Guidelines for Statistical Projects: Coding and Typography

Marius Hofert¹, Ulf Schepsmeier²

2018-07-25

Abstract
Guidelines for conducting, implementing (in \LaTeX{} and R) and documenting statistical (research) projects are provided in order to improve readability and to reduce the error rates of theses, scientific papers, reports and source code. This is meant to save supervisors, students, package maintainers and practitioners a lot of time. It is clear, however, that such guidelines cannot be exhaustive. The given recommendations should therefore rather serve as a starting point for improving your workflow and to avoid common pitfalls in statistical projects of larger scale.

Keywords
Implementation, source code, typography, \LaTeX, R.

MSC2010
68U15, 68U20, 68U05, 97R60.

Contents

1 Introduction 2

2 General suggestions 3

2.1 Forget about the Pareto principle (80–20 rule) ........................................ 3
2.2 When solving a particular problem for the first time, spend time on it ........ 4
2.3 The English language in mathematics ......................................................... 4
2.4 Be concise, consistent, structured, self-contained and reproducible .......... 6
  2.4.1 Be concise ................................................................. 6
  2.4.2 Be consistent ........................................................... 8
  2.4.3 Be structured ........................................................... 8
  2.4.4 Be self-contained ....................................................... 9
  2.4.5 Be reproducible ....................................................... 10
  2.4.6 Be organized .......................................................... 10
2.5 Optimize communication, meetings and preparation ............................ 10

3 Editors and integrated development environments 13

4 \LaTeX{} 14

4.1 Typographic recommendations for mathematical documents .................. 14

¹Department of Statistics and Actuarial Science, University of Waterloo, 200 University Avenue West, Waterloo, ON, N2L 3G1, marius.hofert@uwaterloo.ca. The author would like to thank NSERC for financial support for this work through Discovery Grant RGPIN-5010-2015.

²Department of Mathematics, Technische Universität München, 85748 Garching, Germany, ulf.schepsmeier@tum.de
# 1 Introduction

We present guidelines meant for scholars (ranging from students to supervisors) and practitioners who would like to write, participate in or supervise a project such as a Bachelor’s thesis, a Master’s thesis, a Ph.D. thesis or a (scientific) paper. Our target area is the intersection of mathematics,
statistics and computer science. Our guidelines address general recommendations as well as software tools like editors, \LaTeX\ and \texttt{R} for planning, conducting, implementing and documenting the project of interest.

Before going into detail, some remarks are in order:

**The science of coding** Coding (in \LaTeX, \texttt{R} etc.) is like handwriting. From the corresponding files and style of coding one can read a lot. Writing correct, numerically sound, readable, well-documented and easy-to-maintain code is a science on its own and one that is not taught explicitly at university level unless one specifically studies computer science, but this course of studies rarely addresses \LaTeX\ and \texttt{R}. However, with the ever increasing complexity of statistical simulations and projects, it becomes important to have coding guidelines – otherwise it might be difficult for others to

- understand what you actually want to show or do with your code;
- improve and extend your code; and to
- maintain your code in the long run (maintenance time is typically highly underestimated!).

**Motivation** These guidelines are motivated from *our own work* and *our work with students and practitioners*. After pointing out improvements, making code more readable, correcting common mistakes and improving submitted documents over and over again, we hope that these guidelines help all parties involved in a project to avoid (what we believe are) common pitfalls and to act as a time-saver.

**Focus** The guidelines reflect our *personal recommendations* and *experience* using \LaTeX\ and \texttt{R} and related tools. The examples we use to support our points lie in our *personal areas of research*, which lies in the intersection of mathematics, statistics and computer sciences. Adoption to other research fields should be possible easily. It is clear that such a guide cannot be exhaustive. In particular, this is *not an introduction* to the topics presented!

**Goal** Our *goal* with these guidelines is not to make a document or code snippet 100% perfect. There are exceptions to almost any rule and describing all of them would extend the page count of this document well beyond what you would be willing to read. Furthermore, some aspects are discussed in more detail than others.

**Style** This guide is built in a modular way. Each section, subsection or paragraph aims to be self-contained and self-explaining. Also, where needed we state short examples for illustration.

**Disclaimer** This document does not exist to torture you(r workflow) to use a specific kind of operating system, editor, software, etc. It should rather point out how *we tackle certain problems* (and partly, but not always (!), why we solve them the way we recommend). You may or may not find this helpful, the principle *Do not like it? Do not use it!* applies. Last but not least, given the fast development of the research areas we work in, it should be clear that our guidelines may become outdated to some degree at some point in time.

## 2 General suggestions

### 2.1 Forget about the Pareto principle (80–20 rule)

**Definition** The Pareto principle (or 80–20 rule) says that 80% of the final outcome, result or effect (of projects, events etc.) is achieved by 20% of the input or causes.

**Meaning** Essentially, this means that one should stop after having spent 20% of the overall time or effort, since all additional effort would just improve the outcome by the remaining 20%.
2 General suggestions

Why not? The Pareto principle is frequently used in many areas. However, it does not apply to scientific work. If you write a research paper, for example, you can (and should) expect it to return to your desk at some point in time. You certainly do not want to realize a year or two later, that you now actually have to start over with the whole project. Furthermore, if a referee or supervising authority feels that you only spent 20% of the effort on the submitted work, this most likely results in negative feedback (for example, a rejection). Do your homework, work hard and exclusively on the topic, focus on obtaining good results (instead of rushing out another publication) and you can be happy with your work for a long time. Also, your supervisor is typically delighted to learn about what comes “after the 80%”; keep that in mind at any stage.

2.2 When solving a particular problem for the first time, spend time on it

In programming, there is this basic (unwritten) law, which applies equally well to research projects in general:

1) If you have a problem, search for it. The chance that you are the first one working on it is small; others may have already solved the problem in an elegant, optimal, fast, reliable and readable way.

2) If you cannot find a solution, search more, search differently, search longer – but search for it!

3) If you still cannot find it, go back to 2).

4) If you are sure there exists no good solution, write your own. Spend a lot of time on it to ensure the solution is excellent. Then make it available to the public and maintain it in the future so that others can benefit from your work.

Concerning 1) and the links we provide below, one approach is to look for solutions provided by senior members of mailing lists, forums, blogs etc., as there can be significant differences in quality.

In short, if there is a good solution available, learn from it. If there is not, develop your own solution, but make sure it is of such a good quality that others can benefit from it when they find themselves in the same position as you (searching for a solution to the given problem).

2.3 The English language in mathematics

The language science speaks is English; there is already some debate possible here: American versus British English. Even if it is only a comment in a script you write, a file name, variable, or function etc., consistently use (either American or British) English. Besides various other advantages, it will be easier for others to find, understand and cite your work. Additionally, we want to mention some basic rules for mathematical typography in English in what follows; here we partly follow Halmos (1970) and Higham (1993) (we also learned a lot from Paul Embrechts and Nick Bingham).

Short(er) sentences Use short sentences. Long sentences are not as conventional in English as they may be in other languages (such as German, for example). Formulate (sufficiently) simple and simple-to-understand sentences. This improves readability of your text, code documentation etc.

Pluralis majestatis In scientific documents one typically uses “we” instead of “I”, even if there is only one author – the “we” represents the author and the reader.

4
2 General suggestions

Readable text instead of mathematical operators  In the English mathematical literature, words such as “there exists” or “for all” are to be preferred over their operator equivalents “∃” and “∀”, and one aims at formulating full sentences instead of sequences of mathematical symbols. This makes the text more readable. Also, at least in full sentences, mathematical formulas, equations etc. should be considered as parts of the sentence and thus be equipped with proper punctuation.

**Bad:**
The generalized Pareto distribution (GPD) is given by

$$G_{\xi,\beta}(x) = \begin{cases} 1 - (1 + \xi x/\beta)^{-1/\xi}, & \xi \neq 0 \\ 1 - \exp(-x/\beta), & \xi = 0 \end{cases}$$

∀β > 0 and the support is x ≥ 0 ∀ξ ≥ 0 and x ∈ [0, −β/ξ] ∀ξ < 0.

**Good:**
The *generalized Pareto distribution (GPD)* is given by

$$G_{\xi,\beta}(x) = \begin{cases} 1 - (1 + \xi x/\beta)^{-1/\xi}, & \xi \neq 0 \\ 1 - \exp(-x/\beta), & \xi = 0 \end{cases}$$

where β > 0, and the support is x ≥ 0 when ξ ≥ 0 and x ∈ [0, −β/ξ] when ξ < 0.

Also note that we used *italics* for “generalized Pareto distribution” (including its abbreviation GPD) as this is a definition (and appearing for the first time).

Comma rules in English  Non-native English speakers are often unsure if and when a comma has to be set in a sentence. Here are some rules:

- A nonrestrictive element, which does not limit scope but merely provides additional information, is indicated by being set off by commas.
  Example: “The greatest mathematician of the ancient **world, Archimedes, was** also a physicist, engineer, inventor, and astronomer.”

- A restrictive relative clause is introduced with “that” and is not set off by commas.
  Example: “The real **numbers that cannot** be expressed as the quotient of two integers are called irrational.”

- A nonrestrictive relative clause is introduced with “which” and is always set off by commas.
  Example: “The set of rational **numbers, which** is an abelian group under addition, is in fact a subfield of the real numbers.”

- Use a semicolon only where you could also use a full stop.

- Mind commas in if-clauses: “If you knew all that I know, you would know what I mean”, but “You would know what I mean if you knew all that I know”.


Different project leaders or scientific journals may require different styles and rules to follow. Overall, it thus helps to be consistent (see also below) throughout a whole project. If required, search-and-replace tools can then be used at a more final editing stage to change certain language rules. However, never blindly apply such tools on your whole project – check each appearance case one-by-one. Also, be aware of the fact that your search tool might not find strings extending over more than one line or might be case sensitive.
2 General suggestions

2.4 Be concise, consistent, structured, self-contained and reproducible

2.4.1 Be concise

Be precise Most importantly, be mathematically correct (for example, note the difference between $\in$ and $\subseteq$, a quite common mistake). Furthermore, be concise in your descriptions, documentation, proofs, etc. This especially applies to assumptions made in mathematical statements.

Common careless errors Beware of mistakes (supervisor or author names, dates, spelling of affiliations etc.) on title pages, covers, bibliographies etc. One tends to not check such pages again after they have been created and mistakes can easily remain unseen. The same applies to elementary formulas.

Example from Cherubini et al. (2004, p. 49):

Bad: 

$$F^{-1}_i(t) = \inf\{u : F_i(u) \geq t, \ 0 < t < 1\}$$

Good: 

$$F^{-1}_i(t) = \inf\{u : F_i(u) \geq t\}, \ 0 < t < 1$$

Also note that, for example, on page 87 of Cherubini et al. (2004), the name “Klüppelberg” is written as “Klüppenberg”. Spell checkers only partially help here as they often detect a huge number of unknown “words” in the document due to the appearance of LaTeX formulas, names etc.

Be short Besides being less to type and less to proofread, correct and grade, twenty well-written pages are much more interesting and fun to read than one hundred sloppy and boring pages. The less is more principle applies here. Also, keep in mind the limited time your supervisor or boss has to look at your project – leave him the time to give you good feedback rather than taking away time from her/him by making her/him read a hundred pages! Even more so, other researchers or practitioners interested in your work will also not have the time to read and understand your project in detail – nowadays, no one has.

Everywhere in the documentation Being concise applies to various parts of a project, for example, the documentation:

Headings Headings and the table of contents should provide a golden thread or structure which should be easy to grasp without even reading the text.

Figures and tables Figures and tables, including their captions, should be easy to read and understand without having to search in the text for the corresponding explanations. Use legends to describe and distinguish different line types or colors right in the respective plot, for example.

Bad: 

Figure 4: Normal distributions

Good: 

Figure 4: Probability density function of the normal distribution with different mean and variance (blue: $\mu = 0, \sigma^2 = 0.2$, red: $\mu = 0, \sigma^2 = 1.0$, green: $\mu = -2, \sigma^2 = 0.5$).

Formulas Put important formulas in a displayed equation. In the same spirit as before, a displayed equation/formula etc. should make sense as much as possible without looking at the text. Check that the main ideas of your work can be followed by just looking at the displayed equations. Only number those displayed equations which also appear in the text
at some point. Also, when introducing a function \( f \) for the first time, make it more precise by providing its domain.

**Bad:**
Consider \( f(x) = \log x \).

**Good:**
We now introduce the major function \( f \) of interest, given by

\[
f(x) = \log x, \quad x \in (0, \infty).
\]

Conversely, more complicated formulas should always be explained in verbal form in the text as well. This, together with the displayed equation or formula, gives the reader the chance to understand the topic on two different levels, one language-based and one formula-based. Sometimes, one can also provide a third, graphical-based level by illustrating the (complicated) formula with a graph. Providing a publicly available implementation provides another such level.

**File, variable and function names** Name files, variables and functions in a meaningful way (both from a mathematical and a programming point of view). Label versions of your files by starting with the date in ISO 8601 date format (such as `2013-12-31_my_project.R`). This way, they are displayed in chronological order if files in the current folder are sorted by name. This is helpful when searching for older versions of `2013-12-31_my_project.R`.

