Guidelines for Statistical Projects: Coding and Typography

Marius Hofert[1], Ulf Schepsmeier[2]

2014-10-01

Abstract

Guidelines for conducting, implementing (in LaTeX and R) and documenting statistical (research) projects are provided in order to improve readability and reduce the error rates of theses, scientific papers, reports and especially code. This is meant to save supervisors, package maintainers, students 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
Coding, typography, LaTeX, R.

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

Contents

[1]Department of Statistics and Actuarial Science, University of Waterloo, 200 University Avenue West, Waterloo, ON, Canada N2L 3G1, `marius.hofert@uwaterloo.ca`
[2]Department of Mathematics, Technische Universität München, 85748 Garching, Germany, `ulf.schepsmeier@tum.de`

1 Introduction

These guidelines are meant for students, professors and practitioners who would like to write, participate in, or supervise a project such as a bachelor, master, Ph.D. thesis, or scientific paper in the intersection of mathematics, statistics and computer science. Besides some general recommendations, we focus on the software tools LaTeX and R for conducting, implementing and documenting the project.

Before going into detail, some remarks are in order:

The science of coding Coding (in LaTeX, R etc.) is like a handwriting. From the corresponding files and style of coding one can read a lot. Writing correct, 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 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 "say" or do with your code. Besides being correct, your code should be easily readable and extendable by others. Larger projects involve more and more contributors, each of which should be able to *easily* follow your code and adjust it if required, hence some guidelines are in order.

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

Focus The guidelines reflect our *personal recommendations* and *experience* using LaTeX and R (and related tools mentioned below) in our *personal areas of research* (which lies in the intersection of mathematics, statistics and computer sciences). It is clear that such a guide cannot be exhaustive. In particular, this is *not an introduction* to the topics presented! If you feel that we missed an important aspect not easily found or addressed in other guidelines or tutorials, or if you can improve this document, please let us know.

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; the left-out exceptions are the 10–20% not discussed here. Furthermore, some aspects are discussed in more detail than others (which is motivated by our work/judgement). Aspects addressing more advanced users are marked with an A.

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 like this). You may or may not find this helpful, the principle *do not like it? do not use it!* applies.

We will constantly update this document. Therefore, there will never be a version which can be considered final.

The guidelines are organized as follows. In Section 2, we give general suggestions for written or coding intensive projects. Section 3 briefly addresses the importance and choice of text editors. Section 4 and Section 5 point out recommendations when working with LaTeX and R, respectively.

2 General suggestions

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

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

Meaning Essentially, this means that one should stop after having spent 20% of the time/effort one could spend on the project, since all additional effort would just improve the outcome by the remaining 20%.

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, it will come back for revision at some time. You certainly do not want to realize a year later, that you now actually have to start over with the whole work (instead of doing just a revision). Furthermore, if a referee feels that you only spent 20% of the effort on the submitted work, this most likely results in a rejection. Do your homework, work hard and exclusively on the topic and you will get a result you can be happy with. Also, your supervisor is happy to learn about what comes (far) after the 80% (in contrast to hearing about well-known results). 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 (maybe applying as well to research in general):

1) If you have a problem, search for it.
   The chance that you are the first one working on this problem is small. Others may have already solved the problem (in an elegant, optimal, fast 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 (publicly) available.

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

In short, if there is a good solution available, learn from it. If there is not, write your own, but make sure it is of good quality so that others can benefit from it when they find themselves in the same position.

2.3 English in mathematics

The language science speaks is (American) English. Even if it is only a comment in a script you write, a file name, variable, or function etc., use (American) English. It will be easier for others to find and understand your work (besides various other advantages).

Additionally, we want to mention some basic rules for mathematical typography in English; here we follow Halmos (1970) and Higham (1993).

Short(er) sentences Use short sentences in your theses or project document. Long sentences are not as conventional in English as they are in German, for example. So German students, at least, are advised to follow this rule. Formulate (sufficiently) simple and simple to understand sentences. This improves the readability of your text.

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

Passive mode In English it is often easier and more elegant to formulate a statement in passive mode. But do not use it too often, especially in American journals the active mode is often preferred.

