Data Analysis using R

Dr. Detlef Steuer
Helmut-Schmidt-Universität Hamburg
steuer@hsu-hh.de
http://fawn.unibw-hamburg.de/∼steuer

IRWS Akademie Sankelmark, 4 - 9 October 2009

HELMUT SCHMIDT
UNIVERSITÄT
Universität der Bundeswehr Hamburg

Aim of this course

- Give you a tool to work on your thesis. If your work requires data analysis, you know one tool.

- Understanding of underlying concepts of R.

- How to get data in and reports out of R.

Way of learning

- No traditional lecture, more of a hands-on tutorial.
  (there's no other way to learn a programming language)
- Course tries to be "self-contained".
- Practical exercises.
- Any questions, any time.

What is R?

(Citations taken from www.r-project.org)

- R is a language and environment for statistical computing and graphics.
- R provides a wide variety of statistical and graphical techniques, and is highly extensible.
- One of R's strengths is the ease with which well-designed publication-quality plots can be produced, including mathematical symbols and formulae.
- Many users think of R as a statistics system. We prefer to think of it as an environment within which statistical techniques are implemented.

Why R? (Infrastructure)

- Open Source Software done right.
- Platform independent (running on MacOS, Windows (starting from Win 95), Linux (development platform), proprietary unixes (IRIX/SunOS etc.), even on some PDAs (Sharp Zaurus).
- Free, but better-than-money-can-buy support on the mailing list.
- Professionally organized quality control ('make check').
- Exceptional documentation (Reference Manual around 1500 pages included).

Why R? (User side)

- Easily expendable (so called packages), very quick prototyping, because R works as an interpreter.
- Great graphics capabilities (mathematical symbols / reproducible plots).
- Integration with TEX resp. LATEX, more recently even Openoffice (sweave / odfweave).
- Full I18n (Interface; help pages remain untranslated for the moment).
- If needed C, C++ or FORTRAN modules can be used.

Origins of R

- "R is not completely unlike S".

- S is a programming language standard developed at Bell Laboratories starting in the 1980's. Lead developers were Rick Becker, John Chambers and Alan Wilks.

- In 1998 John Chambers received the "Software System Award" of the ACM (Association for Computing Machinery) for the design and implementation of S.

- From the laudatio: [...] the S system, which has forever altered the way people analyze, visualize, and manipulate data [...] S is an elegant, widely accepted, and enduring system, with conceptual integrity, thanks to the insight, taste and effort of John Chambers. [...]

(Very) short history of R

- Starting in the 1980's S existed for internal usage at Bell Labs.
- Since around 1990 there exists a commercial variation of S named Splus.
- In 1992 Ross Ihaka and Robert Gentleman created a first free implementation of the S language core for teaching purposes at the University of Auckland. (On Apple Macintosh!).
- In 1995 R became GPL software.
- In 1997 the so-called "Core-Team" is founded. Today 19 members from universities and corporations all over the world, including John Chambers.
- In 1998 CRAN gets installed.

(Very) short history of R

- In 2001 R news starts: Peer reviewed electronic journal devoted to R.
- In 2002 The R Foundation is founded.
- During 2007 R-forge was launched.
- January 2009, R gets a title page story in the NYT:
  http://www.nytimes.com/2009/01/07/technology/business-computing/07program.html
- In 2009 The R Journal was founded. Peer reviewed, incorporates R news.

(Very) short history of R

- Today: R is a (the?) standard tool in method development in statistics. But even in industry it gets a stronger foothold every day. Personally I know of applications at Vodafone, Lufthansa, Roche, and car industry. FDA looks at certifying R!

What R offers to its user

- R is an interpreter for the S language (esp. not compiled!).
- R is a consistent framework for data management, analysis and presentation.
- A huge collection ($\approx$ 2000) of packages adding functionality to
- thousands of predefined routines in R-base.
- Each of the packages on CRAN must pass a thorough quality check.

What R offers to its user

- Transparent network access.
- Very supportive mailing list.
- All data can be saved in a portable text representation.
- (S4 Classes: Programming with data. Object oriented data analysis. Not covered here.)

Installation

Before we start:

- You find R (for Windows and Mac) on the CD.
- Everybody should install R-2.9.2 now! (current version, R-2.10.0 to arrive on October, 26th)
- Linux users: go to cran.r-project.org, download, and install the appropriate version.
- Everybody set-up?

User Interfaces for R

- Built in CLI. This is a good thing! R is a perfect tool to implement reproducible research.
- There are some GUIs, but not part of standard installation
- Best practice for using R: Interfacing with an external editor (Emacs(!), WinEdt etc.)
- Recommendation for Windows: Tinn-R.
- Batch mode or R as server (with apache web server).
- Somewhat exotic: use R as shared library in other languages (Perl, Python, Java, C++ etc).

Start a first R-session

```
steuer@gaia:~> R

R version 2.9.2 (2009-08-24)
Copyright (C) 2009 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
```

Start a first R-session

```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

First steps: interactive use of R

- R as calculator

```
> 3 + 4
[1] 7

> log(0)
[1] -Inf

> log(-1)
[1] NaN
Warning message:
NaNs were generated in: log(x)
```

First steps: interactive use of R

```
> pi  #case sensitive
[1] 3.141593

> x <- .Last.value
# x = .Last.value is syntactically correct nowadays

> ls()
[1] x

> rm(x)
> q()
```

Working with packages

- Lots for additional functionality in external *packages* (or *bundles* or *views*)
- Adding to your R installation

```
install.packages("scatterplot3d")
```

- Loading into your session `library(scatterplot3d)` or `require(scatterplot3d)`
- Unloading `detach(package:scatterplot3d)`
- For views: `install.packages("ctv") ; library(ctv)` and use `install.views()` afterwards

Building own packages

- Beyond the scope of this course
- But: easy to start with and the way to go, if you know you'll implement new or special methods for your work.
- See: 'Writing R extensions' included in your installation

Mathematical operators

Symbol	Function
^ or **	Raise to power
*, /, +, -	Multiplication, division, addition, subtraction
%/%, %%	integer, resp. modulo division
%*%	Matrix multiplication

Of course there are the usual arithmetical functions like `round()`, `sin()`, `abs()`, `sqrt()` etc.

Funny: These are all ordinary functions:

```
> "+"(3,4)
[1] 7
```

Mathematical operators

- It's important to know the identifiers for 'special' numbers: NaN, Inf, -Inf, NULL, TRUE, FALSE, NA.
- All calculations involving NaN result in NaN!
- R implements IEEE arithmetic.
- Attention: `pi` is not `PI`! R is case sensitive!

Logical operators

- $==$ both objects are **identical**. There is `identical()` for complex objects.
- $!=$ not equal.
- $<$, $>$, $<=$, $>=$ less than, greater than (or equal).
- $\&$, $|$, ! (logical) AND, OR, NOT.

A snatch (or two)