Do not call a variable `variable` or `var`. Instead, give it a context-related and self-explaining name (ideally even such that the type (integer, real, etc.) of the variable is obvious from its name), such as \( n \) for a sample size, \( i \) for the index running through all samples or \( \tau \) for a certain value of Kendall’s tau (similar to the standard notations \( n \), \( i \) and \( \tau \) used in statistics). Choose variable names as close as possible to their mathematical equivalents. In the same spirit, do not call a function `function` or `fun`; note that R, for example, would not even allow to use the reserved name `function`.

**Bad:**
1. `var` # reserved expression
2. `variable` # no reference to a math.
3. `a` # equation or standard
4. `b` # notation

**Good:**
1. `tau` # Kendall’s tau
2. `rho` # Spearman’s rho
3. `n` # number of observations
4. `d` # dimension

Also, do not encode a certain method or outcome in numbers if it is not a number naturally. For example, colors 1, 2 and 3 are much less self-explaining than colors “blue”, “green” and “red”.

**Bad:**
1. `plot(x, y, type="l", col=2)`

**Good:**
1. `plot(x, y, type="l", col="red")`

The following basic rule typically provides compact and readable code: The more often you need a variable or function, the shorter its name should be. But, as mentioned above, make sure the name is readable, context-related and self-explaining. Short variable names can often be generated by leaving out vocals, the human eye typically “interpolates” the letters correctly and directly recognizes the corresponding word. Short but readable function names can often be found by using underscores to separate parts of the function name. For example, a function computing bounds on Value-at-Risk (mathematical abbreviation: VaR) in the case of equal marginal distribution functions (mathematical parlance: homogeneous case) could be called
2 General suggestions

VaR\_bounds\_hom; this is preferred over camel-case naming which might lead to inconsistencies and is often less readable.

2.4.2 Be consistent

**Stick to (your) rules** Consistently use the same notations for the same quantities throughout the text. More generally, stick to the (typographical/coding) rules you use exactly in the same way throughout the whole file (.tex document, .R script, .txt files etc.), from the very first to the very last character in the file; even be consistent in the way you use white space. This will significantly help you when search-and-replace is in order.

Say, you use a special rule for writing nested parenthesis, for example one of

$$\left(\left(\left(\left(a_7 x + a_6\right) x + a_5\right) x + a_4\right) x + a_3\right) x + a_2\right) x + a_1 x + a_0$$

or

$$\left\{\left(\left(\left(a_7 x + a_6\right) x + a_5\right) x + a_4\right) x + a_3\right) x + a_2\right\} x + a_1 x + a_0.$$

It does not matter much which is to be preferred on first writing, as long as you stick to the very same rule throughout the whole document. For publications in scientific journals, for example, the style of nested parenthesis is often determined by the journal’s style guide. Be extremely picky here; for example, if you work with quantile functions $F^{-j},\ j \in \{1, \ldots, d\}$, always write them as $F_j^{-}$ and not once as $F^{-j}_j$ and once as $F^{-j}_{-j}$ in \LaTeX. Obviously, the result looks the same, but such inconsistencies make it more difficult to search (and possibly replace) these instances at a later editing stage.

2.4.3 Be structured

**Abstract, introduction and conclusion come last** Do not start to write your paper or project documentation by thinking about the introduction. The abstract, introduction and conclusion are the last parts you should write in your project. First concentrate on the content. In the very last step, think about the abstract, introduction and conclusion; you may note keywords (for example, about the motivation behind the present work) or the major research question, though, at any time. Additionally, you can also determine the most important three or so words on every page of your document. This can help in creating a golden thread for the paper as a whole.

**Numbering** To structure your manuscript into meaningful parts you can use chapters (but only in large manuscripts like books or theses), sections, subsections, or paragraphs. Do not use too many levels of headings. In most cases, three numbered levels are sufficient. In smaller reports or papers even two levels are typically fine. Only use subsections if you have more than one meaningful subsection. Otherwise omit the subsection and work with paragraphs.

**Two possible ways** There are typically two ways to start a document and structure it.

**Bottom-up** Collect all your ideas, write them down, and then structure them into associated parts, sections and chapters.

**Top-down** Think about a logical way of reading/following your paper. Write down the chapter and section headings you have in mind and order them. Then fill the sections with content.
2 General suggestions

**Listings**  Sometimes, lists of bullet points can be helpful to write down several connected statements in a compact way. If they have a natural order one should use an ordered list. You can use different numbering styles, for example, arabic or roman numbers, or alphabetical items. In unordered lists, different bullet point styles are also available. Nested lists should differ by different bullet points or numberings. But do not nest too deeply. More than two or three levels are typically not required. This would also decrease readability.

### 2.4.4 Be self-contained

**Outsource**  Instead of reproducing known results, properly refer to papers, books, source code or packages your work is based on; when referring to books, either always provide a page number or always provide a (sub)section number. Also mention the edition of the book in the references. Note that R packages should also be cited, by their manuals.

Example (.bib file entry):

```latex
@Manual{qrmtools2015,
  title = {{q}rmtools: Tools for Quantitative Risk Management},
  author = {Marius Hofert and Kurt Hornik},
  year = 2015,
  note = {R package version 0.0-3},
  url = {http://CRAN.R-project.org/package=qrmtools},
  urldate = {2015-12-29}
}
```

**How to cite**  If possible, use author-year citation style such as “Name1 and Name2 (YYYY)” or “Name1 et al. (YYYY, pp. 17)”; this provides the most readable and memorable citations. Also, do not cite references alone. Instead, enhance references by providing further information and connections to your work. This way, the main idea can be followed without having to read the references first, which makes the document more self-contained. Note that “p. 17” is used to refer to page 17 directly and “pp. 17” to refer to page 17 and thereafter.

**Bad:**  “It follows from [1] and [2] that z holds.”

**Good:**  “In terms of our setup here, A (2000) showed that x holds. With this result, the assumption of the main theorem in B (2010) holds, which states that... One can thus conclude that z holds.”

**Do not cite the world’s literature**  Cite (only) the main or original reference and not a myriad of references which are more or less related to a topic/theorem/definition/statement. Otherwise, it would make the text unnecessarily long and difficult to read (and by Section 2.4.1, we wanted to be concise!). Also, only provide those references (in the back of the document) which you indeed cite in the text at some point.

**Minimal working example**  The intersection of being concise and self-contained also relates to the notion of a minimal working example. A minimal working example is an example (typically source code) which contains everything necessary to demonstrate and reproduce a problem but is as minimal as possible, omitting everything not directly related to the problem; see [http://minimalbeispiel.de/mini-en.pdf](http://minimalbeispiel.de/mini-en.pdf) for tips on how to construct a minimal working example (here in the context of \LaTeX).
2 General suggestions

2.4.5 Be reproducible

**Meaning** Make sure your results are reproducible. That is, one can repeat the conducted experiments, simulations or proofs with the exact same results.

**Seed and more** To obtain a reproducible statistical simulation, always set a seed! For more on this, including instructions how to conduct simulation studies (in R), see (the ideas and words of warning in) Hofert and Mächler (2014).

**Sweave, Knitr** For manuscripts containing R code or R results one possibility to achieve reproducibility is Sweave (Leisch (2002)) or Knitr (Xie (2015b), Xie (2013), and Xie (2014)). These two R packages allow to combine R and \texttt{\LaTeX} code in an \texttt{Rnw} file. Every time you change a calculation in the R code and compile the whole document, the R results are automatically updated and propagated to the PDF or HTML file created by \texttt{\LaTeX}.

Both packages are very useful for small projects and short reports. Some editors like RStudio (see Section 3) support Sweave and Knitr and offer buttons to easily incorporate so-called chunks – pieces of R code in \texttt{\LaTeX}. Furthermore, these tools have a good documentation and an intuitive handling.

However, tools like Sweave and Knitr also have their drawbacks. Mixing R and \texttt{\LaTeX} code does not necessarily provide all the features that either one provides (there are restrictions). Furthermore, having text mixed between different chunks of code may distract you from coding (typically, text – besides comments – does not help in writing sophisticated code); navigation within the document also does not get easier. Moreover, debugging (that is, searching for errors that appear in some piece of code) is significantly more difficult when mixing R with \texttt{\LaTeX} code. Finally, if a code chunk uses more run time, one has to store the result in an object in order to load it for the next compilation (and thus save time). This is tedious and prone to errors. For example, when the source code is changed but still the formerly generated output object loaded.

Overall, we thus do not recommend to use Sweave or Knitr for large projects, unless

1) there is a significant amount of source code to be displayed in the written companion of a project (which is rarely the case and if so one should rather think about writing a package to make the source code available);

2) the source code runs sufficiently fast (which is typically rather rare as well); and

3) all coauthors’ knowledge about \texttt{\LaTeX} and R is sufficiently advanced.

2.4.6 Be organized

**Directories** In larger projects (and even in some smaller ones) a clear structure of directories and files helps to be structured and reproducible. Data, R source code, \texttt{\LaTeX} source code and figures should be stored in different subdirectories of your project directory. Together with the concise file naming (see Section 2.4.1) a re-entry in the project after several weeks or months is much easier. Baker (2015) suggests the following directories for statistical projects (with extensions from the authors):

- code (\texttt{src}): R-files (if not in a package)
- figures (\texttt{fig}): illustrations and figures produced by R
- raw: raw data sets such as csv’s, txt’s, zipped files, etc.
2 General suggestions

- **data**: If the data is not structured and cleaned in the raw data, the working data should be stored separately. One of the first analysis steps is to structure and clean the data. The results can be stored in `.rda` files. Also aggregated data should be stored in a separate file. This makes it easier and faster for subsequent steps to work with the data.

- **logs**: For time consuming calculations or computations on a cluster, log files may help to debug errors.

- **reports**: short reproducible reports for example produced by Knitr or Rmarkdown

- **test**: unit-tests, function-tests, test scenarios, etc.

- **literature (docs)**: PDF files of the used literature

- **paper or thesis**: LaTeX files

- **old**: containing different stages (outdated files, backups of running scripts etc.) of the various files developed (unless the files are already part of a version control system; see Section 6).

**Separate steps** in the analysis of data should be addressed in separate sections of a script. Much longer source code should even be stored in separated files (with a concise file naming). Common steps in a data analysis are:

- **Cleaning** (handling of missing or incomplete data, duplicates,...);

- **Structuring** (converting the format, setting keys, “tidying” your data, that is, arranging your data such that each column is a variable and each row is an observation);

- **Aggregation** (aggregating over one or more variables to summarize their main characteristics like sums or means);

- **Filtering** (removing all unnecessary columns and rows; for example, shrinking the data to the analyzed time period); and

- **Analysis** (explorative data analysis (EDA), for example, histograms, scatter plots, odd ratios, etc.).

Each step should be separated from the others to simplify reproduction. The results of each step should be stored (at the end of the corresponding step). There is no strict order of the first four steps. They depend on the given data set and problem. They may also iterate.

**Example:**

```
## Required packages
require(data.table)
require(plyr)

## Get data (here: CRAN logs)
start ← as.Date("2015-12-01")
end ← as.Date("2015-12-05")
all_days ← seq(start, end, by="day")
year ← 1900 + as.POSIXlt(all_days)$year
files ← paste0("raw/",all_days,".csv.gz") # store the downloaded files in "raw/"
urls ← paste0("http://cran-logs.rstudio.com/",year,"/",files)
res ← lapply(seq_along(urls), function(i)
  download.file(urls[i], destfile=paste0(all_days[i],".csv.gz")))
stopifnot(res == 0) # sanity check
```
## No cleaning here

## Filter the data (here: quite simple by deleting unused columns)

```r
dat ← dat[, c("date", "package", "ip_id"), with=FALSE] # delete unused columns
```

## Structure data

## Add some keys and define variable types

```r
dat[, date:=as.Date(date)] # convert dates to class Date
dat[, package:=factor(package)] # convert packages to factors
dat[, week:=strftime(as.POSIXlt(date), format="%Y-%W")]) # introduce week
setkey(dat, package, date, week) # sort w.r.t. 'package', 'date', 'week'
```

## Save the data object in the directory "data/"

```r
logfile ← paste0("data/CRAN_logs_",start,"--",end,".rda")
save(dat, file=logfile)
```

## Load the data object (only required in a new R session; shown here on purpose)

```r
load(logfile)
```

## Aggregation

```r
dl ← dat[, length(week), by=package]
colnames(dl) ← c("package", "downloads")
dl ← dl[, order(downloads, decreasing=TRUE), ]
```

## Actual task (display download statistics of the package "mvtnorm")

```r
pkg ← "mvtnorm"
(total ← dl[package==pkg,]) # total downloads in [start, end]
daily ← dat[J(pkg), length(unique(ip_id)), by=c("date", "package")]) # downloads/day
colnames(daily) ← c("date", "package", "downloads")
plot(daily$date, daily$downloads, type="b",
     xlab=paste0("Days (from ",start," to ",end,")"),
     ylab=paste("Number of downloads of package", pkg))
```
but the for-loop involved is too slow. Do you know how the source code attached can be improved in this regard?"

- In the next paragraph of the email, give all the relevant details about the problem.
- Check your email before you send it. Is your problem formulated as a minimal working example? Is the announced attachment attached? Is your question clearly formulated? Did you answer all the questions the supervisor asked you to in her/his last email?
- An unwritten law (at least applying replies to your supervisor) states that emails should be answered within 24h (otherwise one could equally well send a carrier pigeon). Therefore, check and answer your email at least once a day.