Readable text instead of operators In the English mathematical literature, words such as “there exists” or “for all” are to be preferred over their operator equivalents “$\exists$” and “$\forall$”; the former make the text more readable. This contrasts, for example, German mathematical typography.

Another symbol frequently used in German but not English mathematical typography is ":=" ("=:") for defining the quantity on the left-hand (right-hand) side by the one on the right-hand (left-hand) side.

Comma rules in English As non-native English speaker one is often unsure if and when a comma has to be set in a sentence. In general the regulation regarding commas in English is less restrictive as in German, for example, but there are some rules:

- A nonrestrictive element, which does not limit scope but merely provides additional information, is indicated by being set off by commas.
- A restrictive relative clause is introduced with "that" and is not set off by commas.
- A nonrestrictive relative is introduced with "which" and is always set off by commas.
- 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".

2.4 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 or `.R` script), from the very first to the very last character in the file (even when using spaces). This will significantly help you when search-and-replace is in order (after a supervisor's or referee's feedback, or if you would like to make changes).

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

$$\Big(\Big(((((a_7x+a_6)x+a_5)x+a_4)x+a_3)x+a_2\Big)x+a_1\Big)x+a_0 \tag{1}$$

or

$$[\{([\{(a_7x+a_6)x+a_5\}x+a_4]x+a_3)x+a_2\}x+a_1]x+a_0.$$

It does not matter (much) which is to be preferred on first writing (and for publications in scientific journals this is often determined by the journal's style guide), as long as you stick to the very same rule throughout the whole document.

2.5 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, proofs, etc. In mathematics, this especially applies to *assumptions* made for certain statements to hold.

Be short Besides being precise, be short. Twenty well-written pages are much more interesting to read (besides being less to type and less to correct or grade) than one hundred sloppy and boring pages.

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 explanation.

Formulas Put important formulas in a displayed equation and check that the main ideas of your work can be followed by just looking at the displayed equations. In the same spirit, a displayed equation/formula etc. should make sense as much as possible without looking at the text[3]. Conversely, more complicated formulas should always be explained in verbal form in the text as well. This, together with the displayed equation/formula (do not use text here), gives the reader the chance to understand the topic on two different levels, one language-based and one formula-based. Ideally, there should also be third, graphical-based level by illustrating the (complicated) formula with a graphic.

File, variable and function names Naming files, variables and functions (both from a mathematical and a programming point of view) in a meaningful way is important. 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.

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 `tau` for a certain value of Kendall's tau or `n` for a sample size (similar to the standard notation n in statistics). Choose variable names in scripts as close as possible to their mathematical equivalents. In the same spirit, do not call a function `fun`; note that R, for example, would not even allow the (reserved) name **`function`**.

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”.

The following basic rule typically provides compact and readable code: The more often you need a variable (this partly also applies to functions), the shorter its name should be. Often, short names can be generated by leaving out vocals, the human eye typically “interpolates” correctly and directly recognizes the corresponding word (and thus the meaning of the name). In general, omit superfluous parts in function names; for this and other naming conventions more specifically in R, see Section 5.3.2.

2.6 Be structured

Introduction, abstract and summary come last Do not start to write your paper or project document by thinking about the introduction. The introduction, abstract and summary are the last parts you should write in your project. First concentrate on the content. At the very last, think about the introduction and the end. Write down headwords, for example for the motivation of the topic and finally write out the introduction in full. Additionally, you can

[3] Also, when introducing a function f for the first time, do not just write

$$f(x) = \log x.$$

Instead, make it more precise by providing its domain, so

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

also note the most important three or so words on every page of your document. This can help in creating a golden thread.

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 even two levels are typically fine. Only use subsections if you have more than one meaningful subsection. Otherwise work with paragraphs.

Two possible ways During our scientific career we learned two ways of starting a document and structuring it.

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

Top-down Think about a logical way of reading/following your paper. Write down the chapters and sections you have in mind and order them. Then write down your ideas and text in the corresponding sections.

Listings Sometimes, list of bullet points are very helpful to write down several connected statements in a compact way. If they have an order you may use an ordered list, otherwise an unordered list. You can use different numbering styles, e.g., arabic or roman numbers, or alphabetical items. In unordered lists different bullet point styles are also available (we mainly use filled squares in this document). Even minor headings are possible, see for example this guide.