```
> a <- 3
> b <- 2.1/0.7
> a == b
[1] FALSE
```

What's going on? Solution: function `all.equal()`

```
> all.equal(a, b)
[1] TRUE
```

`all.equal()` checks for numerical equality up to an ϵ
Default: `sqrt( .Machine.double.eps )`

A snatch (or two)

Naïvely one expects the following to work:

```
> a <- NA
> a == NA
[1] NA
> a <- NaN
> a == NaN
[1] NA
```

But it doesn't.
For these cases R provides

```
> a <- NA ; is.na(a)
> a <- NaN ; is.nan(a)
```

Basic statistics

Many, many functions built in.

- `mean()`, `var()`, `sd()`.
- `runif()`, `rnorm()` `etc.` Random number generation.
- `pnorm()`, `dnorm()` `etc.` Distribution, density etc.
- `fivenum()`, `range()`, `summary()`, `stem()` Tukey's numbers, range, stem-and-leaf plot.
- `boxplot()`, `piechart()`, `hist()`, `barplot()` basic plots.
- Important option `'na.rm'` handles NAs in calculations i.e. `mean(x, na.rm=TRUE)`, also global option `na.action`.

Appetizer data analysis

```
> data(iris)
> names(iris)
> str(iris)
> ?iris
> summary(iris)
> attach(iris)
> species.n <- as.numeric(Species)
> plot(iris, col=species.n)
> hist(Petal.Length)
> op <- par(mfrow=c(2,2))
> for (i in 1:4){boxplot(iris[,i] ~ Species, main = colnames(iris[i]))}
> par(op)
> library(rpart)
> (rpo <- rpart(Species ~ ., data=iris))
```

HELMUT SCHMIDT UNIVERSITÄT
Universität der Bundeswehr Hamburg

Data analysis (cont.)

```
> plot(rpo, margin = 0.1, branch = 0.5)
> text(rpo)
> library(MASS)
> (ldao <- lda(Species ~ .,data=iris))
> plot(ldao, abbrev = TRUE, col = species.n)
> detach(iris)
> ls()
```

Simulation

```
> set.seed(123)
> x <- rnorm(100)
> y <- x + rnorm(100 , sd = 0.5)
> plot(x, y)
> (lmo <- lm(y ~x))
> summary(lmo)
> abline(lmo , lwd=2)
> plot(lmo)
> hist(x, freq = FALSE)
> lines(density(x), lwd=2, col="blue")
> qqnorm(x)
> qqline(x, lwd=2 col="blue")
> q()
```

Getting help

- http://cran.r-project.org/doc/contrib/Short-refcard.pdf
  The cheat-sheet for R!
- `help` or „?“: equivalent to RTFM: try `help(plot)` or `?plot`
- If you don't know the exact command and `help` doesn't help: try `apropos()`, `find()` or `help.search()`
- If you don't understand the help page, try `example(`*command*`)` or `demo(`*command*`)`
- `help.start()` shows the documentation in your standard browser
- Most contributed packages contain a vignette, a short handbook in PDF format, command `vignette()`.

External help

- Documentation on CRAN: cran.r-project.org
  Wast amount of well written documentation! Installation handbook, reference manual, data-exchange, **FAQ** etc.
- Mailing list archives and search capabilities on CRAN found at http://cran.r-project.org/search.html
- `RSiteSearch()` gives access to resources on `r-project.org` from within your session.
- New package `sos` to even search in the source code!

Ultima ratio

- Ask on r-help. Respect the posting guide or you will get 'ripleyed'. A few thousand readers, more than a hundred mails a day. You will get an useful answer!
- If your problem gets solved, be nice and add your wisdom to the community knowledge in the R-wiki http://wiki.r-project.org/rwiki/doku.php

Bookkeeping

Always work on copies of your original data!

- `ls()` shows all objects in existence in your current session. (try `ls.str()` for more verbose info) These get saved in a file `.Rdata` in your current working directory if you answer 'y' after `q()`.
- Try `getwd()` and `setwd()`.
- Pro: automatic data saving!
- Contra: a binary format! Never use it as main storage of your additional data!
- See also `save()` and `save.image()`.

Bookkeeping

- Therefore it's natural to use a directory of its own for each project.
- A history of the most recent commands gets saved in .Rhistory.
- It is always wise to take the time to clean up your transscript and to comment it!

Choosing an user interface

- Bare R has a command line interface. At least it supports things like C-p or cursor keys for repeating the previous command etc.
- There is no general 'R GUI' but lots of add-ons to have something like a point and click interface.
- My recommendation: (X)Emacs+ESS, steep learning curve but well worth it.
- Choose one of the text editors with R support listed on CRAN.

Concepts of R

- R is a classical interpreter, working in a read-eval-print loop.
- Reads input line by line, may be out of a file! (see `source()`)
- Each evaluation works on one R expression (`eval(EXPR)`).
- Functions are just objects. User defined functions are easily added (lexical scoping!):

```
> twiceasmuch <- function(x) {invisible(2*x)}
> x<-2
> twiceasmuch(x)
# no output ! Often useful inside more complicated functions.)
> (twiceasmuch(x)) # same as print(twiceasmuch(x))
```

Atomic data types used in R

Data type	Example
NULL	NULL
logical	FALSE
numeric	3.14
complex	3+i
character	"Hello"
factor	Ford, GM, Mercedes

- For a number you have a mode and a type.
- Find out about the type with `is.logical()`, `is.numeric()` etc.
- Force conversion to a type with `as.factor()`, `as.numeric()` etc.

Combined data types in R

- R is list oriented. A *list* is **the** basic data type in R . All complex data types are built out of lists.
- Most important for calculations are `vectors`, which are indexed list of elements of same type.
- A vector consists of elements of same type. If elements differ in type when building up a vector, they are forced to agree! (automatic conversion)
- Elementary command `c()` (**`c`**`ombine`).
- Scalars are vectors of length `1`.

Working with vectors (creation)

```
> (x <- c(1, 3, 4.5))
[1] 1.0 3.0 4.5
> typeof(x)
[1] "double"
> (x <- c(1, x, 3))
[1] 1.0 1.0 3.0 4.5 3.0
> (x <- c( 1, 4, partthree="Hello"))
                  partthree
     "1"       "4"  "Hello"
> length(x) ### length of vector
[1] 3
> t(x)       ### transposing (vectors / matrices)
             partthree
[1,] "1" "4" "Hello"
```

Working with vectors (subsetting)

```
> (x <- c(2, 3, 5, 7, 11, 13, 17, 19, 23))
[1]  2  3  5  7 11 13 17 19 23
> x[1]              ### access using the index
[1] 2
> x[2:4]            ### accessing multiple elements
[1] 3 5 7
> x[-(2:4)]         ### except some elements
[1]  2 11 13 17 19 23
> x[-c(1, 7, 9)] ### set of elements may be scattered over indices
[1]  3  5  7 11 13 19
> x[]               ### same as x
[1]  2  3  5  7 11 13 17 19 23
```

Working with vectors (conditional subsetting)

