Journal Name

ARTICLE TYPE

Cite this: DOI: 00.0000/xxxxxxxxxx

# ASpecD: A Modular Framework for the Analysis of Spectroscopic Data Focussing on Reproducibility and Good Scientific Practice

Till Biskup,*[a‡] and Jara Popp[a]

Received Date
Accepted Date

DOI: 00.0000/xxxxxxxxxx

Reproducibility is at the heart of science. However, most published results usually lack the information necessary to be independently reproduced. Even more, most authors will not be able to reproduce the results from a few years ago due to lacking a gap-less record of every processing and analysis step including all parameters involved. There is only one way to overcome this problem: developing robust tools for data analysis that, while maintaining a maximum of flexibility in their application, allow the user to perform advanced processing steps in a scientifically sound way. At the same time, the only viable approach for reproducible and traceable analysis is to relieve the user of the responsibility for logging all processing steps and their parameters. This can only be achieved by using a system that takes care of these crucial though often neglected tasks. Here, we present a solution to this problem: a framework for the analysis of spectroscopic data (ASpecD) written in the Python programming language that can be used without any actual programming needed. This framework is made available open-source and free of charge and focusses on usability, small footprint and modularity while ensuring reproducibility and good scientific practice. Furthermore, we present a set of best practices and design rules for scientific software development and data analysis. Together, this empowers scientists to focus on their research minimising the need to implement complex software tools while ensuring full reproducibility. We anticipate this to have a major impact on reproducibility and good scientific practice, as we raise the awareness of their importance, summarise proven best practices and present a working user-friendly software solution.

## 1 Introduction

Newton's famous sentence "If I have seen further it is by standing on y$^{e}$ shoulders of giants"[1,2] reveals a basic assumption about science probably every scientist will share: Progress in science is always based on the results obtained by others prior to us. Therefore, these results should better be correct, and to ensure this, they need to be independently reproducible and reproduced. Hence, reproducibility is a key aspect of science.[3–7] This emphasis on independent testability can be traced back at least to Karl Popper and his critical rationalism, particularly his notion that scientific hypotheses can never be proven, but only falsified.[8]

Defining the term "science" has been notoriously difficult,[9] although it has been characterised as a systematic endeavour early on.[10] However, key to empirical science and to what is known as the "scientific method"[11] is a structured sequence of events summarised in Fig. 1. Everything starts off with an observation that catches our curiosity, resulting in an approach to quantify the effect, *i.e.*, a measurement. The data thus acquired need to be displayed first,[12] before we can think of any model to describe them. Finally, after having modelled the data, we can present the results, leading to new observations that start again the whole cycle. Without properly documenting each of the steps, no reliable scientific knowledge can be built upon it.

With the advent of computers in the experimental research, the amount of data has grown exponentially, and today, data analysis is mostly computer-based. Therefore, the approach of a hand-written lab book to document each processing step is at best impracticable. The result: Most scientists will have hard time to trace a figure in their own publications back to the original dataset and give a detailed and gapless account of each processing step that eventually led to the representation.[13–15] This poses a severe problem to science,[16] rendering the results obtained this way eventually unscientific.[17]

Another problem arises from scientists being more and more forced to write their own software, whereas they have never been

[a] *Physikalische Chemie, Albert-Ludwigs-Universität Freiburg, Freiburg, Germany. E-mail: research@till-biskup.de*

‡ Present address: Physikalische Chemie, Universität des Saarlandes, Saarbrücken, Germany.

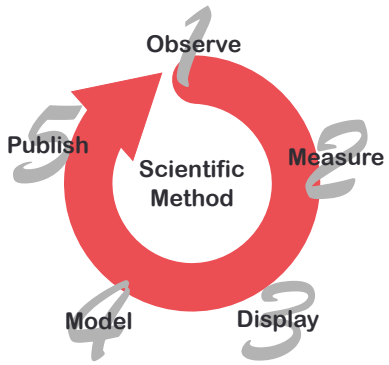


**Fig. 1** Workflow of empirical science following the scientific method. Nowadays, each of the steps two to five involve some help of computers and often programming. For details see the text.

trained how to do so in a reliable fashion.[18] The results obtained by such software are often irreproducible and sometimes even inadvertently wrong, depending on the complexity of the matter.[19] Teaching essential aspects of software development, besides some very introductory courses in programming, basically has no part in the curricula of most science courses.[6,20–22] However, knowing and following a few basic rules both, in writing software for scientific data analysis as well as in analysing data using such software, can greatly enhance the reproducibility and in turn lead to better and more reliable research.[23–25]

There are some investigations into how far scientific results are reproducible and correct,[16,19,26,27] as well as a poll on the topic.[28] Another aspect often discussed is the possible contributions of journals and publishers towards a greater availability of research by encouraging authors to provide both, data and source code, together with the publication of the results.[29–31] However, just publishing the source code of analysis software along with the raw data and results is not sufficient. Rather, independent reproducibility needs to be a design goal early on in data handling and analysis.[32]

Much has been written about the underlying general problem.[6,15,18,20,22–25,33–35]. For different scientific disciplines, the specific problems arising from computational science for reproducible research have been described,[36–38] and some areas have developed tools that are reasonably widespread.[39] Furthermore, there exist short contributions summing up "best practices"[24,40–46]. However, many of these articles have appeared either in engineering journals or in PLoS Computational Biology. The authors are, however, not aware of any publication in a journal regularly consulted by spectroscopists. Furthermore, no software package seems available addressing the aforementioned problems in their area of research.

To this end, we present here a modular framework for the *a*nalysis of *spec*troscopic *d*ata, termed ASpecD, built upon more than a decade of personal practical experience.[47–49] The result is a number of best practices and design rules for both, obtaining data and developing computer programs for their analysis. The ASpecD framework focusses on usability, small footprint and modularity, providing the user with a maximum of personal freedom while ensuring reproducibility and good scientific practice and tremendously reducing the effort necessary to achieve these goals. Few such systems have been described before, perhaps most prominent is the work of Claerbout and his group in geology.[50,51] However, to the best of our knowledge, no such system exists for spectroscopic data. Often spectroscopists seem even not to be aware of the importance of systematic and reliable tools for data processing and analysis, or they (understandably) lack ideas how to practically achieve the required reproducibility.

Two key concepts of the ASpecD framework are the unified representation of (numerical) data and accompanying metadata in a dataset independent of the original raw format and the possibility to analyse data without having to program a single line using "recipe-driven data analysis". Here, the user only provides a list of datasets and tasks that should be performed on these datasets, including processing, analysis, and representation of the data.

After shortly discussing the aims and requirements of scientific software development in general, we introduce the basic concepts and components of a framework for reliable and reproducible analysis of spectroscopic data, followed by highlighting a few essential aspects of its general implementation. Afterwards, two examples for packages based on the ASpecD framework are given and finally the relation to a complete workflow including aspects such as data storage and unified access is shortly discussed.

## 2 Scientific software development

As a matter of fact, programming skills are a fundamental requirement for many scientists nowadays, particularly in but not restricted to the field of spectroscopy. Furthermore, the role of scientific thought in software development has been highlighted early on.[52] Therefore, scientists should be well equipped to develop reliable software for their own use if they would care as much about it as they do about their experiments and theories.[6]

### 2.1 Aims and prerequisites

Aspects central to scientific data analysis aiming at reproducibility and good scientific practice are summarised in Fig. 2 and will be detailed below. While originating from the context of scientific software development, the criteria listed here are true for all kinds of data analysis, regardless whether computers are involved.

