...,results='asis',...>>=`

  the chunk produces LaTeX-output which should be inserted here

Furthermore you can include small parts of output in the text:
`\Sexpr{...}`

Elements of a knitr-document

```
\documentclass{article}
\begin{document}
<<>>=
opts_chunk[["set"]](dev='tikz', external=FALSE, fig.width=4.5,
                    fig.height=3, echo=TRUE, warning=TRUE,
                    error=TRUE, message=TRUE,
                    cache=TRUE, autodep=TRUE,
                    size="footnotesize")
@
\usepackage{tikz}
```

- *dev='tikz',external=FALSE* sets the format for plots

  Since tikz is at the moment not part of the standard R packages, you have to install with *install.packages("tikzDevice", repos="http://R-Forge.R-project.org")* This works well on Unix based systems. On a Microsoft Windows system you may need the Windows toolset for R which is not part of the standard distribution.

- *fig.width=4.5,fig.height=3* the the size for plots

- *echo=TRUE, warning=TRUE, error=TRUE, message=TRUE* what kind of output is shown
- *cache=TRUE, autodep=TRUE* do calculate chunks only when they have changed
- *size="footnotesize"* size of the output

All these values can be overridden for specific knitr chunks.

Words of caution There is still something that might break:

In case something in R changes in the future, better put somewhere in your document:

```
This document has been generated on \today, with
\Sexpr{version$version.string}, on
\Sexpr{version$platform}.
```

This document has been generated on 18th July 2013, with R version 3.0.1 (2013-05-16), on x86_64-pc-linux-gnu

7.5 Advantages

- Accuracy (no more mistakes from copying and pasting)
- Reproducability (even years later, it is always clear how results were generated)
- Dynamic document (changes are immediately reflected everywhere, this speeds up the writing process)

7.6 Practical issues

What if some calculations take too much time Usually you will not be able (or willing) to do the entire journey from your raw data to the paper in one single step.

The typical workflow is rather

1. raw data: *long list of files*
2. cleaning and preparing the data

   myProjectPrepare_130715.Rnw generates *myProjectClean_130715.Rdata*

   This step can be expensive (takes a lot of computing time)
3. presenting the results in a paper or in slides

   myProjectPrepare_130605.Rnw

   In the paper you have a line

   ```
   load(myProjectPaper_130715.Rdata)
   ```

   hence, you know what data you use and your result is reproducible
4. The condition is, of course, that, once public, you never ever change *myProjectPrepare_130715.Rnw* or *myProjectClean_130715.Rdata*.

Alternatively: caching intermediate results knitr can also cache intermediate results:

```
<<expensiveStep,cache=TRUE>>=
intermediateResults <- ....
@
```

The above chunk is executed only once (unless it changes), results are stored on disk and can be used lateron.

7.7 When R produces tables

7.7.1 Tables

You can save a lot of work if you harness R to create and format your tables for you. A versatile function is *xtable*:

```
(x <- rbind(c(1,2,3),c(4,5,6)))

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

```
<<results='asis'>>
```

```
library(xtable)
print(xtable(x),floating=FALSE)
```

	1	2	3
1	1.00	2.00	3.00
2	4.00	5.00	6.00

```
@
```

7.7.2 Estimation results

Estimation results in tabular form from *mtable* are typeset by *toLatex*:

```
library(Ecdat)
data(Caschool)
est1 <- lm(testscr ~ str,data=Caschool)
est2 <- lm(testscr ~ str + elpct,data=Caschool)
est3 <- lm(testscr ~ str + elpct +avginc,data=Caschool)
toLatex(mtable(est1,est2,est3))
```

	est1	est2	est3
(Intercept)	698.933***	686.032***	640.315***
	(9.467)	(7.411)	(5.775)
str	−2.280***	−1.101**	−0.069
	(0.480)	(0.380)	(0.277)
elpct		−0.650***	−0.488***
		(0.039)	(0.029)
avginc			1.495***
			(0.075)
R-squared	0.051	0.426	0.707
adj. R-squared	0.049	0.424	0.705
sigma	18.581	14.464	10.347
F	22.575	155.014	334.889
p	0.000	0.000	0.000
Log-likelihood	−1822.250	−1716.561	−1575.374
Deviance	144315.484	87245.293	44540.732
AIC	3650.499	3441.123	3160.748
BIC	3662.620	3457.284	3180.950
N	420	420	420

Nicer names for variables and equations