```
> which(x < 10) ### index numbers, where x satisfies some condition
[1] 1 2 3 4
> x
[1]  2  3  5  7 11 13 17 19 23

> x > 10
[1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE

> sum(x>10) # number of vector elements fulfilling condition
5

> x [ x > 10 ] # indexing over logical
[1] 11 13 17 19 23
```

Generation of special vectors `seq()`, `rep()`

```
> 1:5; 5:1      ### distance 1
[1] 1 2 3 4 5
[1] 5 4 3 2 1
> seq(1,5,2/3) ### chosen distance != 1
[1] 1.000000 1.666667 2.333333 3.000000 3.666667 4.333333 5.000000
> seq(along=x) ### numbering of vector x
[1] 1 2 3 4 5 6 7 8 9
> rep(TRUE, 5)
[1] TRUE TRUE TRUE TRUE TRUE
> rep(c("red","blue"),c(4,7))
 [1] "red"  "red"  "red"  "red"  "blue" "blue" "blue" "blue"
 [9] "blue" "blue" "blue"
```

Calculations with R-vectors

```
> (x <- seq(3,7))
[1] 3 4 5 6 7
> 1+x ### recycling of 1
[1] 4 5 6 7 8
> 2*x ### elementwise result
[1]  6  8 10 12 14
> x*x ### elementwise multiplication of vectors
[1]  9 16 25 36 49
> x%*%x  ### scalarproduct, implicite transposition!
     [,1]
[1,]  135
```

HELMUT SCHMIDT UNIVERSITÄT
Universität der Bundeswehr Hamburg

Recycling of vectors

```
> 6:1 - 1:3 ### shorter vector recycled
[1]  5  3  1  2  0 -2
> 7:1 - 1:3 ### use only if sure!
[1]  6  4  2  3  1 -1  0
Warning message:
longer object length
is not a multiple of shorter object length in: 7:1 - 1:3
```

Matrices I

- Generate a matrix with `matrix()`
  Ex.: `matrix(1:9,nrow=3, byrow=TRUE)`
- Important functions for Matrices: `%*%`, `crossprod()`, `solve()`, `diag()`, `eigen()`, `qr()`, `svd()`, `t()`
- A trap: beware of losing dimensions!

```
> x <- matrix(1:9,nrow=3)
> x[1,1]
[1] 1
> x[1,1, drop=FALSE]
     [,1]
[1,]    1
```

HELMUT SCHMIDT UNIVERSITÄT
Universität der Bundeswehr Hamburg

Matrices II and arrays

- Two indices describe the position of an matrix element. x[i,j] is in i-th row and j-th column. Wild cards just like the vector case!
  Ex.: x[1,] , x[2:4, 5:7]

- General arrays are created using `array(data, dim=c(...))`:

```
> partof <- array(1:12, dim=c(3, 2, 2))
> partof[1,,]
     [,1] [,2]
[1,]    1    7
[2,]    4   10
```

- Otherwise like matrices

Lists

- Combining different data type in a single object:

```
> (LL <- list(ab="hi", numbers=c(2, 4, 5),
                               xx=matrix(1:4,2)))
$ab
[1] "hi"
$zahlen
[1] 2 4 5
$xx
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

- Access list elements using index `LL`$[[i]]$ or name `LL$ab`.

HELMUT SCHMIDT UNIVERSITÄT
Universität der Bundeswehr Hamburg

Data frames (!)

- Data frames formally are lists, that may only contain vectors of identical length. Easy to visualize: rectangular scheme, each column contains one variable, each row contains one observation.
- **The** data structure in R!
- Access over index or name like lists and matrices.
- Some 'comfort' functions `attach()`, `detach()`, `with()`.
- Important data handling functions: `subset`, `%in%`, `split()`, `merge()`.

Example data-frame functions

```
data(iris) # read dataset iris
str(iris) # explore structure of iris
splitted <- split(iris, iris$Species)
attach(iris) # add variables inside iris to the search path
partofiris <- subset(iris, Species %in%
            c("virginica","versicolor") & Sepal.Length > 6)
give.me.iris.back <- unsplit(splitted, iris$Species)
iris.one.more <- cbind(iris, number=seq(along=iris$Species))
# there is rbind(), too
str(iris.one.more)
iris.one.less <- subset(iris.one.more, select=1:5)
# selecing/removing colums from a data frame
detach(iris)
iris[30:40,] # for choosing data rows
with(iris, table(Species))
```

Flow control I

- `if ()`
  Syntax: `if` (*condition*) { *command block*} [`else` { *command block* }]
  - curly braces necessary only if command block contains more than one command
  - `else` is optional
  - `if` not vector valued!

- `ifelse`()
  Syntax: `ifelse(`*`condition`*`,` *`TRUE - results`*`,` *`FALSE - results`*`)`
  - vector valued!

- `switch(EXPR, list of commands , ...)`
  - very general conditional execution command
  - see example or `?switch`

Examples if/ifelse

```
> x <- c( 7, 9)
> if (x > 8) {cat (x)} # only first element is used
> x <- 5
> if ( x == 5) {x <- x + 1 ; y<-3} else y <- 7
> x ; y
> ifelse (x ==  c(5, 6), c("A1", "A2"), c("A3", "A4"))
> ?switch
> switch(2, TRUE, FALSE)
> switch("five", a=1, five=5)
```

Exercise 1:

Exercise 1: Reinvent a simplified `if` using `switch` !
Return TRUE, if a given expression is fulfilled, FALSE otherwise.

Exercises 1:

Exercise 1: Reinvent a simplified `if` using `switch` !
Solution:

```
> x<-5 ; switch((x==5)+1,FALSE,TRUE)
[1] TRUE
> x<-6 ; switch((x==5)+1,FALSE,TRUE)
[1] FALSE
>  ownif <- function(expr) {switch(eval(expr) + 1, FALSE, TRUE)}
> ownif(x==5)
[1] FALSE
```

Flow control II

- `while`
  Syntax: `while`(*condition*) { *command block* }
  - Loop as long and only start it if *condition* fulfilled.

- `repeat`
  Syntax: `repeat`{ *command block* }
  - repeat until loop explicitly terminated.

- `for`
  Syntax: `for (i in M)`{ *command block* }
  - Repeat the command block using all given values in M for i.

- `next` jumps immediately to begin of loop.

- `break` immediately terminates loop.