#### 2.1.1 Reproducible

The eventual goal of a framework for spectroscopic data analysis is the full reproducibility from the raw data to the final publication and *vice versa*. The latter means to start from a published figure (or table) and trace it back to the original data, including the complete provenance and history of the processing steps involved.[50,51]

Putting aside such aspects as long-term archival, unified access to data, and availability of the required software including the underlying operating system and compatible hardware,[17] we focus here on design rules of a modular framework taking care of the provenance of eventually published data, *i.e.* the history of processing steps from the raw data up to the final representation and publication. Hence, the strategies discussed are entirely within the realm of the individual scientist's work and personal

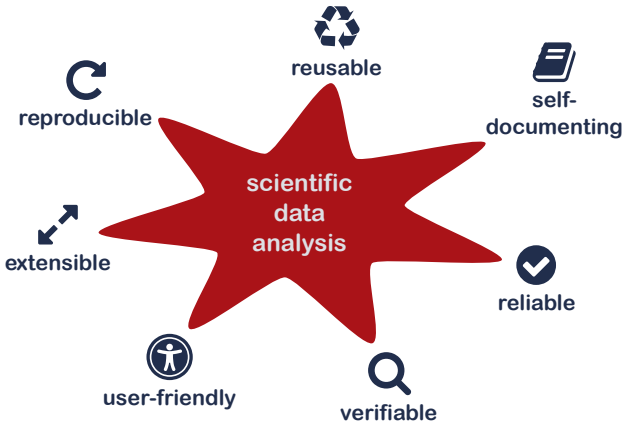


**Fig. 2** Aspects central to scientific data analysis aiming at reproducibility and good scientific practice. While originating from the context of scientific software development, the criteria listed here are true for all kinds of data analysis, regardless whether computers are involved. For details see the text.

responsibility.

We note that there is some inconsistency in the literature regarding the usage of the two terms "reproducible" and "replicable", even between different disciplines, that have been addressed recently.[53–55] Here, we focus on the workflow from the data acquisition to the final publication of the analyses of the results and the traceability in both directions.

From a software development perspective, two aspects are crucial for obtaining reproducible results: a version control system (VCS) in conjunction with a scheme for unique version numbers and making the source code of the programs available (open source). Using a VCS is of utmost importance, as frequently highlighted in descriptions of *best practices* for scientific software development.[40] In the world of professional software development, there is no dispute about the importance of using a VCS,[56–59] such as git.[60].

Providing the source code of software underlying scientific data analysis open source and free of charge is, as far as possible, a crucial step towards reproducible research and intellectual honesty.[31,61–64] From a scientific point of view, there is no good reason *not* to publish your source code, and the worst of these reasons is that it is not "good" enough.[65,66] However, source code can not always be published entirely, particularly if commercial programs are involved in data analysis, both as programming languages as well as tools.[67] Furthermore, while open source is a necessary requirement, it is not sufficient on its own to allow for reproducibility.[32] In particular, appropriate metadata accompanying the raw data and the processing and analysis steps performed on them are of outstanding importance.

### 2.1.2 Reusable

Every question addressed by scientists is new in a certain sense. However, often a series of routine steps is carried out that should be performed automatically as much as possible. Furthermore, code reuse is quite important, as the main focus of (most) scientists is not on developing software, but on performing research using software. Reusing existing components and combining them in new ways is a very general theme not only in science and a matter of efficiency. To achieve this goal, software needs to be as modular as possible and sensible. Therefore, it needs to consist of cohesive and loosely coupled components.[56,68] To design a modular system, regardless of the actual context, requires abstraction.[69] Abstraction in turn requires a solid understanding of the "problem domain". Software development should be driven by solving problems of the problem domain, as highlighted by Evans coining the term "domain-driven design".[70] All this is well-known and at the heart of software engineering, as presented in numerous books throughout the last decades.[56,57,71–76] Many different tools have been developed to aid with finding appropriate abstractions and to implement them in software, starting with the now fundamental paradigms of structured,[77,78] object-oriented[79] and functional programming[80] all the way up to complicated frameworks for developing mobile apps, hiding most of the underlying complexity from the developers.

From another angle, reusing software components requires a solid understanding of the individual modules. This is even more true in science, where it is usually not sufficient to use tools that give just about sensible results, but to know their exact specifications and range of validity. Therefore, scientific software needs to be as readable and understandable as possible, as discussed in the next section.

### 2.1.3 Self-documenting

The term "self-documenting" has two meanings in the given context. From a scientific point of view, all parameters of the data processing and analysis should be (automatically) logged. This is at the heart of reproducibility and traceability. However, with the advent of computer-aided data processing, this aspect seems to have been gone lost. As a result, reproducibility of results published in the scientific literature is currently probably worse than it was previously, with tremendous impact on the reliability of science as such.[17]

From a programming perspective, self-documenting implies the code to be readable on its own, ideally without the help of comments within the source code. Many different metaphors for programming have been brought forward, and the one seeing the programmer as an author might be most useful here.[81] This culminates in the advice for the software developer: "Write programs for people, not computers".[40,82] Other scientists are the audience, and they should be able to read the code and make sense of it, even without too much prior knowledge of the programming language used. However, this is not restricted to science, but is a general theme highlighted over the last decades of professional software development[56,72,81] and recently summed up and advocated for under the term "clean code".[76,81]

### 2.1.4 Reliable

A basic requirement for software, particularly in science, should be correct and functioning code that is flexible and comprehensible.[17] This requires rigorous tests of the software during its development. The idea of tests is, however, not to prove the program to work correctly, as this is mostly impossible based on logi-

cal grounds. Tests aim at finding errors in existing software.[17] While the importance of tests is not disputed and often highlighted,[56,57,59,74,83–85] testing software is a difficult matter.[86] This led some prominent professional programmers with decades of experience to inverting the process of writing functional code and corresponding tests. The idea of "test-driven development" is to first write a (failing) test and afterwards the piece of software necessary to make it pass.[87,88] Used properly, this leads to software that is much more reliable and of higher quality. Important is to test already the smallest building blocks of software using automated unit tests and a testing framework to help with this.[88–90]. On aside, software that has been developed with testability in mind is intrinsically more modular, thus aiding with reuse and other of the aspects highlighted here. Furthermore, a high test coverage allows to enhance code readability without affecting the functionality, as advocated under the term "refactoring".[84]

#### 2.1.5 Verifiable

Strictly speaking, scientific hypotheses cannot be verified in sense of a formal proof, but only be falsified.[8] The same is true for most software. Both, the the general impossibility to formally prove most software,[91] as well as the general lines of Popper's thinking on science,[8] is rooted in Gödel's discovery of the fundamental limits of proofs in formal logic and mathematics.[92]

However, data analysis as well as the programs used for it should be verifiable in a sense that each step is fully transparently documented and all necessary information available required for an independent audit. This includes all parameters as well as the access to the very version of a software used.

From a programming perspective, a few more aspects are important. First of all, the source code should be readable and understandable on its own ("clean code").[81] Furthermore, underlying concepts should be documented. While ideally, source code can be read and understood without additional comments, the underlying concepts and the interplay between different parts of the code need to be independently documented, *e.g.*, in form of a user manual. This is particularly true for the implementation of complicated algorithms. Eventually, (automated) tests are a good way to ensure a routine to work reasonably correct, at least as far as the behaviour is covered by the tests.

#### 2.1.6 User-friendly

Aspects that might be less obvious on first thought but eventually decide whether a system will be adopted in practice can be summarised under the term "usability". Only a system that behaves intuitively and is easy to use will be used on a daily basis. As any system aiming at providing a gap-less record of data processing requires additional work and discipline of its users, the obvious benefits from using the system need to clearly outperform the additional effort in order to not undermine the very reason it has been implemented in the first place.

Aspects from a programming perspective include intuitive and stable interfaces, robust code that gracefully handles errors as well as wrong user input, a proper user manual, and the habit of the developers to listen to the users and their needs.