Preparing meetings Prepare the questions you have and would like to ask your supervisor or boss. Send them to your supervisor or boss in the same email in which you kindly ask for an appointment. Suggest two possible time spans for the meeting, for example, “Thursday afternoon” or “Monday between 09:00 and 12:00”. Bring a paper and pencil to the meeting. Also, be able to briefly summarize your work or problems (which should be easy since you have already formulated the related questions); your supervisor is usually involved in several projects simultaneously and can not remember all details of your project. The more precisely you can nail down a problem or ask a question, the more likely you will get the answer you were looking for.

During meetings Be awake (!), polite and alert. Do not answer emails or phone calls (switch off your cell phone) during a meeting. Take notes of the answers, comments, suggestions, etc. of your supervisor or boss.

Wrap-up Complete and structure your meeting notes right after the meeting. Put the points you need to act on in your files (.tex or .R) with a string TODO in front of them (this allows you to search for all such points to see whether there is anything left to do in the document). Finally, go through all files again, work on the TODOs, and take notes of the questions that arise (to have them ready for the next meeting).

Feedback and corrections from supervisors Note that your supervisor typically only corrects the first instance of a mistake in a project document. It is your responsibility to completely go through the files and make the corresponding corrections everywhere (which should be easy since you follow Section 2.4.2 above!).

3 Editors and integrated development environments

Why care It seems impossible to overestimate the importance of a good (text) editor for modern software development. Indeed, besides auxiliary programs (such as a PDF viewer, for example), advanced programmers mainly work with a tool accepting lines of commands (the “terminal” on Unix systems), a browser and a good editor. An editor is an application which allows to edit files – one of the major tasks when writing documents or software.

Everybody can use her/his own favorite editor, we will not really recommend one here. However, we will give some suggestions what a good editor should have and how the editor can support and improve our coding style and workflow. Advanced editors can even support several of the style-related points addressed later.

Two sophisticated choices Although many pieces of software now provide their own integrated development environment (IDE), there are some powerful editors that can be used for various
different tasks and thus provide a notion of “economies of scale” for development. Two very sophisticated editors are GNU Emacs and Vim. Both editors go far beyond simple tasks such as syntax highlighting or navigation within files. Their rivalry is known as “editor war”. Both editors are highly customizable and can be further expanded to allow for much more advanced tasks (such as managing files including bookmarks or computing file differences to find changes to backup files, for example, via Emacs’ M-x ediff-buffers). They can partly even act as email program or web browser. Especially for working \LaTeX and R, Emacs is suited well, with the well-developed tools AUCTeX and Emacs Speaks Statistics (ESS). The customizability comes at the price of a rather steep learning curve, though. Although powerful editors such as Emacs or Vim can be recommended to work with in the long run, it takes time to become proficient in using them.

A popular choice for Windows is the free program Notepad++. This is a powerful editor which is (partly) customizable and goes beyond syntax highlighting or navigation within files. Many coding languages are supported and extensions are possible. “Find and replace” or other editing functions are well implemented and can be applied to several files simultaneously. A popular and powerful choice under Mac OS X is TextMate.

**Less powerful but easier to learn choices** For working with \LaTeX and R, there are also specific editors and IDEs available which are comparably easy to use. For \LaTeX, examples are Kile or TeXmaker (primarily for Linux), TextMate or TeXShop (for Mac) and TeXnicCenter (for Windows).

For R, RStudio is quite popular and more or less the standard IDE for beginners, students and practitioners. It is available on Linux, Mac and Windows. It combines R with an editor allowing direct code execution, file directory, tools for plotting, history, help pages and output windows. R packages can be easily installed/updated and loaded. Even whole projects such as packages can be managed in RStudio-projects. Furthermore, RStudio supports Sweave, Markdown and Knitr, see Section 2.4.5, as well as the new shiny web framework for building interactive reports and visualizations using R.

## 4 \LaTeX

We assume the reader to be familiar with basic syntax and usage of \LaTeX. For an introduction, see Oetiker et al. (2011). For \LaTeX packages and other material around \TeX, see [http://www.ctan.org/](http://www.ctan.org/). Typically very good help on more advanced topics can be found on [http://tex.stackexchange.com/](http://tex.stackexchange.com/).

### 4.1 Typographic recommendations for mathematical documents

**Getting help** Although books like Ritter (2002) can provide guidance with many good ideas not mentioned here, keep in mind that by far not all recommendations apply equally well to mathematical or scientific documents or projects.

**Lazy eye principle** To assess whether a document looks good, apply the lazy eye principle: hold the page a meter away from your eyes and try to “see through” (like your grandmother would do without her glasses). Check whether the page structure (including white space, figures, margins etc.) is appealing.

One advice which is often implied by the lazy eye principle is to use headings in heads of propositions, theorems, examples etc. to make it easier to follow the overall golden thread of
the document and thus to distinguish between main and auxiliary results.

**Character protrusion** Use the \textit{\LaTeX} package \texttt{microtype} for character protrusion and font expansion (only with \texttt{pdf\LaTeX}). By stretching line endings with certain characters further out in the margin than others, this provides a visually more appealing justification than by forcing each line to have precisely the same length.

**New paragraphs** Use paragraph indentation (\texttt{parindent}) instead of paragraph skip (\texttt{parskip}). The reason is that in mathematical documents with displayed equations, a paragraph skip is difficult/impossible to distinguish from a vertical space after a displayed equation (which is a problem when a paragraph ends with the latter).

Create a new paragraph by an empty line in your \texttt{.tex} file, not by using \texttt{\par}. Furthermore, before each new (sub)section, use an empty line (except when a new (sub)section directly follows a new (sub)section). This makes it visually easier to navigate through your document. Of course, a good editor and PDF viewer supports forward and \texttt{backward synchronization} for jumping from the \texttt{.tex} file to the corresponding location in the \texttt{.pdf} file and vice versa; this significantly simplifies navigation.

**Title case** If at all, only use title case in the title of projects, not in section headings, table headings etc. Besides inconsistency, there is quite a debate about which words should be capitalized in title case.

**Capitalization** If you refer to a table/figure/theorem in your text use upper case letters, for example “In Figure 2, we illustrate...” or “The proof of Theorem 3 is given in...”. But if you do not refer to a specific numbered environment, use lower case letters, so “In the figure shown below, we illustrate...” or “The proof of the following theorem is given in...”.

**Punctuation** As mentioned before, use punctuation marks, also in displayed mathematical formulas. After all, mathematics is also a language (the language of nature) and thus deserves proper punctuation.

**Abbreviations** The abbreviations “i.e.” (“that is”), “e.g.” (“for example”) and “c.f.” (“see”) are always preceded by a comma (unless, for example, used right after an opening parenthesis of course). They are typically also followed by one. In general, it is considered good style to avoid them completely and write “that is”, “for example” and “see” instead.

**Footnotes** Try to avoid footnotes. They distract from the reading flow, are rarely accepted by scientific journals and can almost always be omitted anyways. Exceptions are the authors’ affiliations on the first page of a scientific paper, for example.

**Introducing new quantities** If you introduce or define a new term or notion verbally, make it visible via \texttt{\textit{...}} and, if you have a longer document with an index (such as a thesis), refer to it in the index.

Always introduce definitions, figures, tables, etc. before they appear in the text. However, do not introduce them too long before they actually appear, rather right before. This is also considered good practice in programming in general. If you define a variable too early, the reader (or even yourself) might have forgotten about it by the time it is used.

**Large numbers** Use \texttt{\,}, to visually separate numbers larger than or equal to 10\textsuperscript{4}, so write 10,\,000, 100,\,000, 1,000,000, etc. for 10\textsuperscript{4}, 100\textsuperscript{4}, 1,000\textsuperscript{4}, etc.

**Page ranges** For page ranges (such as “1–10”), compound names (Poincaré–Sylvester sieve formula; also known as the inclusion–exclusion principle), or dashes, use -- and not just --; the latter is reserved for hyphens!
Sets The positive integers, the real numbers, the complex numbers etc. can be nicely formatted via \texttt{\textbackslash mathbbm{N}}, \texttt{\textbackslash mathbbm{R}}, \texttt{\textbackslash mathbbm{C}} etc. with the \LaTeX{} package \texttt{bbm}.

Parentheses, square brackets and braces Use \texttt{\textbackslash bigl}, \texttt{\textbackslash bigr}, \texttt{\textbackslash Bigl}, \texttt{\textbackslash Bigr}, \texttt{\textbackslash biggl}, \texttt{\textbackslash biggr} and \texttt{\textbackslash Biggl}, \texttt{\textbackslash Biggr} instead of \texttt{\textbackslash left}, \texttt{\textbackslash right} unless they cannot be used easily or you really need larger parentheses; see \url{http://tex.stackexchange.com/questions/12773/or-left-parentheses} and \url{http://tex.stackexchange.com/questions/1454/what-is-the-correct-way-to-do-delimiters}. Also, do not use the unspecified versions \texttt{\textbackslash big} and related commands as they create too much horizontal space; see \url{http://tex.stackexchange.com/questions/1232/difference-between-big-and-bigl}.

Nested parentheses This is a complicated topic and there exists no easy solution. We suggest to (typically) follow the rule: For two subsequent parentheses use the same size, then go to the next larger size; see for example Equation (1) in Section 2.4.2.

The space after a parenthesis In displayed equations, large (typically opening) parentheses may reach into the actual formula. With $\textbackslash$, one can create some additional space; see the following example.

\begin{verbatim}
Bad:
\texttt{biggl(}\sum_{i=1}^n (n \sum_{i=1}^n

Good:
\texttt{biggl(,}\texttt{,sum}_{i=1}^n\texttt{-n (n \sum_{i=1}^n

\end{verbatim}

Labeling As mentioned before, only label those displayed equations etc. that you actually refer to from somewhere in your document (hence a label should indicate a more important equation); this is similar to references (only display those which you refer to from within the text). Do not label every displayed equation by default. The \LaTeX{} package \texttt{refcheck} can help finding unused labels and provides many nice features in the document development phase.

If you do not want to label a certain line in a multi-line equation, use \texttt{\textbackslash notag} (before the line break induced by $\textbackslash\textbackslash$). If you want to change the label, use \texttt{\textbackslash tag\{\$\$\}} right before \texttt{\textbackslash label\{...\}}.

Referring to equations Referring to equations can be done via \texttt{\textbackslash eqref\{eq:label\}} instead of \texttt{\textbackslash ref\{eq:label\}}; the latter version bears the risk of forgetting the adjacent parentheses.

Vectors Vectors are column vectors, but written as a tuple $X = (X_1, \ldots, X_d)$. Note that a transpose sign is only used if required, for example, as in $a^\top X$; use \texttt{\textbackslash \textasciitilde\{	extbackslash\top\}} to generate a transpose sign. Furthermore, use the command \texttt{\textbackslash bm}{} from the \LaTeX{} package \texttt{bm} to generate bold symbols such as vectors; this has the advantage of also working for Greek letters.

Ruler Use the package \texttt{vruler} with the setting \texttt{\setvruler[10pt][1][1][4][1][0pt][0pt][-30pt][\texttt{text\textasciitilde}\texttt{height}]} (or similarly; see the documentation) to display line numbers in your document. This greatly simplifies referring to and discussing certain parts of your document in the development phase.

Quotation marks The \LaTeX{} quotation marks in (American) English start with “ (typically obtained via the key with the tilde symbol) and end with ” (the key with the single quotation marks). Do not use ‘.
4.2 Technical tricks to improve typography

4.2.1 Citations

**How-to** Use **BibTeX**, or – much better – its successor **BibLaTeX**, to manage references and bibliographies in a `.bib` file. There are several free software tools available to organize and manage references for **BibTeX** or **BibLaTeX**, for example JabRef. Emacs’ **AUCTeX** and **RefTeX** also provide functionality for conveniently working with `.bib` files.

**Where to put references** References can often be nicely added at the end of a sentence with a construction of the form “; see, for example, <reference>”.

**Numbers as references** Some journals prefer numbers instead of (Name, Year) as references. In such cases adjust your sentences accordingly. For example: “*f*(x) is defined as ... [3]” instead of “[3] defined *f*(x) as ...”.

4.2.2 Spaces and alignment

**Escaping spaces after dots and avoiding line breaks** If a word, title of a person, or abbreviation ends with a dot, note that **LaTeX** cannot distinguish it from the end of a sentence. **LaTeX** therefore creates a space which is larger than what you actually want. In order to get the correct spacing, you have to escape the space. This can be done using a backslash, for example: As a Ph.D. student, I have to work hard.

Also, some commands require a space to be escaped; for example, \textbf{LaTeX} is powerful produces “**LaTeX** is powerful” whereas \textbf{LaTeX} correctly produces “**LaTeX** is powerful”.

Another instance where one should escape spaces is when referring to figures or tables. In this case one can use a tilde to avoid a line break between the label “Figure” or “Table” and its number: As shown in Table~\ref{foo} and Figure~\ref{bar} dots, where foo and bar refer to the table and figure labels, respectively.

**Breaking terms over lines** If you want to break, for example, a vector $X = (X_1, \ldots, X_d)$ over a line, use $\,$ to allow **LaTeX** to break the line. For example, write $\textbf{bm}(X) = (X_1, \,$ $ \text{dots}, X_d)$ or $\textbf{bm}(X) = (X_1, \text{dots}, X_d)$$. In general, give **LaTeX** more freedom to nicely break lines by creating more (but shorter!) mathematical formulas etc.; see the following example (note in particular the nicer, text-like spacing on the right-hand side). In the same spirit, write $\textbf{bm}(X)_{\text{-}i}$, $i \in \{1, \ldots, d\}$ instead of $\textbf{bm}(X)_{\text{-}i}$, $i \in \{1, \ldots, d\}$.