```
toLatex(relabel(mtable("simple"=est1,"intermediate"=est2,
   "final"=est3),c(str="student/teacher",
       elpct="English learners",avginc="average income",
       "(Intercept)"="Constant")))
```

	simple	intermediate	final
Constant	698.933***	686.032***	640.315***
	(9.467)	(7.411)	(5.775)
student/teacher	−2.280***	−1.101**	−0.069
	(0.480)	(0.380)	(0.277)
English learners		−0.650***	−0.488***
		(0.039)	(0.029)
average income			1.495***
			(0.075)
R-squared	0.051	0.426	0.707
adj. R-squared	0.049	0.424	0.705
sigma	18.581	14.464	10.347
F	22.575	155.014	334.889
p	0.000	0.000	0.000
Log-likelihood	−1822.250	−1716.561	−1575.374
Deviance	144315.484	87245.293	44540.732
AIC	3650.499	3441.123	3160.748
BIC	3662.620	3457.284	3180.950
N	420	420	420

Requirements The default of *toLatex* assumes the *dcolumn* package, i.e. in the preamble of the document we have to say something like:

```
\usepackage{dcolumn}
\let\toprule\hline
\let\midrule\hline
\let\bottomrule\hline
```

7.7.3 Mixed effects

If we use *lmer* to estimte models with mixed effects, *toLatex* needs a *summary.mer* method. The following is one example:

```
bootSize <- 1000
getSummary.mer <- function(mer) {
  msd <- sqrt(diag(vcov(mer)))
  coefs <- fixef(mer)
  mz<-mcmcsamp(mer,bootSize)
  mf <- mz@fixef
  mzp <- 2*pnorm(-abs(mzt <- (mzcoef <- apply(mf,1,mean))/
                                    (mzsd <- apply(mf,1,sd))))
  mzci <- cbind(coefs) %*% c(1,1) + cbind(mzsd) %*%
                                     rbind(qnorm(c(.025,.975)))
  coef <- cbind(coefs,mzsd,mzt,mzp,mzci)
  colnames(coef) <- c("est", "se", "stat", "p", "lwr", "upr")
  smer<-summary(mer)
  AIC <- smer@AICtab$AIC
  BIC <- smer@AICtab$BIC
  logLik <- smer@AICtab$logLik
  deviance <- smer@AICtab$deviance
  REMSdev <- smer@AICtab$REMSdev
  N <- length(mer@resid)
  # below we assume two random effects: one for the independent
  #                     observations and one for the participants
  # this is frequently the case for experiments but need not
  #                              always be the case for other mer-s
  ngrps<-min(smer@ngrps)
  mgrps<-max(smer@ngrps)
  sumstat <- c(deviance=deviance,AIC=AIC,BIC=BIC,
         logLik=logLik,N=N,ngrps=ngrps,mgrps=mgrps)
  list(coef=coef,sumstat=sumstat,call = mer@call)
}
setSummaryTemplate(mer=c("Log-likelihood" = "($logLik:f#)",
                 Deviance      = "($deviance:f#)",
                 AIC           = "($AIC:f#)",
                 BIC           = "($BIC:f#)",
                 N             = "($N:d)",
                 "indep.obs."="($ngrps:d)",
                 "participants"="($mgrps:d)"))
setCoefTemplate(pci=c(est="($est:#)($p:*)",
                      ci="[($lwr:#);($upr:#)]"))
```

We should note that our definition of indep.obs. and participants as the smallest and largest number of groups, respectively, is often reasonable if we have, indeed, two random effects, one for inde-

pendent observations, the other for participants. This is frequently the case for experiments but need not always be the case for other mixed effects models.

We should also note that there are several ways to bootstrap p-values. In the example we use *mcmcsamp* and we assume that the distribution of coefficients follows a normal distribution.

7.8 The magic of make

In the same directory where I have my Rnw file, I also have a file that is called *Makefile*. Let us assume that the current version of my Rnw file is called *myProject_130601.Rnw*. Then here is my *Makefile*

```
PROJECT = myProject_130601

pdf: $(PROJECT).pdf

%.pdf: %.tex
     pdflatex $<

%.tex: %.Rnw
     echo "library(knitr);knit(\"$<\");" | R --vanilla
```

Let us go through the individual lines of this Makefile.