#### 2.1.7 Extensible

Any framework for scientific data analysis needs to be highly modular and extensible, as frequent changes in the requirements and questions adressed to the data are at the heart of research. Many aspects of data analysis are routine tasks. However, a key aspect of scientific research is to creatively combine the available tools in new and unforeseen ways. This is why software as a separate component of computers was developed originally and where its name stems from. To be able to adapt a system to new requirements in a highly flexible manner by (only) changing software while using general-purpose standard hardware was a necessary requirement for the unprecedented trail of success of computers that has revolutionised human life ever since.[69]

From a programming perspective, this translates to highly modular code where every routine performs only one task. Furthermore, code needs to be easy to read and understand, as it will be read far more often than it is written. Automated tests are a necessary prerequisite for extending code and adding functionality, as they allow the programmer to make changes with reasonable confidence to not break existing functionality. Similarly, a VCS greatly facilitates expanding software. Finally, extensible software requires the underlying concepts as well as the implemented interfaces to be properly documented, most probably in terms of an up-to-date user and developer documentation.

### 2.2 Further aspects

Besides the criteria mentioned above, additional prerequisites are immediately obvious from a software implementer's perspective: Due to the heterogeneity of the IT infrastructure in science, platform independence is imperative. Furthermore, such a system needs to be easy to integrate into existing workflows of data processing. Here, modularity is a great benefit, making it possible to start small using (parts of) the system without risk. Scientists are not software engineers, and software development will never receive the attention it might deserve in science. Hence, adopting the system "on the go" while still focussing on the actual science will be the likely usage scenario the system has to account for and adopt to. Additionally, using well-developed and freely available programming languages and making the source code available help to maintain and further develop the system even if the original developers are no longer available.

## 3 Reproducible data analysis: The ASpecD framework

To achieve the aforementioned goals, a framework for the *A*nalysis of *Spec*troscopic *D*ata (ASpecD) has been developed. The framework itself is implemented in Python and available open-source and free of charge.[93] Therefore, it can be used out of the box to derive processing and analysis routines for the specific data at hand. At the same time, it can serve as starting point for own developments aiming at better reproducibility. Therefore, we will first give an overview of its basic concepts and components. Afterwards, its user interface that is radically different to all approaches currently employed in scientific data processing will be introduced: "recipe-driven data analysis".

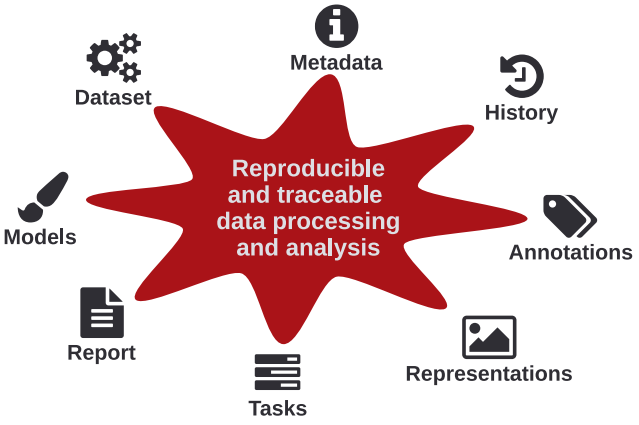


**Fig. 3** Basic concepts and components of the ASpecD framework. One key concept is the dataset containing both, (numerical) data as well as information about these data (termed metadata). At the same time, a dataset contains a history of each processing step performed on its data together with the full set of parameters necessary to reproduce it. Representations are stored as abstract descriptions (another form of metadata), and annotations are used to store information that cannot be obtained by any automatic analysis, but requires human intervention. Reports are finally a way to obtain a well-formatted overview of certain or all aspects stored within a dataset. Tasks are at the heart of recipe-driven data analysis and can be thought of as abstract summary of any action taken on a single dataset or a defined list of datasets. Models are parameterised mathematical descriptions of the numerical data that can be fitted to the latter. For details see the text.

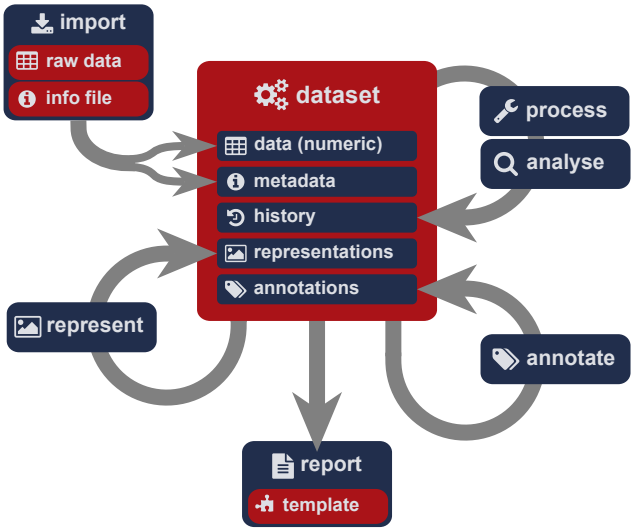


**Fig. 4** A dataset-centric view of the interplay of the different components of the ASpecD framework. Key is the dataset as a unit consisting of (numerical) data, metadata, and a full history of each processing and analysis step performed on the (numerical) data. Metadata of both, representations and annotations are stored as well within the dataset. The difference between processing and analysis steps is that the former modify the data of a dataset, whereas the latter return an independent result. "info file" refers to a specific textual format for storing metadata accompanying a measurement. Its file format was developed to be easily (human) writable while retaining machine readability. For details see the text.

### 3.1 Basic concepts and components

A graphical overview of the basic concepts and components of the ASpecD framework is given in Fig. 3. Each of the components will be discussed hereafter in some detail. Their interplay is graphically represented in Fig. 4. All parts have been implemented in a highly modular fashion (for implementation details see below) and can readily be used to develop packages for processing and analysis of spectroscopic data. Developers can entirely focus on implementing the science rather than ensuring reproducibility that is taken care of by the framework. Users, on the other hand, do not require any programming skills at all thanks to recipe-driven data analysis.

#### 3.1.1 Dataset

The dataset is one key concept of the ASpecD framework, storing (numerical) data, metadata and the history of processing and analysis steps in one place. One can think of it as an entity containing a field for the numerical data, a structured array (called associative array, dictionary, map, hash, or struct, depending on the programming language, but essentially providing a key–value store) for the metadata, and a list of processing and analysis steps, *i.e.* their parameters and all information necessary to reproduce such a step, forming the history. If stored as a single unit, a dataset becomes the independent smallest component of the ASpecD framework, aiding much to reproducibility while maintaining modularity and a small footprint, *i.e.* a minimum of external dependencies.

Furthermore, the dataset serves as universal exchange format within the ASpecD framework: While each measurement software will come with its own file format and metadata organisation, the dataset abstracts from this diversity and represents a unified interface for all routines of the framework. This even allows to mix datasets of different origin, *e.g.* plotting the data together in one figure. Starting from the initial import of the data and metadata, every routine operates exclusively on datasets, thus maintaining full reproducibility by automatically logging processing and analysis steps in the history and representations and annotations in their respective lists. At the same time, using the dataset as universal exchange format for all the different routines of the framework and its derived programs dramatically simplifies the interfaces of these routines. Additionally, this provides all functionality necessary to fulfil the requirements for a reproducible, modular and extensible data analysis.

#### 3.1.2 Metadata

Numerical data on their own are rather useless if they are not accompanied by additional information, here termed metadata. This insight is central to the development of the ASpecD framework. For a start, metadata of a measurement are all the information available at the very moment of the data acquisition, such as spectrometer type and software version, sample details and temperature of the setup. Hence, we need a way to record this information in an organised fashion, using a format that is both, human and machine readable. In its simplest form, this may just be a bare text file. Details will be discussed later on. Furthermore, it is the responsibility of the individual scientist to reduce the vast amount of information available during data acquisition to the relevant aspects. However, important for now is the inti-

mate connection between data and associated metadata, as well as recording the metadata *during* data acquisition. The latter is crucial, as we cannot rely on our memory, not to mention that all too often, we will never provide the required information later on.