**Bad:**

$\textbf{bm}(X)_{\text{-}i}$, $i \in \{1, \ldots, d\}$

**Good:**

$\textbf{bm}(X)_{\text{-}i}$, $i \in \{1, \ldots, d\}$

$X_i, i \in \{1, \ldots, d\}$

**Watch out for bold indices** Watch out for the difference between \textbf{X}_{\text{-}i} and \textbf{X}_{\text{-}i}; the former creates a bold index while the latter does not. Bold indices have their own meaning and are typically only used for vectors of indices.

**Horizontal spaces** Use \texttt{\quad} in displayed equations to separate formulas from text or domains from function definitions etc. A greater separator is \texttt{\qquad} often used to separate different columns of displayed equations.

**Use align and alignat** For one-column displayed equations, one can use amsmath’s \texttt{align} environment for both one-line or (possibly aligned) multi-line displayed equations. This has the slight disadvantage of creating vertical space between the last line of some text and the environment
following the text independently of how much this last line of text is filled (furthermore, \qedsymbol is not correctly put when proofs end with an align environment). The advantage is that one can consistently use the align environment independently of the number of lines displayed in the displayed equation. amsmath’s equation environment chooses this vertical space adaptively but only applies to single-line displayed equations.

For multi-column multi-line displayed equations, one can use amsmath’s alignat environment. For more details including why not to use variants such as $$..$$, see http://tex.stackexchange.com/questions/40492/what-are-the-differences-between-align-equation-and-displaymath.

Allow page breaks in displayed equations You can use \allowdisplaybreaks to allow \LaTeX to break multi-line displayed equations over different pages. But this is only recommended on the very last iteration of your document preparation phase. Ideally, it should not be necessary as it is often more natural and readable to separate long align environments into two ore more, with some text in-between.

The powerful phantom command Use \phantom{...} to properly align follow-up lines of displayed equations. For example,

\begin{align*}
f(x) &= \biggl( \Bigl( \bigl( (a_n x + a_{n-1}) x + a_{n-2} \bigr) x + a_{n-3} \bigl) x + a_{n-4} \Bigr) x + a_{n-5} \biggr) x + a_{n-6} x + a_{n-7} x + \dots.
\end{align*}

shows a properly vertically aligned second line:

\[
f(x) = \left( \left( \left( (a_n x + a_{n-1}) x + a_{n-2} \right) x + a_{n-3} \right) x + a_{n-4} \right) x + a_{n-5}.\]

Note that the {} around the equality sign within the \phantom command represents an empty math object and thus properly replicates the (larger) space around such signs or symbols in math mode; in certain situations, only \phantom{=} might be required. In general, do not work with \hspace{} in these situations to achieve vertical alignment.

4.2.3 Figures (and tables)

Template for including (side-by-side) figures For including two figures side-by-side, one can use a construction of the following form.

\begin{figure}[htbp]
  \centering
  \includegraphics[width=0.48\textwidth]{my_figure_1_without_ending}
  \includegraphics[width=0.48\textwidth]{my_figure_2_without_ending}
  \caption{The plots show \dots (left) and \dots (right).}
  \label{fig:label}
\end{figure}

For including just one figure, omit \hfill and the obvious second \includegraphics command. Similarly, for two tables side-by-side, one can use one embracing table environment containing
two tabular environments. In general, we recommend to use the \LaTeX\ packages \texttt{siunitx} (for units and alignment of table entries), \texttt{multirow} (for table entries spanning several rows) and, especially, \texttt{booktabs} (for its paradigm and visual guidance) to create tables; however, note that a figure or graph typically conveys much more information. Sideways figures and tables can be produced with the \LaTeX\ package \texttt{rotating}, but a nicer solution is actually to temporarily change the paper to landscape format to obtain readable figures or tables.

For some (separate) text and a figure side-by-side, one can use the following.

\begin{minipage}[t]{0.48\textwidth}
\vspace{0mm}
\includegraphics[width=\textwidth]{my_figure_without_ending}
\end{minipage}
\hfill
\begin{minipage}[t]{0.48\textwidth}
\vspace{0mm}
Some text
\end{minipage}

This is useful on slides, for example. If the figure (or table) shall be embedded in a longer text, however, one can use the \LaTeX\ package \texttt{wrapfig}. Use this with care, as this will not look nice if the figure is pushed beyond the end of a page.

\textbf{\LaTeX\ graphics} One can draw graphics directly within \LaTeX. The package \texttt{TikZ} of Tantau (2013) can produce portable graphics in both PDF and PostScript formats. It comes with very good documentation and an extensive collection of examples is available on \url{http://www.texample.net/tikz/}.

Example for drawing an undirected tree with edge labels:

\begin{verbatim}
\usepackage{tikz}
\usetikzlibrary{shapes,snakes}
\usepackage[active,tightpage,psfixbb]{preview}
\PreviewEnvironment{pgfpicture}
\setlength{\PreviewBorder}{0pt}
\begin{document}
\begin{figure}[ht]
\centering
\begin{tikzpicture}
\tikzstyle{place}=[ellipse,draw=black,thick,inner sep=0pt,minimum size=8mm]
\node[style={text width=2cm}] (Tree1) at (2.5,1) {$T_1$};
\node[place] (N2) at ( 0,0) {2};
\node[place] (N1) at ( 2,-2) {1};
\node[place] (N3) at ( 4,0) {3};
\node[place] (N4) at ( 0,-4) {4};
\node[place] (N5) at ( 4,-4) {5};
\draw [-,thick] (N2) -- (N1) node [sloped,midway,above] {1,2};
\draw [-,thick] (N1) -- (N3) node [sloped,midway,above] {1,3};
\draw [-,thick] (N1) -- (N4) node [sloped,midway,above] {1,4};
\draw [-,thick] (N4) -- (N5) node [sloped,midway,above] {4,5};
\end{tikzpicture}
\end{figure}
\end{document}
\end{verbatim}
Editing eps figures with \LaTeX\ The package \texttt{psfrag} allows to overlay Encapsulated PostScript (EPS) figures with arbitrary \LaTeX\ constructions. This may be helpful to add simple equations or Greek letters to a title, axis or label. A text tag in the eps will be replace by the specified \LaTeX\ construction.

4.2.4 Miscellaneous

Short versions of commands The command \texttt{\ldots} can often be replaced by \texttt{\ldots}, for example, $X_1, \ldots, X_d$ correctly produces $X_1, \ldots, X_d$. Also, use \texttt{\le} and \texttt{\ge} instead of \texttt{\leq} and \texttt{\geq}, respectively.

Easier to read letter l Use \texttt{\ell} (displayed as ℓ) instead of \texttt{l} (displayed as l) for the log-likelihood; the latter is typically only used for indices.

Emphasize Emphasize text using the \LaTeX\ command \texttt{\emph} (instead of \texttt{\textit}). Also, do not use \texttt{\underline}.

Create your own \LaTeX\ commands If you need a certain \LaTeX\ command several times in your document you can code your own \LaTeX\ commands for short cuts.

Example:

```latex
1 \newcommand*{\TODO}[1]{\textcolor{red}{TODO #1}} % TODO command with argument
2 \newcommand*{\IR}{\mathbb{R}} % the real numbers
3 \newcommand*{\Exp}{\operatorname{Exp}} % the exponential distribution
4 \newcommand*{\I}{\mathbbm{1}} % indicator function
5 \newcommand*{\R}{\textsf{R}} % statistical software R (yes, it’s a sans serif font!)
```

Commands come in handy when notational changes have to be made at a later editing stage. But use them only for those notations which are frequently used in the text.

The package \texttt{algorithm} can be used for displaying pseudo code. It supports automatic indenting, line numbers, keyword highlighting, labels and much more. Here is an example from the manual Brito (2009, p.14):

```latex
1 \begin{algorithm} % enter the algorithm environment
2 \caption{Calculate $y = x^n$} % give the algorithm a caption
3 \label{alg1} % and a label for \ref{} commands later in the document
4 \begin{algorithmic} % enter the algorithmic environment
5 \REQUIRE $n \geq 0 \lor x \neq 0$
6 \ENSURE $y = x^n$
```
It gives the following output:

Algorithm 1 Calculate $y = x^n$

Require: $n \geq 0 \lor x \neq 0$

Ensure: $y = x^n$

```latex
\begin{algorithm}
  \STATE $y \leftarrow 1$
  \IF{$n < 0$}
    \STATE $X \leftarrow 1 / x$
    \STATE $N \leftarrow -n$
  \ELSE
    \STATE $X \leftarrow x$
    \STATE $N \leftarrow n$
  \ENDIF
  \WHILE{$N \neq 0$}
    \IF{$N$ is even}
      \STATE $X \leftarrow X \times X$
      \STATE $N \leftarrow N / 2$
    \ELSE[$N$ is odd]
      \STATE $y \leftarrow y \times X$
      \STATE $N \leftarrow N - 1$
    \ENDIF
  \ENDWHILE
\end{algorithm}
```

Note that an algorithm is a sequence of unique statements; make sure you leave no room for (mis)interpretation!

5 R

R, see [www.r-project.org/about.html](http://www.r-project.org/about.html), is a free software environment for statistical computing and graphics. This combination of focus on statistics and providing graphics is one of the
many strengths of R. By being open source and providing tools for package development, many
people have contributed to the usage of R for virtually all statistical tasks by providing packages.
Furthermore, new research results in the statistical community are often published together with
new or further improved R packages.

This part of our guidelines covers statistical software development in R. By software development
we do not mean writing R packages, but rather code snippets or scripts (.R files) in “good shape”,
which could be served as a basis for packages or which could be sent to package maintainers
(without them getting headaches and nightmares from looking at your code). Many of the points
addressed are also valid for other programming or script languages like C, C++ or MATLAB.
The general goal of this chapter is to help you to write code which is easy to read, efficient, not
too bad to be distributed and reproducible.

Many of the following rules or recommendations are not strict and come from our own background
and reflect our experiences and opinions. They are biased. You can find several guidelines for R
in the web addressing different topics of programming. Here we collected the most important
(to us) points. In particular, we want to mention the keynote speech “Good Practices in R
Programming” of Martin Mächler at the useR conference 2014 in Los Angeles, USA. The slides
are available at http://stat.ethz.ch/Teaching/maschler/R/useR_2014/ and the video at
https://www.youtube.com/watch?v=ytbX-T1A8wE. Several of his “rules” can be found in the
following sections.

5.1 Getting started

We assume the reader to be familiar with basic syntax and usage of R. For an introduction,
see, for example, Venables et al. (2012). For R packages and other material around R, see
http://cran.r-project.org/.

Getting help There are nowadays many mailing lists, forums, blogs, etc. available for obtaining
help on how to use R. For general R related questions, https://stat.ethz.ch/mailman/
listinfo/r-help is one of the major mailing lists. Also, http://stackoverflow.com/ with
tags for R provides a good contact point with useful answers typically within a short period
of time. For more specific questions such as platform-dependent or topic-dependent, see
the special mailing lists on http://www.r-project.org/mail.html, such as https://stat.
ethz.ch/mailman/options/r-sig-hpc/ for high performance computing. Furthermore, see
http://www.rseek.org/ for searching R related sites, help files, manuals, mailing list archives
etc.

Installing packages There are various ways to install R packages, the most common are:

from CRAN, the Comprehensive R Archive Network (CRAN); see http://cran.r-project.
org/. This is the most typical way to install R packages.

install.packages("myPkg")

Note that "myPkg" can also be a vector of packages, so c("myPkg1", "myPkg2").

from R-Forge, a central platform for the development of R packages, R-related software and
further projects; see https://r-forge.r-project.org/. If a package is developed on
R-Forge, then the latest version is available there (uploads to CRAN are typically only made
every once in a while). This means that if you ask a package maintainer for a change in a
package (which is developed on R-Forge; many packages are), you most likely have to install
the package from R-Forge to get the desired change.
install.packages("myPkg", repos="http://R-Forge.R-project.org")

from .tar.gz This is source code. Windows or Mac need pre-compiled code. How to produce pre-compiled code from source see for example http://www-m4.ma.tum.de/en/teaching/theses/r-package-manual/.

install.packages("~/my/folder/myPkg.tar.gz", repos=NULL)

from GitHub GitHub is a powerful online collaboration platform for code review and code management for open source and private projects, see https://github.com/. To installs packages from GitHub use

require(devtools)
install_github("myPkg")

5.2 Documentation
5.2.1 Citing R and R packages

Many volunteers have invested a lot of time and effort in creating R and R packages, please cite R and the packages you use for data analysis. Use the citation() command to cite R or R packages. To cite R itself, citation() provides a plain text references and a BibTeX entry. For R packages, use citation("pkgname"), where pkgname is the name of the R package to be cited. For example

require(VineCopula)
citation("VineCopula")
gives


A BibTeX entry for LaTeX users is

@Manual{,
  title = {VineCopula: Statistical inference of vine copulas},
  author = {Ulf Schepsmeier and Jakob Stoeber and Eike Christian Brechmann and Benedikt Graeler},
  year = {2013},
  note = {R package version 1.2-1},
}

Here is an example with a list of entries:

require(copula)
(ci <- citation("copula"))
ci[1] # including BibTeX entry; see also toBibtex(ci)
gives

To cite the R package copula in publications use:


23
5.2.2 Run time information

In many statistical projects one compares different methods, models, algorithms or just different variations of the former. Beside statistical measures often run time informations are given. Whenever you state run times of your algorithm name the software, for example, R and R-packages, and the machine you used for your calculations. State all necessary information for a possible rerun. Also do not forget to give the time unit, usually seconds (short sec). Here an example form Schepsmeier (2015):

In all of the forthcoming simulation studies we used \( B=2500 \) replications and the number of observations were chosen to be \( n=500, n=750, n=1000 \) or \( n=2000 \). As model dimension we chose \( d=5 \) and \( d=8 \) and the critical level \( \alpha \) is \( 0.05 \). As before all calculations are performed using the statistical software \( \texttt{R} \) and the \texttt{R}-package \texttt{VineCopula} of \cite{VineCopula}. ...