Flow control examples

Exercise 2 Standard exercise: use each loop construct to calculate the first 20 Fibonacci numbers
Remember: For Fibonacci numbers $F_i, i \in \mathbb{N}$ holds:

$$F_i = F_{i-1} + F_{i-2} \text{ and } F_1 = 1, F_2 = 1.$$

Solutions for Exercise 2

```
> fib <- c(1,1)
> i <- 3
> while (i < 21) {fib <- c(fib, fib[i-1] + fib[i-2])
                  i <- i+1 }
```

or

```
> repeat { fib <- c(fib, fib[i-1] + fib[i-2]) ; i <- i+1
if (i == 21) break}
```

or

```
> for (i in 3:20) fib <- c(fib, fib[i-1] + fib[i-2])
```

or

```
> fib <- function(n)
          if ((n==1) | (n==2)) 1 else fib(n-1) + fib(n-2)
```

Remarks to exercise 2

- Solutions lack efficiency, because lots of internal copies of `fib` are generated. It is much better to allocate enough memory (if possible) beforehand

```
> fib <- c(1,1) ; i <- 3
> system.time(for (i in 3:10000)
            fib <- c(fib, fib[i-1] + fib[i-2]))
[1] 2.38 0.02 2.42 0.00 0.00
```

but

```
> fib <- rep(1,10000); i <- 3
> system.time(for (i in 3:10000)
            fib[i] <- fib[i-1] + fib[i-2])
[1] 0.30 0.00 0.35 0.00 0.00
```

No recursion, please!

```
> fib <- function(n)
          if ((n==1) | (n==2)) 1 else fib(n-1) + fib(n-2)
> runtime <- 1:10
> for (i in seq(3,30,3)) {
     cat(i, " of 30 \n")
     runtime[i%/%3] <- system.time(fib(i))[1]
}
> plot(runtime[1:10], t="l")
```

Very bad beheaviour in recursions, because lots and lots of ever increasing internal copies of the evaluation stack get created. If possible avoid recursion in R!

Advanced alternative looping techniques

R is vector based to avoid explicit loops. Faster alternatives working on vectors and lists:

- `apply`, `mapply`, `lapply`, `sapply`, `tapply`
- `replicate`

Best practice: Try your methodological idea using easily verifiable structure. Explicit loops are not bad in that stage! If it appears a) to be correct and b) to be time critical in context try to reprogram avoiding explicit loops!

The `apply` family of functions

- There is a whole series of specialized functions working efficiently on vectors.
  - %*% vector- or matrix multiplication
  - %o% resp. `outer()` outer product (useful for working on grids)
  - %x% resp. `kronecker()` Kronecker product
  - colSums(), rowSums(), colMeans(), rowMeans()
  - apply(), row- or column-wise application of specified functions on matrices or vectors
  - lapply(), apply function on data frame elements, lists
  - sapply(), like lapply, but result 'simplified'
  - tapply(), tables grouped by factors

t/l/s/m-apply() in detail I

- apply()
  Syntax: apply(X , MARGIN, FUN, ...)

```
> x <- matrix(1:25,nrow=5)
> apply(x,1,sum)
[1] 55 60 65 70 75
> apply(x,1, function(x) diff(range(x)))
[1] 20 20 20 20 20
> apply(x,1:2, sum)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

t/l/s/m-apply() in detail II

- lapply()
  Syntax: lapply(list, function, ...)

```
> data(anscombe)
> attach(anscombe)
> ans.reg <-vector(4, mode="list")
> for (i in 1:4) {
+ x <- get(paste("x", i, sep=""))
+ y <- get(paste("y",i,sep=""))
+ ans.reg[[i]]<-lm(y~x)
+ }
>  lapply(ans.reg,coef)
```

`t/l/s/m-apply()` in detail III

- `apply` and `lapply` return lists, `sapply` returns a vector if possible. (Often simpler to work on vectors of numbers than on lists)

- `replicate` is a special case of `sapply`:
  `hist(replicate(100, median(rnorm(100))))`

Exercise 3: Generate a list of five 3x3 matrices and calculate the sum of second column for each!

Solution for Exercise 3

```
> mliste <- list(5)
> for (i in 1:5) mliste[[i]]<- matrix(rnorm(9),nrow=3)
> lapply(mliste, function(x){sum(x[,2])})
[[1]]
[1] -0.4334959
[[2]]
[1] 2.263779
[[3]]
[1] 0.2048796
[[4]]
[1] -1.807335
[[5]]
[1] 0.05715369
> sapply(mliste, function(x){sum(x[,2])})
[1] -0.43349593  2.26377852  0.20487959 -1.80733452  0.05715369
```

t/l/s/m-apply() in detail IV

- mapply very special, s. ?mapply

- tapply
  Syntax: tapply(data, factor, function, ...)
  - In case you work on categorical variables. Instead of rows or columns you work on data grouped by a factor.

```
> data(iris) ; attach(iris)
> tapply(Sepal.Width, Species, range)
$setosa
[1] 2.3 4.4
$versicolor
[1] 2.0 3.4
$virginica
[1] 2.2 3.8
```

Saving and loading data I
R-objects

- `save()`: `save(data, file="filename")`
  Syntax: `save( ..., list, file,compress=FALSE, ascii=FALSE)`
  Ex.: `save(x, y, file="mydata.RData")`

- `q()`: q("yes"), q("no") or interactively
  Saves complete workspace in a file `.RData` in current working directory if "yes".

- `saveimage()`: equivalent to `save(list=ls(all=TRUE), file=".RData")`
  May be used to transport or archive different projects.

- Binary format! (even for ascii=TRUE!)

Saving and loading data II

- optional: compressed saving.
- `load(file)` loads a workspace image back into an R session. Does no clean up before loading!
- `dump()`: output of R objects in an human readable, editable, and machine independent format.
  Syntax: `dump(list,file="dumpdata.R", append=FALSE, ...)`.
- `dput()`: Special format, not safely restorable with source(). Avoid!
- `source()`: Reads files containing valid R expressions (read: programs).

The file argument may be a *connections* in many cases, i.e. output of other programs, pipes.

Saving and loading data III
Example session

```
set.seed(123)
x <- runif(100)
y <- runif(100)
ls()
save(x,y,file="myobjects.Rdata")
### filesize of image?
file.info("myobjects.RData")
save(x,y,file="myobjects.Rdata",compress=FALSE)
### and now?
file.info("myobjects.RData")
save.image()
rm(list=ls(all=TRUE))
ls()
```

```
load("myobjects.Rdata")
ls()
load(".RData")
### No Warning!
dump(c("x","y"),"myobjects.txt")
### readable!
dump(ls(patt='^[xy]'),"myobjects.txt")
rm(list=ls(all=TRUE))
source("myobjects.txt")
ls()
```

Strings

- Already seen: numbers, factors, Boolean values and missings
- Other kind of data: strings and date and time objects

(Some) functions for string handling I

- `print()`: very mighty, but **very** slow!
- `paste()` and `cat()`
  `paste()` simple construction of strings from shorter parts, `cat()` for showing strings on screen
- `formatC()` resp. `sprintf()` for very flexible output of numbers and strings

(Some) functions for string handling II

- `grep()`, `match()`, `pmatch()`, `%in%` looking for matches in strings
- `parse()`, `deparse()` converting strings and expressions
- `strsplit()`, `sub()`, `gsub()` manipulation of strings
- `nchar()` number of characters in a string
- `tolower()`, `toupper()` conversion to lower- resp upper-case letters

(Some) functions for string handling III
Example session strings

```
##### character strings
paste("file", 1:3, ".txt", sep="")
cat("file", 1:3, ".txt\n", sep="\t")
grep("A",letters)
grep("B",c(LETTERS,LETTERS))
all.equal(letters, tolower(LETTERS))
```

Example session strings cont.