#### 3.1.3 History

The key to reproducibility is to automatically log all tasks that are performed on a given set of data. Not only the processing and analysis steps as such, but as well a *complete* set of parameters and their values, including implicit or default parameters, need to be recorded. The version number of the routines, information about the underlying operating system, a timestamp, and the name of the operator are added as well, resulting in a gap-less history of processing steps on a given set of raw data. Additionally, this history gets stored such that it can be used directly to replay every step automatically. This opens the way for repeating (and systematically varying) the processing and analysis on the original dataset as well as applying it to other sets of data in a similar fashion, greatly assisting with comparing results and analyses.

At the same time, this allows for a straight-forward implementation of an undo/redo mechanism. An additional read-only field contains the original raw data of the dataset. This prevents the application from having to reacquire the raw data from wherever they were loaded originally. Thus, reverting a processing step can be translated into starting off from the raw data and performing each processing step stored in the history except for the last one. Similarly, not immediately throwing away the metadata of the reverted processing step allows for its easy reapplication, termed "redo" in most software applications.

#### 3.1.4 Annotations

Whereas many routine processing steps on data can be carried out in a fully automated fashion, the actual data analysis always involves human intervention, at least with interpreting the results of preceding data processing and analysis steps. Often, there is the need to store additional information about a dataset, be it a comment applying to the entire data or highlighting an area or a single point. This is what annotations are meant for. Whereas the term is borrowed from bioinformatics and here from annotating DNA and protein sequences, it can be easily applied to spectroscopic data as well. In the context of the ASpecD framework, an annotation is characterised by its intrinsic human origin: The information contained cannot be (re)created in an automated fashion. Hence, excessively annotating datasets imposes severe constraints to the way the information is stored, regarding data safety (against accidential losses) as well as security (restricting access to the information).

#### 3.1.5 Representations

A picture is worth a thousand words. Mostly, data are represented in a graphical or tabular manner in a publication. Whereas creating good figures is a topic on its own,[12,94,95] ideally, representations should be fully reproducible by replaying a "recipe" stored in form of an instructional text file containing all required parameters and information. This abstract description of a graphical or tabular display of (characteristics of) a dataset is termed representation in the ASpecD framework. If a representation depends solely on the data contained in a single dataset, its metadata will usually be stored within the dataset.

#### 3.1.6 Reports

Finally, all the information collected by using the framework and contained in a dataset needs to be accessed in simple yet powerful ways. This is the realm of reports. Automatically generated from a dataset as source of structured information, they can be fully customised, resulting *e.g.* in a LaTeX file that gets automatically rendered to a PDF document. A key aspect is to separate formatting and content using template engines.[75] This allows for using a large list of different output formats fully independent of both, the underlying data source, in the given context a dataset, and the programming language used for data analysis as well as report generation. Similarly, reports of identical content but in different (human) languages can easily be created. Additionally, writing and modifying templates needs next to no programming experience and is fully independent of the other aspects of the data analysis framework. Hence, template engines are a very powerful application of the concept known as "separation of concerns".[69,71]

#### 3.1.7 Tasks

Processing data consists of lots of different single tasks that can mostly be automated. This is the idea behind *recipe-driven data analysis*: lists of datasets and tasks that can easily be created by a user and processed fully automated. Here, "tasks" has a broad meaning, covering every aspect of data analysis that can be automated, such as processing and analysis steps, creating representations and annotations, and finally reports. For details, see the next section.

#### 3.1.8 Models

To make sense of and to interpret the physical reality reflected in numerical data, usually mathematical models are used. These models typically depend on a number of parameters that may or may not be adjustable to fit the actual data. Models can therefore be seen as abstraction to simulations. In this respect, they play a central role in conjunction with fitting models to data by adjusting their respective parameters, a quite general approach in science and particularly in spectroscopy. Therefore, models provide the interface towards a fitting framework that is currently being developed in the authors' group.

## 4 No programming skills needed: Recipe-driven data analysis

The components previously described can all readily be used, adopting the perspective of a programmer. Therefore, they can form the basis for own scripts and programs as well as whole graphical user interfaces, while providing all the prerequisites for fully reproducible data analysis. Still, the user would need to be familiar with the underlying programming language to interconnect the different functions in a short program. In contrast, recipe-driven data analysis provides a wholly different level of abstraction, relieving the user from needing any programming skills. For a first impression, see Listing 1.

**Listing 1** Example of a recipe used for recipe-driven data analysis within the ASpecD framework. Here, a list of datasets is followed by a list of tasks. The user needs no programming skills, but can fully focus on the tasks to be performed. "Cooking" this recipe is a matter of issuing a single command on the terminal. For comparison, Listing 2 shows the Python code of the same analysis carried out fully programmatically instantiating objects of the respective classes.

```
datasets:
  - /path/to/first/dataset
  - /path/to/second/dataset

tasks:
  - kind: processing
    type: BaselineCorrection
    properties:
      parameters:
        kind: polynomial
        order: 0
  - kind: singleplot
    type: SinglePlotter1D
    properties:
      filename:
        - first-dataset.pdf
        - second-dataset.pdf
```

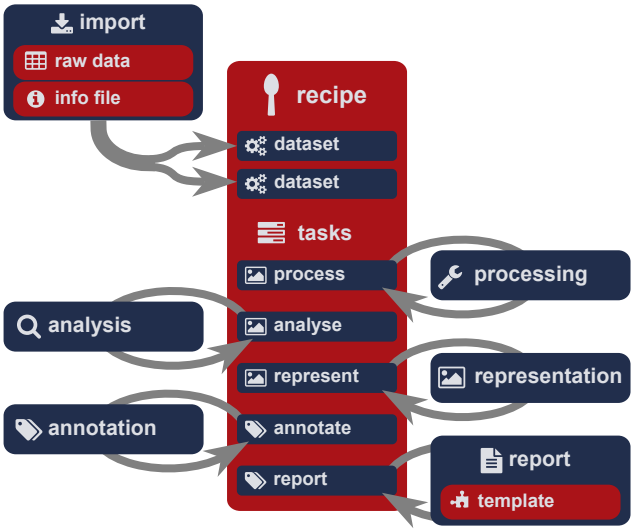


**Fig. 5** Recipe-driven data analysis as implemented within the ASpecD framework. Recipes provide a level of abstraction that empower users not familiar with programming to describe the details of arbitrarily complex data analysis step-by-step in a formalised way. They can be thought of as a special kind of user interface. A recipe consists of a list of datasets to operate on and a list of tasks that shall be performed. Tasks can be everything implemented within a package based on the ASpecD framework, from simple processing steps to complex reports. For details see the text.

Basically, recipe-driven data analysis can be thought of a special type of user interface to the ASpecD framework and derived packages. The normal user of such packages has a clear idea how to process and analyse data, as this belongs to the realm of science. At the same time, however, the user may not necessarily be interested in actually programming a lot. Furthermore, reproducible science requires the history of each and every processing and analysis step to be recorded and stored in a way that can be used and understood long after the steps have been carried out (think of many years rather than weeks or months).

From the user's perspective, all that is required is a human-writable file format and a list of datasets followed by a list of tasks to be performed on these datasets. For each task, the user will want to provide all necessary parameters, *i.e.*, the metadata of the data analysis. Introducing the metaphor of recipe and cook circumvents using the term "metadata" in different meanings and the confusion this might cause. A recipe is a list of datasets and tasks to be performed on them. Such recipe is processed by a cook invoking the respective routines for each task. This is the origin of the term "recipe-driven data analysis". An actual example of how such a recipe may look like is given in Listing 1. Of course, this is only a very simple example, and further details can be found in the online documentation[93] and the Supporting Information. "Serving" the results of this recipe is as simple as issuing a single command on the terminal.

Essential aspects of the recipe-driven data analysis are detailed below and a graphical representation, contrasting the dataset-centric view, is shown in Fig. 5. Eventually, this abstraction renders scientific data analysis programing language agnostic and provides a complete new level of reproducible research. From own experience, the dramatically simplified user interface has the potential to revolutionise the way spectroscopic data are analysed, paving the way for truly reproducible research.