Of cause the computation time for the different proposed GOF tests is also a point of interest for practical applications. Therefore, in Table \ref{tab:Summary} the computation times in seconds for the different methods run on a Intel(R) Core(TM) i5-2450M CPU @ 2.50GHz computer for \( n=1000 \) are given alongside with a summary of our findings.
5.2.3 Code documentation

The documentation of your code is one of the most important tasks in the software development. It enables other users, maintainers or your supervisor to follow your ideas of coding and allow for easy application. Even you self will profit from a proper documentation.

There are two ways to document a code - in the coding itself and externally in extra files. While the first one is absolute necessary the second one is optional and depends on the scale of the project and the demands of your supervisor. External files are usually needed in R-packages and are more extensive in their description, giving for example additional explanations on the statistics and simple application examples.

Internal documentation

Comments Writing comments (as explanations, for example, or to point out the mathematical calculations behind the scenes) is good. In R, the # symbol can be used to start a comment. The following example code shows the usage of comments (forget about the meaning of the other parts, just look for the comments).

```r
### fast rejection algorithm, R version ########################################################
##' Sample a vector of random variates St \sim \tilde{S}(\alpha, 1,
##' (\cos(\alpha \pi/2) V_0)^{1/\alpha}, V_0 I_{\alpha = 1},
##' h*I_{\alpha \neq 1}; 1) with LS transform
##' exp(-V_0((h+t)^\alpha-h^\alpha)) with the fast rejection
##' algorithm; see Nolan’s book for the parametrization
##'
##' @title Sampling an exponentially tilted stable distribution
##' @param alpha parameter in (0,1]
##' @param V0 vector of random variates
##' @param h non-negative real number
##' @return vector of variates St

retstableR <- function(alpha, V0, h=1) {
  stopifnot(is.numeric(alpha), length(alpha) == 1,
            0 <= alpha, alpha <= 1) # alpha > 1 => cos(pi/2 *alpha) < 0
  n <- length(V0)
  if(alpha == 1 || n == 0) return(V0) # alpha == 1 => point mass at V0
  if(alpha != 1) call fast rejection algorithm with optimal m
  m <- m.opt.retst(V0)
  mapply(retstablerej, m=m, V0=V0, alpha=alpha)
}
```

Note the difference between inline comments (comments for a statement in a single line; starting with #), comments addressing several lines of code (on a new line right before the corresponding chunk, starting with ##) and comments separating larger parts of code (starting with ###; typically only used for much larger code chunks or to visually separate different functions or other bigger parts in an R script).

The comments starting with ##' are part of a certain way of documenting functions called Roxygen documentation. One first starts with a short explanation what the function computes. After a blank line, a one-line title (starting with @title) giving the main purpose of the function
is provided. Then, explanations for all arguments of the function follow (by @param), explaining
the types of the corresponding arguments. The return value of the function is given via
@return, followed by the author(s) of the function (@author); additionally, a @note may follow.
Roxygen documentation can directly be converted to a help file for an R package containing
the corresponding function, although help files typically contain much more information (such
as example calls; see also the R-package roxygen2 of Wickham et al. (2015)).

External Files (FILE, .Rd, .pdf)

FILE General description of the source code or package. Also special dependencies on other
packages or required software such as gsl should be explained in such files. Typical naming
convention: DESCRIPTION, README, INSTALL (without endings).

.Rd Help files in R packages. The coding is adapted from L\LaTeX but is different.

.pdf Manuals generated from the help files or vignettes.

5.3 Programming style

5.3.1 Writing correct source code

Ranges of numbers Let n be an integer. It is convenient to write 1:n for the sequence of numbers
from 1 to n. However, use this only if you are absolutely sure that n is greater than or equal to 1.
Often, for n less than 1, one would expect the empty sequence, for example in a for(i in 1:n)
loop. To get this behavior, write for(i in seq_len(n)) instead.

if and else Note that else has to follow the closing brace of an if statement on the same line.

\begin{tabular}{ll}
\textbf{Bad:} & \textbf{Good:} \\
1 res <- if(x > 0) { & 1 res <- if(x > 0) { \\
2 "positive" & 2 "positive" \\
3 } & 3 } else { \\
4 else { & 4 "non-positive" \\
5 "non-positive" & 5 } \\
6 } &
\end{tabular}

Let us remark here that if() itself is a function, so we can assign its return value to a variable.

5.3.2 Writing readable source code

Even before writing efficient code, it is important to write readable and structured code. This
significantly improves debugging but also avoids making programming errors in the first place.

80 characters rule Note that lines should contain less than or equal to 80 characters, the only
exception being strings, which should not be broken over lines. This is the typical rule for
editors, terminal emulators, printers, debuggers etc.

Assignment operator In variable assignments, use x <- 1 instead of x = 1, except for arguments
in function calls.

Omit useless code In R statements do not have to end with a semi-colon, so omit it; semi-colons
are only used to separate two statements on the same line, which is rather rarely useful.

Another such example is return() if used in the last line of a function body. It can be omitted;
see Section 5.3.4.
As an exception, write 0.1 instead of .1 (although the 0 can be omitted here). We agreed on being as close to mathematical notation as possible (which votes in favor of 0.1) and the eye also can not accidentally read this number as 1.

**Variable and function names** As mentioned in Section 2.4.1, variables and functions should have meaningful names and should be shorter the more often they are needed (the latter applying at least to variables).

Use dots and lowercase letters for variables (for example, my.variable.name <-) and arguments of functions (see, for example the arguments of pnorm()). On the contrary, in function names, dots are typically (only) used to separate the method name from the class name when declaring S3 methods; see plot.default(), the default method for the function plot(). In function names, rather use underscores to separate parts of the function names, so my_function <- function() {}; some prefer camel-case naming, (but) see above.

Omit superfluous parts, for example, a function length0fObject(object) which determines the length of a given object is preferably to be called just length, since, when called, length(object) immediately reveals what the return value is.

Functions returning a boolean can often be named starting with is, for example isPos <- function(x) x > 0 determines whether a given object x is positive (note that isPos() is also vectorized!). Do not use names like isNotPos(), since it is not immediately clear what the doubly-negated !isNotPos() means.

Other useful prefixes are get... or find..., or also n... (for example, nPts() for determining the number of points).

**Specify arguments** When calling a function, specify the argument names for all except the first argument.

<table>
<thead>
<tr>
<th>Bad:</th>
<th>Good:</th>
</tr>
</thead>
<tbody>
<tr>
<td>rnorm(10, 3, 2)</td>
<td>rnorm(10, mean=3, sd=2)</td>
</tr>
</tbody>
</table>

**Indentation** Indent your code. If your editor does not provide automatic code indentation for R, use four spaces per level of indentation as a rule of thumb. Do not use tabulators.

As an example, consider:

<table>
<thead>
<tr>
<th>Bad:</th>
<th>Good:</th>
</tr>
</thead>
<tbody>
<tr>
<td># No comments, no (clear) indentation</td>
<td>if(a &lt; 0) { # first level</td>
</tr>
<tr>
<td># no new line after closing bracket</td>
<td>b ← 1</td>
</tr>
<tr>
<td>if(a&lt;0){</td>
<td>d ← 2</td>
</tr>
<tr>
<td>b ← 1</td>
<td>} else {</td>
</tr>
<tr>
<td>d ← 2</td>
<td>if(e == 0) { # second level</td>
</tr>
<tr>
<td>}else{</td>
<td>b ← 0</td>
</tr>
<tr>
<td>if(e == 0) {</td>
<td>d ← 3</td>
</tr>
<tr>
<td>b ← 0</td>
<td>} else {</td>
</tr>
<tr>
<td>d ← 3</td>
<td>b ← 2</td>
</tr>
<tr>
<td>}else{</td>
<td>d ← 2</td>
</tr>
<tr>
<td>b ← 2</td>
<td>}</td>
</tr>
<tr>
<td>d ← 2</td>
<td>}</td>
</tr>
<tr>
<td>}} bd ← c(b, d)</td>
<td>bd ← c(b, d)</td>
</tr>
</tbody>
</table>

Besides the indentation, there are various interesting points here. if() is indeed also a function. This feature can be used to write much more readable code (in complicated nested if-statements).
Our output here are two values, $b$ and $d$. By putting them in one vector, each if/else statement has only one return value (namely the vector containing the values of $b$ and $d$). For one-line functions one can omit the curly braces {}, thus we can save space. Furthermore, we can directly assigning the return value of `if()` to a variable. Taking these tricks together, we obtain the following compact form or our code (note that we now only have a single line and the human eye directly detects the assignment at the beginning, knowing that this is the quantity which is defined here – which is not obvious by looking at the above version, where each line starts with `if()` or `else()` but which does not directly reveal which quantities are defined).

```
bd ← if(a < 0) c(1, 2) else if(e == 0) c(0, 3) else c(2, 2)
```

But note that you should use one-line statements only for easy statements such as here, where the terms involved are not too sophisticated and thus the risk of not immediately understanding the statement is minimal.

Finally, let us remark that, normally, the opening brace of a function is put at the end of the line containing the function head (see the above example with `if()` involving braces). For longer function bodies, the opening brace can also be put on a new line in the same column as the closing brace after the function body, so that one can more easily identify code blocks.

**Vertical alignment of code** Assignments which belong semantically together can be vertically aligned as follows:

```
Good:
1 n ← 100
2 set.seed(271)
3 stnrm ← lapply(1:10, function(i) rnorm(i))
4 t4 ← lapply(1:10, function(i) rt(i, df=4)) # vertically align the '\textless \textasciitilde\textgreater'
```

**Spaces around operators** Place one space before and after binary operators (<-, +, -, etc.) and after a comma. As an exception, for arguments of functions, use a space only to separate the arguments, not for the assignment.

```
Bad:
1 x←1
2 optimize(function(x) x*x,interval = c(1, 10),maximum = TRUE)
```

```
Good:
1 x ← 1
2 optimize(function(x) x*x, interval=c(1, 10), maximum=TRUE)
```

**Hint.** The R-package `formatR` of Xie (2015a) can help to transform your code into more readable code. For example it checks for indentation, spaces around operators and assignment operators.

### 5.3.3 Test your source code

Write (small) testing examples to verify your code. Write examples for each sub-function and each auxiliary function. Test different input parameters, test critical parameters, test false user input, test run times etc.; see also Sections 5.5.3 on missing values the handling of warnings and errors.

For advanced users: Create your own package and use the auto-testing functionality of R (via `R CMD check`). An R-package can be created and checked easily in the R-environment software RStudio or by hand, see R Development Core Team (2006).

There are also some R-packages available for unit testing, for example `RUnit`, `testthat`. 

---

28
5.3.4 Writing safe, fast, flexible and sophisticated functions

Write (auxiliary) functions R is about writing functions. Everything is an object or a function in R. If a sufficiently large block of lines appears several times in your code, outsource it, write a function which calculates this block of code. If, at some point, you are required to improve/optimize that part of the code, you do not have to do it several times. Furthermore, having a separate “black box” in terms of a function improves readability of the code. Moreover, the function can be tested individually for correctness or debugged in case of an error.

Check user input for errors At least when writing larger, important functions with several arguments, check user input for validity (rule of thumb: the larger the run time of the function, the more time can be spend on checking inputs). This can often be done with the functions stopifnot() or stop() which may return useful error information to the user in case an argument is not valid. Some books suggest here to imagine the most incompetent user, but it is hardly possible to check all possible user inputs for correctness. Try to catch the most important ones which may lead to a crashes or wrong return values of your function.

Hint. The R-package assertive of Cotton (2015) offers a multitude of suitable functions to check for user input. For example, for the correct variable type: is_numeric(), is_character(), is_data.frame(); for correct variable length: is_scalar(), is_empty(), is_non_empty(), is_of_length()

Warnings Use warning() if an input is correct but not very meaningful (or if a function is deprecated etc.). Warnings can also be useful for accompanying outputs of functions, for example if an algorithm does not converge but stops after a maximum number of iteration steps it is advisable to inform the user (which could, of course, also be done by returning a corresponding status).

Note that warnings are not errors! A warning is nothing bad!

NA, NaN, Inf and other limiting cases R functions typically deal well with “limiting cases” such as NA, NaN, Inf etc. If possible, make sure that the functions you write can also correctly handle such limiting cases. To find NAs do not use x == NA but is.na(x).

Remark: is.na(x) checks if x is NA or NaN, while is.nan(x) checks only for NaN.

Hint. To this end, the R-package assertive also offers some useful functions. Especially the opposite checks may be useful, such as is_not_na() or is_not_null().