```
PROJECT = myProject_130601
```

Here we define a variable. This is useful, since this most of the time the only line of the Makefile I ever have to change (instead of changing every occurence of the filename)

```
pdf: $(PROJECT).pdf
```

The part *pdf* before the colon is a target. Since it is the first target in the file it is also the default target. I.e. make will try to make it whenever I just say

```
make
```

Make will do the same when I call it explicitely

```
make pdf
```

The part after the colon tells make on which file(s) the target actually depends (the prerequisites). Here it is only one but there could be several. If all prerequisites exists, and if they are up-to-date (newer than all files they depends on), make will apply the rule. Otherwise, make will try to create the prerequisites (the *pdf* file in this case, with the help of other rules) and then apply this rule.

```
%.tex: %.Rnw
     echo "library(knitr);knit(\"$<\");" | R --vanilla
```

This is a rule that make can use to create *tex* files. So above we requested the *pdf* file *myProject_130601.pdf*, and now make knows that we require a file *myProject_130601.tex*. If this already exists and is up-to-date (i.e. newer than all files it depends on), make will apply this rule. Otherwise, make will first try to create the prerequisite (the single *tex* file in this case would be created with the help of other rules) and then apply this rule.

To create our *pdf* it is now sufficient to say (from the command line, not from R)

```
make
```

and make will do everything that is needed.

Note: In this context a simple shell script would work almost as well. However, *make* is very helpful when your pdf file depends on more than one *tex* or *Rnw* file.

A Makefile for a larger project When I wrote this handout I split it into several Rnw files. This saves time. When I make changes to one part, only this part has to be compiled again. The files were all in the same directory. The directory also contained a "master"-tex file that would assemble the tex-files for each Rnw-file.

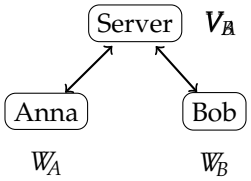
The following example shows how we assemble the output of several files to make one document:

```
PROJECT = myProject_130601
RPARTS        = $(wildcard $(PROJECT)_[1-9].Rnw)
TEXPARTS      = $(RPARTS:.Rnw=.tex)

pdf: $(PROJECT).pdf

# our project depends on several files:
$(PROJECT).pdf: $(TEXPARTS) $(PROJECT).tex
        pdflatex $(PROJECT)

# only the tex files who belong to Rnw files
#                               should be knitted:
$(TEXPARTS)   : %.tex  : %.Rnw ; \
        echo "library(knitr);knit(\"$<\");" | R --vanilla
```

8 Version control

8.1 Problem I – concurrent edits

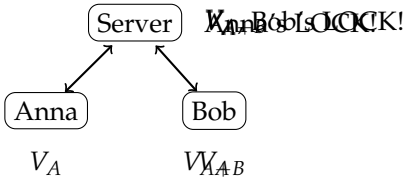
What happens if two authors, Anna and Bob, simultaneously want to work on the same file. Chances are that one is deleting the changes of the other. (This problem is similar to one author working on two different machines)



- Anna's work is lost — very inefficient (50% of the contribution is lost)

8.2 A "simple" solution: locking

Serialising the workflow might help. Anna could put a "lock" on a file while she wants to edit this file. Only when she is finished, the "unlocks" the file and Bob can continue.

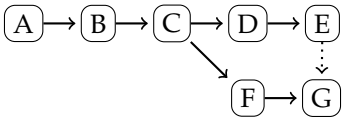


- Bob can only work with Anna's permission — very inefficient (50% of the time Anna and Bob are forced to wait)

8.3 Problem II – nonlinear work

Even when Anna works on a problem on her own she can be in conflict with herself. Imagine the following: Anna successfully completed the steps A, B, and C on a paper and has now something readable that she could send around. Perhaps she actually has sent it around. Now she continues to work on some technical details D and E, but so far her work in incomplete – D and E are not ready for the public. Suddenly the need arises to go back to the last public

version (C) and to add some work there (e.g. Anna decides to submit the paper to a conference, but wants to rewrite the introduction and the conclusion. It will take too much time to first finish the work on D and E, so she has to go back to C. Rewriting the introduction and conclusion are steps F and G. Once the paper (G) has been submitted, Anna wants to return to the technical bits D and E and merge them with F and G.



8.4 Solution to problem II: nonlinear work

Before we create our first git-repository, we have to provide some basic information about ourselves:

```
git config --global user.name "Your Name Comes Here"
git config --global user.email you@yourdomain.example.com
```

Now we can create our first repository:

```
git init
```

We can check the current "status" as follows:

```
git status
```

git status

```
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```

now we create a file *test.Rnw*

git status

```
# On branch master
#
# Initial commit
#
```

```
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#        test.Rnw
nothing added to commit but untracked files present (use "git add" to track)
```

```
git add test.Rnw
```

git status

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#        new file:   test.Rnw
```

```
git commit -a -m "first version of test.Rnw"
```

git status

```
# On branch master
nothing to commit, working directory clean
```

```
git log --oneline
```

git log -oneline

```
3ea6194 first version of test.Rnw
```

Note that git denotes versions with identifiers like “3ea6194” (and not A, B, C).

After some changes to *test.Rnw*...

git status

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#        modified:   test.Rnw
#
no changes added to commit (use "git add" and/or "git commit -a")
```

```
git commit -a -m "introduction and first results"
```

git status

```
# On branch master
nothing to commit, working directory clean
```

git log -oneline

```
74fd521 introduction and first results
3ea6194 first version of test.Rnw
```

More changes and…

```
git commit -a -m "draft conclusion"
```

more changes and…

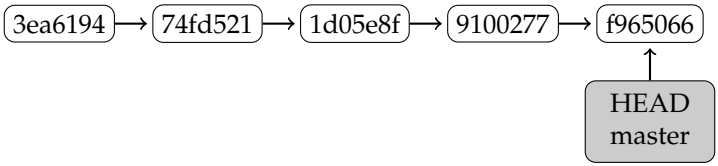
```
git commit -a -m "improved regression results (do not fully work)"
```

more changes and…

```
git commit -a -m "added funny model (does not fully work yet)"
```

git log -oneline

```
f965066 added funny model (does not fully work yet)
9100277 improved regression results (do not fully work)
1d05e8f draft conclusion
74fd521 introduction and first results
3ea6194 first version of test.Rnw
```



Assume we want to go back to *1d05e8f* but not forget what we did between *1d05e8f* and *f965066*.

Remember current state:

```
git branch funny
```

Now that we have given the current branch a name we can revert to the old state:

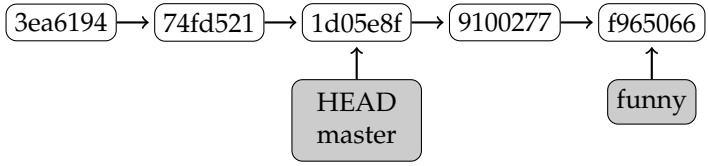
```
git reset 1d05e8f
```

```
Unstaged changes after reset:
M       test.Rnw
```

```
git checkout test.Rnw
```

git status

```
# On branch master
nothing to commit, working directory clean
```

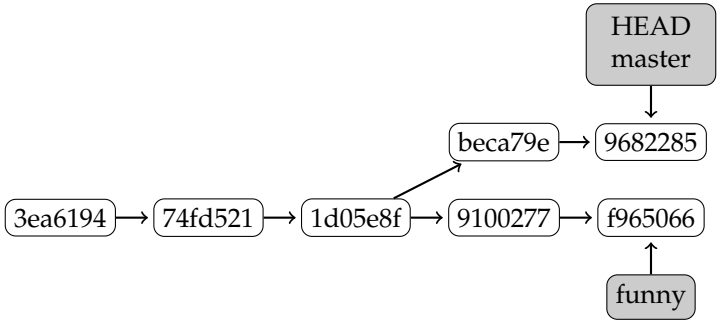


do more work…

```
git commit -a -m "rewrote introduction"
```

do even more work…

```
git commit -a -m "rewrote conclusion, added literature"
```



eventuelly we want to join the two branches:

```
git merge funny
```

now two things can happen: Either this…

```
Merge made by recursive.
 test.Rnw |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

or that...