### 4.1 Easy handling of multiple datasets

Besides providing an entirely new level of abstraction and a user-friendly interface to data analysis, recipe-driven data analysis overcomes an essential limit of the dataset-centric usage scenario shown in Fig. 4. namely its intrinsic limitation to operating on single datasets. However, data analysis often requires comparing different datasets. Therefore, history, representations and reports need to be abstracted from the single dataset and connected to a higher level, in this case the recipe allowing to span arbitrary numbers of datasets that should be processed and analysed together and compared in some way or other. Nevertheless, recipe-driven data analysis can at the same time be used in exactly the same way as the dataset-centric view, processing and analysing only one dataset at a time, but providing a much more user-friendly interface. Perhaps most crucial here is the lack of any programming skills needed, as the tasks are defined in textual, though structured way. This may be most obvious when comparing the elegance of the recipe given in Listing 1 with the corresponding Python code in Listing 2.

### 4.2 Reproducible and automated

Processing data consists of lots of different single tasks that can mostly be automated. This is the idea behind recipe-driven data analysis: lists of datasets and tasks that can easily be created by a user and processed fully automated. "Tasks" has a broad meaning here, covering basically every automatable aspect of data analy-

sis, including processing and analysis steps, creating representations and annotations, and finally reports.

Storing the (somewhat abstract) recipes rather than scripts consisting of code depending too much on implementation details helps with reproducibility. On the one hand, problems with other versions of the underlying programs should be less frequent, and on the other hand, a human-readable list of datasets and tasks with their respective parameters is much easier to understand than actual code. For a direct comparison of the two approaches, see Listings 1 and 2 as well as the accompanying discussion.

Automatisation comes with several advantages. Generally, everything that can be automated can be delegated to computers. That does not mean that the tasks carried out are necessarily correct, but they are usually consistent. And if mistakes in the input are detected, this can be easily fixed, resulting in a consistent (hopefully more correct) result. Additionally, every automated step saves the users from performing boring and error-prone routine tasks and allows them to use their brains for good—thinking of ways how to process and analyse the data and to make sense of the results, *i.e.* things computers cannot really do for us. The power of automatisation is nicely reflected in a quote by Whitehead in his "Introduction to Mathematics": "Civilization advances by extending the number of important operations which we can perform without thinking about them."[96]

The real important aspect of data analysis in science is to think about the data and the results obtained from automatically processing the (raw) data and to *interpret* these results. Everything else can (and should) be delegated to the computer as much as possible.

### 4.3 Fully unattended

Recipe-driven data analysis is carried out fully unattended (*i.e.*, non-interactive). This allows for using it in context of separate hardware and a scheduling system. Situations particularly benefiting from this approach are either many datasets that need to be processed all in the same way, or few datasets requiring expensive processing such as simulation and fitting[97]. The latter is even more true in context of global fitting and/or sampling of different starting parameters, such as Monte-Carlo[98] or Latin-Hypercube[99] sampling approaches.

Furthermore, this approach allows for decoupling the place the actual data processing will be carried out from the input. This leads naturally to containerisation (*e.g.*, docker[100]) where the actual packages derived from ASpecD are located in a container that is self-contained and could basically even be shared with others. [§] Particularly in case of the above-mentioned long-running fitting tasks, having the data analysis run not on the personal computer, but on dedicated hardware can be a great advantage.

§ Although an entirely different topic, containerisation would allow to even share a fully working installation of your analysis packages with collaboration partners. In such context, an easy-to-use and somewhat intuitive user interface as provided by *recipe-driven data analysis* will further help.

### 4.4 History

Processing a recipe always results in documenting all tasks performed. This includes the complete set of available information necessary to reproduce and replay the tasks, both parameters and version information of the actual routines and all package dependencies used. For ease of use, these protocols can be used again as recipes. Therefore, the history connected now with the recipe rather than a single dataset represents a generalisation of the history concept described above. Besides that, all tasks operating on datasets will still write a history on each of the datasets.

### 4.5 Human-writable

As the recipe is used as kind of a user interface, it should be as simple to write and read as possible. Besides that, the format should be platform-independent and longterm-stable. Therefore, a simple text format is the most natural choice.[58] The actual file format of the recipes is an unimportant detail. However, at least in a first implementation of the ASpecD framework, the YAML file format[101] is used. Being very easy to write and read by humans due to the minimum of formatting required is perhaps its biggest advantage. Furthermore, due to its hierarchical structure, it allows for formulating even complex tasks such as a graphical representation with all the details that can be controlled and explicitly specified. For more complex examples see the Supporting Information. This format has been proven useful in similar settings, *e.g.*, automatic provisioning of software using Ansible[102].

### 4.6 Comparison with programmatic approach

Both usage scenarios described above, the dataset-centric approach and the recipe-driven data analysis, have their particular strengths. The dataset-centric approach to the ASpecD framework is user-interface agnostic and hence compatible to application in context of an interactive graphical user interface (GUI). Recipe-driven data analysis, in contrast, provides a particular type of user interface aiming at an abstract description of the different tasks to be performed on a list of datasets. For a direct comparison of the two approaches, cf. Listings 1 and 2.

Generally, with reproducibility in mind, every kind of interactive graphical user interface usually poses a certain risk of lacking the complete history of events. This is leveraged using the dataset-centric approach and the dataset as universal exchange format between every routine. For a first overview of the characteristics of a single dataset, a graphical user interface clearly has its advantages, while for routine processing and comparative analysis of multiple datasets, the recipe-driven data analysis approach shows its full potential. Besides full reproduciblity spanning multiple datasets, recipe-driven data analysis is at its most impressive when coming to fully unattended and automated routine data processing and analysis and when comparing a larger list of datasets in various ways. Furthermore, from the user's point of view, recipe-driven data analysis is much simpler, as it does not require programming skills and allows to focus entirely on the scientific side of data processing and analysis.

**Listing 2** Example of a dataset-centric scenario of data analysis using the ASpecD framework. Here, objects of each of the respective classes need to be instantiated, the properties set appropriately and the interaction of the different classes directly implemented in code, calling the respective methods. For comparison, Listing 1 shows a recipe performing the same analysis.

```
import aspecd

dataset_filenames = [
    '/path/to/first/dataset',
    '/path/to/second/dataset'
    ]
figure_filenames = [
    'first-dataset.pdf',
    'second-dataset.pdf'
    ]

importer_factory = aspecd.io.ImporterFactory()
baseline_subtraction =
    aspecd.processing.BaselineCorrection()
baseline_subtraction.parameters = {
    "kind": "polynomial",
    "order": 0
    }
plotter = aspecd.plotting.SinglePlotter1D()

for idx, source in enumerate(dataset_filenames):
    dataset = aspecd.dataset.Dataset()
    importer = importer_factory.get_importer(source)
    dataset.import_from(importer)
    dataset.process(baseline_subtraction)
    plot = dataset.plot(plotter)
    saver = aspecd.plotting.Saver()
    saver.filename = figure_filenames[idx]
    plot.save(saver)
```

## 5 Implementation details

In contrast to engineering with a clear distinction between the design and the real object, in software development, the source code of the actual program is the design.[103] Furthermore, software is intrinsically subject to frequent changes, even more during active development. Therefore, the implementation details of the ASpecD framework laid out in the following are necessarily abstract to a certain extent. However, due to it being open-source and fully documented, the interested reader is invited to have a look at the source code itself.

Software development in general and scientific software development in particular is a topic filling entire volumes.[57,59,70,73–75,104] Hence we will only provide a very brief overview here of the important aspects that shaped the implementation of the ASpecD framework. Further details can be found in the Supporting Information. Generally, three aspects can be distinguished: the necessary infrastructure for development and usage, the general implementation, and the actual code providing the functionality.