2.7 Be self-contained

Outsource Instead of reproducing known results, properly refer to papers, books, or code/packages your work is based on; when referring to books, always provide a page number (and mention the edition of the book in the references).

How to cite Typically, author-year citation style (such as "paperAuthorLastName (YYYY)" or "bookAuthorLastName (YYYY, pp. 17)") provides the most readable and memorable citations. Also, instead of just "It follows from A (2000) and B (2010) that z holds", write "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 therefore conclude that z holds". In this way, the main idea can be followed without having to read "A (2000)" and "B (2010)", 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.

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. It makes the text unnecessarily long and difficult to read (and by Section 2.5, we wanted to be concise!).

2.8 Be reproducible

Meaning Make sure your results are reproducible, that is, one can repeat the experiment or simulation (or even a proof (!)) and obtains the exact same result.

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 or Knitr. These two R packages allow to combine R and LaTeX code in one file (`.Rnw` files). 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 file created by LaTeX (there are many more options).

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 LaTeX. Furthermore, these tools have a good documentation and an intuitive handling.

Tools like Sweave and Knitr also have their drawbacks. Mixing LaTeX and R 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 LaTeX code. Finally, run time is longer (although intermediate results can be caught and stored, but this again makes the code longer).

Overall, we thus do not recommend to use Sweave or Knitr for large projects, unless 1) there is a significant amount of code to be displayed in the written companion of a project (which is rarely the case); 2) the code runs sufficiently fast; and 3) unless the user's knowledge about LaTeX and R is sufficiently advanced.

2.9 Optimize communication, meetings and preparation

Getting in contact If you contact a researcher/instructor etc. for the first time, start by (briefly!) saying/writing who you are (what is your status? master/Ph.D. student? practitioner?), what the goal of your project is (thesis? software development?) and who you work with on this project (supervisor, colleagues, etc.).

Email communication Communication between you, your supervisor and a potential third party such as a tutor will often be mainly by email. We advise you to consider the following points:

- Choose a short but meaningful subject line. Subject matters such as "Hi" or "Dear Professor" are not meaningful. A concise subject would be "Problem master thesis: for-loop too slow", for example.
- Check your email before you send it. Is the announced attachment attached? Is the question clearly formulated? Do I have answered all the questions the supervisor asked me in her/his last email?
- An unwritten law (at least applying to students) 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. Send them to your supervisor (in the same email in which you ask for an appointment). Make a suggestion for two possible dates for the meeting. Bring a paper and pencil to the meeting Also, be able to briefly summarize your work/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, the more likely you will directly get the answer you were looking for.

During meetings Take notes of the answers, comments, suggestions, etc. your supervisor mentions during the meeting.

Wrap-up Complete and structure your write-up right after the meeting. Put the points you have 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 in the document to do). Finally, go through all files again, work on the `TODO`s, and take notes of the questions that arise (to have them ready for the next meeting).

Feedback 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 above!).

Matter of course During meetings, be awake (!) and polite. Do not answer emails or phone calls during a meeting (yes, it happened to us!)

3 Editors and integrated development environments

Why to think about it It seems difficult 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 command lines (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 his/her own favorite editor. We will not really recommend one. But we will give some suggestions what a good editor should have and how the editor can support and improve our coding style. Most of the more advanced editors, which we introduce below, support automatically many of the style guides we will give in Section 4 and 5. Especially the ones in Section 5.2 and 5.3.

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 task 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 as Getting Things Done ("GTD") software, partly even 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 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 used in several files simultaneously.

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 (a more general text editor) or TeXShop (for Mac), or TeXnicCenter (for Windows).

For R, we can recommend RStudio which is available on Linux, Mac and Windows. It combines R with an editor, file directory, 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 the easy use of Sweave and Knitr; see Section 2.8.

4 LaTeX

4.1 Getting started

Introduction 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/`.

Help Typically very good help on more advanced topics is provided by `http://tex.stackexchange.com/`.

4.2 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.

Common careless errors Beware of mistakes (supervisor names, dates, spelling of affiliation etc.) on title pages, covers, etc., one typically does not check such pages again after they have been created.