```
### strsplit etc.
myname<- "Dr. Detlef Steuer"
strsplit(myname," ")
strsplit(myname,".")
### Attention Regular Expression!
strsplit(myname,"\\.")
sub("Dr.","Herr",myname)
myname
```

Fun with characterstrings I

Copyright: Die Missfits

```
to.feminispraech <- function(string){
 gsub("man","frau",
      gsub("mann","frau",
           gsub("u","ü",
                gsub("o","ö",gsub("a","ä",
 gsub("er","sie",tolower(string)))))))}
```

What is this function about?

Fun with characterstrings II

```
to.feminispraech <- function(string){
 gsub("man","frau",
      gsub("mann","frau",
           gsub("u","ü",
                gsub("o","ö",gsub("a","ä",
 gsub("er","sie",tolower(string)))))))}

to.feminispraech("Peter verkauft Füllfederhalter")
[1] "petsie vsiekäüft füllfedsiehältsie"
```

Dates and times

- Dates are a very important data type: time series, stock data etc.
- But very complicated: DST, computer time, leap years, leap seconds(!).
- `?DateTimeClasses` implements POSIX compliant date and time classes in R.
- Most important function: `strptime()`.
- Standard accuracy is one second, nowadays guaranteed accuracy for fractional values is down to better than one microsecond

Dates and times (Example session)

```
dates <- c("12/15/92","12/20/95","12/25/97")
times <- c("10:01:00","06:00:00","02:30:00")
x <- paste(dates, times)
x
(z <- strptime(x, "%m/%d/%y %H:%M:%S"))
class(z)
z[2]-z[1]
as.Date(z)
ISOdate ( 2008, 10, 9, 10, 30)
format(Sys.time(), "%a %b %d %H:%M:%S %Y")
```

Reading data: Excel

- Most commen question: How to read and write Excel sheets.
- You can, but you shouldn't!
- No clearly defined format, unannounced changes etc.
- If you must, there are options for Excel pre-2007.
- Package `xlsReadWrite`: Windows only, no OpenSource.
- For all Version: access using RODBC and look at the sheet as a simple database.
- Best practise: Convert to CSV-Files!

Reading data I

read.table-family of functions

- `read.table`: Reading data in tabular form. Returns a *data-frame*. Syntax:

```
read.table(file, header = FALSE, sep = "", quote = "\"'",
           dec = ".", row.names, col.names, as.is = FALSE,
           na.strings = "NA", colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "\#", allowEscapes = FALSE)
```

- An obviously very mighty command!

read.table() Details:

```
read.table(file,
# file may stand for a literal file or for a ‘‘connection’’
# Practical problem: encoding of files
# Can be solved using a connection of type file

     header = FALSE, sep = "", quote = "\"’", dec = ".",
     row.names, col.names, as.is = FALSE, na.strings = "NA",
     colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
     fill = !blank.lines.skip, strip.white = FALSE,
     blank.lines.skip = TRUE, comment.char = "\#",
     allowEscapes = FALSE)
```

read.table() Details:

```
read.table(file,  header = FALSE,
# header defines, whether the first line contains column names
# if yes: use given names
# else: just generate generic column names (V1, V2 etc.) and use
# the first line as data

     sep = "", quote = "\"'", dec = ".",
     row.names, col.names, as.is = FALSE, na.strings = "NA",
     colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
     fill = !blank.lines.skip, strip.white = FALSE,
     blank.lines.skip = TRUE, comment.char = "\#",
     allowEscapes = FALSE)
```

read.table() Details:

```
read.table(file,  header = FALSE,  sep = "",
# How are columns separated. An empty separater means
# columns are separated by whitespace: space, tab, carriage return

     quote = "\"'", dec = ".",
     row.names, col.names, as.is = FALSE, na.strings = "NA",
     colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
     fill = !blank.lines.skip, strip.white = FALSE,
     blank.lines.skip = TRUE, comment.char = "\#",
     allowEscapes = FALSE)
```

read.table() Details:

```
read.table(file,  header = FALSE,  sep = "",  quote = "\"'",
# quote contains all characters that are used to mark begin and end of
# fields containing strings. May be multiple different characters.
# String fields may contain whitespace

     dec = ".",
     row.names, col.names, as.is = FALSE, na.strings = "NA",
     colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
     fill = !blank.lines.skip, strip.white = FALSE,
     blank.lines.skip = TRUE, comment.char = "\#",
     allowEscapes = FALSE)
```

read.table() Details:

```
read.table(file,  header = FALSE,  sep = "",  quote = "\"'",
    dec = ".",
# character used as decimal point

     row.names, col.names, as.is = FALSE, na.strings = "NA",
     colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
     fill = !blank.lines.skip, strip.white = FALSE,
     blank.lines.skip = TRUE, comment.char = "\#",
     allowEscapes = FALSE)
```

read.table() Details:

```
read.table(file,  header = FALSE,  sep = "",  quote = "\"'",
     dec = ".", row.names,
# names of rows, given either as list of names, or number or name of th
# column containing rownames. If header=TRUE and the first line
# contains one column less, then the first column is used as row names
# row.names=NULL forces pure line numbering (and no names)

     col.names, as.is = FALSE, na.strings = "NA",
     colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
     fill = !blank.lines.skip, strip.white = FALSE,
     blank.lines.skip = TRUE, comment.char = "\#",
     allowEscapes = FALSE)
```

`read.table()` Details:

```
read.table(file,  header = FALSE,  sep = "",  quote = "\"'",
      dec = ".", row.names, col.names,
# column names, list, default "V1", "V2" etc.

      as.is = FALSE,
# use string fields as factors or not
# may be boolean for global definition for all columns
# or a vector of numbers or names of columns for which
# to assume as.is=TRUE

      na.strings = "NA",
      colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
      fill = !blank.lines.skip, strip.white = FALSE,
      blank.lines.skip = TRUE, comment.char = "\#",
      allowEscapes = FALSE)
```

read.table() Details:

```
read.table(file,  header = FALSE,  sep = "",  quote = "\"'",
     dec = ".", row.names,  col.names, as.is = FALSE,
     na.strings = "NA",
# list of strings which shall denote NAs (may be multiple
# strings)
     colClasses = NA,
# vector of classes which shall be assigned to columns
# (forcing conversion!)
# Attention! Access using column number and counting the
# column containing row names
     nrows = -1, skip = 0, check.names = TRUE,
     fill = !blank.lines.skip, strip.white = FALSE,
     blank.lines.skip = TRUE, comment.char = "\#",
     allowEscapes = FALSE)
```

read.table() Details:

```
read.table(file,  header = FALSE,  sep = "",  quote = "\"'",
     dec = ".", row.names,  col.names, as.is = FALSE,
     na.strings = "NA", colClasses = NA, nrows = -1,
# Number of rows to read. -1 means ``all''!
     skip = 0,
# number of rows to skip at the beginning of a file
# Ignore some header
     check.names = TRUE,
# check if variable names are valid in R
# modify names if needed
     fill = !blank.lines.skip, strip.white = FALSE,
     blank.lines.skip = TRUE, comment.char = "\#",
     allowEscapes = FALSE)
```

read.table() Details:

```
read.table(file,  header = FALSE,  sep = "",  quote = "\"'",
     dec = ".", row.names,  col.names, as.is = FALSE,
     na.strings = "NA", colClasses = NA, nrows = -1,
     skip = 0, check.names = TRUE,  fill = !blank.lines.skip,
# boolean. Implicitely fill blank lines??

     strip.white = FALSE,
# remove whitespace from read in strings?

     blank.lines.skip = TRUE,
# ignore blank lines?

     comment.char = "\#",
     allowEscapes = FALSE)
```

read.table() Details:

```
read.table(file,  header = FALSE,  sep = "",  quote = "\"'",
      dec = ".", row.names,  col.names, as.is = FALSE,
      na.strings = "NA", colClasses = NA, nrows = -1,
      skip = 0, check.names = TRUE,  fill = !blank.lines.skip,
      strip.white = FALSE, blank.lines.skip = TRUE,
      comment.char = "\#",
# how are comment lines in my data file identified

      allowEscapes = FALSE
# are escape seqences like \n or \t  allowed in strings or not?

)
```

HELMUT SCHMIDT UNIVERSITÄT
Universität der Bundeswehr Hamburg

`read.table()` **service function**

Aliases of read.table with modified defaults

- `read.csv()`, `read.csv2()` - defaults for .csv files (*comma separated values*), resp. in read.csv2 semicolon separated values and German conventions for the decimal point.

- `read.delim()`, `read.delim2()` - same as csv variants but for TAB separated values

- `read.fwf()` Reading of data present in a *fixed width format*. One example for such data are SAS files. These files often have no separator, but use positional parameters to interpret the contents of a file.

(Simple) exercise: reading data

- Download elbe.zip and flensburg0-89.pdf from my Laptop
- Save data to a directory of your choice
- Real-world files of flow of Elbe river and floods in Hamburg
- Read the file `'1979-2004-T-NDA.csv'` (and create a simple plot)

Solution

```
> setwd("/home/steuer/Flensburg/Daten/Elbe")
> NDA <- read.csv2("1979-2004-T-NDA.csv", header=TRUE, sep=";")
> plot(NDA[,2], t="l")
> title("Average Flow at Neu-Darchau")
```

(Medium) exercise: reading data

- Load `'1950-2005-Sturmfluten.csv'` in a data frame named `'floods'` in your R Session!
- Rearrange the data: Add a column containing the catory of each flood.
- Collapse the three columns tide level into one.
- For visual inspection create a simple plot of tide levels.

Solution:

```
>  floods <- read.csv2("1950-2005-Sturmfluten.csv",skip=2)
>  floods <- floods[1:211,1:4]
#  recoding
>  kategorie <- rep(NA,211)
>  for (zeile in 1:dim(floods)[1])
>  {
>   if (!is.na(floods[zeile,2])) kategorie[zeile] <- "normal"
>   if (!is.na(floods[zeile,3])) kategorie[zeile] <- "schwer"
>   if (!is.na(floods[zeile,4])) kategorie[zeile] <- "sehr schwer"
>   floods[zeile, 2] <- max(floods[zeile,2:4],na.rm=TRUE)
> }
> floods <- cbind(floods[1:2],kategorie)
> names(floods) <- c("Datum", "Pegel in cm", "Kategorie")
> rm(kategorie)
> plot(sturmfluten[,2],t="l")
```

Improvements needed

- Distances between points don't represent time between floods
- Nice axis labels

Solution

```
> sturmfluten <- cbind(sturmfluten,
       strptime(as.character(sturmfluten[,1]),"%d.%m.%Y" ))
> plot(as.numeric(sturmfluten[,4]), sturmfluten[,2],t="l" ,
       main="Tide level during stormfloods HH", xlab="time",
       ylab="tide level", axes=FALSE)
> axis(2)
> axis(1,at=sturmfluten[,4],label=sturmfluten[,4] )
> points(as.numeric(sturmfluten[,4]), sturmfluten[,2],
  col=sturmfluten$Kategorie)
```

Saving your work (data)

- Most important: Take care of your R-code!
- Save your manipulated data with `write.table()`.
- `write.table()` takes a dataframe and a filename, options analogous to `read.table()`
- Ex.: `write.table(sturmfluten, file="sturmfluten.txt", sep=";", row.names=FALSE)`

Reading Data II

- `scan()`: Mightier than `read.table` Data isn't passed to a data frame but into a vector. Finds its application, if data isn't nicely tabulated.

- `readLines()`: Used if data has still less structure to rely on . . . .
  Reads data line by line. Important, if data and describing text are heavily intertwined.

- `library(foreign)`: Package to directly read binary formats of other programs. I.e. S3 binary files, SPSS, SAS XPORT files, Octave, Excel, Stata, Openoffice sheets . . .

- `readBin()`: Reading of files with known binary format.

Reading Data III

- Interface to databases (SQL)
  - interfaces for specific databases, i.e. MySQL, Postgres
  - standardised interface ODBC in package `RODBC`, i.e. DB2 or accessing Excel using ODBC in Windows

- Distributed over lots of contributed packages support for special formats: wave files, maps resp. images . . . .

Advance Exercise: reading data

- Generate ideas how to read in the `*.asc` files
- Solution will be presented!

Solution

```
pegelread <- function (file)
{
  tmpdata <- readLines(file)[-(1:4)]
  # first 4 lines contain header
  for (zeile in 1:length(tmpdata) )
  {
    ### trim the different amount of TABs around the data
    tmpdata[zeile]<-gsub('\t',' ',
             sub('^\t+','',sub('\t+$','',tmpdata[zeile])))
    ### standardize dates
    aktzeile <- unlist(strsplit(tmpdata[zeile]," "))
```

Solution II

```
  ### 1 or 2 mesurements per line
  for (pos in seq(1,length(aktzeile),3))
  {
    datumv <- unlist(strsplit(aktzeile[pos],"\\."))
    aktzeile[pos] <- paste(datumv[3:1],collapse="-")
  }
  tmpdata[zeile] <- paste(aktzeile,collapse=" ")
}

tmpdata <- unlist(strsplit(paste(tmpdata,collapse=" ")," "))
noobs <- length(tmpdata)/3
```

Solution III

```
  ### store dates as POSIX dates and levels as integer
  noobs <- 3*noobs
  obs0 <- seq(1, noobs, 3)
  obs1 <- seq(2, noobs, 3)
  obs2 <- seq(3, noobs, 3)
  result <- data.frame(
     Zeitstempel = as.POSIXct(paste(tmpdata[obs0],tmpdata[obs1])),
     Pegel = as.integer(tmpdata[obs2]))

  ### force correct sequence according to times
  result <- result[sort(result$Zeitstempel,
                  index.return=TRUE)$ix,]
}
```

Solution IV

Afterwards, you can mass process such files!