Independent reproducibility is at the heart of the scientific method,[3–7] and hence all software developed for scientific data processing and analysis should be made available for the community.[31,61–64] This requires a permissive license allowing others to use and potentially modify the code.[58,59] As software is always in constant development, a version control system (VCS) such as git[60,105] is the next important requirement,[56–59] allowing to restore the exact version of the software that was used for a particular task. Only in conjunction with a scheme for version numbers can this be achieved, with one version number referring to one and only one actual state of the software. In mathematical terms, this is is a one-to-one or bijective mapping between version number and program version. ASpecD follows the semantic versioning[106] scheme. As software for scientific data analysis is intrinsically complex, even the ideal implementation where the code fully speaks for itself cannot convey the overarching design principles. Therefore, extensive documentation of both, the public interfaces (API) of the routines as well as the underlying concepts is essential for the software to be usable and to get used.[58,59] This can only be achieved by employing tools that automatically generate well-formatted documentation from the source code, such as Sphinx[107]. Finally, we need to ensure the software to be correct as much as possible. This requires automatic tests to be present together with a high test coverage.[56,57,59,74,83–86] The best way to achieve testability and high test coverage is by using a test-first approach[87,88] and automatic testing via a unittest framework[88–90].

With this infrastructure in place, implementing the actual software can begin. An essential prerequisite of the modularity and hence easy extensibility of the ASpecD framework is its use of the object-oriented programming paradigm.[108,109] Next is choosing a programming language that is easy to use for non-experts and comes at the same time with a large set of libraries for scientific data processing and an active user community. This is why Python[59,110–113] has been chosen for implementing the ASpecD framework, in conjunction with the NumPy,[114,115] SciPy,[116] and matplotlib[117] packages. A last aspect worth mentioning is file formats. The time scales of reproducibility in science and of the availability of digital file formats differ quite substantially. Nevertheless, scientific data need to be accessible after years and even decades. Therefore, only fully documented and open formats should be used. Furthermore, information about the numerical data (*i.e.*, metadata) need to be stored together with the actual numerical data that are the results of measurements or computations. Generally, structured text files should be used wherever possible[58], with the YAML format[101] providing a good compromise of being human-writable while at the same time being processable by machines. For storing floating-point numbers in binary format, the IEEE 754 standard[118] is an acceptable exception from this rule, as long as its use is restricted to numeric data. Key to reproducible science is to collect all necessary information (metadata) about the original measurement or calculation as well as each individual processing and analysis step. As long as this information is collected and saved in a structured way in open formats, converting it to exchange formats for deposit in public databases and sharing with other scientists is possible and can usually be automated as well.

The actual modules the ASpecD framework consists of are basically identical to its components described above. For more de-

tails, the reader is referred to the Supporting Information and the extensive documentation available online[93]. Both include details as well of how to write and develop analysis packages based on the ASpecD framework. Two such packages that are currently actively developed in the authors' group are shortly outlined in the next section.

# 6 Examples of concrete analysis packages

Currently, the implementation of analysis tools for data from both, time-resolved[48] and conventional continuous-wave[49] electron paramagnetic resonance (EPR) spectroscopy is under way in the authors' group, helping to further develop the framework. The results are provided open-source and free of charge, and two Python packages derived from the ASpecD framework focussing on EPR spectroscopy, namely the trepr[119] and cwepr[120] packages, are available online via PyPI and GitHub. Both packages are now actively employed in the authors' group on a daily basis. These packages define as a minimum an explicit description of the metadata relevant for the respective spectroscopic method, importers for the different vendor-specific file formats, and processing and analysis routines that can be based on existing functionality provided by the ASpecD framework. A particular highlight, demonstrating the modularity and validity of the approach taken with the ASpecD framework, is the capability of processing both, cw-EPR and tr-EPR data within one recipe and creating joint plots with both types of spectra. Details of these two packages will be published elsewhere.

# 7 Outlook

As with most software projects, developing the ASpecD framework is an ongoing process, and a few concrete ideas for further development are worth mentioning here. This involves both, additional features as well as embedding the ASpecD framework into the larger context of an open-source laboratory information system aiming at even fuller reproducibility.

## 7.1 Additional features

Besides implementing further generally applicable processing and analysis steps and providing pre-defined templates for each task to make report generation a lot easier, interfacing with a fitting framework is probably most important.

Fitting models to data is a frequent requirement in spectroscopy, and the Python SciPy package[116] provides excellent capabilities to this end. Developing a Python package ("FitPy") providing a unified interface to the different fitting strategies, including stochastic sampling of starting conditions, is currently under way in the authors' lab. Fitting is an analysis step, and the models to be fitted are derived from the abstract model class provided by the ASpecD framework. Independently, simulation routines for EPR spectra are developed (Python package "SpinPy"). Using FitPy together with SpinPy or alternative simulation routines[121,122] will allow to fully automatically process and analyse EPR data all the way from data acquisition to the final publication-ready figure.

## 7.2 ASpecD in context of a laboratory information system

Whereas the ASpecD framework focusses on reproducibility of scientific data processing, important aspects for a fully reproducible research such as long-term archival of data as well as unique identifiers for accessing data and metadata, though logical extensions, have been set aside here. Unique identifiers become important as soon as more than one dataset is involved in the analysis, and one possible solution may be a system similar to the digital object identifiers (DOI) familiar from scientific publishing. Other aspects include a fully automated and metadata-driven workflow for routine processing, starting with the raw data and ending with reports posted, *e.g.*, to an internal web page.

Due to its highly modular design, the ASpecD framework well integrates in such larger infrastructure. Existing ideas for a modular laboratory information system built from open-source components are currently actively implemented in the authors' group and will be detailed elsewhere.[123]

# 8 Conclusions

Reproducible research as well as adhering to good scientific practice is of high demand. Here, we have presented a set of best practices as well as design rules for scientific software for data processing and analysis. These are implemented in a modular framework for analysing spectroscopic data, termed ASpecD and focussing on usability while minimising system requirements and detailed knowledge in software development that is notoriously scarce in science. The implementation is driven by own needs and based on long-standing experience with both, experimental science and the intrinsic limitations with regard to software engineering in academia. Therefore, we are optimistic that the ideas presented here will encourage other scientists to focus more on reliable and reproducible software-based data analysis, thus ultimately contributing to the overall quality of research. Providing concepts as well as their implementation that take care of most of the equally important and tedious tasks, the additional effort besides personal discipline is rather negligible. Finally, recipe-driven data analysis not requiring any programming skills renders truly reproducible research easily achievable for spectroscopists.

# Conflicts of interest

There are no conflicts to declare.

# Acknowledgements

The ideas presented have evolved over more than a decade, and many people have helped shape the ideas and implemented parts of previous programs eventually leading to the framework described here. To name the most important persons, in chronological order of their contributions: T. Berthold, B. Paulus, D. Meyer, M. Schröder. Thanks as well to all users of previous programs of the author, all colleagues discussing the ideas, and all the students attending TB's course on scientific software development over the years. TB furthermore thanks S. Weber for providing a fruitful environment that made it possible to develop the ideas presented here. The German Research Foundation (DFG, Grant BI-1249/3-1) is gratefully acknowledged for financial support.

## Supporting Information

Examples of recipe-driven data analysis, a dataset-centric approach to the ASpecD framework, implementation details, writing software based on the ASpecD framework.

## Software availability

The ASpecD framework is provided open-source and free of charge under a BSD license. Details can be found on its website (https://www.aspecd.de/) together with a detailed documentation for both, users and developers (https://docs.aspecd.de/). The ASpecD framework is available via the Python Package Index (PyPI) (https://pypi.org/project/aspecd/) and the source code via GitHub (https://github.com/tillbiskup/aspecd). The latest version is always available via the DOI 10.5281/zenodo.4717937.

## Notes and references


1 I. Newton, letter to Robert Hooke, 5th February 1676.

2 R. K. Merton, *On the Shoulers of Giants. A Shandean Postscript. The Post-Italianate Edition*, The University of Chicago Press, Chicago, 1993.

3 *Implementing Reproducible Research*, ed. V. Stodden, F. Leisch and R. D. Peng, CRC press, Boca Raton, Florida, 2014.