Lazy eye principle To access whether a document looks good, apply the *lazy eye principle*: hold the page a meter away from your eyes and try to "view 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, to see which are the main results or which are only auxiliary results etc.

Character protrusion Use the LaTeX package `microtype` for character protrusion and font expansion (only with PDFLaTeX). By stretching lines ending with certain characters further out in the margin than others, this, for example, provides a visually more appealing justification than by forcing each line to have precisely the same length.

New paragraphs Use paragraph indentation (`\parindent`) instead of paragraph skip (`\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 `.tex` file, not by using `\par`.

Furthermore, before each new ((sub)sub)section, use an empty line (except when a new (sub)subsection directly follows a new (sub)section).

Title case If at all, only use title case in the title of (larger) projects, not in section headings, table headings etc.

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 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 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 used right after a "(" of course) and, in American English, also followed by one.

Footnotes Do not use footnotes. They distract from the reading flow, are rarely accepted by scientific journals and can almost always be omitted anyways.

Introducing new quantities If you introduce/define a new term or notion, make it visible via `\emph{...}` 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 as 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 `\,` to visually separate numbers larger than or equal to 1 000, so write `1\,000`, `1\,000\,000`, etc.

Page ranges For page ranges (such as "1–10"), compound names, or dashes, use `--` (and not just `-`, which is reserved for hyphens!).

Sets The positive integers, the real numbers, the complex numbers etc. can be nicely formatted via `\mathbbm{N}`, `\mathbbm{R}`, `\mathbbm{C}` etc. from the LaTeX package `bbm`.

For indices, note that $i \in \{1, \dots, n\}$ is a more precise statement than $i = 1, \dots, n$.

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

Size of 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 (1).

The space after a parenthesis In displayed equations, large (typically opening) parentheses may reach into the actual formula. With `\,` one can create some additional space; see the difference between `\biggl(\sum_{i=1}^n` and `\biggl(\,\sum_{i=1}^n`:

$$\biggl(\sum_{i=1}^n \quad \text{versus} \quad \biggl(\,\sum_{i=1}^n$$

Labeling Only label those displayed equations etc. that you actually refer to from somewhere in your document (hence a label should indicate a more important or not so easy to remember equation). Do not label every displayed equation, theorem etc. by default. If you do not want to label a certain line in a multi-line equation, use `\notag` (before the line breaking `\\`). If you want to change the label, use `\tag{$*$}`, for example (right before `\label{...}`).

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

Vectors Vectors are column vectors, but written as a tuple $\boldsymbol{X} = (X_1, \dots, X_d)$. Furthermore, use the command `\bm{}` from the LaTeX package `bm` to create bold symbols such as vectors; this also works for greek letters. Note that a transpose sign is only used if required, for example, as in $\boldsymbol{a}^\top \boldsymbol{X}$; use `^{\top}` to generate a transpose sign.

Ruler Use the package `vruler` with the setting `\setvruler[10pt][1][1][4][1][0pt][0pt ][-30pt][\textheight]` (or similarly; see the documentation) to display line numbers in your document. This greatly simplifies discussing certain parts of the document (by email).

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 on), not `"`.

4.3 Technical tricks to improve typography

4.3.1 Citations

How-to Use BibTeX, or – even better – 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 (typically) put references References can often be nicely added at the end of a sentence via a semicolon without disturbing the reading flow; see ....

4.3.2 Spaces and alignment

Escaping spaces after dots and to avoid 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 Ph.D.\ student, I have`...

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~1 and Figure~3`...

Breaking terms over lines If you want to break, for example, a vector $\boldsymbol{X} = (X_1, \dots, X_d)$ over a line, use `$ $` to allow LaTeX to break the line. For example, write `$\bm{X}=(X_1,$ $\dots,X _d)$` or `$\bm{X}$ $=(X_1,\dots,X_d)$`. In the same spirit, write `$\bm{X}_i$, $i\in\{1,\ dots,d\}$` instead of `$\bm{X}_i, i\in\{1,\dots,d\}$`. First, this the former gives LaTeX more freedom in nicely breaking the line and, second, it creates a more readable space between `$\bm{X}_i$` and `$i\in\{1,\dots,d\}$`:

$$\boldsymbol{X}_i, i \in \{1, \dots, d\} \quad \text{versus} \quad \boldsymbol{X}_i,\ i \in \{1, \dots, d\}$$

Watch out for bold indices Watch out for the difference between `\bm{X_i}` and `\bm{X}_i`; the former creates a bold index while the latter does not. Bold indices are typically only used for vectors of indices.