```
### the datareading
pegelBlankeneseUnterfeuer <- pegelread("79-04-T-BLAPegel.asc")
pegelBrunsbuettel <- pegelread("79-04-T-BRUPegel.asc")
....
```

Saving plots

- R uses a device model. Basically you open a *device* by name, i.e. `x11()`, `windows()`, `win.metafile()`, `postscript()`, `pdf()`, `png()` etc.
- Many more devices supported in contributed packages, i.e. `SVG()`.
- Subsequent plotting commands will act on the *active* device. Most of the time the one opened latest.
- Many device may be open at once. (But difficult to use interactively.)
- dev.off() closes the *active* device, i.e. closes window on screen and cleanly closes files.
- Convenience functions: `dev.print()` and `dev2bitmap()`.

Simple generating of graph in a pdf file

```
> pdf(file='out.pdf')
> par(mfrow=c(2,1))
> plot(sturmfluten[,2],t="l")
> plot(as.numeric(sturmfluten[,4]), sturmfluten[,2],t="l",
       main="Tide level during stormfloods HH",
       xlab="time", ylab="tide level", axes=FALSE)
> axis(2)
> axis(1,at=sturmfluten[,4],label=sturmfluten[,4] )
> points(as.numeric(sturmfluten[,4]), sturmfluten[,2],
         col=sturmfluten$Kategorie)
> dev.off()
```

Graphics I

- special plots: `boxplot()`, `dotplot()`, `curve()`
- universal command plot()
- complemented by `lines()`, `points()`, `polygon()`, `abline()` ...
- and `title()`, `par()`, `text()`, `mtext()`, `plotmath()`, `legend()`, ...
- alternative graphics system: Trellis plots in library(lattice) and very new ideas in library(ggplot2)
- flexible layout in `library(grid)`

Graphics II

- 3-D graphics: `scatterplot3D()`, `contour()`, `persp()`, `image()`, . . .
- barely supported: animated graphics and interactive graphics. Partly realized in `library(xgobi)` resp. `ggobi`

Graphics III

- `demo(graphics)`
- `library(grid); example(grid)`
- `library(lattice) ; demo(lattice)`
- `detach(package:lattice)`: method to remove a loaded package from current workspace and search path
- `demo(plotmath)`: mathematical symbols, greek letters, formulae in plots. Uses TEX-syntax
- Fantastic source for inspiration and detailed examples: `http://addictedtor.free.fr/graphiques`

Graphics IV: `plot()` details

- `plot(x, y, ...)` is a so-called generic function. Arguments can be either x and y coordinates of data points or an R-object for which a plot-method exists!

- `methods(plot)` shows all methods, implemented in R-base for the generic `plot()`

```
 [1] plot.acf*            plot.data.frame*    plot.Date*
 [4] plot.decomposed.ts* plot.default        plot.dendrogram*
 [7] plot.density        plot.ecdf           plot.factor*
[10] plot.formula*       plot.hclust*        plot.histogram*
[13] plot.HoltWinters*   plot.isoreg*        plot.lm
[16] plot.medpolish*     plot.mlm            plot.POSIXct*
[19] plot.POSIXlt*       plot.ppr*           plot.prcomp*
[22] plot.princomp*      plot.profile.nls*   plot.spec
[25] plot.spec.coherency plot.spec.phase     plot.stepfun
[28] plot.stl*           plot.table*         plot.ts
[31] plot.tskernel*      plot.TukeyHSD
```

Graphics V: `plot()` details II

- Still missing and interesting the ... part in `plot(x, y, ...)`
- Passing arguments for defining the appearance of the plot.
- Most important:
  - `type=` : „p“ points, „l“ lines, „b“ both, „n“ no plot
  - `main=, sub=, xlab=, ylab=` : each a string for title, subtitle and axes description
  - `xlim=, ylim=`, each a pair of numbers `c(lowerbound, upperbound)` defining the area of the plot in data coordinates.
  - `axes=FALSE`, to individually define axes
  - `pch=`, character to plot single points

Graphics VI: `plot()` details III

- Some parts of a plot can be defined by specific functions
  - `title()` : title, subtitle, etc.
  - `axis()` : controls individual axes, including tick-marks, labels etc.
  - `text()`, `mtext()` : add strings to specific parts of the plot
  - `legend()`: adding a legend to a plot.
- Lots of parameters can be set using `par()`, i.e.
  - `lwd`, `lty` : width and type of line
  - `col` : color of plotting symbols
  - `mfrow` : layout of multiple plots inside one side of plots. For example par(mfrow=c(3,4) results in 3 rows and 4 columns of plots filled serially from upper left corner to lower right, 12 plots per page
- `help(par)` has 500 lines . . .

The *formula*-concept of R

- In R models can be specified in a compact, elegant and formalized form
  `lm(y ~ x)` expresses: x explains y (not only `lm`!)

- The intercept is implicitly included in a model.
  `lm(y ~ x -1)` can be used to eliminate it.

- Multiple independent variables are separated by '+' symbols:
  `lm(y ~ x + z)`

- Higher order models are expressed with symbols '*' and ':'
  (and a few more for more complicated situations; see ?formula)

- Transformations of variables must be isolated using the `I()`-operator.
  `lm( y ~ x + I(log(x))` calculates a model with log(x) as variable.

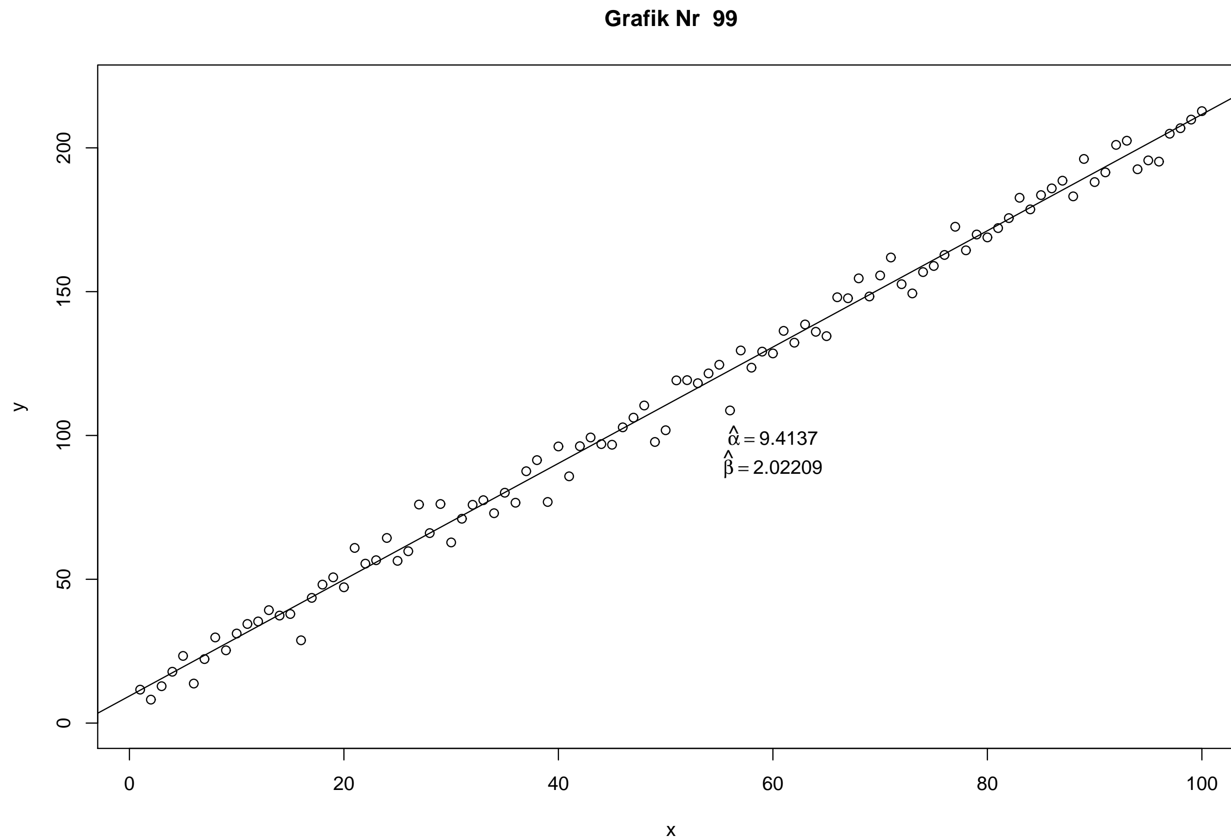
Graphics VII: Exercise

- Generate a series of 100 plots, on each page of plots 2x5 subplots,
- Each plot for a model of kind $y = \alpha + \beta \cdot x + \varepsilon$ with $\varepsilon \sim N(0, 5)$ and $x_1 = 1, x_2 = 2, \ldots, x_{100} = 100$, $\alpha = 10$, $\beta = 2$ for data generation.
- Data must be simulated using random numbers, $\hat{\alpha}, \hat{\beta}$ must be estimated from that simulations.
- The graphs must contain the points and regression line, plot title must contain plot number.
- Furthermore each plot must contain the Greek symbols for the estimators and the estimated values $\hat{\alpha}, \hat{\beta}$.

Strategy

- Starting from sub-problems working the way up to a loop over all plots.
- Identify sub-problems:
  - How to generate random numbers?
  - Estimate model, how to access estimators?
  - How to plot Greek symbols?
  - How to put a count in the title?
  - Layout and loop.

One of the plots:



Plot generating R-code