Hint. Also useful may be the following functions to check scalars or vectors to be infinite or negative infinite. Note identical(x, Inf)) do not work for vectors.

```
1 isInf ← function(x) x > 0 & is.infinite(x) # check for Inf
2 isNInf ← function(x) x < 0 & is.infinite(x) # check for -Inf
```

Bad:
```
1 if(x == NA){
2   x ← 10^(-100)
3 } else {
4   do.something()
5 }
```

Good:
```
1 if(is.na(x)){
2   stop("x is not available.")
3 } else {
4   do.something()
5 }
```

Never (ever) truncate the return value of a function at an arbitrary large (hard-coded) value such as 10^{100}, just that the function returns something finite. This is never good enough for all inputs. Rather think about why the return value is so large, does it naturally tend to infinity for certain inputs, can we find a good approximation near the limit, or can we implement a
proper logarithm (not just the logarithmic value of the function – all accuracy would be lost!) to be able to do calculations on a more moderate scale?

**Parallelize, Matricize** If you (easily) can, write functions which do not only operate on single numbers, but also on vectors or matrices. This way, such functions will typically be much faster when called, for example, with a vector instead of calling it for each element of the vector.

**Avoid for, while and repeat loops if possible** If possible, avoid loops (for, while, repeat), as they are comparably slow in R. The main point is that they are very rarely needed (one example being a rejection algorithm, but if speed really matters one can also implement it in C). Rather work with the functions `sapply()`, `lapply()`, `vapply()`, `apply()`, `mapply()`, or `tapply()`. It is by no means possible to explain all of them here, but here is a basic example how a for loop can be avoided:

**Bad:**
```r
def = numeric(100)
for(i in 1:100) x[i] = i*i  # first creating and then filling an object```

**Good:**
```r
def = sapply(1:100, function(i) i*i)  # directly create the correct object
def = unlist(lapply(1:100, function(i) i*i))  # typically faster  
def = vapply(1:100, function(i) i*i, NA_real_ )  # typically fastest```

**Return all (useful) results** Computations are expensive, do not throw away other calculated results just because you are not interested in them at the moment. In contrast, return all (useful) results/quantities computed, especially if the computations are time consuming.

For example, if your function computes an optimum via `optim()`, do not just return the optimum, return all related computed values (for example, the convergence status). Note that `optim()` itself follows this paradigm. You never know if you or a user may need it.

**Bad:**
```r
def = function(x)
  optim(c(-1.2, 1), fn=function(z) x * (z[2]-z[1]^2)^2 + (1-z[1])^2)$par```

**Good:**
```r
def = function(x)
  optim(c(-1.2, 1), fn=function(z) x * (z[2]-z[1]^2)^2 + (1-z[1])^2)```

As another example, if you are interested in the behavior of an estimator, do not just return the average over all estimators computed in your simulation study, return the estimators themselves (and actually even more results such as whether there were warnings or errors during the computation, run time etc.; see Hofert and Mächler (2014)). This allows you to look at histograms, box plots etc., which provides much more information than just the mean.

**Arguments with defaults** We can make the function `def()` above more flexible.

**Bad:**
```r
def = function(x, init=c(-1.2, 1))
  optim(init, fn=function(z) x * (z[2]-z[1]^2)^2 + (1-z[1])^2)```

**Good:**
```r
def = function(x)
  optim(c(-1.2, 1), fn=function(z) x * (z[2]-z[1]^2)^2 + (1-z[1])^2)
```

Instead of using the vector `c(-1.2, 1)` “hard-coded” in the function body, we make it an additional parameter, with default value `c(-1.2, 1)`. This way, `def()` can be called with one argument `x` as before, but one has the flexibility of using a different vector `init` if required. Note that it is sometimes advisable not to give a formal argument a default value. This way, a user is forced to think about a reasonable value.
Note that arguments with default values should come after arguments without default values.

**Ellipsis argument** Sometimes, you want to write a function, such as `myOpt()` above, which calls another function, such as `optim()` above, which itself has many arguments. One way of passing arguments from `myOpt()` to `optim()` is via the *ellipsis argument*. Incorporating it in the above example, we obtain:

```r
myOpt ← function(x, init=c(-1.2, 1), ...) 
  optim(init, fn=function(z) x * (z[2]-z[1]^2)^2 + (1-z[1])^2, ...) 
```

We can now call `myOpt()` with, for example, `method="BFGS"`, which is then passed to `optim()`, so `myOpt(100, method="BFGS")` computes the optimum with the "BFGS".

**Return value** A function should always return an object of the same type, no matter what the input is (so, for example, do not write a function which returns either a character string or a number depending on the input).

Furthermore, separate computations from graphics. A function should either compute some values or create/plot a graphic, do not plot something and also return the plotted values in the same function. Note that a function which produces a base graphic (such as obtained with `plot()`) typically has `invisible()` as last line, that is, return value.

The last statement/line of a function is used as return value by default, you do not have to write `return(res)`, just `res` will return the result `res` already. `return()` is typically only used when a function should be exited before the last line.

### 5.3.5 Learn from others, learn from the masters

As we already announced in Section 2.2 search for already existing solutions for your problem. For R-programming this means look for R-packages providing your wanted functionality. Use it! Use their knowledge, use their programming skills and learn from it.

Learn from others, learn from the masters and read the source code of packages. This is the major advantage of open source. The code is publicly available. Therefore, download the source package from CRAN, R-forge or GitHub. Only in the source package (.tar.gz) the functions are not pre-compiled. Read the code, R, C, C++ and/or Fortran, and find your specific function. Read the function(s) closely and reveal the hidden functionality or special treatment of critical input parameters. Experiment with the function(s), run the auxiliary functions separately and most important learn form it! Learn how to structure functions, learn how to write code, learn how others provide solutions.

Furthermore, you can reveal what the software does exactly. And you can change some code to implement a modification of an algorithm or fix a bug. For further reading, see Ligges (2006).

**Note:** The R source of an R-package (in source state) is inside `<pkg>/R/*.R`, and not what you get when you display the function in R (by typing its name).

### 5.3.6 Specific hints

**Do not grow objects** Define an object in advance and specify its length/dimension. Do not let an object grow if it is not necessary. For example, if you can not avoid a `for` loop.
5.4 Tables and graphics

Tables and graphics are a topic of their own and we could already fill a whole script with guidelines and tips with them. Both are used in manuscripts like theses or scientific papers to illustrate results and findings. In what follows we collect some important points to consider when creating tables or graphics in R.

Graphics instead of tables Instead of tables containing a myriad of numbers, use plots to graphically display your results; see also Hofert and Mächler (2014). The reason is that the human eye is not able to compare more than two or three numbers at a time. Graphics typically reveal much more information about the underlying laws and make it easier to see “structure”. In most cases they even allow to save space in comparison to tables. Also, when preparing a presentation, graphics are much easier to grasp within a short period of time than tables (we have all seen slides with a myriad of numbers on them, accompanied with the speaker saying “...as it becomes clear from these results”, switching to the next slide before one even has a chance to look at more than three numbers!).

It is clear that a table is significantly easier to create, but a graphic has another advantage. A plot has the potential to reveal numerical problems as well, something often overlooked when the results are only displayed in tables.

Rounding If you still decide for presenting your results in a table, carefully think about how to display the numbers, for example, how many digits to use. Usually few digits are enough. Using too many digits gives the impression of high precision which might not be adequate because of numerical issues, see Wainer (1993).

If we wish to report results with two digits we need the standard error of this estimated proposition to be \( \leq 0.005 \) (\( 1.96 \times 0.005 \approx 0.01 \)). Thus the standard error of a reported result should be reported too, so a reader can gauge the accuracy.

Also, at least in every column, use the same number of digits and align the numbers according to their decimal point. Important results can be highlighted, for example in bold.

Hint. To export tables from R to \( \LaTeX \) one can use the R package xtable of Dahl (2014), which creates \( \LaTeX \) code from R; see also Hofert and Mächler (2014).

Detecting and distinguishing different lines/points In graphics, do not use too light colors, as they are typically difficult to detect, especially in presentations. Ideally, create a graphic in

Use TRUE and FALSE, not ‘T’ and ‘F’ It can cause conflicts.

\[
\begin{aligned}
\text{Bad:} & \quad \text{Good:} \\
1 \quad \text{rmat} & \leftarrow \text{NULL} & \quad 1 \quad \text{rmat} & \leftarrow \text{matrix}(0., \text{n}, \text{k}) \\
2 \quad \text{for( i in 1:n) } & \{} & \quad 2 \quad \text{for( i in 1:n) } & \{} \\
3 \quad \quad \text{rmat} & \leftarrow \text{rbind(rmat, some.further.} & \quad 3 \quad \text{rmat}[i, ] & \leftarrow \text{some.further.computation.} \\
\quad \quad \quad \quad \quad \text{computation.depending.on.i)} & \quad \quad \quad \quad \quad \text{depending.on.i} \\
4 \quad \} & \} \\
\end{aligned}
\]

Since T is zero the function will return a result with \( \log = \text{FALSE} \) (\( 0 \equiv \text{FALSE}, \ 1 \equiv \text{TRUE} \)).
such a way that the results are still readable when printed in black/white. Also, use colors suitable for color-blind people.

Other options to distinguish different lines (or points) are by using different line types (see lty), line widths (see lwd), or different symbols (see pch). Since each illustration should be self-explanatory, use legends and, of course, label axes, sometimes also a title or sub-title can be useful (if not, a suitable caption can be given in \LaTeX{} to place such pieces of information).

**Label sizes** Use sufficiently large plot axis/legend labels or titles. By default, they are often too small in R. When plotting to a .pdf file, this can typically be solved by choosing a smaller default width and/or height parameter when opening the PDF device via pdf(...); for example, pdf(..., width=6, height=6) for square plotting regions and pdf(..., width=10, height=6) for rectangular ones.

**Cropping white space around figures** Crop the white margins of PDF files before putting them into your .tex document. This is important for their correct alignment in \LaTeX{} and such that no space is wasted and a large area is covered with the graphic of interest.

The function dev.off.pdf() of the R package simsalapar of Hofert and Maechler (2015) may help in cropping white space. It is based on the Unix tool pdfcrop. You can also use the latter manually in the shell or use any other program of your choice to crop white space around the margins of a plot (note that there are also settings in R to do that, but one can not crop the white space perfectly/maximally, at least not with a trial-and-error procedure for each plot individually).

**Base, lattice, ggplot2, grid graphics** Besides base graphics, there are some other options, including lattice graphics (based on the R package lattice; see Sarkar (2008)), ggplot2 graphics (based on the R packages ggplot2; see Wickham (2009)), or, the more low-level grid graphics engine for designing your own graphics functions or modifying existing ones. We recommend to work with base graphics and, for very special purposes (such as three-dimensional wire-frame or cloud plots), use lattice.

## 5.5 Data handling

Statistics is all about data. Data are the basis of our research, our model building process and conclusions we draw. They enable us to answer questions coming from economics, medicine, geology, hydrology, etc. We should thus be careful about data and their handling. Look at your data carefully! What kind of data do you have? Do you have enough data? Is the data set complete? etc.

In this section we would like to wake awareness for correct data handling and how this can be achieved in R. The following points are taken partly from Quick-R (http://www.statmethods.net/input/) and the course slides of Stefan Haug and Ulf Schepsmeier for the Graduate School course “Using R for statistical data analysis I” at TU Munich.

### 5.5.1 Import and export of data

**Read data from a file** The first step is to import data into R. R allows for many different data file formats. There are five functions to read a file in table format and create a data frame from it: read.table(), read.csv(), read.csv2(), read.delim() and read.delim2(). read.csv() and read.csv2() are identical to read.table() except for the defaults. They are intended for reading “comma separated value” files (.csv). Similarly, read.delim() and read.delim2() are for reading delimited files, defaulting to the TAB character for the delimiter. The functions
mainly differ by their field separator and the decimal point symbol. These functions return an object of class `data.frame`. 

Look at the data before and after reading it into R (header, decimal symbol, missing values). Furthermore, R will read all columns as character columns and then convert them using `type.convert` to logical, integer, numeric, complex or factor as appropriate. The functions allow for manual conversion, too. But be careful with it (see “Casting data types” below). Further functions for file reading/handling are `load()`, `source()`, `scan()`, `attach()`. For more information see the help files of R.

**Hint.** Recently, the developers of RStudio published the R-package `readr` (Wickham and Francois (2015)) for fast data reading. Compared to the equivalent base functions, `readr` functions are around 10 times faster. They’re also easier to use because they’re more consistent, they produce data frames that are easier to use (no more `stringsAsFactors = FALSE`). `readr` makes it easy to read many types of tabular data but works the same way as the standard read functions (see [http://blog.rstudio.org/2015/04/09/readr-0-1-0/](http://blog.rstudio.org/2015/04/09/readr-0-1-0/)).

**Hint.** For Excel files (except for csv-files) there is the package `xlsx` of Dragulescu (2014):

```r
library(xlsx)
xlsx.file <- read.xlsx("test.xls", sheetIndex = 1)
class(xlsx.file)
```

**Hint.** To read input form other statistical software the `foreign` package (R Core Team (2015)) contains functions to read in output of different formats, for example SAS or the R predecessor S-PLUS. There is also the `Hmisc` package for SPSS and SAS of Jr et al. (2015).

### Read data from database

Another way to call (and store) data are databases. R does not have a direct interface to standard databases like ODBC, SQL or Oracle. But several R-packages provide access to databases through suitable interfaces.

- `RODBC` ODBC based databases (including Microsoft Access and SQL Server)
- `RMySQL` interface to MySQL
- `ROracle` interface to Oracle
- `RJDBC` access to databases through a JDBC interface

```r
## RODBC Example
## Import 2 tables (Crime and Punishment) from a DBMS into R
library(RODBC)
myconn <- odbcConnect("mydsn", uid="Rob", pwd="aardvark")
crimedat <- sqlFetch(myconn, Crime)
pundat <- sqlQuery(myconn, "select * from Punishment")
close(myconn)
```

### Write data to a file

There are numerous methods for exporting R objects into other formats. For SPSS, SAS and Stata one needs to load the R-package `foreign`. But the basis write function is `write.table()`, which exports data to a text file with delimiter of your choice. `write.csv()` and `write.csv2()` are sub-functions of `write.table()` for CSV files.