```
Auto-merging test.Rnw
CONFLICT (content): Merge conflict in test.Rnw
Automatic merge failed; fix conflicts and then commit the result
```

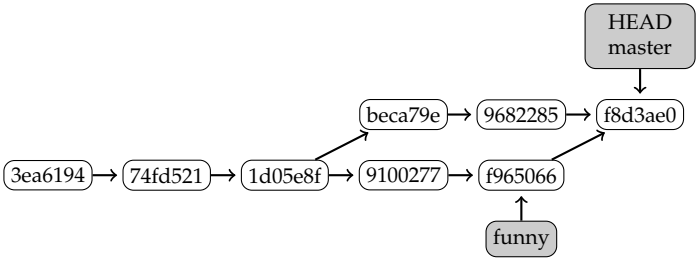
We can fix this with *git mergetool*:

```
git mergetool
```

```
  Merging:
test.Rnw

Normal merge conflict for 'test.Rnw':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (kdiff3):
```

Now we can make detailed merge decisions in an editor.

```
git commit -m "merged funny"
```



8.5 Solution to problem I: concurrent edits

Version control allows all authors to work on the file(s) simultaneously.

In this example we start with an empty repository. In a first step both Anna and Bob "checkout" the repository, i.e. they create a local copy of the repository on their computer.

Anna creates a file, adds it to version control and commits it to the repository. Bob then updates his copy and, thus, obtains Anna's changes.

- First step: create a "bare" repository on a "server"

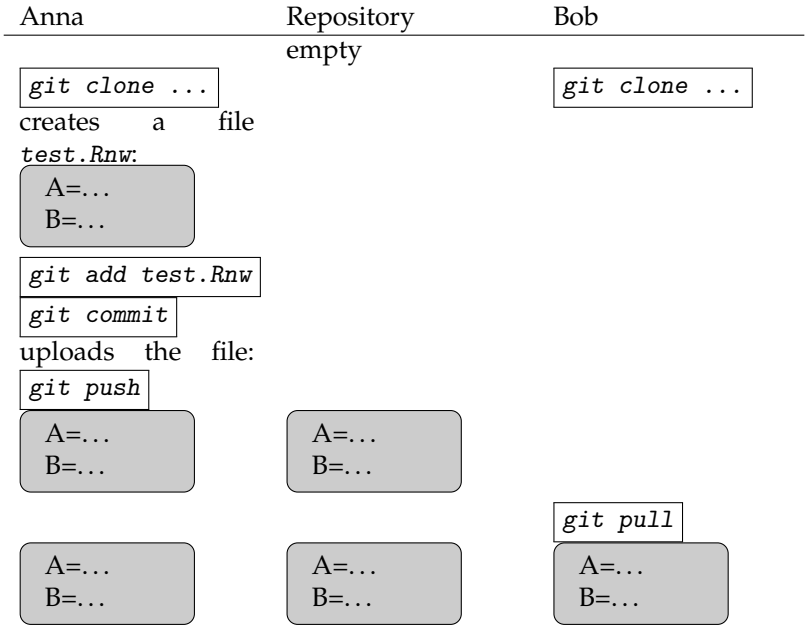
```
git --bare init
```

- This repository can now be accessed from "clients", either on the same machine...