Horizontal spaces Use `\quad` in displayed equations to separate formulas from text or domains from the actual equations etc. A greater separator is `\qquad`.

Use align and alignat For one-column displayed equations, one can use `amsmath`'s `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 text before the environment independently of how much this line is filled (furthermore, `\qedhere` is not correctly put when a proof ends with an `align` environment). One can use `amsmath`'s `equation` environment instead, however, only if the displayed equation only has one line. 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, e.g., `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 displayed equations over different pages. But this is only recommended on the very last iteration of your document preparation process. Ideally, it should not be necessary as it is often more natural 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(\bigl(\bigl(((a_nx+a_{n-1})x+a_{n-2})x+a_{n-3}
        \bigr)x+a_{n-4}\bigr)x+a_{n-5}\Bigr)\\
  &\phantom{{}={}}\biggl(\Bigl(}\cdot x+a_{n-6}\Bigr)x+a_{n-7}\biggr)x+\dots.
\end{align*}
```

shows a properly vertically aligned second line:

$$\begin{aligned} f(x)&=\biggl(\Bigl(\Bigl(\bigl(\bigl(((a_nx+a_{n-1})x+a_{n-2})x+a_{n-3}\bigr)x+a_{n-4}\bigr)x+a_{n-5}\Bigr)\\ &\phantom{{}={}\biggl(\Bigl(}\cdot x+a_{n-6}\Bigr)x+a_{n-7}\biggr)x+\dots. \end{aligned}$$

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 in math mode; in some situations, only `\phantom{={}}` might be required.

4.3.3 Figures

Template for including (side-by-side) figures For including two figures side-by-side, one can use a construction of the following form (for including just one figure, omit `\hfill` and the obvious second `\includegraphics` command).

```
\begin{figure}[htbp]
  \centering
  \includegraphics[width=0.48\textwidth]{my_figure_1_without_ending}%
  \hfill
  \includegraphics[width=0.48\textwidth]{my_figure_1_without_ending}%
```

```
  \caption{Plot of \dots\ (left) and \dots\ (right).}
  \label{fig:label}
\end{figure}
```

4.3.4 Miscellaneous

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

Easier to read letter l Use `\ell` (ℓ) instead of `l` (l) for the log-likelihood.

Emphasize Emphasize text using the LaTeX command `\emph`, not `\textit`. Do not use `\underline`.

5 R

R, see `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.

5.1 Getting started

Introduction 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/`.

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** `install.packages("myPkg")` installs the package `myPkg` from the Comprehensive R Archive Network (CRAN); see `http://cran.r-project.org/`. This is the most typical way to install R packages. Note that `"myPkg"` can also be a vector of packages, so `c("myPkg1", "myPkg1")`.

- **from R-Forge** `install.packages("myPkg", repos="http://R-Forge.R-project.org")` installs the package `myPkg` 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.
- **from .tar.gz** `install.packages("~/my/folder/myPkg.tar.gz", repos=NULL)` installs a package available as a `.tar.gz` file. 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/`.
- **from GitHub** The command `install_github("myPkg")` from the package `devtools` installs packages from GitHub; see `https://github.com/`.

Installed packages can be updated with `update.packages(ask=FALSE, checkBuilt=TRUE)` and removed with `remove.packages("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

```r
require(VineCopula)
citation("VineCopula")
```

gives

```
Ulf Schepsmeier, Jakob Stoeber, Eike Christian Brechmann and Benedikt
Graeler (2013). VineCopula: Statistical inference of vine copulas. R
package version 1.2-1.