```r
## To write a CSV file for input to Excel one might use
x <- data.frame(a = "bla", b = pi)
write.table(x, file = "foo.csv", sep = ",", col.names = NA, qmethod = "double")
```

```r
## and to read this file back into R one needs
```
read.table("foo.csv", header = TRUE, sep = ",", row.names = 1)
## NB: you do need to specify a separator if qmethod = "double".

## Alternatively
write.csv(x, file = "foo.csv")
read.csv("foo.csv", row.names = 1)

The function `save()` writes an external representation of R objects to the specified file. `save.image()` is just a short-cut for “save my current workspace”, that is, `save(list = ls(all = TRUE), file = ".RData")`.

### 5.5.2 Data types, casting and merging data

Here we want to start with some general information about data types in R, casting data types and merging data. These are potential sources for erratic function calls, function returns or even erroneous results! Take this in mind when performing detection and correction of missing or false data (see Subsection 5.5.3).

**Data types in R** R has a wide variety of data types including scalars, vectors (numerical, character, logical), matrices, data frames, and lists. Although this should be known since you first met R we still see sometimes false usage of data types. Vectors as well as matrices can have numerical, character or logical values.

A data frame is more general than a matrix, in that different columns can have different modes (numeric, character, factor, etc.). Matrices allow only for one mode! Similar to the data frame, lists allow you to gather a variety of (possibly unrelated) objects under one name. In contrast to a data frame the elements are not ordered in columns and thus do not have to have the same length.

**Dates** are not a data type of their own, but R can handle dates. In R like in many other computer languages, dates are represented as days since 1970-01-01, with negative values for earlier dates. Special functions are provided by R to convert characters to dates and vice versa, to read the current date or format a date value for printing, using expressions like `format="\%m \%d, \%Y"` for the display mm dd, yyyy.

**Example:**

```r
today ← Sys.Date()
format(today, format="%B %d %Y")
"September 11 2014"

## Convert date info in format 'mm/dd/yyyy'
strDates ← c("01/05/1965", "08/16/1975")
mydates ← as.Date(strDates, format="\%m/\%d/\%Y")
## Number of days between 01/05/1965 and 08/16/1975
## Convert the date back to character data
strDates ← as.character(mydates)
```

**Casting data types** First rule: Try to avoid!! Second rule: If you can not avoid casting a data vector/matrix/frame to another data format or a numeric/character/logical value to a different data type, be careful and double check the results. For the former R provides functions like `as.vector()`, `as.matrix()` or `as.data.frame()` and many more (see R help). The latter can be more problematic. Generally it should be avoided. Nevertheless, R has some functions for it
too, as we have seen for example for date values. Similarly, `as.numeric()` creates or coerces objects of type numeric or `as.character()` creates objects of type character.

**Merging data** Here we have to differentiate between combining data and merging data. While the former is in general no problem, the latter has to be treated carefully.

Small bundles of data can be combined by standard R-functions to a new joint object. Bigger data frames or matrices should be considered to be edited in a data base software, which is more efficient.

To merge two data frames by common columns or row names, or for other database joining operations one can use the function `merge()`.

### 5.5.3 Detection and correction of missing or false data

In practice, data are often not complete, meaning that for some observation lines not all columns have a value. The reasons for incomplete data are manifold. For examples in clinical trials the patient does not fill in all fields of the form or there is a public holiday and no trading/measuring is performed. If you have such data the very first thing to consider is to be aware of the problem, that is, to notice that there are missing values. For detection we already mentioned the function `is.na(x)`. Some further functions and methods to check your data are given in the next paragraphs.

**Sanity checks** One of the most important – and we think sometimes also very interesting – points in data handling is the first look at the data set. Whenever you obtain a new data get a “feeling”, an impression of the data. Plotting and looking at graphs is of utmost importance here. Learn the dimension of the data, that is, how many lines and and columns do you have. Which types of data do you have? Only numerics or characters, too. Is there a natural ordering of the data as for example in time series?

As we already said at the top of this section: Look at your data carefully! Before you start any modeling check your data for sanity, errors, erroneous data or missing values, etc. The following methods and functions should give you selected tools for such checks.

**head(), tail() and summary()** R comes with several functions to have a first look at your data or an overview of it. To asses the data structure one can use `head()` to list the first 6 lines or `tail()` for the last 6 lines. Also `str()` gives you an overview of the data structure. The function `summary()` gives a compressed overview of an object enumerating the minimum, maximum, mean, median and some quantiles of each column. Check these results for sanity. For example if you have a variable `weight` and `summary(weight)` returns a negative value for its minimum you know that there are one or more false data entries, since `weight` has to be positive. Also large outliers can be detected.

**Plot data** A s mentioned above, a powerful tool to detect outliers are plots. R provides a palette of plotting functions to visualize your data. Make use of it! A plot can tell you much more than a table. The latter allows the user to compare two numbers at a time whereas a plot may reveal the full “story”. You do not need fancy plots here. This is just to get an impression of your data and find important features or erratic data points.

**Time series - date/time** Time series are a special type of data. Additional to their information stored in each entry it contains more hidden information – the time ordering. Usually an extra column with dates or times is provided. This opens the door to further checks:
**Chronological order** Are the dates in chronological order? For time series analysis this is important.

**Equidistance** Are the measurements in equidistant time intervals? Some analysis methods assume equidistant entries. The R-function `diff()` returns suitably lagged and iterated differences.

```r
mydates ← as.Date(c("2014-09-13","2014-09-14","2014-09-16"))
diff(mydates)
Time differences in days
[1] 1 2  # not equidistant
```

**Missing dates** Are there missing dates? For example on public holidays there are no trading data of financial time series. If all your time series have the same missing dates (see also the next point) and no equidistant measurements are needed this is no problem. Otherwise one can use automated gap fill functions (see below).

**Length and starting/end point** Have all your time series the same length and starting/end point? To this end one can use `length()` or `dim()`. `head()` and `tail()` may help here as well.

**Adjust/correct data** Here we want to address some methods how to adjust/correct data. The main question is, do we delete a data entry/row/column or fill missing values? Both have their advantages and disadvantages and depend on the specific data set and problem at hand. We can not – and will not – give any advice for one or the other method. You have to decide which one is suitable to your specific data set.

Nevertheless, we would like to point out possible choices/tools:

**Base R functions** We already introduced the function `is.na()` for NA detection. Similarly, `is.nan()` and `is.finite()` check for impossible numbers (division by zero, for example) represented in R as `NaN` (not a number) and finite numbers.

Some functions can handle missing values, for example,

```r
x ← c(1,2,NA,3)
mean(x)  # returns NA
mean(x, na.rm=TRUE)  # returns 2
```

The function `complete.cases()` returns a logical vector indicating which cases are complete. Similarly, `na.omit()` returns the object with column-wise deletion of missing values.

```r
## List rows of data that have missing values
mydata[!complete.cases(mydata),]
## Create new dataset without missing data
newdata ← na.omit(mydata)
```

**Educated guess** If you have expert knowledge or further information at hand, you may want to fill missing values by good guess. Be aware that you have to justify your guess and mark your changes in your work.

**Automated gap fill** To fill gaps by the last or next available value the package `zoo` of Zeileis and Grothendieck (2005) provides the functions `na.locf()` and `na.fill()`. Here some examples:

```r
require(zoo)
na.locf(c(2,3,NA,1,4,5,2))  # fill gaps by last available value
# [1] 2 3 3 1 4 5 2
```
Version control

When working on the same project, it is necessary to exchange and “merge” individual contributions at certain points in time. One approach would be to exchange the corresponding files by email. This seems fine in projects where only two people are involved, as long as not both work on the same parts of the file simultaneously. To make sure that one does not accidentally take over an outdated part of the file, both participants have to use a “diff tool” and, every time they receive a file, compare it to their “local version” (to find the “difference” so-to-speak) before taking over newly added parts. Possible overlaps have to be fixed by hand. Needless to say, even if only two people are involved and files are not exchanged very often, this is tedious and prone to errors; especially if the project involves several files. Furthermore, it would be advisable to keep older versions of the files in case a result has been erroneously deleted previously, for example. But how do you know in which of the backup files the latest version of the result resides? There are endless of such problems and thus people have automatized these procedures. Below we briefly mention some approaches of version control systems, in increasing sophistication.

6.1 Dropbox

Dropbox is not a version control system as you may still face the problem of tripping over changes of other authors. We only mention it here since it is more sophisticated than “sending around files”, simplifies the above process (at least provides versions of files for up to 30 days) and does not have a learning curve as steep as for the tools below. Note that one can also combine Dropbox with the tools described below, but we omit further details in this introduction here.

6.2 SVN

Apache Subversion (SVN) is a widely used version control system. It is a server-based version control system meaning that there is a copy of your collection of files, typically a folder, the so-called repository, stored on a remote server (the SVN server). There are free SVN servers available, they often come at the price of projects being either public (ly available) or private which is fine in most cases; an example of a free private hosting service (also allowing Git access, see Section 6.3 below) is https://cloudforge.com/, for example.

Once the server has been set up, one can checkout the repository, which creates a local working copy. One can then add files to the working copy, commit changes to the (remote) repository, update ones working copy in case a project member has committed changes to the repository, display a log of changes to the repository (including commit messages of the various file versions), etc.
display the status of a file (is the file under version control or not) and differences between the working copy and the version in the repository on the server. In what follows, we describe these processes with some basic example commands often used (GUIs are widely found on the internet, the workflow remains the same). For more information use `svn help ...`; for example, `svn help commit` for the commit command.

How to set up an SVN server or how to start an SVN project on hosting services like cloudForge are explained in several places on the internet. Hence, we do not explain this step in more details below (also not for Git, see Section 6.3). In the following we suggest that there is already an SVN server running or an SVN project set up. Besides online resources, local IT services can typically also be contacted for assistance.

### 6.2.1 Checkout

In the beginning of the project, you want to checkout the (remote) repository, so that your local working copy is created. The working copy is stored in a subfolder `.svn` of your current working directory. Note that unless files were manually uploaded to the server or if the repository has already been set up by a colleague, the repository (and thus the working copy) is empty. All we need for the checkout is a URL to the SVN server or project (you will be asked for your user credentials as have been provided previously for setting up the repository on the server):

```bash
1   svn co <url> # check out project (create working copy from the repository)
```

As example URLs, see

1. `svn://svn.r-forge.r-project.org/svnroot/nacopula/`
2. `svn://r-forge.r-project.org/svnroot/vinecopula/`

which are the SVN URLs for the R packages `copula` and `VineCopula`, respectively, hosted by R-Forge.

After we have checked out the repository (created the working copy) we can add files to the working copy (see below) or modify existing files which are already under version control (for example, if they have been downloaded during the checkout process in case the repository was non-empty).

Clearly, the same repository can be checked out by many people (including one person on several devices, for example), which makes this tool especially interesting when a larger number of people collaborate. Furthermore, the following variants can also be useful:

```bash
1   svn co <url> output_Dir_name # check out the project into a certain directory
2   svn co -r110 <url> # check out version 110 of the repository
```

If you use a GUI (like SVN Torquise on Windows) these steps, as well as all the following functions, can be done via easy-to-handle menus. As mentioned before, the workflow remains the same.

### 6.2.2 Add and (re)move

Adding a file (or folder) “foo” to the working copy can be done as follows. One first copies or moves the file to the local directory (the directory containing the working copy `.svn`) and then adds it to the working copy (thus putting it under version control) via:

```bash
1   svn add foo # add directory or a file 'foo' to working copy
```

It is important to note that although “foo” is now under version control, the remote repository (on the server) does not see “foo” yet and so your collaborators do not see “foo” yet either. For
6.2.3 Update, commit

Once you have made changes to your local files, you want to commit these changes to the repository on the server so that your collaborators can see these changes after they update their working copies with the server’s version. Before committing your changes, you should conduct an update of your working copy yourself so that you see changes your collaborators have committed in the meanwhile (and so that you can solve possible conflicts, see Section 6.2.5); note that this update only updates your working copy but does not overwrite your local files (or file changes). You can (and should) thus safely do an update before committing, even if you have already changed files. An update and commit can be done as follows:

1. `svn up` # update working copy (does not overwrite local changes)
2. `svn ci -m 'added file ‘foo’ and fixed an error'` # use -m for commit messages!

Use short commit messages so that your collaborators can get an idea about what you changed when looking at the log file, see Section 6.2.4.

Note that when removing a file not using `svn rm`, `svn up` will draw the repository’s version of the file, so it will appear again. To remove it and propagate the changes, you should do `svn rm foo` followed by `svn ci -m ‘removed ‘foo’’`.

6.2.4 Log, status, list, diff

After an update of your working copy, you can display the log message to get a feeling for changes submitted by your collaborators (if they provided commit messages, which they should). This can be done, for example, as follows:

1. `svn log | head -20` # displays the first 20 lines of the log message

   If one would like to get an overview about the status of certain local files, one can use the following command:

   `svn st` # display the status of files

   The first columns of the output contain one character abbreviations like “A” (for “added”), “C” (for “conflicted”), “D” (for “deleted”), “M” (for “modified”), “?” (for items not under version control) and “!” (for missing items, for example, if a file was removed by a non-svn command); see `svn help status` for more details. `svn st` can also be helpful to detect which files have recently been modified in case one would like to commit only some of them, for example.

   To see which files are under version control at all, use:

   `svn list`

   The command `svn diff` can be used to display changes in (differences between) the local files and the working copy. Some possible usages are:
6 Version control

### 6.2.5 Conflicts

It may happen that you commit a change to the file “foo”, but one of your collaborators has already committed changes of “foo” to the repository. This is in general no problem if the two sets of changes do not overlap. SVN will then merge the file changes automatically. However, if the changes overlap, your commit will fail. You should then use `svn up` (which you should have used before committing anyways). SVN will then display that a conflict has been discovered and offers the possibilities to postpone the conflict to be resolved later (“p”), make a full diff, that is, to show all changes made to the merged file (“df”), edit the merged file (“e”) and to show further options (“s”). “df” reveals how much work it is to solve the conflict, if it is only a minor one, one can use “e”, for larger ones one would typically use “p” and fix the conflict by hand. Using “p” creates the additional files “foo.mine” (with your version), “foo.<last>” (the original file you worked with; “<last>” denotes the version number), “foo.<current>” (current version of your colleague), whereas the original file “foo” contains modifications of the following type:

```
<<<<<<< .mine
  here is what you wrote
=======
  here is what your colleague wrote
>>>>>>> .<current>
```

For each of these chunks, just decide which of the versions you would like to keep and delete the other (including “«««” and “=======” etc.). After that, tell SVN that the conflict has been resolved via `svn resolve foo`; this will automatically delete the additional files and you can then update and commit again. There are other possibilities as well, for example, `svn resolve --accept=theirs-full foo` would directly solve the conflict by always accepting your collaborator’s version.

### 6.3 Git

Git has recently become a popular version control system. In contrast to SVN, Git is a distributed control system meaning that your local working copy is also a full repository (with its own local history and branch structure) and you can commit to it. This implies that there are possibly two repositories involved, your local one and, possibly (but not necessarily), one on the server. This decentralized version control system has the advantage that if you are off-line for some hours, say, but still want to revert back to an older version you can do this (via your local repository). However, having two repositories involved adds another level of complexity, for example, there is not only one but two update commands, one that updates your local repository from the remote repository (`git fetch`) and one that updates the local repository from the server and merges the changes into your actual files (`git pull`, essentially a `git fetch` followed by `git merge`). Git is faster than SVN and more sophisticated when it comes to branching, merging and solving conflicts. An interesting discussion on strengths and weaknesses of Git and SVN can be found under [http://stackoverflow.com/questions/871/why-is-git-better-than-subversion](http://stackoverflow.com/questions/871/why-is-git-better-than-subversion). We omit further details here. The usage of git and gitHub for the development of R-packages and R-projects is well described in the book of Wickham (2015).
6.3.1 Branches

But one interesting point we want to mention here about GIT: Branches. What we have seen so far was that all users work in one shared directory/repository, add and update their files to it. Branches now allow to fork the repository into different working lines or different versions. It can be a part of your everyday development process. When you want to add a new feature or fix a bug - no matter how big or how small - you spawn a new branch to encapsulate your changes. This makes sure that unstable code is never committed to the main code base, and it gives you the chance to clean up your feature’s history before merging it into the main branch. How to create a branch, submitting to it and finally merge the changes to the master branch, that is, the main branch, we will explain in the following.

But before we come to that some more explanation about branches to get an idea of it: A branch represents an independent line of development. But Git does not copy all files to the new branch. Instead a branch is a reference to a commit. One can understand it as a pointer to commits. When you create a branch, all Git needs to do is create a new pointer – it does not change the repository in any other way.

Step-by-step workflow:

**git branch** lets you create, list, rename, and delete branches.

To list all branches of your repository type

```
1 git branch
```

To create or delete use

```
1 git branch <branch>
2 git branch -d <branch>
```

respectively. The branch command is closely connected to the checkout and merge command.

**git checkout** lets you navigate between the branches. Checking out a branch updates the files in the working directory to match the version stored in that branch, and it tells Git to record all new commits on that branch. It also makes the called branch the current branch.

```
1 git checkout <branch>
```

One can also combine a checkout command with the branch command to create a new branch before check out.

```
1 git checkout -b <branch>
```

**git merge** lets you take the independent lines of development and integrate them into a single branch. Note that all of the commands presented below merge into the current branch set with the checkout command. Thus only the current branch will be updated to reflect the merge.

```
1 git merge <branch>
```

The complete workflow is illustrated in the following example.

```
1 ## Start a new feature
2 git checkout -b new-feature master
3
4 ## Edit some files
5 git add <file>
6 git commit -m "Start a feature"
```
7 Submitting a paper

Publishing your research should be the goal and final step of your scientific work. In this chapter we list some points we think are useful to know when submitting a paper; they may also serve as a checklist before the submission. Here we follow Davidian (2005) who provides a good outline in her set of publicly available slides. We also updated some suggestions and added personal practical experience.

In the following we assume that the manuscript has already been (mostly) written and is ready for being handed in to a journal or conference proceedings. Furthermore, we assume that you followed the journal’s guidelines concerning style and structure of a paper. The 3-step procedure we describe in the next subsections is the same for all paper submissions independent of the journal or the conference.

7.1 Purpose of journals: How to find the best fitting journal for my research

The very first and important step to publish one’s research is to find the best fitting journal. There are hundreds of different (statistical) journals available having different objectives, focus and scope. Major journals such as the *Journal of American Statistical Association – Theory and Methods* or *Annals of Statistics* cover several objects and scopes. Other journals specialize in a major topic, for example, *Statistics in Medicine* or *Journal of Agricultural, Biological, and Environmental Statistics*, or in one or two scopes like *Computational Statistics and Data Analysis* or *Journal of Computational and Graphical Statistics*.

Classify and rank your own work according to the points below. This will narrow down the amount of adequate journals and may assist you in finding the best fitting journal(s).

**Objective and focus** Journals can be distinguished by their major objective or focus of statistics via the area of application (for example, medicine), the method (for example, time series analysis) or the goal (for example, dependence modeling).

**Scope** The objective or focus of a journal is obvious in most cases whereas the scope is often not. Each journal propagates its own goals and scopes they want to cover. According to Davidian (2005), there are six different scopes in statistics; below we added two more.

**New theory** Situation in which suitable methods are not available – propose such a method.

**New theory based on existing work** Methods are available, but have limitations – extend, improve, relax assumptions.

**New theory based on existing work II** Methods are available – propose a competing one and compare and illustrate.
Application or problem driven An important subject-matter application has specific issues – show how to handle these with existing or modified methods.

Extensions Properties of existing or new procedures are unknown – work out formal theory.

Simulation studies Properties of existing or new procedures are unknown – carry out extensive simulations.

Surveys Overview of existing methods and comparison. Usually a relative new method is added, too.

Manuals and Vignettes Software descriptions, manuals, vignettes and code snippets for public available statistical functions, packages and software. They can reach from presenting theoretical methods and algorithms to details about implementation or numerical challenges.

Usually a journal’s scope is a mixture of several points. Classify your own work and check if it fits in the journal’s scope. In the cover letter, one should also mention how the paper fits in the journal’s scope.

Audience The objective and scope of a journal naturally implies the corresponding audience. The audience can be academics, graduate students, practicing statisticians, researchers of other disciplines etc. Identify your audience. Who would be most interested in your work? What journals tend to be read by researchers in the corresponding area?

Journal rankings intend (but often also fail) to reflect the journal’s impact within its field and beyond. Rankings are also an indicator for the quality of a journal, but do not have to be (if you have found a journal which seems “right” for your research, go for it, do not listen to rankings – only use them to compare adequate journals which you are unsure about otherwise). Rankings are facilitated by analyzing citations of scientific (and non-scientific) publications. The most common measure is the impact factor which reflects the average number of citations to articles published in science and social science journals; see http://en.wikipedia.org/w/index.php?title=Journal_ranking&oldid=608376667 (accessed June 18, 2014). But there are several other measures, too.

Practical hints

1) Have a look at the journals in your reference list. Usually they are of the same field of statistics using similar methods or have similar areas of application.

2) Examine papers published in a recent issue of the journal for style, level of technical detail or discussed topics.

3) Have a look at the list of Editors and Associate Editors of the journal. One of them will handle your paper. You may get an idea who it could be by looking at their research interests. Maybe you know her/him and, as an exception in case you are absolutely unsure, can address her/him personally.

7.2 Preparations before submission

Before you can submit the manuscript to a journal or conference proceedings you have to accurately prepare your submission. False styles or missing material such as figures or separately listed tables may be a reason for a rejection. Thus invest a good quantity of time to prepare your submission. Also watch out for typographical errors, they certainly do not contribute to leave a good impression about the quality of your work. Ideally, have a colleague (native speaker) read the paper and give you feedback.
First of all, *read the journal’s guidelines for authors* and carefully follow these instructions! The journal typically lists all necessary properties and style requirements, conventions on math, figures, tables, font size, font style, spacing, etc. For (statistical) journals, the most important points are:

**Scope** As mentioned above, check whether your manuscript fits into the scope of the journal.

**Length** Some journals limit the number of pages. Important is here the number of pages in the journal’s style format with correct page boundaries, spacing, font size, etc. Do not go beyond the limits, rather stay well below; also, a shorter paper typically takes less time to review.

**Style of article (including references)** Most journals already demand a paper submission in the journal’s publishing style. Therefore, they offer *L*\textsc{atex} style files defining page boundaries, spacing, font size, etc. or Microsoft Word samples and guidelines.

**Authors and authors affiliation** Watch out for double-blinded submission requirements (neither the author knows the name of the reviewer (as usual) nor the reviewer knows the names of the authors).

**Figures and tables** Usually not all figure formats are accepted. Check which ones are preferred by the journal. There are several open source software tools to convert one format into another (for example, \texttt{ps2pdf} or \texttt{epstopdf}). Make sure that all graphics have a high enough resolution. Sometimes it is requested to put all figures and tables separately at the end of the manuscript. Furthermore, each figure or table should have a reasonable caption. A graph or table including the caption should be self-explaining without requiring to read the article’s text. Therefore captions and legends have to be accurate.

**The abstract** is a concise summary of what you will present in the paper and provides the key findings (not just the conclusions). Formulas and citations to other works should be avoided in an abstract. Some journals even prohibit formulas and citations in the abstract. The golden thread should become clear without reference to the rest of the manuscript. The abstract should be no more than 200–250 words (depending on the journal) and is typically written in passive.

**Keywords** Immediately after the abstract, provide a maximum of 6 keywords. Be sparing with abbreviations: only abbreviations firmly established in the field may be eligible. These keywords will be used for indexing purposes.

**Highlights** are a short collection of bullet points that convey the core findings of the article. Usually highlights are used for online publications on the journal’s web page to highlight the article’s main findings.

**File naming** Some journals require to follow special file naming conventions for additional material such as figures (to work in an automated compilation process).

### 7.3 Submitting a paper

Nowadays most journals offer (and prefer to use) a submission management system. On submission of a paper, one has to provide several details such as the work address, title, abstract, keywords, and finally one has to upload the manuscript (figures, etc.) and a cover letter. Follow it step by step.

**Check for completeness** Check all your answers in the forms, check for completeness, check if you followed the journal instructions for authors, check the cover letter (see below). Modern submission management systems create a final .\texttt{pdf} containing all the documents handed in.
Have a last look at the file before the final submission. If it is required to submit a .tex file, it will be converted to .pdf. Check it carefully! Are all figures included? All mathematical symbols displayed correctly? Are references shown properly? Check on the formatting, page style and spacing.

Cover letter Enclose a short cover letter making clear your intention (submission of a paper for publication) and note any conflict of interest (such as sponsored work). The cover letter is either uploaded to the manuscript management system or sent in an accompanying email to the editor, most often the former.

Congratulation! You submitted your hard work offering your research to the community and the public audience. Now it is the time to wait for feedback from the journal (typically, reports are sent by one to three reviewers and sometimes also the Associate Editor). This can take a frustrating amount of time, though (up to several years). In the meanwhile, most journals offer the possibility (check the journal’s policy) to put the paper either on the authors’ websites or on special publication servers such as http://arxiv.org. This has the advantage of making the results available right away and also avoids the paper to be rejected by a reviewer who then publishes the results on his own (before your paper is published); normally, the reviewer is an expert in the same area and thus there is the potential of this situation to happen. Unfortunately, the review process in its current form bears risks of this (and other) type.

Acknowledgments
We would like to thank our students and the many researchers and practitioners who sent us their source code in the past for inspiring this guide. The first author would also like to thank NSERC for financial support for this work through Discovery Grant RGPIN-5010-2015.

References

Higham, N. (1993), Handbook of Writing for the Mathematical Sciences, SIAM.
References


R Core Team (2015), foreign: Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, Weka, dBase, ... R package version 0.8-63, http://CRAN.R-project.org/package=foreign.


Trzeciak, J. (1995), Writing Mathematical Papers in English, EMS.