```
x <- seq(1,100)
y <- 10 + 2*x + rnorm(100,0,5)
mod1 <- lm( y ~ x)
mod1$coef
plot(x,y,ylim=c(0,220))
abline(mod1)
text(60,100,bquote(hat(alpha) == .(as.numeric(mod1$coef[1]))))
text(60,90,bquote(hat(beta) == .(as.numeric(mod1$coef[2]))))
i<-99
title(main=paste("Grafik Nr ",i))
```

Databases + SQL + R I

- Why are databases used?
  - Volume of data: R isn't designed to work on massive data sets. If it runs out of main memory R is helpless.
  - dynamically changing data
  - databases are optimized to handle huge data set. There are even file systems implemented only for usage in a database

Databases + SQL + R II

- What is a (relational) database
  - First: just a collection of tables defining relations between objects for which data are stored.
  - There is a complete theory how to correctly (efficiently) specify these tables.
  - Aim: remove redundancy and possibility of inconsistencies in data.
  - Normally implemented by IT-staff.

Databases + SQL + R III

- Technically: data are hold by a *database server*
- A *database client* allows access to data. Session structure is the same for all clients and servers:
  - Connect to server: host, port, user, password
  - Choose one of the provided (logical) databases
  - Send queries to the server
  - Read the resulting table
  - Disconnect
- No local server required. Normally access over network.

(Minimal) SQL

- Specialized language for database requests:
  SQL (Structured Query Language)
- Exististing norms:
  a) SQL92 or ISO/IEC9075 or ANSI X3.135-1992
  b) SQL99
- Example Queries

```
SELECT State, Murder FROM USArrests WHERE Rape > 30 \
                           ORDER BY Murder
SELECT sex, COUNT(*) FROM student GROUP BY sex
SELECT sch, AVG(sestat) FROM student GROUP BY sch \
                           LIMIT 10
```

(Minimal) SQL II

- Always returns tables.
- There are rudimentary aggregation functions in SQL: COUNT, AVG, MIN, MAX, SUM.
- These work column-wise, often faster than transferring data to R and calculate statistics there.

SQL via R I

Example session using RMySQL. Very similar for different database systems

```
> library(RMySQL) # loads automatically DBI,
                  # which is shared by all db interfaces

## Connect to a local database
## normally authentification required!
## Hostname, port, username, password
> con <- dbConnect(dbDriver("MySQL"), dbname = "test")

## Which tables are present in the database?
> dbListTables(con)

## pure SQL: > show tables;
```

SQL via R II

```
## Transferring a dataframe to a database
> data(USArrests)
> dbWriteTable(con, "arrests", USArrests, overwrite = TRUE)
TRUE
> dbListTables(con)
[1] "arrests"
## Table "‘arrests"’ freshly created
## read whole table fram database
> dbReadTable(con, "arrests")
        Murder  Assault     UrbanPop    Rape
Alabama 13.2    236         58          21.2
Alaska  10.0    263         48          44.5
Arizona 8.1     294         80          31.0
...
## a dataframe is returned as result
```

SQL via R III

```
## Select from a loaded table
> dbGetQuery(con, "select row_names, Murder from arrests
                   where Rape > 30 order by Murder"))
row_names Murder
1 Colorado 7.9
2 Arizona 8.1
3 California 9.0
4 Alaska 10.0
5 New Mexico 11.4
...
## remove test data
> dbRemoveTable(con, "arrests")
## Close Connection
> dbDisconnect(con)
```

Thank you very much!

steuer@hsu-hh.de