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:

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

gives

```
To cite the R package copula in publications use:

Marius Hofert, Ivan Kojadinovic, Martin Maechler and Jun Yan (2013).
copula: Multivariate Dependence with Copulas. R package version
0.999-8. URL http://CRAN.R-project.org/package=copula

Jun Yan (2007). Enjoy the Joy of Copulas: With a Package copula.
Journal of Statistical Software, 21(4), 1-21.
URLhttp://www.jstatsoft.org/v21/i04/.

Ivan Kojadinovic, Jun Yan (2010). Modeling Multivariate Distributions
with Continuous Margins Using the copula R Package. Journal of
Statistical Software, 34(9), 1-20. URL
http://www.jstatsoft.org/v34/i09/.

Marius Hofert, Martin Maechler (2011). Nested Archimedean Copulas
Meet R: The nacopula Package. Journal of Statistical Software, 39(9),
1-20. URL http://www.jstatsoft.org/v39/i09/.
```

and

```
Marius Hofert, Ivan Kojadinovic, Martin Maechler and Jun Yan (2013).
copula: Multivariate Dependence with Copulas. R package version
0.999-8. URL http://CRAN.R-project.org/package=copula

A BibTeX entry for LaTeX users is

@Manual{,
  title = {copula: Multivariate Dependence with Copulas},
  author = {Marius Hofert and Ivan Kojadinovic and Martin Maechler and Jun Yan},
  year = {2013},
  note = {R package version 0.999-8.},
  url = {http://CRAN.R-project.org/package=copula},
}
```

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, e.g. 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 (2013):

```
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 \R\ and the \R-
    package \textbf{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).

```
### fast rejection algorithm, R version #################################

##' Sample a vector of random variates St ~ \tilde{S}(alpha, 1,
##' (cos(alpha*pi/2)*V_0)^{1/alpha}, V_0*I_{alpha = 1},
##' h*I_{alpha ≠ 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
##' @author Marius Hofert, Martin Maechler
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)
    ## case alpha == 1
    if(alpha == 1 || n == 0) return(V0) # alpha == 1 => point mass at V0
    ## else 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).

external Files (.txt, .Rd, .pdf)

.txt General description of the code or package. Also special dependencies on other packages or required software tools such as gsl should be explained in such files. Often naming: Description.txt, README.txt, install.txt

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

.pdf Manuals generated from the help files or vignettes.

5.3 Programming style

5.3.1 Writing correct 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.

Bad:

```r
res <- if(x > 0) {
    "positive"
}
else {
    "non-positive"
}
```

Good:

```r
res <- if(x > 0) {
    "positive"
} else {
    "non-positive"
}
```

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 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 5.3.3.

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 2.5, 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 camel case, so `myFunctionName <- function() {}`.

Omit superfluous parts, for example, a function `lengthOfObject(object)` which determines the length of a given object is preferrably 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, so use `rnorm(10, mean=3, sd=2)` instead of `rnorm(10, 3, 2)`.

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:

```
if(a < 0) { # first level
    b ← 1
    d ← 2
} else {
    if(e == 0) { # second level
        b ← 0
        d ← 3
    } else {
        b ← 2
        d ← 2
    }
```

```
}
bd <- c(b, d)
```

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:

```
n <- 100
set.seed(271)
stnrm <- lapply(1:10, function(i) rnorm(i))
t4    <- lapply(1:10, function(i) rt(i, df=4)) # vertically align the '<-'
```

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:

```
x<-1
optimize(function(x) x*x,interval = c(1, 10),maximum = TRUE)
```

Good:

```
x <- 1
optimize(function(x) x*x, interval=c(1, 10), maximum=TRUE)
```

5.3.3 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.

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 `NA`s do not use `x == NA` but `is.na(x)`.

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
x <- numeric(100)
for(i in 1:100) x[i] <- i*i # first creating and then filling an object
```

Good:

```r
x <- sapply(1:100, function(i) i*i) # directly create the correct object
x <- unlist(lapply(1:100, function(i) i*i)) # typically faster
x <- 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:

```
myOpt <- function(x)
    optim(c(-1.2, 1), fn=function(z) x * (z[2]-z[1]^2)^2 + (1-z[1])^2)$par
```

Good:

```
myOpt <- 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 `myOpt()` above more flexible.

```
myOpt <- function(x, init=c(-1.2, 1))
    optim(init, 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, `myOpt()` 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.