4 S. R. Piccolo and M. B. Frampton, *GigaScience*, 2016, **5**, 30.

5 W. P. Walters, *J. Chem. Inf. Model.*, 2013, **53**, 1529–1530.

6 G. Wilson, *Comput. Sci. Eng.*, 2006, **8**, 66–69.

7 M. R. Munafò, B. A. Nosek, D. V. M. Bishop, K. S. Button, C. D. Chambers, N. P. du Sert, U. Simonsohn, E.-J. Wagenmakers, J. J. Ware and J. P. A. Ioannidis, *Nat. Hum. Behav.*, 2017, **1**, 0021.

8 K. Popper, *Logik der Forschung*, Mohr Siebeck, Tübingen, 11th edn., 2005.

9 A. F. Chalmers, *What is this thing called Science?*, Open University Press, Berkshire, UK, 3rd edn., 1999.

10 I. Kant, *Metaphysische Anfangsgründe der Naturwissenschaft*, Felix Meiner Verlag, Hamburg, 1997.

11 H. Andersen and B. Hepburn, in *The Stanford Encyclopedia of Philosophy*, ed. E. N. Zalta, Metaphysics Research Lab, Stanford University, Summer 2016 edn., 2016.

12 E. R. Tufte, *The Visual Display of Quantiative Information*, Graphics Press, Cheshire, Connecticut, 2nd edn., 2001.

13 J. P. Mesirov, *Science*, 2010, **327**, 415–416.

14 D. Monroe, *Commun. ACM*, 2015, **58**, 12–14.

15 P. Vandewalle, J. Kovačević and M. Vetterli, *IEEE Signal Process. Mag.*, 2009, **26**, 37–47.

16 M. G. Weller, *Methods Protoc.*, 2021, **4**, 12.

17 L. Hatton and M. van Genuchten, *IEEE Softw.*, 2019, **36**, 137–144.

18 Z. Merali, *Nature*, 2010, **467**, 775–777.

19 L. Hatton and A. Roberts, 1994, **20**, 785–797.

20 G. Wilson, *IEEE Comput. Sci. Eng.*, 1996, **3**, 46–55.

21 G. V. Wilson, *Am. Scientist*, 2006, **94**, 5–6.

22 C. Goble, *IEEE Internet Comput.*, 2014, **18**, 4–8.

23 S. M. Baxter, S. W. Day, J. S. Fetrow and S. J. Reisinger, *PLoS Comput. Biol.*, 2006, **2**, e87.

24 G. K. Sandve, A. Nekrutenko, J. Taylor and E. Hovig, *PLoS Comput. Biol.*, 2013, **9**, e1003285.

25 J. M. Perkel, *Nature*, 2018, **560**, 513–515.

26 Open Science Collaboration, *Science*, 2015, **349**, aac4761.

27 J. O. A. Ioannidis, *PLoS Med.*, 2005, **2**, e124.

28 M. Baker, *Nature*, 2016, **533**, 452–454.

29 B. A. Nosek, G. Alter, G. C. Banks, D. Borsboom, S. D. Bowman, S. J. Breckler, S. Buck, C. D. Chambers, G. Chin, G. Christensen, M. Contestabile, A. Dafoe, E. Eich, J. Freese, R. Glennerster, D. Goroff, D. P. Green, B. Hesse, M. Humphreys, J. Ishiyama, D. Karlan, A. Kraut, A. Lupia, P. Mabry, T. Madon, N. Malhotra, E. Mayo-Wilson, M. McNutt, E. Miguel, E. L. Paluck, U. Simonsohn, C. Soderberg, B. A. Spellman, J. Turitto, G. VandenBos, S. Vazire, E. J. Wagenmakers, R. Wilson and T. Yarkoni, *Science*, 2015, **348**, 1422–1425.

30 T. Poisot, *Ideas in Ecology and Evolution*, 2015, **8**, 50–54.

31 E. P. White, *Ideas in Ecology and Evolution*, 2015, **8**, 55–57.

32 X. Chen, S. Dallmeier-Tiessen, R. Dasler, S. Feger, P. Fokianos, J. B. Gonzalez, H. Hirvonsalo, D. Kousidis, A. Lavasa, S. Mele, D. R. Rodriguez, T. Šimko, T. Smith, A. Trisovic, A. Trzcinska, I. Tsanaktsidis, M. Zimmermann, K. Cranmer, L. Heinrich, G. Watts, M. Hildreth, L. L. Iglesias, K. Lassila-Perini and S. Neubert, *Nat. Phys.*, 2019, **15**, 113–119.

33 R. D. Peng, *Science*, 2011, **334**, 1226–1227.

34 R. J. LeVeque, I. M. Mitchell and V. Stodden, *Comput. Sci. Eng.*, 2012, **11**, 13–17.

35 Yale Law School Roundtable on Data and Code Sharing, *Comput. Sci. Eng.*, 2010, **12**, 8–13.

36 D. L. Donoho, A. Maleki, M. Shahram, I. U. Rahman and V. Stodden, *Comput. Sci. Eng.*, 2009, **11**, 8–18.

37 B. Lawlor and P. Walsh, *Bioengineered*, 2015, **6**, 193–203.

38 R. D. Peng, F. Dominici and S. L. Zeger, *Am. J. Epidemiol.*, 2006, **163**, 783–789.

39 M. McCormick, X. Liu, J. Jomier, C. Marion and L. Ibanez, *Front. Neuroinformatics*, 2014, **8**, 13.

40 G. Wilson, D. A. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. Guy, S. H. D. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley, B. Waugh, E. P. White and P. Wilson, *PLoS Biol.*, 2014, **12**, e1001745.

41 D. De Roure and C. Goble, *IEEE Softw.*, 2009, **26**, 88–95.

42 A. Prlić and J. B. Procter, *PLoS Comput. Biol.*, 2012, **8**, e1002802.

43 J. M. Osborne, M. O. Bernabeu, M. Bruna, B. Calderhead, J. Cooper, N. Dalchau, S.-J. Dunn, A. G. Fletcher, R. Freeman, D. Groen, B. Knapp, G. J. McInerny, G. R. Mirams, J. Pitt-Francis, B. Sengupta, D. W. Wright, C. A. Yates, D. J. Gavaghan, S. Emmott and C. Deane, *PLoS Comput. Biol.*, 2014, **10**, e1003506.

44 W. Michener, *PLoS Comput. Biol.*, 2015, **11**, e1004525.

45 A. Goodman, A. Pepe, A. W. Blocker, C. L. Borgman, K. Cranmer, M. Crosas, R. Di Stefano, Y. Gil, P. Groth, M. Hedstrom,