```
git clone /path/to/repository/
```

...or on a different machine via *ssh* (where *user* has access rights):

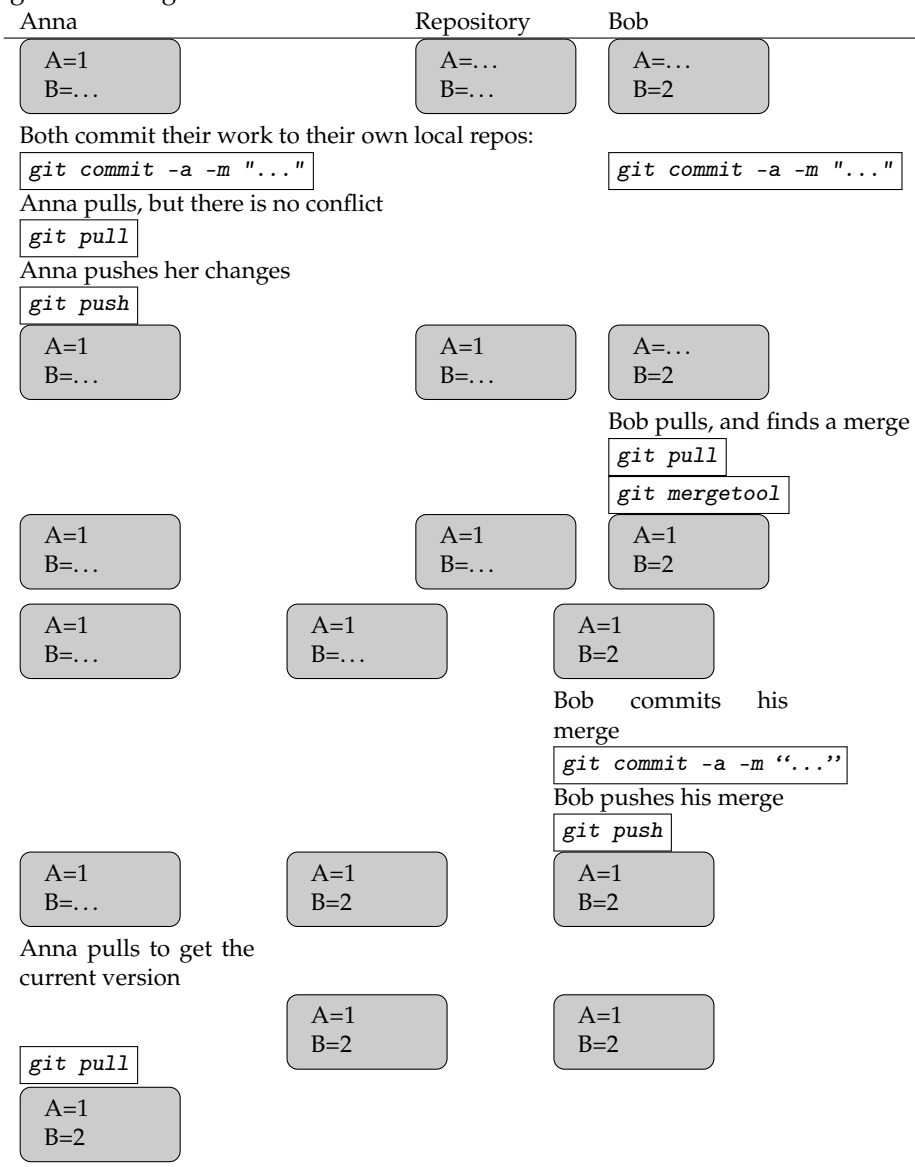
```
git clone ssh://user@my.server.org/path/to/repository
```



8.6 Edits without conflicts:

To make this more interesting we now assume that both work on the file. Anna works on the upper part (A), Bob works on the lower part (B). Both update and commit their changes. Since they both edit different parts of the file, the version control system can silently

merge their changes.



8.7 Going back in time

Version control is not only helpful to avoid conflicts between several people, it also helps when we change our mind and want to have a look into the past. `git log` provides a list of the different revision of a file:

```
git log -oneline
```

```
f965066 added funny model (does not fully work yet)
9100277 improved regression results (do not fully work)
1d05e8f draft conclusion
74fd521 introduction and first results
3ea6194 first version of test.Rnw
```

`git blame` allows you to inspect modifications in specific files. If we want to find out who introduced or removed "something specific" (and when), we would say...