A **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:

```
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.

5.3.4 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).

5.3.5 Test your 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. See treatment of missing values and the section about error handling and warnings.

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`.

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, replace

```
        rmat <- NULL
        for( i in 1:n) {
                rmat <- rbind(rmat, some.further.computation.depending.on.i)
        }
```

by

```
        rmat <- matrix(0., n, k)
        for( i in 1:n) {
                rmat[i, ] <- some.further.computation.depending.on.i
        }
```

Use `TRUE` and `FALSE`, not 'T' and 'F' It can cause conflicts. For example if you have a variable T, which has the value 0 (`T <- 0`) (e.g. a test statistic) and call a function by `fct.a(data =data, log=T)` instead of `fct.a(data=data, log=TRUE)`. Since T is zero the function will return a result with `log=FALSE` ($0 \hat{=}$ FALSE, $1 \hat{=}$ TRUE).

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 ≤ 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. To export tables from R to LaTeX one can use the R package `xtable`, 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 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` 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`), ggplot2 graphics (based on the R packages `ggplot2`), 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.

6 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), display the *status* of a file (is the file *under version control* or not) and *diff*erences 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):

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

As example URLs, see

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

```
svn co <url> output_Dir_name # check out the project into a certain directory
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:

```
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 this you have to commit your changes, see Section 6.2.3, essentially propagating your changes to the repository. Furthermore, let us remark that only important files should be added, for example, only add `.tex` files, but not the corresponding `.pdf` or the myriad of files generated on the way.

Similar to adding files, (re)moving a file or folder from the working copy can be done via:

```
svn mv foo bar # moves 'foo' to 'bar'
svn rm foo # removes 'foo'
```

Again, do not forget to commit afterwards.

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:

```
svn up # update working copy (does not overwrite local changes)
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:

```
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:

```
svn diff # display diff between local files and working copy
svn diff -r 110:117 # diff of (older) version 110 to (newer) 117
svn diff -r 110 foo # diff of (older) version 110 to working copy of ``foo''
```

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.r<last>” (the original file you worked with; “<last>” denotes the version number), “foo.r<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
>>>>>>> .r<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 offline 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`. We omit further details here.

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, e.g., *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 (e.g., medicine), the method (e.g., time series analysis) or the goal (e.g., 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[4]. 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 journals 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 LaTeX 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).

[4] Wikipedia contributors, "Journal ranking," Wikipedia, The Free Encyclopedia, `http://en.wikipedia.org/w/index.php?title=Journal_ranking&oldid=608376667` (accessed June 18, 2014).

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 (e.g., `ps2pdf` or `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 `.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 formating, 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.

Acknowledgements

We would like to thank our students and the many people which have sent us code in the past for inspiring this guide. Also, we would like to thank the following friends and colleagues for feedback on these guidelines: Matthias Kirchner (ETH Zürich), Aleksey Min (Technische Universität München), Matthias Scherer (Technische Universität München).

References

Davidian, M. (2005), Academic publication: Journals and the journal editorial process, Department of Statistics, North Carolina State University, `http://www4.stat.ncsu.edu/~davidian/st810a/journals_handout.pdf`.

Halmos, P. (1970), How to write mathematics, *Enseign. Math.* 16(2), 123–152.

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

Hofert, M. and Mächler, M. (2014), Parallel and other simulations in R made easy: An end-to-end study.

Ligges, U. (2006), Help Desk: Accessing the sources, *R News*, 6(4), 43–45, `http://CRAN.R-project.org/doc/Rnews/Rnews_2006-4.pdf`.

Oetiker, T., Partl, H., Hyna, I., and Schlegl, E. (2011), The Not So Short Introduction to LaTeX 2_ε, `http://mirror.switch.ch/ftp/mirror/tex/info/lshort/english/lshort.pdf` (2014-01-26).

R Development Core Team (2006), Writing R Extentions, R Foundation for Statistical Computing, Vienna, Austria, `http://www.R-project.org`.

Ritter, R. (2002), The Oxford Guide to Style, Oxford University Press.

Schepsmeier, U. (2013), Efficient goodness-of-fit tests in multi-dimensional vine copula models, submitted for publication, `http://arxiv.org/abs/1309.5808`.

Venables, W. N., Smith, D. M., and R Core Team (2012), An Introduction to R, `http://cran.r-project.org/` (2013-02-03).

Wainer, H. (1993), Visual Revelations: Tabular Presentation, *Chance*, 6(3), 52–56.