D. W. Hogg, V. Kashyap, A. Mahabal, A. Siemiginowska and A. Slavkovic, *PLoS Comput. Biol.*, 2014, **10**, e1003542.
46 E. M. Hart, P. Barmby, D. LeBauer, F. Michonneau, S. Mount, P. Mulrooney, T. Poisot, K. H. Woo, N. B. Zimmerman and J. W. Hollister, *PLoS Comput. Biol.*, 2016, **12**, e1005097.
47 T. Biskup, *Mol. Phys.*, 2013, **111**, 3698–3703.
48 T. Biskup, *Front. Chem.*, 2019, **7**, 10.
49 T. Biskup, *Appl. Phys. Lett.*, 2021, **119**, 010503.
50 J. F. Claerbout and M. Karrenbach, in *Electronic documents give reproducible research a new meaning*, 1992, pp. 601–604.
51 M. Schwab, M. Karrenbach and J. Claerbout, *Comput. Sci. Eng.*, 2000, **2**, 61–67.
52 E. W. Dijkstra, in *EWD447: On the role of scientific thought*, Springer-Verlag, New York, 1982, pp. 60–66.
53 L. A. Barba, *CoRR*, 2018, **abs/1802.03311**, 1802.03311.
54 R. S. Kenett and G. Shmueli, *Nat. Methods*, 2015, **12**, 699.
55 S. N. Goodman, D. Fanelli and J. P. A. Ioannidis, *Sci. Transl. Med.*, 2016, **8**, 341ps12.
56 A. Hunt and D. Thomas, *The Pragmatic Programmer*, Addison-Wesley, Boston, 1999.
57 S. McConnell, *Code Complete. A practical handbook of software construction*, Microsoft Press, Redmond, Washington, 2004.
58 E. S. Raymond, *The Art of UNIX Programming*, Addison Wesley, Boston, 2004.
59 A. Scopatz and K. D. Huff, *Effective Computation in Physics*, O'Reilly, Sebastopol, 2015.
60 S. Chacon and B. Straub, *Pro Git*, Apress, New York, NY, 2nd edn., 2014.
61 D. C. Ince, L. Hatton and J. Graham-Cumming, *Nature*, 2012, **482**, 485–488.
62 R. Schuwer, M. van Genuchten and L. Hatton, *IEEE Softw.*, 2015, **32**, 81–83.
63 G. A. Landrum and N. Stiefl, *Future Med. Chem.*, 2012, **4**, 1885–1887.
64 A. Morin, J. Urban, P. D. Adams, I. Foster, A. Sali, D. Baker and P. Sliz, *Science*, 2012, **336**, 159–160.
65 N. Barnes, *Nature*, 2010, **467**, 753.
66 D. McCafferty, *Commun. ACM*, 2010, **53**, 16–17.
67 A. J. Nederbragt, *Genome Biology*, 2014, **15**, 113.
68 E. Yourdon and L. L. Constantine, *Structured Design. Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1979.
69 E. W. Dijkstra, *Commun. ACM*, 1972, **15**, 859–865.
70 E. Evans, *Domain-Driven Design*, Addison Wesley, Boston, 2004.
71 E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
72 B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style*, McGraw-Hill, New York, 2nd edn., 1978.
73 E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, 1995.
74 R. C. Martin, *Agile Software Development. Principles, Patterns, and Practices*, Prentice Hall, Upper Saddle River, New Jersey, 2003.
75 M. Fowler, *Patterns of Enterprise Application Architecture*, Addison Wesley, Boston, 2003.
76 R. C. Martin, *Clean Architecture. A Craftman's Guide to Software Structure and Design*, Prentice Hall, Boston, 2018.
77 E. W. Dijkstra, *Commun. ACM*, 1968, **11**, 147–148.
78 *Structured Programming*, ed. C. A. R. Hoare, Academic Press, London, 1972.
79 O.-J. Dahl and K. Nygaard, *Commun. ACM*, 1966, **9**, 671–678.
80 J. McCarthy, *Commun. ACM*, 1960, **3**, 184–195.
81 R. C. Martin, *Clean Code. A Handbook of Agile Software Craftmanship*, Prentice Hall, Upper Saddle River, New Jersey, 2008.
82 D. E. Knuth, *Literate Programming*, Center for the Study of Language and Information, Stanford, 1992.
83 B. W. Kernighan and R. Pike, *The Practice of Programming*, Addison Wesley, Boston, 1999.
84 M. Fowler, *Refactoring. Improving the Design of Existing Code*, Addison Wesley Longman, Boston, 1999.
85 S. Freeman and N. Pryce, *Growing Object-Oriented Software, Guided by Tests*, Addison Wesley, Upper Saddle River, New Jersey, 2010.
86 J. A. Whittaker, *IEEE Softw.*, 2000, **17**, 70–79.
87 R. C. Martin, *IEEE Softw.*, 2007, **24**, 32–36.
88 K. Beck, *Test-Driven Development By Example*, Addison-Wesley, Boston, 2003.
89 G. Meszaros, *xUnit Test Patterns*, Addison-Wesley, Boston, 2007.
90 R. Osherove, *The Art of Unit Testing*, Manning, Shelter Island, 2nd edn., 2014.
91 A. M. Turing, *Proc. London Math. Soc.*, 1936, **42**, 230–265.
92 K. Gödel, *Monatsh. Math. Phys.*, 1931, **38**, 173–198.
93 T. Biskup, *ASpecD framework*, 2021, `https://docs.aspecd.de/`, doi:10.5281/zenodo.4717937.
94 N. P. Rougier, M. Droettboom and P. E. Bourne, *PLoS Comput. Biol.*, 2014, **10**, e1003833.
95 C. O. Wilke, *Fundamentals of Data Visualization: A Primer on Making Informative and Compelling Figures*, O'Reilly, Sebastopol, CA, 2019.
96 A. N. Whitehead, *An Introduction to Mathematics*, Dover Publications, Mineola, 2017.
97 D. G. Luenberger and Y. Ye, *Linear and Nonlinear Programming*, Springer, Cham, 4th edn., 2016.
98 N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller, *J. Chem. Phys.*, 1953, **21**, 1087–1092.
99 M. D. McKay, R. J. Beckman and W. J. Conover, *Technometrics*, 1979, **21**, 239–245.
100 Docker, Inc., *Docker*, `https://www.docker.com/`.
101 *YAML: YAML Ain't Markup Language*, `https://yaml.org/`.

102 Red Hat, Inc., *Ansible*, https://www.ansible.com/.

103 J. W. Reeves, in *The Source Code is the Design*, Prentice Hall, Upper Saddle River, New Jersey, 2003, pp. 517–524.

104 S. Oliveira and D. Stewart, *Writing Scientific Software. A Guide to Good Style*, Cambridge University Press, Cambridge, 2006.

105 J. D. Blischak, E. R. Davenport and G. Wilson, *PLoS Comput. Biol.*, 2016, **12**, e1004668.

106 T. Preston-Werner, *Semantic Versioning 2.0.0*, https://semver.org/.

107 J. U. Hasecke, *Software-Dokumentation mit Sphinx*, hasecke.com, Solingen, 2019.

108 B. Meyer, *Object-Oriented Software Construction*, Prentice Hall PTR, Upper Saddle River, New Jersey, 1997.

109 M. Weisfeld, *The Object-Oriented Thought Process*, Addison-Wesley, Upper Saddle River, New Jersey, 4th edn., 2013.

110 G. van Rossum and J. de Boer, *CWI Q.*, 1991, **4**, 283–304.

111 D. Beazley and B. K. Jones, *Python Cookbook*, O'Reilly, Sebastopol, CA, 3rd edn., 2013.

112 K. Reitz and T. Schlusser, *The Hitchhiker's Guide to Python*, O'Reilly, Sebastopol, 2016.

113 W. McKinney, *Python for Data Analysis*, O'Reilly, Sebastopol, 2nd edn., 2017.

114 T. E. Oliphant, *Guide to NumPy*, Continuum Press, Austin, Texas, 2nd edn., 2015.

115 C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke and T. E. Oliphant, *Nature*, 2020, **585**, 357–362.

116 P. Virtanen, R. Gommers, T. E. Oliphant, M. HaberlAND, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VANDerPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt and SciPy 1.0 Contributors, *Nat. Methods*, 2020, **17**, 261–272.

117 J. D. Hunter, *Comput. Sci. Eng.*, 2007, **9**, 90–95.

118 D. Goldberg, *ACM Comput. Surv.*, 1991, **23**, 5–48.

119 J. Popp, M. Schröder and T. Biskup, *trEPR Python package*, 2021, https://docs.trepr.de/, doi:10.5281/zenodo.4897112.

120 M. Schröder and T. Biskup, *cwepr Python package*, 2021, https://docs.cwepr.de/, doi:10.5281/zenodo.4896687.

121 S. Stoll and A. Schweiger, *J. Magn. Reson.*, 2006, **178**, 42–55.

122 S. Rein, *Ph.D. thesis*, Albert-Ludwigs-Universität Freiburg, 2019.

123 T. Biskup, *LabInform: A modular laboratory information system built from open source components*, manuscript in preparation.