```
git blame -L '/something specific/' test.Rnw
```

```
19eb9bac (w6kiol2 2013-06-17 ...) therefore important to study something specific wh
dd0647f7 (w6kiol2 2013-06-21 ...) switched our focus to something else and continue
```

There is a range of GUIs that allow you to browse the commit tree.

Try, e.g., `gitk`

8.8 git and subversion

- git-Server: requires ssh access to the server machine
- subversion-Server: provided by the URZ at the FSU Jena

  git can use subversion as a remote repository:

git clone	git svn clone
git pull	git svn rebase
git push	git svn dcommit

- Conceptual differences:
  - subversion has only one repository (on the server), git has one or more local repositories plus one or more on different servers.
  - inconsistent uploads to a server:
    subversion will not complain if after a `push/commit` the state on the server is different from the state on any of the clients. git will not allow this (git forces you to pull first, merge, commit, and push then)

8.9 Steps to set up a subversion repository at the URZ at the FSU Jena

If you need to set up a subversion repository here at the FSU, tell me about it and tell me the ⟨*urz-login*⟩s of the people who plan to use it. Technically, setting up a new repository means the following:

- ssh to subversion.rz.uni-jena.de
- `svnadmin create /data/svn/ewf/`⟨*repository*⟩
- `chmod -R g+w /data/svn/ewf/`⟨*repository*⟩
- set access rights for all involved ⟨*urz-login*⟩s in `/svn/access-ewf`
- then, at the local machine in a directory that actually contains only the files you want to add: `svn -username` ⟨*urz-login*⟩ `import . https://subversion.rz.uni-jena.de/svn/ewf/`⟨*repository*⟩`-m "Initial import"`

  (this "imports" data into the repository)
- then, at all client machines,

  `svn -username` ⟨*urz-login*⟩ `checkout https://subversion.rz.uni-jena.`

8.10 Setting up a subversion repository on your own computer

- On your own computer: `svnadmin create` ⟨*path*⟩`/`⟨*repository*⟩
  (⟨*path*⟩is a complete path, e.g. `/home/user/Documents/` or `/C:MyDocuments/`)
- then, in a directory that actually contains only the files you want to add:
  `svn import . file://`⟨*path*⟩`/`⟨*repository*⟩ `-m "Initial import"`
- then, wherever you actually want to work on your own computer:
  `svn checkout file://`⟨*path*⟩`/`⟨*repository*⟩
- if you have `ssh` access to your computer you can also say from other machines:
  `svn checkout svn+ssh://`⟨*yourComputer*⟩`/`⟨*path*⟩`/`⟨*repository*⟩

8.11 Usual workflow with git

While setting up a repository looks a bit complicated, using it is quite simple:

- `git pull` check whether the others did something
- editing
  - `git add` add a file to version control
  - `git mv` move a file under version control
  - `git rm` delete a file under version control
- `git commit` commit own work to local repository

- `git pull` check whether the others did something
- `git mergetool` merge their changes
- `git commit` commit merge
- `git push` upload everything to the server

8.12 Exercise

Create (in ⟨*path*⟩) four directories `A`, `B`, `C`.

From `A` create a repository: `svnadmin create ../R`

In `A` create a file `test.txt` with some text:

```
A=...
B=...
```

Initial import. In `A` say:

```
svn import .  file://⟨path⟩/R -m "My first initial import"
```

in `B`:	in `C`:
`svn checkout file://⟨path⟩/R`	`svn checkout file://⟨path⟩/R`
in `B/R`:	in `C/R`:
Simultaneous changes to `test.txt`	
`A=1` `B=...`	`A=...` `B=2`
Commit changes	
`svn commit`	`svn commit`
Update	
`svn update`	`svn update`

9 Exercises

Exercise 1

Have a look at the dataset `Workinghours` from the library `Ecdat`. Compare the distribution of "other household income" for whites and non-whites. Do the same for the different types of occupation of the husband.

Exercise 2
Read the data from a hypothetical experiment from `rawdata/Coordination`. Does the `Effort` change over time?

Exercise 3-a
Read the data from a hypothetical z-Tree experiment from `rawdata/Trust`. Do you find any relation between the number of siblings and trust?

Exercise 3-b
For the same dataset: Attach a label (description) to `siblings`. Attach value labels to this variable.

Exercise 3-c
Make the above a function.

Also write a function that compares the offers of all participant with n siblings with the other offers. This function should (at least) return a p-value of a two-sample Wilcoxon test (`wilcox.test`). The number n should be a parameter of the function.

Exercise 4
Read the data from a hypothetical z-Tree experiment from `rawdata/PublicGood`. The three variables `Contrib1`, `Contrib2`, and `Contrib3` are contributions of the participants to the other three players in their group (in gruops of four).

1. Check that, indeed, in each period, players are equally distributed into four groups.

2. Produce for each period a boxplot with the contribution (i.e. 16 boxplots in one graph).

3. Add a regression line to the graph.

4. Produce for each contribution partner a boxplot with the contribution (i.e. 3 boxplots in one graph).

5. Produce an `Sweave` file that generates the two graphs. In this file also write when you estimate the average contribution reaches zero.